**Terraform AWS Provider: Lab Guide for Resource Provisioning and Upgrades**

**Objective**

In this lab, you will:

1. Understand the Terraform configuration for AWS resources.
2. Plan, test, and validate the upgrade of the AWS provider from version 5.76.0 to 5.77.0.
3. Provision AWS resources using Terraform, such as:
   * EC2 Instances with Elastic IP
   * S3 Bucket
   * Configure security group and outputs
4. Export outputs for integration with external tools (e.g., CI/CD pipelines or applications).

**Lab Prerequisites**

1. AWS account with the following:
   * IAM user credentials: **Access Key** and **Secret Key**.
2. Terraform installed (v1.0+).
3. AWS CLI configured with necessary permissions.
4. Network setup: Ensure proper VPC, Subnet, and Security Group setup.
5. Access to GitHub Release Page:
   * [Terraform AWS Provider Releases](https://github.com/hashicorp/terraform-provider-aws/releases)

**Step 1: Review Current Provider and Plan the Upgrade**

1. **Current AWS Provider Version:**
   * Version currently used: 5.76.0.
   * Your configuration:

hcl

Copy code

provider "aws" {

region = "us-east-1"

access\_key = "<Access Key>"

secret\_key = "<Secret Key>"

}

1. **Plan for Upgrade:**
   * Visit the [AWS Provider Release Notes](https://github.com/hashicorp/terraform-provider-aws/releases).
   * Identify changes introduced in version 5.77.0:
     + Fixes/Features/Enhancements.
     + Verify no breaking changes impact your existing resources.
2. **Upgrade Procedure:**
   * Replace version 5.76.0 with 5.77.0.
   * Test in **QA Environment**, then **UAT** before moving to Production.

**Step 2: Configure Terraform Resources**

1. **Create Terraform Configuration File (first.tf):**

Use the following configuration:

hcl

Copy code

# AWS Provider Configuration

provider "aws" {

region = "us-east-1"

# version = "5.77.0" # Set provider version after upgrade

access\_key = "AKIAZ7FSO3B5YFZTT5QT"

secret\_key = "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"

}

# Create an S3 Bucket

resource "aws\_s3\_bucket" "buck" {

bucket = "my-tf-test-buckettttttttttttttttttttttttttttt"

tags = {

Name = "My bucket"

Environment = "Dev"

}

}

# Launch an EC2 Instance

resource "aws\_instance" "ec2" {

depends\_on = [aws\_s3\_bucket.buck]

ami = "ami-0e2c8caa4b6378d8c" # Change to valid AMI in your region

instance\_type = "t2.micro"

vpc\_security\_group\_ids = ["sg-0e8552c15babe88bc"] # Adjust SG as needed

tags = {

Name = "raman-first-server"

}

}

# Allocate Elastic IP for the EC2 instance

resource "aws\_eip" "lb" {

instance = aws\_instance.ec2.id

domain = "vpc"

}

# Manage the State of the EC2 Instance

resource "aws\_ec2\_instance\_state" "ec22" {

instance\_id = aws\_instance.ec2.id

state = "running"

}

# Outputs for External Use

output "vm\_details" {

value = {

instance\_type = aws\_instance.ec2.instance\_type

private\_dns = aws\_instance.ec2.private\_dns

state = aws\_instance.ec2.instance\_state

publicip = aws\_instance.ec2.public\_ip

}

description = "Details of the EC2 instance."

}

output "elasticIp" {

value = aws\_eip.lb.public\_ip

}

output "s3details" {

value = aws\_s3\_bucket.buck.arn

}

**Step 3: Apply Configuration**

**1. Initialize Terraform**

bash

Copy code

terraform init

* This command downloads the AWS provider plugins.

**2. Validate Terraform Configuration**

bash

Copy code

terraform validate

* Ensures the syntax and resources are correctly defined.

**3. Preview the Changes**

bash

Copy code

terraform plan

* Review planned resources:
  + EC2 instance with specified AMI.
  + Elastic IP.
  + S3 Bucket.

**4. Apply Changes**

bash

Copy code

terraform apply -auto-approve

**Step 4: Verify Resources**

1. **Retrieve Outputs:**

bash

Copy code

terraform output

1. Example Output:

json

Copy code

vm\_details = {

"instance\_type" = "t2.micro"

"private\_dns" = "ip-172-31-91-158.ec2.internal"

"publicip" = "18.232.150.23"

"state" = "running"

}

elasticIp = "18.232.150.23"

s3details = "arn:aws:s3:::my-tf-test-buckettttttttttttttttttttttttttttt"

1. **Save Outputs to a File for Integration:**

bash

Copy code

terraform output > extapp.json

Verify the file:

bash

Copy code

cat extapp.json

**Step 5: Destroy Resources (Optional)**

To clean up the resources created:

bash

Copy code

terraform destroy -auto-approve

**Step 6: Upgrade Provider**

1. Update the AWS provider version in the configuration:

hcl

Copy code

provider "aws" {

version = "~> 5.77.0"

}

1. **Re-run Initialization:**

bash

Copy code

terraform init -upgrade

1. Validate, plan, and apply:

bash

Copy code

terraform validate

terraform plan

terraform apply -auto-approve

1. Verify outputs after the upgrade:

bash

Copy code

terraform output

**Step 7: Test in Environments**

1. Perform the upgrade in:
   * **QA Environment:** Validate changes.
   * **UAT Environment:** Confirm no impact on functionality.
2. Deploy to Production after validation.

**Step 8: Create CAB Ticket for Production Upgrade**

Provide the following information in the ticket:

* Purpose: Upgrade AWS provider from 5.76.0 to 5.77.0.
* Resources Impacted:
  + 10 EC2 Instances, 5 Firewalls, 3 Databases, 2 Lambda Functions, and an S3 Bucket.
* Testing Steps:
  + Tested in QA and UAT with no issues.
* Maintenance Window:
  + Schedule based on criticality.

**Summary**

This guide provided step-by-step instructions to:

1. Configure Terraform to provision AWS resources.
2. Manage outputs for external integrations.
3. Safely upgrade the AWS provider version and test changes.

You can now use these Terraform configurations to deploy and upgrade AWS resources in any environment efficiently.

**Lab Guide: Centralizing Values Using Variables in Terraform**

**Objective**

This lab covers:

1. Introduction to **Terraform variables** and their usage.
2. Refactoring code to use variables instead of hardcoded values.
3. Overriding variables with:
   * Command-line -var.
   * \*.tfvars files like prod.tfvars or uat.tfvars.
4. Practical implementation with:
   * **AWS resources** (S3 bucket, EC2 instance, Elastic IP).
   * **Security Group with dynamic variables**.

**Prerequisites**

1. AWS account with necessary permissions.
2. Terraform installed on your system.
3. AWS CLI configured:

bash

Copy code

aws configure

# Or export AWS credentials

export AWS\_ACCESS\_KEY\_ID="Your\_Access\_Key"

export AWS\_SECRET\_ACCESS\_KEY="Your\_Secret\_Key"

export AWS\_REGION="us-east-1"

**Step 1: Define Terraform Variables**

1. **Create variable.tf**: Centralize all variable definitions.

hcl

Copy code

# variable.tf

# Common CIDR block for firewall rules

variable "cidr" {

description = "CIDR block for ingress rules"

default = "0.0.0.0/0" # Can be overridden for specific environments

}

# Protocol for security group

variable "protocol" {

description = "Protocol for security group rules"

default = "tcp" # Default to TCP

}

# Instance Type Variable

variable "instance\_type" {

description = "EC2 instance type"

default = "t2.micro"

}

# S3 Bucket Name Variable

variable "s3\_bucket\_name" {

description = "S3 bucket name"

default = "my-terraform-bucket-demo"

}

**Step 2: Use Variables in Terraform Configuration**

**1. security.tf - Security Group Configuration**

Replace hardcoded values with variable references:

hcl

Copy code

# security.tf

resource "aws\_security\_group" "ec3" {

name = "raman-variables-demo"

# Ingress Rules Using Variables

ingress {

from\_port = 443

to\_port = 443

protocol = var.protocol

cidr\_blocks = [var.cidr]

}

ingress {

from\_port = 80

to\_port = 80

protocol = var.protocol

cidr\_blocks = [var.cidr]

}

ingress {

from\_port = 22

to\_port = 22

protocol = var.protocol

cidr\_blocks = [var.cidr]

}

}

**2. first.tf - AWS Resources with Variables**

Define S3, EC2, and Elastic IP resources using variables:

hcl

Copy code

# first.tf

# AWS Provider

provider "aws" {

region = "us-east-1"

}

# S3 Bucket

resource "aws\_s3\_bucket" "demo\_bucket" {

bucket = var.s3\_bucket\_name

tags = {

Name = "Terraform Demo Bucket"

Environment = "Dev"

}

}

# EC2 Instance

resource "aws\_instance" "example" {

ami = "ami-0e2c8caa4b6378d8c" # Replace with a valid AMI

instance\_type = var.instance\_type

tags = {

Name = "Terraform Example Instance"

}

}

# Elastic IP

resource "aws\_eip" "example\_eip" {

instance = aws\_instance.example.id

domain = "vpc"

}

**Step 3: Default Variable Values**

Default values are defined in variable.tf. You can override these in multiple ways during deployment.

**Step 4: Using Custom Variable Files**

**Create prod.tfvars**

Define production-specific values:

hcl

Copy code

# prod.tfvars

cidr = "192.168.1.0/24"

protocol = "udp"

instance\_type = "t3.medium"

s3\_bucket\_name = "prod-terraform-bucket-demo"

**Create uat.tfvars**

For UAT:

hcl

Copy code

# uat.tfvars

cidr = "10.0.0.0/16"

protocol = "tcp"

instance\_type = "t2.large"

s3\_bucket\_name = "uat-terraform-bucket-demo"

**Step 5: Running Terraform**

1. **Initialize the Terraform directory**:

bash

Copy code

terraform init

1. **Override Variable on the Fly**:

Use -var to override default values:

bash

Copy code

terraform plan -var="protocol=udp" -var="cidr=192.168.1.0/24"

1. **Use Custom Variable Files**:

Example with production values:

bash

Copy code

terraform plan -var-file="prod.tfvars"

Example with UAT values:

bash

Copy code

terraform plan -var-file="uat.tfvars"

1. **Apply the Terraform Configuration**:
   * With **default values**:

bash

Copy code

terraform apply -auto-approve

* + With a specific tfvars file:

bash

Copy code

terraform apply -var-file="prod.tfvars" -auto-approve

**Step 6: Verify Outputs**

Create outputs for verification and external use.

**Add Outputs to outputs.tf:**

hcl

Copy code

output "instance\_id" {

value = aws\_instance.example.id

}

output "s3\_bucket" {

value = aws\_s3\_bucket.demo\_bucket.arn

}

output "security\_group\_name" {

value = aws\_security\_group.ec3.name

}

output "elastic\_ip" {

value = aws\_eip.example\_eip.public\_ip

}

**Retrieve Outputs:**

bash

Copy code

terraform output

* Example Output:

json

Copy code

instance\_id = "i-07eb34bd36598f2ed"

s3\_bucket = "arn:aws:s3:::prod-terraform-bucket-demo"

security\_group\_name = "raman-variables-demo"

elastic\_ip = "54.172.26.195"

**Step 7: Clean Up**

Destroy all created resources:

bash

Copy code

terraform destroy -auto-approve

For specific environment variables:

bash

Copy code

terraform destroy -var-file="uat.tfvars" -auto-approve

**Key Notes**

1. Variables are flexible and allow overriding:
   * On the fly: -var="key=value".
   * Using files: -var-file.
2. Use environment-specific .tfvars files for better organization (e.g., prod.tfvars).

**Conclusion**

This lab demonstrated:

* How to centralize variable values for better maintainability.
* Override values dynamically with command-line flags or .tfvars files.
* Manage AWS resources like **EC2**, **S3**, and **Security Groups** using Terraform.

You can now efficiently manage environment configurations (Dev, UAT, Prod) in Terraform without duplicating code!

4o

**Lab Guide: Managing Terraform Variables and Isolating Sensitive Data**

**Objective**

This lab focuses on:

1. Defining **variables** for resources like EC2, Security Groups, and S3 Buckets.
2. Isolating sensitive information (e.g., AWS credentials) using a terraform.tfvars file.
3. Creating a structured and modular Terraform configuration:
   * EC2 instance with an Elastic IP.
   * Security Groups with dynamic values for ingress rules.
   * S3 bucket resource.
4. Running environment-specific configurations using \*.tfvars files.

**Prerequisites**

1. **Tools**:
   * Terraform installed.
   * AWS CLI configured locally.
2. **Access**:
   * AWS credentials (Access Key, Secret Access Key).
3. **Directory Structure**:
   * Create a working directory and initialize Terraform.

**Step 1: Directory and Files Setup**

Organize all Terraform configuration files as follows:

plaintext

Copy code

project-directory/

│── first.tf # EC2, EIP, S3 resources

│── security.tf # Security Group definitions

│── variable.tf # Variables definition

│── terraform.tfvars # AWS credentials (secure)

│── prod.tfvars # Production-specific variable values

│── uat.tfvars # UAT environment-specific variable values

**Step 2: Define Variables (variable.tf)**

This file centralizes **variables** for credentials and AWS resource configurations:

hcl

Copy code

# variable.tf

# Ingress CIDR block

variable "rk" {

description = "CIDR block for ingress rules"

default = "192.168.1.0/24"

}

# Protocol for security rules

variable "kr" {

description = "Protocol for security group"

default = "udp" # Default protocol can be overridden

}

# Instance type

variable "ak" {

description = "EC2 instance type"

default = "t2.micro"

}

# AWS credentials

variable "access\_key" {

description = "AWS Access Key"

}

variable "secret\_access\_key" {

description = "AWS Secret Access Key"

}

**Step 3: Store Sensitive Data in terraform.tfvars**

Store sensitive data such as credentials in terraform.tfvars (Terraform automatically loads this file):

plaintext

Copy code

# terraform.tfvars

access\_key = "AKIAZ7FSO3B5YFZTT5QT"

secret\_access\_key = "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"

**Step 4: Create Environment-Specific Variable Files**

Define environment-specific values:

**Production (prod.tfvars):**

hcl

Copy code

# prod.tfvars

rk = "0.0.0.0/0" # Allow access from anywhere

kr = "tcp" # Use TCP protocol

ak = "t2.large" # Larger EC2 instance

**UAT (uat.tfvars):**

hcl

Copy code

# uat.tfvars

rk = "10.0.0.0/16"

kr = "udp"

ak = "t2.medium"

**Step 5: Configure AWS Provider (security.tf)**

hcl

Copy code

# security.tf

provider "aws" {

region = "us-east-1"

access\_key = var.access\_key

secret\_key = var.secret\_access\_key

}

# Security Group Definition

resource "aws\_security\_group" "ec3" {

name = "raman-variables-demo"

ingress {

from\_port = 443

to\_port = 443

protocol = var.kr

cidr\_blocks = [var.rk]

}

ingress {

from\_port = 22

to\_port = 22

protocol = var.kr

cidr\_blocks = [var.rk]

}

ingress {

from\_port = 80

to\_port = 80

protocol = var.kr

cidr\_blocks = [var.rk]

}

}

**Step 6: Define AWS Resources (first.tf)**

Create an EC2 instance, assign an Elastic IP, and define an S3 bucket.

hcl

Copy code

# first.tf

# EC2 Instance

resource "aws\_instance" "ec2" {

depends\_on = [aws\_s3\_bucket.buck]

ami = "ami-0e2c8caa4b6378d8c" # Valid AMI ID

instance\_type = var.ak

vpc\_security\_group\_ids = [aws\_security\_group.ec3.id]

tags = {

Name = "raman-first-server"

}

}

# S3 Bucket

resource "aws\_s3\_bucket" "buck" {

bucket = "my-tf-test-bucket-unique-id"

tags = {

Name = "My bucket"

Environment = "Dev"

}

}

# Elastic IP for EC2 instance

resource "aws\_eip" "lb" {

instance = aws\_instance.ec2.id

domain = "vpc"

}

# Outputs for Verification

output "vm\_details" {

value = {

instance\_type = aws\_instance.ec2.instance\_type

private\_dns = aws\_instance.ec2.private\_dns

public\_ip = aws\_instance.ec2.public\_ip

}

}

output "elasticIp" {

value = aws\_eip.lb.public\_ip

}

output "s3details" {

value = aws\_s3\_bucket.buck.arn

}

**Step 7: Initialize and Plan the Terraform Configuration**

1. **Initialize Terraform**:

bash

Copy code

terraform init

1. **Plan Using Default terraform.tfvars**:

bash

Copy code

terraform plan

1. **Plan Using Production Configurations**:

bash

Copy code

terraform plan -var-file="prod.tfvars"

1. **Plan Using UAT Configurations**:

bash

Copy code

terraform plan -var-file="uat.tfvars"

**Step 8: Apply and Verify**

1. Apply the default configuration:

bash

Copy code

terraform apply -auto-approve

1. Apply with the prod.tfvars configuration:

bash

Copy code

terraform apply -var-file="prod.tfvars" -auto-approve

1. **Verify Outputs**: Retrieve outputs:

bash

Copy code

terraform output

Example Output:

plaintext

Copy code

vm\_details = {

"instance\_type" = "t2.large"

"private\_dns" = "ip-10-0-0-10.ec2.internal"

"public\_ip" = "54.172.26.195"

}

elasticIp = "54.172.26.195"

s3details = "arn:aws:s3:::my-tf-test-bucket-unique-id"

**Step 9: Destroy Resources**

To clean up AWS resources:

bash

Copy code

terraform destroy -var-file="prod.tfvars" -auto-approve

**Key Notes**

* **Variables**:
  + Centralized in variable.tf for better code reusability.
  + Overridden dynamically via terraform.tfvars or -var-file.
* **Sensitive Data**:
  + Credentials are safely stored in terraform.tfvars, and this file should be added to .gitignore.
* **Modularization**:
  + Security Group, EC2, and S3 are separated into logical files for better management.

**Conclusion**

You have successfully:

1. Configured Terraform with variables and sensitive data isolation.
2. Created a scalable setup for **production** and **UAT** environments using prod.tfvars and uat.tfvars.
3. Provisioned AWS resources like Security Groups, EC2 instances, and S3 buckets with flexible configuration.

You are now equipped to manage multi-environment Terraform configurations securely and efficiently!

**Lab Guide: Using Terraform Data Sources to Fetch Dynamic Information**

**Objective**

This lab focuses on **Terraform Data Sources**, which allow you to fetch information about existing infrastructure or discover resource properties dynamically. You will:

1. Use aws\_instance as a data source to fetch attributes of an existing EC2 instance.
2. Use aws\_ami as a data source to discover the latest Amazon Linux 2 AMI for a specific AWS region.
3. Create new infrastructure leveraging information retrieved from the data sources.

**Prerequisites**

1. **Tools**:
   * Terraform installed.
   * AWS CLI installed and configured.
2. **Access**:
   * AWS credentials: Access Key ID and Secret Access Key.
   * Permissions to use AWS resources like **EC2** and **AMI**.
3. **AWS Existing Infrastructure**:
   * At least one existing EC2 instance (with a known instance\_id).

**Understanding Data Sources**

Data sources in Terraform allow you to **read** existing resources in your AWS account. You can use this fetched information dynamically to configure new infrastructure without hardcoding values.

Key benefits:

* **No manual input**: Retrieve IDs, attributes, or properties automatically.
* **Dynamic setups**: Build infrastructure based on existing data (e.g., AMIs, EC2s, VPCs).

**Lab 1: Fetch Attributes from an Existing EC2 Instance**

**Step 1: File Structure**

plaintext

Copy code

data-source-lab/

│── main.tf # Terraform configuration

│── variables.tf # Optional variable definitions

│── outputs.tf # Outputs to display fetched data

**Step 2: Terraform Configuration (main.tf)**

This code fetches an existing EC2 instance's details and uses its attributes to provision a new EC2 instance.

hcl

Copy code

# Provider Configuration

provider "aws" {

region = "us-east-1"

access\_key = "AKIAZ7FSO3B5YFZTT5QT"

secret\_key = "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"

}

# Data Source: Fetch an Existing EC2 Instance

data "aws\_instance" "existing\_instance" {

instance\_id = "i-0643c858fb3e16db8" # Replace with the actual EC2 instance ID

}

# Resource: Create a New EC2 Instance Using Fetched Data

resource "aws\_instance" "new\_instance" {

ami = data.aws\_instance.existing\_instance.ami

instance\_type = data.aws\_instance.existing\_instance.instance\_type

tags = {

Name = "new-ec2-instance"

}

}

# Output: Show Existing EC2 Instance Details

output "existing\_instance\_details" {

value = {

ami = data.aws\_instance.existing\_instance.ami

instance\_type = data.aws\_instance.existing\_instance.instance\_type

public\_ip = data.aws\_instance.existing\_instance.public\_ip

private\_ip = data.aws\_instance.existing\_instance.private\_ip

}

}

**Step 3: Initialize Terraform**

Run the following commands:

1. **Initialize the working directory**:

bash

Copy code

terraform init

1. **Validate the configuration**:

bash

Copy code

terraform validate

**Step 4: Plan and Apply**

1. Generate an execution plan:

bash

Copy code

terraform plan

Verify the plan summary to confirm that Terraform retrieves data from the existing EC2 instance and uses it to create a new EC2 instance.

1. Apply the changes:

bash

Copy code

terraform apply -auto-approve

**Step 5: Verify Outputs**

The terraform output command will show the attributes of the **existing EC2 instance**:

bash

Copy code

terraform output

Example output:

plaintext

Copy code

existing\_instance\_details = {

"ami" = "ami-0abcdef1234567890"

"instance\_type" = "t2.micro"

"public\_ip" = "54.174.35.125"

"private\_ip" = "172.31.1.2"

}

**Lab 2: Use Data Source to Fetch the Latest Amazon Linux 2 AMI**

**Step 1: Configuration for Latest AMI Lookup (main.tf)**

Fetch the **most recent Amazon Linux 2 AMI** dynamically and use it to launch a new EC2 instance.

hcl

Copy code

# Provider Configuration

provider "aws" {

region = "us-west-1"

access\_key = "AKIAZ7FSO3B5YFZTT5QT"

secret\_key = "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"

}

# Data Source: Fetch Latest Amazon Linux 2 AMI

data "aws\_ami" "latest\_amzn\_linux" {

most\_recent = true

owners = ["amazon"]

filter {

name = "name"

values = ["amzn2-ami-hvm\*"] # Pattern for Amazon Linux 2 AMIs

}

}

# Resource: Launch a New EC2 Instance Using the Latest AMI

resource "aws\_instance" "new\_amzn\_linux" {

ami = data.aws\_ami.latest\_amzn\_linux.id

instance\_type = "t2.micro"

tags = {

Name = "latest-amzn-linux-instance"

}

}

# Output: Display AMI ID Used

output "latest\_ami\_details" {

value = {

ami\_id = data.aws\_ami.latest\_amzn\_linux.id

ami\_name = data.aws\_ami.latest\_amzn\_linux.name

}

}

**Step 2: Initialize Terraform**

Initialize and validate the Terraform directory:

bash

Copy code

terraform init

terraform validate

**Step 3: Plan and Apply**

1. Create an execution plan:

bash

Copy code

terraform plan

1. Apply changes to launch a new instance:

bash

Copy code

terraform apply -auto-approve

**Step 4: Verify Outputs**

After deployment, fetch outputs using:

bash

Copy code

terraform output

Example output:

plaintext

Copy code

latest\_ami\_details = {

"ami\_id" = "ami-09b90e92e0c0405f3"

"ami\_name" = "amzn2-ami-hvm-2.0.20230503.0-x86\_64-gp2"

}

**Step 5: Clean Up Resources**

After verifying the outputs, destroy the resources to avoid unnecessary AWS charges:

bash

Copy code

terraform destroy -auto-approve

**Key Takeaways**

1. **Data Sources**:
   * Use aws\_instance to fetch attributes of existing EC2 instances.
   * Use aws\_ami to discover the latest AMIs dynamically.
2. **Dynamic Configurations**:
   * Avoid hardcoding IDs or AMI values. Fetch them dynamically using data sources.
   * Data sources help maintain consistency and reusability across Terraform configurations.
3. **Outputs**:
   * Always verify outputs to confirm fetched values before deployment.

**Conclusion**

This lab demonstrated how to use Terraform **data sources** to fetch existing EC2 attributes and discover AMIs dynamically. By the end of the lab, you successfully:

1. Retrieved details of an existing EC2 instance dynamically.
2. Deployed a new EC2 instance based on an existing instance's properties.
3. Dynamically retrieved the latest **Amazon Linux 2 AMI** and used it to provision a new instance.

You are now ready to use Terraform data sources to design more dynamic and flexible infrastructure