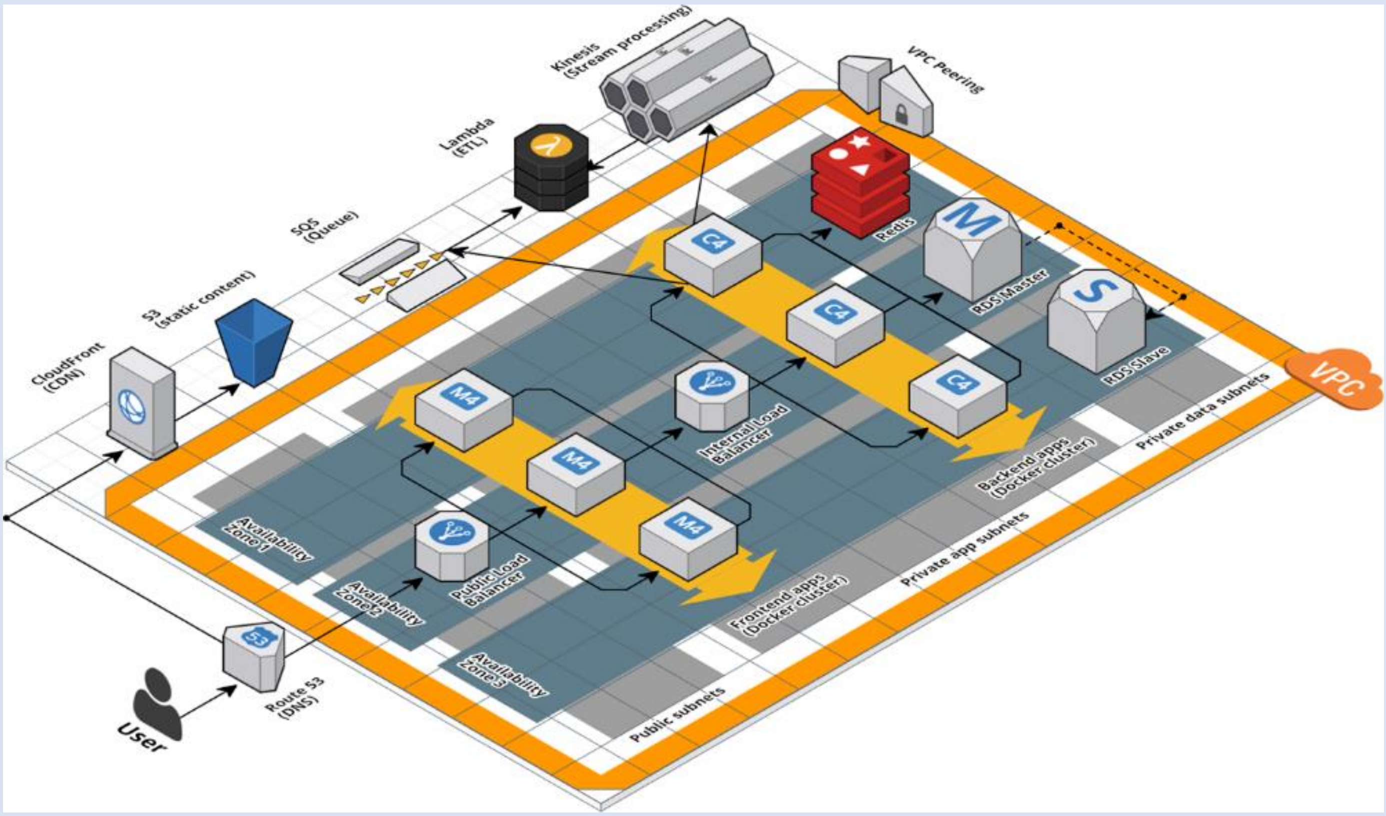
HyperAutomation using Terraform: Infrastructure as Code

DEI
DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Instituto Superior Técnico
Universidade de Lisboa
Integração Empresarial / Enterprise Integration

The goal of this document is to present the details related with Terraform. How to use it using a step-by-step approach. Terraform is a solution to create cloud environments using code instead of using the cloud providers User Interfaces. Therefore, the management of the cloud resources takes less effort and the creation, or update, process is faster.

The following figure taken the main literature reference in this field: *Terraform Up & Running, Writing Infrastructure as Code, Yevgeniy Brikman, 3rd edition (2022)"* exemplifies a complex infrastructure that could be created using Terraform.

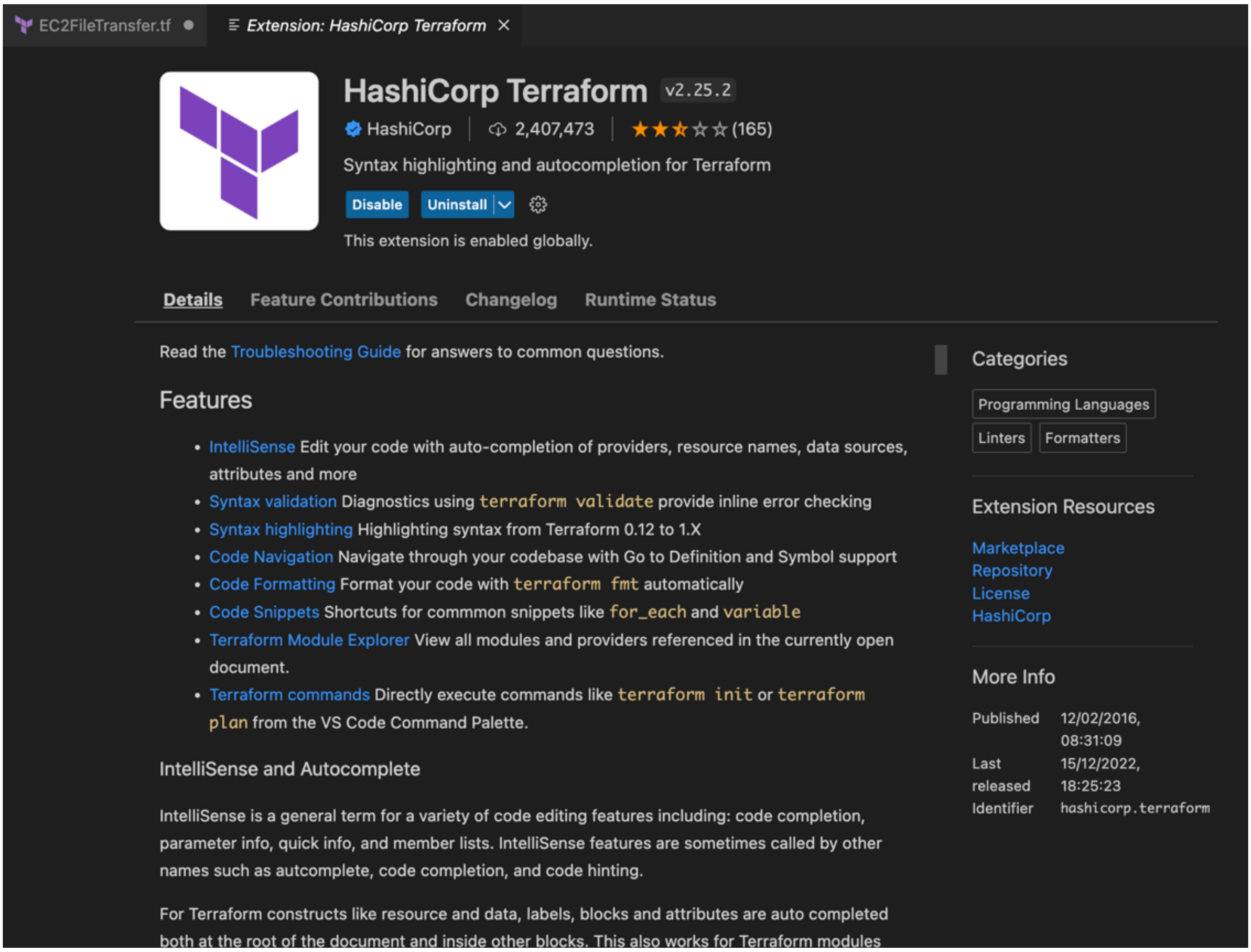


Contents

A. Terraform installation procedure

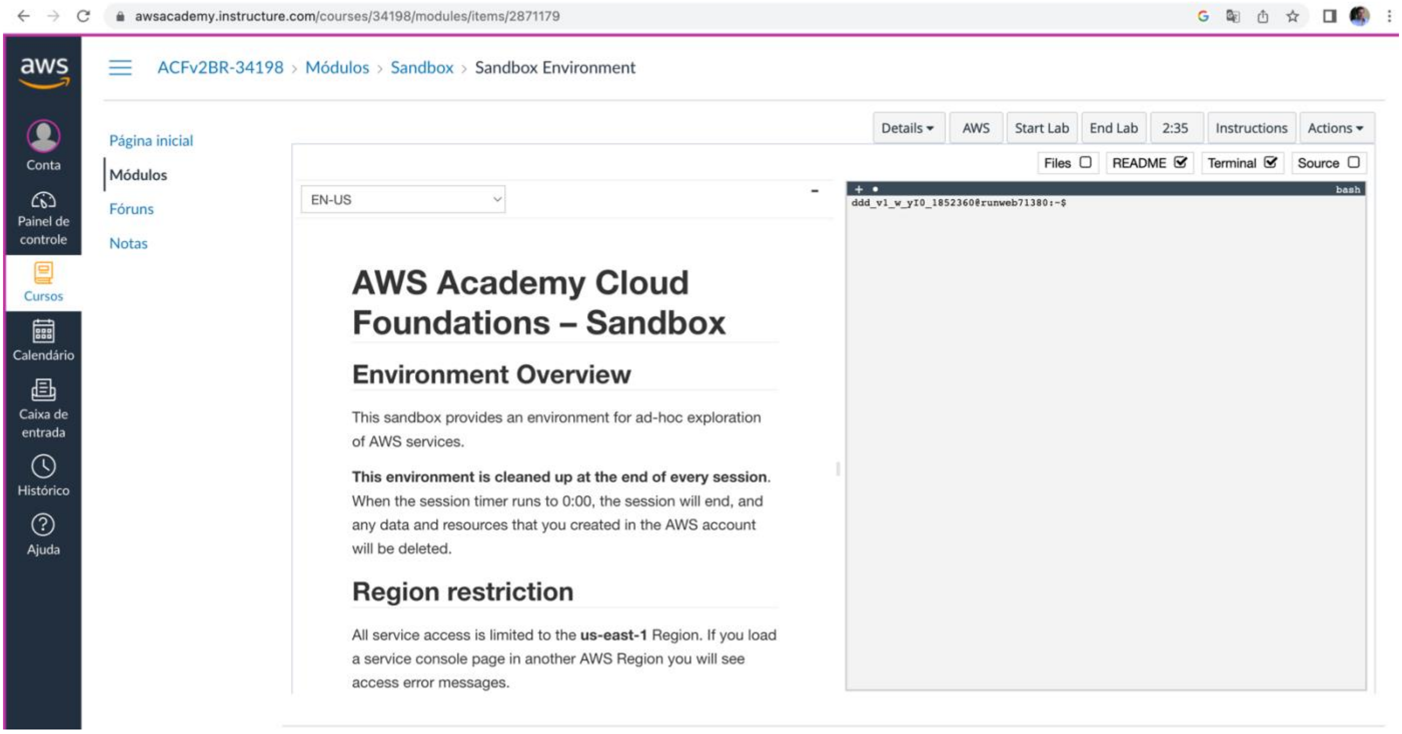
Use the URL https://developer.hashicorp.com/terraform/downloads and choose the package appropriate for your operating system and then follow the indications.

For better Terraform code rendering, you can optionally, install the Terraform extension for VSCode.



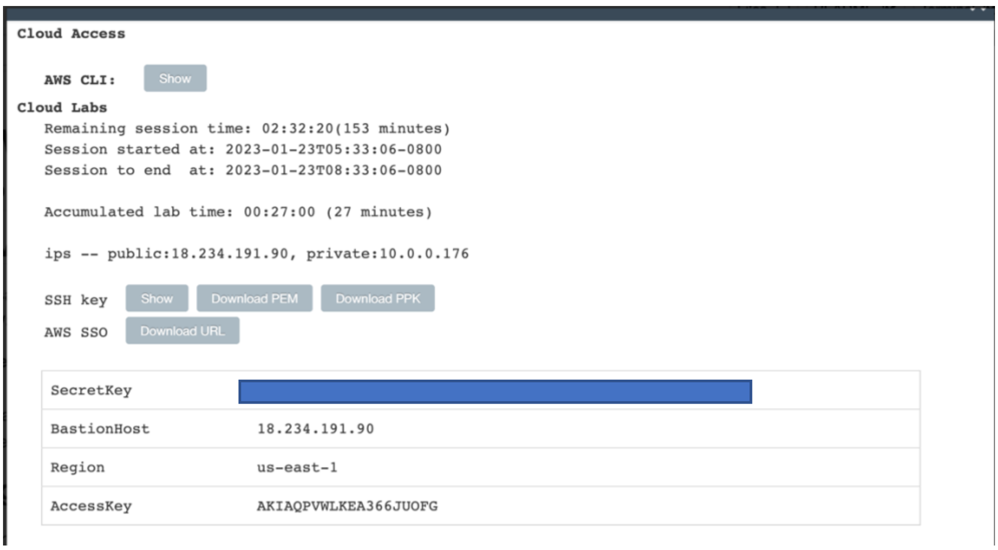
B. Obtaining the access for sandbox or AWS academy learner lab

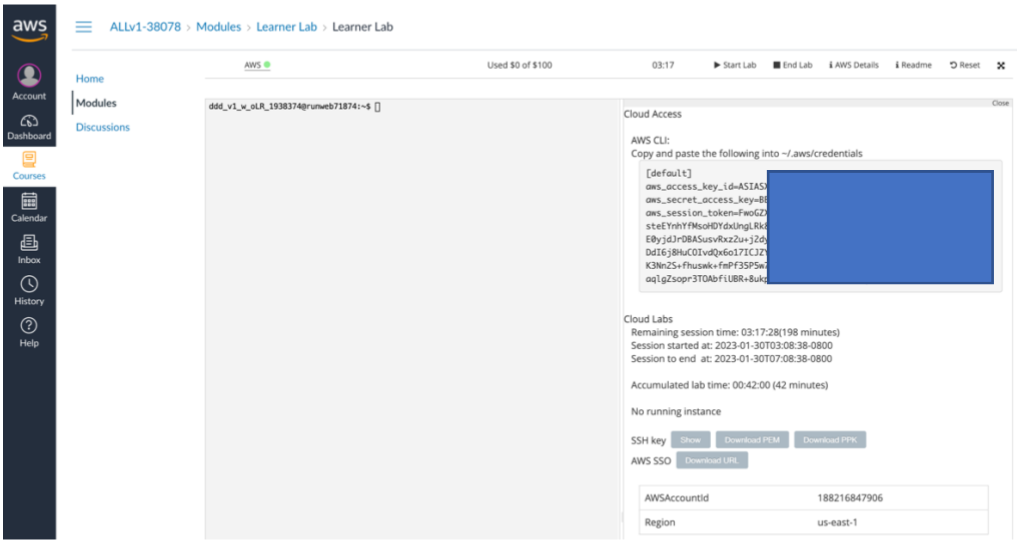
B.1. Choose your learner lab, then start it.



B.2. When the lab is started, select **Details**, and then **Show**:

B.3. A similar access configuration is available for you (they are both examples that could be available, the first one is secretKey/accessKey, and the second is secretKey/accessKey/token). You will have to adapt your terraform calls accordingly with the provided AWS access.




aws
Account
Dashboard
Courses
Calendar
Inbox
History
Help
ALLv1-38078 › Modules › Learner Lab › Learner Lab
Home
Modules
Discussions
AWS
Used $0 of $100
03:17
Start Lab
End Lab
AWS Details
Readme
Reset
Close
Cloud Access
AWS CLI:
Copy and paste the following into ~/.aws/credentials
[default]
Cloud Labs
Remaining session time: 03:17:28(198 minutes)
Session started at: 2023-01-30T03:08:38-0800
Session to end at: 2023-01-30T07:08:38-0800
Accumulated lab time: 00:42:00 (42 minutes)
No running instance
SSH key
Show
Download PEM
Download PPK
AWS SSO
Download URL
AWSAccountId
188216847906
Region
us-east-1

C. Deploying a single EC2 instance

C.1. Configure the AWS access inside the terraform script file. For that, create the following script, replacing the **access_key**, **secret_key**, **token**, **region**, the **ami**, and the **instance_type** accordingly with your needs. Save the file in text format with a **.tf** extension.

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
  access_key  = "YOUR_ACCESS_KEY"
  secret_key  = "YOUR_SECRET_KEY"
  token       = "YOUR_TOKEN"
}

resource "aws_instance" "example" {
  ami           = "ami-045269a1f5c90a6a0"
  instance_type = "t2.micro"

  tags = {
    Name = "terraform-example"
  }
}
```

C.2. Deploy the code with the following commands. You should use only one .tf file in each directory. A local configuration is created and then reflected in the cloud environment:

```
terraform init
terraform apply
```

or sequentially:

```
terraform init && terraform apply
```

C.3. You can verify the configuration of the current configuration using the following command:

```
terraform show
```

C.4. Clean up when you're done:

```
terraform destroy
```

Hint: to facilitate the terraform command you can also use the `-auto-approve` *option*

D. Deploying a single EC2 instance and uploading files

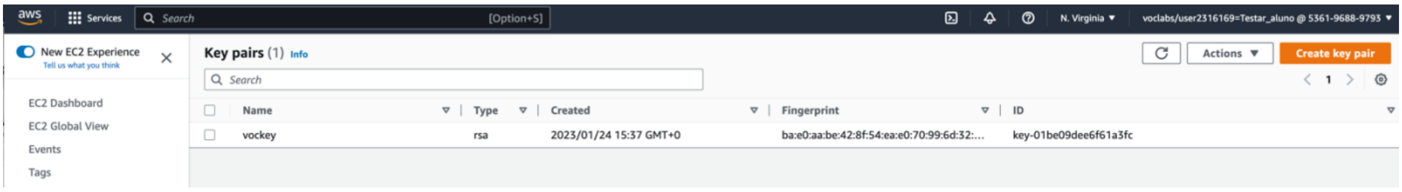
D.1. As previous, configure your AWS access keys as environment variables, and then, create the following .tf file, adapting the **region**, the **ami**, and the **instance_type** accordingly with you needs.

Notice that **access_key**, **secret_key** and **token** could be directly written in the source code.

Also get available the kafka tgz file and test.pem on your local path.

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region      = "us-east-1"
  access_key  = "YOUR_ACCESS_KEY"
  secret_key  = "YOUR_SECRET_KEY"
  token       = "YOUR_TOKEN"
}

resource "aws_instance" "exampleFileTransfer" {
  ami                         = "ami-045269a1f5c90a6a0"
  instance_type               = "t2.micro"
  vpc_security_group_ids      = [aws_security_group.instance.id]
  key_name                    = "vockey"
  tags = {
    Name = "terraform-example"
  }

  connection {
    type        = "ssh"
    user        = "ec2-user"
    private_key = file("test.pem")
    host        = "${self.public_dns}"
  }

  provisioner "file" {
      source      = "kafka_2.13-3.9.0.tgz"
      destination = "/home/ec2-user/kafka_2.13-3.9.0.tgz"
    }
}

resource "aws_security_group" "instance" {
  name = var.security_group_name
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

variable "security_group_name" {
  description = "The name of the security group"
  type        = string
  default     = "terraform-example-instance"
}
```

Where **key_name** is obtained from the name of the keypairs available directly from the AWS console:



The **provisioner "file"** tag can be repeated as many times as needed, e.g.:

```
provisioner "file" {
  source      = "kafka_2.13-3.6.2.tgz"
  destination = "/home/ec2-user/kafka_2.13-3.6.2.tgz"
}

provisioner "file" {
  source      = "/root/zookeeper/apache-zookeeper-3.8.4-bin.tar.gz"
  destination = "/home/ec2-user/apache-zookeeper-3.8.4-bin.tar.gz"
}
```

Take attention to the **security resource** that allows SSH connections to your instance. If any other ingress is needed, you can add more in the same .tf file.

Also, the connection provides the **connection** configuration to enable the file upload (it uses a scp mechanism).

Also take note that provisioner is last action taken.

D.2. After changing for your configuration. Try it. Deploy the code:

```
terraform init
terraform apply
```

or

```
terraform init && terraform apply
```

D.3. Check if the file is uploaded correctly using a SSH connection directly to the EC2 instance:

```
ssh -i test.pem ec2-user@YOUR_DNS_NAME
```

To remember: Use the following command to set the permissions of your private key file so that only you can read it.

```
chmod u=rwx,g=,o= myKeyAWS.pem
```

D.4. Clean up when you're done:

```
terraform destroy
```

E. Sending commands during boot of a single EC2 instance

User_data is a shell script or cloud-init directive that will be executed during **instance creation only**.

The result of the **user_data** execution can be seen in EC2 Console (select the Instance, click Actions -> Monitor and troubleshoot -> Get system log) and you can find its execution log on the EC2 Instance itself (typically in /var/log/cloud-init*.log), both of which are useful for debugging, and neither of which is available with provisioners.

E.1. As previous, configure your AWS access keys as environment variables, and then, create the following .tf file, adapting the **region**, the **ami**, and the **instance_type** accordingly with you needs.

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region      = "us-east-1"
  access_key  = "YOUR_ACCESS_KEY"
  secret_key  = "YOUR_SECRET_KEY"
  token       = "YOUR_TOKEN"
}

resource "aws_instance" "exampleKAFKA" {
  ami                     = "ami-045269a1f5c90a6a0"
  instance_type           = "t2.small"
  vpc_security_group_ids  = [aws_security_group.instance.id]
  key_name                = "vockey"

  user_data = "${file("creation.sh")}"

  user_data_replace_on_change = true

  tags = {
    Name = "terraform-example-kafka"
  }
}

resource "aws_security_group" "instance" {
  name = var.security_group_name
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
    ingress {
    from_port   = 2181
    to_port     = 2181
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
    ingress {
    from_port   = 9092
    to_port     = 9092
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
}
```

```
variable "security_group_name" {
  description = "The name of the security group"
  type        = string
  default     = "terraform-example-instance"
}
```

Where the script **user_data = "${file("creation.sh")}"** is a local file that could contain some work that you need to perform.
For instance, installing and starting ZooKeeper and Kafka, automatically!

```
#!/bin/bash
echo "Starting..."
cd
sudo wget https://dlcdn.apache.org/zookeeper/zookeeper-3.9.3/apache-zookeeper-3.9.3-bin.tar.gz
sudo tar -zxf apache-zookeeper-3.9.3-bin.tar.gz
sudo mv apache-zookeeper-3.9.3-bin /usr/local/zookeeper
sudo mkdir -p /var/lib/zookeeper
echo "tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181" > /usr/local/zookeeper/conf/zoo.cfg

sudo yum -y install java-17-amazon-corretto-devel.x86_64

sudo /usr/local/zookeeper/bin/zkServer.sh start

sudo wget https://downloads.apache.org/kafka/3.9.0/kafka_2.13-3.9.0.tgz
sudo tar -zxf kafka_2.13-3.9.0.tgz
sudo mv kafka_2.13-3.9.0 /usr/local/kafka
sudo mkdir /tmp/kafka-logs

sudo /usr/local/kafka/bin/kafka-server-start.sh -daemon /usr/local/kafka/config/server.properties

echo "Finished."
```

The **egress** tag allows the EC2 instance to outbound to another machine in internet.
Recall that a provisioner is the last action taken. That is why the binary files for Zookeeper and Kafka are being downloaded instead of being received by a provisioner.

If needed, the logging of the EC2 instance provisioning can be consulted at **/var/log/cloud-init-output.log**

F. Change the listener of a Kafka broker deployed with terraform

F.1. Use the previous terraform file for deploying kafka broker in AWS.

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region      = "us-east-1"
  access_key  = "YOUR_ACCESS_KEY"
  secret_key  = "YOUR_SECRET_KEY"
  token       = "YOUR_TOKEN"
}

resource "aws_instance" "exampleKAFKAConfigured" {
  ami                     = "ami-045269a1f5c90a6a0"
  instance_type           = "t2.small"
  vpc_security_group_ids  = [aws_security_group.instance.id]
  key_name                = "vockey"

  user_data = "${file("creation.sh")}"

  user_data_replace_on_change = true

  tags = {
    Name = "terraform-example-kafka2"
  }
}

resource "aws_security_group" "instance" {
  name = var.security_group_name
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
    ingress {
    from_port   = 2181
    to_port     = 2181
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
    ingress {
    from_port   = 9092
    to_port     = 9092
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
}

variable "security_group_name" {
  description = "The name of the security group"
  type        = string
  default     = "terraform-example-instance"
}
```

F.2. Then, the script for provisioning the kafka broker need to be adapted with the new configuration for the listener at file /usr/local/kafka/config/server.properties. For that the name of the EC2 instance need to be known. The instance metadata service available at https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html could be used for that purpose.

```
#!/bin/bash

echo "Starting..."
cd
sudo wget https://dlcdn.apache.org/zookeeper/zookeeper-3.9.3/apache-zookeeper-3.9.3-bin.tar.gz
sudo tar -zxf apache-zookeeper-3.9.3-bin.tar.gz
sudo mv apache-zookeeper-3.9.3-bin /usr/local/zookeeper
sudo mkdir -p /var/lib/zookeeper
echo "tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181" > /usr/local/zookeeper/conf/zoo.cfg

sudo yum -y install java-17-amazon-corretto-devel.x86_64

sudo /usr/local/zookeeper/bin/zkServer.sh start

sudo wget https://downloads.apache.org/kafka/3.9.0/kafka_2.13-3.9.0.tgz
sudo tar -zxf kafka_2.13-3.9.0.tgz
sudo mv kafka_2.13-3.9.0 /usr/local/kafka
sudo mkdir /tmp/kafka-logs

ip=`curl http://169.254.169.254/latest/meta-data/public-hostname`
sudo sed -i "s/#listeners=PLAINTEXT:\/\/:9092/listeners=PLAINTEXT:\/\/$ip:9092/g" /usr/local/kafka/config/server.properties

# due to AWS network stablishment process, check if 60 seconds is enough for your situation
(sleep 120 && sudo /usr/local/kafka/bin/kafka-server-start.sh -daemon /usr/local/kafka/config/server.properties )&

echo "Finished."
```

G. Adding more Kafka brokers to cluster, and configuring each one individually

Solving this problem in small solutions:

- First solution: simply add the count element to your aws instance. When you execute the command "terraform apply", those correspondingly number of similar instances will be created. It's a fast and easy way of provision multiple EC2 instances. However, all the internal detailed configurations of the installed software need to be done manually EC2 instance by EC2 instance: which at scale could be unfeasible.

```
resource "aws_instance" "exampleCLUSTER" {
  ami                    = "ami-045269a1f5c90a6a0"
  instance_type          = "t2.small"

  count = 4

  vpc_security_group_ids  = [aws_security_group.instance.id]
  key_name               = "vockey"

  user_data = "${file("creation.sh")}"

  user_data_replace_on_change = true

  tags = {
    Name = "terraform-example-kafka"
  }
}
```

- Second solution: being able to provision multiple EC2 instance, and at the same time configure all the internal detailed of each one of the EC2 instances. It has the advantage of accelerating the manual configurations by automation, and at scale. The drawback is the complexity and the lack of global configuration, i.e., configuration related with all the EC2 instances for all the EC2 instances.

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region      = "us-east-1"
  access_key  = "YOUR_ACCESS_KEY"
  secret_key  = "YOUR_SECRET_KEY"
  token       = "YOUR_TOKEN"
}

variable "nBroker" {
  description = "number of brokers"
  type        = number
  default     = 3
}

resource "aws_instance" "exampleCluster" {
  ami                    = "ami-045269a1f5c90a6a0"
  instance_type          = "t2.small"
  count                  = var.nBroker
  vpc_security_group_ids  = [aws_security_group.instance.id]
  key_name               = "vockey"

  user_data     = base64encode(templatefile("creation.sh",  {
    idBroker = "${count.index}"
    totalBrokers = var.nBroker
  }))

  user_data_replace_on_change = true

  tags = {
```

```
    Name = "terraform-example-kafka.${count.index}"
  }
}

output "publicdnslist" {
  value = "${formatlist("%v", aws_instance.exampleCluster.*.public_dns)}"
}

resource "aws_security_group" "instance" {
  name = var.security_group_name
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
    ingress {
    from_port   = 2181
    to_port     = 2181
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
    ingress {
    from_port   = 2888
    to_port     = 2888
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
    ingress {
    from_port   = 3888
    to_port     = 3888
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
    ingress {
    from_port   = 9092
    to_port     = 9092
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
}

variable "security_group_name" {
  description = "The name of the security group"
  type        = string
  default     = "terraform-example-instances"
}
```

```
#!/bin/bash

echo "Starting..."
cd
sudo wget https://dlcdn.apache.org/zookeeper/zookeeper-3.9.3/apache-zookeeper-3.9.3-bin.tar.gz
sudo tar -zxf apache-zookeeper-3.9.3-bin.tar.gz
sudo mv apache-zookeeper-3.9.3-bin /usr/local/zookeeper
sudo mkdir -p /var/lib/zookeeper
echo "tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
maxClientCnxns=60
initLimit=10
syncLimit=5" > /usr/local/zookeeper/conf/zoo.cfg

sudo yum -y install java-17-amazon-corretto-devel.x86_64

echo ${idBroker} > /var/lib/zookeeper/myid

sudo /usr/local/zookeeper/bin/zkServer.sh start

sudo wget https://downloads.apache.org/kafka/3.9.0/kafka_2.13-3.9.0.tgz
sudo tar -zxf kafka_2.13-3.9.0.tgz
```

```
sudo mv kafka_2.13-3.9.0 /usr/local/kafka
sudo mkdir /tmp/kafka-logs

ip=`curl http://169.254.169.254/latest/meta-data/public-hostname`
sudo sed -i "s/#listeners=PLAINTEXT:\/\/:9092/listeners=PLAINTEXT:\/\/$ip:9092/g"
/usr/local/kafka/config/server.properties

sudo sed -i "s/broker.id=0/broker.id=${idBroker}/g" /usr/local/kafka/config/server.properties

sudo sed -i "s/offsets.topic.replication.factor=1/offsets.topic.replication.factor=${totalBrokers}/g"
/usr/local/kafka/config/server.properties
sudo sed -i
"s/transaction.state.log.replication.factor=1/transaction.state.log.replication.factor=${totalBrokers}/g"
/usr/local/kafka/config/server.properties
sudo sed -i "s/transaction.state.log.min.isr=1/transaction.state.log.min.isr=${totalBrokers}/g"
/usr/local/kafka/config/server.properties

# due to AWS network stablishment process, check if 30 seconds is enough for your situation
(sleep 30 && sudo /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties )&

echo "Finished."
```

As explained in tutorial P4 - Distributed Kafka service, two global configurations are still missing in all the EC2 instances. Login in each machine and setup the following configuration:

At /usr/local/kafka/config/server.properties

```
#zookeeper connectivity (one per EC2 VM of this cluster)
zookeeper.connect=ec2-54-90-57-82.compute-1.amazonaws.com:2181,ec2-54-173-171-63.compute-
1.amazonaws.com:2181,ec2-54-236-47-54.compute-1.amazonaws.com:2181
```

And at /usr/local/zookeeper/conf/zoo.cfg

```
server.1=ec2-54-90-57-82.compute-1.amazonaws.com:2888:3888
server.2=ec2-54-173-171-63.compute-1.amazonaws.com:2888:3888
server.3=ec2-54-236-47-54.compute-1.amazonaws.com:2888:3888
```

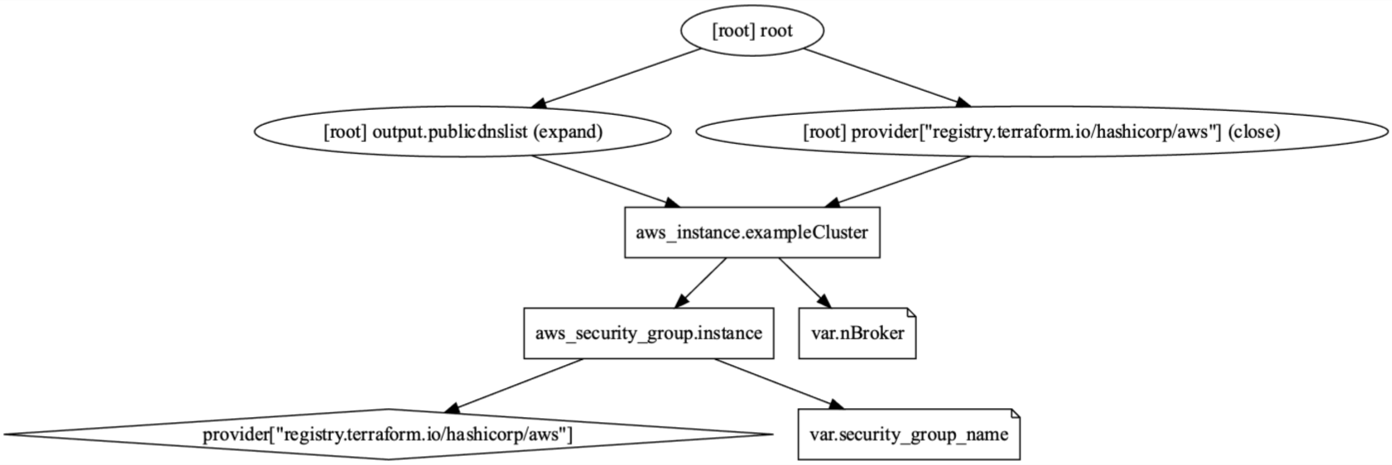
H. How to show the dependencies in terraform?

To show the dependencies of your deployment environment use the following command:

```
terraform graph -draw-cycles
```

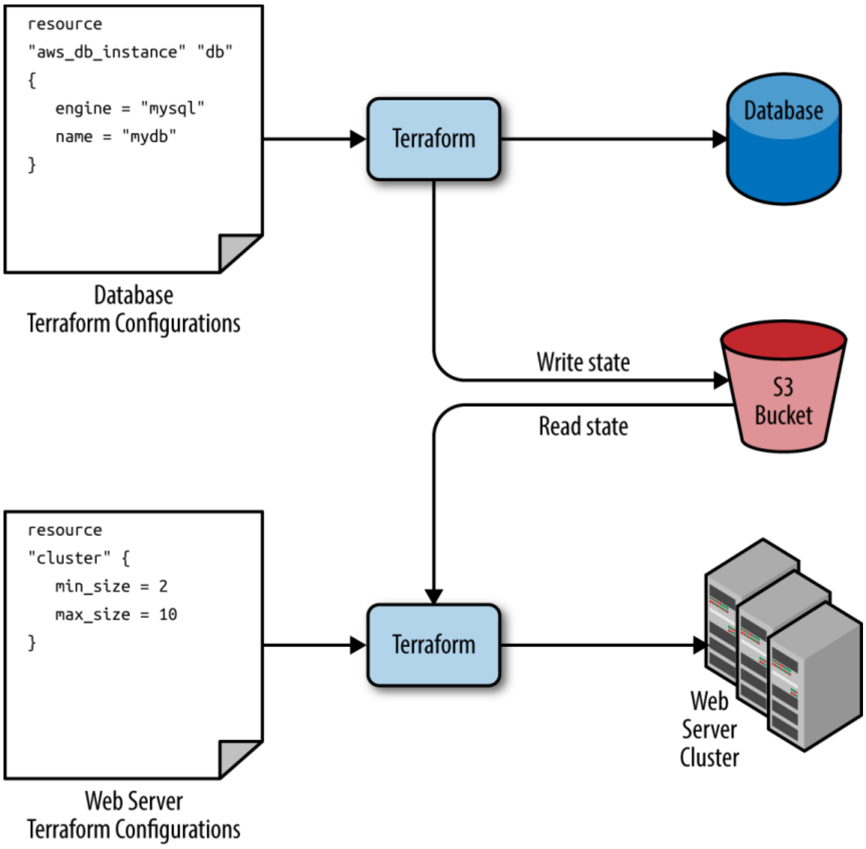
Then copy+paste the output for the graphviz, e.g., on https://dreampuf.github.io/GraphvizOnline.

Here is an example of you obtain:



I. Sharing terraform state between resources using AWS S3

The terraform state sharing allows the provisioning of an environment where the posterior resources depend on the formers, as depicted in the next figure. It could be done using a cloud storage or a local storage. S3 is an AWS service to store data.



Source: Brikman, Y. (2022). Terraform: Up and Running. " O'Reilly Media, Inc.".

Notice that to write the state in the storing an output should be defined in the terraform script file, i.e., to store locally the state of terraform:

```
output "publicdnslist" {
  value = "${formatlist("%v", aws_instance.exampleCluster.*.public_dns)}"
}
```

Then, you can list all the outputs, e.g.:

```
terraform state list
```

returning:

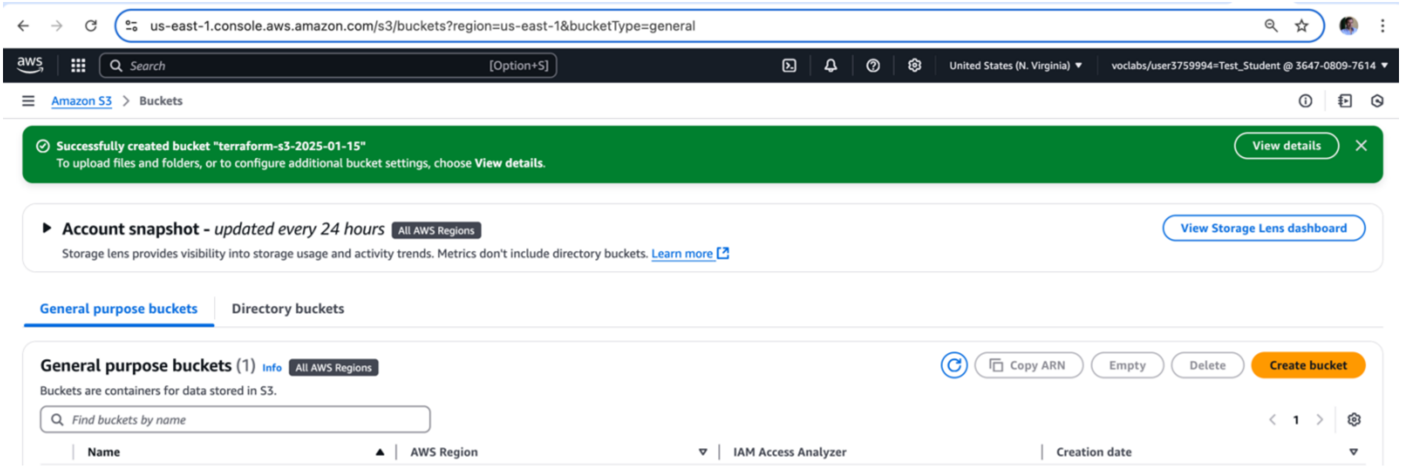
```
aws_instance.exampleCluster[0]
aws_instance.exampleCluster[1]
aws_instance.exampleCluster[2]
aws_security_group.instance
```

Or just show one of them:

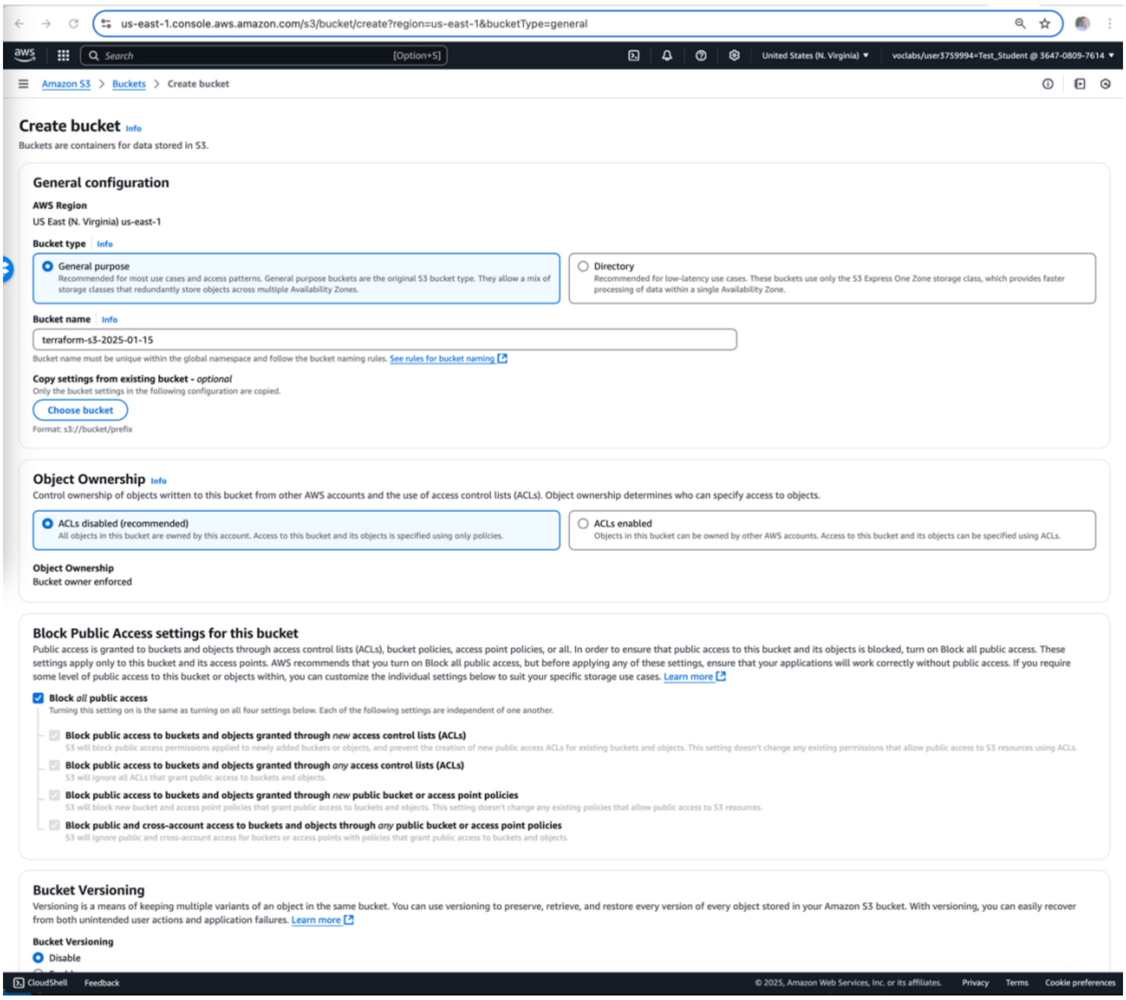
```
terraform state show 'aws_instance.exampleCluster[0]'
```

In Terraform, the state is a representation of the configuration of your infrastructure that is stored locally or remotely. Accessing the state locally involves using the terraform state command.

I.1. Choose the AWS S3 service, then, click on "**create bucket**":



Give it a unique name:



I.2. Terraform state will be stored in the S3 cloud environment. For that, firstly, create in a separate directory, a terraform file to create an AWS RDS (*e.g.*: mySQL cloud database) and store remotelly the name of the DB.

```
terraform {
  backend "s3" {
    bucket = "terraform-s3-2025-01-15"
```

```
    key= "stage/data-stores/mysql/terraform.tfstate"
    region = "us-east-1"
    access_key  = "YOUR_ACCESS_KEY"
    secret_key  = "YOUR_SECRET_KEY"
    token       = "YOUR_TOKEN"
}
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region = "us-east-1"
  access_key  = "YOUR_ACCESS_KEY"
  secret_key  = "YOUR_SECRET_KEY"
  token       = "YOUR_TOKEN"
}

variable "db_username" {
  description = "The username for the database"
  type        = string
  sensitive   = true
  default     = "teste"
}

variable "db_password" {
  description = "The password for the database"
  type        = string
  sensitive   = true
  default     = "testeteste"
}

variable "db_name" {
  description = "The name to use for the database"
  type        = string
  default     = "example_database_stage"
}

resource "aws_db_instance" "example" {
  identifier_prefix   = "terraform-up-and-running"
  engine              = "mysql"
  allocated_storage   = 20
  instance_class      = "db.t4g.micro"
  skip_final_snapshot = true
  publicly_accessible = true
  vpc_security_group_ids  = [aws_security_group.rds.id]
  db_name             = var.db_name

  username = var.db_username
  password = var.db_password
}

output "address" {
  value       = aws_db_instance.example.address
  description = "Connect to the database at this endpoint"
}

output "port" {
  value       = aws_db_instance.example.port
  description = "The port the database is listening on"
}

resource "aws_security_group" "rds" {
  name = var.security_group_name
  ingress {
    from_port   = 3306
    to_port     = 3306
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
```

```
  }
}

variable "security_group_name" {
  description = "The name of the security group"
  type        = string
  default     = "terraform-rds-instance"
}
```

Execute the usual command:

```
terraform init && terraform apply
```

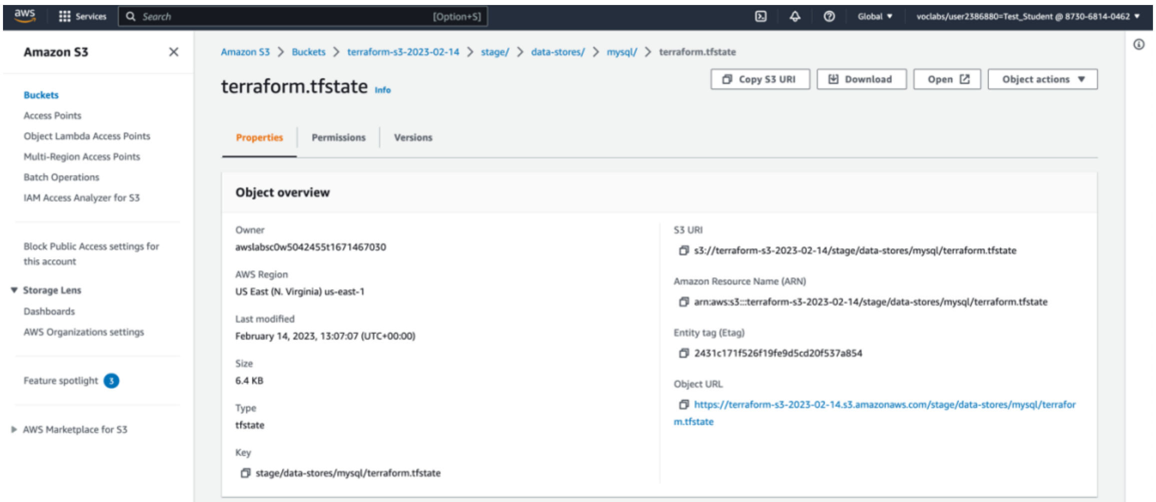
You can now check the outputs were written correctly to terraform state using the following commands:

```
terraform state list
```

```
terraform state show 'aws_db_instance.example'
```

```
terraform output
```

Moreover, if you navigate until:



You'll see that the *terraform state* of the **recently created mysql database** is stored there.

I.3. Now, in another directory, create a terraform file that can create an AWS EC2 instance writing the previous database address and port to a text file inside the EC2. The address and port are read from the state file of the previous step. With this mechanism you can create resources that are configured with parameters from the previous created resources. Facilitating the automation provisioning process!

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

data "terraform_remote_state" "db" {
  backend = "s3"
  config = {
    bucket = "terraform-s3-2025-01-15"
```

```
    key= "stage/data-stores/mysql/terraform.tfstate"
    region = "us-east-1"
    access_key  = "YOUR_ACCESS_KEY"
    secret_key  = "YOUR_SECRET_KEY"
    token       = "YOUR_TOKEN"
  }
}

provider "aws" {
  region = "us-east-1"
  access_key  = "YOUR_ACCESS_KEY"
  secret_key  = "YOUR_SECRET_KEY"
  token       = "YOUR_TOKEN"
}

resource "aws_instance" "exampleCluster" {
  ami                     = "ami-045269a1f5c90a6a0"
  instance_type           = "t2.small"
  vpc_security_group_ids  = [aws_security_group.instance.id]
  key_name                = "vockey"

  user_data     = base64encode(templatefile("creation.sh",  {
    address = data.terraform_remote_state.db.outputs.address
    port    = data.terraform_remote_state.db.outputs.port
  }))

  user_data_replace_on_change = true

  tags = {
    Name = "terraform-example-sharestate"
  }
}

resource "aws_security_group" "instance" {
  name = var.security_group_name
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
}

variable "security_group_name" {
  description = "The name of the security group"
  type        = string
  default     = "terraform-example-instance2"
}
```

And creation.sh with the following source code:

```
#!/bin/bash
cd

echo ${address} > address.txt
echo ${port} > port.txt
```

Execute the usual command:

```
terraform init && terraform apply
```

Login in the newly created EC2 instance and verify in the root directory that two new files are now created with the content provided by DB configuration.

J. Creating Lambda function and API Gateway trigger using S3

J.1. To create a lambda function, written in JAVA 8, that you previously compiled locally, you need to transfer the .jar file to S3, then create the AWS lambda function resource, and finally, add an API Gateway trigger for that lambda function.

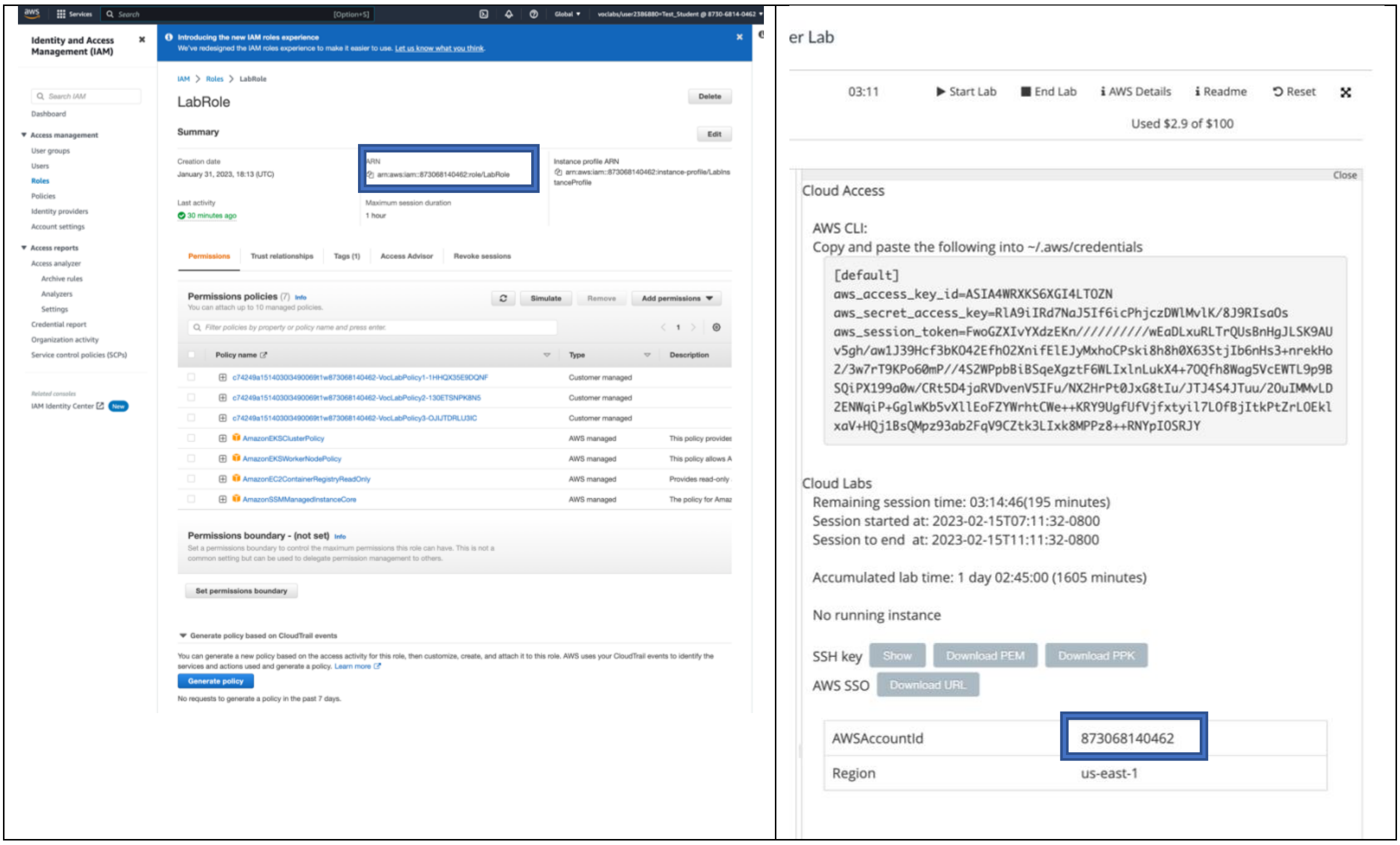
```
terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.15.0"
    }
    random = {
      source  = "hashicorp/random"
      version = "~> 3.1.0"
    }
    archive = {
      source  = "hashicorp/archive"
      version = "~> 2.2.0"
    }
  }
  required_version = "~> 1.2"
}

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

## S3 PART
resource "aws_s3_bucket" "lambda_bucket" {
  bucket = "terraform-lambda-2025-01-16"
}

 resource "aws_s3_object" "lambda_hello_world" {
   bucket = aws_s3_bucket.lambda_bucket.id
   key    = "lambda-java-example-0.0.1-SNAPSHOT.jar"
   source = "${path.module}/lambda-java-example-0.0.1-SNAPSHOT.jar"
 }

## LAMBDA FUNCTION PART
resource "aws_lambda_function" "hello_world" {
  function_name = "HelloWorld"
  s3_bucket = aws_s3_bucket.lambda_bucket.id
  s3_key    = aws_s3_object.lambda_hello_world.key
  runtime = "java11"
  handler = "example.Hello::handleRequest"
  role = "arn:aws:iam::8730681408695462:role/LabRole"
}

resource "aws_cloudwatch_log_group" "hello_world" {
  name = "/aws/lambda/${aws_lambda_function.hello_world.function_name}"
  retention_in_days = 30
}

### API GATEWAY PART
resource "aws_api_gateway_rest_api" "example" {
  name        = "ServerlessExample"
  description = "Terraform Serverless Application Example"
}

resource "aws_api_gateway_resource" "proxy" {
  rest_api_id = "${aws_api_gateway_rest_api.example.id}"
  parent_id   = "${aws_api_gateway_rest_api.example.root_resource_id}"
  #path_part    = "{proxy+}"
  path_part   = "helloworldpath"
}

resource "aws_api_gateway_method" "proxy" {
  rest_api_id   = "${aws_api_gateway_rest_api.example.id}"
  resource_id   = "${aws_api_gateway_resource.proxy.id}"
  http_method   = "ANY"
  authorization = "NONE"
}

resource "aws_api_gateway_integration" "lambda" {
  rest_api_id = "${aws_api_gateway_rest_api.example.id}"
```

```
  resource_id = "${aws_api_gateway_method.proxy.resource_id}"
  http_method = "${aws_api_gateway_method.proxy.http_method}"

  integration_http_method = "POST"
  type                    = "AWS_PROXY"
  uri                     = "${aws_lambda_function.hello_world.invoke_arn}"
}

resource "aws_api_gateway_deployment" "example" {
  depends_on = [
    aws_api_gateway_integration.lambda,
  ]
  rest_api_id = "${aws_api_gateway_rest_api.example.id}"
  stage_name  = "test"
}

resource "aws_lambda_permission" "apigw" {
  statement_id  = "AllowAPIGatewayInvoke"
  action        = "lambda:InvokeFunction"
  function_name = "${aws_lambda_function.hello_world.function_name}"
  principal     = "apigateway.amazonaws.com"

  # The /*/* portion grants access from any method on any resource
  # within the API Gateway "REST API".
  source_arn = "${aws_api_gateway_rest_api.example.execution_arn}/*/*"
}

# Output value definitions
output "lambda_bucket_name" {
  description = "Name of the S3 bucket used to store function code."
  value = aws_s3_bucket.lambda_bucket.id
}

output "function_name" {
  description = "Name of the Lambda function."
  value = aws_lambda_function.hello_world.function_name
}

output "base_url" {
  value = "${aws_api_gateway_deployment.example.invoke_url}"
}
```

Where the field role of resource "aws_lambda_function":

```
  role = "arn:aws:iam::8730681408695462:role/LabRole"
```

could be consulted on your AWS account, in the IAM service, or replacing your AWSAccountId number directly in the bold part of the field:



J.2. To test your lambda function, externally from any machine, you need to connect to AWS API Gateway, as explained in the Lambda-AWS tutorial:

```
curl -i -H "Content-Type: Application/json" --data "@body.json"  -X POST https://<your URL>/test/helloworldpath
```

And the result obtained is the following:

```
HTTP/2 200
content-type: application/json
content-length: 43
date: Wed, 15 Feb 2023 16:36:51 GMT
x-amzn-requestid: f2027a23-803d-42d3-8537-e2fe256d7480
x-amz-apigw-id: AY6FiH8XoAMFbvA=
x-custom-header: my custom header value
x-amzn-trace-id: Root=1-63ed0a23-652880af6750972f7523767e;Sampled=0
x-cache: Miss from cloudfront
via: 1.1 cb4f40303e252a22c4df5918669814ac.cloudfront.net (CloudFront)
x-amz-cf-pop: LIS50-C1
x-amz-cf-id: CAQtv4l6vFuEzDsjQv7tj56smEc7s4ZkcEffM3VobsGSKnB-1H2y4g==

{"message":"Hello Integracao Empresarial!"}
```

K. Kong and Konga deployed in AWS EC2 using Terraform

K.1. To install Kong you will need to follow the indications as explained in the Kong tutorial. They are now adapted to be hyperautomated using Terraform.

Where the .tf file can include the following instructions:

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

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

resource "aws_instance" "exampleInstallKong" {
  ami                    = "ami-045269a1f5c90a6a0"
  instance_type          = "t2.small"
  vpc_security_group_ids  = [aws_security_group.instance.id]
  key_name               = "vockey"

  user_data = "${file("deploy.sh")}"

  user_data_replace_on_change = true

  tags = {
    Name = "terraform-example-Kong"
  }
}

resource "aws_security_group" "instance" {
  name = var.security_group_name
  ingress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
}

variable "security_group_name" {
  description = "The name of the security group"
  type        = string
  default     = "terraform-kong-instance"
}

output "address" {
  value       = aws_instance.exampleInstallKong.public_dns
  description = "Connect to the database at this endpoint"
}
```

And the correspondingly script with the following:

```
#!/bin/bash
echo "Starting..."
```

```
sudo yum install -y docker

sudo service docker start

sudo docker network create kong-net

sudo docker run -d --name kong-database \
  --network=kong-net \
  -p 5432:5432 \
  -e "POSTGRES_USER=kong" \
  -e "POSTGRES_DB=kong" \
  -e "POSTGRES_PASSWORD=kongpass" \
  postgres:13

sudo docker run --rm --network=kong-net \
-e "KONG_DATABASE=postgres" \
-e "KONG_PG_HOST=kong-database" \
-e "KONG_PG_PASSWORD=kongpass" \
-e "KONG_PASSWORD=test" \
kong/kong-gateway:3.9.0.0 kong migrations bootstrap

sudo docker run -d --name kong-gateway \
--network=kong-net \
-e "KONG_DATABASE=postgres" \
-e "KONG_PG_HOST=kong-database" \
-e "KONG_PG_USER=kong" \
-e "KONG_PG_PASSWORD=kongpass" \
-e "KONG_PROXY_ACCESS_LOG=/dev/stdout" \
-e "KONG_ADMIN_ACCESS_LOG=/dev/stdout" \
-e "KONG_PROXY_ERROR_LOG=/dev/stderr" \
-e "KONG_ADMIN_ERROR_LOG=/dev/stderr" \
-e "KONG_ADMIN_LISTEN=0.0.0.0:8001, 0.0.0.0:8444 ssl" \
-e "KONG_ADMIN_GUI_URL=http://localhost:8002" \
-e KONG_LICENSE_DATA \
-p 8000:8000 \
-p 8443:8443 \
-p 8001:8001 \
-p 8002:8002 \
-p 8445:8445 \
-p 8003:8003 \
-p 8004:8004 \
-p 127.0.0.1:8444:8444 \
kong/kong-gateway:3.9.0.0

echo "Finished."
```

K.2. To install the UI Konga again you can follow the indications as explained in the Kong tutorial. They are now adapted to be hyperautomated using Terraform.

Where the .tf file can include the following instructions:

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

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

resource "aws_instance" "exampleInstallKonga" {
  ami                     = "ami-045269a1f5c90a6a0"
  instance_type           = "t2.small"
  vpc_security_group_ids  = [aws_security_group.instance.id]
  key_name                = "vockey"
```

```
  user_data = "${file("deploy.sh")}"

  user_data_replace_on_change = true

  tags = {
    Name = "terraform-example-Konga"
  }
}

resource "aws_security_group" "instance" {
  name = var.security_group_name
  ingress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
}

variable "security_group_name" {
  description = "The name of the security group"
  type        = string
  default     = "terraform-konga-instance"
}

output "address" {
  value       = aws_instance.exampleInstallKonga.public_dns
  description = "Connect to the database at this endpoint"
}
```

And the correspondingly script with the following:

```
#!/bin/bash
echo "Starting..."

sudo yum install -y docker

sudo service docker start

sudo docker pull pantsel/konga

sudo docker run -d --name konga -p 1337:1337 pantsel/konga

echo "Finished."
```

L. Camunda Engine deployed in AWS EC2 using Terraform

L.1. To install Camunda Engine you will need to follow the indications as explained in the Camunda tutorial. They are now adapted to be hyperautomated using Terraform.

Where the .tf file can include the following instructions:

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

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

resource "aws_instance" "exampleInstallCamundaEngine" {
  ami                    = "ami-045269a1f5c90a6a0"
  instance_type          = "t2.small"
  vpc_security_group_ids  = [aws_security_group.instance.id]
  key_name               = "vockey"

  user_data = "${file("deploy.sh")}"

  user_data_replace_on_change = true

  tags = {
    Name = "terraform-example-Camunda"
  }
}

resource "aws_security_group" "instance" {
  name = var.security_group_name
  ingress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
}

variable "security_group_name" {
  description = "The name of the security group"
  type        = string
  default     = "terraform-Camunda-instance"
}

output "address" {
  value       = aws_instance.exampleInstallCamundaEngine.public_dns
  description = "Connect to the database at this endpoint"
}
```

And the correspondingly script with the following:

```
#!/bin/bash
echo "Starting..."

sudo yum update -y

sudo yum install -y docker

sudo service docker start

sudo docker pull camunda/camunda-bpm-platform:latest

sudo docker run -d --name camunda -p 8080:8080 camunda/camunda-bpm-platform:latest

echo "Finished."
```

M. Quarkus project deployed in AWS EC2 using Terraform

M.1. To deploy a previous created and compiled Quarkus project you will need to have the image previously pushed to git, Then, use the following files for reference. Where the .tf file can include the following instructions:

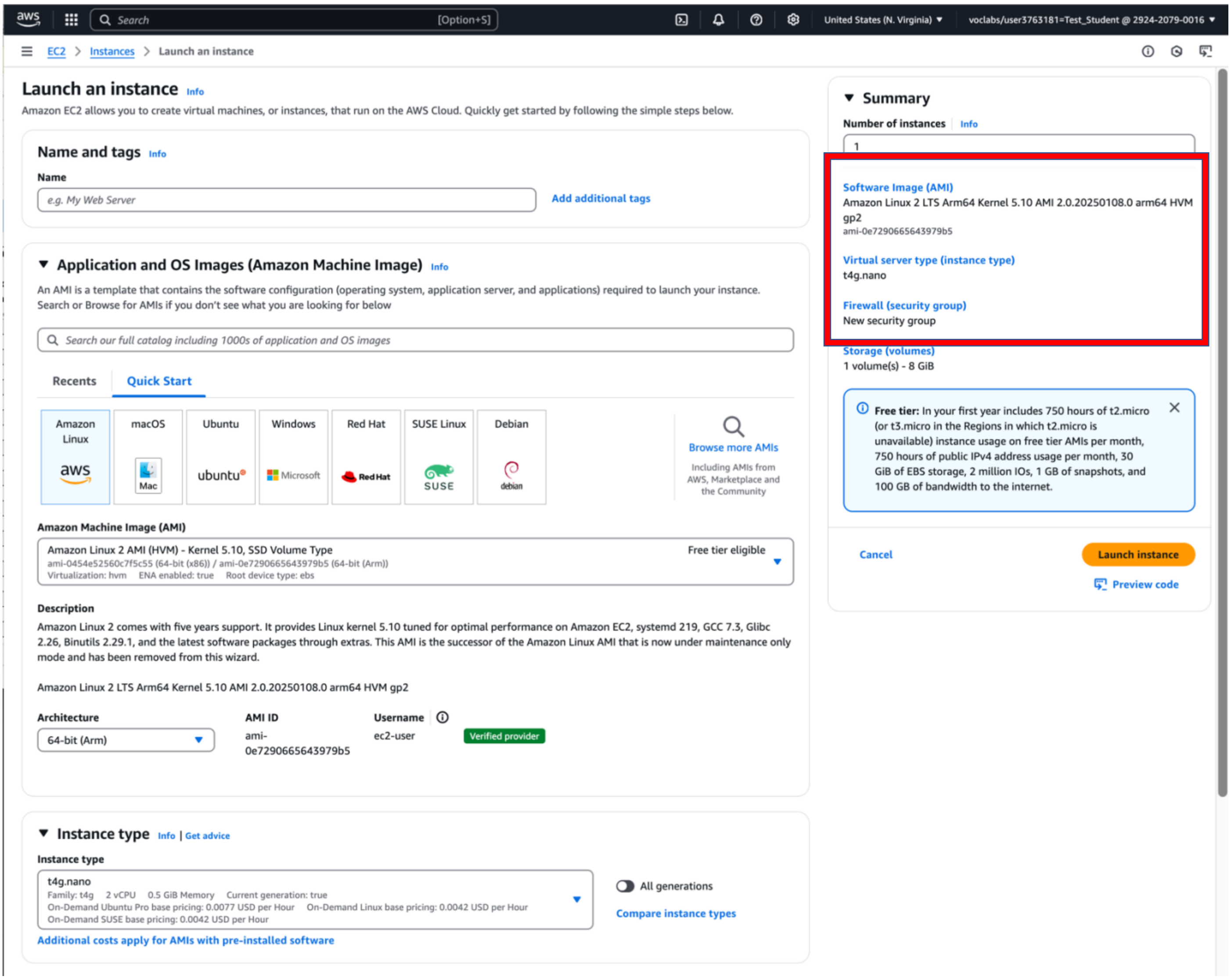
```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region       = "us-east-1"
  access_key   = "YOUR ACESS KEY"
  secret_key   = "YOUR SECRET KEY"
  token        = "YOUR TOKEN"
}


resource "aws_instance" "exampleDeployQuarkus" {
  ami                    = "ami-0e7290665643979b5" # Amazon Linux ARM AMI built by Amazon Web Services - FOR
DOCKER image COMPATIBILITY if compiled previously on ARM
  instance_type          = "t4g.nano"
  vpc_security_group_ids  = [aws_security_group.instance.id]
  key_name               = "vockey"

  user_data = "${file("quarkus.sh")}"

  user_data_replace_on_change = true

  tags = {
    Name = "terraform-deploy-QuarkusProject"
  }
}

resource "aws_security_group" "instance" {
  name = var.security_group_name
  ingress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
}

variable "security_group_name" {
  description = "The name of the security group"
  type        = string
  default     = "terraform-Quarkus-instance"
}

output "address" {
  value       = aws_instance.exampleDeployQuarkus.public_dns
  description = "Address of the Quarkus EC2 machine"
}
```

And the correspondingly script with the following:

```
#!/bin/bash
echo "Starting..."

sudo yum install -y docker

sudo service docker start

sudo docker login -u "YOUR DOCKER USERNAME" -p "YOUR DOCKER PASSWORD"

sudo docker pull YOUR DOCKER USERNAME/tryout1:1.0.0-SNAPSHOT

sudo docker run -d --name tryout2 -p 9000:9000 YOUR DOCKER USERNAME/tryout1:1.0.0-SNAPSHOT

echo "Finished."
```

Docker images built for ARM architecture may not work on AMD architecture machines. This is because ARM and AMD processors use different instruction sets and have different hardware architectures.

For this example, the AMI identifier and instance type were extracted from the launching instance AWS GUI.



N. Ollama deployed in AWS EC2 using Terraform

N.1. To install Ollama in a AWS EC2 instance using Terraform follows the following the **.tf file** to create the needed resources:

```
terraform {
  required_version = ">= 1.0.0, < 2.0.0"

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 4.0"
    }
  }
}

provider "aws" {
  region      = "us-east-1"
  access_key  = "YOUR AWS ACCESS KEY"
  secret_key  = "YOUR AWS SECRET KEY"
  token = "YOUR AWS TOKEN"
}

resource "aws_instance" "exampleOllamaConfiguration" {
  ami                     = "ami-045269a1f5c90a6a0"
  instance_type           = "t2.medium"
  count                   = 1
  vpc_security_group_ids  = [aws_security_group.instance.id]

  root_block_device {
    volume_size = 24 # In Gb
  }

  key_name  = "vockey"
  user_data = "${file("creation.sh")}"

  user_data_replace_on_change = true

  tags = {
    Name = "terraform-example-ollama"
  }
}

resource "aws_security_group" "instance" {
  name = var.security_group_name
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
    ingress {
    from_port   = 11434
    to_port     = 11434
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port        = 0
    to_port          = 0
    protocol         = "-1"
    cidr_blocks      = ["0.0.0.0/0"]
    ipv6_cidr_blocks = ["::/0"]
  }
}

variable "security_group_name" {
  description = "The name of the security group"
  type        = string
  default     = "terraform-ollama-example-instance"
}

output "address" {
  value       = aws_instance.exampleOllamaConfiguration[*].public_dns
  description = "Address of the Kafka EC2 machine with ollama"
}
```

N.2. And the following bash script to be executed after EC2 resource creation by the following script **creation.sh**:

```
#!/bin/bash
cd
sudo yum update -y

sudo curl -fsSL https://ollama.com/install.sh | sh

export HOME=$HOME:/usr/local/bin
sudo sed -i "s/\[Install\]/Environment=\"OLLAMA_HOST=0.0.0.0:11434\"\n\[Install\]/g"
/etc/systemd/system/ollama.service

sudo systemctl enable ollama
sudo systemctl start ollama

ollama pull llama3.2
```

Hint 1: Additionally, if inside the EC2 instance you would like to check the models loaded use the following command:

```
[ec2-user@ip-172-31-23-142 ~]$ ollama list
NAME            ID              SIZE      MODIFIED
llama3.2:1b     baf6a787fdff    1.3 GB    7 minutes ago
```

Hint 2: Additionally, if inside the EC2 instance you would like to check the processors loaded use the following command:

```
[ec2-user@ip-172-31-23-142 ~]$ ollama ps
NAME            ID              SIZE      PROCESSOR    UNTIL
llama3.2:1b     baf6a787fdff    2.2 GB    100% CPU     4 minutes from now
```

Hint 3: Additionally, a request example can be tested with the following remote curl command:

```
curl http://<EC2 DNS NAME>:11434/api/generate -d '{
  "model": "llama3.2",
  "prompt": "Why is the sky blue?",
  "stream": false
}'
```

The response obtained is similar with the following:


```
{"model":"llama3.2","created_at":"2025-02-12T15:18:11.983219976Z","response":"The sky appears blue because of a phenomenon called scattering, which occurs when sunlight interacts with the tiny molecules of gases in the Earth's atmosphere.\n\nHere's a simplified explanation:\n\n1. **Sunlight**: When the sun shines, it emits white light, which contains all the colors of the visible spectrum (red, orange, yellow, green, blue, indigo, and violet).\n2. **Atmospheric scattering**: As sunlight enters the Earth's atmosphere, it encounters tiny molecules of gases such as nitrogen (N2) and oxygen (O2). These molecules are much smaller than the wavelength of light.\n3. **Scattering**: When sunlight hits these tiny molecules, it scatters in all directions. However, not all wavelengths scatter equally. The shorter (blue) wavelengths are scattered more than the longer (red) wavelengths by the smaller gas molecules.\n4. **Blue light dominates**: As a result of this scattering effect, the blue light is dispersed throughout the atmosphere, while the red and orange light continue to travel in a straight line.\n5. **Our eyes perceive the sky as blue**: From our perspective on the surface of the Earth, we see the scattered blue light coming from all directions, which gives the sky its blue appearance.\n\nThis phenomenon is more pronounced during the daytime when the sun is overhead, and the scattering effect is greater. During sunrise and sunset, the sun's rays have to travel through more of the atmosphere, scattering off even more molecules and creating the warm hues we see.\n\nSo, in short, the sky appears blue because of the scattering of sunlight by tiny molecules in the Earth's atmosphere, which favors shorter wavelengths (like blue light) over longer wavelengths (like red light).","done":true,"done_reason":"stop","context":[128006,9125,128007,271,38766,1303,33025,2696,25,6790,220,2366,18,271,128009,128006,882,128007,271,10445,374,279,13180,6437,30,128009,128006,78191,128007,271,791,13180,8111,6437,1606,315,264,25885,2663,72916,11,902,13980,994,40120,84261,449,279,13987,35715,315,45612,304,279,9420,596,16975,382,8586,596,264,44899,16540,1473,16,13,3146,31192,4238,96618,3277,279,7160,65880,11,433,73880,4251,3177,11,902,5727,682,279,8146,315,279,9621,20326,320,1171,11,19087,11,14071,11,6307,11,6437,11,1280,7992,11,323,80836,4390,17,13,3146,16888801,33349,72916,96618,1666,40120,29933,279,9420,596,16975,11,433,35006,13987,35715,315,45612,1778,439,47503,320,45,17,8,323,24463,320,46,17,570,4314,35715,527,1790,9333,1109,279,46406,315,3177,627,18,13,3146,3407,31436,96618,3277,40120,13280,1521,13987,35715,11,433,1156,10385,304,682,18445,13,4452,11,539,682,93959,45577,18813,13,578,24210,320,12481,8,93959,527,38067,810,1109,279,5129,320,1171,8,93959,555,279,9333,6962,35715,627,19,13,3146,10544,3177,83978,96618,1666,264,1121,315,420,72916,2515,11,279,6437,3177,374,77810,6957,279,16975,11,1418,279,2579,323,19087,3177,3136,311,5944,304,264,7833,1584,627,20,13,3146,8140,6548,45493,279,13180,439,6437,96618,5659,1057,13356,389,279,7479,315,279,9420,11,584,1518,279,38067,6437,3177,5108,505,682,18445,11,902,6835,279,13180,1202,6437,11341,382,2028,25885,374,810,38617,2391,279,62182,994,279,7160,374,32115,11,323,279,72916,2515,374,7191,13,12220,64919,323,44084,11,279,7160,596,45220,617,311,5944,1555,810,315,279,16975,11,72916,1022,1524,810,35715,323,6968,279,8369,82757,584,1518,382,4516,11,304,2875,11,279,13180,8111,6437,1606,315,279,72916,315,40120,555,13987,35715,304,279,9420,596,16975,11,902,54947,24210,93959,320,4908,6437,3177,8,927,5129,93959,320,4908,2579,3177,570],"total_duration":72783017580,"load_duration":3846465248,"prompt_eval_count":31,"prompt_eval_duration":3108000000,"eval_count":344,"eval_duration":65784000000}
%
```

References

URLs with other resources

- Terraform Up & Running, Writing Infrastructure as Code, Yevgeniy Brikman, 3rd edition (2022). https://www.terraformupandrunning.com/
    - The github with all the source code from the book is at: https://github.com/brikis98/terraform-up-and-running-code
- https://developer.hashicorp.com/terraform/downloads
- https://developer.hashicorp.com/terraform/tutorials/aws-get-started
- https://stackoverflow.com/questions/41596412/how-to-use-terraform-output-as-input-variable-of-another-terraform-template
- https://developer.hashicorp.com/terraform/tutorials/aws/lambda-api-gateway
- https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html
- https://registry.terraform.io/providers/hashicorp/aws/2.34.0/docs/guides/serverless-with-aws-lambda-and-api-gateway
- Ollama API documentation: https://github.com/ollama/ollama/blob/main/docs/api.md
- Ollama FAQs documentation: https://github.com/ollama/ollama/blob/main/docs/faq.md
- Ollama readme page https://github.com/ollama/ollama?tab=readme-ov-file