TERRAFORM CLASS-1

The Traditional Approach to Cloud Management

When starting with cloud computing, most users prefer using the **management console**. It's a simple, graphical interface that allows you to create resources by clicking through menus. For small organizations operating in a single region, this approach works well.

But what happens when your organization grows?

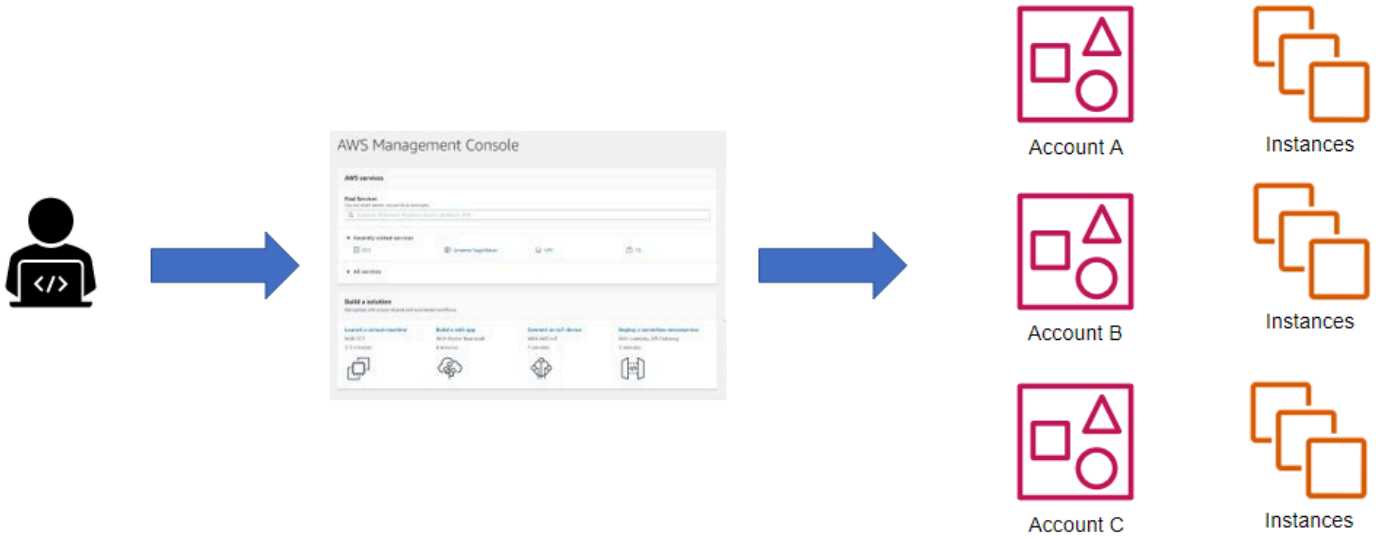
Let's take an example: You've created a web application infrastructure with servers, databases, and load balancers using the management console. Now, your company decides to expand and needs the same setup in a different region or account.

You would have to repeat all those steps manually—setting up each server, configuring the database, and linking everything together. This is fine for one or two regions, but imagine having to do this across **5 or 10 regions** or **multiple accounts**.

Now, let's make it worse. Suppose your security team requires you to update a configuration, like changing the encryption settings or updating access rules. You'd have to go into the console and manually update this in **every single environment**.

Real-life pain point:
A mid-sized e-commerce company using this approach once had to replicate their infrastructure across three regions during a flash sale event. A single typo in one region's setup caused servers to misconfigure, leading to downtime and lost sales in that region.

This traditional, manual approach becomes inefficient, error-prone, and time-consuming as your cloud usage grows.



Infrastructure as Code (IaC): The Scalable Solution

Think of **Infrastructure as Code (IaC)** as a **blueprint** for your organization's cloud environment—like a reusable recipe for building your infrastructure. But it's more than just a blueprint; it brings in powerful tools that developers already use, such as **version control**, **testing**, **code reviews**, and **CI/CD pipelines**.

Here's how it works: Instead of manually clicking through the management console to set up servers, databases, or load balancers, developers write code to define the entire infrastructure. This code is saved in files (like YAML, JSON, or Terraform scripts) and can be reused to create consistent environments.

Real-World Example:

Let's say a SaaS company needs to set up environments for **development**, **testing**, and **production**. Using IaC, the infrastructure for the **development environment** can be written once and then used to spin up identical environments for **testing** and **production** with minimal effort.

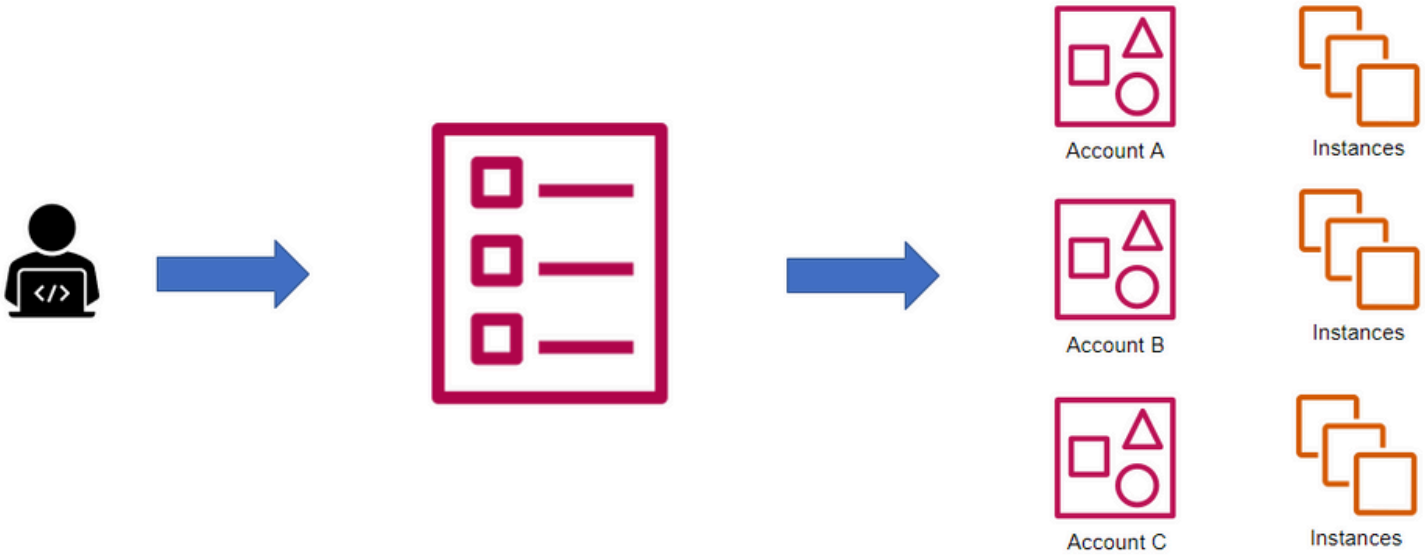
Benefits:

1. **Consistency**: The same code creates identical environments every time, reducing human errors.
2. **Speed**: Need to launch a new region or replicate infrastructure in a new account? Just run the IaC script, and it's ready in minutes.
3. **Cost-Efficiency**: Automation saves time and reduces the need for manual effort.
4. **Easy Updates**: When a security policy changes, you can update the IaC code and apply the change across all environments in one go.

Real-Life Pain Point Solved:

A fintech startup faced issues with their production environment setup when they expanded to Europe. Using IaC (with Terraform), they reused their U.S. environment blueprint to deploy an identical setup in Europe. This saved weeks of manual effort and ensured compliance with regional regulations.

In short, IaC turns manual, error-prone tasks into automated, reliable, and efficient workflows.

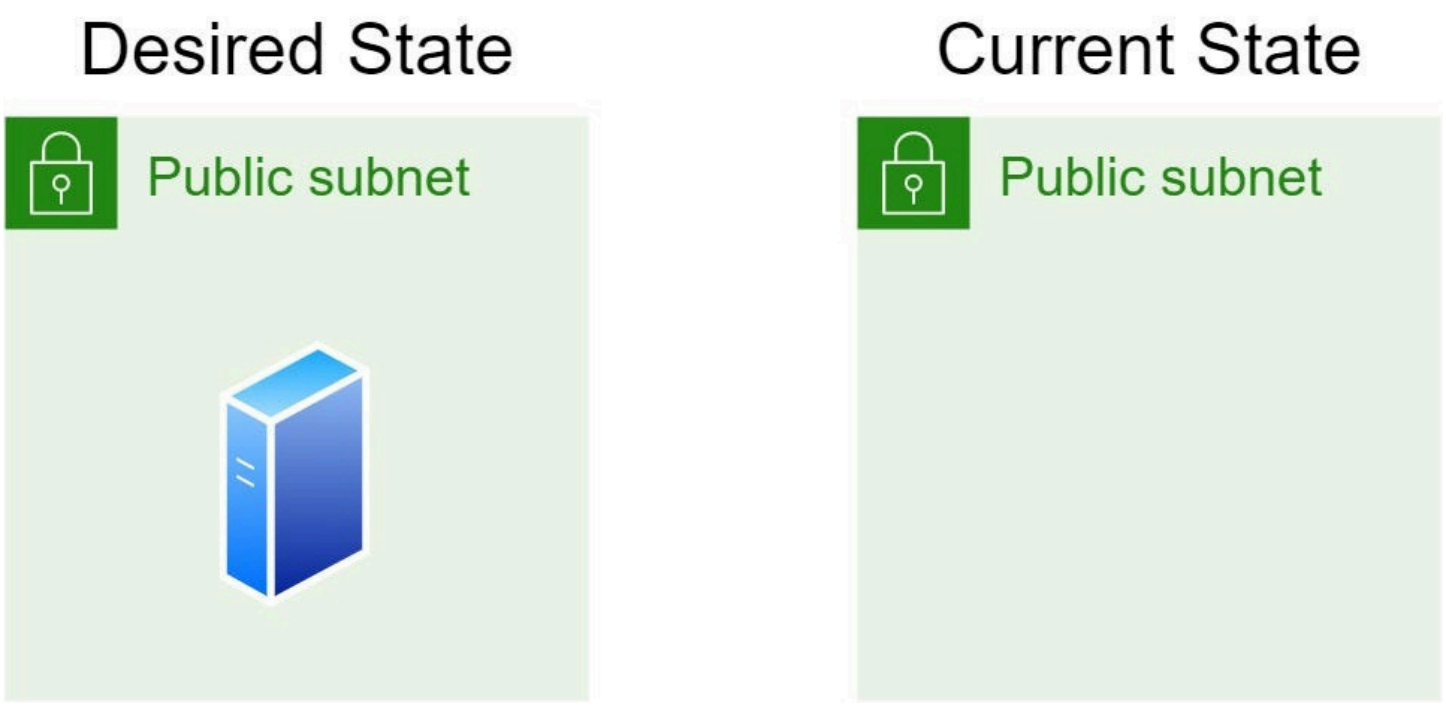


Terraform:

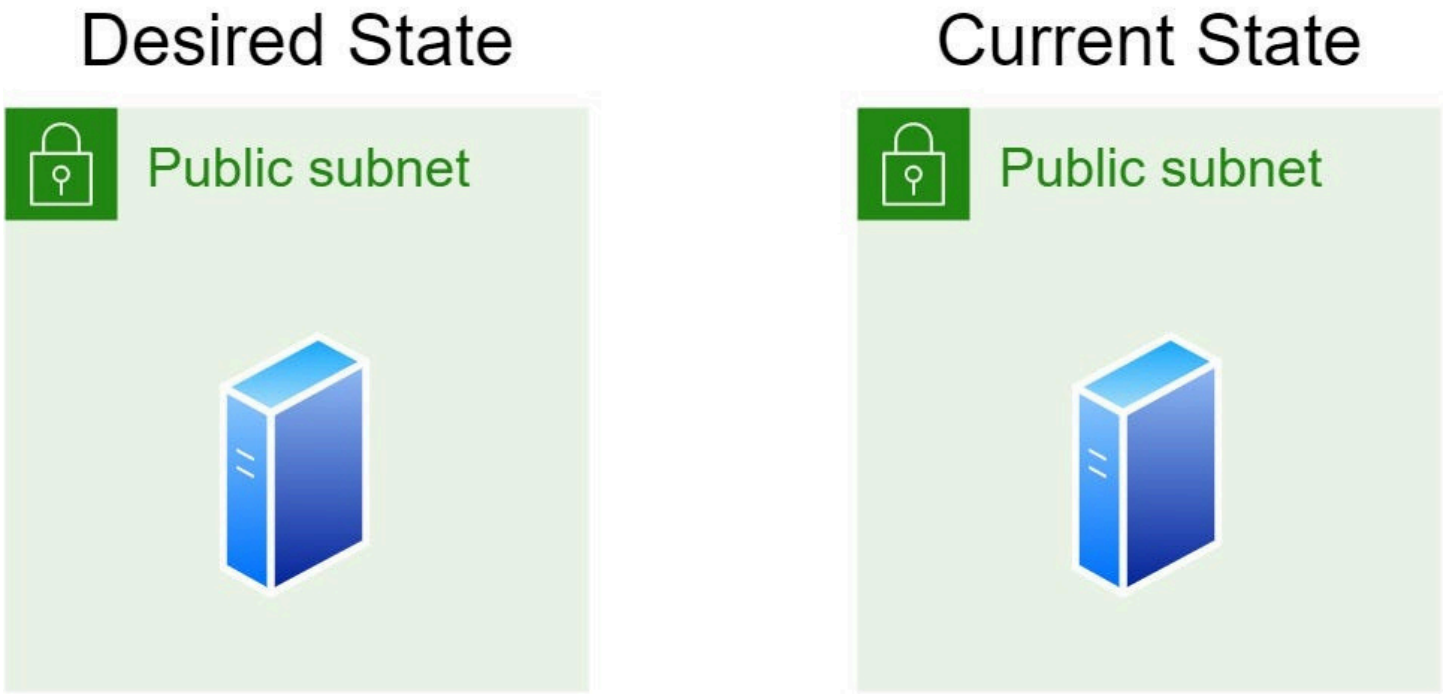
- HashiCorp's Terraform supports both multi-cloud and on premises.
- It's an open-source tool with an Enterprise version that uses HashiCorp's own HashiCorp Configuration Language (HCL).
- By using this terraform, we don't need to learn multiple services and languages for each cloud. We can use HCL's for all Cloudss.
- Hashicorp provides documentation on Terraform for developers and their Terraform Registry can be accessed for both official and community templates
- Terraform has the ability to import existing resources to be tracked through code if they were created outside Terraform
- Terraform supports over 170 Official and Verified Providers to include AWS, Azure, Google Cloud, Oracle Cloud, Alibaba Cloud, Kubernetes, VMware, Docker, Databricks, GitHub, Gitlab, Splunk and Grafana to name a few.

Understanding Terraform State File

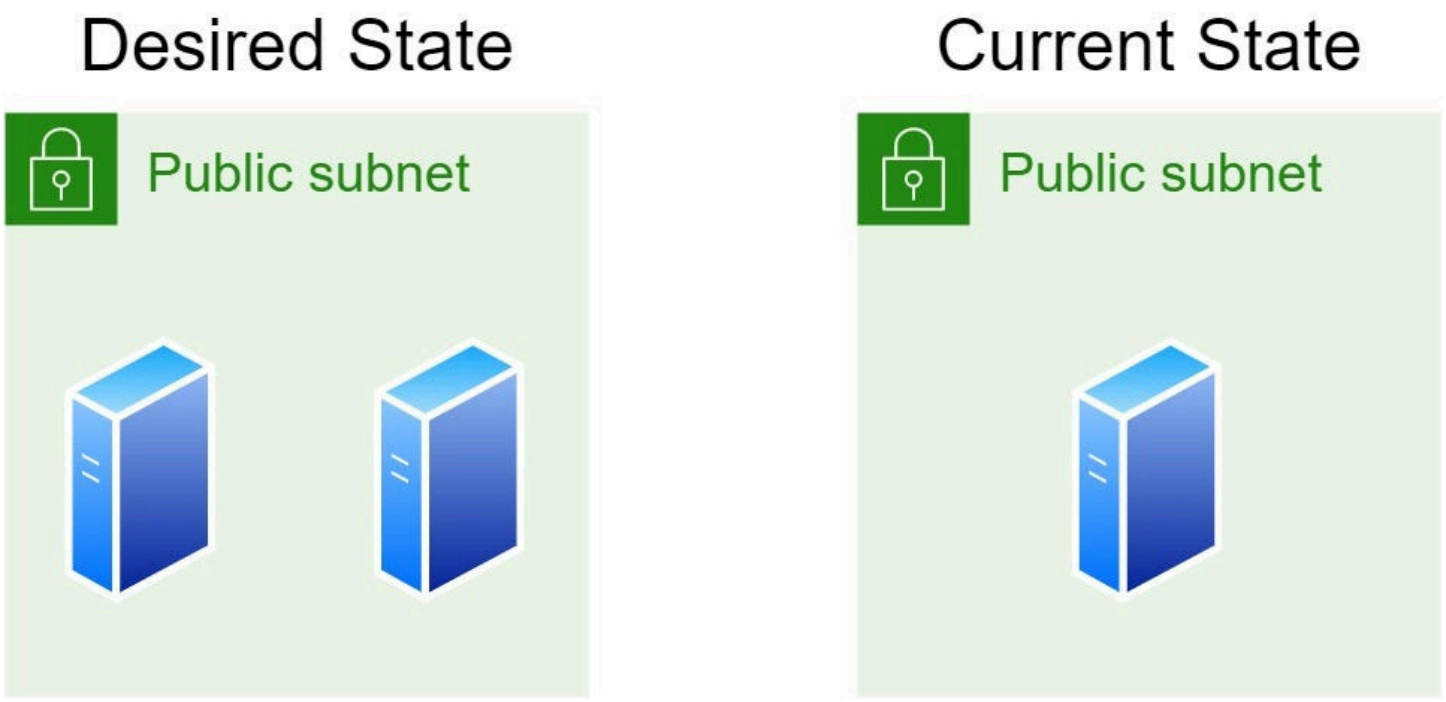
1. **What is the Terraform State File?**
   - The Terraform **State File** is like a map that tracks your infrastructure's current and desired states.
   - It ensures Terraform knows what resources exist and their configurations.
2. **How Does It Work?**
   - Terraform compares the **desired state** (defined in your code) with the **current state** (actual resources in your environment).
   - Based on the difference, Terraform takes actions like creating, updating, or deleting resources.
3. **Example in Action**:
   - **Scenario**: You write a Terraform file to create one virtual server.
   - **Current State**: No virtual servers exist in your environment.
   - **Terraform Action**: When you run the code, Terraform detects the difference and creates one virtual server to match the desired state.
4. **Why is the State File Important?**
   - It ensures **accuracy** by keeping track of what's already been deployed.
   - Prevents duplicate or unnecessary resource creation.
   - Helps manage infrastructure changes safely and effectively.
5. **Real-World Example**:
   - A team managing a multi-region application used Terraform to track their infrastructure. When they updated their Terraform file to add a new load balancer, Terraform only added the new resource without affecting the existing setup, thanks to the State File.



If run again with making any changes, the Desired State would match the Current State and no action would be taken.



If the user was to update the Terraform file again to create two virtual servers and then run the Terraform file, Terraform would see that now the Desired State and Current State no longer match and add one virtual server.



Terraform Workflow

Each Terraform project will follow the Terraform Workflow consisting of Write, Plan, Apply.

Write

Write the IaC in HCL using blocks, arguments, variables, and expressions.

Plan

Run terraform plan to get a preview of the modifications that will be made to the environment. The plan output will notate what will be added, destroyed, and changed.

Apply

If satisfied with the plan output, run terraform apply to apply the desired changes to reach your desired state.

Tearing Down Environments

To delete a specific resource from your environment, simply remove it from the Terraform configuration file. When you run terraform apply, Terraform will compare the current state with the updated desired state and automatically delete the resource that’s no longer defined in the file.

If you want to completely remove all resources in an environment, you can use the terraform destroy command. This command will delete every resource managed by Terraform in that environment, effectively tearing it down entirely.

This approach ensures clean and consistent removal of resources, saving time and avoiding manual effort.

Conclusion

Terraform revolutionizes infrastructure management by bringing automation, consistency, and scalability to the cloud and on-premises environments. By using its state file, HCL language, and support for hundreds of providers, it simplifies complex tasks like provisioning, updating, and tearing down resources. Whether you're managing a single region or multi-cloud deployments, Terraform empowers developers to work efficiently and avoid costly manual errors.

With Terraform, organizations can focus more on innovation and less on repetitive tasks, making it a critical tool for modern DevOps practices. Embracing Terraform is not just a step towards automation—it's a leap towards building reliable, future-ready infrastructure.

LAUNCH AN INSTANCE WITH TERRAFORM

```
provider "aws" {
 region = "us-east-1"
 access_key = "AKIASE5KRCBQADV22P5T"
 secret_key = "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"
}
```

```
resource "aws_instance" "myserver" {
 tags = {
 Name = "flm-server"
 Environment = "Dev"
 Project = "Zomato"
 }
 ami = "ami-0dc3a08bd93f84a35"
 instance_type = "t2.micro"
 key_name = "helm"
 availability_zone = "us-east-1b"
 root_block_device {
 volume_size = 10
 }
 count = 5
}
```