ARCHITECTING AND IMPLEMENTING DEVOPS

for Infrastructure Management in Azure

Designed by Freepik

ASHISH RAJ

Configure, Deploy, & Automate Your Azure resources with terraform, GitHub Actions and Ansible

Architecting and Implementing DevOps for Infrastructure Management in Azure

Infrastructure DevOps in Azure: Configure, Deploy, & Automate Your Azure resources with terraform, GitHub Actions and Ansible

Ashish Raj

This book is available at http://leanpub.com/devopsinazure

This version was published on 2025-02-02

Contents

Chapter 1 : Foundation of Cloud & Infrastructure as Code

In this chapter, we will learn about the components of cloud infrastructure and how its features, such as high availability, scalability, fault tolerance, agility, etc., compare against traditional on-premises infrastructure. We will understand how organizations can benefit from the Azure cloud and its architectural components in deploying more resilient and cost-effective solutions. We will learn about Infrastructure-as-Code (IaC) and how to use Terraform to implement Infrastructure-as-Code in Azure.

This chapter introduces crucial concepts for starting your cloud journey:

- Introduction to Cloud
- Azure Cloud Architecture
- Infrastructure-as-Code
- Getting started with Terraform in Azure

Technical requirements

You should have a basic understanding of infrastructure resources such as Virtual Machines, Storage, Networks, etc. You should be familiar with IT infrastructure services and their usage for application deployment. To install the tools and practice following the instructions in this chapter, you will need a PC/Laptop with at least 8 GB or more RAM and a 1.6 GHz or faster processor. Please refer to the system requirements of the respective tools to make sure your system meets the minimum requirements.

Azure CLI: Install Azure CLI[1]

Visual Studio Code: Install Visual Studio Code[2]

PowerShell Core: Install PowerShell Core[3]

[1] https://learn.microsoft.com/en-us/cli/azure/install-azure-cli
[2] https://code.visualstudio.com/Download
[3] https://learn.microsoft.com/en-us/powershell/scripting/install/installing-powershell?view=powershell-7.3

Introduction to Cloud

In today's world, it is hard to imagine any organization, irrespective of its size, not using one or many cloud platforms in one way or another. Cloud has enabled organizations to break through many boundaries that were either impossible using traditional infrastructure or were too costly to implement.

The concept of the cloud can vary, contingent upon the lens through which it's viewed. For everyday users, it's an easily accessible service, available over the Internet whenever needed, without delving into the underlying technological intricacies. For businesses, cloud represents a rapid, cost-effective IT infrastructure alternative, alleviating concerns about hefty capital investments and operational overheads associated with on-premises setups. People such as Infrastructure Engineers, software developers, or architects, however, look at the cloud as a means to develop and deploy highly scalable and resilient solutions using automated and predictable methods.

Despite this different perspective, the benefits of the cloud remain consistent across personas, unified by its inherent advantages:

1. High Availability
2. Fault tolerance
3. Scalability
4. Elasticity
5. Global Reach
6. Agility
7. Predictive cost
8. Security

High Availability

In today's landscape, organizations must not only plan their current needs but also ensure the availability of services in case of data center failures in one or multiple geographical regions. To have globally available infrastructure, organizations must maintain servers across diverse geographical regions, which involves significant capital investments in skilled resources to architect and manage such systems. It is also almost impossible to predict future needs, and that often necessitates upfront hardware procurement, even if immediate utilization isn't required.

Cloud platforms are offered by providers such as Microsoft, Amazon, or Google, all of which have infrastructure presence across the globe. These cloud providers have a lot of experience in maintaining such systems on scale while continuously investing in cloud infrastructure. This gives them the capability to offer the cloud platform with inherent support for architecting a solution that can be highly available, globally replicated, or configured to load balance across multiple servers. With the cloud, organizations can get highly available services with a click of a button with no

capital investments. Cloud providers offer their services mostly in pay-as-you-go models without organizations having to go through major capital investments. Organizations can start using cloud services and scale their cloud infrastructure as they may need.

Fault Tolerance

No matter how well an organization plans its infrastructure in a data center, it is always prone to certain unavoidable events, such as server reboots due to a firmware or driver upgrade on the physical servers. To be able to manage such events, one must invest in architecting the data center in such a way that workloads can be shifted from one hardware to another, without the user noticing any difference when consuming the service. This means organizations must plan for additional hardware in the data centers to be able to execute such activity. Cloud providers, housing vast arrays of servers in their data centers, offer configuration options to consumers for hosting fault-tolerant solutions. For instance, Azure provides capabilities like Availability Sets or Availability Zones, empowering users to deploy fault-tolerant Virtual Machines (VMs). We will learn more about such features and their usage in subsequent chapters when we architect and deploy VM infrastructure.

Scalability

Any organization, when starting to deploy their services, tends to begin small and initially may not have a significant need for high availability configuration. However, they must also be ready to scale the infrastructure as the solution's usage grows. Mostly, these scaling needs are not very predictable for organizations. With traditional infrastructure, organizations must invest in high availability configuration right from the start, even if that is not needed for the next few months, and sometimes, these on-premises systems are underutilized for years.

Cloud providers allow both horizontal scaling (adding more instances of service) and vertical scaling (increasing capacity of service) on demands. Moreover, these providers enable automation based on usage metrics, such as CPU load or RAM usage, allowing the cloud service to dynamically scale per demand. For example, Azure Virtual Machine Scale Sets (VMSS) allow auto-scaling the number of VM instances based on CPU usage percentage. With such features from the cloud, organizations do not need to invest in large hardware configurations when they do not need them and can automatically scale the services in the cloud based on usage.

Elasticity

In managing services on IT infrastructure, organizations not only require scalability as demand surges, but also the flexibility to scale down or even deactivate certain services based on usage patterns. Consider a database service that demands high CPU and memory usage solely during specific periods, like batch processing for database backups or ETL (Extract, Transform, Load) operations. Configuring such elasticity of infrastructure resources is often challenging or non-existent in traditional infrastructure setups, particularly when attempting automation.

Cloud infrastructure services are made with elasticity as one of the critical components. Teams consuming services from cloud providers can easily benefit from service elasticity in their solution without worrying about additional capital investments.

Global Reach

In today's globalized landscape, organizations often operate applications used by a diverse user base across various geographical locations. Compliance and regulatory obligations frequently mandate enterprises to host applications in specific regions. Additionally, organizations seek to optimize user experience by hosting applications closer to users, reducing network latency. Traditionally, expanding global reach demands extensive planning and substantial capital investment.

Public cloud providers are equipped with a multitude of data centers strategically positioned across numerous geographical regions. These providers offer configuration options enabling seamless deployment of applications across different countries, eliminating the need for meticulous planning and hefty investments associated with traditional on-premises infrastructure. For example, Microsoft Azure currently has over 60 regions and 300+ data centers across the globe and is still expanding. For example, an organization can deploy one instance of an application in one of the data centers in the European region with all their application data complying with European GDPR compliance, while another instance is running in the United States that complies with the United States regulatory requirements.

Agility

In today's world, business needs are changing faster than ever. So is the technological advancement to achieve those business needs. To match the fast-changing business requirements, you need an agile infrastructure that can meet the application's changing needs as soon as possible. With traditional on-premises infrastructure, this has always been a bottleneck for application development teams, preventing them from leveraging the latest available technology to meet the fast-changing business requirements. With on-premises systems, it takes time to procure the new tooling and hardware and plan their implementation. It also takes further time to make it available to developers for use in their applications.

Cloud providers continuously add new services and features that can be used by organizations consuming their cloud. This allows organizations to use the latest technologies on demand without much detailed planning or upfront cost. This allows teams to quickly adopt the latest technologies best suited for implementing new features and further enhances their go-to-market strategy for business. For example, organizations have struggled for years to leverage the benefits of AI and machine learning, as it takes time and requires significant investment—both in the skills needed to develop machine learning models and in the infrastructure required for training them. The recent advancements by OpenAI and the OpenAI services offered on Microsoft Azure now allow organizations to quickly use OpenAI models, train them with their own data, and consume them in their own applications quickly to gain the advantages of AI and machine learning.

Predictive Cost

When using traditional on-premises infrastructure, not only do you have capital investments, but it is also hard to make cost prediction of how much your services are really going to consume out of it. It also limits your resources when your needs grow more than what you have invested in, and it takes time to further plan the upgrade of on-premises infrastructure.

Contrarily, cloud services provide a closer approximation of service consumption costs. Even leading public cloud providers, like Azure, offer tools such as the Total Cost of Ownership (TCO) calculator[4]. TCO facilitates the calculation and comparison of hosting expenses between Azure and on-premises servers. It encompasses diverse deployment options, encompassing hardware, software, support costs, and ancillary expenses like cooling and power for on-premises setups. TCO shows you the long-term benefits of hosting applications on Azure. Azure also offers a simple Azure pricing calculator[5] that helps you select services in Azure, select their configuration options, and quickly calculate the cost based on your application needs.

Security

Securing your on-premises infrastructure against targeted attacks such as DoS (Denial of Service) or vulnerabilities. Such tasks are always complex for any organization. Moreover, with advancements in technology, different geographical regions have different security and compliance requirements that you must make sure your solutions are complying with.

Cloud providers invest a lot in making sure their systems are not only updated with the right set of tooling and architecture to defend against targeted attacks such as DoS, but also provide services such as firewall-as-a-service, vulnerability, and network scanning, etc., so that you can deploy applications per your security requirements. Since the cloud provider continually expands their data centers across different regions, they ensure their platform meets all the regulatory and compliance requirements for respective regions. Microsoft Azure even offers the concept of Landing Zones that enable you to create such security boundaries and deploy your application in the most secure way.

Azure Cloud Foundation

Microsoft Entra ID

Using Azure starts with a Microsoft Entra ID tenant (formerly known as Azure Active Directory tenant). When creating an Azure Account, you can either create a new Entra ID tenant or select an existing Microsoft Entra ID tenant. The following figure shows a Microsoft Entra ID tenant attached to an organization.

[4] https://azure.microsoft.com/pricing/tco/calculator
[5] https://azure.microsoft.com/pricing/calculator/

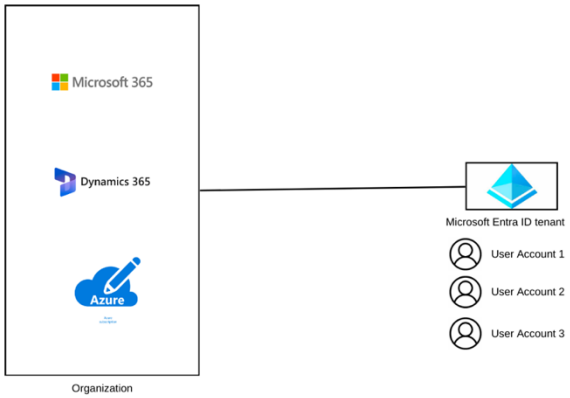

Figure 1. Microsoft Entra ID tenant

Microsoft Entra ID is an Identity and Access Management service that allows you to authenticate and authorize access to resources in Azure, Microsoft 365, or Dynamics 365. Small organizations may use the same tenant to configure all these services, but large organizations may have separate tenants for Office 365, Dynamics 365, and a separate Microsoft Entra ID tenant for Dynamics 365.

Azure Subscriptions

When creating an Azure Account, you choose a subscription, which can be based on options such as pay-as-you-go, an enterprise agreement, or through a Cloud Solution Partner. If you already have an Azure Account, you can also purchase an additional subscription with your existing account. With the initial account creation process, you get a subscription attached to either a new Microsoft Entra ID tenant or a selected existing tenant. Subscription in Azure is the billing unit. So, before deploying a service such as a Virtual Machine or Database, you must first purchase a subscription, which you can then select during the deployment of services like Virtual Machines. The cost of the deployed services will be attached and billed to you through subscriptions.

An organization can have multiple subscriptions within a Microsoft Entra ID tenant, or a large organization may have multiple tenants, each dedicated to different types of subscriptions.

Management Groups

Once you have Azure subscription, it further comes under something called Management Group. Management groups are the basic pillar of establishing governance in Azure. An organization, by default, gets at least one management group called Root Management Group. Organizations can further create a hierarchy of management groups, as shown in the following figure.

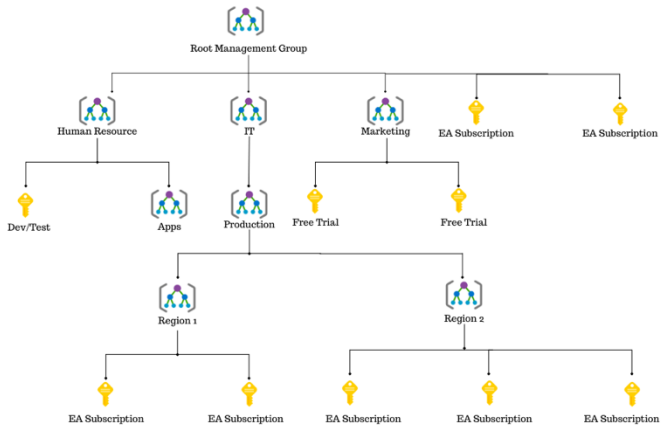

Figure 2. Management groups

Subscriptions can then be attached to respective management groups based on business units or types of environments for governance purposes. For example, an organization can have a separate management group called HR under the root management group, which will contain all subscriptions used by the HR team's Azure resources. Similarly, there can be an IT management group that can have two further management groups for production resources in one region and production resources in another.

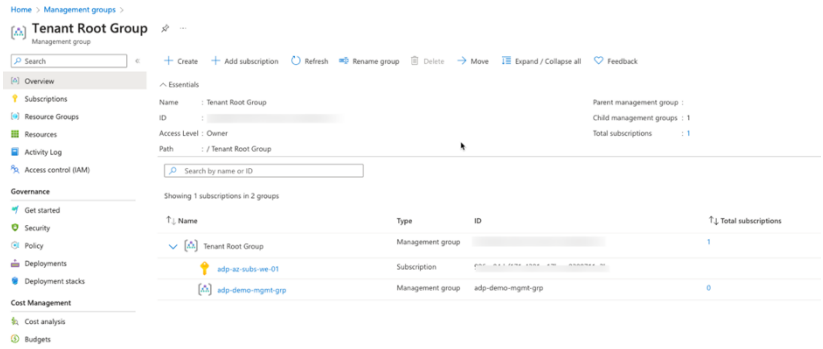

Figure 3. Management group in Azure portal

As shown in the above figure, management groups allow you to have a comprehensive look at resources, manage access to them, or enforce governance using Azure policies. We will discuss Azure policies in detail in the upcoming chapter and will go through enforcing governance in Azure using it.

Resource Groups

In addition to management groups and subscriptions, Azure allows you to manage resources in a logical container called Resource Groups. The following figure further shows the hierarchy of resources, starting from a management group to a resource.

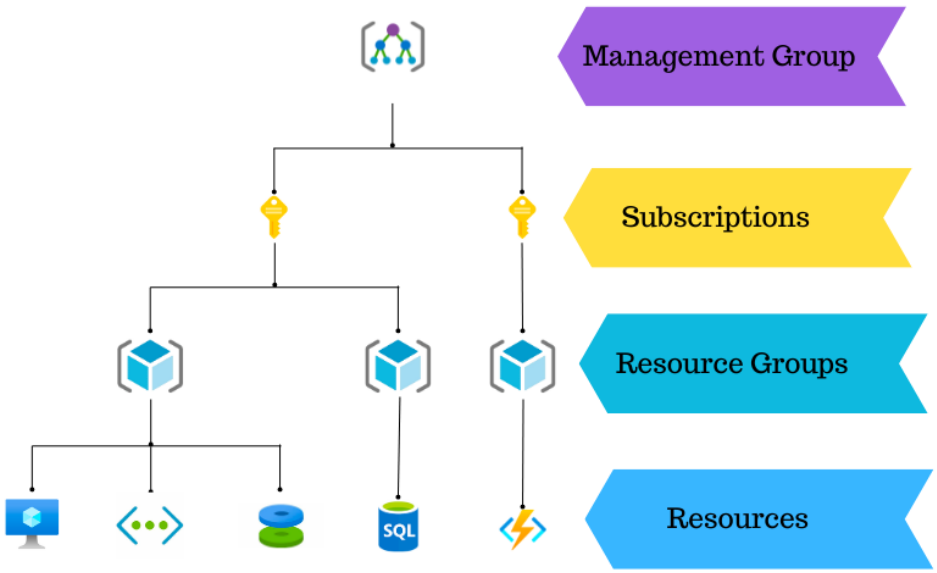

Figure 4. Azure resource hierarchy

Usually, it is common to use the same resource group for deploying related services. For example, if you are deploying a web application infrastructure that needs multiple resources in Azure, such as virtual machines, virtual networks, and a database, then it is recommended to have all these resources in the same resource group.

There is no limitation to the functionality and ability to connect with each other if resources are deployed in two different resource groups. For example, you can have a VM in one resource group and a database in another resource group and still be able to use their functionality in the same way as if they were in the same resource group. So, it entirely depends on your organizational pattern how you want to group resources in resource groups.

When you start working with Azure, you may have a very simple setup with one management group, one subscription, and a few resource groups containing few resources. However, as organizations grow their cloud adoption, they need to make sure these components are configured in a scalable architecture. Azure cloud adoption framework suggests an Azure landing zone that helps you plan for such a scalable setup. We will discuss more about landing zone design areas and how to plan landing zone implementation for an enterprise organization in Chapter 2.

Azure Resource Manager

Any interaction with Azure to deploy resources happens through Azure Resource Manager (ARM). To work with ARM, you usually do not directly interact with ARM; rather, Microsoft offers different interfaces through which you can easily communicate with ARM and manage resources in Azure. The following figure shows different methods to interact with ARM.

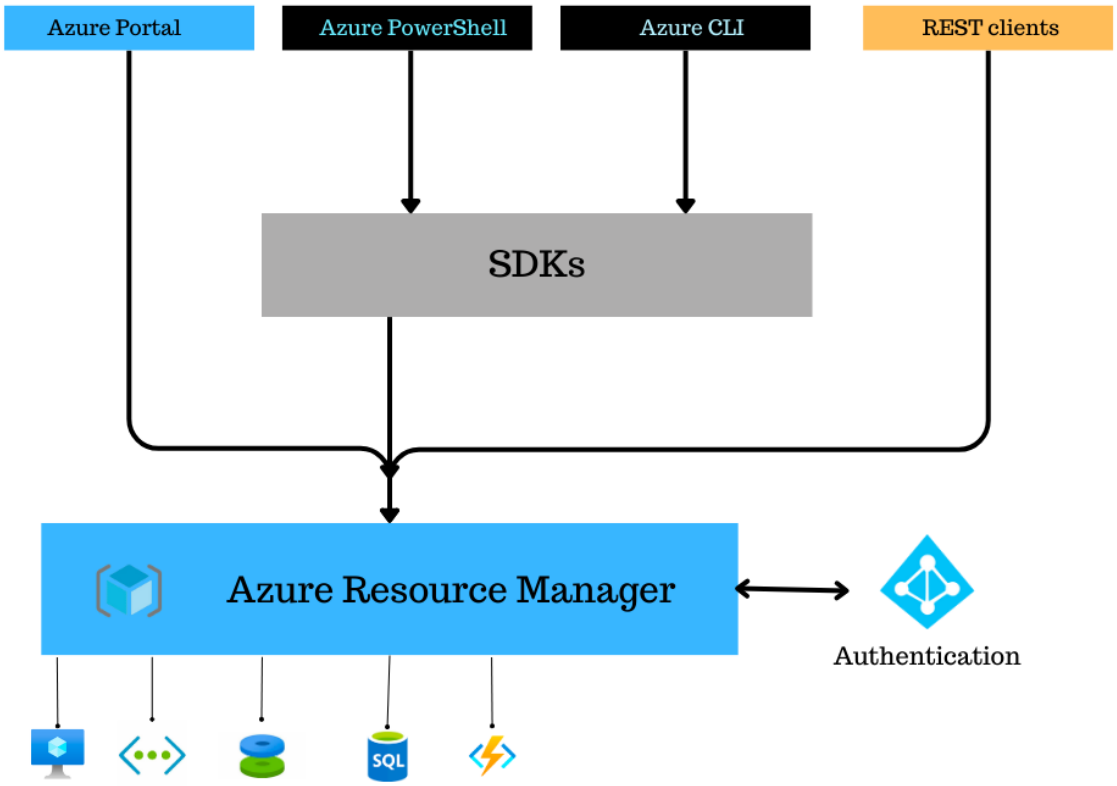

Figure 5. Azure Resource Manager

As shown in the above figure, you can use Azure portal, Azure PowerShell, Azure CLI, or any other rest client to communicate to Azure Resource Manager. ARM has different providers that work with

respective resources in Azure.

Azure portal is the easiest option, a web-based application that can be accessed through any modern web browser. The Azure portal provides a user-friendly, UI-based approach to create and manage resources in Azure. However, for more scalable and automated infrastructure deployment, the Azure portal is not an efficient approach. For more scalable and automated infrastructure, you should use Infrastructure as Code. In the upcoming section, we will discuss Infrastructure as Code and how to use this to do automated and scalable deployment in Azure.

DevOps & Infrastructure as Code

DevOps

With the growing change in business requirements and fast-changing technology, development teams must adopt agile methodologies in the application development life cycle.

A team following agile can frequently make changes and deliver business value, compared to traditional approaches like waterfall. Agile teams need to plan their deliverables in a short sprint, usually a 2-3 weeks cycle. The sprint cycle completely depends on team size, organization culture, and many other factors, but what is common among agile development teams is the delivery of business value in each sprint. This also helps development teams accommodate fast-changing business requirements between sprints and gather feedback on delivered solutions quickly.

Delivery of business value means the changes made by development teams must have a measurable impact on the business. For example, if the development team spends the entire sprint developing a feature that could not be deployed due to any reason, then in terms of agile, no business value was delivered during the sprint, as efforts spent are still not available for users.

Agile has not only impacted the way application developers work but also has a huge impact on infrastructure teams, as agile code changes need to be applied with the same level of speed by the infrastructure teams as well.

Once the agile development team has developed the required application changes that will deliver the business value, they need the required IT infrastructure as soon as possible. If Infrastructure teams are not equipped to deliver the infrastructure as fast as development teams deliver the code changes, then, as discussed previously, no business will be delivered. This means no matter how efficiently the development team has worked to deliver the application code changes, the goal of agile is not met. To be able to match the expectations of the agile development team, the infrastructure team needs to align with agile processes. This led to the emergence of a culture called DevOps.

Donavan Brown at Microsoft defines DevOps as

> DevOps is the union of people, process, and products to enable continuous A>delivery of value to the end user.

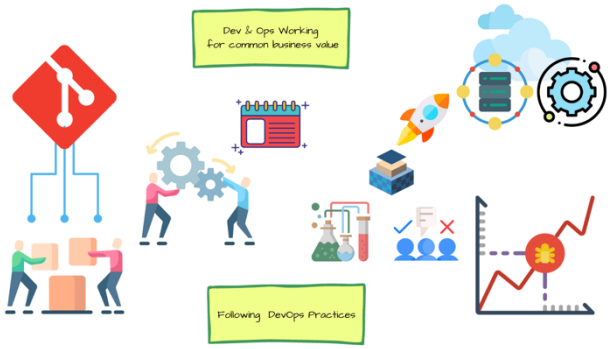

Figure 6. DevOps team

Agile teams come with a mindset of "Change fast, fail fast, and redeploy." however, traditional infrastructure teams usually prefer the mindset of "Don't change, if it's running". With this mindset, Infrastructure teams build all their approval processes and deployment processes that require manual review of any change to a running system. This restrictive mindset was somehow a valid argument in traditional infrastructure where infrastructure deployment does not offer the same level of automation.

But with the growing adoption of cloud infrastructure, it is possible now that infrastructure teams can utilize automation and other cloud infrastructure features that allow them to meet the expected mindset of "change fast, fail fast, and redeploy." This allows both developer and infrastructure teams to work with the same mindset and collaborate towards a common goal to deliver business value faster. DevOps teams commonly follow these seven practices to deliver changes faster.

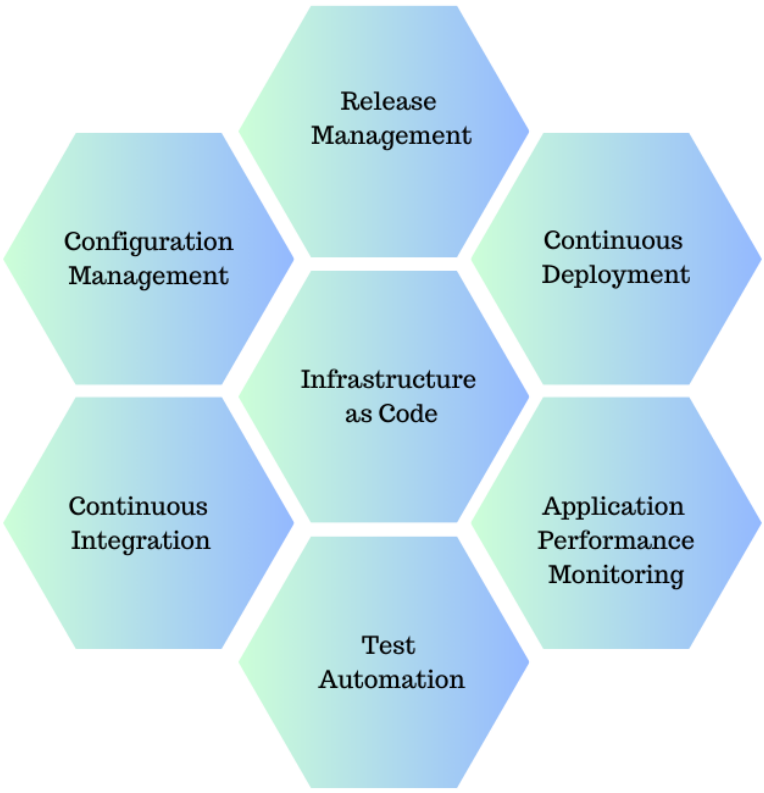

Figure 7. DevOps Practices

These DevOps practices are applied to both developer and infrastructure teams, though they may have their own implementation approaches. Some of these practices are more focused on the infrastructure team and will have minimal visibility in a development team and vice versa. So, DevOps does not necessarily mean a dev must do all Ops or an Ops must do all dev. They both will still have their own areas of responsibility, but now they are more aligned as a team to deliver the business value. For example, with DevOps, the infrastructure team has clarity on what must be delivered in the sprint so that changes made by development can be deployed, and together, business value is delivered.

Configuration management

Configuration management involves maintaining systematic changes to the server environment and running applications in a consistent manner. With traditional configuration management tools and processes, it's often very difficult to keep track of each change. Over time, system configurations become unknown due to several reasons, such as unmanaged manual changes or patch updates. The servers with unknown configuration drifts are commonly known as snowflake servers. Often,

teams avoid making any changes on such snowflake servers as it's hard to predict how the changes will have an impact due to those unknown configurations. For such systems, the infrastructure team becomes very restrictive. This forces them into the mindset of "Don't change, if it's running". They try to enforce strict manual processes around configuration changes to make sure servers do not get frequent changes.

To avoid snowflake servers and help infrastructure teams deploy configuration changes faster, DevOps introduces Configuration as Code. Configuration as Code focuses on maintaining configurations using codes and maintaining them with a version control system. With the help of Configuration as code and automated mechanism, it becomes easier to track changes and make frequent changes. In upcoming chapters, we will discuss implementing Configuration as Code using tools like Ansible.

Infrastructure as Code

Infrastructure as Code, Just like Configuration as Code, refers to maintaining infrastructure resource deployment definitions using code and maintaining them in a version control system like git. Earlier, infrastructure teams had fewer options to automate infrastructure deployment, and because of this, no matter how agile application development was, it was always a time-consuming process to build the infrastructure needed to deploy the application. With infrastructure as code and configuration as code, it is possible to define infrastructure components using tools like ARM templates, Bicep templates, PowerShell scripts, Terraform, Ansible, etc.

Infrastructure as code is maintained in a version control system that helps track changes and has an automated review system for change implementations.

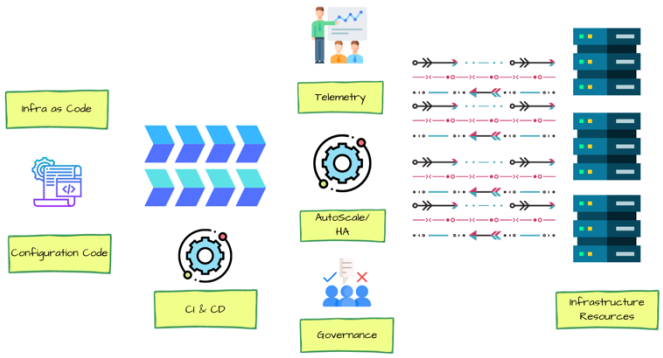

Figure 8. Infrastructure as Code

This allows the infrastructure team to plan infrastructure changes in a similar agile way and deploy frequent changes to infrastructure with more automated and resilient infrastructure.

Often, Infrastructure as Code and Configuration as Code overlap with each other, as it is frequently essential to maintain configurations alongside infrastructure definitions. For example, while deploying a VM infrastructure, you might want to specify a set of third-party tooling to be installed

during the VM deployment process. It becomes easier if you maintain this configuration alongside the VM deployment using Infrastructure as code. But often, it is essential to have a separate Configuration as Code tool to define configurations and maintain them centrally and isolate such configurations from infrastructure as code definitions. For example, an organization might have a standard method for installing the Apache web server on any VM. In such cases, it is better to define and deploy these configurations through a central system like Ansible. This approach prevents each VM from having its own way of configuring the Apache web server, which could occur if configurations were managed individually within each VM's Infrastructure as Code. Having separate configuration as code managed centrally allows code reusability; same configuration can be consumed by different VM infrastructure as code instead of having their own copy of the configuration.

An organization must establish a combination of Infrastructure as Code and Configuration as Code that is perfectly aligned with its people and processes. Often, teams try to replace one with the other, such as keeping all configurations within Infrastructure as Code and replacing the practice of Configuration as Code. In an enterprise environment, however, both must exist and should be established based on organizational processes and needs. We will discuss more about Configuration as Code and its need with Infrastructure as code practice in upcoming chapters.

For infrastructure as code in Azure, many tools exist, such as arm templates, Bicep, Terraform, and Pulumi. The arm templates, Biceps, or Terraform allow you to define infrastructure resources using declarative syntax. However, Pulumi allows you to use general-purpose programming languages such as Python, TypeScript, JavaScript, Go, C#, F#, Java, and YAML.

While declarative syntax keeps infrastructure as code definition simple, each of these tools requires infrastructure developers to learn the respective syntax. For example, terraform uses a declarative syntax called HashiCorp Configuration Language (HCL). Similarly, bicep has its own syntax that has some format similarity to HCL but is a different language and uses its own language construct. Pulumi offers the ability to work with general-purpose languages, so it does not require someone to learn a new language, and existing language knowledge can be used to define Infrastructure as Code. But, Pulumi requires expert-level programming knowledge in the infrastructure team, which is usually common in application development teams. This can often be challenging for infrastructure teams, as it introduces a steep learning curve for an infrastructure engineer. The simplicity of declarative syntax has a shorter learning curve than a general-purpose programming language, which makes Terraform or Azure Bicep more popular choices among infrastructure teams.

Choosing between general-purpose language-based tools like Pulumi or declarative syntax-based tools like Terraform entirely depends on the current and future skill set that your team will be ready to acquire. If your team's primary responsibility will always be to design infrastructure as code properly with the right architecture and would like to automate infrastructure deployment with the least amount of custom development, then using Terraform or Bicep will be the better tool of choice. These tools allow engineers to deploy infrastructure with automation using features developed by the tools, and infrastructure engineers will only need to declare their needs with a shorter learning curve. But if your team wants to be able to not only deploy infrastructure solutions but also want to customize deployment processes or write custom logic to handle these processes that might not

be available with Terraform or Bicep, then using a tool like Pulumi will make more sense. Using Pulumi, teams can write much more complex infrastructure as code logic that may not be available in tools like Terraform. However, this also means that the infrastructure team will need to have expert programmers with domain knowledge of Infrastructure. Essentially, this means you are going to develop your own version of tools like Terraform or Bicep.

For most Infrastructure teams, using one of the declarative approaches makes sense as it requires a shorter learning curve for the team members, and they can focus on their main objective of designing and deploying resources on cloud. The two most popular tools in Azure for declarative Infrastructure as Code are Bicep and Terraform.

When Microsoft introduced declarative Infrastructure as Code, they started with the Azure Resource Manager (ARM) templates. ARM templates are Infrastructure as Code in json file format. Later and most recently, Microsoft introduced Bicep, which is another infrastructure as Code language. Bicep is an evolved version of Azure Resource Manager templates; internally, Bicep also translates to ARM templates. For infrastructure engineers, Bicep offers better code readability and other features, such as better code reusability, improved coding experience, etc, that were major concerns for ARM templates. Bicep is made for Azure cloud and can only define Infrastructure as Code for resources deployed in Azure private and public cloud. While there are initiatives where Bicep may help in multi-cloud Infrastructure as Code scenarios in the future, it currently only offers Infrastructure as Code with Azure for production-level deployment.

Another popular infrastructure as Code tool in Azure is Terraform, a multi-platform Infrastructure as Code tool from HashiCorp. It has huge support for multiple platforms, including all major public cloud providers and many other private and on-premises infrastructure platforms. Terraform supports different platforms through its providers' ecosystem. Providers are sort of intermediate agents between Terraform and the platform that help terraform CLI to interact with the respective platform. While working with terraform, you select a provider based on the target platform where you want to deploy your resources using terraform. There are many official providers which are developed and maintained by HashiCorp. Apart from official providers, terraform also has a vast number of providers that are developed, published, and maintained by HashiCorp partners. Other than the official and partner providers, there are providers published by individual maintainers and these providers are known as community providers.

While Bicep only supports Azure cloud, terraform support for multi-platform makes it one of the popular choices in declarative Infrastructure as code tools. Today, organizations will not only have cloud resources in one cloud, but many big organizations have multi-cloud environments.

Continuous Integration (CI)

Once Infrastructure as Code is defined and hosted on a git version control system, it is necessary to have a feedback system that gives instant feedback to the developer when they make any changes to Code and commit to the git repository. Continuous Integration (CI) ensures deployable artifacts are generated from code changes in development.

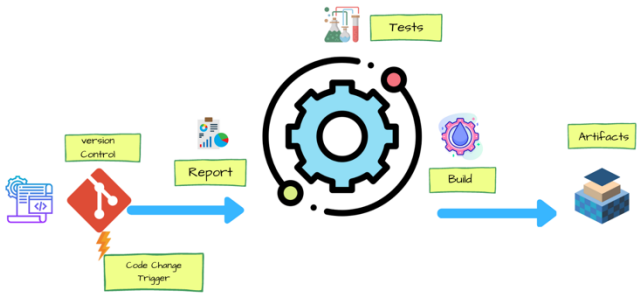

Figure 9. Continuous Integration

Similarly, Infrastructure as Code (IaC) teams utilize CI systems like Azure Pipelines and GitHub Actions to assess changes committed to version control.

These CI systems operate via pipelines, executing automated tests and linting tools on infrastructure code. They offer early feedback on changes, aiding in bug detection before or during deployment. By validating infrastructure templates and configurations, IaC teams achieve agility akin to application development, fostering frequent and reliable deliveries. In subsequent chapters, we'll take a closer look at how to implement CI pipelines for Infrastructure as Code teams.

Continuous Deployment (CD) is the next step after CI in the application development cycle. For Infrastructure as code pipelines, CD takes the artifacts from CI and triggers the deployment to the cloud. In CD, you should define different stages, such as deployment to the QA environment, pre-production, and then prod environment with appropriate approval gates.

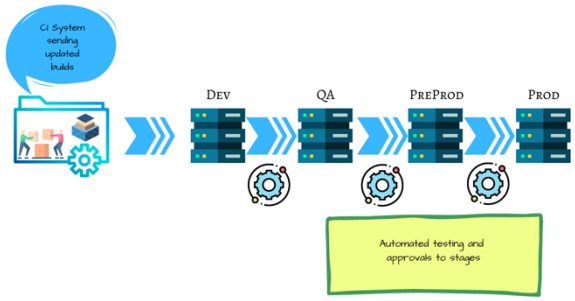

Figure 10. Continuous Deployment

Release Management is the process typically responsible for planning and deploying updated artifacts across different stages. At the same time, this process has a comprehensive list of tasks, such as automated tests, integration tests, load tests, and manual QA validation, as part of the application development lifecycle. Initially, Infrastructure as Code should at least have validation, such as integration tests and applicable automated tests for infrastructure readiness to prevent unsuccessful deployment outcomes. Progressively, the tests and validation in Release management improve as

you add more Infrastructure as Code and as you understand the mechanics of your Infrastructure as code deployment differences in different stages.

Test Automation ensures having a good testing strategy to avoid deploying breaking changes to production systems. Application development has been following this practice for quite a long time and has evolved many different types of tests and tools to implement them. With Infrastructure as Code, this practice is comparatively new and still evolving with a new set of tools. Terraform supports writing tests against Infrastructure as Code.

Application Performance Monitoring allows you to monitor the application and report it to the infrastructure team before it is reported by end users. With Infrastructure as Code, this practice usually refers to instrumenting infrastructure monitoring with the code itself. While deploying Azure resources with Infrastructure as Code, monitoring can be configured with appropriate log collection and alerting based on applicable infrastructure metrics such as high CPU usage or low memory.

Infrastructure as Code Tooling Stack

While following these practices, you will come across a stack of tooling and services that help in achieving the results while following the DevOps practices discussed in the previous section. The following figure shows the tooling stack commonly known when implementing Infrastructure as code practices in Azure.

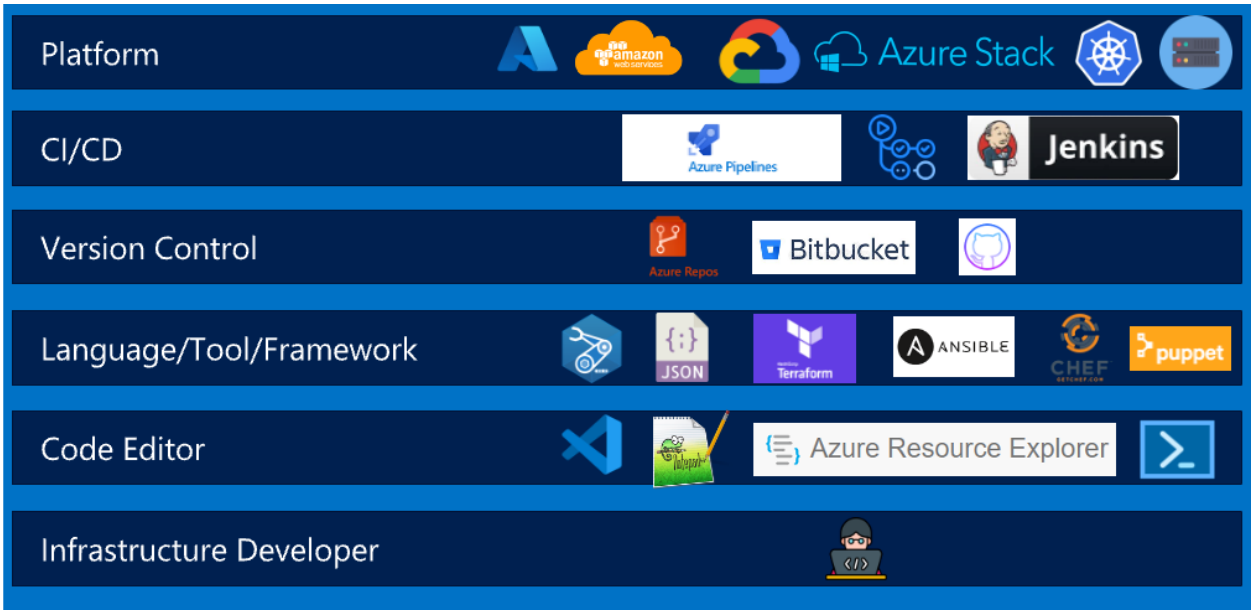

Figure 11. IaC Tool Stack

As shown in the figure above, the first thing an infrastructure engineer has to work with is a code editor. For Infrastructure as Code, you can use any simple text editor like Notepad ++ or a more

developer-friendly tool like Visual Studio Code. Then, the engineer has to choose a language/tool, such as Bicep, Terraform, Ansible, etc., for writing Infrastructure as Code. The Infrastructure as Code needs a version control system such as GitHub or Azure Repos. Continuous Integration can be configured using one of the popular CI & CD pipelines in tools like GitHub Actions, Azure Pipelines, or Jenkins. CI & CD pipelines can deploy resources to Azure public cloud, Azure stack hub private cloud, other cloud providers, or even on-premises infrastructure platforms.

In further chapters, we will move across this stack and use one tool on each level to demonstrate the implementation of the DevOps practices we discussed so far.

Getting started with Terraform in Azure

Tools and Setup

Terraform CLI

Terraform CLI allows you to work with terraform files that are written in HashiCorp Configuration Language (HCL). To install Terraform CLI, you can go to here[6]. Terraform CLI is available for many popular operating systems, including Windows, Linux, and macOS. Follow the instructions for the appropriate operating system and install Terraform CLI.

Once you have installed Terraform CLI, you can run terraform version in the terminal windows, which will show the version of terraform CLI.

```
$ terraform version

Terraform v1.6.4

on darwin_arm64
```

Visual Studio Code

To write Infrastructure as Code, you need a good code editor. Visual Studio (VS) Code, an open-source editor, offers robust features for Terraform-based Infrastructure as Code development. VS Code is available for Windows, Linux, and Mac OS operating systems. Simply [visit] (https://code.visualstudio.com/), download, and install it according to your system requirements.

Once installed, launch VS Code, navigate to the Extension Marketplace, search for ‘terraform,’ and install the HashiCorp Terraform extension, as illustrated in the following image.

[6] https://developer.hashicorp.com/terraform/install

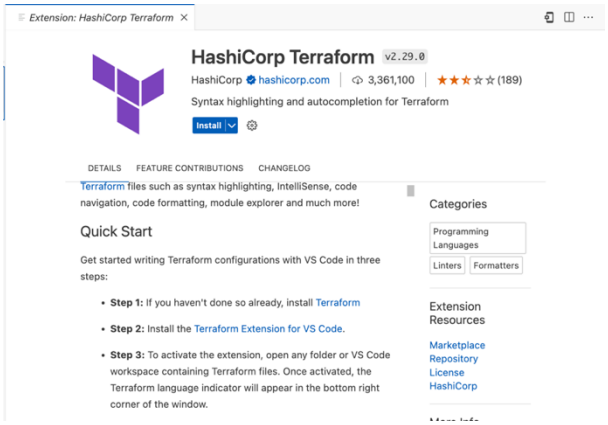

Figure 12. HashiCorp Terraform Extension in VS Code

This extension enhances your experience in VS Code by offering syntax highlighting, code IntelliSense, formatting tools, and more. It simplifies writing Infrastructure as Code using Terraform.

AZ CLI

The AZ CLI, a Microsoft command line tool, facilitates resource management in Azure. It provides commands for deploying, configuring, and managing Azure resources. Compatible with different OS and Azure Cloud Shell, AZ CLI enables communication between Terraform CLI and Azure for code validation and deployment. Among the authentication options for Terraform CLI, using AZ CLI authentication is straightforward.

To install AZ CLI on your computer, go to here[7] and select the appropriate option for your computer operating system.

Once you have installed *az cli*, you can run following command to verify the AZ acli installation and version.

```
$ az -v

azure-cli 2.52.0 *

core 2.52.0 *

telemetry 1.1.0

Dependencies:

msal 1.24.0b1

azure-mgmt-resource 23.1.0b2
```

[7] https://learn.microsoft.com/en-us/cli/azure/install-azure-cli

Once you have the *az cli* installed you can login to azure using az login command.

```
$ az login

A web browser has been opened at https://login.microsoftonline.com/organizations/oau\
th2/v2.0/authorize. Please continue the login in the web browser. If no web browser
is available or if the web browser fails to open, use device code flow with `az logi
n --use-device-code`.
```

The *az login* command opens a browser window and ask you to login using your azure login credentials. Once you are logged in to Azure in a browser, the CLI automatically gets the list of Azure subscription you have access to and lists them in the terminal console.

Now, once you are logged in to Azure using az cli in the terminal console, this authenticated session is also available to terraform cli in the same terminal window. We will use this approach to authenticate with Terraform while deploying resources to Azure using Infrastructure as Code. Terraform offers many other options for authenticating with Azure, you can find more details for other options here[8]

Git

The next tool you need is git client on your local computer. Before we get into git installation, let's briefly discuss what Git is.

Git is a distributed version control system that allows team collaboration for code changes. Git has two components: a git server, which hosts your Code on a remote location, and a git client on your local computer that can fetch and push code changes from the local copy of the git repository (commonly known as repository clone) to the remote repository on the git server.

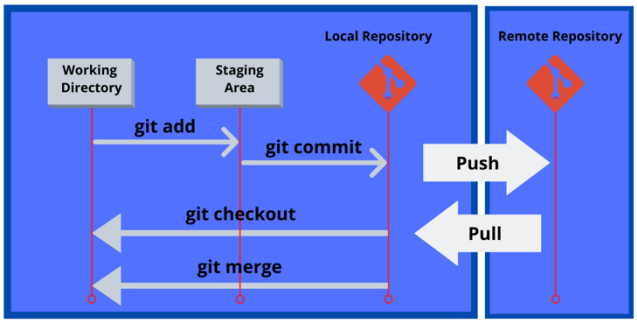

Figure 13. Git Version Control

[8] https://registry.terraform.io/providers/hashicorp/azuread/latest/docs#authenticating-to-azure-active-directory

You can host your own git server or use a Software-as-a-Service (SaaS) solution like GitHub or Azure Repos to host your remote git repository. In this book, we will use GitHub as a remote git server for hosting our Infrastructure as Code.

GitHub allows you to create git repository and create pipelines using GitHub Actions workflow. We will know more about using GitHub as we progress in this book.

To work with a remote git repository, you will need a git client on your local computer. Git client is available for Linux, Windows, and macOS. To install git client, go to here[9] , download, and install the appropriate version based on your computer operating system.

Once you have installed git on your local computer, you may need to configure it for the first time.

```
$ git config ---global user.name "Your Name"

$ git config ---global user.email "YourEMail@Address.com"
```

Git uses this information to tag your commits in the remote repository. Now, with git installed and configured on your local computer, you can clone our book GitHub repository, which contains all the examples code. Git clone helps you make a copy of the repository from the remote server, i.e., GitHub to your computer, enabling you to make changes locally. All our examples are hosted on this[10] Github repository.

To clone this repository on your local computer, in the terminal windows, go to a folder location where you want to make a copy of this repository and type the following command

```
$ git clone https://github.com/PacktPublishing/Practical-Azure-Infrastructure-Manage\
ment.git
```

This command will download the git repository from GitHub to the current folder.

Terraform File Basics

Now that you have the required tooling on your computer, we can start to work with Terraform. Let's look at a basic terraform file that creates a resource group and a storage account in Azure. You can find this file here[11].

All terraform files have .tf extension and are written in a language commonly known as HashiCorp Configuration Language (HCL). HCL is a declarative infrastructure as code language, but it does not need programming expertise, unlike general-purpose languages such as C# or javascript. Declarative languages such as HCL tend to follow a syntax where you need to define or declare what you want

[9] https://git-scm.com/downloads
[10] https://github.com/ashishrajsrivastava/Architecting-and-Implementing-DevOps-for-Infrastructure-Management-in-Azure
[11] https://github.com/ashishrajsrivastava/Architecting-and-Implementing-DevOps-for-Infrastructure-Management-in-Azure/blob/main/ch1/terraform/terraform-basics.tf

to deploy, and the tooling behind knows how to get that done. The terraform CLI, in combination with terraform provider, implements the logic to understand and apply declared Infrastructure as Code written with HCL.

Terraform configuration files contain many blocks where different configurations related to terraform are declared. The first block in our example file is the terraform block, as shown below:

```
terraform {

required_version = ">= 1.0.0"

required_providers {

azurerm = {

source = "hashicorp/azurerm"

version = "=3.0.0"

}

}

}
```

This block defines the providers required by this terraform file to be able to target the appropriate platform for deploying Infrastructure resources. In the examples above, we are configuring terraform to download the azurerm provider.

Terraform providers are the medium through which terraform CLI connects with remote systems. In this case azurerm provider allows terraform cli to connect with Azure Resource Manager APIs to deploy the resources.

To download the *azurerm* provider, we need to declare two properties: *source* and *version*. Source property defines how and from where the provider will be downloaded. This is a repository location where the provider is hosted. In our case, *azurerm* is an official provider from HashiCorp, so we have configured the official repository *hashicorp/azurerm*.

The next property for the *azurerm* provider is *version*, which defines the version of the provider to be downloaded. Terraform constantly updates the official providers and publishes newer versions. While working with a resource provider in terraform, it is important to control the version of the provider that has been used to define your Infrastructure as Code. Locking a version is often not mandatory but always recommended.

The next block is *provider* configuration.

```
provider "azurerm" {

features {}

}
```

This block allows configuration for *azurerm* provider. It allows you to define authentication methods such as Service Principal, Managed Identity, or OpenID connect that need to be used by terraform cli. Since we will be using az cli for authentication, no special declaration is needed in this block. The *feature* block within the *azurerm* provider block allows you to define custom behavior for certain resources. For example, some users may want to expand a managed disk of a VM without downtime or vice versa. Since we are going to use the default behavior of the Azure provider for all the resources we will create in this example, we have kept it empty.

The *resource* block declares a resource type that needs to be created in azure. In this example, we are creating a resource group in azure using *azurerm_resource_group* resource provider.

```
resource "azurerm_resource_group" "rg" {

name = "example-resources"

location = "West Europe"

}
```

The *azurerm* provider offers different resource blocks to deploy and manage respective resources in azure. The *azurerm_resoruce_group* block is responsible for deploying and managing resource groups in azure. The rg part in resource declaration is known as local reference. This is like a pointer that can be used to reference arguments and attributes from this resource. Every resource provider offers different attributes, which are essentially output properties of the resource after creation. For example, id is one of the attributes that gets generated once a resource group is created. Similarly, you define input properties for creating resource, and these are called arguments. For example, in the above code, we are declaring a resource group with *name* argument having the value example-resources and the *location* argument set to West Europe. These are arguments for this resource block. Upon deploying this terraform file, a resource group in azure will be created with name example-resources in West Europe azure location (also known as region).

The next *resource* block declares a storage account in Azure using *azurerm_storage_account* block. Just like previous block, this is also a resource block but with different resource type *azurerm_storage_account*

```
resource "azurerm_storage_account" "sa" {

name = "tflearnsa01"

resource_group_name = azurerm_resource_group.rg.name

location = azurerm_resource_group.rg.location

account_tier = "Standard"

account_replication_type = "LRS"

}
```

The *azurerm_storage_account* resource block helps you deploy and manage storage accounts in Azure. As this is a different resource, it has a different set of arguments and attributes available. Here, we can see how to use the local reference name as we are using the resource group local reference name *azurerm_resoource_group.rg* argument to get the resource group name and location to set the respective argument for the storage account.

Terraform Workflow

When working with terraform files, you will go through a set of commands in a workflow sequence during different development stages of terraform files.

Figure 14. Terraform command workflow

The above figure shows five terraform commands that are usually followed in this sequence. The *terraform init* command is used to initialize the directory containing a terraform file. During this command, terraform cli reads the provider block in the terraform file and downloads the corresponding provider in .terraform folder. To initialize the file we just discussed, navigate to the repository location on your local computer, then go to the ch1 folder and the terraform folder. Run terraform init in the terminal:

```
$ terraform init

Initializing the backend...

Initializing provider plugins...

- Finding hashicorp/azurerm versions matching "3.0.0"...

- Installing hashicorp/azurerm v3.0.0...

- Installed hashicorp/azurerm v3.0.0 (signed by HashiCorp)

Terraform has created a lock file *.terraform.lock.hcl* to record the provider
```

It will download azurerm provider into the *.terraform* folder and create a lock file.

Figure 15. Terraform lock file

The lock file is created to make sure the provider version is locked for this terraform configuration file.

The provider binary usually has a huge file size. For example, the *azurerm* provider it downloaded for us has about 219MB. It is not recommended to commit such big-size binary files in a git repository. In the case of terraform, this file is not mandatory to maintain in the version control repository either. You need this file locally when you run terraform CLI commands. When a new user starts working with these files, they can clone the repo and run *terraform init* to download their own copy of the provider file locally on their computer.

To avoid certain files from being committed to the git version control repository, you can use a special file *.gitignore*. In this file, you can declare files or entire folders that needs to be ignored by git for version control.

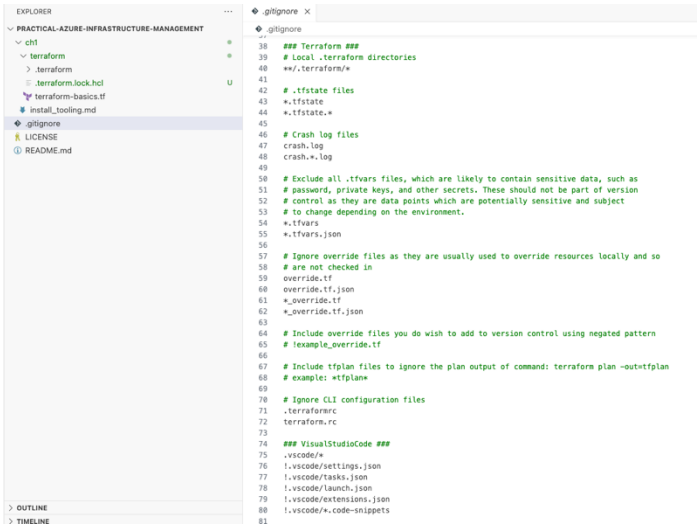

Figure 16. .gitignore file

For example, we have a *.gitignore* file in the repository folder that contains all sample code files. This file ignores all files in the *.terraform* folder. We have also added exceptions for other files, such as files generated by VS Code or operating systems that we do not want to push to the remote repository.

Coming back to the terraform workflow, the next command is *terraform validate*. The command allows you to validate the configuration for any syntax error, such as correct attribute names and value types, etc. If the configuration has no syntax error, then running *terraform validate* gives the following output

```
$ terraform validate

Success! The configuration is valid.
```

This command does not connect with any external resources, such as provider API or remote state store, to validate the actual configuration of resources. State files are special files with name *terraform.tfstate* locally or in a remote backend. When you deploy infrastructure as code with terraform, state files are created either locally or on a remote backend, containing the state of the deployed resources. Upon subsequent deployment, this state file is used to evaluate the current state of infrastructure and any new changes being applied with any code change in terraform files. In the next command, we will discuss how this file is used.

Once you have validated the configuration, you can run the *terraform plan* command to evaluate the infrastructure that will be created or modified with the content in the terraform files.

```
$ terraform plan

+ create

Terraform will perform the following actions:

# azurerm_resource_group.rg will be created

+ resource "azurerm_resource_group" "rg" {

+ id = (known after apply)

+ location = "westeurope"

+ name = "example-resources"

}

# azurerm_storage_account.sa will be created

+ resource "azurerm_storage_account" "sa" {

+ access_tier = (known after apply)

+ account_kind = "StorageV2"

+ account_replication_type = "LRS"

+ account_tier = "Standard"

}

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

After running the *terraform plan*, you will get more detailed output containing more resource attributes in your terminal when you run the command. We have removed additional attributes to make it readable on this page. This command evaluates the terraform blocks in the configuration file and gives you a preview of resources that are going to be created, modified, or deleted if we deploy these terraform files to Azure.

The command *terraform plan* itself does not apply any changes to the resources in azure; rather, it reads the *terraform.tfstate* state file, if present, and compare it against the current state of infrastructure and changes being deployed. If no state file is present, terraform plan will evaluate all changes as part of a new deployment and show all resources with a create state.

Now that we see terraform plan gives us expected preview of resources, we move to the next command, which is *terraform apply*. This command deploys resources defined in the terraform file to Azure.

```
$ terraform apply

+ create

Terraform will perform the following actions:

# azurerm_resource_group.rg will be created

+ resource "azurerm_resource_group" "rg" {

+ id = (known after apply)

+ location = "westeurope"

+ name = "example-resources"

}

# azurerm_storage_account.sa will be created

+ resource "azurerm_storage_account" "sa" {

+ access_tier = (known after apply)

+ account_kind = "StorageV2"

+ account_replication_type = "LRS"

+ account_tier = "Standard"

+ table_encryption_key_type = "Service"

}

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

Do you want to perform these actions?

Terraform will perform the actions described above.
```

```
40
41 Only 'yes' will be accepted to approve.
42
43 Enter a value: yes
44
45 azurerm_resource_group.rg: Creating\...
46
47 azurerm_resource_group.rg: Creation complete after 2s
```

After running *terraform apply*, command asks for your confirmation before deploying the resources to azure. You will need to type yes, as shown in the above output, to confirm. In cases where you want to directly deploy resources without the confirmation prompt, you can use *terraform apply -auto-approve*.

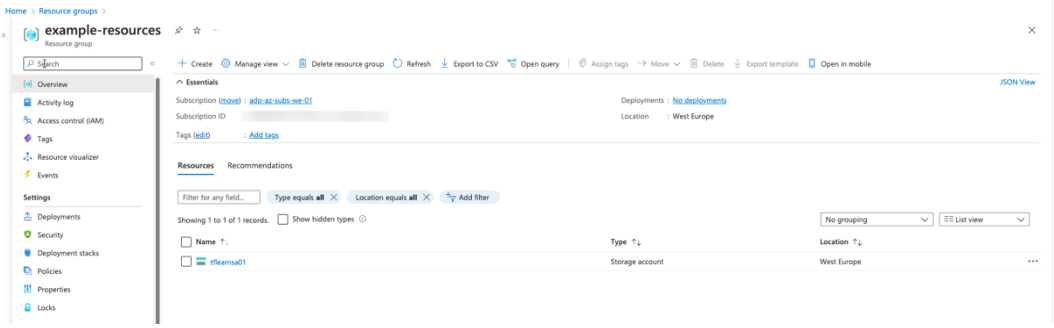

Figure 17. Azure portal showing created resources

You can view the resource group and storage account that has been deployed to Azure by going to the Azure portal.

Figure 18. Terraform state file

Once resources are deployed successfully to Azure using the *terraform apply* command, it creates the local state file named *tfstate.state* that is used for evaluating the state of Infrastructure as code, as discussed previously. In the next exercise, we will use a remote backend to maintain the state file.

The next and last command in terraform workflow is *terraform destroy*. This command is used to

delete or destroy the infrastructure deployed to Azure. This command will delete all the resources declared in the terraform configuration file. Usually, you will either create new resources or modify existing resources using the *terraform apply* command after making changes to terraform configuration files. But sometimes, terraform is used to create ephemeral resources that are needed for a short period and must be completely deleted later. For example, you may create a VM and associated resources in Azure that will be used by a performance test pipeline, and once the performance test is executed, the VM, along with other associated resources, needs to be deleted. You can use terraform to define this entire infrastructure and use *terraform apply* to create all the resources, run the performance tests, and then run *terraform destroy* to delete the entire VM and associated resources.

In our case, since we have completed this first exercise of setting up terraform, we will delete the resources by using *terraform destroy*.

```
$ terraform destroy

azurerm_resource_group.rg: Refreshing state... [id=/subscriptions/xxxx/resourceGroup\
s/example-resources]

azurerm_storage_account.sa: Refreshing state...[id=/subscriptions/xxxx/resourceGroup\
s/example-resources/providers/Microsoft.Storage/storageAccounts/tflearnsa01]

Terraform used the selected providers to generate the following execution plan. Reso\
urce actions are indicated with the following symbols:

- destroy

Terraform will perform the following actions:

# azurerm_resource_group.rg will be destroyed

- resource "azurerm_resource_group" "rg" {

- id = "/subscriptions/996ec04d-f171-4281-a17b-ca0209711e2b/resourceGroups/example-r\
esources" -> null

- location = "westeurope" -> null

- name = "example-resources" -> null

- tags = {} -> null

}
```

```
30
31  # azurerm_storage_account.sa will be destroyed
32
33  - resource "azurerm_storage_account" "sa" {
34
35  }
36
37  Plan: 0 to add, 0 to change, 2 to destroy.
38
39  azurerm_storage_account.sa: Destruction complete after 5s
40
41  azurerm_resource_group.rg: Destruction complete after 1m17s
42
43  Destroy complete! Resources: 2 destroyed.
```

Just like *terraform apply*, the *terraform destroy* also asks for a confirmation prompt. After confirmation, it shows you the details of deleted resources with their respective attributes. This command also accepts *-auto-approve* to skip the confirmation prompt.

After deleting the created resource, we have completed all the commands in the terraform workflow. While we ran each command in sequence, many times you may not run all the commands every time you work with the same terraform file. For example, *terraform init* is used for initializing the provider, and after that, you may not run it again unless you delete the provider folder or clone a new repository of terraform files. Similarly, you may not use *terraform destroy* frequently, as you may not delete entire resources declared in the terraform configuration files; rather, you would modify or delete selected resources from the terraform configuration file and use the *terraform apply* command for updating resource states in Azure. The most commonly used commands in day-to-day terraform development are *validate*, *plan*, and *apply*.

Summary

In this chapter, we covered the fundamentals of cloud infrastructure and how organizations benefit from cloud as compared to on-premises infrastructure. We learned about Microsoft Azure cloud architecture and its foundational components. We discussed Azure Resource Manager and how you can interact with it using different tools. Then, we discussed what DevOps practices are and how infrastructure teams improve their efficiency in automated deployment using practices such as Infrastructure as Code. Furthermore, we initiated our journey with terraform, understanding its workflow and deploying a basic terraform configuration on Azure. In the upcoming chapter, we'll explore Azure Landing Zone, focusing on customizing it using Infrastructure as Code and deploying it for the establishment of enterprise-grade Azure foundational components.

Chapter 2 : Architect and deploy Azure Landing Zone with terraform

In this chapter, we will look at the Microsoft cloud adoption framework and how an organization can achieve its cloud adoption objectives by following the Microsoft cloud adoption framework. We will learn how to follow the cloud adoption framework design principles using Azure Landing Zone and set up Azure cloud adoption on an enterprise scale. We will learn how to implement Azure landing zone with Terraform using infrastructure as code. We will also go through the steps you need to follow to customize the Terraform module for the landing zone, ensuring it aligns with your organization's specific requirements.

In this chapter, we will explore the following topics that are very critical for a successful cloud adoption journey:

- Microsoft Cloud Adoption Framework
- Azure Landing Zone
- Customizing and Deploying Landing zone using terraform

Technical requirements

By now, you should be familiar with the cloud concept discussed in Chapter 1. You should understand Azure cloud components such as management groups, subscriptions, and resource groups and be familiar with their usage. Following the practical exercises, you will need the tools installed in Chapter One and should be familiar with the Terraform command workflow. You can find all the exercise code used in this chapter here[1] . While following the practical exercises, you will need access to an Azure account with an active subscription. If you do not have an Azure account, please get a free Azure account here[2]

Microsoft Cloud Adoption Framework

Organizations must go through various best practices, tools, and discussions when they plan for cloud adoption on an enterprise scale. Based on the customer cloud adoption journey and in collaboration with internal teams, Microsoft has developed a framework called Microsoft Cloud Adoption Framework. Microsoft Cloud Adoption Framework is a full lifecycle of different processes,

[1] https://github.com/ashishrajsrivastava/Architecting-and-Implementing-DevOps-for-Infrastructure-Management-in-Azure
[2] https://azure.microsoft.com/en-us/free/

guides, and tools that help different stakeholders and guide them through the cloud adoption journey. As per Microsoft Cloud Adoption Framework, an organization goes through the following lifecycle repeatedly during its entire journey to cloud adoption.

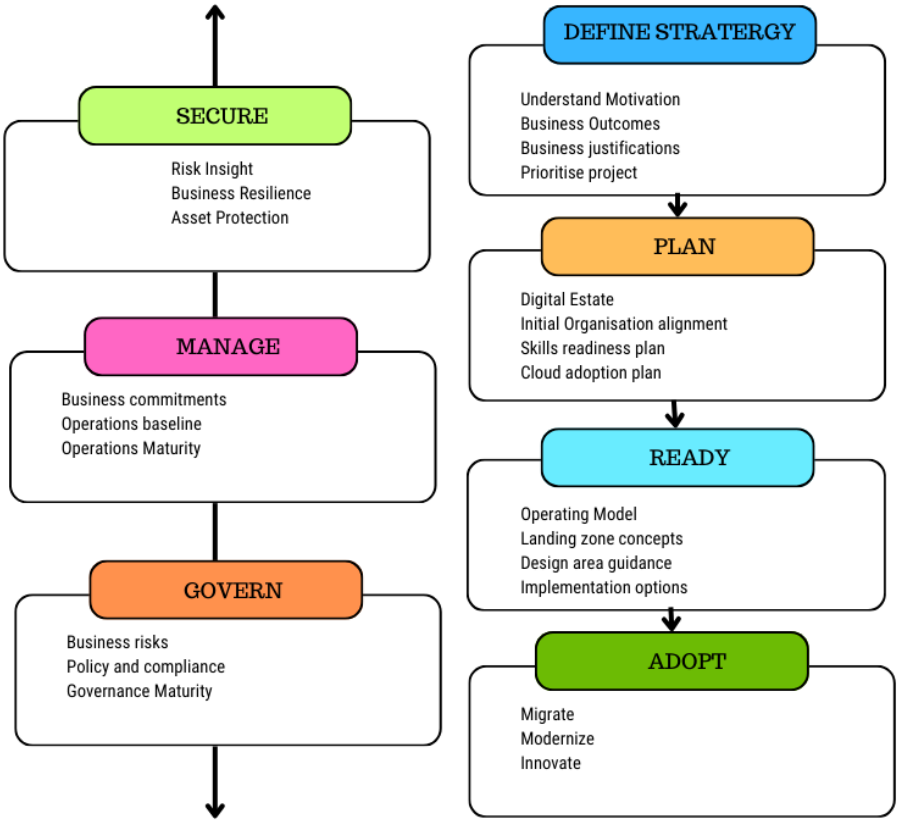

Figure 19. Figure 2.1 Microsoft Cloud Adoption Framework Lifecycle

Each of these phases comes with a comprehensive set of guides, tools, and processes, which would require an entire book to explain in detail. While this book primarily focuses on one phase under the ***Ready*** phase, where we deploy the landing zone, having a basic understanding of the other phases is essential to grasp the full context of our purpose. We will look at these phases to ensure we understand their significance before implementing a landing zone using Infrastructure as Code with Terraform.

Define Strategy

This is the first step in the cloud adoption journey, where different stakeholders align on the motivations for why organizations must adopt the cloud. These motivations can vary from organization to organization. For example, saving costs can be the primary motivation for

some organizations toward cloud adoption. However, for other organizations, infrastructure modernization can motivate them to adopt cloud. Based on what motivates an organization to think about cloud adoption, the business outcomes of cloud adoption are decided. These outcomes are well-defined goals that must be met with cloud adoption. Organizations align these goals with the business objectives and list down justifications for cloud adoption initiatives. With cloud adoption, organizations will also go through a financial transformation with respect to their spending on IT Infrastructure. Since cloud changes the way organizations spend money on IT infrastructure, moving them from the CAPEX to OPEX model. This also means that organizations will no longer have to maintain many data centers and can save on associated costs. Businesses must also evaluate the need for on-premises infrastructure presence based on industry regulatory requirements. Such technical details and associated business requirements are very important in cloud adoption strategy and aligning project priority.

Plan

After defining strategy and cloud adoption goals, organizations need to plan their adoption journey by rationalizing their digital estate based on the business goals defined in the strategy phase. Organizations need to evaluate digital assets by discovering their current state, determining installed components on workloads, and analyzing asset usage and dependencies on other applications. Organizations often use the five Rs as a base for digital estate rationalization:

1. ***Rehost***–Also known as lift and shift. No change in the app code or its usage.
2. ***Refactor***–The application may not need a significant amount of architectural changes, but refactoring some application code or configurations can give you more benefits from cloud. For example, you may have an on-premises web application that uses an on-premises Microsoft SQL Server for databases. With little change in code and configuration, you can start using Azure SQL databases which not only takes away the burden of operational activities of SQL server but also gives you a database with better scalability and availability.
3. ***Rearchitect***–You may have a part of the solution that may benefit more from cloud if rearchitected with cloud-native design principles such as employing microservices patterns and using cloud services such as Azure Kubernetes Services.
4. ***Rebuild***–If an application is to be moved and is not following cloud-native-supported architecture, you need to rebuild the application with a completely new stack of technologies. For example, if you have an application developed in Classic ASP, no cloud service directly supports this framework. You may need significant refactoring, and even then, it might not yield the desired results for hosting the solution in the cloud. It's better to rebuild the application using a more modern framework, such as ASP.NET.
5. ***Replace***–The application is no longer needed, as expected business value can be easily achieved using a SaaS solution. For example, O365, as a mailing solution, has almost replaced the on-premises mail exchange servers in many organizations.

To rationalize, each project is evaluated by going through a complete inventory of assets such as software, hardware, operating systems, etc. This can take time in big enterprises. This time-

consuming process can slow the cloud adoption journey and may result in unsatisfying results compared to expected business outcomes.

To avoid such delays, organizations should initially follow an incremental approach to digital estate planning. They should not wait for the entire digital estate mapping and then only start the next step; rather, they should take iterative steps to evaluate projects, consider them based on priority, and come up with a project backlog for cloud adoption.

With cloud adoption, people and processes need to keep pace with the transformation journey. Organizations may form different functions, such as a Cloud Center of Excellence, Governance, Platform, Automation, Adoption, and Operations. These functions ensure that cloud adoption in the organization goes in the right direction while aligned with business goals.

To make a successful cloud adoption journey, it is important to have the right skill readiness plan in place. As the organization goes through a cloud adoption journey, the skills required by different roles may change significantly. It may raise various concerns among different staff members due to a new set of skills and processes. Organizations must capture these concerns, evaluate the impacts, and look for the available resources to address these concerns right from the beginning.

Ready

In the ***Ready*** phase, the organization begins the setup of Azure foundational components required for starting the project migration in the ***Plan*** phase. These components involve setting up management groups, subscriptions, policies, and other platform or application-centric resources in Azure. Microsoft Cloud Adoption Framework introduces Azure Landing Zone to help you set up these components. In upcoming sections, we will discuss Azure Landing Zone and its deployment options.

Adopt

As the foundational components in Azure are ready, the project team can plan cloud adoption for their applications. Adopt phase processes help the project team plan their ***Migration***, ***Modernization***, ***Innovation*** or ***Relocate***.

Usually, projects start with a migration objective to move out of the on-premises environment and quickly kick off the cloud adoption journey. As cloud adoption matures, projects evolve to modernize their applications, aiming to achieve more from the cloud in terms of cost optimization, reliability, performance, and more. Cloud also empowers organizations to innovate with data and application capabilities that help them build better predictive analytics, high-performing applications, and more customer-focused solutions. As organizations grow and expand globally, they need solutions deployed in different geographical regions. Cloud allows organizations to easily Relocate the solutions to meet the business requirements for global expansions, data sovereignty, and lower latency requirements for users.

Manage

A thorough cloud strategy, planning, and adoption lay the foundation for delivering a successful cloud strategy. However, it's important to note that managing ongoing assets is what drives business outcomes for organizations. It is very important for organizations to document the workload deployed to cloud, their operational commitments such as SLA, and have a clear picture of impacts on cloud investments due to any outages to cloud workloads.

It involves maintaining a cloud asset inventory, along with the tools and processes needed for configuration management, as well as platform tools to protect, recover, and optimize workload operations.

Govern

Cloud introduces new technologies within an organization, which means the organization must develop new ways of governing these technologies, compared to traditional on-premises solutions. Organizations must align cloud-ready corporate policies, regulatory compliance, cloud security and data classifications. Initially, it is recommended to follow a ***minimum viable product (MVP)*** approach when setting up governance for cloud workload and plan iteratively to mature governance as cloud adoption grows in the organizations. As the organization goes through cloud adoption, it learns the cloud-native approaches to meet its compliance needs. Progressively, organizations should implement disciplines such as cost management, security baseline, identity baseline, resource consistency, and DevOps practices for deployment

Secure

With cloud adoption, organization security posturing also changes with new set of tools and processes. Cloud security is a continuous process that matures as organizations move further in their cloud adoption journey. While applications hosted in the cloud offer better security options, it's equally important for organizations to actively use those features and controls. Cloud security is a shared responsibility between the cloud provider and the organizations using the cloud. Organizations must align their business strategies to gain risk insights, integrate security, and ensure business continuity even in the event of cyber-attacks, with the ability to quickly regain full control.

Organizations must set up ***Access Control*** to establish a zero-trust approach in the network. A mature security operation discipline ensures that there is a setup for threat detection, a response mechanism, and speedy recovery of victimized assets in the event of such an attack. In addition to security operations, organizations must establish an Asset Protection discipline to identify vulnerable assets, classify them, and implement methods to protect these assets. As an organization moves forward in its cloud adoption maturity, it adds new technology stacks with varied security posture requirements. Due to this frequent change in the technology stack, it is important to have a Security governance. Security governance ensures security postures are reviewed regularly to discover compliance issues and provide further input to improve security postures. With traditional

infrastructure approaches, security often comes in at the last step of the application life cycle, but not with the cloud. With the growing popularity of the DevSecOps approach, security is recommended to shift left in the application development life cycle.

Azure Landing Zone

Based on the Cloud Adoption Framework, the Azure landing zone is guided by key design principles to implement eight design areas of Azure Cloud. These design principles serve as a guide for deciding on design across technical domains in cloud adoption. We will quickly review these guiding principles before we discuss the eight design areas of the Azure Landing Zone.

When moving workload to cloud, the application team first needs a subscription where the application can be hosted. The design principle recommends we follow subscription democratization for subscription vending. It encourages the use of subscriptions as the unit of management for applications onboarded to the cloud. This means that the application team should dedicate subscriptions to host their workloads with autonomy. However, with autonomy, organizations also want these subscriptions to remain compliant with their internal policies and regulatory requirements. So, the next design principle suggests that each subscription in the landing zone should follow the appropriate management group hierarchy, and policy should be used to enforce governance. Organizations are also recommended to use a single control and management plan to establish a set of policies and access controls that are appropriate for the organization. Another design principle suggests that application onboarding or migration should always be application-centric, rather than relying on lift-and-shift methods or PaaS services. This means that design choices for platforms should not be based on IaaS, PaaS, or new or old applications. It should give the same level of security and controls irrespective of the type of workload. Design principles also advocate that organizations should *align with the Azure Road* map while making design decisions to ensure new Azure capabilities are readily available in the organization environment.

Now using these above principles, Azure Landing Zone provides an environment across eight design areas. The following figure shows these design areas:

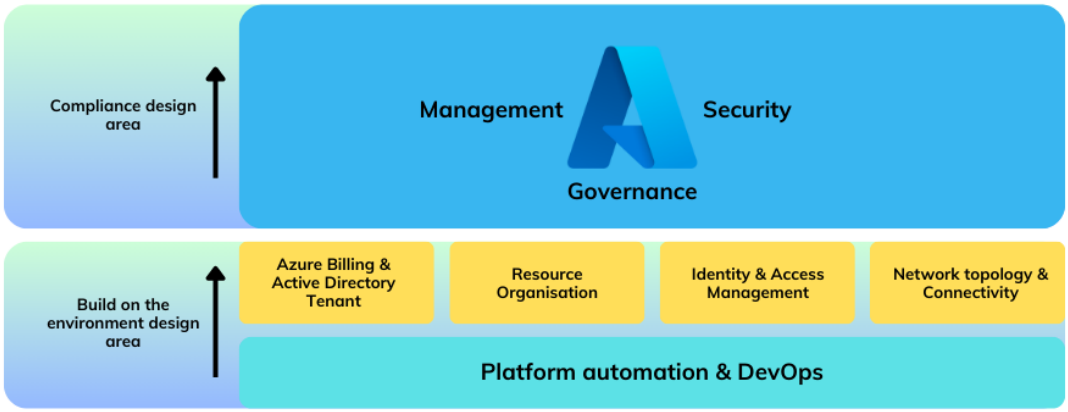

Figure 20. Azure Landing Zone Design Areas

1. ***Azure Billing and Active Directory***: This design area guides you through setting up enterprise enrollment, subscription vending, Azure accounts, and billing. It is the first area to plan when starting the cloud onboarding process. Initially, this may involve a manual setup of enterprise agreements, configuring the appropriate Azure Active Directory, and creating a few platform-based subscriptions. Later on, you'll primarily need subscription vending to deploy various application landing zones, which can be automated using infrastructure as code.
2. ***Identity and Access management***: This design area covers setting up roles, permissions, and appropriate access policies. It is important to have a centralized Identity and Access management system to ensure that proper identity and access management is applied to landing zone resources.
3. ***Resource organization***: This design area guides you through the appropriate resource hierarchy from the management group to the application landing zone subscriptions. This hierarchy helps establish proper governance policies in the landing zone.
4. ***Network topology and connectivity***: This design area guides you through setting up a centralized network and its connectivity with application-specific landing zone network

structure and configurations. It also guides you through setting up other centralized resources, such as Firewalls, DNS zones, and other network-related configurations that must be applied from the platform landing zone to other application landing zones.

5. ***Security***: This design area covers different security and processes that ensure proper security postures are applied to a cloud environment. This includes various security-specific controls on a network level, firewall policies, governance policies, and other cloud-native security tools like Azure Defender for cloud setup.
6. ***Management***: This design area guides you through various management-related landing zone resources, including Azure automation accounts, Log Analytics workspaces, resource organization, and backup and disaster recovery setup.
7. ***Governance***: This also guides you through similar resources, such as Azure policies and log analytics workspace setup, but it focuses more on setting up auditing and governance policies.
8. ***Platform automation and DevOps***: This design area guides you through setting up the right processes and tools to deploy landing zones.

Microsoft Azure Landing Zone has a reference architecture that uses these design areas, as shown in the following figure:

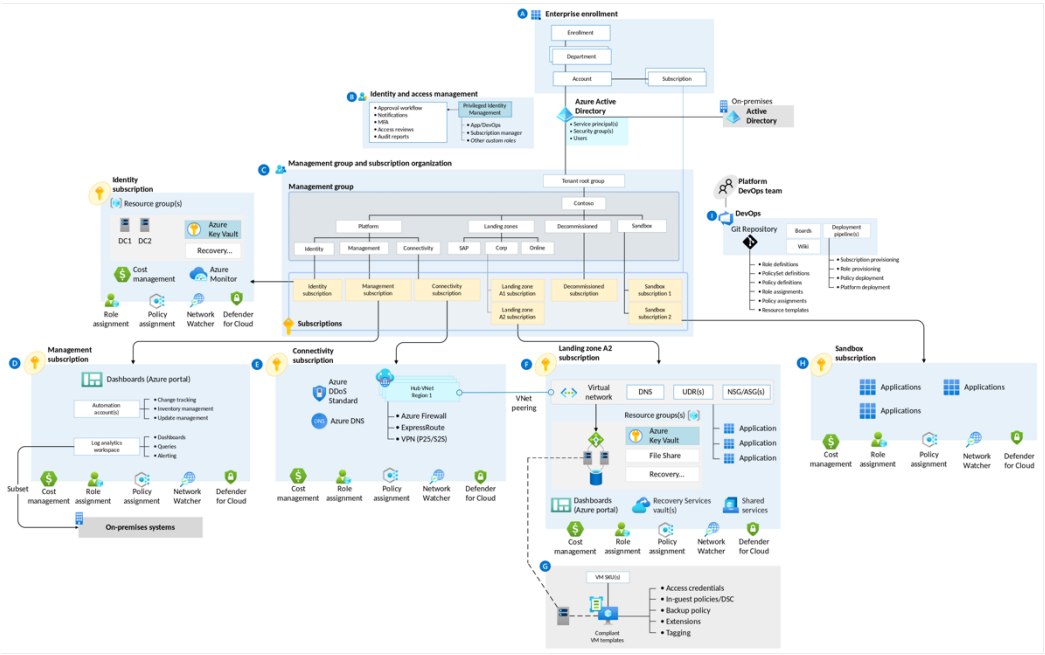

Figure 21. Azure Landing Zone Conceptual Architecture

The above figure shows the practical mapping of azure components with the design areas we discussed before. Here is the mapping of design areas to the resource components in the Azure Landing Zone reference architecture:

Design Areas	Resource Componentes
A	Azure Billing and Active Directory
B	Identity and Access Management
E	Network Topology and connectivity
C	Resource Organization
F	Security
D, G, H	Management
C, D	Governance
I	Platform and Automation

When we deploy these resources, as shown in the reference architecture, we categorize landing zone deployment into two types:

- ***Platform Landing Zone***: The Platform Landing Zone consists of subscriptions and resources used for deploying central management resources or shared services such as identity, connectivity, or management-related resources. These landing zone resources are deployed, managed, and used centrally by a central IT team.
- ***Application Landing Zone***: The application Landing Zone consists of subscriptions and resources required by workload resources. The application landing zone depends on the services and configurations provided by the platform landing zone. The application landing zone is controlled by the management group for governance but is used by application teams to host their application-specific resources. As the application landing zone is used by workload hosting, the application team needs the required accesses to manage their specific workload resources in Azure. So, management of application landing zone can be achieved using three different approaches:

  - Completely managed by a central IT team.
  - Application landing zone managed by application team.
  - Shared responsibility model between platform team and application team.

It depends on your organization's teams and processes to decide which one of the three approaches works for you.

Deploy landing zone with Terraform

As discussed in the previous section, application landing zone resources depend on shared or central resources in the platform landing zone. These shared services include networking, identity management, governance policies, and monitoring. These centralized services enable organizations to set up an operational standard for all application landing zone workloads. So, before we deploy the application landing zone, we must have these resources offered by the platform landing zone.

Typically, landing zone deployment goes through four iterative phases, as shown in the following figure:

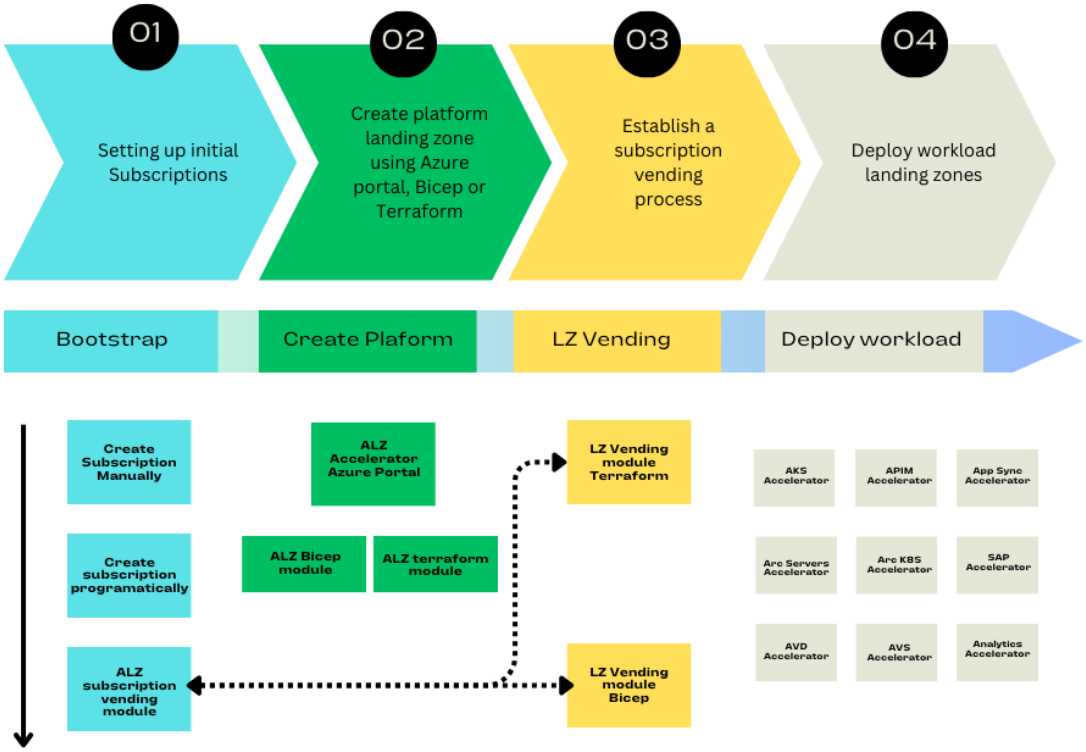

Figure 22. Azure Landing Zone deployment journey

The first phase involves creating subscriptions. Whether you already have a landing zone or are starting fresh with a greenfield deployment, subscriptions will be required. The subscriptions are used by both platform and application landing zones to host landing zone resources. The subscriptions can be created manually or programmatically using REST API, PowerShell, or AZ CLI. In a greenfield deployment where you first deploy the platform landing zone, you manually create a platform subscription and deploy the platform landing zone. With further advancement in your application landing zone deployment, you will implement a subscription vending process using Infrastructure as Code using Bicep or Terraform.

The next phase is the deployment of the platform landing zone. The platform landing zone deploys all the foundational resources that are shared with or used to manage application landing zones. Platform landing zone can be deployed from the Azure portal or using Infrastructure as code with Bicep and Terraform.

Once the platform landing zone is set up, you will need to make sure you have subscriptions to configure application landing zones. Here, you will use the subscription vending process that ensures the created subscriptions have appropriate resources such as virtual networks, role assignments, and other preconfigured settings. Microsoft offers Terraform and Bicep modules for automating this process. Application landing zone can also be deployed using Infrastructure as code with Bicep or Terraform. Microsoft offers various application or workload-specific landing zone accelerator modules.

As mentioned earlier, landing zone deployments are available in both Bicep and Terraform, but this book will focus on the Terraform approach. With Terraform, you have two options for Azure Landing Zone deployment:

- ***ALZ Terraform Accelerator***: This is a preconfigured implementation of Terraform Azure Landing Zone modules. ALZ Terraform Accelerator only requires you to manually do a few configurations for prerequisite, bootstrap certain resources in Azure DevOps/GitHub and Azure. You get preconfigured terraform files with CI and CD setup in Azure DevOps/GitHub, and you can simply run that. For this approach, you need not edit any Infrastructure as code modules, as all recommended settings are already configured. Though it may take some time to set up prerequisite and bootstrap resources, it is one of the quickest and simplest ways to set up the entire Azure Landing Zone in your organization. The downside is that the Azure Landing Zone deployed is not customized as per your organization's needs. This option is good to start exploring the resources deployed in a Landing Zone deployment in the early phases. This option can help you evaluate your organization's needs for landing zone customization but may not give a landing zone for production use.
- ***Azure Landing Zones Terraform Module (caf-enterprise-scale)*** : This is a Terraform module with different capabilities covering Azure Landing zone design areas. The ALZ Terraform Accelerator option discussed above uses this Terraform module for deployment with preconfigured settings. As discussed, the ALZ Accelerator module preconfigured settings may not align with your organization's landing zone needs. So, instead of using ALZ Terraform Accelerator, you can simply use the underlying module to either customize or deploy a default landing zone capabilities offered by this module.

The Azure landing zone terraform module capabilities are built around different design areas of the Cloud Adoption Framework. The following reference architecture shows the Azure landing zone terraform module focus areas as per Azure Landing Zone reference architecture:

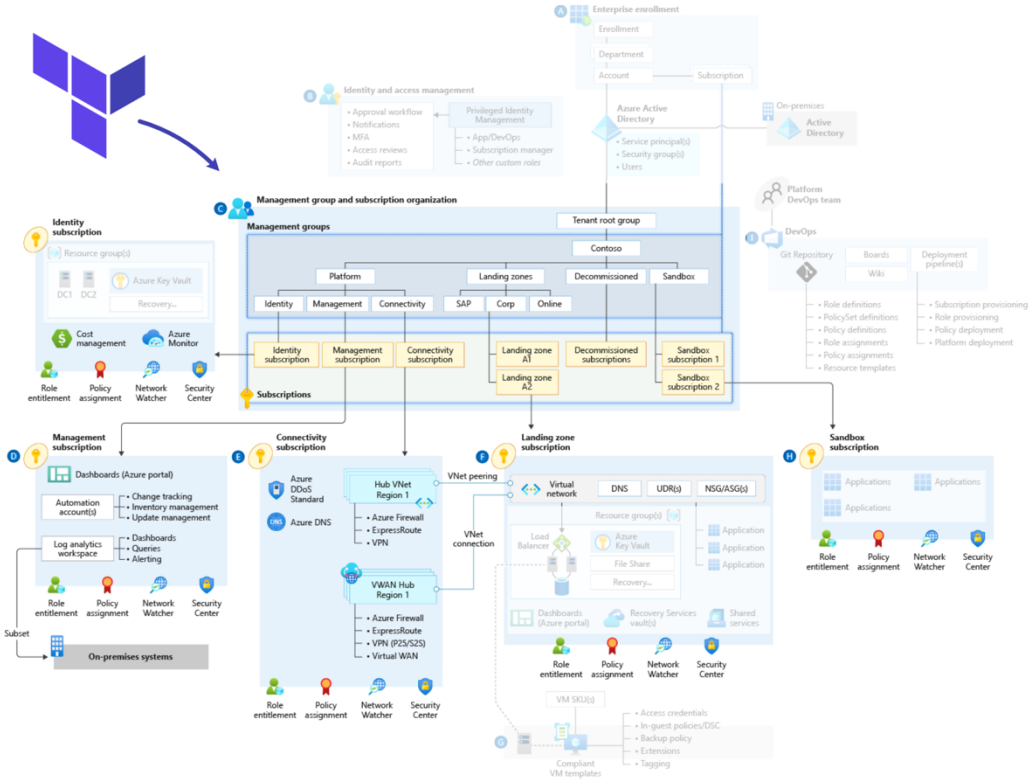

Figure 23. Azure landing zone terraform module capabilities

As shown in the above figure, the azure landing zones terraform module covers these four capabilities:

- ***Core Resources***–This capability focuses on the resource organization design area and deploys foundational resources according to the Azure Landing Zone reference architecture.

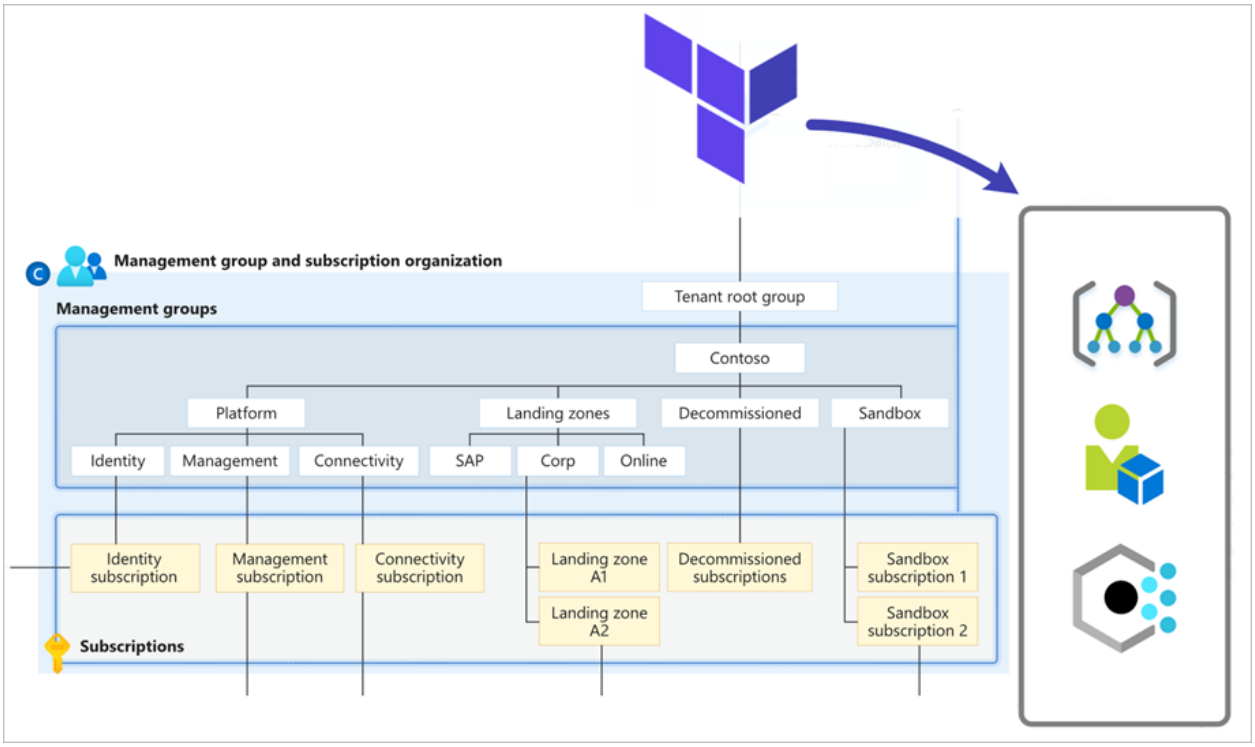

Figure 24. Core Resources

As shown in the above figure, core capability deploys management groups and associated governance resources and focuses on resource organization design areas.

- ***Management Resources*** - It covers the **management** design area from the Cloud Adoption Framework and deploys different management-related resources, such as the Log Analytics workspace, Automation Account, and associated resource groups and relevant policy assignments.

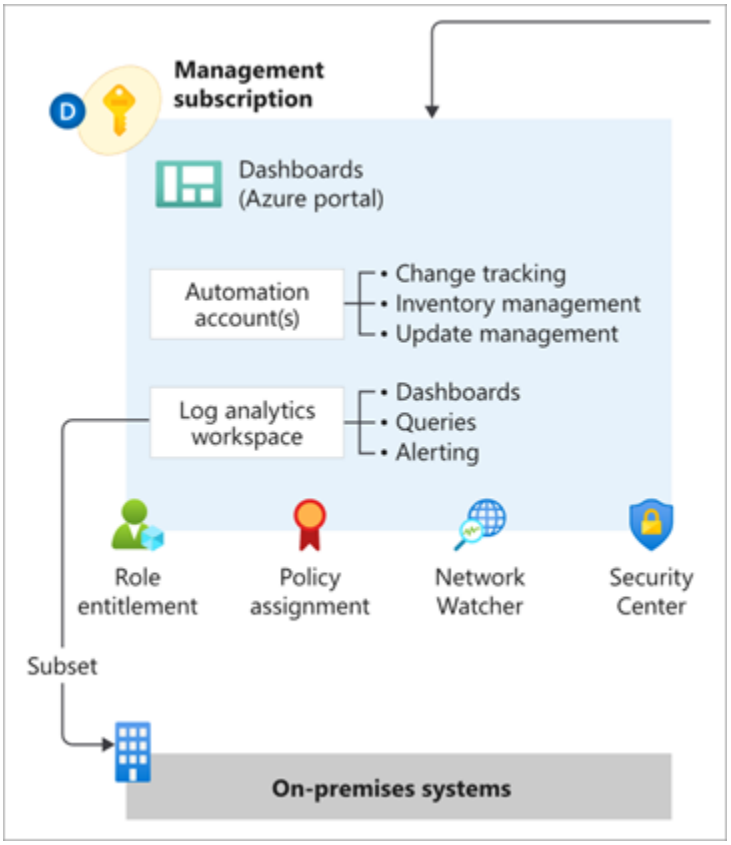

Figure 25. Management Resoruces

The above figure shows the management design area resources from the Azure Landing zone reference architecture, which are deployed by the management capability of the Terraform module.

- ***Connectivity Resources*** - This capability deploys resources that are part of the network topology and connectivity design area of the Azure Landing Zone reference architecture.

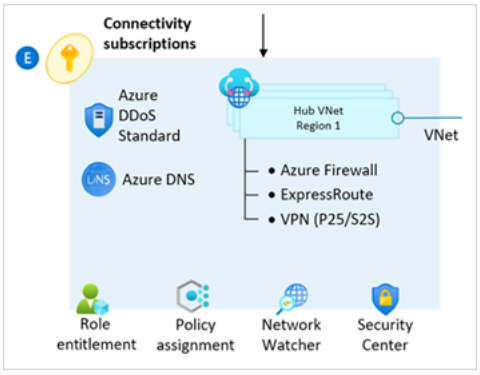

Figure 26. Connectivity Resources

As shown in the diagram above, this capability deploys resources such as Virtual Networks, associated subnets, Azure Firewall, public IP addresses, and other associated networking configurations such as peering.

- ***Identity Resources*** - This capability is aligned with identity management and access management design area in Azure Landing zone reference architecture. This capability does not deploy any resource in Azure but only configures appropriate Azure policies:

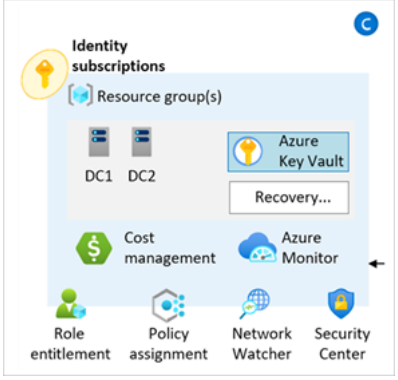

Figure 27. Identity Resources

The above figure shows the resources that belong to the identity and access management design area of the Azure Landing Zone reference architecture. As we have already said, the Terraform ALZ module does not deploy any resources but only configures policies belonging to this design area. The resources shown above need to be deployed by your Infrastructure as code terraform configuration based on your organization's needs.

Landing zone deployment

We discussed earlier that there are two options available for landing zone deployment using Terraform – the ALZ accelerator or the Terraform ALZ Module. We will not deploy the landing zone using the ALZ Accelerator, which sets up everything with default configurations. This approach does not allow us to customize the resources as per organizational requirements. Also, it is important to note that you may not need all the capabilities that come along with the ALZ accelerator approach or may want a few capabilities with a default configuration but also want to be able to customize other capabilities deployment as per your organization's needs. Based on your organization's requirements, you can opt not to adopt one of the capabilities at all and choose to have your own approach to handle that design area. Since the Azure Landing Zones Terraform module allows you to selectively customize and deploy capabilities, we will start with this approach.

We will begin with deploying core capability with a default configuration that deploys resource organization design area of Azure Landing Zone.

Before we start working on the Landing zone deployment configuration, let's go back and open the terraform configuration we deployed in *Chapter 1* and have a look at the first block:

Figure 28. Terraform Configuration

```
terraform {
required_version = "\>= 1.0.0"
required_providers {
azurerm = {
source = "hashicorp/azurerm"
version = "=3.0.0"
           }
      }
}
```

As discussed in *Chapter 1*, this block dictates the terraform CLI version required, the provider version, and its configuration. Now, let's create a similar file with the name *main.tf*. This time, we will add a new block named backend and its associated arguments. You can find this file on our GitHub repository under the *ch2* folder here: here[3]:

Figure 29. Terraform Configuration with Backend

```
terraform {
required_providers {
azurerm = {
source = "hashicorp/azurerm"
version = ">= 3.74.0"
          }
    }
backend "azurerm" {
resource_group_name = "iac-terraform-state-rg"
storage_account_name = "iacbookstate2023"
container_name = "alzcoretfstate"
key = "dev.alz.terraform.tfstate"
                }
     }
provider "azurerm" {
features {}
}
```

We have added a *backend* block within the *terraform* block. The backend block allows you to configure the *.tfstate* files, which contains the current state of deployed resources using the terraform configuration files. By default, terraform uses a local backend that stores the tfstate file in the local folder. Declaring the local backend is optional unless you require custom behavior. That's why in chapter one example, when we ran terraform apply command to deploy the resources,

[3]https://github.com/ashishrajsrivastava/Architecting-and-Implementing-DevOps-for-Infrastructure-Management-in-Azure/blob/main/ch2/alz-core/main.tf

we could see a *.tfstate* file, which was automatically created by the local backend. Having this *.tfstate* file locally is fine if only one person is making changes to the terraform configuration and applying those changes, but that is not the practical case in organizations. You usually need to collaborate with people, use many tools, and share this state when they want to make changes to infrastructure resources. For example, you may have deployed the storage account as demonstrated in Chapter 1, but someone from your team may have to add more configuration to make changes to the storage account. For such collaborative Infrastructure as Code development, they not only need this terraform configuration file that created the storage account but also the state file that contains the currently deployed state of the storage account. In a DevOps team, it is essential to have a proper state strategy and use a remote backend that is accessible to all team members, including CI/CD tools. Terraform allows you to configure different types of remote backends, including azurerm which stores the state files in a configured Azure storage account container. Apart from azurerm, terraform supports various backend providers such as gcs, s3, and http, etc. You can visit[4] to see a list of all supported backend currently.

In our example, we are configuring *azurerm* backend with a storage account named *iacbookstate2023* in the resource group *iac-terraform-state-rg*. The state file will be stored in the storage account container named *alzcoretfstate* with the file name *dev.alz.terraform.tfstate*.

But before we can use this storage account as a remote backend, we need to deploy this storage account in Azure. Let's deploy it with another terraform configuration file with a local backend. You can deploy this one-time storage account for state management manually from portal, az cli or Azure PowerShell.

Here is the terraform configuration for deploying the required resource group, storage account and container for the remote backend. You can find this configuration in ch2/statestore/statestorage.tf

```
terraform {
required_version = "\>= 1.0.0"
required_providers {
azurerm = {
source = "hashicorp/azurerm"
version = "=3.0.0"
        }
    }
}
provider "azurerm" {
features {}
}
resource "azurerm_resource_group" "state_rg" {
name = "iac-terraform-state-rg"
location = "West Europe"
}
```

[4] https://developer.hashicorp.com/terraform/language/settings/backends/configuration

```
resource "azurerm_storage_account" "state_store_sa" {
name = "iacbookstate2023"
resource_group_name = azurerm_resource_group.state_rg.name
location = azurerm_resource_group.state_rg.location
account_tier = "Standard"
account_replication_type = "LRS"
}
resource "azurerm_storage_container" "state_store_container" {
name = "alzcoretfstate"
storage_account_name = azurerm_storage_account.state_store_sa.name
container_access_type = "private"
}
```

It's pretty much same configuration as we deployed in *Chapter 1* with one additional resource i.e. storage container using *azurerm_storage_container* block.

Let's open the configuration file location in terminal and run *terraform init*:

```
$ terraform init

Initializing the backend...

Initializing provider plugins...

- Finding hashicorp/azurerm versions matching "3.0.0"...

- Installing hashicorp/azurerm v3.0.0...

- Installed hashicorp/azurerm v3.0.0 (signed by HashiCorp)
```

Now, let's validate the configuration for any syntax error by running the command *terraform validate*:

```
$ terraform validate

Success! The configuration is valid.
```

Looks like the configuration is valid and has no syntax error. The next command we will run is the *terraform plan*. As we have already explained this command in *Chapter 1*, we will skip explaining that here. When you practice this exercise, you should run terraform plan locally to validate the resources being created. We ran terraform plan, and it shows the list of resources, indicating that the configuration is correct so far. We will go to the next command – *terraform apply*, to deploy the resources:

```
$ terraform apply

Terraform used the selected providers to generate the following execution plan. Reso\
urce actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

# azurerm_resource_group.state_rg will be created

+ resource "azurerm_resource_group" "state_rg" {

+ id = (known after apply)

+ location = "westeurope"

+ name = "iac-terraform-state-rg"

}

# azurerm_storage_account.state_store_sa will be created

+ resource "azurerm_storage_account" "state_store_sa" {

+ table_encryption_key_type = "Service"

}

# azurerm_storage_container.state_store_container will be created

+ resource "azurerm_storage_container" "state_store_container" {

+ name = "alzcoretfstate"

+ resource_manager_id = (known after apply)

+ storage_account_name = "iacbookstate2023"

}
```

We have removed additional properties from the terminal output to make it readable in the book. You shall expect a lengthier output and more properties in the terminal when you run this command.

As shown in the terminal output, it deploys three resources in Azure. Let's confirm the resources in the Azure portal:

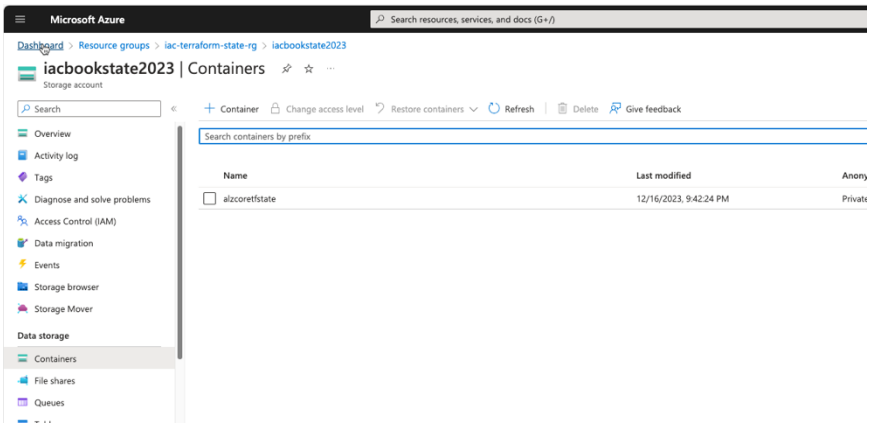

Figure 30. Azure storage account for remote backend

The screenshot above shows the resource group with a storage account and container in it, as expected.

When we open the container, as shown in the following screenshot, it is empty as we have not used it as a remote backend in any terraform configuration deployment yet:

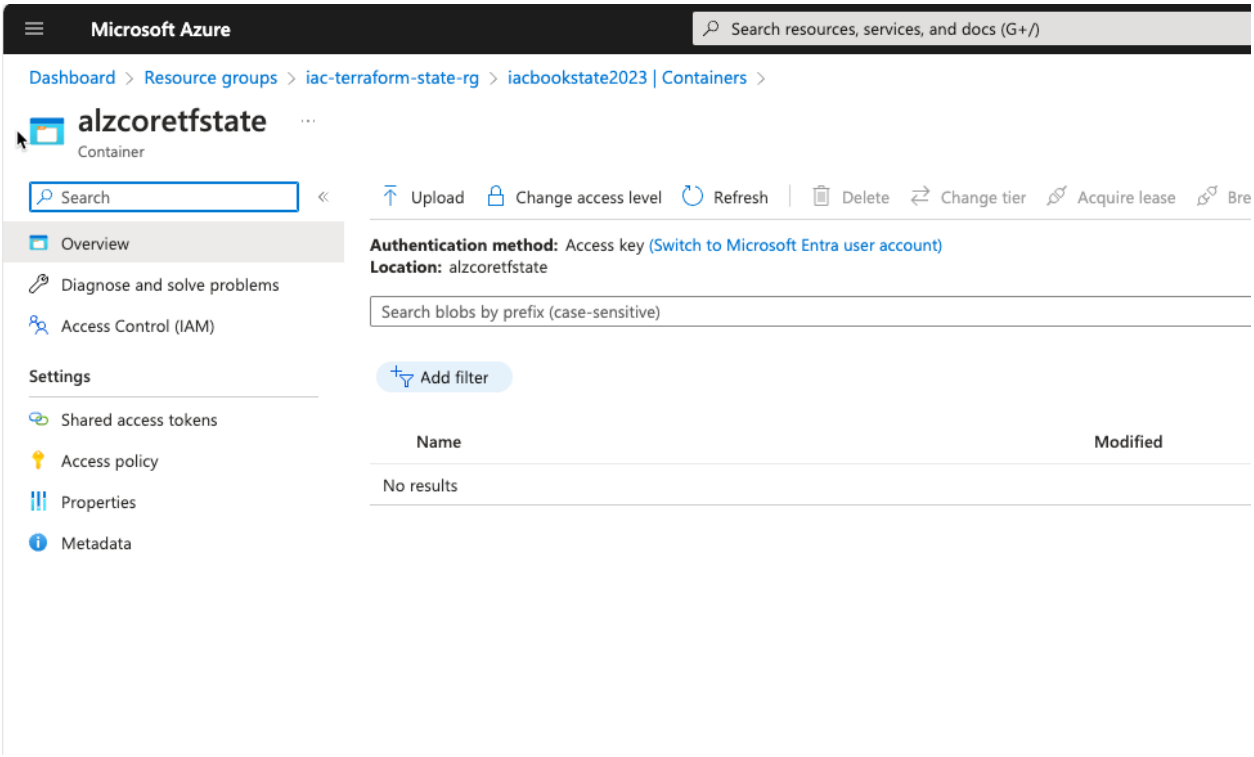

Figure 31. Remote backend container

Since we will use this storage account for all other chapters, we will not destroy it using *terraform destroy*, unlike how we did for the *Chapter 1* exercise resources. If you have a subscription with sufficient credits, you can also keep it for further exercises. But if you are using a free Azure account with limited credits, then you can destroy this after completing the exercise in the chapter by using the *terraform destroy* command. When you practice the exercises in upcoming chapters, you can apply the provided terraform file in the *ch2/statestore* folder to deploy this backend storage account again. Though this storage account does not cost much to keep for the next few days, it's better to destroy and recreate if you are using a subscription with no free credits. Now that we have a storage account deployed, we can use it as a remote backend in Azure landing zone deployment.

Let's come back to *main.tf* in the *alz-core* folder to the next blocks relevant to the terraform ALZ module:

```
terraform {

required_providers {

azurerm = {

source = "hashicorp/azurerm"

version = "\>= 3.74.0"

}

}

backend "azurerm" {

resource_group_name = "iac-terraform-state-rg"

storage_account_name = "iacbookstate2023"

container_name = "alzcoretfstate"

key = "dev.alz.terraform.tfstate"

}

}

provider "azurerm" {

features {}

}

data "azurerm_client_config" "core" {}

module "enterprise_scale" {

source = "Azure/caf-enterprise-scale/azurerm"

version = "5.0.3"

default_location = "westeurope"
```

```
44
45  providers = {
46
47  azurerm = azurerm
48
49  azurerm.connectivity = azurerm
50
51  azurerm.management = azurerm
52
53  }
54
55  root_parent_id = data.azurerm_client_config.core.tenant_id
56
57  root_id = "iac-alz"
58
59  root_name = "IaC ALZ"
60
61  }
```

The next new block we have added as shown in the above code is *azurerm_client_config* data source block.

The data block is used to connect with data sources (or Azure resources) that are defined outside Terraform. They are just like resource blocks, except they are only used for reading the data. In contrast, resource blocks allow Terraform to create, update, or delete resources. Each provider may offer a set of data sources for their resources. For example, you can view one of the data sources offered by azurerm provider for virtual machine resources here[5].

In our *main.tf* file, we are using the data source *azurerm_client_config* for getting configuration such as tenant id of *azurerm* provider. You can find the documentation for this data source here[6]

While working with complex terraform configurations that need to deploy different resources and access data sources, you will need to know all the configuration options for these resources and data sources. You will need to review the documentation pages of these resources and data sources to know the arguments and attributes offered by the respective resource and data source.

The next block is a module block that uses a module named *enterprise_scale*. Modules in Terraform allow you to develop reusable Infrastructure as Code. Module files are maintained in *.tf or .tf.json* format, which defines a set of resources and configurations that need to be reused by preconfigured declarations in modules. Essentially, the *main.tf* file is a root module for the *ch2/alz-core* folder. A root module is defined as a set of terraform configuration files in the root directory. Now, coming back to the module block, this is referred to as a child module being called within the root module. These child modules can be sourced from the local file system or hosted in a remote location. The

[5] https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/data-sources/virtual_machine
[6] https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/data-sources/client_config

module block can be declared with a source destination. In this chapter exercise, we are using a child module named *enterprise_scale* with a source from the Terraform registry.

Just like resource block arguments vary with the type of resources, module block arguments can also be different based on the resources that the module deploys. You can see the full list of arguments accepted by this module at here[7]. Here you can also find all other information like outputs, resources and examples.

We will go with the default configuration of this module, where we declare the *root_id* and *root_name* for it. The *root_id* sets the id for the root management group created by landing zone deployment. The *root_name* sets the Management group name. The tenant ID where this landing zone management group will be deployed will be set by *root_parant_id*, which is set by accessing the Microsoft Entra ID tenant ID from the data source block.

Let's run command *terraform init* to initialize the provider for this configuration:

```
$ terraform init

Initializing the backend...

Successfully configured the backend "azurerm"! Terraform will automatically

use this backend unless the backend configuration changes.

Initializing modules...

Downloading registry.terraform.io/Azure/caf-enterprise-scale/azurerm 5.0.3 for enter\
prise_scale...

- enterprise_scale in .terraform/modules/enterprise_scale

- enterprise_scale.connectivity_resources in .terraform/modules/enterprise_scale/mod\
ules/connectivity

- enterprise_scale.identity_resources in .terraform/modules/enterprise_scale/modules\
/identity

- enterprise_scale.management_group_archetypes in .terraform/modules/enterprise_scal\
e/modules/archetypes

- enterprise_scale.management_resources in .terraform/modules/enterprise_scale/modul\
es/management

```

[7] https://registry.terraform.io/modules/Azure/caf-enterprise-scale/azurerm/latest?tab=inputs

```
- enterprise_scale.role_assignments_for_policy in .terraform/modules/enterprise_scal\
e/modules/role_assignments_for_policy

Initializing provider plugins...

- Finding hashicorp/random versions matching ">= 3.1.0"...

- Installing hashicorp/azurerm v3.85.0...

- Installed hashicorp/azurerm v3.85.0 (signed by HashiCorp)

Partner and community providers are signed by their developers.

If you'd like to know more about provider signing, you can read about it here:

https://www.terraform.io/docs/cli/plugins/signing.html

Terraform has been successfully initialized!
```

This time, *terraform init* has done a bit more than it did for our previous configuration file. You will first notice that it has initialized the remote backend that we declared in the terraform backend block.

Another new thing it did was to download modules in addition to providers. Since the module source is configured as the Terraform registry, it was downloaded from there. Like the provider, modules are also downloaded to the *.terraform* folder, and you can open this folder to have a look:

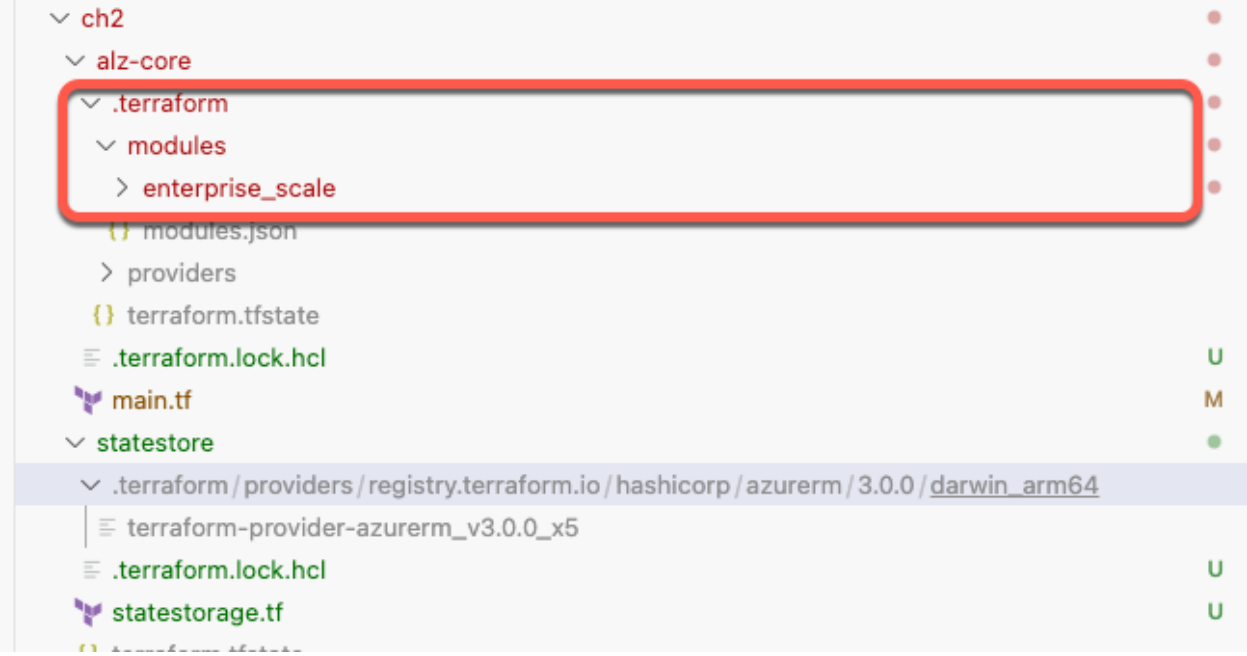

Figure 32. Downloaded terraform modules

Now that we have initialized the provider and modules required by this terraform configuration, let's run *terraform validate*:

```
$ terraform validate

Success! The configuration is valid.
```

As the configuration is valid from a syntax point of view, let's run *terraform plan* to see what and how many resources this configuration deploys if we apply it.

We are skipping the complete terminal output of this command as the output is quite long and instead will just show you the following line from the output:

```
Plan: 233 to add, 0 to change, 0 to destroy.
```

When you run this command on your computer, you should be able to review all the resources displayed in the complete output. Now, we will go ahead and deploy this configuration with the *terraform apply* command. As you can imagine, creating 233 resources will take a while, so go grab a coffee while Terraform deploys all these resources. Terraform will continue the deployment and keep updating the terminal with resources being created. Once it has finished deploying all the resources, you will see the following final message:

```
Apply complete! Resources: 233 added, 0 changed, 0 destroyed.
```

Once all the resources are deployed, you can go to the Azure portal and view the newly created management group structure:

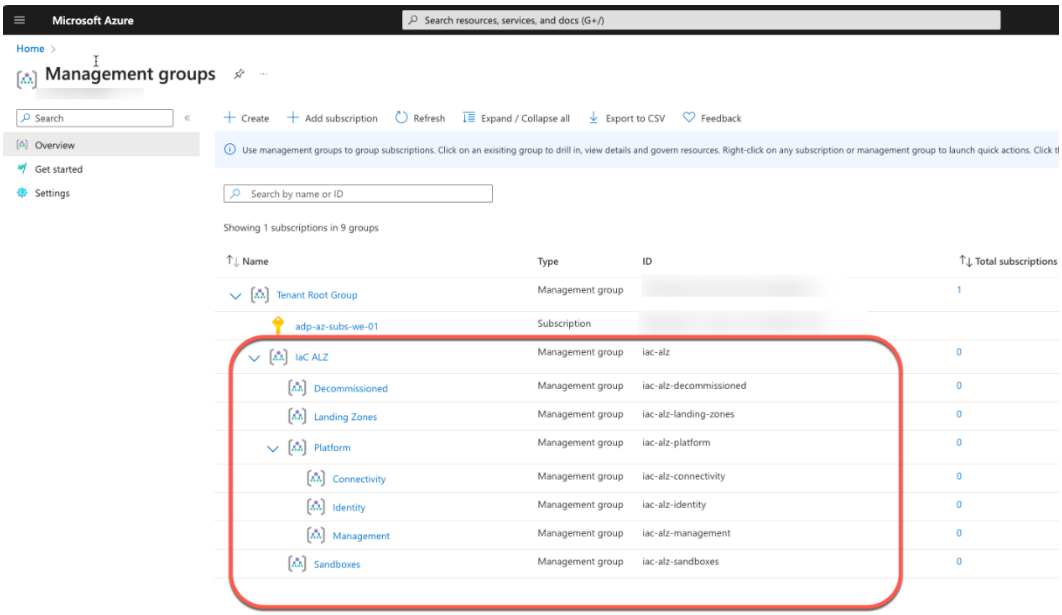

Figure 33. Azure Landing Zone Core Capability deployed

As shown, a management group hierarchy has been created according to the Azure Landing Zone reference architecture we discussed earlier.

In the next section, we will deploy another capability from the Azure Landing Zone Terraform module, but with some customizations.

Customize the landing zone and deploy with terraform

Organizations often have their own regulations and processes that they want to be able to apply while deploying landing zones. For example, organizations may want further management groups to enforce governance rules as per organizational structure. To be able to do that, you will need to customize the landing zone and add those additional management groups and policies that enforce regulations specific to organizations. We will take the same core capability but will customize it to accommodate the following organizational requirements:

- The root management group will be called ADP ALZ with id adp-alz
- There should be two child management groups under *Landing Zone* management group called *ADP Online Global* and *ADP Online EU* with management group id *adp-alz-online-glb* and *adp-alz-online-eu* respectively.

- • The management group *ADP Online Global* should be allowed to deploy resources and resource groups in all Azure regions, whereas *ADP Online EU* should be restricted to the West Europe Azure region for both resources and resource groups.

We will create a set of new files to deploy this requirement in Azure:

- *terraform.tf*
- *main.tf*
- *variable.tf*
- *lib/archetype_definition_adp_online.json*

You can find all the files here[8]

Let's have a look on *terraform.tf* file:

```
terraform {

required_providers {

azurerm = {

source = "hashicorp/azurerm"

version = ">= 3.74.0"

}

}

backend "azurerm" {

resource_group_name = "iac-terraform-state-rg"

storage_account_name = "iacbookstate2023"

container_name = "alzcoretfstate"

key = "prod.alz.terraform.tfstate"

}

```

[8]https://github.com/ashishrajsrivastava/Architecting-and-Implementing-DevOps-for-Infrastructure-Management-in-Azure/tree/main/ch2/custom-alz

```
}

provider "azurerm" {

features {}

}
```

This file contains only the terraform block with provider configuration. We have created this file separately here to have a separation of concern. Now, this configuration does not need to be defined in the *main.tf* file, and that file will only have configuration related to landing zone deployment. You can also notice that the remote backend remains the same, but we have a different state file name now.

Now, let's have a look at the *main.tf* file now:

```
data "azurerm_client_config" "core" {}

module "enterprise_scale" {

source = "Azure/caf-enterprise-scale/azurerm"

version = "5.0.3"

default_location = "westeurope"

providers = {

azurerm = azurerm

azurerm.connectivity = azurerm

azurerm.management = azurerm

}

root_parent_id = data.azurerm_client_config.core.tenant_id

root_id = var.root_id

root_name = var.root_name

library_path = "${path.root}/lib"
```

```
28
29  custom_landing_zones = {
30
31  "${var.root_id}-online-glb" = {
32
33  display_name = "${upper(var.root_id)} Online Global"
34
35  parent_management_group_id = "${var.root_id}-landing-zones"
36
37  subscription_ids = []
38
39  archetype_config = {
40
41  archetype_id = "adp_online"
42
43  parameters = {}
44
45  access_control = {}
46
47  }
48
49  }
50
51  "${var.root_id}-online-eu" = {
52
53  display_name = "${upper(var.root_id)} Online EU"
54
55  parent_management_group_id = "${var.root_id}-landing-zones"
56
57  subscription_ids = []
58
59  archetype_config = {
60
61  archetype_id = "adp_online"
62
63  parameters = {
64
65  Deny-Resource-Locations = {
66
67  listOfAllowedLocations = ["westeurope"]
68
69  }
70
```

```
71  Deny-RSG-Locations = {
72
73  listOfAllowedLocations = ["westeurope"]
74
75  }
76
77  }
78
79  access_control = {}
80
81  }
82
83  }
84
85  }
86
87  }
```

You will notice that we do not have the terraform block anymore in the *main.tf*. The *terraform.tf* file will be used for that now in this terraform configuration.

You can also notice we are no longer using hardcoded values for *root_id* and *root_name* attributes. We are using a variable to declare it; the variable is declared in a separate file called *variables.tf*. It is not necessary to declare variables in a separate *variable.tf* file and they can also be declared in the *main.tf* as well. Just to maintain separation of concerns, we are defining variables in a separate file, as we demonstrated for the terraform block. Variables in Terraform are used to declare a value that can be provided as input at runtime, and its value can be used at different places in a configuration file. Let's have a look at the *variables.tf* file:

```
1   variable "root_id" {
2
3   type = string
4
5   default = "adp-alz"
6
7   }
8
9   variable "root_name" {
10
11  type = string
12
13  default = "ADP ALZ"
14
15  }
```

As shown, we are declaring two variables named *root_id* and *root_name*, both of string type, each initialized with a default value. Other than default value and type, variables can have some other arguments such as description, sensitive, nullable, and validation for applying validation rules such as length of the value being provided, etc. The declared variable value can be accessed using the expression *var.*. As you can see, in *main.tf*, we are assigning *root_id = var.root_id*. We will extensively use variables in many exercises and discuss their different usage pattern throughout the book.

Now, let's come to the next new argument we have declared in the alz module, which is *library_path = "${path.root}/lib"*. This declared library path for something called archetype definitions files. Archetypes in Azure Landing Zone allow us to enforce governance guardrails on a specific management group level. The guardrails can be policies, RBAC rules, or other centrally managed resources such as virtual networks, Microsoft Defender for cloud, logging, etc.

Landing zone default deployment already configured landing zone management groups within the built archetype. You can find all the in-built archetype definitions here[9]. In our exercise, we will create a custom archetype called *adp-online* and its definition will be in the lib folder. Let's have a look at this file:

```
{

"adp_online": {

"policy_assignments": ["Deny-Resource-Locations", "Deny-RSG-Locations"],

"policy_definitions": [],

"policy_set_definitions": [],

"role_definitions": [],

"archetype_config": {

"parameters": {

"Deny-Resource-Locations": {

"listOfAllowedLocations": [

"westeurope",

"eastus",

```

[9] https://github.com/Azure/terraform-azurerm-caf-enterprise-scale/tree/main/modules/archetypes/lib/archetype_definitions

```
"eastus2",

"westus",

"northcentralus",

"southcentralus"

]

},

"Deny-RSG-Locations": {

"listOfAllowedLocations": [

"westeurope",

"eastus",

"eastus2",

"westus",

"northcentralus",

"southcentralus"

]

}

},

"access_control": {}

}

}

}
```

In the above custom archetype definition, we are defining a custom archetype definition called *adp_-*

online. This file follows a special schema accepted by the Azure Landing Zone terraform module. In this archetype definition, we are creating two policy assignments for denying resource group and resource location. You can find more details about archetype schema for terraform module here[10]

We are using this archetype definition in the *main.tf* under *custom_landing_zone* attribute with *archetype_config* parameter.

The *custom_landing_zone* attribute of the ALZ terraform module allows us to define the additional management group and their associated configuration. We have defined two additional landing zone management groups, with the landing zone management group as a parent and with a custom archetype for restricting geolocation as per organizational requirements.

Now, since we have all the required customization, We can run *terraform init* to initialize the provider and modules. If you deleted the backend created in the previous exercise, then please first run *terraform apply* by going into the ch2/statestore folder in the terminal. Since we have not deleted the backend, we can safely run *terraform init* in the *ch2/custom-alz* folder:

```
$ terraform init

Initializing the backend...

Successfully configured the backend "azurerm"! Terraform will automatically
```

We have now configured the backend and downloaded all providers and modules. We have removed unnecessary details from the terminal output to show only the important bits in the book. In your system, you will get a lengthier terminal output with details of all the providers and modules.

Now we will run *terraform validate* and *terraform plan* to validate the syntax and resources being created by this configuration:

```
Plan: 239 to add, 0 to change, 0 to destroy.
```

As we can see, it's going to create about 239 resources, so again, this will take time after running the command *terraform apply*. So again, grab a cup of coffee while Terraform is doing its work:

```
Apply complete! Resources: 239 added, 0 changed, 0 destroyed.
```

The deployment was completed after a few minutes. Let's have a look into to Azure portal:

[10] https://github.com/Azure/terraform-azurerm-caf-enterprise-scale/wiki/%5BUser-Guide%5D-Archetype-Definitions

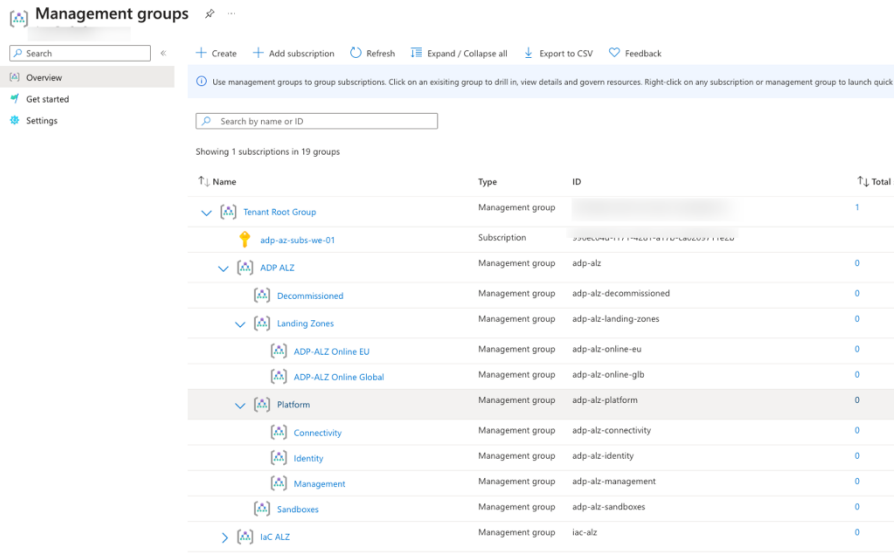

Figure 34. Custom Azure Landing Zone

As shown in the screenshot above, the landing zone now contains two more management groups, as provided in the terraform configuration. If we look at the policy assignments of *ADP - ALZ Online EU* management group, we can see that configured restrictions for geolocation are applied via Azure policy:

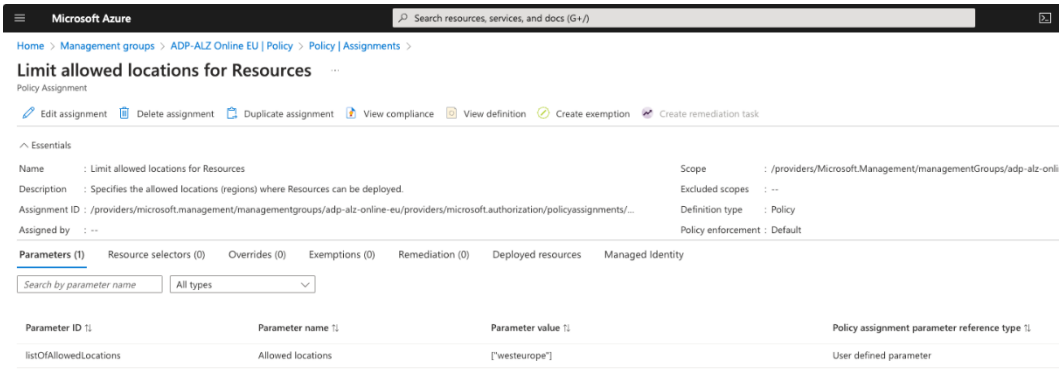

Figure 35. Policy assignment for geo region restrictions

So, as you can see, we have successfully deployed the Azure landing zone with custom configurations for core capabilities. Following this approach, you can deploy further capabilities either with custom or default configurations. Each capability offers different types of resources, so the kind of customization will vary. For example, when working with connectivity resources, it makes sense to customize the virtual network and IP addressing, rather than deploying the default IP address schema provided by the standard configurations. Depending on your customization needs, you will adjust various resource attributes and variables.

Summary

We explored the Microsoft Cloud Adoption Framework and how it helps organizations develop strategies, plan, and further adopt the cloud. We discussed the different phases of the Cloud Adoption Framework and how organizations can leverage it for a successful cloud adoption journey. We introduced Azure Landing Zone and the design areas it implements from the Microsoft Cloud Adoption Framework. We later introduced the Azure landing zone terraform module for Azure landing deployment with Infrastructure as Code. We learned how to deploy a default configuration of Azure landing zone with Terraform. In the final step, we also learn how you can customize the Azure landing zone deployment with the Terraform module to meet an organization's governance needs.

Chapter 3 : Deploying Highly Available Azure VM and Networks with Terraform

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Deploy Web App VM Infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Deploying Azure Virtual Machine

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Deploy Application Gateway

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Connect to VM securly using Azure Bastion Host

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Configure multi-region VM infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Global Routing using Azure Front Door

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Chapter 4 : Implementing Continuous Integration for Terraform with GitHub Actions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Introduction to Continuous Integration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Git–Source Code Version Control

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Continuous Integration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Continuous Delivery & Continuous Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Working with GitHub Actions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Creating GitHub Repository

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

GitHub Actions Workflow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Deploying Terraform IaC using GitHub Actions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Triggers in GitHub Actions workflow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

GitHub Actions Workflow Jobs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

GitHub runners

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Running GitHub Actions Workflow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Preview Terraform Changes in Pull Request

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Deploying Terraform IaC using GitHub Actions workflow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Chapter 5 : Deploying an Azure Container App Infrastructure using terraform and GitHub actions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Introduction to Containers

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Container Hosting Options in Azure

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Azure Web Apps

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Azure Functions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Azure Kubernetes Services

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Azure Container Apps

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Deploying Azure Container Apps Environment

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Deploying Azure Container Apps

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Deploying Azure Container Registry & Container Image

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Deploying Container Image from ACR to Azure Container Apps

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Deploying Container Apps using GitHub Actions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Chapter 6 : Configuration Management using Ansible in Azure

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Configuration Management

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Introduction to Configuration as Code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Configuration Management Tools

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Chef

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Puppet

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Ansible

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Getting Started with Configuration Management in Azure

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Creating Linux VM with Ansible

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Chapter 7 :Combining Configuration Management with Infrastructure as Code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Infrastructure as Code & Configuration as Code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Planning Infrastructure as Code & Configuration as Code

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Using Terraform and Ansible together

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/devopsinazure.