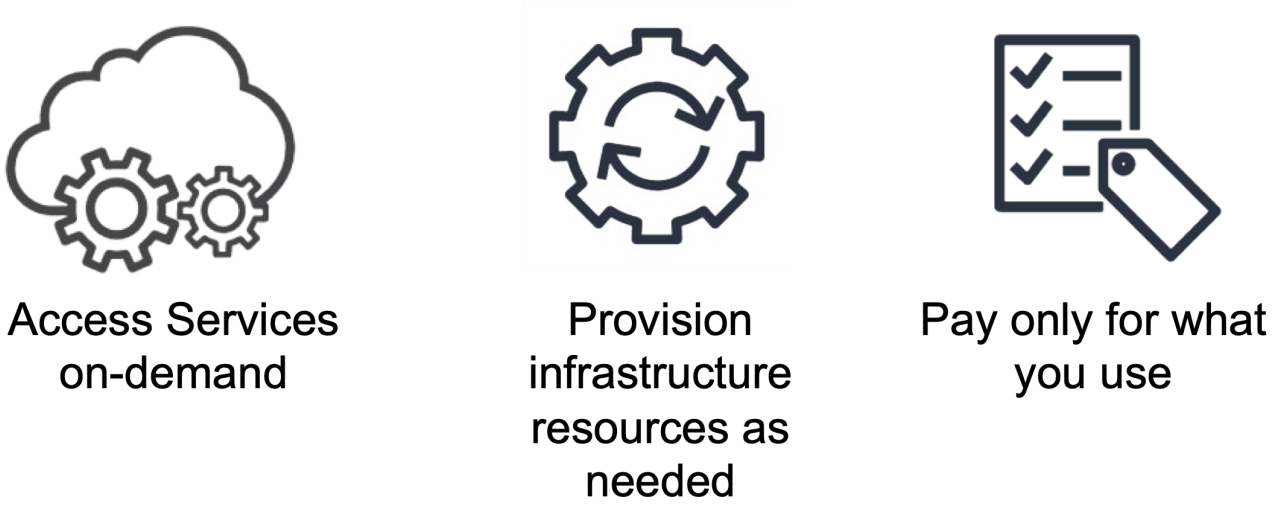
Cloud Overview

Insight 1. Cloud computing definition

Cloud computing is a technology that allows users to access and use computer resources (such as servers, storage, databases, networking, software, and more) over the internet, typically through a service provided by a third-party provider. Instead of owning and managing physical hardware and software, users can rent or subscribe to these resources on a pay-as-you-go basis, often referred to as a "pay-as-you-grow" model. These resources can be rapidly provisioned and released with minimal management effort.

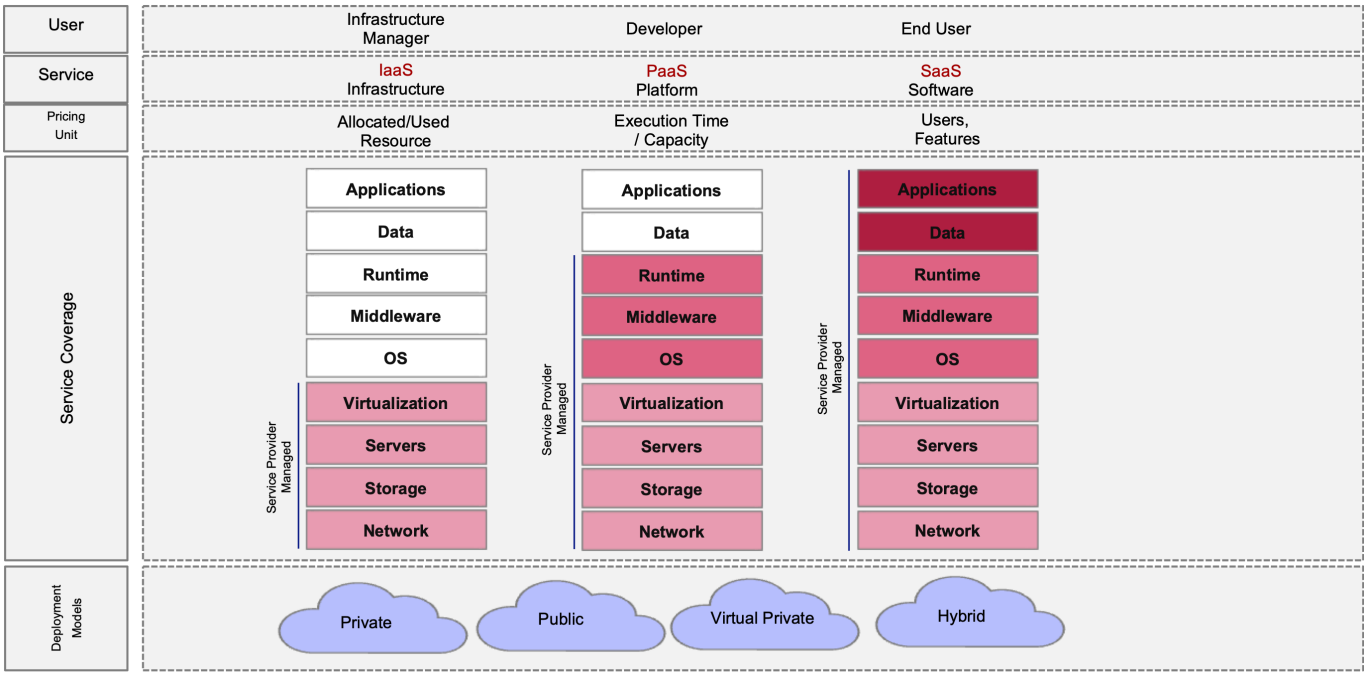


Cloud computing providers deliver these services to businesses, organizations, and individuals, allowing them to scale their IT infrastructure and services easily without the need for substantial upfront investments in hardware and software.

Cloud computing has become a fundamental part of modern IT infrastructure and is used for various purposes, such as hosting websites and web applications, data storage and backup, running virtual machines, and supporting big data analytics, artificial intelligence, and machine learning workloads, among others. Popular cloud computing providers include Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), and many others.

Insight 2. Cloud computing services

SaaS, PaaS, and IaaS are three different categories of cloud computing services that provide varying levels of control and management over the underlying infrastructure and applications.



Here's a breakdown of the key differences between SaaS, PaaS, and IaaS:

Software as a Service (SaaS)

- Definition: SaaS is a cloud computing model in which software applications are hosted and provided as a service to users over the internet. Users access the software through a web browser, and the software is managed, maintained, and updated by the SaaS provider.
- User Control: Users have the least control over the underlying infrastructure and application. They can customize settings and configurations within the application but cannot modify the underlying software or infrastructure.
- Examples: Popular SaaS applications include Gmail, Microsoft 365 (formerly Office 365), Salesforce, and Dropbox.

Platform as a Service (PaaS)

- Definition: PaaS is a cloud computing model that provides a platform or environment for developers to build, deploy, and manage applications. It includes tools, frameworks, and services for application development, such as databases, development frameworks, and application hosting.
- User Control: Developers have control over the applications they build and deploy, including the code and application settings. However, they have limited control over the underlying infrastructure.
- Examples: Microsoft Azure App Service, Google App Engine, and Heroku.

Infrastructure as a Service (IaaS)

- Definition: IaaS is a cloud computing model that provides virtualized computing resources over the internet. Users can rent virtual machines, storage, and networking resources, giving them more control over the infrastructure compared to SaaS and PaaS.
- User Control: Users have more control and responsibility for managing the virtual machines, operating systems, and software applications installed on them. They can configure and manage the infrastructure according to their needs.

- Examples: Amazon Web Services (AWS) EC2, Microsoft Azure Virtual Machines, and Google Compute Engine.

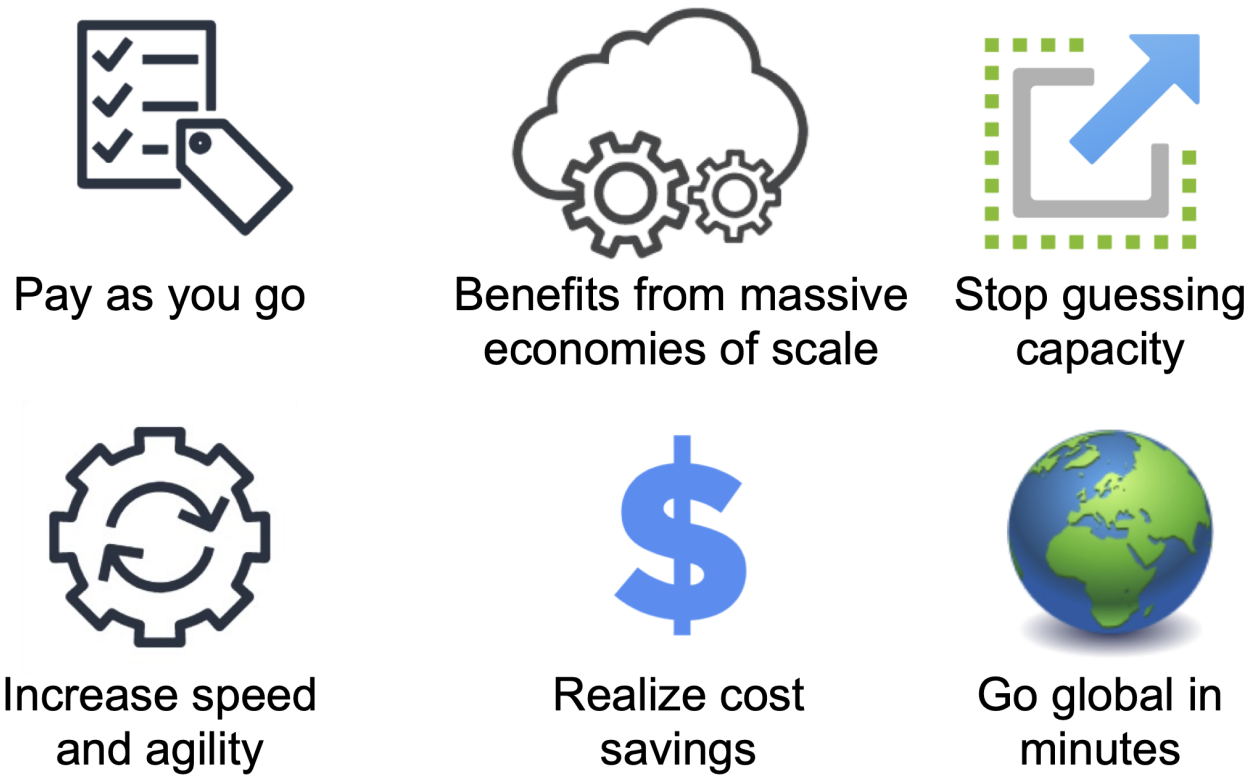
In summary, the key difference between these cloud service models is the level of control and management they offer to users:

- SaaS provides fully managed software applications, offering the least control over infrastructure.
- PaaS offers a platform and development environment for building applications, with more control over applications and less over infrastructure.
- IaaS provides virtualized infrastructure resources, giving users the most control and responsibility for managing infrastructure components.

The choice between these models depends on the specific needs of an organization or project, ranging from a focus on software usage (SaaS) to application development (PaaS) or infrastructure management (IaaS). Many organizations also use a combination of these models to meet different requirements within their IT ecosystems.

Insight 3. Cloud computing benefits

Cloud computing offers a wide range of benefits for individuals, businesses, and organizations of all sizes. Here are some of the key advantages of using cloud services:



- **Scalability:** Cloud services are highly scalable, allowing users to easily increase or decrease their computing resources as needed. This flexibility is particularly useful for businesses with fluctuating workloads, as they can avoid over-provisioning or under-provisioning resources.
- **Cost-Efficiency:** Cloud computing follows a pay-as-you-go model, which means you only pay for the resources you actually use. This can result in cost savings compared to traditional on-premises

infrastructure, as it eliminates the need for upfront capital expenditures on hardware and reduces ongoing maintenance costs.

- **Accessibility:** Cloud services are accessible from anywhere with an internet connection, enabling remote access to applications and data. This accessibility is essential for remote work, collaboration, and business continuity, especially in times of crisis.
- **Reliability and Redundancy:** Leading cloud providers offer high levels of reliability and redundancy. They have data centers in multiple geographic regions and employ advanced backup and failover mechanisms to ensure that services remain available even in the face of hardware failures or disasters.
- **Security:** Cloud providers invest heavily in security measures and compliance certifications, making it easier for businesses to meet industry-specific and regulatory requirements. They often provide tools and services for identity management, encryption, and access control.
- **Automatic Updates and Patching:** Cloud providers handle software updates, patching, and maintenance tasks, reducing the burden on IT staff. This ensures that applications and services are running the latest, most secure versions.
- **Backup and Disaster Recovery:** Cloud services often include built-in backup and disaster recovery capabilities. Data can be automatically backed up and replicated to secure locations, helping organizations recover from data loss or disasters more effectively.
- **Collaboration and Productivity:** Cloud-based collaboration tools and productivity suites enable seamless teamwork, document sharing, and real-time communication among employees, regardless of their physical location.
- **Innovation and Development:** Cloud platforms provide a fertile ground for innovation, allowing developers to access a wide range of tools and services to build and deploy applications quickly. This accelerates the development cycle and fosters innovation.
- **Environmental Benefits:** Cloud computing can be more environmentally friendly than traditional on-premises data centers because cloud providers can optimize resource utilization and energy efficiency on a large scale, reducing overall energy consumption and carbon emissions.
- **Global Reach:** Cloud services are available globally, allowing businesses to expand their operations and reach customers in different regions without the need to set up and manage physical data centers in those locations.
- **Resource Utilization:** Cloud providers often use virtualization and resource pooling techniques to maximize resource utilization, leading to more efficient use of computing resources.

Overall, cloud computing offers agility, cost savings, and access to a wide array of services that can transform the way businesses operate and innovate. However, it's important to choose the right cloud services and providers based on your specific needs and requirements to fully realize these benefits.

Insight 4. AWS EC2 example

Amazon Elastic Compute Cloud (Amazon EC2) is a core component of Amazon Web Services (AWS) and one of the most widely used cloud computing services. EC2 provides scalable, resizable, and flexible virtual

machine instances known as "EC2 instances" that run on AWS's cloud infrastructure. These instances allow users to run a wide variety of applications and workloads, providing on-demand access to computing resources.

Here are some key features and characteristics of Amazon EC2:

- **Virtual Machine Instances:** EC2 instances are virtualized computing resources that emulate physical servers. Users can choose from a wide range of instance types optimized for different use cases, such as general-purpose computing, memory-intensive applications, CPU-intensive workloads, and GPU-accelerated tasks.
- **Scalability:** EC2 instances can be easily scaled up or down to meet changing demands. Users can launch additional instances when needed and terminate them when they are no longer required. This scalability is especially beneficial for applications with fluctuating workloads.
- **Variety of Operating Systems:** EC2 supports multiple operating systems, including various flavors of Linux, Windows, and other operating systems. Users can select the operating system that best suits their application requirements.
- **Customization:** Users have control over the configuration of EC2 instances, including the choice of instance type, storage capacity, network settings, and security parameters. This allows for fine-tuning to meet specific performance and security needs.
- **Amazon Machine Images (AMIs):** AMIs are pre-configured templates that contain an operating system, software, and configurations. Users can choose from a wide range of public AMIs provided by AWS or create custom AMIs tailored to their applications.
- **Storage Options:** EC2 instances can be attached to various types of storage, including Amazon Elastic Block Store (EBS) for block storage and Amazon Elastic File System (EFS) for scalable file storage. These storage options provide durability and flexibility.
- **Security and Networking:** EC2 instances can be placed within Virtual Private Cloud (VPC) environments, allowing users to control network access, security groups, and subnets. AWS Identity and Access Management (IAM) can be used to manage permissions and access control.
- **Load Balancing:** AWS Elastic Load Balancing (ELB) can be used to distribute incoming traffic across multiple EC2 instances, ensuring high availability and fault tolerance for applications.
- **Auto Scaling:** AWS Auto Scaling allows users to automatically adjust the number of running EC2 instances based on traffic and performance metrics, ensuring that applications are responsive and cost-effective.
- **Monitoring and Management:** Amazon CloudWatch provides monitoring and management capabilities, enabling users to collect and analyze metrics, set alarms, and automate responses to changes in resource utilization.
- **Global Reach:** EC2 instances can be launched in multiple AWS regions around the world, allowing users to deploy applications closer to their end-users for improved latency and redundancy.

Amazon EC2 is a fundamental building block for many cloud-based applications and services, offering the flexibility and scalability needed to meet a wide range of computing needs, from hosting simple web

applications to running complex, data-intensive workloads. Users are billed based on the resources they consume, making it a cost-effective choice for businesses of all sizes.

Insight 5. OVH instance example

OVH's cloud instance service is part of OVH's Infrastructure as a Service (IaaS) offerings.

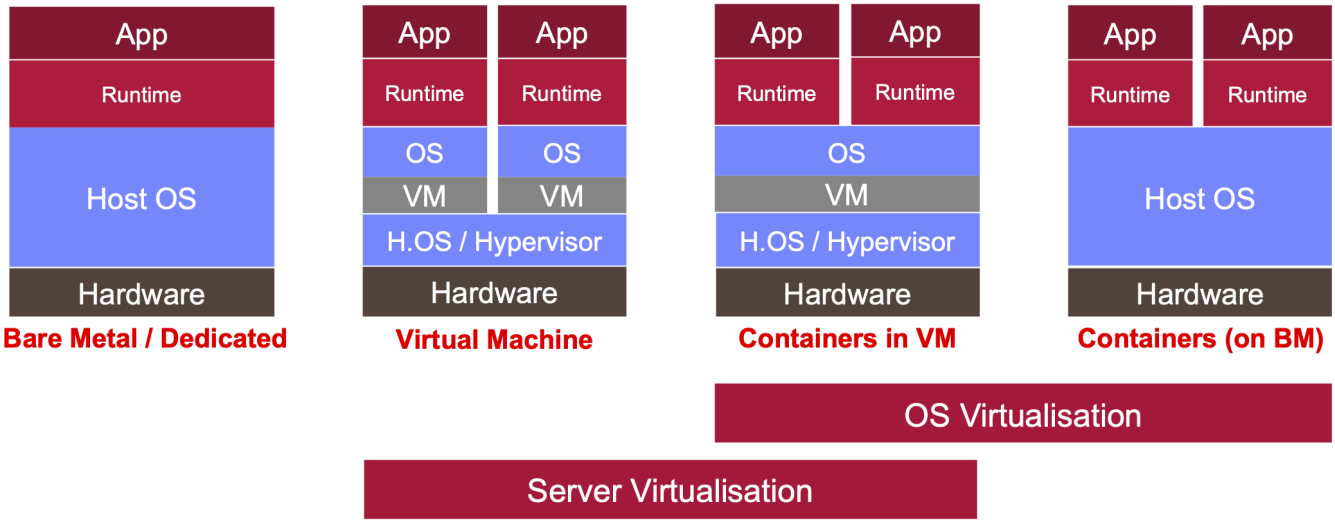
Here is a general overview of what OVH's cloud instance service typically included:

- **Virtual Instances:** OVH cloud instances are virtual machines (VMs) that you can deploy on their cloud infrastructure. These instances can run various operating systems and software applications, depending on your requirements.
- **Instance Types:** OVH offers a range of instance types with varying amounts of CPU, memory, and storage to accommodate different workloads. Users can choose the instance type that best suited their needs.
- **Scalability:** Similar to other cloud providers, OVH's cloud instances are scalable. You can easily resize your instances by adjusting CPU, memory, and storage resources to meet changing workload demands.
- **Operating System Choices:** OVH typically provides a selection of operating systems and templates, including various Linux distributions and Windows Server versions, which you can use to create your instances.
- **Storage Options:** OVH offers different storage options for cloud instances, including local storage and network-attached storage (NAS). Users could select the storage configuration that suited their performance and capacity requirements.
- **Networking and Security:** OVH allows users to configure networking settings, including virtual private networks (VPNs), firewalls, and load balancers, to control access and enhance security for their cloud instances.
- **Data Centers:** OVH has a global network of data centers, and users can choose the data center location where they want to deploy their instances. This allow for geographical redundancy and the ability to place resources closer to end-users.
- **Monitoring and Management:** OVH typically provides monitoring and management tools to help users keep an eye on the health and performance of their cloud instances. This might include dashboards, alerts, and usage metrics.
- **Billing Model:** OVH typically offers a pay-as-you-go billing model, where users are billed based on the resources and usage of their cloud instances. This can be cost-effective for businesses with fluctuating workloads.
- **Support and Documentation:** OVH usually offers customer support services and documentation to assist users with setting up, managing, and troubleshooting their cloud instances.

Containerisation and Virtualization

Insight 1. OS and Server virtualization

Virtualization, in the context of server and operating system (OS) virtualization, is a technology that allows multiple virtual instances or environments to run on a single physical server or host machine. This approach offers various benefits, including improved resource utilization, isolation between virtual environments, and greater flexibility in managing and deploying applications.



Here's a detailed explanation of server and OS virtualization:

Server Virtualization

Server virtualization involves partitioning a physical server into multiple virtual servers, each with its own isolated operating system, applications, and resources. These virtual servers, often referred to as virtual machines (VMs), act as independent entities and can run different operating systems or applications simultaneously on the same physical hardware. Here are some key aspects of server virtualization:

- **Hypervisor**: A hypervisor (also known as a virtual machine monitor or VMM) is a software or firmware layer that manages and controls the virtualization process. It allows multiple VMs to share the underlying hardware resources without interference.
- **Resource Allocation**: Server virtualization allows administrators to allocate specific amounts of CPU, memory, storage, and network resources to each VM. This resource isolation prevents one VM from adversely affecting the performance of others.
- **Isolation**: VMs are isolated from one another. If one VM crashes or experiences issues, it typically does not impact the operation of other VMs on the same physical server.
- **Hardware Consolidation**: Server virtualization enables organizations to consolidate multiple physical servers onto a single physical host, reducing hardware costs and data center space requirements.
- **Migration and Portability**: VMs can be easily migrated between physical servers, making it convenient to perform maintenance, load balancing, and disaster recovery.
- **Snapshot and Cloning**: VMs can be snapshotted or cloned, allowing for quick backups and the rapid deployment of identical virtual environments.

- **Testing and Development**: Virtualization is valuable for software development and testing, as it allows developers to create and test in isolated environments that mimic production setups.
- **Cloud Computing**: Virtualization is a fundamental technology underlying cloud computing services, enabling cloud providers to offer flexible, scalable resources to customers.

Operating System Virtualization

Operating system virtualization, often referred to as containerization or container-based virtualization, is a lighter-weight form of virtualization that virtualizes the OS at the application level. Instead of creating multiple VMs with separate OS instances, containers share the host OS kernel and resources while maintaining isolation at the application level. Here are some key aspects of OS virtualization:

- **Containers**: Containers are lightweight, portable environments that package an application and its dependencies. They are isolated from one another and run on the same host OS.
- **Efficiency**: OS virtualization is more efficient in terms of resource utilization compared to traditional VMs because containers share the host OS kernel and require fewer system resources.
- **Rapid Deployment**: Containers can be deployed quickly, making them ideal for microservices architectures and scalable applications.
- **Consistency**: Containers ensure consistent behavior across different environments (development, testing, production) since they include all required dependencies.
- **Orchestration**: Container orchestration platforms like Kubernetes can manage the deployment, scaling, and monitoring of containerized applications.
- **DevOps Practices**: OS virtualization aligns well with DevOps practices, as it enables developers and operations teams to work with identical environments and automate deployment processes.

In summary, both server virtualization and OS virtualization provide methods for maximizing resource utilization, improving scalability, and enhancing the management of applications and services in modern IT environments. The choice between these approaches depends on specific use cases, performance requirements, and management preferences.

Insight 2. Hypervisors and VMM examples

A hypervisor, also known as a Virtual Machine Monitor (VMM), is a fundamental component of virtualization technology that allows multiple virtual machines (VMs) to run on a single physical server or host machine. Hypervisors provide an abstraction layer between the physical hardware and the virtual machines, enabling the efficient sharing of resources and isolation between VMs. OpenStack and VMware ESXi are two examples of hypervisor technologies, each serving different purposes in the world of virtualization:

- Hypervisor in General:

Definition: A hypervisor is software, firmware, or hardware that creates and manages virtual machines. It abstracts and partitions the underlying physical hardware resources, such as CPU, memory, storage, and network, to allocate them to multiple VMs. **Types of Hypervisors**: There are two main types of hypervisors:

- Type 1 Hypervisor (Bare-Metal Hypervisor):

This type runs directly on the physical hardware without the need for an underlying operating system. It provides high performance and is typically used in enterprise environments. VMware ESXi is an example of a Type 1 hypervisor.

- Type 2 Hypervisor (Hosted Hypervisor)

This type runs on top of a standard operating system and is often used for development, testing, or desktop virtualization. Examples include Oracle VirtualBox and VMware Workstation.

Now, let's specifically discuss OpenStack and VMware ESXi:

OpenStack

- Type: OpenStack is not a traditional hypervisor but rather a cloud computing platform that includes various components for managing and provisioning virtual resources, including compute, storage, and networking resources.
- Role: OpenStack provides the framework for creating and managing cloud environments. Within OpenStack, the "Nova" component is responsible for managing compute resources, which involves interaction with hypervisors. It supports various hypervisors, including KVM, Xen, VMware, and more.
- Use Case: OpenStack is commonly used for building private and public clouds, enabling organizations to offer Infrastructure as a Service (IaaS) to their users. It manages the underlying hypervisors and virtual machine instances.

VMware ESXi

- Type: VMware ESXi is a Type 1 hypervisor.
- Role: ESXi is designed to be installed directly on physical servers and provides a robust and efficient virtualization platform. It manages and monitors VMs running on the host hardware.
- Use Case: VMware ESXi is widely used in enterprise environments for server virtualization. It allows organizations to consolidate multiple workloads onto a single physical server while maintaining high levels of performance, availability, and management features.

In summary, a hypervisor is a critical component in virtualization technology that enables the creation and management of virtual machines. OpenStack is a cloud computing platform that incorporates a hypervisor component (e.g., Nova) to manage compute resources in a cloud environment. VMware ESXi, on the other hand, is a standalone Type 1 hypervisor designed for server virtualization in enterprise data centers. Both play important roles in modern virtualization and cloud computing ecosystems, but they serve different purposes and have distinct use cases.

Insight 3. Docker

Docker is a containerization platform that allows you to package, distribute, and run applications and their dependencies in isolated containers. Containerization is a lightweight form of virtualization that has gained widespread popularity for its ability to simplify application deployment, improve resource utilization, and enhance development and operations workflows. Here's an overview of Docker containerization:

1. Containers:

Definition: Containers are lightweight, standalone executable packages that contain everything needed to run a piece of software, including the code, runtime, system libraries, and settings. Isolation: Containers provide process and file system isolation, allowing multiple containers to run on the same host with minimal interference. Consistency: Containers ensure consistent behavior across different environments (development, testing, production) by bundling all dependencies into a single unit.

2. Docker:

Docker Engine: Docker provides a platform for creating and managing containers. The Docker Engine is the core component responsible for building, running, and managing containers. Docker Hub: Docker Hub is a public registry that hosts a vast collection of pre-built container images that can be used as a starting point for your own containers. Docker Compose: Docker Compose is a tool for defining and running multi-container applications. It allows you to specify the services, networks, and volumes needed for your application in a single YAML file. Docker Swarm and Kubernetes: Docker can be integrated with orchestration tools like Docker Swarm or Kubernetes to manage containerized applications at scale, including load balancing, scaling, and high availability.

3. Benefits of Docker Containerization:

- **Portability**: Docker containers can run consistently on any system that supports Docker, regardless of the underlying infrastructure. This portability simplifies deployment across different environments.
- **Resource Efficiency**: Containers share the host OS kernel, which makes them lightweight and resource-efficient compared to traditional virtual machines (VMs).
- **Isolation**: Containers provide process and file system isolation, preventing conflicts between applications running on the same host.
- **Rapid Deployment**: Containers can be created and deployed quickly, making them ideal for microservices architectures and continuous integration/continuous deployment (CI/CD) pipelines.
- **Version Control**: Container images can be versioned, making it easy to roll back to previous versions if issues arise.
- **Scalability**: Containers can be easily scaled up or down to accommodate changes in traffic or demand.
- **Security**: Containers can be run with fine-grained access controls and are isolated from the host and other containers, enhancing security.

4. Use Cases:

- **Application Packaging**: Docker is commonly used for packaging applications and their dependencies into containers for easy distribution and deployment.
- **Microservices**: Docker is well-suited for building and deploying microservices-based architectures, where each component of an application is containerized and independently scalable.
- **DevOps and CI/CD**: Docker facilitates DevOps practices by streamlining development, testing, and deployment processes through containerization and automation.
- **Hybrid Cloud**: Docker containers are portable, making them an excellent choice for hybrid and multi-cloud deployments.

In summary, Docker containerization simplifies application deployment and management by encapsulating applications and their dependencies into lightweight, portable containers. These containers offer consistent behavior, improved resource utilization, and enhanced development and operations workflows, making Docker a valuable tool in modern software development and deployment.

Insight 4. Runnig a Docker container example

Here's an end-to-end description of running a Docker container, including building a container image using a Dockerfile example:

Scenario: We'll create a simple Docker image for a Python application that displays "Hello, Docker!" when run.

Prerequisites:

- Docker is installed on your machine.
- You have a basic understanding of Docker and have a directory with your application code and a Dockerfile.

Steps:

1. **Create a Directory for Your Project:** Start by creating a directory for your Docker project and navigate to it in your terminal.

```
mkdir my-docker-app
cd my-docker-app
```

2. **Create a Python Application:** Inside your project directory, create a simple Python application. Create a file named `app.py` and add the following code:

```
# app.py
print("Hello, Docker!")
```

3. **Create a Dockerfile:** Create a Dockerfile named `Dockerfile` (without any file extension) in your project directory. This file defines the image's configuration. Here's a simple example for a Python application:

```
# Use an official Python runtime as a parent image
FROM python:3.9-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80
```

```
# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

4. **Build the Docker Image:** Build the Docker image from the Dockerfile using the `docker build` command. Replace `my-docker-app` with your desired image name:

```
docker build -t my-docker-app .
```

   This command tags the image as `my-docker-app`.

5. **List Docker Images:** Verify that the image has been successfully built by listing the available images:

```
docker images
```

   You should see your newly created image listed.

6. **Run a Container:** Deploy a container from the image using the `docker run` command:

```
docker run -p 4000:80 my-docker-app
```

   - `-p 4000:80`: Map port 4000 on your host to port 80 on the container.
   - `my-docker-app`: Specify the image name.

7. **Access the Application:** Open a web browser and navigate to `http://localhost:4000`. You should see "Hello, Docker!" displayed.

8. **View Container Logs:** To view the logs generated by the container, open a new terminal window and use the following command:

```
docker logs -f <container_id>
```

   Replace `<container_id>` with the actual ID of your running container.

9. **Stop and Remove the Container:** When you're done with the container, you can stop and remove it using the following commands:

```
docker stop <container_id>   # Stop the container
docker rm <container_id>     # Remove the container
```

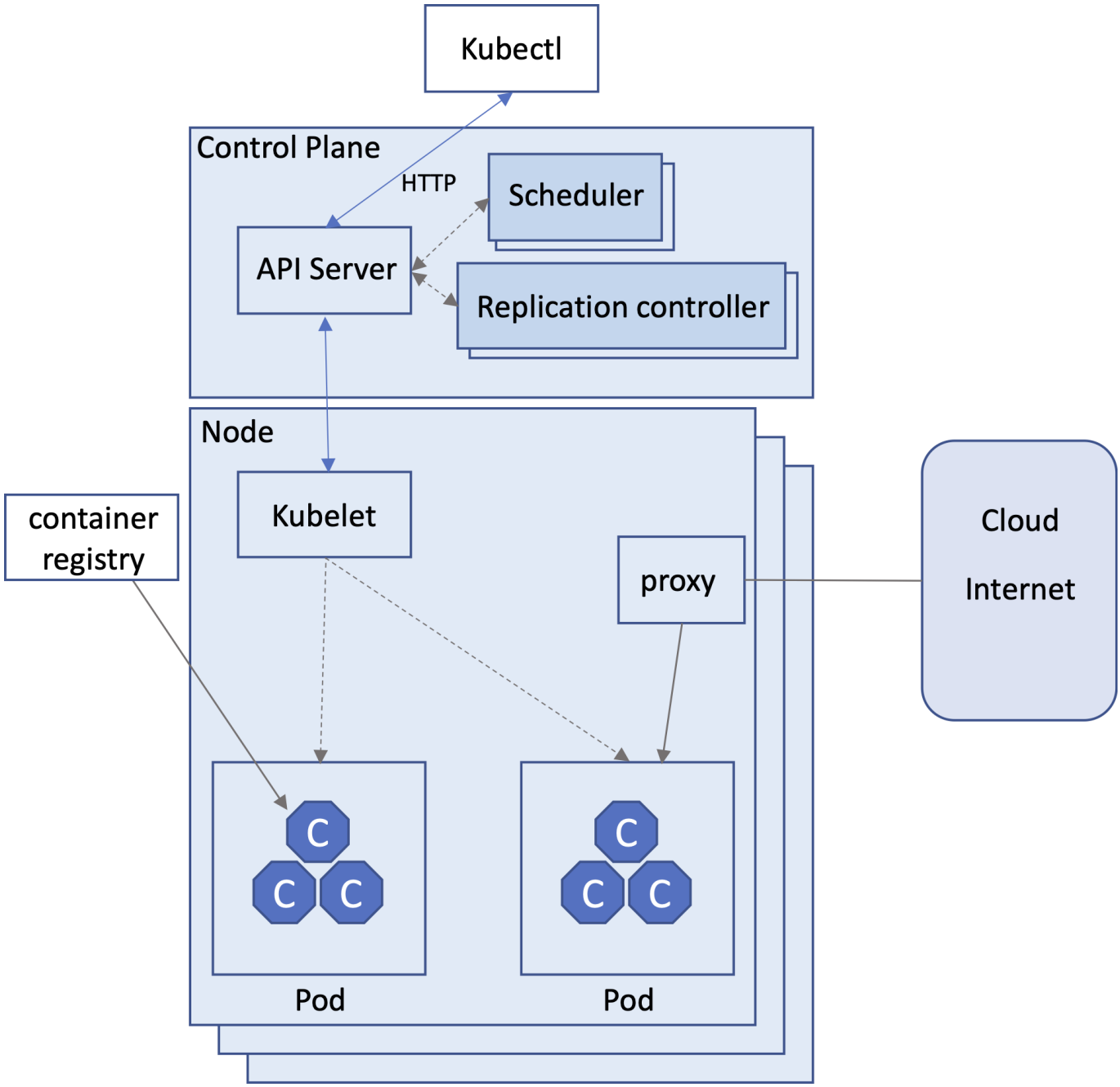
Replace `<container_id>` with the actual container ID.

That's it! You've successfully created a Docker image from your Python application, deployed a container from that image, and accessed your application running inside the container. This example demonstrates the basics of creating and running Docker containers, and you can build upon it for more complex applications and use cases.

Kubernetes Orchestration

Insight 1. Kubernetes overview

Kubernetes, often abbreviated as K8s, is an open-source container orchestration platform designed to automate the deployment, scaling, management, and operation of containerized applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF). Kubernetes is widely used for managing containerized workloads in various environments, from on-premises data centers to cloud infrastructure. Here's an overview of Kubernetes:



Key Components and Concepts:

- **Node**: A node is a physical or virtual machine that runs containers. Nodes are the worker machines in a Kubernetes cluster.
- **Cluster**: A Kubernetes cluster is a set of nodes (machines) that work together to run containerized applications. It includes a control plane (master) and one or more worker nodes.
- **Control Plane**: The control plane is responsible for managing and orchestrating the cluster. It includes components like the API server, etcd (a distributed key-value store for configuration data), controller manager, and scheduler.
- **Pod**: The smallest deployable unit in Kubernetes is a pod. A pod can contain one or more containers that share the same network namespace and storage volumes. Containers within a pod can communicate with each other using localhost.

- **Service**: A service provides a stable network endpoint for accessing a set of pods. It allows applications to discover and communicate with each other, even as pods are dynamically created or scaled.
- **ReplicaSet**: A ReplicaSet ensures that a specified number of pod replicas are running at all times. It can automatically scale the number of replicas up or down based on desired state.
- **Deployment**: A Deployment is a higher-level abstraction that manages ReplicaSets. It allows you to declaratively define the desired state of an application, and Kubernetes will make the necessary changes to achieve that state.
- **Namespace**: Namespaces provide a way to divide a cluster into multiple virtual clusters, each with its own set of resources and policies. They help with resource isolation and organization.
- **ConfigMap and Secret**: ConfigMaps and Secrets are used to store configuration data and sensitive information, respectively, and make them available to pods without hardcoding them.
- **Ingress**: An Ingress controller manages external access to services within the cluster, typically for HTTP and HTTPS traffic. It can route traffic to different services based on URL paths or hostnames.
- **Volume**: Volumes provide a way to persist data beyond the lifetime of a pod. Kubernetes supports various types of volumes, including network storage solutions.

Insight 2. Kubernetes key features and benefits

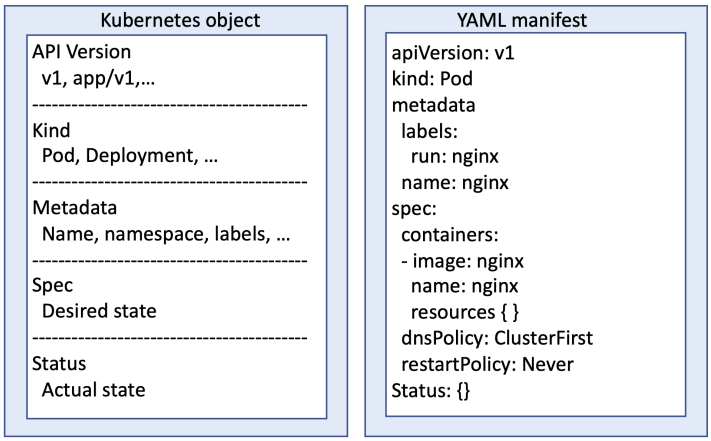
- **Orchestration**: Kubernetes automates the deployment and scaling of containerized applications, reducing manual intervention and human errors.
- **Scalability**: Kubernetes can scale applications horizontally by adding or removing pod replicas as needed to handle varying levels of traffic.
- **Self-Healing**: Kubernetes monitors the health of applications and automatically replaces or reschedules unhealthy pods to maintain the desired state.
- **Rolling Updates and Rollbacks**: Kubernetes supports rolling updates, allowing applications to be updated without downtime. If issues arise, rollbacks can be performed quickly.
- **Resource Management**: Kubernetes manages resource allocation, ensuring that applications get the necessary CPU and memory resources.
- **Multi-Cloud and Hybrid Cloud**: Kubernetes is cloud-agnostic, allowing applications to be deployed consistently across different cloud providers or on-premises infrastructure.
- **Ecosystem**: Kubernetes has a rich ecosystem of extensions, tools, and plugins that enhance its functionality and adapt it to various use cases.

Kubernetes has become the de facto standard for container orchestration and is widely adopted in the containerized application development and deployment ecosystem. It provides a powerful and flexible platform for managing containerized workloads, making it easier to build, deploy, and scale applications in modern computing environments.

Insight 3. Pod configuration example

Configuring a Kubernetes (K8s) Pod involves defining the desired state of the Pod, including the container(s) it will run, resource requirements, environment variables, volumes, and various settings. You can define a Pod's configuration using a Kubernetes manifest file written in YAML or JSON format.

- Equivalent of a class in object oriented programming
- A Pod in Kubernetes is the class of which there can be many instances with their own identity
- Every Kubernetes object has a system-generated unique identifier (also known as UID) to clearly distinguish between the entities of a system
- Each and every Kubernetes primitive follows a general structure, as shown in the picture



Below, we provide a step-by-step guide to configuring a K8s Pod:

- Create a Kubernetes Manifest File: Start by creating a YAML or JSON file that describes the configuration of the Pod. You can use a text editor or an integrated development environment (IDE) to create the file. For example, let's call it my-pod.yaml.
- Define the Pod's Metadata: In your my-pod.yaml file, specify metadata for the Pod, including the name and any labels or annotations you want to associate with it. Here's an example:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    app: my-app
```

- Specify the Container(s): Inside the Pod configuration, define the container(s) you want to run. Specify the image, command, and arguments for each container. Here's an example of a Pod with a single container:

```yaml
spec:
  containers:
    - name: my-container
      image: nginx:latest
      ports:
        - containerPort: 80
```

In this example, we're using the Nginx container image and exposing port 80.

- Set Resource Requirements: You can specify CPU and memory resource requests and limits for the container(s) to ensure resource isolation and proper allocation. Here's an example:

```
spec:
  containers:
    - name: my-container
      image: nginx:latest
      resources:
        requests:
          memory: "256Mi"
          cpu: "100m"
        limits:
          memory: "512Mi"
          cpu: "200m"
```

- Define Environment Variables: If your application requires environment variables, you can set them using the env field. Here's an example:

```
spec:
  containers:
    - name: my-container
      image: nginx:latest
      env:
        - name: MY_ENV_VAR
          value: "my-value"
```

- Mount Volumes: If your application needs to access external data or configuration files, you can define volumes and mount them into the container(s). Here's an example:

```
spec:
  containers:
    - name: my-container
      image: nginx:latest
      volumeMounts:
        - name: my-volume
          mountPath: /app/data
  volumes:
    - name: my-volume
      hostPath:
        path: /host/data
```

In this example, we're mounting a hostPath volume into the container at /app/data.

- Save and Apply the Manifest: Save your my-pod.yaml file. To create the Pod, apply the configuration to your Kubernetes cluster using the kubectl apply command:

```
kubectl apply -f my-pod.yaml
```

Kubernetes will create the Pod based on the configuration you provided.

- Verify the Pod: You can check the status of the Pod by running:

```
kubectl get pods
```

You should see your Pod listed with its current status.

- Inspect Logs: To view the logs generated by the container(s) within the Pod, you can use the following command:

```
kubectl logs my-pod
```

Replace my-pod with the name of your Pod.

Interact with the Pod: You can interact with the Pod by executing commands inside it. For example, to open a shell in the container, use the following command:

```
kubectl exec -it my-pod -- /bin/bash
```

Replace /bin/bash with the appropriate shell for your container.

That's it! You've configured and deployed a Kubernetes Pod. You can further manage and monitor your Pod using various kubectl commands and Kubernetes resources like Services, Deployments, and StatefulSets, depending on your application's requirements.

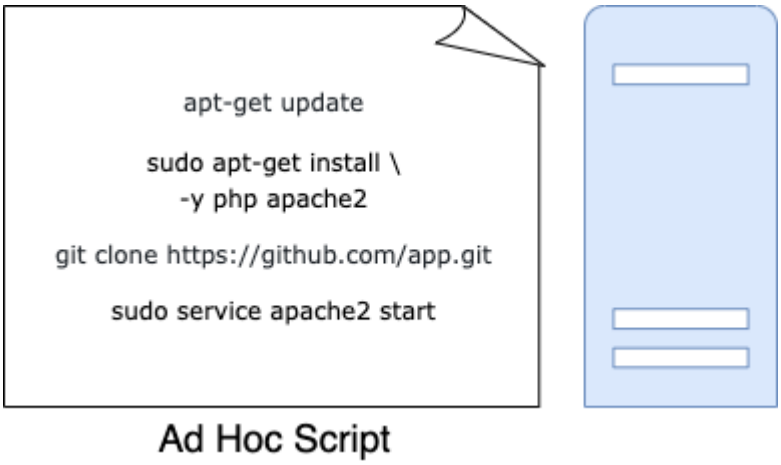
Infrastructure as Code Overview

Insight 1. What is IaC

The idea behind infrastructure as code (IAC) is that you write and execute code to define, deploy, update, and destroy your infrastructure. This represents an important shift in mindset in which you treat all aspects of operations as software—even those aspects that represent hardware (e.g., setting up physical servers). In fact, a key insight of DevOps is that you can manage almost everything in code, including servers, databases, networks, log files, application configuration, documentation, automated tests, deployment processes, and so on.

Insight 2. Ad Hoc Scripts

The most straightforward approach to automating anything is to write an ad hoc script. You take whatever task you were doing manually, break it down into discrete steps, use your favorite scripting language (e.g., Bash, Ruby, Python) to define each of those steps in code, and execute that script on your server.



Ad Hoc Script

For example, here is a Bash script called setup-webserver.sh that configures a web server by installing dependencies, checking out some code from a Git repo, and firing up an Apache web server:

```
# Update the apt-get cache
sudo apt-get update

# Install PHP and Apache
sudo apt-get install -y php apache2

# Copy the code from the repository
sudo git clone https://github.com/brikis98/php-app.git /var/www/html/app

# Start Apache
sudo service apache2 start
```

The great thing about ad hoc scripts is that you can use popular, general-purpose programming languages and you can write the code however you want. The terrible thing about ad hoc scripts is that you can use popular, general-purpose programming languages and you can write the code however you want.

Whereas tools that are purpose-built for IAC provide concise APIs for accomplishing complicated tasks, if you're using a general-purpose programming language, you need to write completely custom code for every task. Moreover, tools designed for IAC usually enforce a particular structure for your code, whereas with a general-purpose programming language, each developer will use their own style and do something different. Neither of these problems is a big deal for an eight-line script that installs Apache, but it gets messy if you try to use ad hoc scripts to manage dozens of servers, databases, load balancers, network configurations, and so on.

If you've ever had to maintain a large repository of Bash scripts, you know that it almost always devolves into a mess of unmaintainable spaghetti code. Ad hoc scripts are great for small, one-off tasks, but if you're going to be managing all of your infrastructure as code, then you should use an IaC tool that is purpose-built for the job.

Insight 3. Configuration Management Tools

Chef, Puppet, Ansible, and SaltStack are all configuration management tools, which means that they are designed to install and manage software on existing servers. For example, here is an Ansible Role called web-server.yml that configures the same Apache web server as the setup-webserver.sh script:

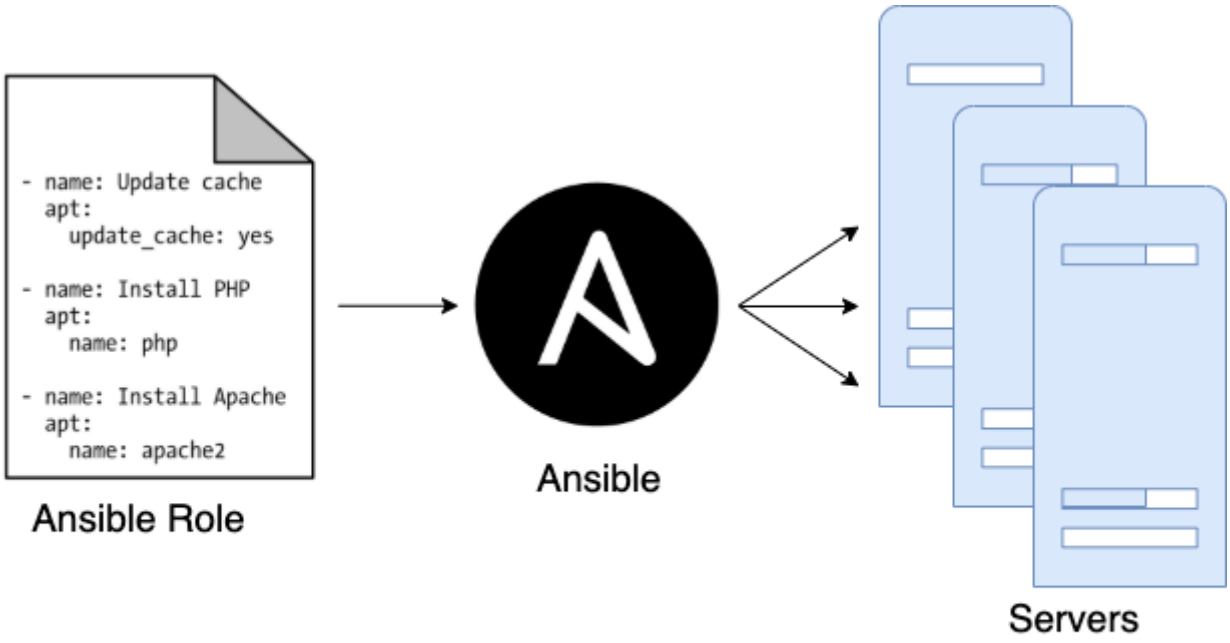
```
- name: Update the apt-get cache
  apt:
    update_cache: yes

- name: Install PHP
  apt:
    name: php

- name: Install Apache
  apt:
    name: apache2

- name: Copy the code from the repository
  git: repo=https://github.com/brikis98/php-app.git dest=/var/www/html/app

- name: Start Apache
  service: name=apache2 state=started enabled=yes
```

The code looks similar to the Bash script, but using a tool like Ansible offers a number of advantages:

- **Coding conventions**: Ansible enforces a consistent, predictable structure, including documentation, file layout, clearly named parameters, secrets management, and so on. While every developer organizes their ad hoc scripts in a different way, most configuration management tools come with a set of conventions that makes it easier to navigate the code.
- **Idempotence**: Writing an ad hoc script that works once isn't too difficult; writing an ad hoc script that works correctly even if you run it over and over again is a lot more difficult. Every time you go to create a folder in your script, you need to remember to check whether that folder already exists; every time you add a line of configuration to a file, you need to check that line doesn't already exist; every time you want to run an app, you need to check that the app isn't already running.

Code that works correctly no matter how many times you run it is called idempotent code. To make the Bash script from the previous section idempotent, you'd need to add many lines of code, including lots of if-statements. Most Ansible functions, on the other hand, are idempotent by default. For example, the web-server.yml Ansible role will install Apache only if it isn't installed already and will try to start the Apache web server only if it isn't running already.

- **Distribution**: Ad hoc scripts are designed to run on a single, local machine. Ansible and other configuration management tools are designed specifically for managing large numbers of remote servers



For example, to apply the web-server.yml role to five servers, you first create a file called hosts that contains the IP addresses of those servers:

```
[webservers]
11.11.11.11
11.11.11.12
11.11.11.13
11.11.11.14
11.11.11.15
```

Next, you define the following Ansible playbook:

```
- hosts: webservers
  roles:
  - webserver
```

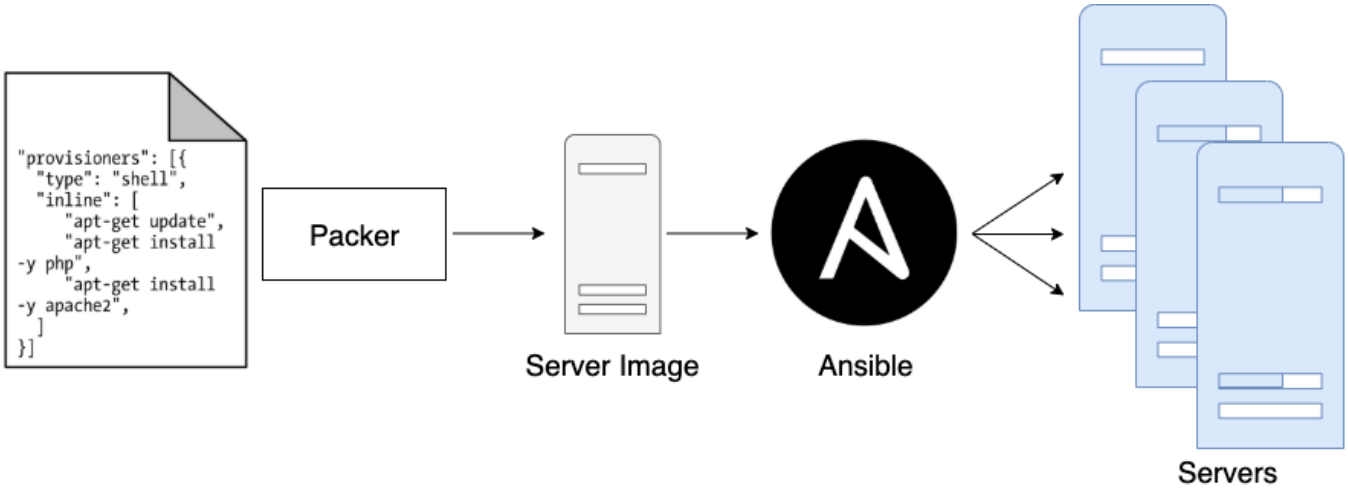
Finally, you execute the playbook as follows:

```
ansible-playbook playbook.yml
```

This instructs Ansible to configure all five servers in parallel. Alternatively, by setting a parameter called serial in the playbook, you can do a rolling deployment, which updates the servers in batches. For example, setting serial to 2 directs Ansible to update two of the servers at a time, until all five are done. Duplicating any of this logic in an ad hoc script would take dozens or even hundreds of lines of code.

Insight 4. Server Templating Tools

An alternative to configuration management that has been growing in popularity recently are server templating tools such as Docker, Packer, and Vagrant. Instead of launching a bunch of servers and configuring them by running the same code on each one, the idea behind server templating tools is to create an image of a server that captures a fully self-contained "snapshot" of the operating system (OS), the software, the files, and all other relevant details. You can then use some other IaC tool to install that image on all of your servers.



Insight 5. Orchestration Tools

Server templating tools are great for creating VMs and containers, but how do you actually manage them? For most real-world use cases, you'll need a way to do the following:

- Deploy VMs and containers, making efficient use of your hardware.
- Roll out updates to an existing fleet of VMs and containers using strategies such as rolling deployment, blue-green deployment, and canary deployment.
- Monitor the health of your VMs and containers and automatically replace unhealthy ones (auto healing).
- Scale the number of VMs and containers up or down in response to load (auto scaling).
- Distribute traffic across your VMs and containers (load balancing).
- Allow your VMs and containers to find and talk to one another over the network (service discovery).

Handling these tasks is the realm of orchestration tools such as Kubernetes, Marathon/Mesos, Amazon Elastic Container Service (Amazon ECS), Docker Swarm, and Nomad. For example, Kubernetes allows you to define how to manage your Docker containers as code. You first deploy a Kubernetes cluster, which is a group of servers that Kubernetes will manage and use to run your Docker containers. Most major cloud providers have native support for deploying managed Kubernetes clusters, such as Amazon Elastic Container Service for Kubernetes (Amazon EKS), Google Kubernetes Engine (GKE), and Azure Kubernetes Service (AKS).

Once you have a working cluster, you can define how to run your Docker container as code in a YAML file:

```yaml
apiVersion: apps/v1

# Use a Deployment to deploy multiple replicas of your Docker
# container(s) and to declaratively roll out updates to them
kind: Deployment

# Metadata about this Deployment, including its name
metadata:
  name: example-app

# The specification that configures this Deployment
spec:
  # This tells the Deployment how to find your container(s)
  selector:
    matchLabels:
      app: example-app

  # This tells the Deployment to run three replicas of your
  # Docker container(s)
  replicas: 3

  # Specifies how to update the Deployment. Here, we
  # configure a rolling update.
  strategy:
    rollingUpdate:
      maxSurge: 3
      maxUnavailable: 0
    type: RollingUpdate

  # This is the template for what container(s) to deploy
  template:

    # The metadata for these container(s), including labels
    metadata:
      labels:
        app: example-app

    # The specification for your container(s)
    spec:
      containers:

        # Run Apache listening on port 80
        - name: example-app
          image: httpd:2.4.39
          ports:
            - containerPort: 80
```

This file instructs Kubernetes to create a Deployment, which is a declarative way to define:

- One or more Docker containers to run together. This group of containers is called a Pod. The Pod defined in the preceding code contains a single Docker container that runs Apache.
- The settings for each Docker container in the Pod. The Pod in the preceding code configures Apache to listen on port 80.

- How many copies (aka replicas) of the Pod to run in your cluster. The preceding code configures three replicas. Kubernetes automatically figures out where in your cluster to deploy each Pod, using a scheduling algorithm to pick the optimal servers in terms of high availability (e.g., try to run each Pod on a separate server so a single server crash doesn't take down your app), resources (e.g., pick servers that have available the ports, CPU, memory, and other resources required by your containers), performance (e.g., try to pick servers with the least load and fewest containers on them), and so on. Kubernetes also constantly monitors the cluster to ensure that there are always three replicas running, automatically replacing any Pods that crash or stop responding.
- How to deploy updates. When deploying a new version of the Docker container, the preceding code rolls out three new replicas, waits for them to be healthy, and then undeploy the three old replicas.

That's a lot of power in just a few lines of YAML! You run kubectl apply -f example-app.yml to instruct Kubernetes to deploy your app. You can then make changes to the YAML file and run kubectl apply again to roll out the updates.

Insight 6. Provisioning Tools

Whereas configuration management, server templating, and orchestration tools define the code that runs on each server, provisioning tools such as Terraform, CloudFormation, and OpenStack Heat are responsible for creating the servers themselves. In fact, you can use provisioning tools to not only create servers, but also databases, caches, load balancers, queues, monitoring, subnet configurations, firewall settings, routing rules, Secure Sockets Layer (SSL) certificates, and almost every other aspect of your infrastructure.

For example, the following code deploys a web server using Terraform:

```
resource "aws_instance" "app" {
  instance_type     = "t2.micro"
  availability_zone = "us-east-2a"
  ami               = "ami-0c55b159cbfafe1f0"

  user_data = <<-EOF
              #!/bin/bash
              sudo service apache2 start
              EOF
}
```

where

- **ami**: This parameter specifies the ID of an AMI to deploy on the server. You could set this parameter to the ID of an AMI built from the web-server.json Packer template in the previous section, which has PHP, Apache, and the application source code.
- **user_data**: This is a Bash script that executes when the web server is booting. The preceding code uses this script to boot up Apache.

In other words, this code shows you provisioning and server templating working together, which is a common pattern in immutable infrastructure.

Insight 7. The Benefits of IaC

When your infrastructure is defined as code, you are able to use a wide variety of software engineering practices to dramatically improve your software delivery process, including the following:

- **Self-service**: If your infrastructure is defined in code, the entire deployment process can be automated, and developers can kick off their own deployments whenever necessary.
- **Speed and safety**: If the deployment process is automated, it will be significantly faster, since a computer can carry out the deployment steps far faster than a person; and safer, given that an automated process will be more consistent, more repeatable, and not prone to manual error.
- **Documentation**: Instead of the state of your infrastructure being locked away in a single sysadmin's head, you can represent the state of your infrastructure in source files that anyone can read.
- **Version control**: You can store your IaC source files in version control, which means that the entire history of your infrastructure is now captured in the commit log.
- **Validation**: If the state of your infrastructure is defined in code, for every single change, you can perform a code review, run a suite of automated tests, and pass the code through static analysis tools—all practices that are known to significantly reduce the chance of defects.
- **Reuse**: You can package your infrastructure into reusable modules, so that instead of doing every deployment for every product in every environment from scratch, you can build on top of known, documented, battle-tested pieces.
- **Happiness**: There is one other very important, and often overlooked, reason for why you should use IaC: happiness. IaC offers a better alternative that allows computers to do what they do best (automation) and developers to do what they do best (coding).

AWS Overview

Amazon Web Services (AWS) is a comprehensive and widely used cloud computing platform offered by Amazon. It provides a vast array of services that enable businesses and individuals to build and deploy various types of applications, manage data, and scale infrastructure without the need for upfront investments in hardware or extensive setup.

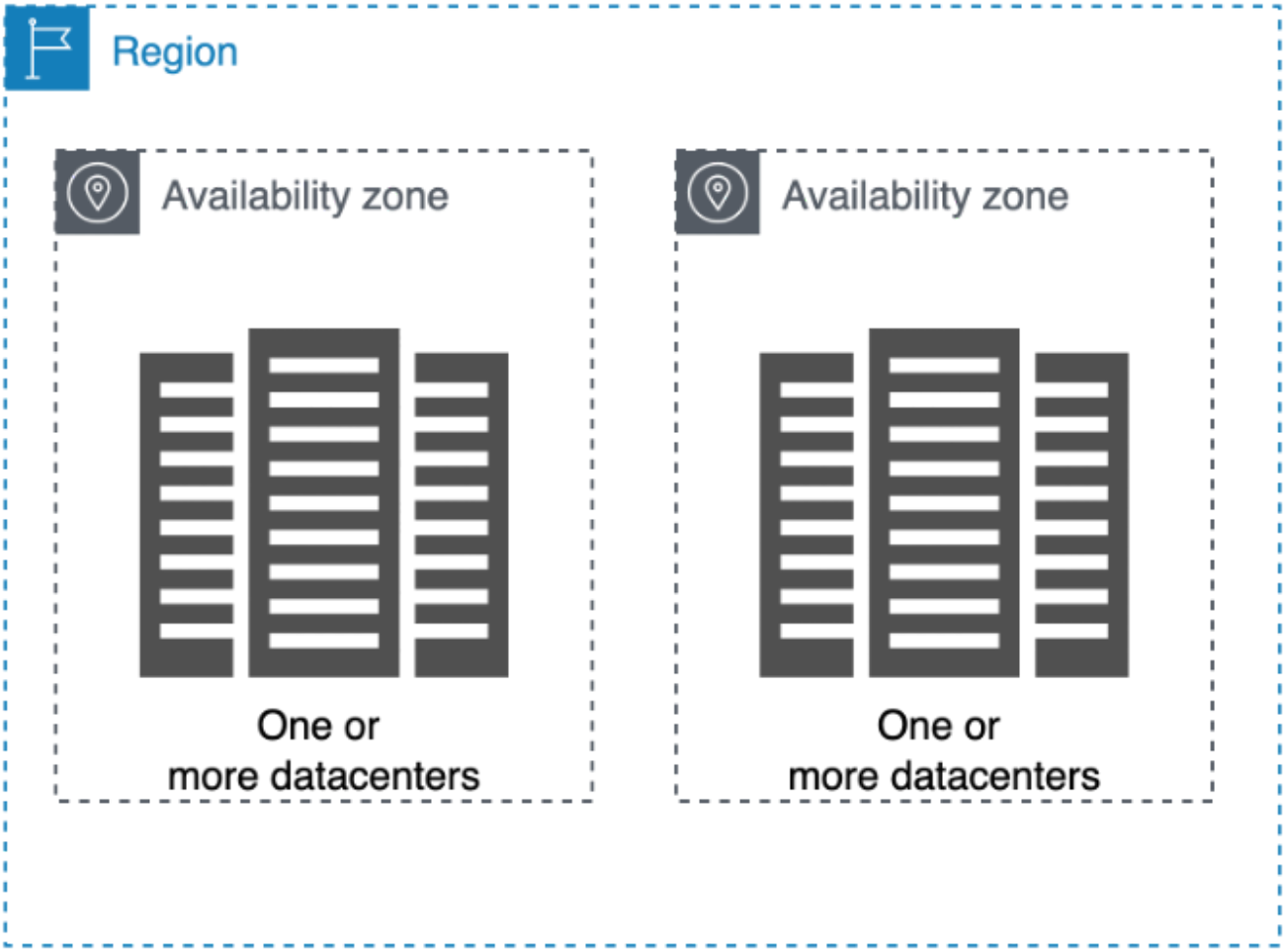
Insight 1. AWS Cloud Architecture

AWS operates through a global network of data centers, called Availability Zones (AZs), located in different regions around the world. Each AZ consists of one or more data centers, and these AZs are isolated from

each other to provide fault tolerance and high availability.

Regions:

Availability Zone:



Insight 2. Key Concepts and Services

Compute Services

- Amazon EC2 (Elastic Compute Cloud): Provides resizable compute capacity in the cloud, allowing you to run virtual servers (instances) on-demand.
- AWS Lambda: Enables serverless computing by allowing you to run code in response to events without provisioning or managing servers.

Storage Services

- Amazon S3 (Simple Storage Service): Offers scalable object storage for data backup, archiving, and web hosting.
- Amazon EBS (Elastic Block Store): Provides persistent block storage for EC2 instances.
- Amazon Glacier: Designed for long-term storage of data archives at a low cost.

Database Services

- Amazon RDS (Relational Database Service): Managed relational databases supporting various engines like MySQL, PostgreSQL, Oracle, etc.
- Amazon DynamoDB: Fully managed NoSQL database service for applications that need seamless and quick scaling.

Networking Services

- Amazon VPC (Virtual Private Cloud): Lets you create isolated networks within the AWS cloud.
- Amazon Route 53: Provides domain name system (DNS) web service and domain registration.
- AWS Direct Connect: Establishes a dedicated network connection between on-premises infrastructure and AWS.

Security and Identity

- AWS Identity and Access Management (IAM): Manages user identities and permissions within AWS.
- Amazon Cognito: Provides identity management for web and mobile applications.
- AWS Key Management Service (KMS): Enables you to create and control the encryption keys used to secure your data.

Management and Monitoring

- AWS Management Console: Provides secure login using your AWS or IAM account credentials.
- AWS CLI: Controls multiple AWS services from the command line and automate them through scripts.
- AWS SDK: Enables access to AWS services using most development languages as part of an application.
- AWS CloudFormation: Automates the provisioning and management of AWS resources using templates.
- Amazon CloudWatch: Monitors resources and applications, providing metrics and alerts.
- AWS Systems Manager: Helps you manage and automate operational tasks across your AWS infrastructure.

Developer Tools

- AWS CodePipeline, AWS CodeBuild, and AWS CodeDeploy: Enable continuous integration, continuous delivery, and deployment of applications.

AI and Machine Learning

- Amazon SageMaker: Provides tools for building, training, and deploying machine learning models.
- Amazon Rekognition: Offers image and video analysis using deep learning.

Analytics

- Amazon Redshift: Data warehousing service for running complex queries on large datasets.
- Amazon EMR (Elastic MapReduce): Provides a managed Hadoop framework for big data processing.

Content Delivery and CDN

- Amazon CloudFront: Content delivery network service for securely delivering data, videos, applications, and APIs.

These are just a subset of the extensive services AWS offers. Businesses can leverage these services to create scalable, reliable, and cost-effective solutions tailored to their specific needs. AWS also provides tools for monitoring usage, controlling costs, and ensuring security.

Insight 3. Shared Responsibility Model

The AWS Shared Responsibility Model is a framework that outlines the division of security and compliance responsibilities between Amazon Web Services (AWS) and its customers. It helps clarify which security aspects are managed by AWS and which aspects are the responsibility of the customer. This model is crucial for ensuring a secure and compliant environment in the cloud.

The model is generally divided into two main categories:

AWS Responsibility

AWS is responsible for the security "of" the cloud infrastructure. This includes the underlying physical data centers, networking, hardware, and foundational services that AWS provides. AWS takes measures to ensure that its infrastructure is resilient, redundant, and protected against common threats. This includes aspects like:

- Physical security of data centers and infrastructure
- Network architecture and protection
- Hardware maintenance and management
- Availability of services
- Patching and updating of infrastructure components

Customer Responsibility

Customers are responsible for the security "in" the cloud. This means that customers are responsible for securing the data, applications, and configurations they deploy on AWS services. This includes aspects like:

- Data protection and encryption
- Identity and access management (IAM)
- Configuration and management of virtual machines and containers
- Security groups and firewall rules
- Application-level security and code

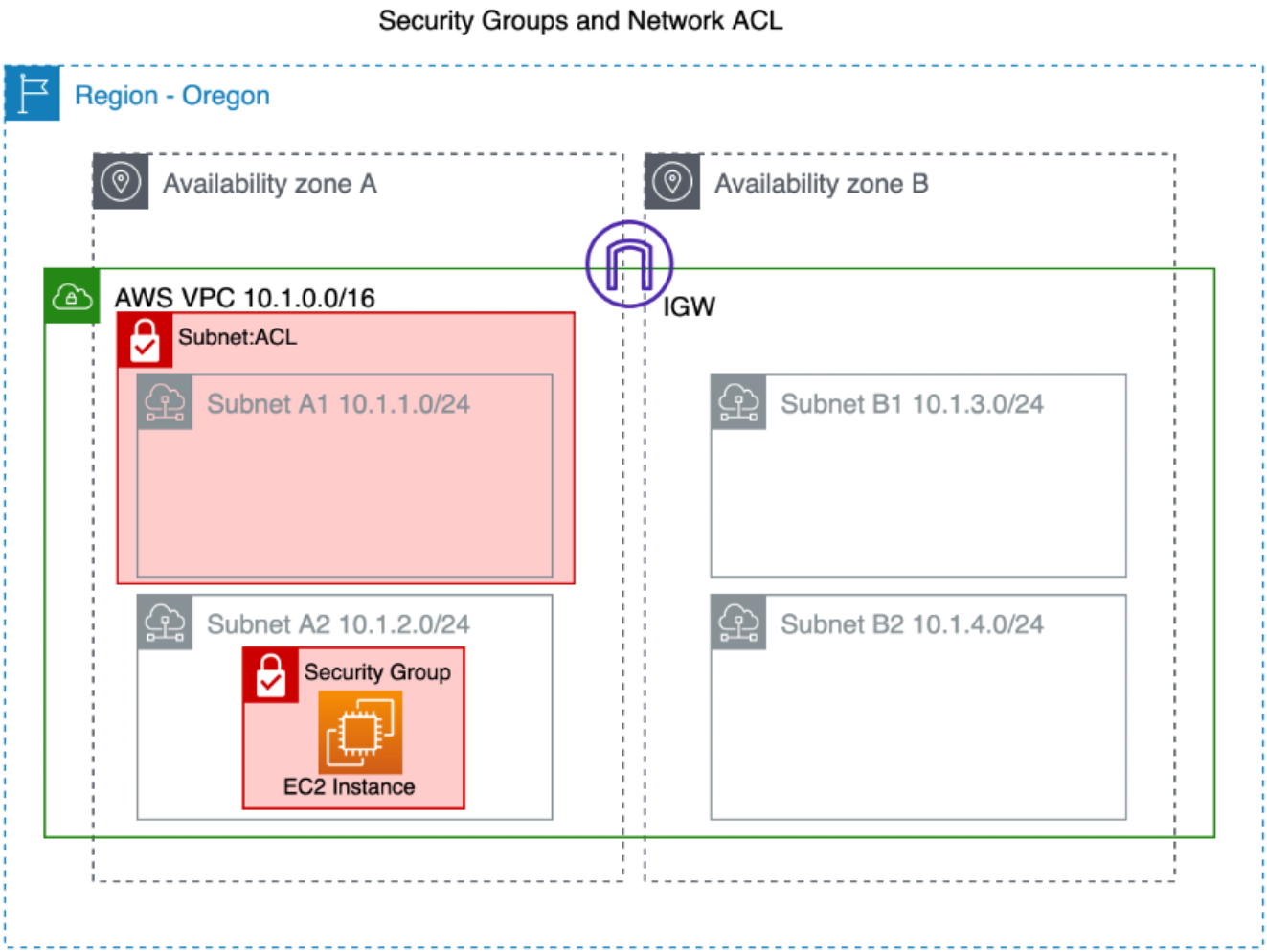
- Compliance with regulations and industry standards

In summary, while AWS ensures the security and reliability of its infrastructure, customers are responsible for securing the applications, data, and workloads they run on AWS services. The Shared Responsibility Model emphasizes collaboration between AWS and its customers to create a secure and compliant cloud environment. By understanding and following this model, customers can take appropriate measures to implement security best practices and protect their assets in the cloud.

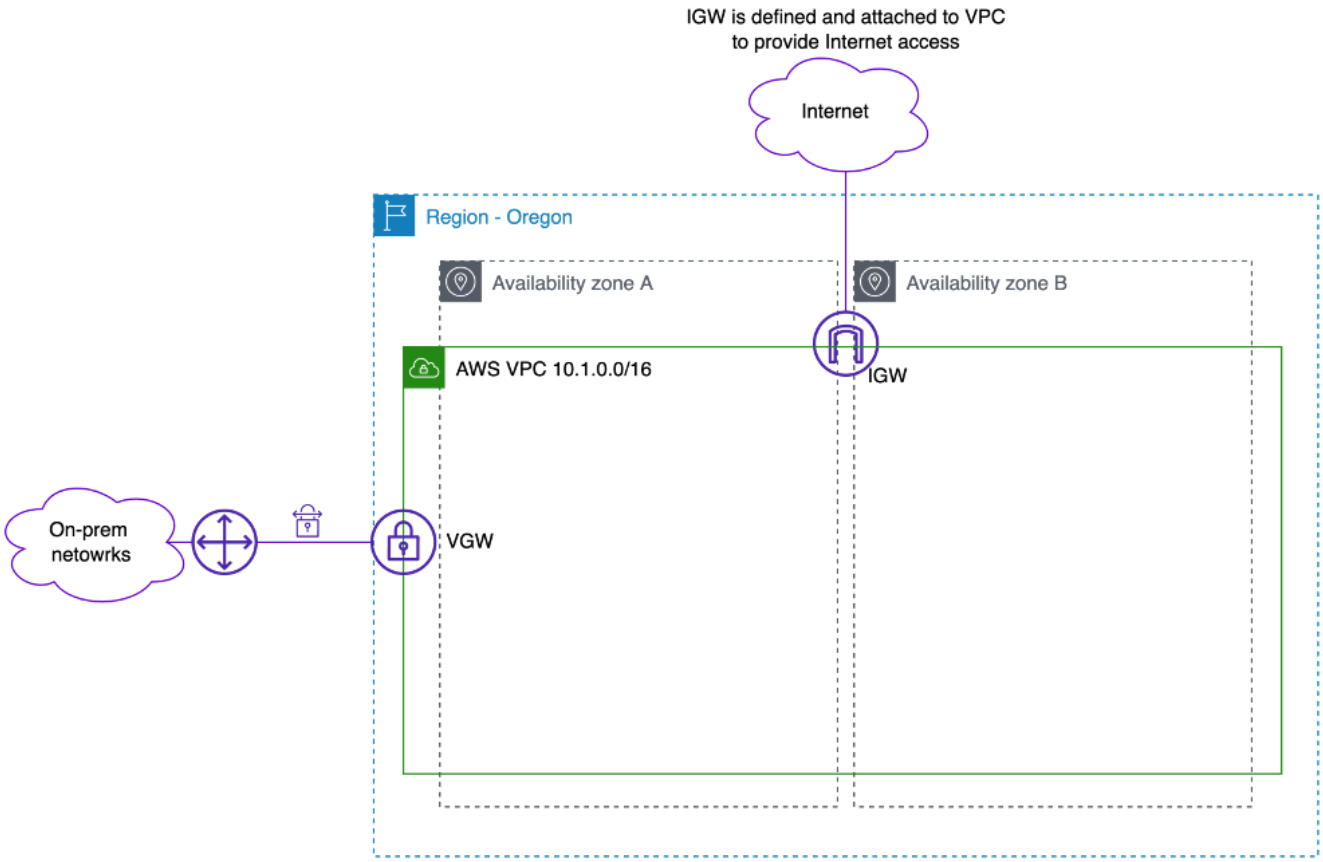
Insight 4. VPC and Networking

Amazon Virtual Private Cloud (Amazon VPC) is a networking service provided by Amazon Web Services (AWS) that allows you to create and manage a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network environment. This virtual network closely resembles a traditional network that you might operate in your own data center, but with the benefits of cloud scalability and flexibility.

VPC, Subnet and Security Group:



VPC, Internet and VPN Gateways:



Key features and components of AWS VPC include:

- Isolation and Control: With VPC, you can define your own virtual network topology, including IP address ranges, subnets, and route tables. This enables you to isolate your resources and control the network traffic flow within your VPC.
- Subnets: VPC allows you to divide your virtual network into subnets, which are smaller segments of the VPC's IP address range. Subnets can be public (accessible from the internet) or private (not accessible from the internet). You can use these subnets to deploy different types of resources based on their security and accessibility requirements.
- Security Groups and Network Access Control Lists (NACLs): Security Groups and NACLs are used to control inbound and outbound traffic to and from your AWS resources. Security Groups are associated with individual resources and operate at the instance level, while NACLs are associated with subnets and operate at the subnet level.
- Internet Gateway (IGW): An Internet Gateway allows resources in your VPC to access the internet and enables inbound traffic from the internet to reach your resources if you configure your subnets to be public.
- Virtual Private Gateway (VGW): A Virtual Private Gateway enables your VPC to establish secure VPN connections to your on-premises network, extending your data center to the AWS Cloud.
- Peering: VPC peering allows you to connect two VPCs together, enabling resources in different VPCs to communicate with each other as if they were on the same network.
- VPN Connections: VPC supports creating encrypted VPN connections to securely connect your on-premises data center to your VPC.

- Direct Connect: AWS Direct Connect provides dedicated network connections from your on-premises data center to AWS, bypassing the public internet.
- Elastic IP Addresses: Elastic IP addresses are static, public IP addresses that you can allocate to your instances, providing a consistent way for your resources to be accessed from the internet.
- Route Tables: Route tables define the traffic routes within your VPC. You can associate different route tables with different subnets to control the flow of traffic.
- Network Address Translation (NAT) Gateways/Instances: NAT gateways or NAT instances allow instances in private subnets to initiate outbound traffic to the internet while preventing inbound traffic from reaching them.

AWS VPC provides a robust and flexible networking foundation for deploying various types of applications in a secure and isolated manner within the AWS Cloud environment. It allows you to design and configure your network infrastructure according to your specific requirements and best practices.

Insight 5. AWS Security services

AWS offers a comprehensive set of security services and features to help you protect your data, applications, and infrastructure in the cloud. These services are designed to help you build a secure and compliant environment. Here are some key AWS security services:

- AWS Identity and Access Management (IAM): IAM enables you to manage access to AWS services and resources securely. You can create and manage users, groups, and roles, and assign permissions to control who can access your resources and what actions they can perform.
- Amazon VPC (Virtual Private Cloud): As described earlier, VPC allows you to isolate and control your network resources in a virtual network environment. You can set up network security groups and access control lists (ACLs) to control inbound and outbound traffic.
- AWS Web Application Firewall (WAF): WAF protects your web applications from common web exploits and attacks by allowing you to define rules that control access to your application and mitigate malicious traffic.
- Amazon GuardDuty: GuardDuty is a threat detection service that continuously monitors for malicious activity and unauthorized behavior in your AWS accounts and workloads.
- Amazon Inspector: Inspector helps you assess the security and compliance of your applications by automatically analyzing your resources for vulnerabilities and deviations from best practices.
- AWS Key Management Service (KMS): KMS enables you to create and manage encryption keys to protect your data. You can use KMS to encrypt data at rest and data in transit.
- AWS CloudTrail: CloudTrail logs and monitors API activity in your AWS account, providing you with a record of actions taken by users, roles, or services. This helps with auditing, compliance, and troubleshooting.
- Amazon Macie: Macie uses machine learning to automatically discover, classify, and protect sensitive data in AWS, helping you maintain data privacy and compliance.

- AWS Shield: Shield provides protection against Distributed Denial of Service (DDoS) attacks, helping to keep your applications and resources available and accessible.
- Amazon Inspector: Inspector assesses the security and compliance of your applications by analyzing your AWS resources for vulnerabilities and deviations from best practices.
- AWS Organizations: Organizations enables you to centrally manage and govern multiple AWS accounts, helping you implement security and compliance standards across your organization.
- Amazon Detective: Detective helps you analyze, investigate, and visualize security data, making it easier to identify potential security issues and understand the root causes.

These are just a few examples of the many security services offered by AWS. Again, it's important to note that security in AWS is a shared responsibility model, where AWS is responsible for the security of the cloud infrastructure, while you are responsible for securing your applications, data, and workloads that you run on AWS. This means that you need to take advantage of these security services and implement best practices to ensure a secure and compliant environment for your applications.

LAB : AWS Introduction

Objectives

- Get familiar with Virtual Private Cloud infrastructures AWS VPC
- Learn VPC key networking constructs
- Create on-demand Virtual Machines with AWS EC2

Task 0. First Connection

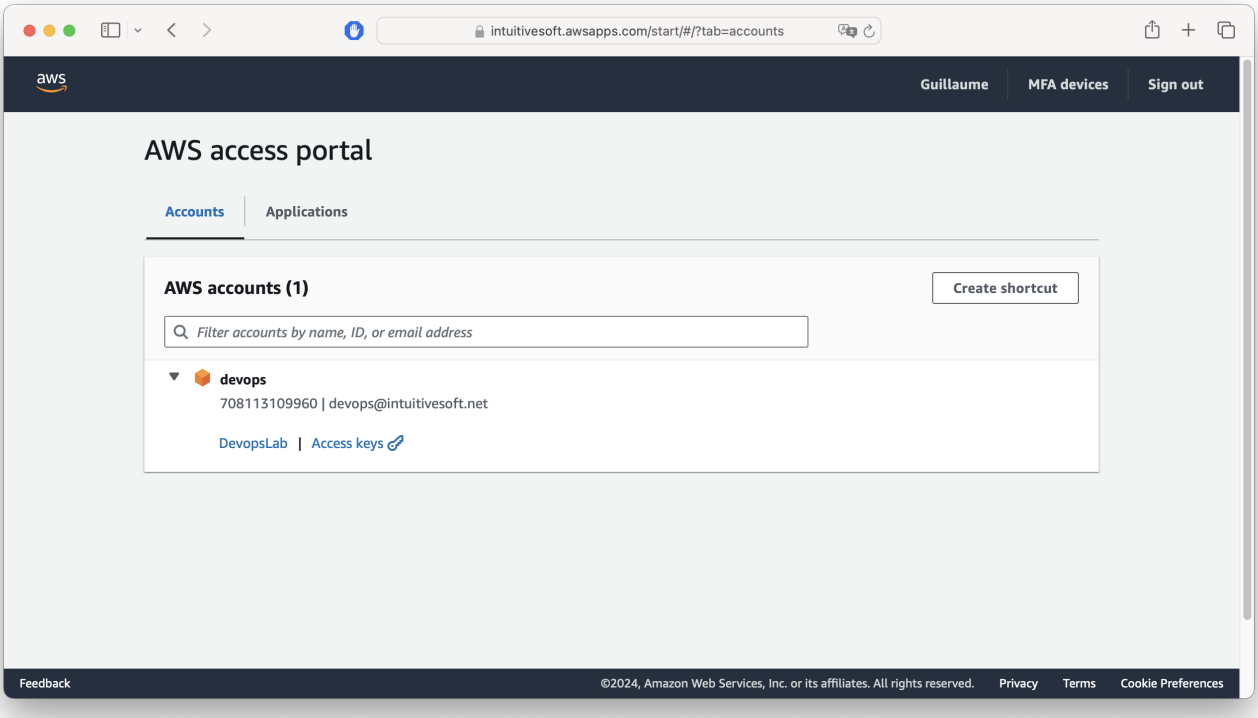
Refer to documentation AWS first connect

Task 1. Connect to AWS Management console

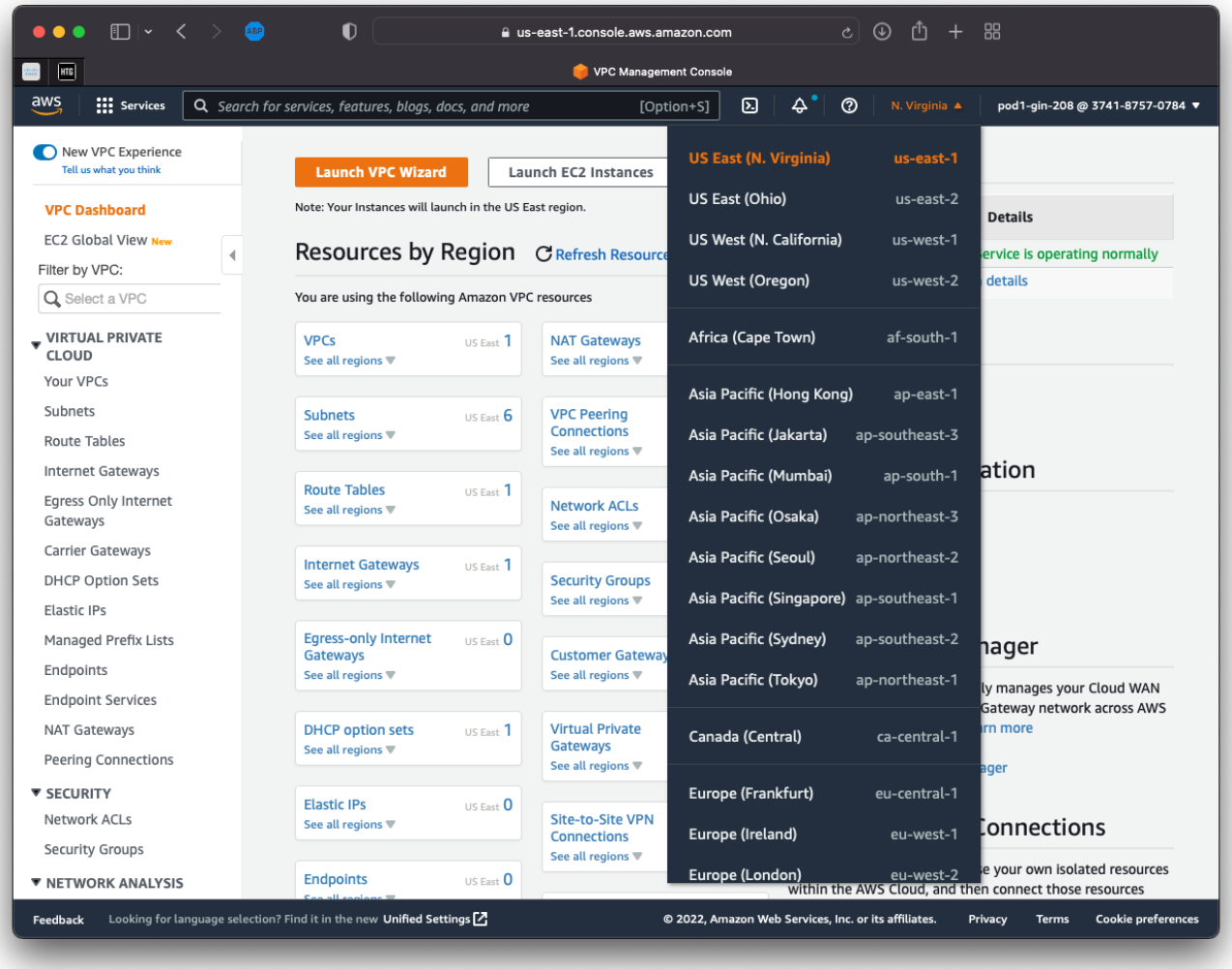
Connect to the AWS management console using the training SSO portal :

https://intuitivesoft.awsapps.com/start#/

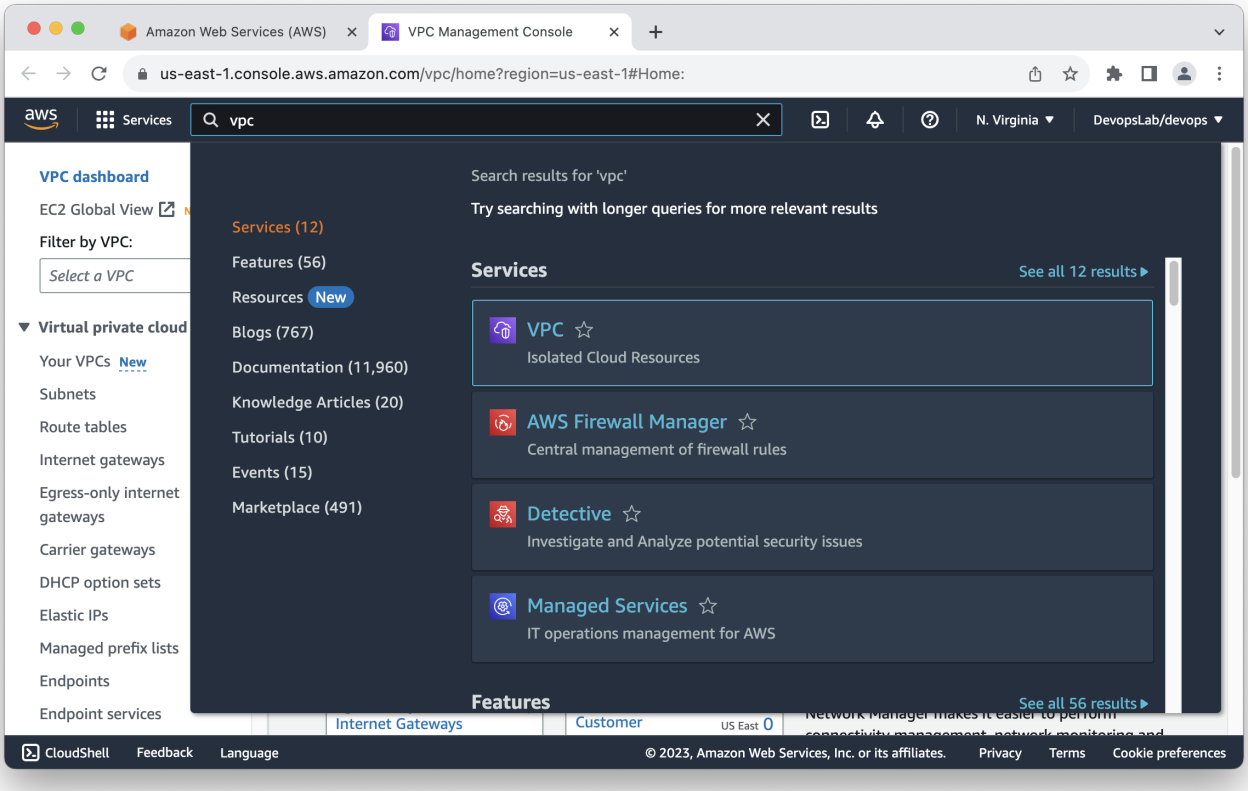
Select your training AWS account and click the link `DevopsLab` to get access to the account main page.



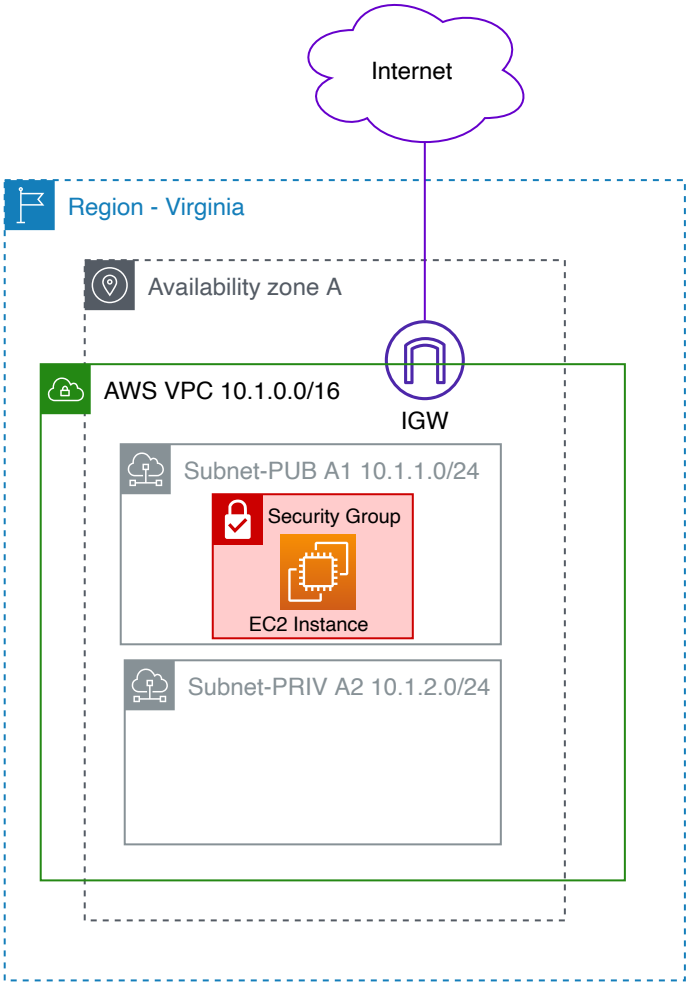
AWS Cloud infrastructure is segmented in administrative regions Once connected verify and switch to the lab region : *us-east-1*



The most convenient way to navigate in between AWS services is to use the research bar on the top side and type the name of the service, or feature inside a service, you'd like to access.



Insight 2. Lab Target



During this lab you will learn how to create the base infrastructure for a virtual data center ready to host your applications. You will get familiar with AWS main constructs for Cloud Networking, Security and Compute.

This is a shared lab environment. During this training you will be instructed to create,edit,delete AWS resources. AWS provides a `tag` system to help you filter through resources with you tag convention. Use it as a best practice to differentiate from other lab environment.

Insight 3. User Management

AWS Identity and Access Management (IAM) is a web service that helps you securely control access to AWS resources. You use IAM to control who is authenticated (signed in) and authorized (has permissions) to use resources. IAM policies are defined as a set of permissions that grant Create Read Update Delete (CRUD) operation on any AWS resources. You can then assign policies to an IAM role. IAM roles are *temporary* identity a `user` or an AWS service can `assume` (present itself as) to gain privileges on the resources defined by the role policies.

For example :

The user `John` has the role `admin` that has the `AdministratorAccess` policy attached, it gives full operations to any resource on AWS. User, `William` has the role `readonly` and `EC2-admin`. It has readonly rights on all AWS resources, he will need to `assume` the role `EC2-admin` if he wants to create an EC2 instance.

Roles are a really handy way to restrain and control user and application access to the strict minimum AWS resources and actions. It's a key component for micro-services and cloud native/serverless application development.

Task 4. Visualize your user permissions

Your user can't modify IAM policies but you can visualize the policy and permissions attached to your user.

- Task 4.1 -

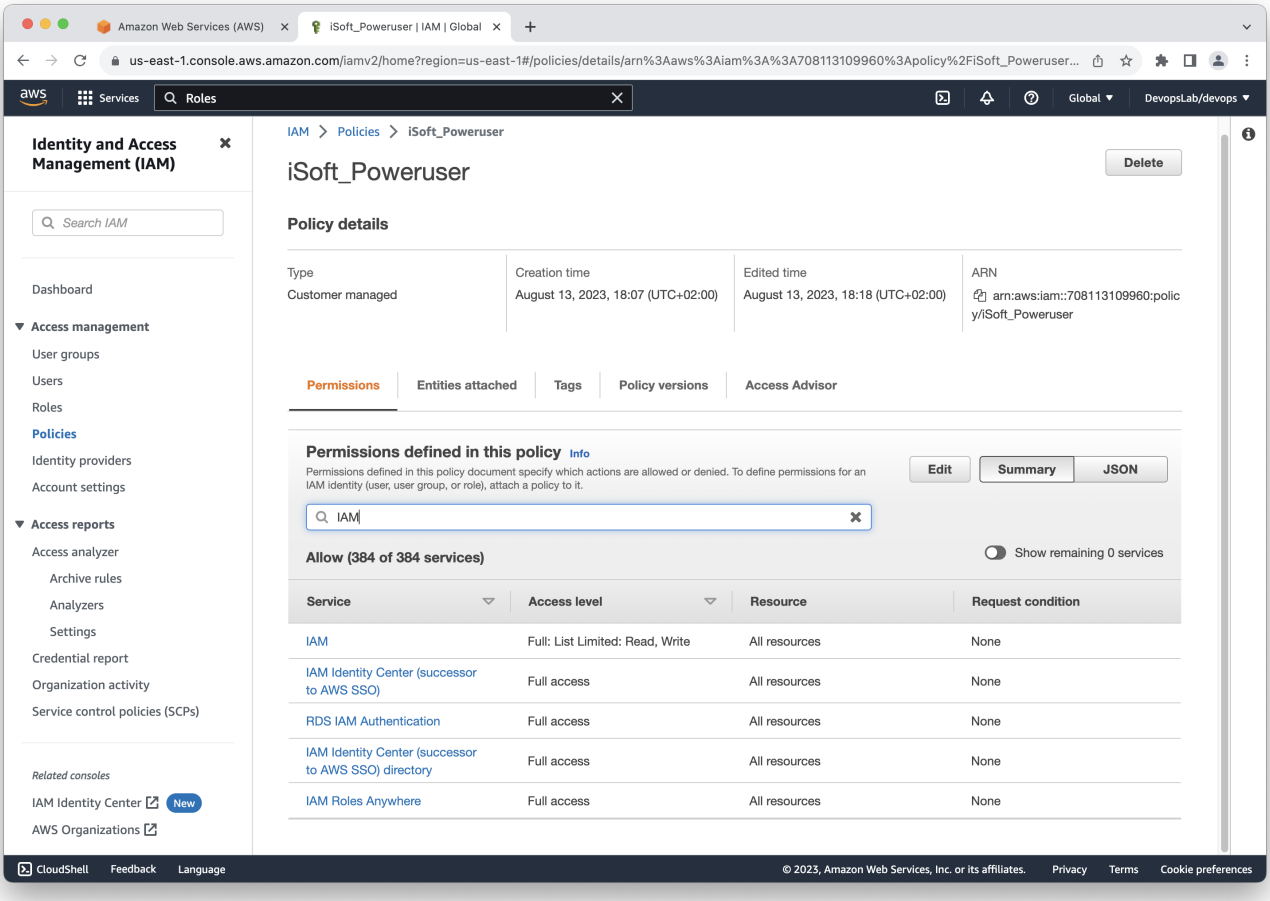
Navigate to the `IAM roles` definitions using the web search console or using the following link : https://us-east-1.console.aws.amazon.com/iamv2/home?region=us-east-1#/roles

- Task 4.2 -

Your user is assigned to role `AWSReservedSSO_DevopsLab_XXXXXXXXXXXXXXXX` navigate to the role and visualize the permissions attached to your role.

- Task 4.3 -

You are a PowerUser you have `Full access` to most of AWS resources but some restrictions on some services such as `IAM`. Filter the permissions attached to service `IAM`, you have limited `Read,Write`



Task 5. AWS Policy

The graphical interface representation is useful to have an overview. You can switch to the declarative representation of the policy by clicking `JSON`

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "NotAction": [
                "iam:*",
                "organizations:*",
                "account:*"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:CreateServiceLinkedRole",
                "iam:DeleteServiceLinkedRole",
                "iam:Get*",
                "iam:List*",
                "organizations:DescribeOrganization",
                "account:ListRegions",
                "account:GetAccountInformation"
            ],
            "Resource": "*"
        }
    ]
}
```

An AWS policy is defined by the following arguments :

Version: Specifies the version of the policy language being used.

Statement: An array of statements that define the permissions granted by the policy.

Effect: "Allow/Deny" right on the API actions specified on the `Action` block.

Action/NotAction: Specifies a list of actions that are explicitly not allowed. In this case, it's denying all actions under the "iam", "organizations", and "account" namespaces.

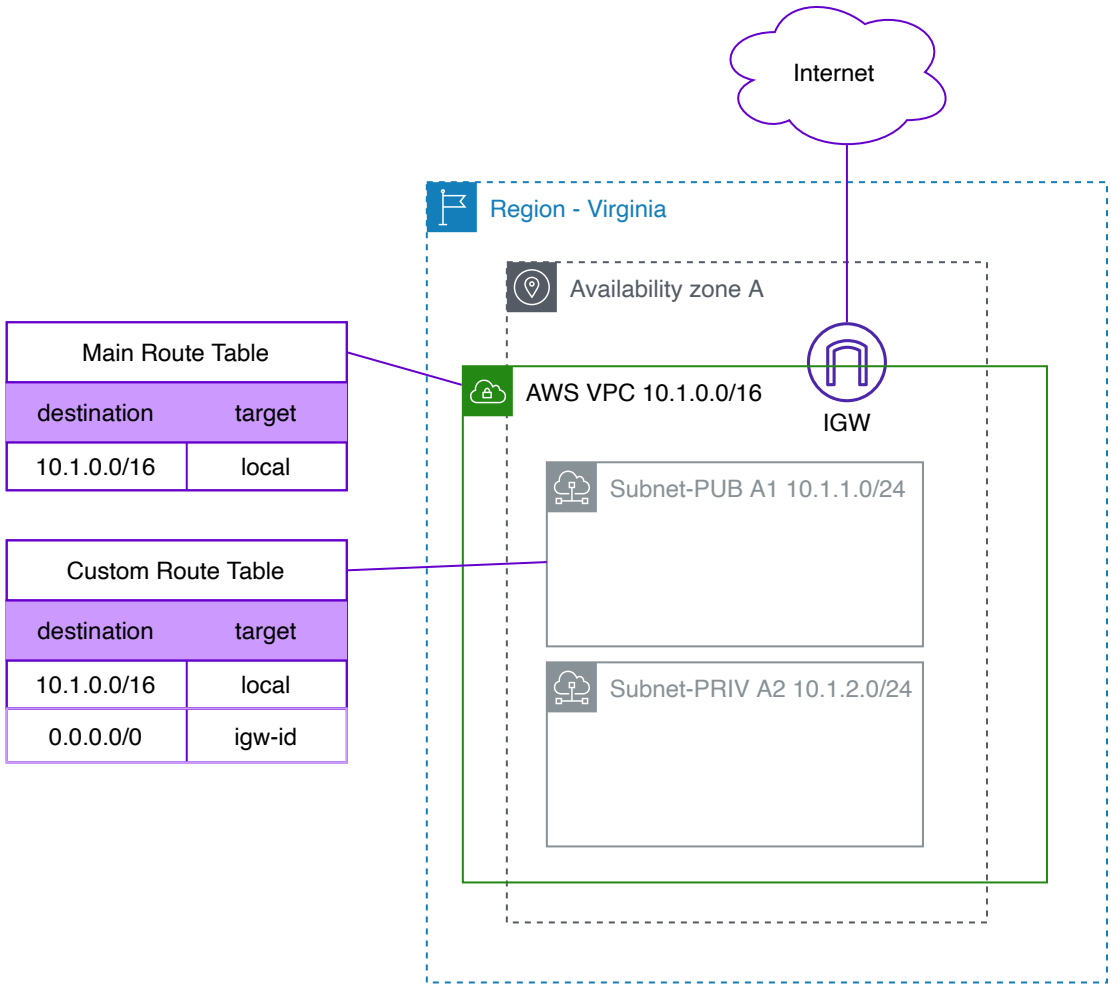
Resource: Specify the exact resource on which the policy applies. The asterisk (*) as the resource value indicates that this denial applies to all resources.

Insight 6. Virtual Private Cloud

Amazon Virtual Private Cloud (Amazon VPC) enables you to launch AWS resources into a virtual network that you've defined. This virtual network closely resembles a traditional network that you'd operate in your own data center, with the benefits of using the scalable infrastructure of AWS. Unless configured otherwise, every VPC is isolated from each other with no resource overlapping and connectivity.

AWS documentation : Amazon VPC

In the following activity you will be instructed to create a VPC with the following architecture :



Task 7. Create your first VPC

- Task 7.1 -

Navigate to the VPC service using the web search console or using the following link : https://us-east-1.console.aws.amazon.com/vpc/home?region=us-east-1#Home:

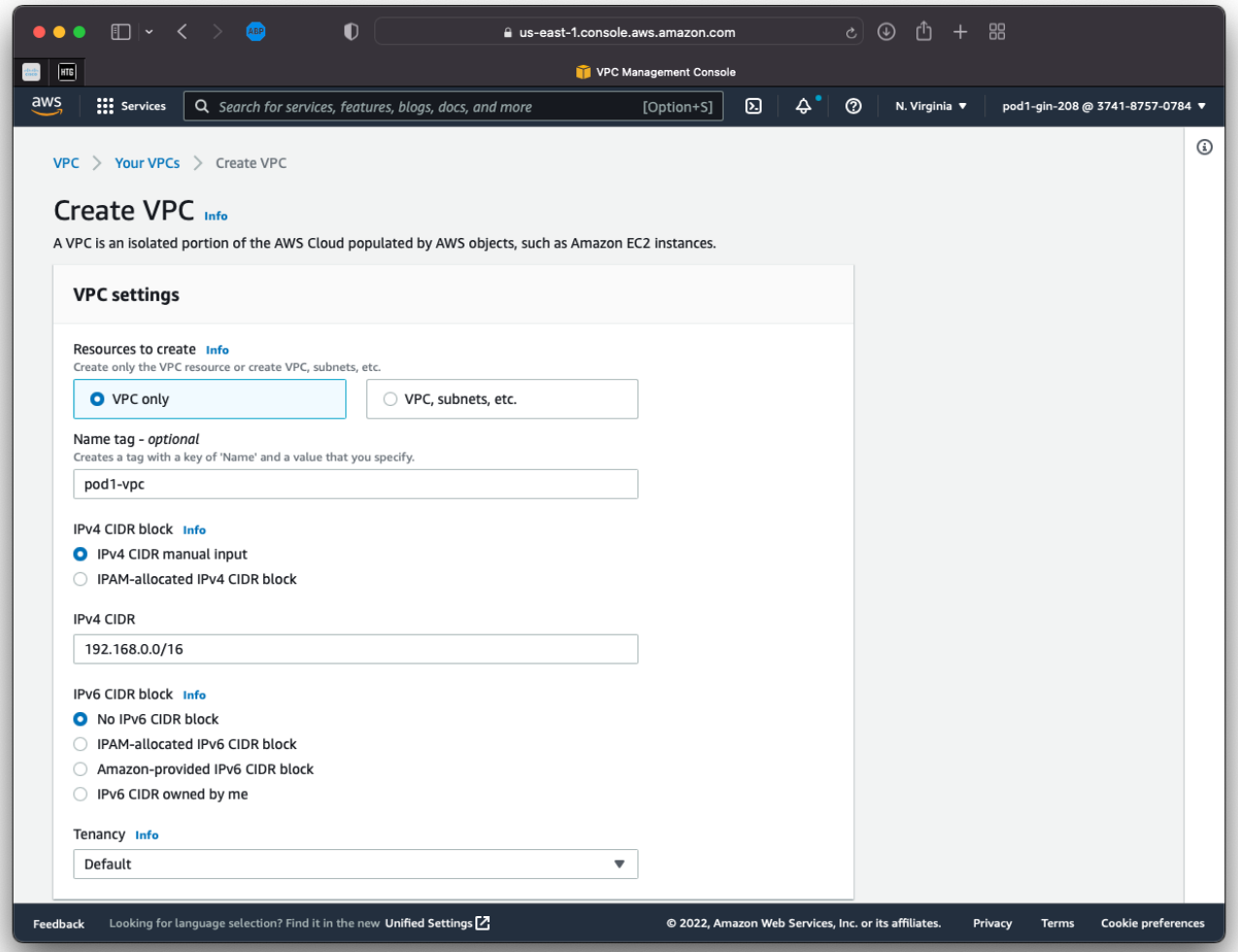
- Task 7.2 -

This lab is meant to be didactic, AWS offers configuration abstraction GUI wizards to help you automate creation of resources. For the purpose of this lab you will **manually** create each VPC objects.

Launch AWS VPC wizard

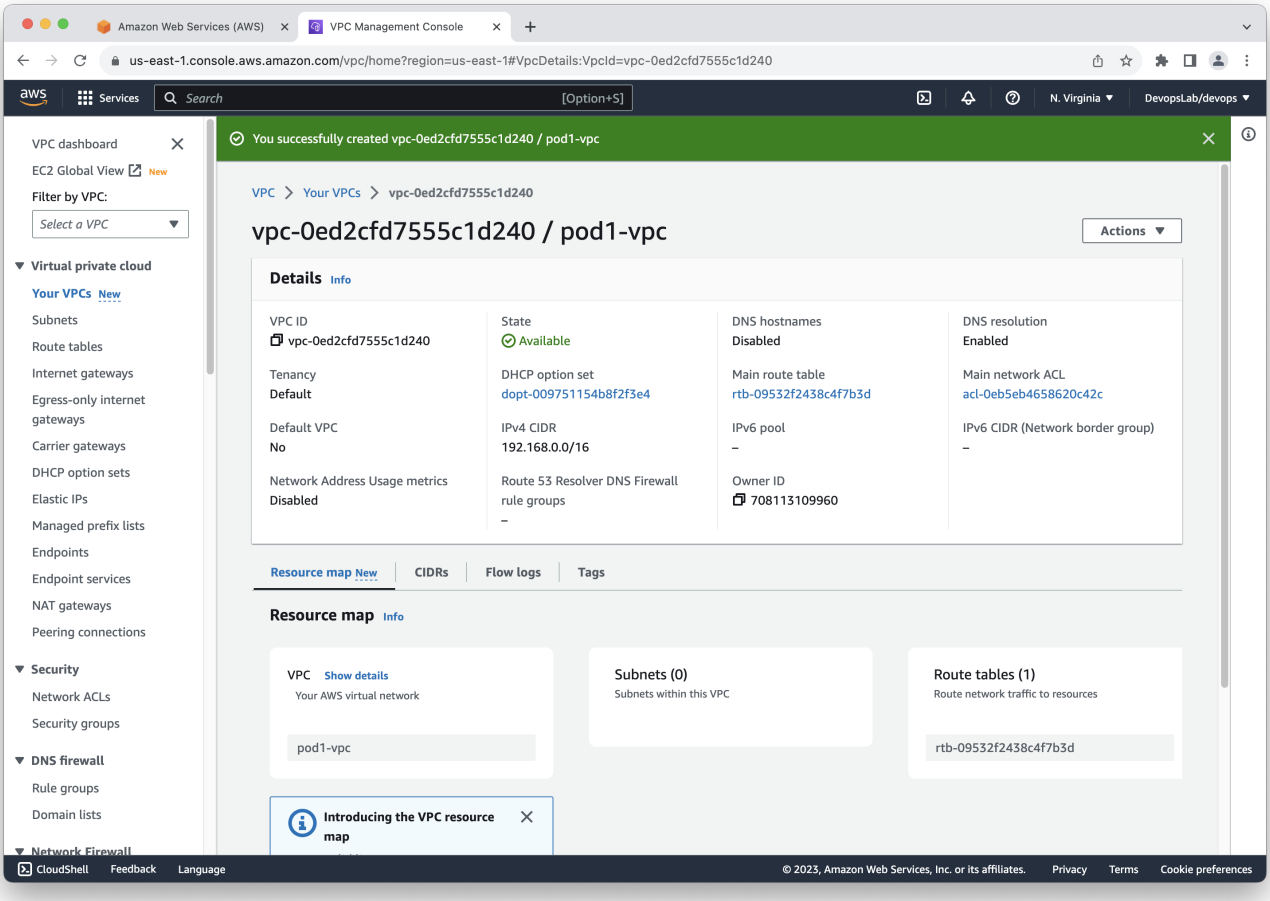
Create a `VPC Only` VPC. With the private CIDR of your choice.

Do not forget to give it a distinctive name and add some tags that could be useful to filter your resources later on like a `lab-id` with a unique identifier for all the resources you are going to create during the lab.



Task 8. Visualize your VPC

Congrats you have a virtual DataCenter with a dummy router connected to nothing ! Take some time to inspect your VPC and the objects created by AWS automatically.

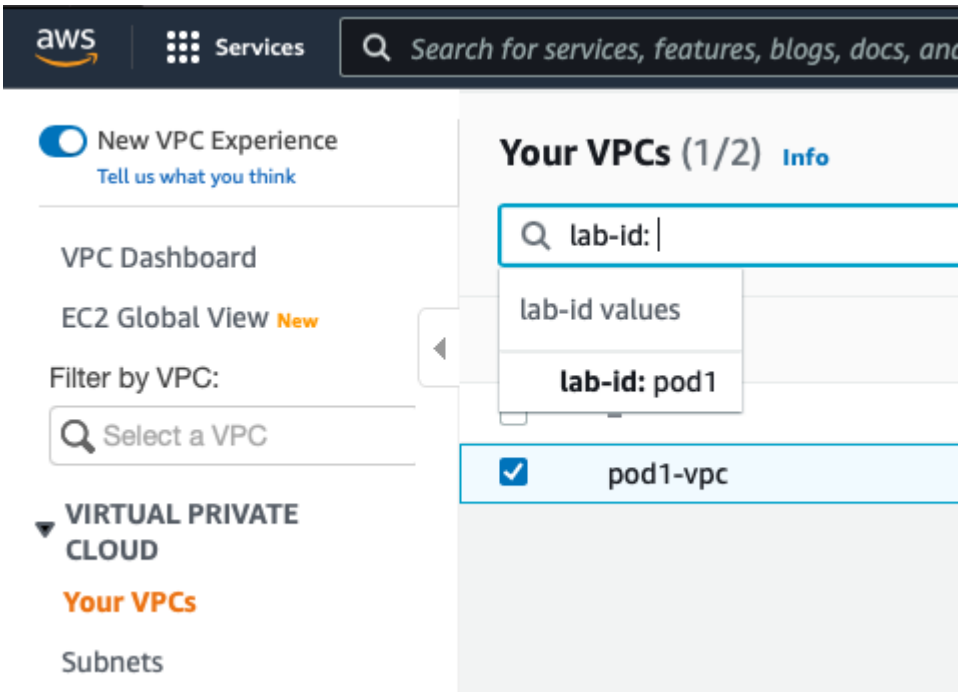


- DHCP options set : DHCP options for the resources requesting IP addresses.
- Main routing table : a dummy router. If not specify differently subnets will be associated with that routing table by default.
- Main network ACL : a dummy Firewall.

AWS documentation : Custom Route Tables

AWS only applied a `tag` to the main element - the VPC - you can manually add the `lab-id` tag and a `name` to those resources to make it easier to find them later on.

Note : When looking at a list of objects in AWS console, you can always filter the resources by `id` or `tags` key/value:



Task 9. Create a Public and Private subnet.

The notion of private and public subnet in AWS is similar to the traditional datacenter design with a DMZ and a private infrastructure. Resources on a DMZ can have a public IPv4 address or can be publicly reached from internet using predefined static NAT (Network Address Translation) entry in conjunction of firewall rules. On the other side, resources on a private network should not be reachable from internet and can only access internet via a router performing dynamic NAPT (Network Address Port Translation). This is enforced by firewall rules.

On a classical - and may be outdated - three tier application, frontend, backend, database, the network connectivity should look like the following :

- You want to serve customer on a Publicly reachable network (most likely behind a load balancer).
- Have your backend service running a on a private network with firewall policies allowing the frontend to reach the backend. The backend could reach internet for package update, and applications download.
- Finally your database on a dedicated network with firewall policies to allow only connections from your backend.

AWS documentation : VPCs and Subnets

AWS documentation : VPC with Public and Private Subnets (NAT)

In AWS a public network is a subnet associated to a routing table that have a route to an Internet Gateway (covered later on).

To be able to give a network access to your EC2 instances in your VPC you need to have at least a subnet.

- Task 9.1 -

Navigate to the `VPC > Subnet` section, or use the following link https://us-east-1.console.aws.amazon.com/vpc/home?region=us-east-1#subnets:

- Task 9.2 -

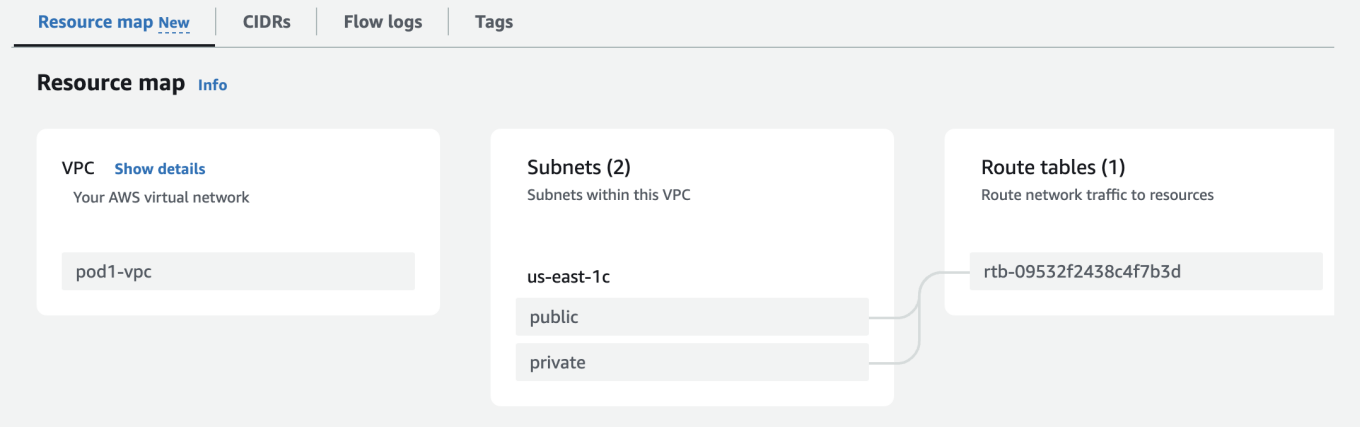
In the AWS `VPC resource` create two new `subnet` from the VPC CIDR one `public` and an other `private`.

Note : You do not have to select an Availability Zone (AZ), AWS will select one for you. AWS define an AZ as an isolated location within a Region. Each AWS datacenter in a region has several L2 switch fabrics isolated from each other. You can decide on which fabric your subnet/resources will be connected/running. For a production setup, it's a best practice to create several subnets in different AZ to prevent downtimes in case of network failure.

- Task 9.3 -

Click on your VPC you should now visualize the two new subnet created.

You now have in your Virtual Data Center, two switches fabrics connected to the previously AWS created router.



You could already instantiate Virtual Machines (EC2) to your VPC but they would be confined in your Datacenter as they are not connected to the WAN.

Insight 10. WAN connectivity

For your VM to have a WAN access you must specify the type of connections :

NAT Gateway (v4) Egress Only Internet Gateway (IPv6) : The construct to build a private network with Internet access.

It's the classical NAT construct. You need to create a NAT Gateway and *pay* a Public IPv4 address per availability zones you are using. The gateway will do the network/port translation of your VMs towards internet but prevent any incoming connections.

Internet Gateway (IGW) : The construct to build a DMZ.

This is the historical construct of AWS, VMs used to be publicly available. Only one can be allocated per VPC and it's available across AZ. Technically speaking the Internet GW is doing 1:1 NAT for your instances. At provisioning time AWS allocates a Public IPv4 address for your instance and keeps a : translation. The public IPv4 address is allocated but not assigned to your instance, AWS can re-allocate another IP if your reboot your VM. You could also buy a static public IP from AWS if needed AWS Documentation : Elastic IP.

If a subnet has a routing table entry to the Internet gateway it's referred as a `public network`.

Task 11. Add an Internet Gateway to your VPC

- Task 11.1 -

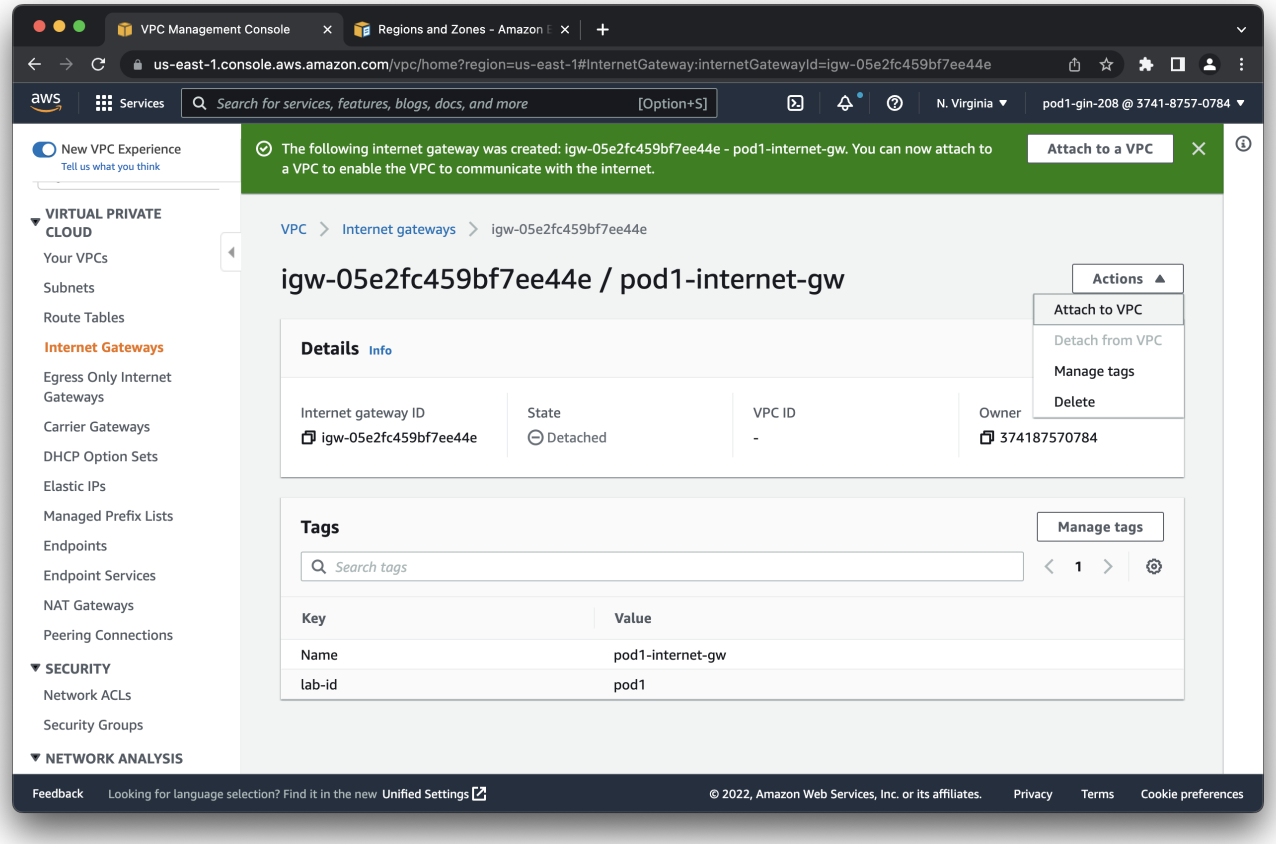
Navigate to the `VPC > Internet Gateway` , or use the following link https://us-east-1.console.aws.amazon.com/vpc/home?region=us-east-1#igws:

- Task 11.2 -

Create an internet gateway.

Note : Do not forget add `name` and custom `tags`

After you created an IGW check that its state is `detached`. Fix it by attaching it to your VPC



Task 12. Routing

- Task 12.1 -

Navigate to the `VPC > Route Tables` , or use the following link https://us-east-1.console.aws.amazon.com/vpc/home?region=us-east-1#RouteTables:

- Task 12.2 -

Create a new `route table` for your VPC.

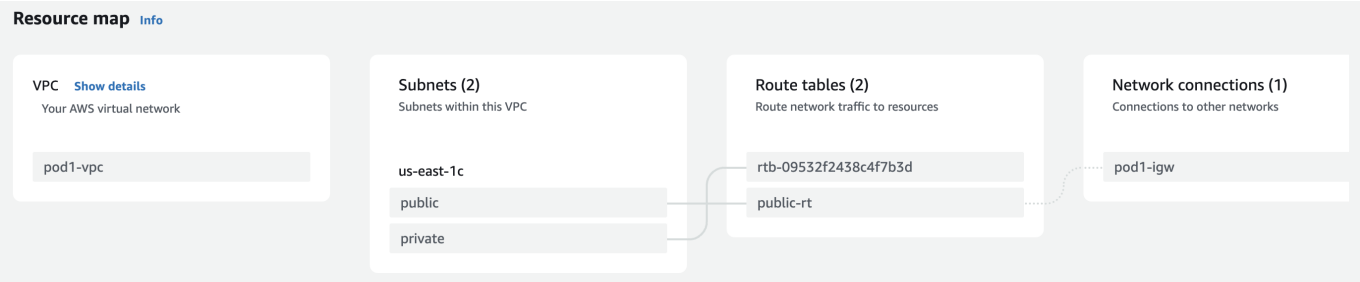
- Task 12.3 -

Attach the route table to your public subnet by editing the route table `Subnet associations`

- Task 12.4 -

Edit the routes of the new `route table` to allow local traffic to access internet.

Expected `VPC Resource map` configuration :

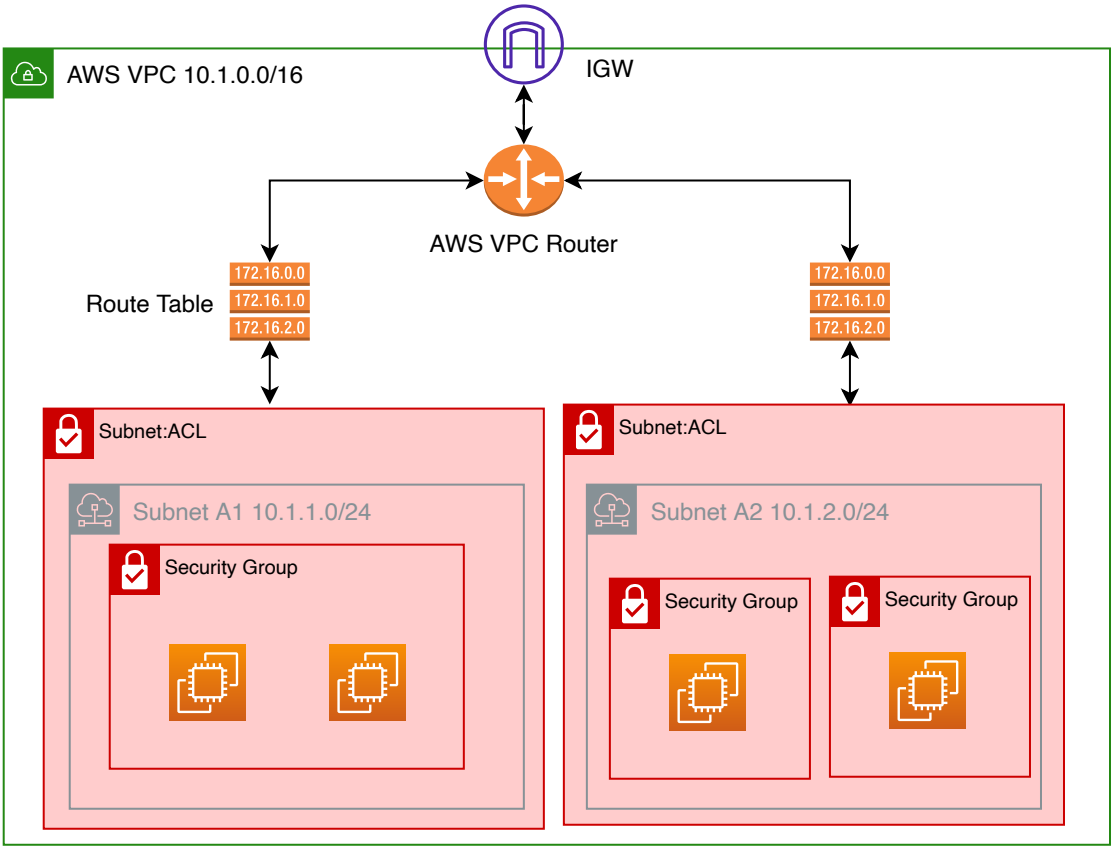


Task 13. Security

Network security is a mandatory requirement of any application/data center design. AWS offers two main constructs to enforce network security : `security groups` and `ACL` They both implement the same logic : port/protocol/IP restriction at a different level.

`ACL` : defines rules in between subnets.

`port-security`: defines rules at the VM or container level.



It's a mandatory construct when you want to host - the next unicorn startup - a web service, but it's also quite handy when you want to SSH your VM without having to rely on a VPN or SSH Bastion.

- Task 13.1 -

Navigate to the `VPC > Security Groups` , or use the following link https://us-east-1.console.aws.amazon.com/vpc/home?region=us-east-1#SecurityGroups:

- Task 13.2 -

Create a `VPC security group`

- Task 13.3 -

Add a rule to allow SSH connections from Internet.

Insight 14. EC2

Amazon Elastic Compute Cloud (Amazon EC2) provides scalable computing capacity in the Amazon Web Services (AWS) Cloud. In practice it's a Web service that will instantiate Virtual Machines. Historically, VM are running on a custom Xen Hypervisor. Since 2017 AWS is using a custom version of KVM called Nitro to support/optimize performances on a wider range of hardware.

An EC2 instance is principally defined by two main attributes an image/software definition called an `AMI` and a server/hardware specification called an `Image type`.

The final pricing of an EC2 instance is mainly driven by the sum of :

- The EC2 instance Image Type price-hour
- The persistent storage used per GB-month
- The volume of network data per GB-month

AWS Documentation : Amazon EC2

Insight 15. AMI

An Amazon Machine Image (AMI) is the snapshot of the VM you want to use.

When creating an EC2 instance you can browse the catalogue of Public AMI created by AWS, vendors, and the community. You can also create your own AMI and export them to AWS.

Note that the Open Source tool Hashicorp Packer can be helpful to create/automate such resources.

Insight 16. Image types

For each application need there's a different Image type available in AWS. Databases tends to favor Memory optimized instances while AI/ML application will required an Instance with accelerated NVIDIA GPU.

Image types define the number of vCPU, memory, instance storage and bandwidth allocated to your EC2 instance.

AWS Documentation : Full list of EC2 instance types

Each image type has a cost per hour (or per second for burst applications) associated that vary depending the region.

AWS Documentation : Price per EC2 Image types

For the purpose of the lab we recommend the usage of the generic, and perfectly balanced as all the things should be, `t2` **or** `t3` **instances**.

Task 17. SSH Key-Pair

Before creating your first EC2 instance lets create the last requirement : a SSH key-pair. On AWS, by default, you cannot access your compute instance using username and password: You need an SSH Key pair. You have the option to create one on AWS or to import yours. For linux instances, AWS will add a pub entry in `~/.ssh/authorized_keys`. If you create your own key you can create a `RSA` or `ED25519` key in `.pem` format.

- Task 17.1 -

Navigate to the `EC2 > Key Pairs` , or use the following link https://us-east-1.console.aws.amazon.com/ec2/home?region=us-east-1#KeyPairs:

- Task 17.2 -

Create or import a new SSH-Key.

This is up to you to decide how many key-pairs you want to create. **You can only download a created key-pair once from AWS** (so please store the key safely in your personnel folder). A single key-pair can be assigned to a EC2 instance at boot time, but noting prevents you to add more keys afterwards.

- Task 17.3 -

Install openssh-server

```
sudo apt update
sudo apt install -y openssh-server
mkdir ~/.ssh
```

- Task 17.4 -

Upload your SSH key to your lab environment and move it to your SSH folder (`~/.ssh`).

```
mv ~/files/*.pem ~/.ssh
```

- Task 17.5 -

SSH private keys **must** have permission READ (400) or READ/WRITE (600) to **your user only**.

Linux UGO Permissions

		User	Group	Others
Read	4	x		
Write	2	x		

	User	Group	Others
Execute	1		
Permission	6	0	0

Set permission on your SSH key to be usable

```
chmod 600 ~/.ssh/*.pem
```

Task 18. Create an EC2 instance

- Task 18.1 -

Navigate to the `EC2` , or use the following link https://us-east-1.console.aws.amazon.com/ec2/home?region=us-east-1#Instances:

- Task 18.2 -

From EC2 service select `Instances` then `Launch instances` wizard.

- Task 18.3 -

Take the time to browse the AMI catalogue

Note : AWS Linux 2, is based on RedHat Enterprise Linux. It comes with long-term support from Amazon and have handy packages to interact with AWS. For the purpose of the lab we recommend to use `ubuntu 22.04` as linux distribution, but feel free to try something else.

- Task 18.4 -

Take the time to browse the Instance types and check the associated cost.

For testing we recommend the flavor : `t2.nano`.

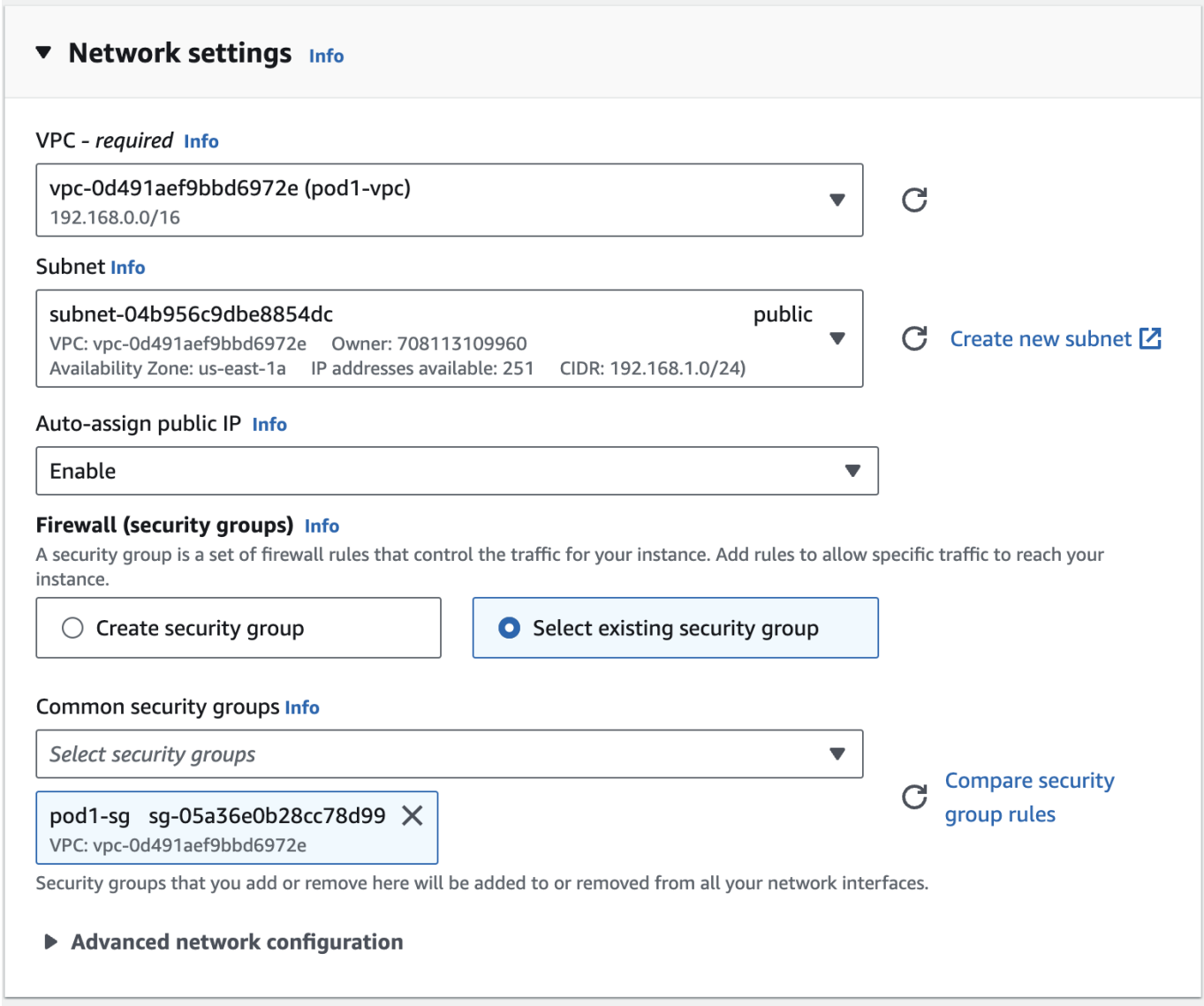
- Task 18.5 -

Specify your SSH Key-Pair to login the instance.

- Task 18.6 -

Edit the Network settings and select your VPC, Public Subnet and Security group previously created.

Enable the public IP addressing for AWS to allocate a temporary public IP to your VM.



Task 19. Connect to your EC2 instance

- Task 19.1 -

Once your instance is booted and `Instance state` is `Running` you can retrieve its public IP address from the instance list try to connect to it using SSH from your lab environment.

```
ssh -i ~/.ssh/YOUR_KEY.pem ubuntu@INSTANCE_PUBLIC_IP_ADDR
```

Expected output :

```
The authenticity of host 'REDACTED (REDACTED)' can't be established.
ECDSA key fingerprint is
SHA256:wbABDsF3ouneOAOB2o9bHuIGnzikZa2qn4HXrIUVaB0.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'REDACTED' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 22.04.2 LTS (GNU/Linux 5.19.0-1025-aws x86_64)

 [...]
```

```
ubuntu@ip-192-168-1-5:~$
```

Default user for linux AMI is based on AMI type :

- For Amazon Linux AMI, the user name is ec2-user.
- For a CentOS AMI, the user name is centos or ec2-user.
- For a Debian AMI, the user name is admin.
- For a Fedora AMI, the user name is fedora or ec2-user.
- For a RHEL AMI, the user name is ec2-user or root.
- For a SUSE AMI, the user name is ec2-user or root.
- For an Ubuntu AMI, the user name is ubuntu.
- For an Oracle AMI, the user name is ec2-user. Otherwise, check with the AMI provider.

- Task 19.2 -

You can exit your SSH session

```
exit
```

LAB: AWS CLI on Linux

The AWS Command Line Interface (AWS CLI) is an open source tool that enables you to interact with AWS services using commands in your command-line shell. With minimal configuration, the AWS CLI enables you to start running commands that implement functionality equivalent to that provided by the browser-based AWS Management Console from the command prompt in your terminal program. It's technically a wrapper around AWS REST API.

Automation tools such as Terraform or Ansible will use the configuration file of the AWS CLI to retrieve the access key when performing calls to AWS API.

Task 1. Install AWS CLI

The following commands needs to be run in your interactive lab terminal.

```
sudo apt update && sudo apt install -y unzip jq less groff mandoc
```

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
"awscliv2.zip"
```

```
unzip awscliv2.zip
```

```
sudo ./aws/install --update
```

Task 2. Test you AWS CLI installation

```
aws --version
```

You should have a result similar to:

```
aws-cli/2.13.9 Python/3.11.4 Linux/5.15.0-1040-aws exe/x86_64.ubuntu.20
prompt/off
```

Task 3. Retrieve your AWS secrets

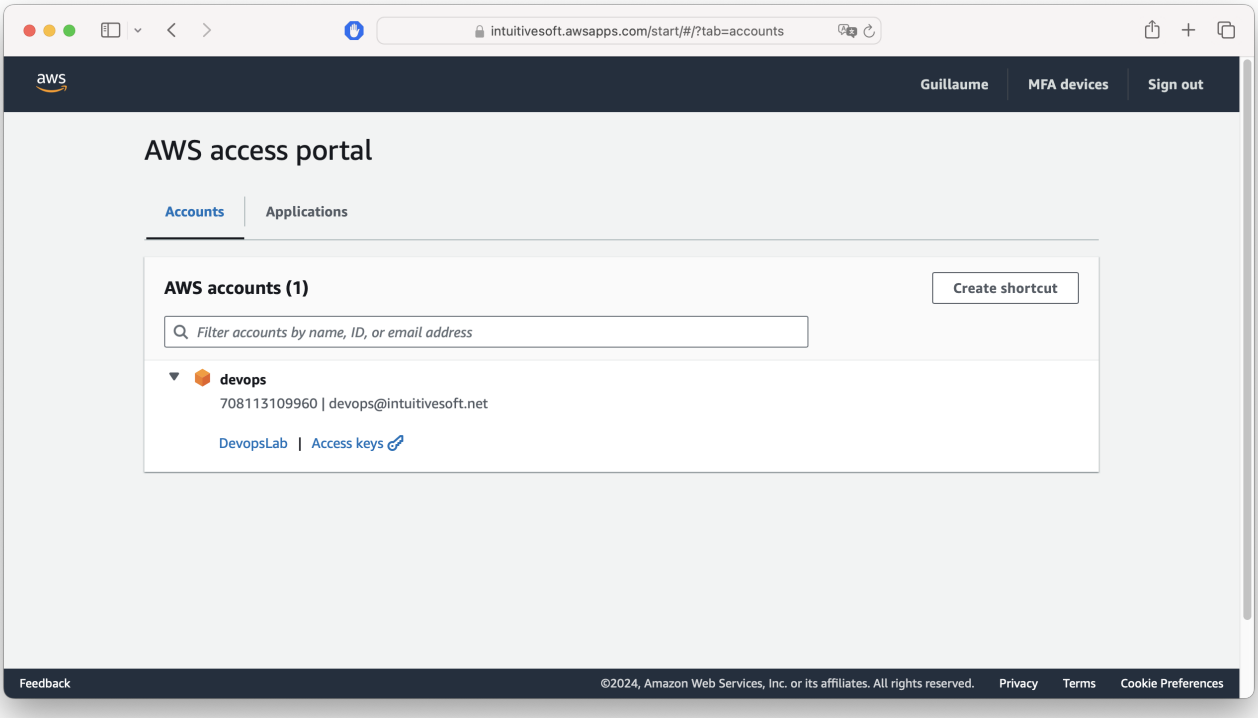
You need to specify application credentials for authentication and permissions on the AWS API.

- Task 3.1 -

Connect to the training SSO portal :

https://intuitivesoft.awsapps.com/start#/

Select your training AWS account and click the link `Command Line or programmatic access` to retrieve your credentials.



- Task 3.2 -

Configure your aws cli with SSO information and grant permission to the instance using your credentials.

```
aws configure sso
```

When prompted enter the following information

SSO session name (Recommended):

```
YOUR_SSO_USERNAME
```

SSO start URL [None]:

```
https://intuitivesoft.awsapps.com/start#
```

SSO region [None]:

```
us-east-1
```

SSO registration scopes [sso:account:access]: Leave with default value press return.

```

```

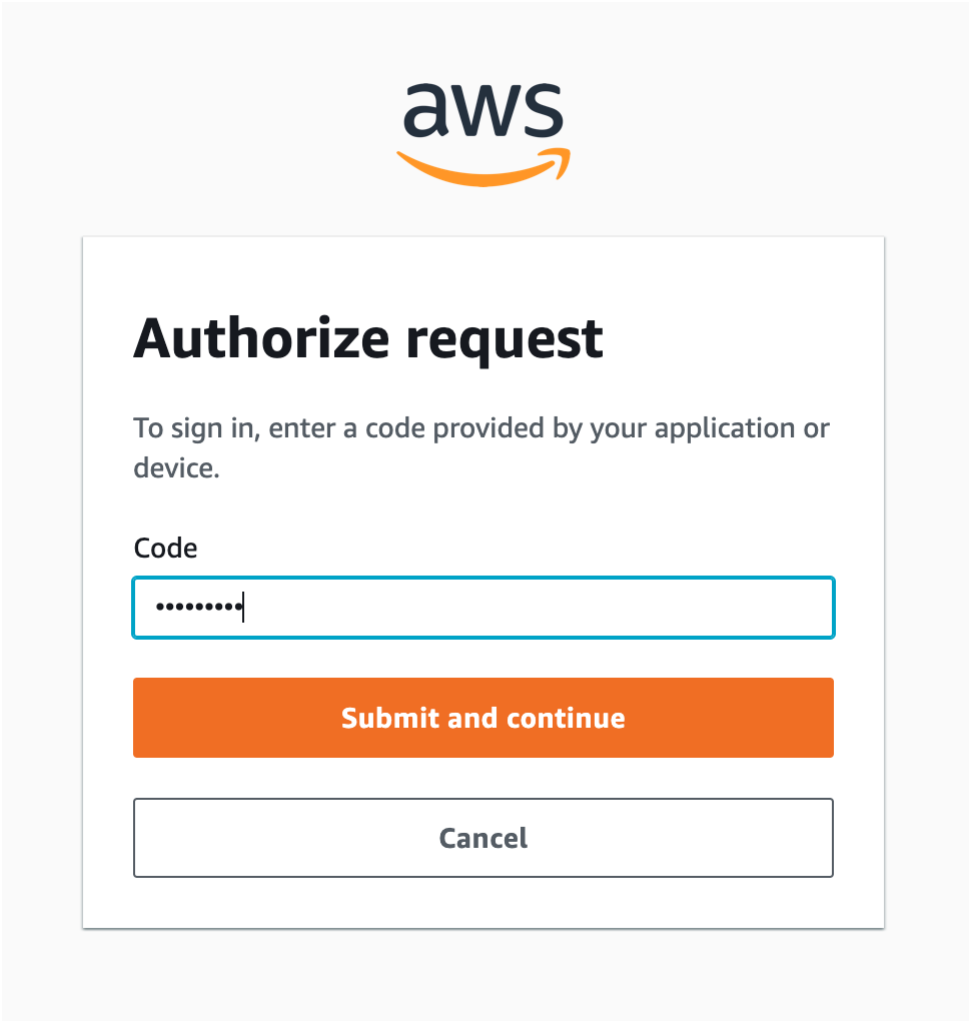
Enter authentication code in browser window

You should have the following procedure

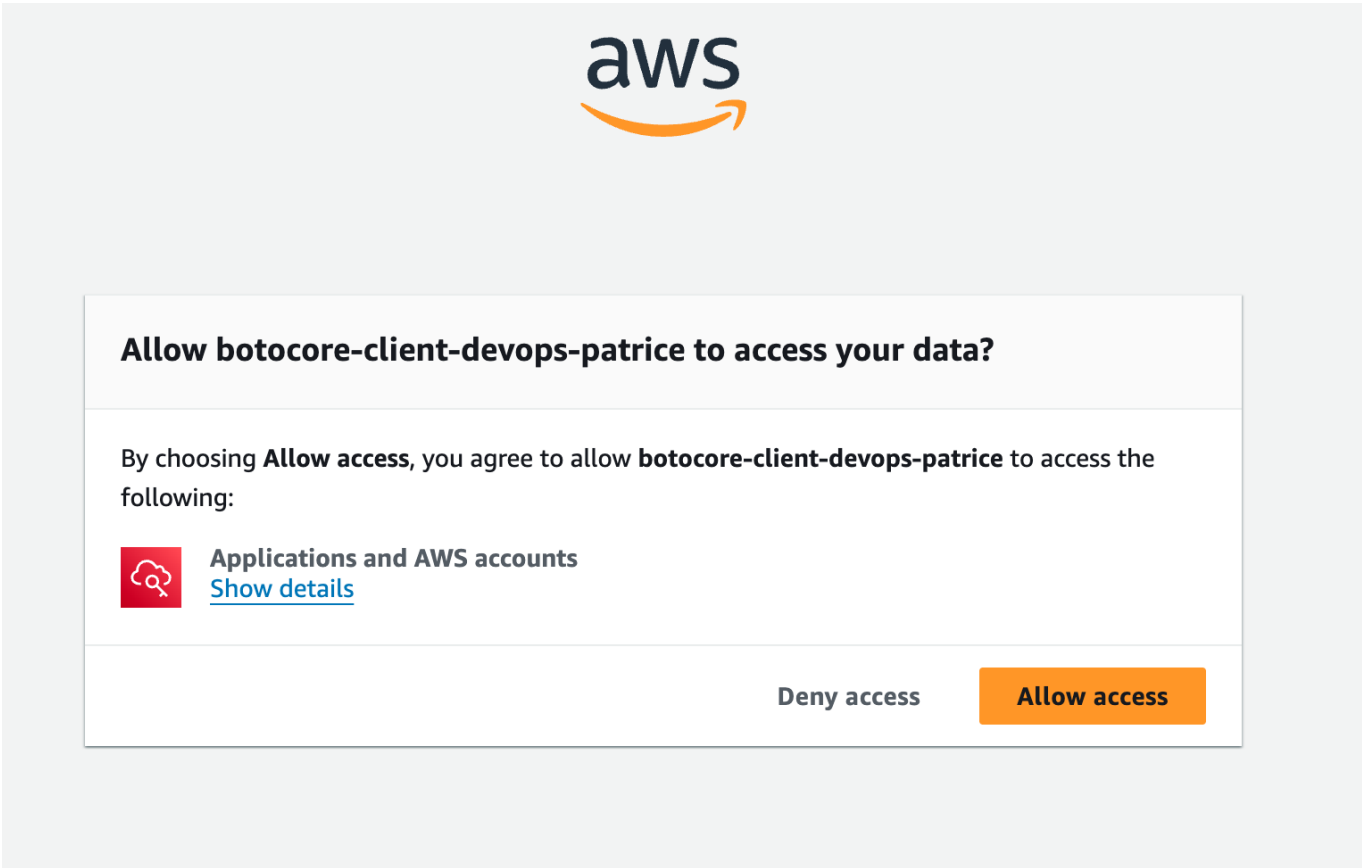
```
Attempting to automatically open the SSO authorization page in your
default browser.
If the browser does not open or you wish to use a different device to
authorize this request, open the following URL:

https://device.sso.us-east-1.amazonaws.com/

Then enter the code:

XXXX-XXXX
```

Follow the screen on your Web browser:

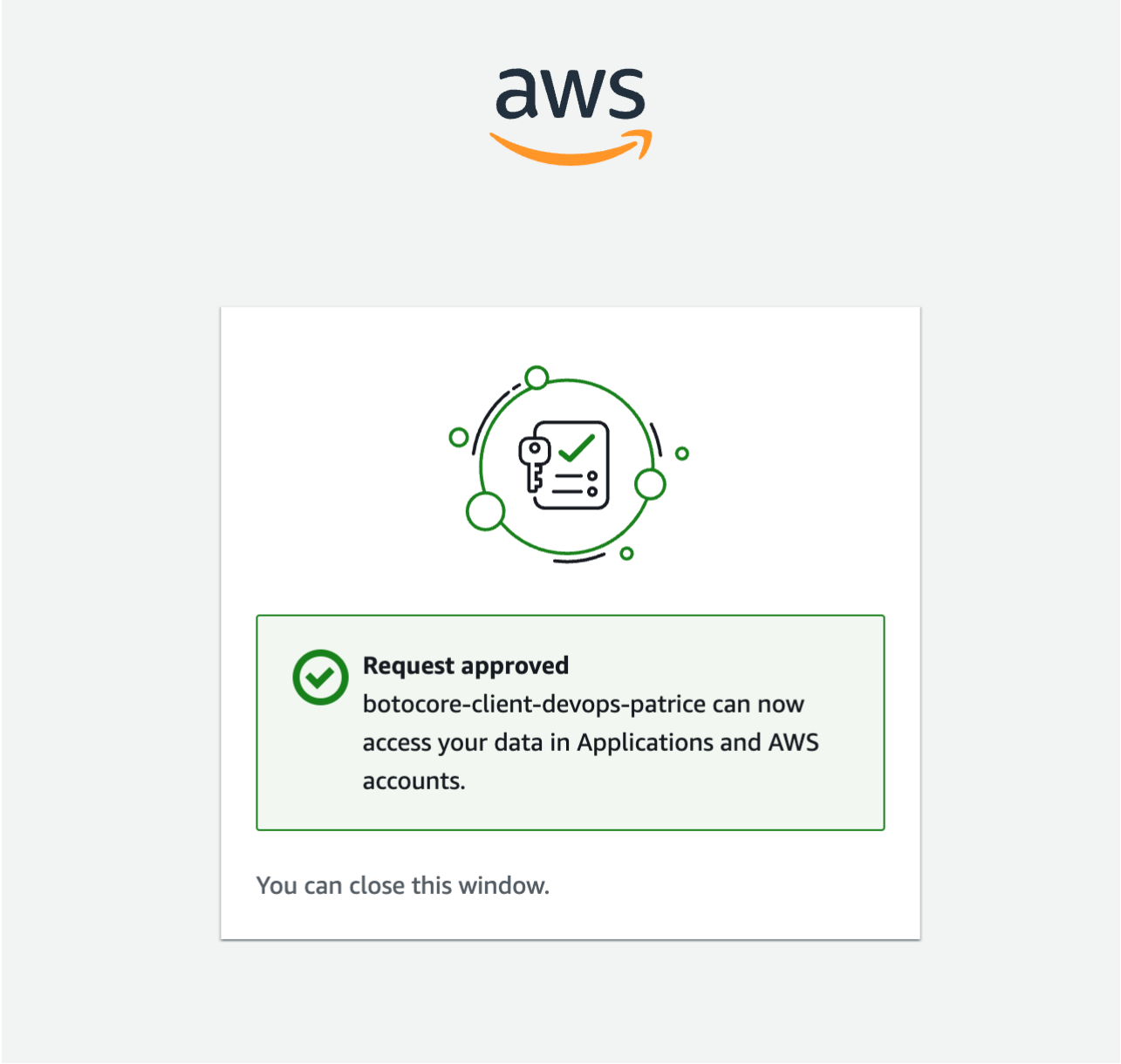
Fill in your code and press the `Submit and Continue` button



Allow access



Check everything is ok



The instructions on the console should resume with:

```
The only AWS account available to you is: 708113109960
Using the account ID 708113109960
The only role available to you is: DevopsLab
Using the role name "DevopsLab"
```

and

CLI default client Region [None]:

```
us-east-1
```

CLI default output format [None]:

```
json
```

CLI profile name [DevopsLab-708113109960]:

```
default
```

Expected output

```
SSO session name (Recommended): YOUR_SSO_USERNAME
SSO start URL [None]: https://intuitivesoft.awsapps.com/start#
SSO region [None]: us-east-1
SSO registration scopes [sso:account:access]:
Attempting to automatically open the SSO authorization page in your
default browser.
If the browser does not open or you wish to use a different device to
authorize this request, open the following URL:

https://device.sso.us-east-1.amazonaws.com/

Then enter the code:

XXXX-XXXX

The only AWS account available to you is: 708113109960
Using the account ID 708113109960
The only role available to you is: DevopsLab
Using the role name "DevopsLab"
CLI default client Region [None]: us-east-1
CLI default output format [None]: json
CLI profile name [DevopsLab-708113109960]: default

To use this profile, specify the profile name using --profile, as shown:

aws s3 ls --profile default
```

If you misconfigured your environment check the FAQ

Task 4. Verify your access to AWS via CLI

```
aws sts get-caller-identity
```

You should have an output similar to :

```
{
    "UserId": "AROA2JXWUSPEDW7XVV72X:devops",
    "Account": "708113109960",
    "Arn": "arn:aws:sts::708113109960:assumed-
role/AWSReservedSSO_DevopsLab_a5ad8dc258100c0a/devops"
}
```

The full documentation of AWS CLI usage can be found here : AWS Documentation : AWS CLI usage

Task 5. Retrieve your VPC information

- Task 5.1 -

```
aws ec2 describe-vpcs
```

You should have an output similar to :

```
{
  "Vpcs": [
    {
      "CidrBlock": "192.168.0.0/16",
      "DhcpOptionsId": "dopt-02cb9007f1d29d0fc",
      "State": "available",
      "VpcId": "vpc-REDACTED",
      "OwnerId": "REDACTED",
      "InstanceTenancy": "default",
      "CidrBlockAssociationSet": [
        {
          "AssociationId": "vpc-cidr-assoc-0e933f3263a8b89b7",
          "CidrBlock": "192.168.0.0/16",
          "CidrBlockState": {
            "State": "associated"
  [...]
```

Note that all the account VPC are displayed

- Task 5.2 -

You can filter on tag by using the `--filters` option.

On user defined tag :

Replace `tag:` and `Values` with the appropriate tag value / VPC name you are using.

```
aws ec2 describe-vpcs --filters Name=tag:lab-id,Values=pod1
```

Or object Name :

```
aws ec2 describe-vpcs --filters Name=tag:Name,Values=pod1-vpc
```

Task 6. Retrieve your EC2 instances

Use `--filters` option to filter to your instance name or tag.

Replace `Values` with the appropriate Instance name you are using.

```
aws ec2 describe-instances --filters Name=tag:Name,Values=REDACTED
```

Review the metadata attached to your instance.

You should have an output similar to :

Press `space` or `return` to advance the Linux CLI pager and press `q` to end/quit the visualization


```
{
    "Reservations": [
        {
            "Groups": [],
            "Instances": [
                {
                    "AmiLaunchIndex": 0,
                    "ImageId": "ami-053b0d53c279acc90",
                    "InstanceId": "i-02d46929fca25538b",
                    "InstanceType": "t2.micro",
                    "KeyName": "pod1-key",
                    "LaunchTime": "2023-08-14T17:49:04+00:00",
                    "Monitoring": {
                        "State": "disabled"
                    },
                    "Placement": {
                        "AvailabilityZone": "us-east-1c",
                        "GroupName": "",
                        "Tenancy": "default"
                    },
                    "PrivateDnsName": "ip-192-168-2-56.ec2.internal",
                    "PrivateIpAddress": "192.168.2.56",
                    "ProductCodes": [],
                    "PublicDnsName": "",
                    "PublicIpAddress": "54.209.177.229",
                    "State": {
                        "Code": 16,
                        "Name": "running"
                    },
                    "StateTransitionReason": "",
```

```
                    "SubnetId": "subnet-0e09b804de9f67b3d",
                    "VpcId": "vpc-0ed2cfd7555c1d240",
                    "Architecture": "x86_64",
                    "BlockDeviceMappings": [
                        {
                            "DeviceName": "/dev/sda1",
                            "Ebs": {
                                "AttachTime": "2023-08-14T17:49:04+00:00",
                                "DeleteOnTermination": true,
                                "Status": "attached",
                                "VolumeId": "vol-04068f16a20c27007"
                            }
                        }
                    ],
                    "ClientToken": "f116a4d6-0bd9-431c-bce3-aa7520589f79",
                    "EbsOptimized": false,
                    "EnaSupport": true,
                    "Hypervisor": "xen",
                    "NetworkInterfaces": [
                        {
                            "Association": {
                                "IpOwnerId": "amazon",
                                "PublicDnsName": "",
                                "PublicIp": "54.209.177.229"
                            },
                            "Attachment": {
                                "AttachTime": "2023-08-14T17:49:04+00:00",
                                "AttachmentId": "eni-attach-
0b057a5cd977a229f",
                                "DeleteOnTermination": true,
                                "DeviceIndex": 0,
                                "Status": "attached",
                                "NetworkCardIndex": 0
                            },
                            "Description": "",
                            "Groups": [
                                {
                                    "GroupName": "launch-wizard-1",
                                    "GroupId": "sg-0fa221a4a7b006186"
                                }
                            ],
                            "Ipv6Addresses": [],
                            "MacAddress": "0a:4b:14:51:fc:fb",
                            "NetworkInterfaceId": "eni-0123540a2df3e4232",
                            "OwnerId": "708113109960",
                            "PrivateIpAddress": "192.168.2.56",
                            "PrivateIpAddresses": [
                                {
                                    "Association": {
                                        "IpOwnerId": "amazon",
                                        "PublicDnsName": "",
                                        "PublicIp": "54.209.177.229"
                                    },
                                    "Primary": true,
```

```
                                "PrivateIpAddress": "192.168.2.56"
                            }
                        ],
                        "SourceDestCheck": true,
                        "Status": "in-use",
                        "SubnetId": "subnet-0e09b804de9f67b3d",
                        "VpcId": "vpc-0ed2cfd7555c1d240",
                        "InterfaceType": "interface"
                    }
                ],
                "RootDeviceName": "/dev/sda1",
                "RootDeviceType": "ebs",
                "SecurityGroups": [
                    {
                        "GroupName": "launch-wizard-1",
                        "GroupId": "sg-0fa221a4a7b006186"
                    }
                ],
                "SourceDestCheck": true,
                "Tags": [
                    {
                        "Key": "Name",
                        "Value": "devops-instance"
                    }
                ],
                "VirtualizationType": "hvm",
                "CpuOptions": {
                    "CoreCount": 1,
                    "ThreadsPerCore": 1
                },
                "CapacityReservationSpecification": {
                    "CapacityReservationPreference": "open"
                },
                "HibernationOptions": {
                    "Configured": false
                },
                "MetadataOptions": {
                    "State": "applied",
                    "HttpTokens": "optional",
                    "HttpPutResponseHopLimit": 1,
                    "HttpEndpoint": "enabled",
                    "HttpProtocolIpv6": "disabled",
                    "InstanceMetadataTags": "disabled"
                },
                "EnclaveOptions": {
                    "Enabled": false
                },
                "PlatformDetails": "Linux/UNIX",
                "UsageOperation": "RunInstances",
                "UsageOperationUpdateTime": "2023-08-
14T17:49:04+00:00",
                "PrivateDnsNameOptions": {
                    "HostnameType": "ip-name",
                    "EnableResourceNameDnsARecord": false,
```

```
                        "EnableResourceNameDnsAAAARecord": false
                    },
                    "MaintenanceOptions": {
                        "AutoRecovery": "default"
                    },
                    "CurrentInstanceBootMode": "legacy-bios"
                }
            ],
            "OwnerId": "708113109960",
            "ReservationId": "r-07c98454b8f0f3201"
        }
    ]
}
```

Task 8. Retrieve and save your Instance Id

From the previous command output we need to retrieve the `InstanceId` information the response payload.

To do it programmatically we can leverage linux JSON utility tool `jq` with a filter.

Save your instance ID as environnement variable as you are going to reuse it later on

- Task 8.1 -

Analyze payload output

```
{
    "Reservations": [                                   # First element
of the Reservations list
        {
            "Groups": [],
            "Instances": [                              # First element
of the Instances list
                {
                    "AmiLaunchIndex": 0,
                    "ImageId": "ami-08a52ddb321b32a8c",
                    "InstanceId": "i-04342ad763fc1a659",  # Key is
InstanceId

  [...]
```

The filter is : `.Reservations[0].Instances[0].InstanceId`

- Task 8.2 -

Re-use previous request and pipe (`|`) its output to `jq` with the crafted filter.

Replace `REDACTED` with the appropriate tag value or instance identifier you are using.

The Instance ID will be saved to environnement variable called `INSTANCE_ID`

```
export INSTANCE_ID=$(aws ec2 describe-instances --filters
Name=tag:Name,Values=REDACTED --output json | jq -r
'.Reservations[0].Instances[0].InstanceId')
```

```
echo $INSTANCE_ID
```

You should have an output similar to :

```
i-04342ad763fc1a659
```

Task 9. Retrieve Instance AMI information

- Task 8.1 -

Using the same logic we can filter to the ImageId of your instance :
`.Reservations[0].Instances[0].ImageId`

```
aws ec2 describe-instances --instance-ids $INSTANCE_ID --output json | jq
-r '.Reservations[0].Instances[0].ImageId'
```

You should have an output similar to :

```
ami-053b0d53c279acc90
```

Task 9. Retrieve information on the AMI

```
aws ec2 describe-images --region us-east-1 --image-ids ami-
053b0d53c279acc90
```

```
{
    "Images": [
        {
            "Architecture": "x86_64",
            "CreationDate": "2023-05-16T03:38:03.000Z",
            "ImageId": "ami-053b0d53c279acc90",
            "ImageLocation": "amazon/ubuntu/images/hvm-ssd/ubuntu-jammy-
22.04-amd64-server-20230516",
            "ImageType": "machine",
            "Public": true,
```

```
            "OwnerId": "099720109477",
            "PlatformDetails": "Linux/UNIX",
            "UsageOperation": "RunInstances",
            "State": "available",
            "BlockDeviceMappings": [
                {
                    "DeviceName": "/dev/sda1",
                    "Ebs": {
                        "DeleteOnTermination": true,
                        "SnapshotId": "snap-0d3283808e9f92122",
                        "VolumeSize": 8,
                        "VolumeType": "gp2",
                        "Encrypted": false
                    }
                },
                {
                    "DeviceName": "/dev/sdb",
                    "VirtualName": "ephemeral0"
                },
                {
                    "DeviceName": "/dev/sdc",
                    "VirtualName": "ephemeral1"
                }
            ],
            "Description": "Canonical, Ubuntu, 22.04 LTS, amd64 jammy
image build on 2023-05-16",
            "EnaSupport": true,
            "Hypervisor": "xen",
            "ImageOwnerAlias": "amazon",
            "Name": "ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-
server-20230516",
            "RootDeviceName": "/dev/sda1",
            "RootDeviceType": "ebs",
            "SriovNetSupport": "simple",
            "VirtualizationType": "hvm",
            "BootMode": "legacy-bios",
            "DeprecationTime": "2025-05-16T03:38:03.000Z"
        }
    ]
}
```


Task 10. Lifecycle of your instance

- Task 10.1 -

Stop/Start your instance

```
aws ec2 stop-instances --instance-ids $INSTANCE_ID
```

Expected output:

```
{
    "StoppingInstances": [
        {
            "CurrentState": {
                "Code": 64,
                "Name": "stopping"
            },
            "InstanceId": "REDACTED",
            "PreviousState": {
                "Code": 16,
                "Name": "running"
            }
        }
    ]
}
```

You can check the status of your instances

```
aws ec2 describe-instance-status --instance-ids $INSTANCE_ID --include-
all-instances
```

Expected output:

```
{
    "InstanceStatuses": [
        {
            "AvailabilityZone": "us-east-1a",
            "InstanceId": "REDACTED",
            "InstanceState": {
                "Code": 80,
                "Name": "stopped"
            },
            "InstanceStatus": {
                "Status": "not-applicable"
            },
            "SystemStatus": {
                "Status": "not-applicable"
            }
        }
    ]
}
```

Finally you can re-start and/or terminate (delete) your instance

```
aws ec2 terminate-instances --instance-ids $INSTANCE_ID
```

Expected output

```
{
    "TerminatingInstances": [
        {
            "CurrentState": {
                "Code": 48,
                "Name": "terminated"
            },
            "InstanceId": "REDACTED",
            "PreviousState": {
                "Code": 80,
                "Name": "stopped"
            }
        }
    ]
}
```

Task 12. Clean your lab environnement

- Task 12.1 -

Make sure your instance is Terminated before cleaning your lab

Create a bash script named `clean-vpc.sh` with the following content :

```
read -p 'Please enter your VPC Name:  ' VPC_NAME

# Retrieve your vpc-id
VPC_ID=$(aws ec2 describe-vpcs --filters Name=tag:Name,Values=${VPC_NAME}
| jq -r '.Vpcs[].VpcId')

# Check if your VPC exist
if [ -z "$VPC_ID" ]; then
    echo "VPC not found with name : ${VPC_NAME}"
    exit 1
fi

echo -e "Cleaning VPC : ${VPC_ID}"

# Detach internet gateway from your VPC
for igw in $(aws ec2 describe-internet-gateways --filters
Name=attachment.vpc-id,Values="${VPC_ID}" | jq -r
'.InternetGateways[].InternetGatewayId'); do
  echo -e "\tDetach Internet Gateway ${igw}"
  aws ec2 detach-internet-gateway --internet-gateway-id=$igw --vpc-
id=$VPC_ID
  # Wait for IGW to be detached
  sleep 5
  # Delete the internet gateway
```

```
  echo -e "\tDelete Internet Gateway ${igw}"
  aws ec2 delete-internet-gateway --internet-gateway-id=$igw
done

# Delete all subnets attached to your VPC
for subnet in $(aws ec2 describe-subnets --filters Name=vpc-
id,Values="${VPC_ID}" | jq -r '.Subnets[].SubnetId'); do
  echo -e "\tDelete Subnet : ${subnet}"
  aws ec2 delete-subnet --subnet-id ${subnet}
done

# Delete all route table attached to your VPC but the main one (Forbidden)
for route_table in $(aws ec2 describe-route-tables --filters "Name=vpc-
id,Values=${VPC_ID}" --query 'RouteTables[?Associations == `[]`]' | jq -r
'.[].RouteTableId'); do
  echo -e "\tDelete Route-Table : ${route_table}"
  aws ec2 delete-route-table --route-table-id "$route_table"
done


# Delete Security Groups attached to your VPC but the default one
(Forbidden)
for sg in $(aws ec2 describe-security-groups --filters Name=vpc-
id,Values="${VPC_ID}" --query "SecurityGroups[?GroupName!='default']" | jq
-r '.[].GroupId'); do
  echo -e "\tDelete Security Group ${sg}"
  aws ec2 delete-security-group --group-id $sg
done
```

- Task 12.2 -

Make the script executable

```
chmod +x clean-vpc.sh
```

- Task 12.3 -

Execute your script

```
./clean-vpc.sh
```

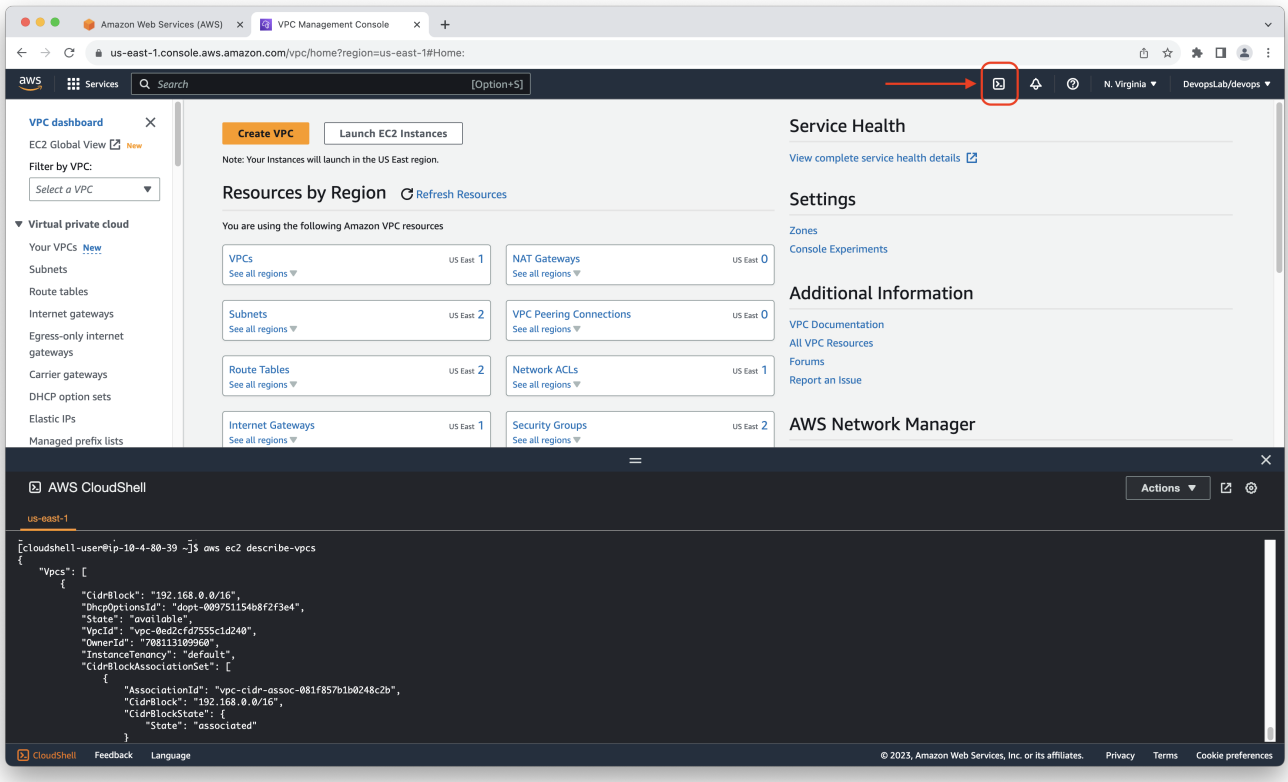
Expected Output :

```
Cleaning VPC : vpc-006ddcf026ca43388
  Detach Internet Gateway : igw-3e5f7b02f8886a834
  Delete Internet Gateway : igw-3e5f7b02f8886a834
  Delete Route-Table : rtb-23555af900v000
```

```
Delete Subnet : subnet-rf0330303030d4
Delete Subnet : subnet-gh9diddfbc9244
Delete Security Group sg-0eaf3b02b8886a626
```

Insight 13. AWS CloudShell

From the AWS Web Console you can alteratively access the AWS Cloud shell CLI for testing.



Terraform Overview

Insight 1. What is Terraform

Terraform is a tool for building, changing, and versioning infrastructure as code. Infrastructure is managed as code meaning it is described using files and a high-level configuration syntax (HCL: Hashicorp Configuration Language, designed to be both human readable and machine friendly). Terraform can manage existing and popular service providers (AWS, GCP, ...) as well as custom in-house solutions (Cisco Intersight, ACI, Kubernetes, VMware, Openstack...).

The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

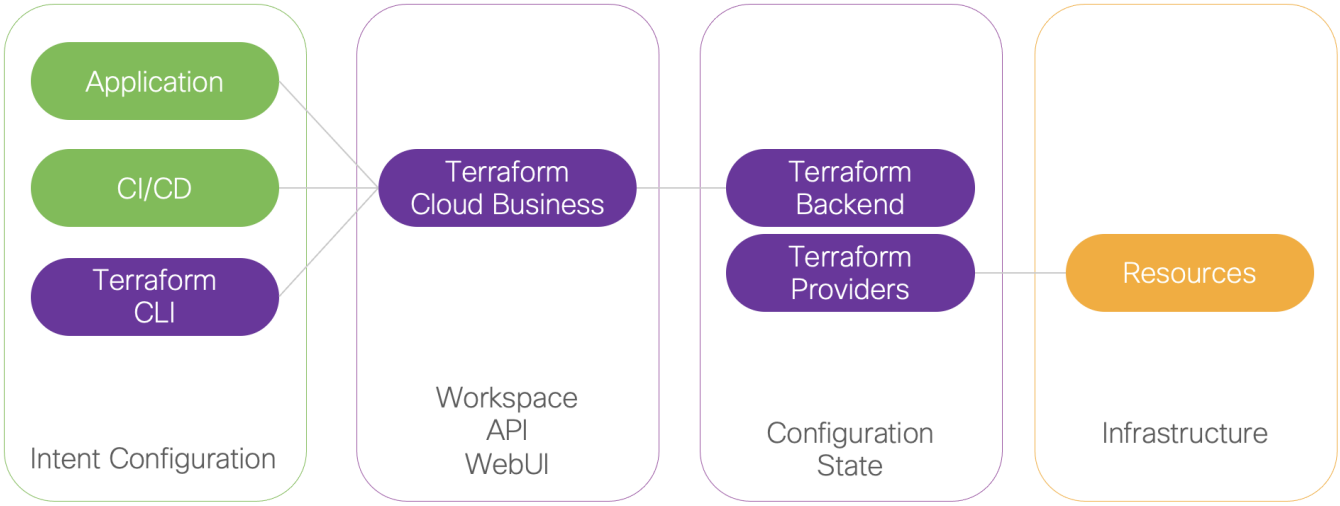
The Terraform key features are listed below:

- Infrastructure as Code: Infrastructure is described using a high-level and declarative configuration syntax (HCL: HashiCorp Configuration Language).
- Execution Plans: Terraform has a "planning" step called execution plan, it shows the changes that will be configured when the plan is applied.

- Resource Graph: Terraform builds a graph of all your resources, and parallelizes the creation and modification of any non-dependent resources.
- Change Automation: Terraform keep configuration states (real view of resources configuration) and figure out the changes and in what order to reach the intent.

Terraform providers abstract the API layer of real resources (Google Cloud, Azure, AWS, Cisco, ...).

Insight 2. Architecture



Insight 3. Configuration Files

Terraform code is written in the HashiCorp Configuration Language (HCL) in files with the extension .tf. It is a declarative language, so your goal is to describe the infrastructure you want, and Terraform will figure out how to create it.

The first step to using Terraform is typically to configure the provider(s) you want to use. Create an empty folder and put a file in it called main.tf that contains the following contents:

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

This tells Terraform that you are going to be using AWS as your provider and that you want to deploy your infrastructure into the us-east-1 region.

For each type of provider, there are many different kinds of resources that you can create, such as servers, databases, and load balancers. The general syntax for creating a resource in Terraform is:

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
  [CONFIG ...]
}
```

where PROVIDER is the name of a provider (e.g., aws), TYPE is the type of resource to create in that provider (e.g., instance), NAME is an identifier you can use throughout the Terraform code to refer to this resource (e.g., my_instance), and CONFIG consists of one or more arguments that are specific to that resource.

For example, to deploy a single (virtual) server in AWS, known as an EC2 Instance, use the aws_instance resource in main.tf as follows:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

where:

- ami: The Amazon Machine Image (AMI) to run on the EC2 Instance. You can find free and paid AMIs in the AWS Marketplace or create your own using tools such as Packer (see “Server Templating Tools” for a discussion of machine images and server templating). The preceding code example sets the ami parameter to the ID of an Ubuntu 18.04 AMI in us-east-2. This AMI is free to use.
- instance_type: The type of EC2 Instance to run. Each type of EC2 Instance provides a different amount of CPU, memory, disk space, and networking capacity. The EC2 Instance Types page lists all the available options. The preceding example uses t2.micro, which has one virtual CPU, 1 GB of memory, and is part of the AWS free tier.

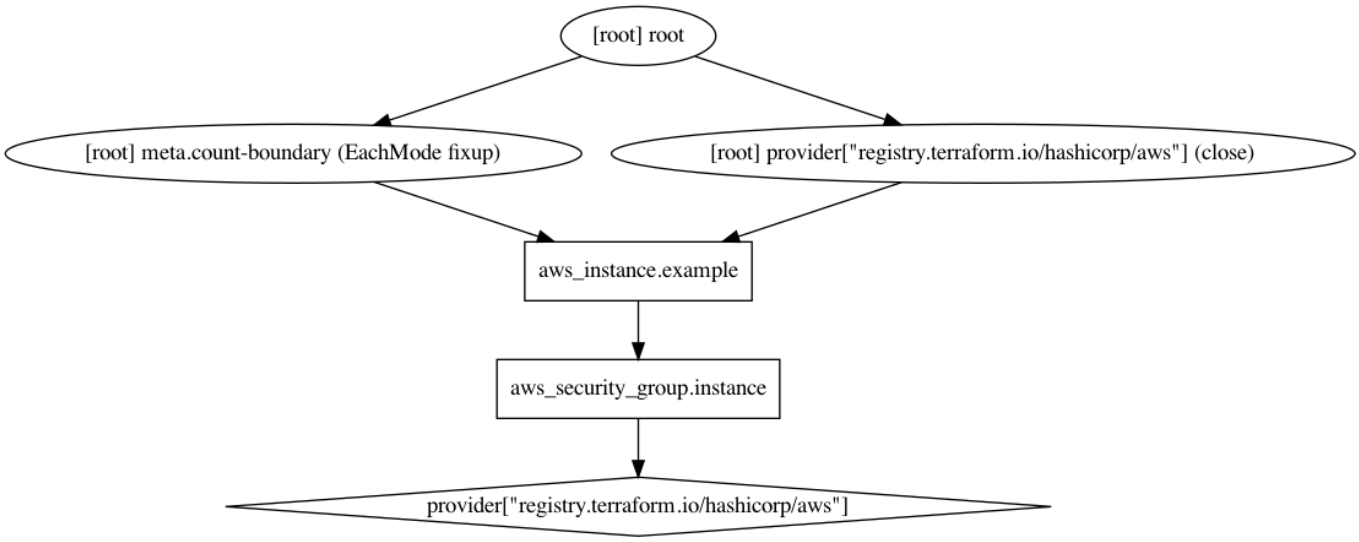
Insight 4. Workflow

Terraform workflow includes the following steps:

1. terraform init: scan the code, figure out which providers you’re using, and download the code for them.
2. terraform plan: let you see what Terraform will do before actually making any changes
3. terraform apply: to actually create/update/delete the resources

Insight 5. Dependencies Graph

When you add a reference from one resource to another, you create an implicit dependency. Terraform parses these dependencies, builds a dependency graph from them, and uses that to automatically determine in which order it should create resources.



Insight 6. State Files

Every time you run Terraform, it records information about what infrastructure it created in a Terraform state file. By default, when you run Terraform in the folder /foo/bar, Terraform creates the file /foo/bar/terraform.tfstate. This file contains a custom JSON format that records a mapping from the Terraform resources in your configuration files to the representation of those resources in the real world. For example, let's say your Terraform configuration contained the following:

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"
}
```

After running terraform apply, here is a small snippet of the contents of the terraform.tfstate file (truncated for readability):

```
{
  "version": 4,
  "terraform_version": "0.12.0",
  "serial": 1,
  "lineage": "1f2087f9-4b3c-1b66-65db-8b78faafc6fb",
  "outputs": {},
  "resources": [
    {
      "mode": "managed",
      "type": "aws_instance",
      "name": "example",
      "provider": "provider.aws",
      "instances": [
        {
          "schema_version": 1,
          "attributes": {
            "ami": "ami-0c55b159cbfafe1f0",
            "availability_zone": "us-east-2c",
```

```
            "id": "i-00d689a0acc43af0f",
            "instance_state": "running",
            "instance_type": "t2.micro",
            "(...)": "(truncated)"
          }
        }
      ]
    }
  ]
}
```

Using this JSON format, Terraform knows that a resource with type aws_instance and name example corresponds to an EC2 Instance in your AWS account with ID i-00d689a0acc43af0f. Every time you run Terraform, it can fetch the latest status of this EC2 Instance from AWS and compare that to what's in your Terraform configurations to determine what changes need to be applied. In other words, the output of the plan command is a diff between the code on your computer and the infrastructure deployed in the real world, as discovered via IDs in the state file.

Insight 7. Glossary

- **Provider**: A provider is an abstraction of the API/service provider such as AWS, GCP, DNSimple, or Fastly. Providers typically require some sort of configuration data such as an API key or credential file.
- **Resource**: A resource represents a component of a provider such as an "AWS instance", "ACI Tenant", or "Fastly service". Resources have both arguments (inputs) and attributes (outputs) which are specific to the resource. Resources also have meta-parameters such as count and lifecycle.
- **(Resource) Argument**: An argument is an input or configuration option to a resource. An AWS EC2 instance accepts ami as an input parameter. This makes ami an argument of the aws_instance resource.
- **(Resource) Attribute**: An attribute is an output or computed value available only after resource creation. An AWS EC2 instance provides public_ip as an output parameter. This makes public_ip an attribute of the aws_instance resource. This makes sense, because an instance's IP address is assigned during creation.
- **Graph**: The graph is the internal structure for Terraform's resource dependencies and order. The graph implements a directed acyclic graph (DAG) which allows Terraform to optimize for parallelism while adhering to dependency ordering. It is possible to generate the graph as a DOT file for human viewing.
- **(Remote/Local) State**: Terraform stores the last-known arguments and attributes for all resources. These contents known as "state" can be stored locally as a JSON file (local state) or stored in a remote shared location like Terraform Enterprise or Terraform Cloud (remote state).
- **Module**: A module is a blackbox, self-contained package of Terraform configurations. Modules are like abstract classes that are imported into other Terraform configurations. Parallels: Chef Cookbook, Puppet Module, Ruby gem
- **Variable**: A variable is a user or machine-supplied input in Terraform configurations. Variables can be supplied via environment variables, CLI flags, or variable files. Combined with modules, variables help make Terraform flexible, shareable, and extensible.
- **Output**: An output is a configurable piece of information that is highlighted at the end of a Terraform run.

- **Interpolation**: Terraform includes a built-in syntax for referencing attributes of other resources. This technique is called interpolation. Terraform also provides built-in functions for performing string manipulations, evaluating math operations, and doing list comprehensions.
- **HashiCorp Configuration Language (HCL)**: Terraform's syntax and interpolation are part of an open source language and specification called HCL.

LAB : AWS Terraform Lab

Objectives

- understand Infrastructure as Code principles
- understand Terraform HashiCorp configuration language (HCL) and workflow
- experiment Terraform with AWS provider
- understand AWS network concepts and modelisation

Insight 1. Access Key for Terraform Provider

AWS Identity and Access Management (IAM) is a web service that helps you securely control access to AWS resources. You use IAM to control who is authenticated (signed in) and authorized (has permissions) to use resources. IAM supports the use of access keys to make programmatic calls to AWS from the AWS CLI, AWS SDKs, direct AWS API calls or automation tools like Terraform. You can have a maximum of two access keys (active or inactive) at a time.

For your protection, you should never share your secret keys with anyone. As a best practice, we recommend frequent key rotation.

Task 2. Retrieve your AWS secrets

If you already configured AWS CLI you can skip this Task

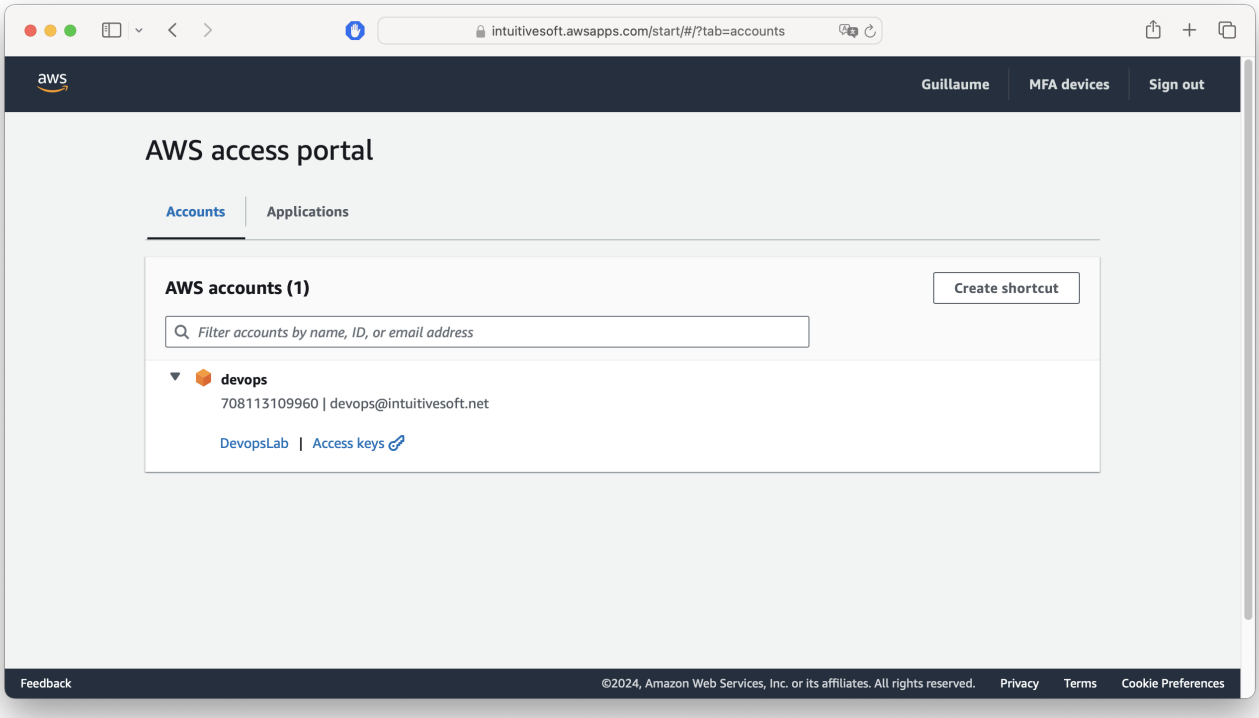
You need to specify application credentials for authentication and permissions on the AWS API.

- Task 2.1 -

Connect to the training SSO portal :

https://intuitivesoft.awsapps.com/start#/

Select your training AWS account and click the link `Command Line or programmatic access` to retrieve your credentials.



- Task 2.2 -

Configure your aws cli with your SSO information

```
aws configure sso
```

Expected output

```
SSO session name (Recommended): YOUR_SSO_USERNAME
SSO start URL [None]: https://intuitivesoft.awsapps.com/start#
SSO region [None]: us-east-1
SSO registration scopes [sso:account:access]:
Attempting to automatically open the SSO authorization page in your
default browser.
If the browser does not open or you wish to use a different device to
authorize this request, open the following URL:

https://device.sso.us-east-1.amazonaws.com/

Then enter the code:

XXXX-XXXX

The only AWS account available to you is: 708113109960
Using the account ID 708113109960
The only role available to you is: DevopsLab
Using the role name "DevopsLab"
CLI default client Region [None]: us-east-1
```

```
CLI default output format [None]: json
CLI profile name [DevopsLab-708113109960]: default

To use this profile, specify the profile name using --profile, as shown:

aws s3 ls --profile default
```

If you misconfigured your environment check the FAQ

Insight 3. Terraform Overview

This lab involves file editing. It's recommended to use the Lab Dashboard Code Editor and installing `Hashicorp Terraform` extension for VSCode

What is Infrastructure as Code with Terraform

Infrastructure as code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface. IaC allows you to build, change, and manage your infrastructure in a safe, consistent, and repeatable way by defining resource configurations that you can version, reuse, and share.

Terraform is HashiCorp's infrastructure as code tool. It lets you define resources and infrastructure in human-readable, declarative configuration files, and manages your infrastructure's lifecycle. Using Terraform has several advantages over manually managing your infrastructure:

- Terraform can manage infrastructure on multiple cloud platforms.
- The human-readable configuration language helps you write infrastructure code quickly.
- Terraform's state allows you to track resource changes throughout your deployments.
- You can commit your configurations to version control to safely collaborate on infrastructure.

Manage any infrastructure

Terraform plugins called providers let Terraform interact with cloud platforms and other services via their application programming interfaces (APIs). HashiCorp and the Terraform community have written over 1,000 providers to manage resources on Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP), Kubernetes, Helm, GitHub, Splunk, and DataDog, just to name a few. Find providers for many of the platforms and services you already use in the Terraform Registry (https://registry.terraform.io/browse/providers).

Terraform can manage existing and popular service providers (AWS, GCP, ...) as well as custom in-house solutions (Cisco Intersight, ACI, Kubernetes, VMware, Openstack...).

The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

The Terraform key features are listed below:

- Infrastructure as Code: Infrastructure is described using a high-level and declarative configuration syntax (HCL: HashiCorp Configuration Language).

- Execution Plans: Terraform has a "planning" step called execution plan, it shows the changes that will be configured when the plan is applied.
- Resource Graph: Terraform builds a graph of all your resources, and parallelizes the creation and modification of any non-dependent resources.
- Change Automation: Terraform keep configuration states (real view of resources configuration) and figure out the changes and in what order to reach the intent.

Standardize your deployment workflow

Providers define individual units of infrastructure, for example compute instances or private networks, as resources. You can compose resources from different providers into reusable Terraform configurations called modules, and manage them with a consistent language and workflow.

Terraform's configuration language is declarative, meaning that it describes the desired end-state for your infrastructure, in contrast to procedural programming languages that require step-by-step instructions to perform tasks. Terraform providers automatically calculate dependencies between resources to create or destroy them in the correct order.

Terraform workflow includes the following steps:

- terraform init: scan the code, figure out which providers you’re using, and download the code for them.
- terraform plan: let you see what Terraform will do before actually making any changes
- terraform apply: to actually create/update/delete the resources

Dependency Graph

When you add a reference from one resource to another, you create an implicit dependency. Terraform parses these dependencies, builds a dependency graph from them, and uses that to automatically determine in which order it should create resources.

Track your infrastructure

Terraform keeps track of your real infrastructure in a state file, which acts as a source of truth for your environment. Terraform uses the state file to determine the changes to make to your infrastructure so that it will match your configuration.

Every time you run Terraform, it records information about what infrastructure it created in a Terraform state file. By default, when you run Terraform in the folder /foo/bar, Terraform creates the file /foo/bar/terraform.tfstate. This file contains a custom JSON format that records a mapping from the Terraform resources in your configuration files to the representation of those resources in the real world.

Collaborate

Terraform allows you to collaborate on your infrastructure with its remote state backends. When you use Terraform Cloud (free for up to five users), you can securely share your state with your teammates, provide a stable environment for Terraform to run in, and prevent race conditions when multiple people make configuration changes at once.

You can also connect Terraform Cloud to version control systems (VCSs) like GitHub, GitLab, and others, allowing it to automatically propose infrastructure changes when you commit configuration changes to VCS. This lets you manage changes to your infrastructure through version control, as you would with application code.

Task 4. Install Terraform on Linux

```
sudo apt update && sudo apt install -y gnupg software-properties-common
wget
```

Add the HashiCorp GPG key.

```
wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/hashicorp-archive-keyring.gpg
```

Add the official HashiCorp Linux repository.

```
echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg]
https://apt.releases.hashicorp.com $(lsb_release -cs) main" | sudo tee
/etc/apt/sources.list.d/hashicorp.list
```

Update to add the repository, and install the Terraform CLI.

```
sudo apt update && sudo apt install -y terraform
```

Task 5. Verify the installation

```
terraform -help
```

You should see something similar to:

```
Usage: terraform [global options] <subcommand> [args]

The available commands for execution are listed below.
The primary workflow commands are given first, followed by
less common or more advanced commands.

Main commands: ...
```

Task 6. Create a subnet (and your first Terraform script)

Create a folder on the Linux machine to host the Terraform script you are going to write for that very first step. Example `mkdir create-subnet`. In that dedicated folder create a file called `main.tf`.

Task 7. Define create-subnet main script

- Task 7.1 -

Terraform code is written in the HashiCorp Configuration Language (HCL) in files with the extension .tf. It is a declarative language, so your goal is to describe the infrastructure you want, and Terraform will figure out how to create it. The first step to using Terraform is typically to configure the provider(s) you want to use. Create an empty folder and put a file in it called `main.tf` that contains the following contents:

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

where:

- `region` is the us-east-1 region

- Task 7.2 -

For each type of provider, there are many different kinds of resources that you can create, such as servers, databases, and load balancers. The general syntax for creating a resource in Terraform is:

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
  [CONFIG ...]
}
```

where `PROVIDER` is the name of a provider (e.g., aws), `TYPE` is the type of resource to create in that provider (e.g., subnet), `NAME` is an identifier you can use throughout the Terraform code to refer to this resource (e.g., this), and `CONFIG` consists of one or more arguments that are specific to that resource.

For example, to deploy a subnet in an AWS VPC, use the aws_subnet resource in `main.tf` as follows:

```
resource "aws_subnet" "this" {
  vpc_id     = "vpc-REDACTED"
  cidr_block = "192.168.1.0/24"

  tags = {
    Name = "podx-public-subnet"
  }
}
```

where

- `vpc_id` is the vpc id of your pod
- `cidr_block` is the IP network you want to allocate to the subnet
- `tag:Name` is the name you want to give to the subnet

The resulting file looks like

```
#Configure aws provider
provider "aws" {
  region = "us-east-1"
}

#Create a subnet
resource "aws_subnet" "this" {
  vpc_id     = "vpc-REDACTED"
  cidr_block = "192.168.1.0/24"

  tags = {
    Name = "podx-public-subnet"
  }
}
```

Task 8. Terraform init

In a terminal, go into the folder where you created main.tf and run the terraform init command:

```
terraform init
```

You should see a result like this:

```
Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/tls...
- Finding latest version of hashicorp/local...
- Finding latest version of hashicorp/aws...
- Finding latest version of hashicorp/random...
...
Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to
see
any changes that are required for your infrastructure. All Terraform
commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
```

```
rerun this command to reinitialize your working directory. If you forget,
other
commands will detect it and remind you to do so if necessary.
```

The `terraform` binary contains the basic functionality for Terraform, but it does not come with the code for any of the providers (e.g., the AWS provider, Azure provider, GCP provider, etc.), so when you're first starting to use Terraform, you need to run `terraform init` to tell Terraform to scan the code, figure out which providers you're using, and download the code for them. By default, the provider code will be downloaded into a `.terraform` folder, which is Terraform's scratch directory (you may want to add it to .gitignore). Terraform will also record information about the provider code it downloaded into a `.terraform.lock.hcl` file. Just be aware that you need to run init any time you start with new Terraform code, and that it's safe to run init multiple times (the command is idempotent).

Task 9. Terraform plan

Now that you have the provider code downloaded, run the terraform plan command:

```
terraform plan
```

You should see a similar output:

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following
symbols:
  + create

Terraform will perform the following actions:

  # aws_subnet.this will be created
  + resource "aws_subnet" "this" {
      + arn                                            = (known after
apply)
      + assign_ipv6_address_on_creation                = false
      + availability_zone                              = (known after
apply)
      + availability_zone_id                           = (known after
apply)
      + cidr_block                                     = "192.168.1.0/24"
      + enable_dns64                                   = false
      + enable_resource_name_dns_a_record_on_launch    = false
      + enable_resource_name_dns_aaaa_record_on_launch = false
      + id                                             = (known after
apply)
      + ipv6_cidr_block_association_id                 = (known after
apply)
      + ipv6_native                                    = false
      + map_public_ip_on_launch                        = false
      + owner_id                                       = (known after
```

```
apply)
      + private_dns_hostname_type_on_launch            = (known after
apply)
      + tags                                           = {
          + "Name" = "podx-public-subnet"
        }
      + tags_all                                       = {
          + "Name" = "podx-public-subnet"
        }
      + vpc_id                                         = "vpc-REDACTED"
    }

Plan: 1 to add, 0 to change, 0 to destroy.

──────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't
guarantee to take exactly these actions if you run
"terraform apply" now.
```

The `plan` command lets you see what Terraform will do before actually making any changes. This is a great way to sanity check your code before unleashing it onto the world. The output of the plan command is similar to the output of the `diff` command that is part of Unix, Linux, and git: anything with a plus sign `(+)` will be created, anything with a minus sign `(-)` will be deleted, and anything with a tilde sign `(~)` will be modified in place. In the preceding output, you can see that Terraform is planning on creating an AWS subnet and nothing else, which is exactly what you want.

Task 10. Terraform apply

To actually create the subnet, run the terraform apply command:

```
terraform apply
```

You should get a similar output:

```
Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following
symbols:
  + create

Terraform will perform the following actions:

  # aws_subnet.this will be created
  + resource "aws_subnet" "this" {
      + arn                                            = (known after
apply)
      + assign_ipv6_address_on_creation                = false
```

```
      + availability_zone                                = (known after
apply)
      + availability_zone_id                             = (known after
apply)
      + cidr_block                                       = "192.168.1.0/24"
      + enable_dns64                                     = false
      + enable_resource_name_dns_a_record_on_launch      = false
      + enable_resource_name_dns_aaaa_record_on_launch = false
      + id                                               = (known after
apply)
      + ipv6_cidr_block_association_id                   = (known after
apply)
      + ipv6_native                                      = false
      + map_public_ip_on_launch                          = false
      + owner_id                                         = (known after
apply)
      + private_dns_hostname_type_on_launch              = (known after
apply)
      + tags                                             = {
          + "Name" = "podx-public-subnet"
        }
      + tags_all                                         = {
          + "Name" = "podx-public-subnet"
        }
      + vpc_id                                           = "vpc-REDACTED"
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value:
```

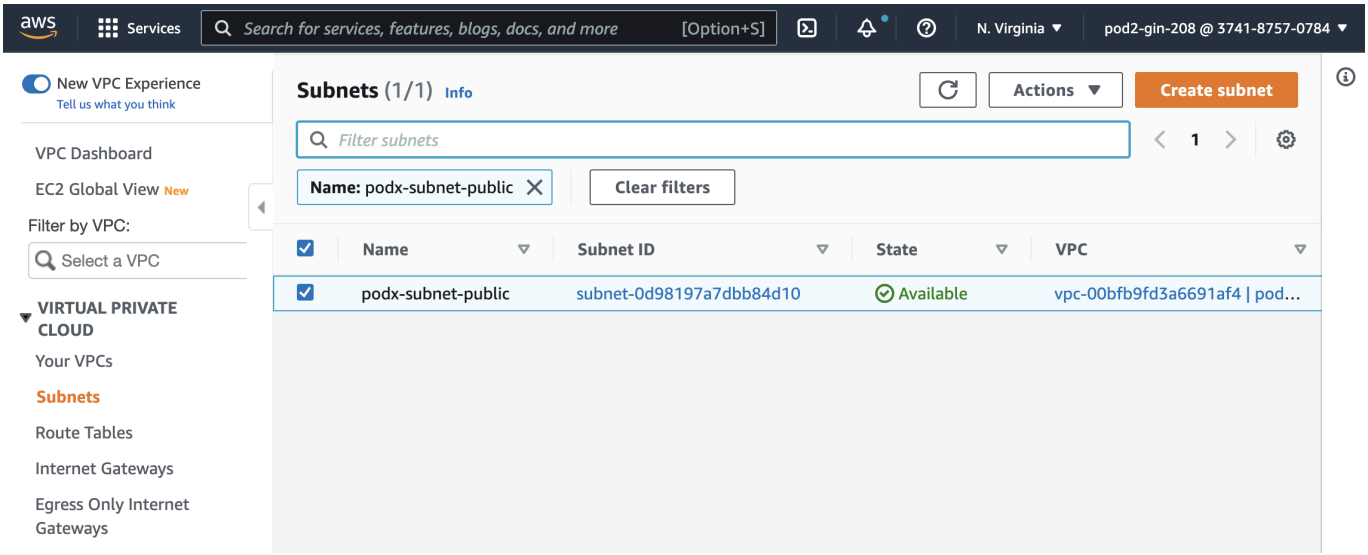
You'll notice that the apply command shows you the same plan output and asks you to confirm whether you actually want to proceed with this plan. So, while plan is available as a separate command, it's mainly useful for quick sanity checks and during code reviews, and most of the time you'll run apply directly and review the plan output it shows you.

Type `yes` and hit Enter to deploy the subnet:

You should get a similar output:

```
aws_subnet.this: Creating...
aws_subnet.this: Creation complete after 1s [id=subnet-026552bf9f58e52f9]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

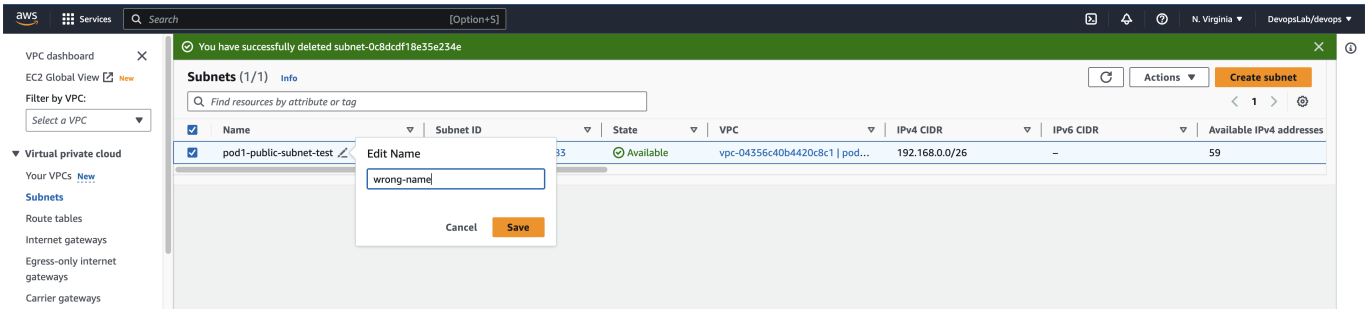
Congrats, you've just deployed a subnet in your AWS account using Terraform! To verify this, head over to the AWS console; and you should see something similar to:



Task 11. Terraform state

You can notice a file named `terraform.tfstate` in the folder from where you run the terraform script. Terraform maintains the state of the configuration in that file, meaning that if you delete the subnet from the AWS console, you will have a mismatch between the configuration intent specified in the `terraform.tfstate` file and what is really configured on AWS.

From AWS console rename the public subnet you have deployed.



If you plan a Terraform deployment again you will notice Terraform can detect the changes.

```
terraform plan
```

You should have an equivalent statement:

```
aws_subnet.this: Refreshing state... [id=subnet-0783f8621ad659e83]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the
following symbols:
  ~ update in-place
```

```
Terraform will perform the following actions:

  # aws_subnet.this will be updated in-place
  ~ resource "aws_subnet" "this" {
        id                                             = "subnet-
0783f8621ad659e83"
      ~ tags                                           = {
          ~ "Name" = "wrong-name" -> "pod1-public-subnet"
        }
      ~ tags_all                                       = {
          ~ "Name" = "wrong-name" -> "pod1-public-subnet"
        }
        # (15 unchanged attributes hidden)
    }

Plan: 0 to add, 1 to change, 0 to destroy.
...
```

It tells you what has been modified and what will be deployed if the terraform plan is applied. Let's apply the deployment again:

```
terraform apply
```

and input `yes` when asked, and check the subnet is created on the AWS console.

Task 12. Terraform import

With the `import` command you can bring an existing AWS resource into your Terraform-managed infrastructure.

Once the resource is defined in your `main.tf` file you can instruct terraform to import a specific instance of the resource by passing its id.

Let's first remove your terraform state file. By doing this terraform loses the state of you infrastructure.

```
rm terraform.tfstate
```

If you were to use `terraform apply`, terraform would have to create a new instance of your subnet and would fail as there will be an overlap on the cidr_block.

Instead let's instruct terraform to `import` the previously created AWS subnet into terraform.

```
terraform import aws_subnet.this subnet-0783f8621ad659e83
```

You should have an equivalent statement:

```
aws_subnet.this: Importing from ID "subnet-0a45cf4cf75814797"...
aws_subnet.this: Import prepared!
  Prepared aws_subnet for import
aws_subnet.this: Refreshing state... [id=subnet-0a45cf4cf75814797]

Import successful!

The resources that were imported are shown above. These resources are now
in
your Terraform state and will henceforth be managed by Terraform.
```

Task 13. Terraform show

When you run `terraform show` you can inspect the configuration state as it is stored by Terraform. You should have something similar to:

```
terraform show
```

You should have an equivalent statement:

```
terraform show
# aws_subnet.this:
resource "aws_subnet" "this" {
    arn                                            = "arn:aws:ec2:us-east-
1:REDACTED:subnet/subnet-05a8e65d3872e6c1a"
    assign_ipv6_address_on_creation                = false
    availability_zone                              = "us-east-1f"
    availability_zone_id                           = "use1-az5"
    cidr_block                                     = "192.168.1.0/24"
    enable_dns64                                   = false
    enable_resource_name_dns_a_record_on_launch    = false
    enable_resource_name_dns_aaaa_record_on_launch = false
    id                                             = "subnet-
05a8e65d3872e6c1a"
    ipv6_native                                    = false
    map_customer_owned_ip_on_launch                = false
    map_public_ip_on_launch                        = false
    owner_id                                       = "REDACTED"
    private_dns_hostname_type_on_launch            = "ip-name"
    tags                                           = {
        "Name" = "pod9-public-subnet"
    }
    tags_all                                       = {
        "Name" = "pod9-public-subnet"
    }
    vpc_id                                         = "vpc-REDACTED"
}
```

Task 14. Terraform destroy

When you're done experimenting with Terraform, either at the end of this section, or at the end of future sections, it's a good idea to remove all of the resources you created so that AWS doesn't't charge you for them. Because Terraform keeps track of what resources you created, cleanup is simple. All you need to do is run the destroy command:

```
terraform destroy
```

You should get a similar output:

```
aws_subnet.this: Refreshing state... [id=subnet-026552bf9f58e52f9]

Terraform used the selected providers to generate the following execution
plan. Resource actions are indicated with the following
symbols:
  - destroy

Terraform will perform the following actions:

  # aws_subnet.this will be destroyed
  - resource "aws_subnet" "this" {
      - arn                                            = "arn:aws:ec2:us-
east-1:REDACTED:subnet/subnet-026552bf9f58e52f9" -> null
      - assign_ipv6_address_on_creation                = false -> null
      - availability_zone                              = "us-east-1a" ->
null
      - availability_zone_id                           = "use1-az4" ->
null
      - cidr_block                                     = "192.168.1.0/24"
-> null
      - enable_dns64                                   = false -> null
      - enable_resource_name_dns_a_record_on_launch    = false -> null
      - enable_resource_name_dns_aaaa_record_on_launch = false -> null
      - id                                             = "subnet-
026552bf9f58e52f9" -> null
      - ipv6_native                                    = false -> null
      - map_customer_owned_ip_on_launch                = false -> null
      - map_public_ip_on_launch                        = false -> null
      - owner_id                                       = "374187570784" ->
null
      - private_dns_hostname_type_on_launch            = "ip-name" -> null
      - tags                                           = {
          - "Name" = "podx-public-subnet"
        } -> null
      - tags_all                                       = {
          - "Name" = "podx-public-subnet"
        } -> null
      - vpc_id                                         = "vpc-REDACTED" ->
null
```

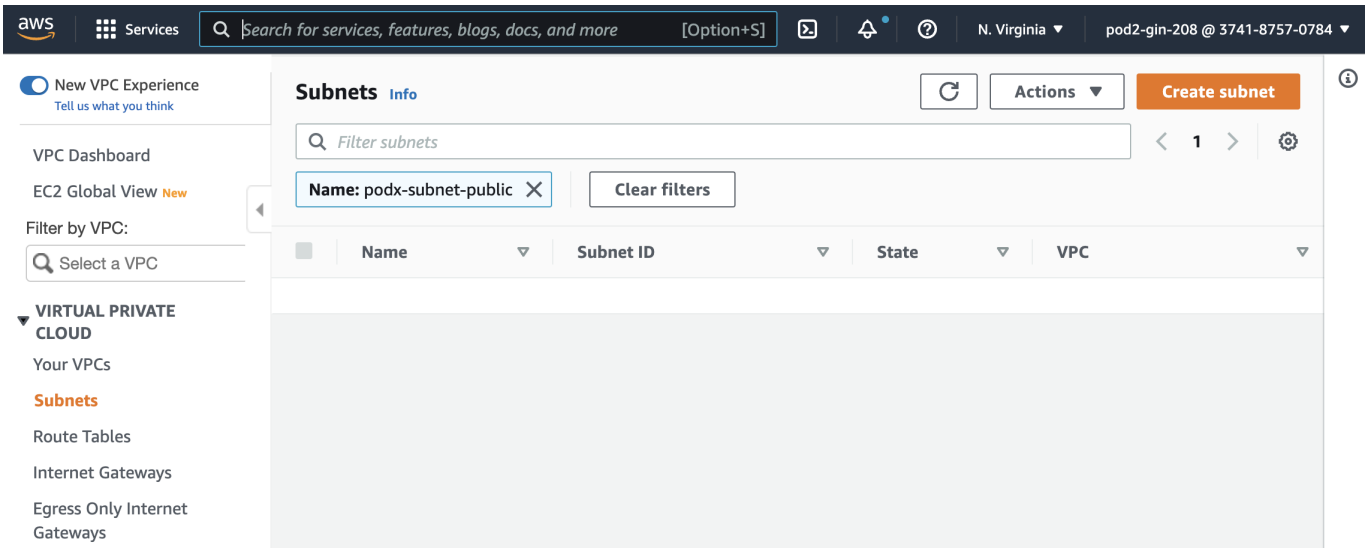
```
    }

Plan: 0 to add, 0 to change, 1 to destroy.

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value:
```

It goes without saying that you should rarely, if ever, run destroy in a production environment! There's no "undo" for the destroy command, so Terraform gives you one final chance to review what you're doing, showing you the list of all the resources you're about to delete, and prompting you to confirm the deletion. If everything looks good, type `yes` and hit Enter; Terraform will build the dependency graph and delete all of the resources in the correct order, using as much parallelism as possible. In a minute, your AWS account should be clean again.

You should get a similar output:

```
aws_subnet.this: Destroying... [id=subnet-026552bf9f58e52f9]
aws_subnet.this: Destruction complete after 0s

Destroy complete! Resources: 1 destroyed.
```

And you can check that there is no more subnet in the AWS Console:



Task 15. Create Terraform variables

-Task 15.1-

copy your current `main.tf` script in another folder named `create-subnet-with-variables`

-Task 15.2-

Input variables let you customize aspects of Terraform modules without altering the module's own source code. This allows you to share modules across different Terraform configurations, making your module composable and reusable.

Let's define the following variables for our script:

- region
- vpc_id
- cidr_block
- subnet_name

They are all of type string so it can be defined in a `variable.tf` definition file like this:

```
variable "region" {
  type = string
}
variable "vpc_id" {
  type = string
}
variable "cidr_block" {
  type = string
}
variable "subnet_name" {
  type = string
}
```

Each input variable accepted by a module must be declared using a `variable` block. The label after the `variable` keyword is a name for the variable, which must be unique among all variables in the same module. This name is used to assign a value to the variable from outside and to reference the variable's value from within the module. The name of a variable can be any valid identifier except the following: source, version, providers, count, for_each, lifecycle, depends_on, locals. These names are reserved for meta-arguments in module configuration blocks, and cannot be declared as variable names.

To set lots of variables, it is more convenient to specify their values in a variable definitions file named `terraform.tfvars`. A variable definitions file uses the same basic syntax as Terraform language files, but consists only of variable name assignments.

```
region = "us-east-1"
vpc_id = "vpc-REDACTED"
cidr_block = "192.168.1.0/24"
subnet_name = "podx-public-subnet"
```

Task 16. Re-write your script with variable references

The new main.tf file is re-written to leverage the variables definition file as they are know accessible through the `var` prefix:

```
#Configure aws provider
provider "aws" {
  region = var.region
}

#Create a subnet
resource "aws_subnet" "this" {
  vpc_id     = var.vpc_id
  cidr_block = var.cidr_block

  tags = {
    Name = var.subnet_name
  }
}
```

Within the module that declared a variable, its value can be accessed from within expressions as var., where matches the label given in the declaration block. Note: Input variables are created by a variable block, but you reference them as attributes on an object named var.

Task 17. Check the create-subnet still work

Check that the re-factored code is still working fine by exercising Terraform workflow:

1. terraform init
2. terraform plan
3. terraform apply
4. terraform destroy

Task 18. Let's use datasource and output in Terraform

copy your current `create-subnet-with-variables` folder in another folder named `create-subnet-with-datasource`.

Keep only the files:

- `main.tf`
- `variables.tf`
- `terraform.tfvars`

You can check everything is cleaned-up in your folder with a `tree . -all` command. If not installed do so with `sudo apt install tree`

```
tree . -all
.
├── main.tf
├── terraform.tfvars
└── variables.tf
```

Task 19. Add datasource to main script

Instead of defining the `vpc_id` as a variable we can use Terraform to read the `aws_vpc` resource we created manually in order to retrieve the configured object structure. Data sources allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions. A data source is accessed via a special kind of resource known as a data resource, declared using a data block

The `main.tf` file is modified like this:

```
#Configure aws provider
provider "aws" {
  region = var.region
}

#Read vpc resource
data "aws_vpc" "this" {
  filter {
    name   = "tag:Name"
    values = var.vpc_name
  }
}

#Create a subnet
resource "aws_subnet" "this" {
  vpc_id     = data.aws_vpc.this.id
  cidr_block = var.cidr_block

  tags = {
    Name = var.subnet_name
  }
}

#Capture vpc_id in output variable
output "vpc_id" {
  value = data.aws_vpc.this.id
}

#Capture vpc_cidr_block in output variable
output "vpc_cidr_block" {
  value = data.aws_vpc.this.cidr_block
}
```

A `data` block requests that Terraform read from a given data source `("aws_vpc")` and export the result under the given local name `("this")`. The name is used to refer to this resource from elsewhere in the same Terraform module, but has no significance outside of the scope of a module.

The data source and name together serve as an identifier for a given resource and so must be unique within a module.

Within the block body (between `filter { }`) are query constraints defined by the data source. Most arguments in this section depend on the data source, and indeed in this example `tag:Name` is the argument defined specifically for the aws_vpc data source.

When the `aws_subnet` resource is created, we can now refer to the `aws_vpc` data source to get the `vpc_id` configuration argument.

Therefore note the `vpc_id = data.aws_vpc.this.id` in the resource configuration definition.

Task 20. Modify variables and values

As we are not using the vpc_id variable anymore but the vpc_name, it has to be defined in the `variables.tf` and `terraform.tfvars` files.

The modified files look like the following:

- variables.tf

```
variable "region" {
  type = string
}
variable "vpc_name" {
  type = list
}
variable "cidr_block" {
  type = string
}
variable "subnet_name" {
  type = string
}
```

- terraform.tfvars

```
region = "us-east-1"
vpc_name = ["pod2-vpc"]
cidr_block = "192.168.1.0/24"
subnet_name = "podx-public-subnet"
```

Remark. Note that the variable *vpc_name* is a "list" and not a string. This is simply due to the AWS API which takes a list as arguments ("values", plural, with an S). The parameter in the '.tfvars' file is thus defined with square brackets ["pod2-vpc"], specifying a list (as in many other languages).

Task 21. Check the create-subnet still work

Check that the re-factored code is still working fine by exercising Terraform workflow: 1. terraform init, 2. terraform plan, 3. terraform apply and 4. terraform destroy.

Note that we are also using a new Terraform construct call `output` in the `main.tf` script. Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use. Output values are similar to return values in programming languages. Each output value exported by a module must be declared using an output block. The label immediately after the output keyword is the name, which must be a valid identifier.

In our case we have defined the following outputs:

```
#Capture vpc_id in output variable
output "vpc_id" {
  value = data.aws_vpc.this.id
}

#Capture vpc_cidr_block in output variable
output "vpc_cidr_block" {
  value = data.aws_vpc.this.cidr_block
}
```

which allow us to display the following information when applying the terraform script:

```
aws_subnet.this: Creating...
aws_subnet.this: Creation complete after 1s [id=subnet-01c421b0f08a51037]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

vpc_cidr_block = "192.168.0.0/16"
vpc_id = "vpc-REDACTED"
```

Task 22. Create 2x subnets script

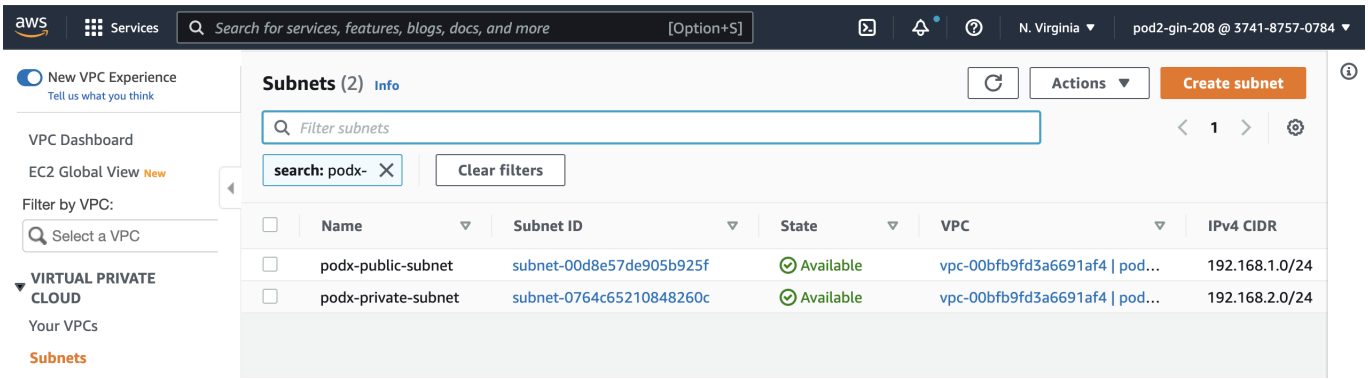
Based on what you have seen from the previous steps, please define and test a terraform script which create two subnets like:

- subnet1:
  - tag:Name = "podx-public-subnet"
  - cidr_block = "192.168.1.0/24"
- subnet2:
  - tag:Name = "podx-private-subnet"
  - cidr_block = "192.168.2.0/24"

Ideally create a folder named `create-2-subnets` to place your script and other files.

Task 23. Test your script

Check that the code is working fine by exercising Terraform workflow: 1. terraform init, 2. terraform plan, 3. terraform apply and 4. terraform destroy. Please check on the AWS console you have achieved the right results, you should see something similar to:



Task 24. Create a custom route table

For this step it is asked to define a Terraform script (with associated variables definition) to create a route table with the following attributes:

- tag:Name = "podx_public_route_table"
- associate the route table with already provisioned `podx-public-subnet`
- add a default route `0.0.0.0/0` via an internet gateway to be provision

We foresee you will use:

- the provider `aws`
- datasource with the appropriate filters:
  - aws_vpc
  - aws_subnet
- resource with the appropriate config arguments:
  - aws_internet_gateway
  - aws_route_table
  - aws_route_table_association

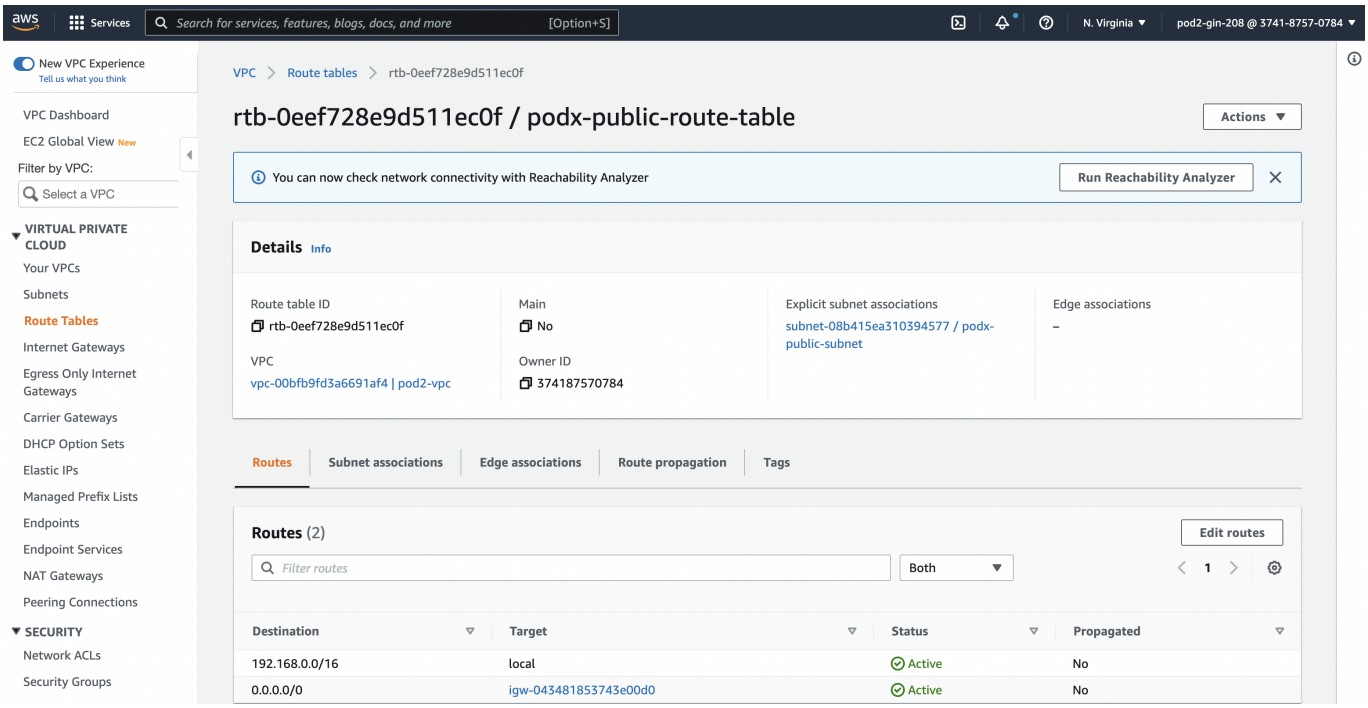
Ideally create a folder named `create-route-table` to place your script and other files.

To help with the exercise don't hesitate to have a look to the Terraform documentation, for example on:

- `aws_internet_gateway`: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/internet_gateway
- `aws_route_table`: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/route_table
- `aws_route_table_association`: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/route_table_association

Task 25. Test your create-route-table script

Check that the code is working fine by exercising Terraform workflow: 1. terraform init, 2. terraform plan, 3. terraform apply and 4. terraform destroy. Please check on the AWS console you have achieved the right results, you should see something similar to:



Task 26. Create a security group

For this task it is asked to define a Terraform script (with associated variables definition) to create a security group with the following attributes:

- Name = "podx-ssh-security-group"
- an ingress rule which allows ssh for IPv4
- an egress rule which allows any IPv4 traffic

We foresee you will use:

- the provider `aws`
- datasource with the appropriate filters:
  - aws_vpc
- resource with the appropriate config arguments:
  - aws_security_group

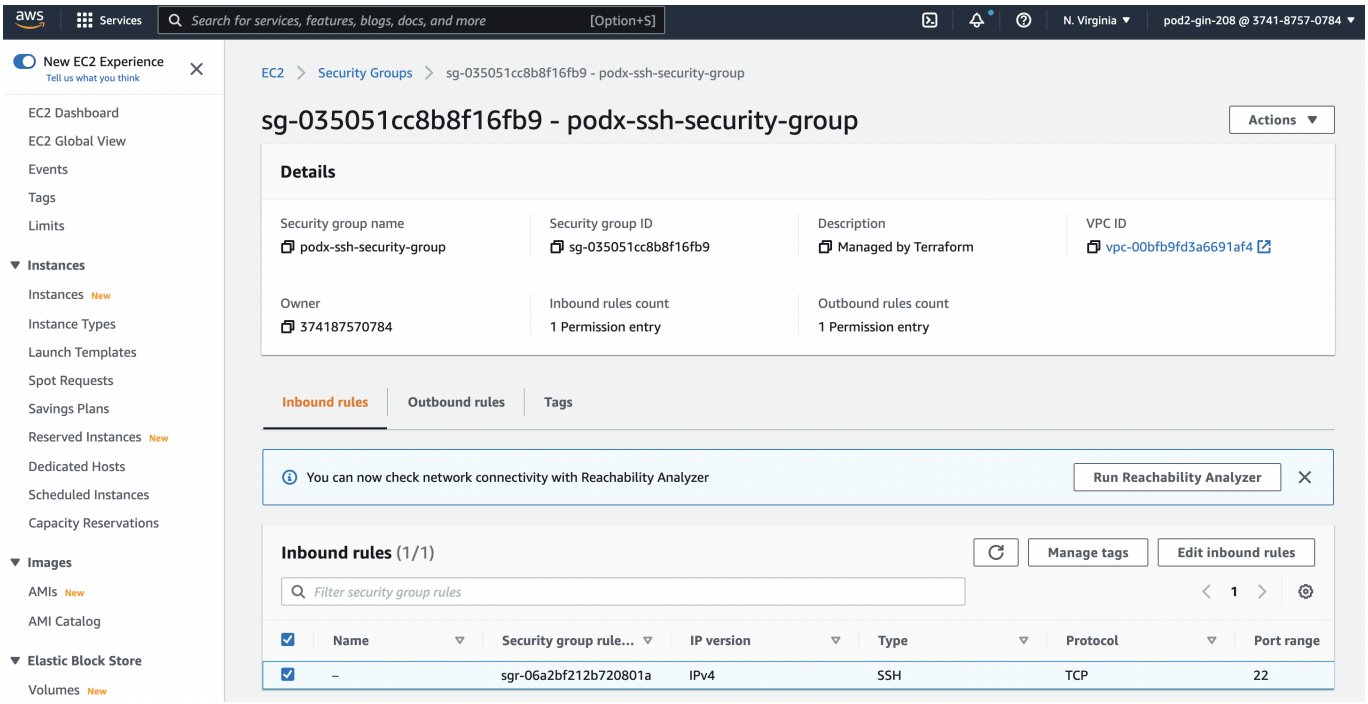
Ideally create a folder named `create-security-group` to place your script and other files.

To help with the exercise don't hesitate to have a look to the Terraform documentation, for example on:

- `aws_security_group`: https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/security_group

Task 27. Test your script

Check that the code is working fine by exercising Terraform workflow: 1. terraform init, 2. terraform plan, 3. terraform apply and 4. terraform destroy. Please check on the AWS console you have achieved the right results, you should see something similar to:



Task 28. Create a key pair

For this task it is asked to define a Terraform script (with associated variables definition) to create a key pairs with the following attributes:

- Name = "podx-vm-key"
- algorithm = "RSA"
- rsa_bits = 4096

We foresee you will use:

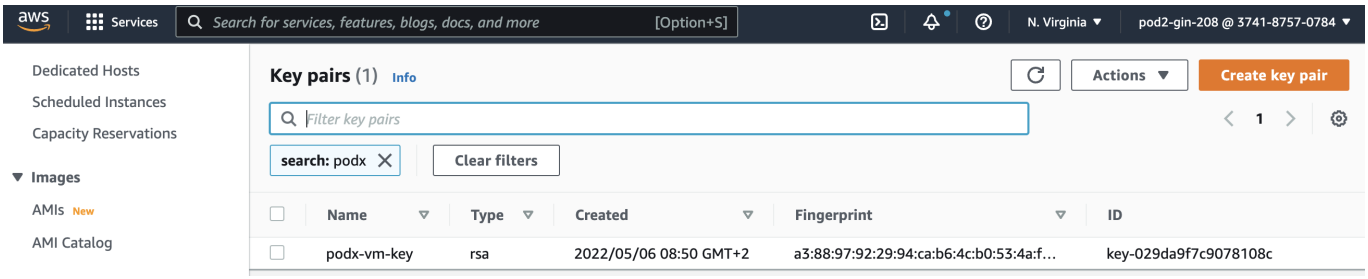
- the provider `aws`
- resource with the appropriate config arguments:
  - tls_private_key: to generate the RSA/4096 key
  - local_file: to save the content `private_key_pem` of the `tls_private_key` resource
  - aws_key_pair: to create the key pair in AWS including the `public_key_openssh` of the `tls_private_key` resource

Ideally create a folder named `create-key-pair` to place your script and other files.

To help with the exercise don't hesitate to have a look to the Terraform documentation.

Task 29. Test your script

Check that the code is working fine by exercising Terraform workflow: 1. terraform init, 2. terraform plan, 3. terraform apply and 4. terraform destroy. Please check on the AWS console you have achieved the right results, you should see something similar to:



Check also that the private key has been saved locally in the folder where you performed the `terraform apply` action with the command `cat podx-vm-key.pem`.
You should have something similar to that:

```
-----BEGIN RSA PRIVATE KEY-----
-----END RSA PRIVATE KEY-----
```

Task 30. Create an EC2 instance

For this task it is asked to define a Terraform script (with associated variables definition) to create an EC2 instance with the following attributes:

- Name = podx-vm
- Image = ubuntu-focal-20.04-amd64-server-20211129
- Subnet = podx-public-subnet
- Security group = podx-ssh-security-group
- Instance type = t2.nano
- Associate public IP address = true
- Source Destination Check = false

We foresee you will use:

- the provider `aws`
- datasource with the appropriate filters:
    - aws_security_group selected with its `name`
    - aws_key_pair selected with its `key_name`
    - aws_subnet filtered with `tag:Name`
    - aws_ami selected with `owners` and filtered with `tag:architecture` and `tag:name`
- resource with the appropriate config arguments:
    - aws_instance

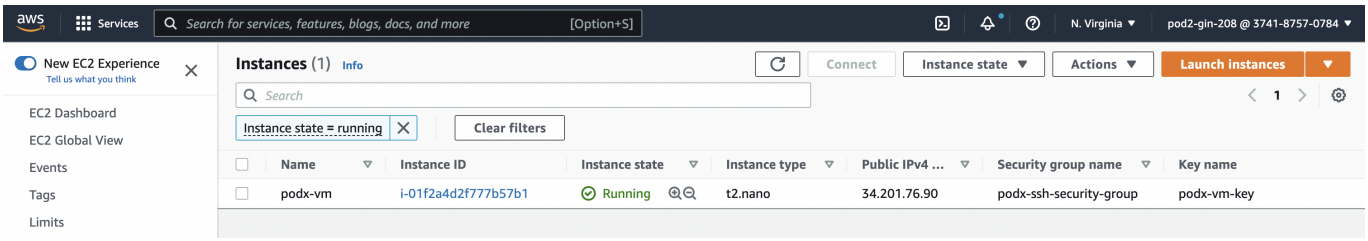
For convenience, you can also capture the public IP address of the instance using an output returning the value `aws_instance.this.public_ip`

Ideally create a folder named `create-ec2-instance` to place your script and other files.

To help with the exercise don't hesitate to have a look to the Terraform documentation !

Task 31. Test your script

Check that the code is working fine by exercising Terraform workflow: 1. terraform init, 2. terraform plan, 3. terraform apply and 4. terraform destroy. Please check on the AWS console you have achieved the right results, you should see something similar to:



Please check also you can access via ssh the EC2 instance:

```
ssh -i podx-vm-key.pem ubuntu@34.201.76.90
```

You should get access to the instance

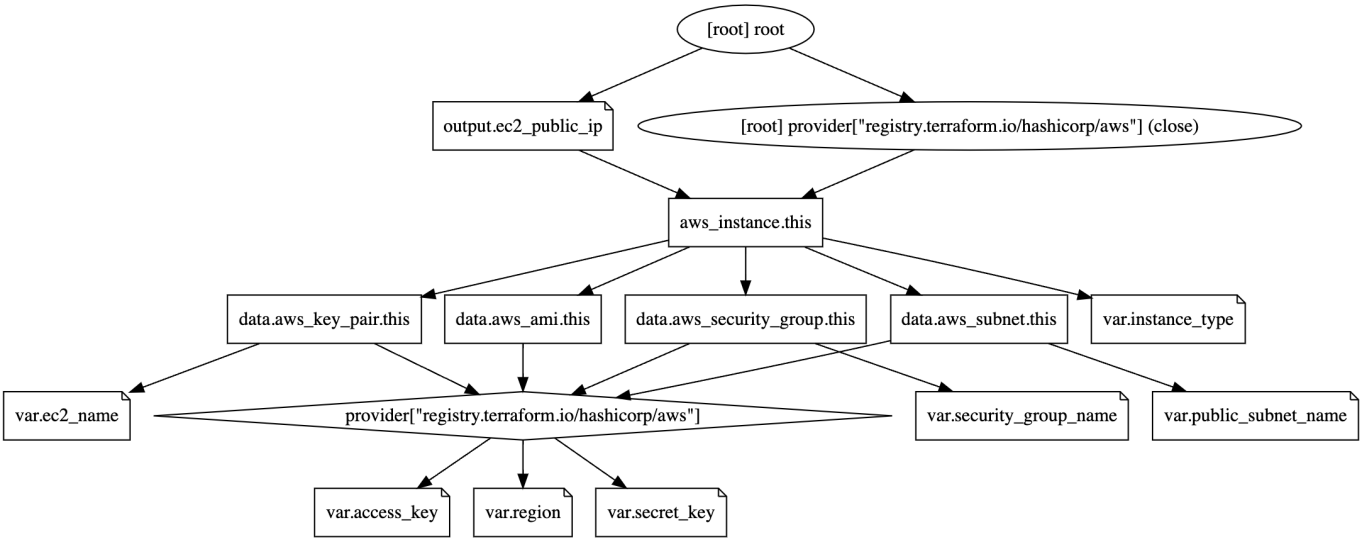
```
Welcome to Ubuntu 20.04.3 LTS (GNU/Linux 5.11.0-1022-aws x86_64)

* Documentation:  https://help.ubuntu.com
* Management:     https://landscape.canonical.com
* Support:        https://ubuntu.com/advantage

  System information as of Fri May  6 08:54:45 UTC 2022

  System load:  0.0               Processes:             98
  Usage of /:   18.4% of 7.69GB   Users logged in:       0
  Memory usage: 42%               IPv4 address for eth0: 192.168.1.46
  Swap usage:   0%


1 update can be applied immediately.
To see these additional updates run: apt list --upgradable


The list of available updates is more than a week old.
To check for new updates run: sudo apt update

Last login: Fri May  6 08:00:45 2022 from 86.242.114.187
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-192-168-1-46:~$
```

Note the prompt of the instance indicating the IP address which has been allocated to the VM on the `podx-public-subnet` subnet.

Task 32. Terraform graph

Terraform manages for you the dependency resolution, i.e. it is capable to understand which resources have to be deployed first as others depend of it, as well as deploying independent resources in parallel.

Execute `terraform graph > digraph.dot` to save the dependency graph of the EC2 instance deployment in `digraph.dot` file. You can visualize the graph with Graphviz tool, and you should have something similar to:



LAB : Ansible Lab

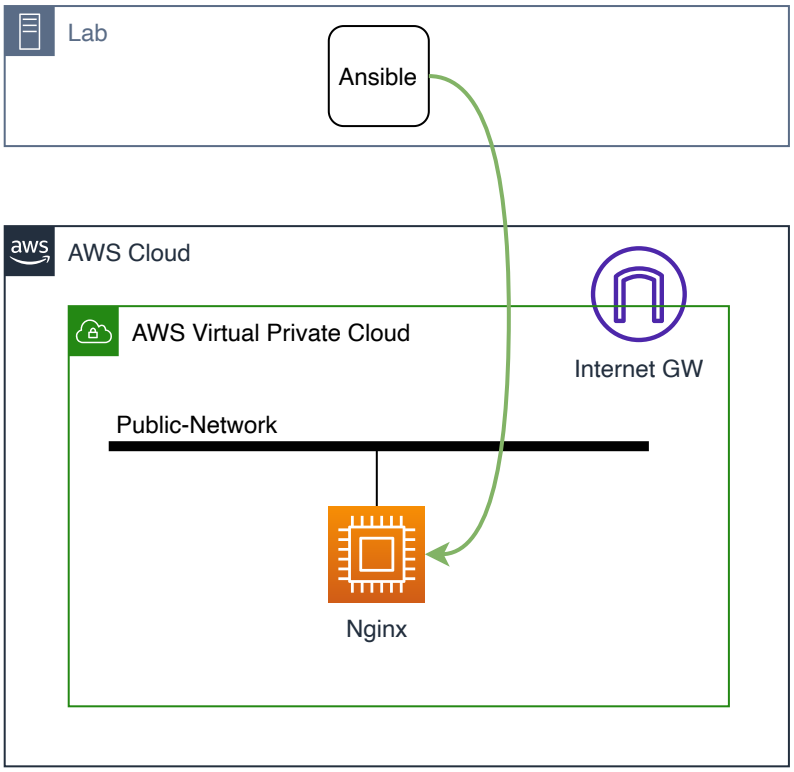
- Prerequisite and objective -

Nginx is a widely used open-source web server and reverse proxy server. It's known for its high performance, scalability, and efficient handling of concurrent connections. Nginx can serve static content, act as a load balancer, and perform other tasks like SSL termination and caching, making it a popular choice for serving websites and applications.

Ansible is a powerful automation tool that simplifies and accelerates the process of setting up and configuring software on remote servers. When it comes to installing and configuring Nginx, Ansible can significantly streamline these tasks, making them repeatable, consistent, and efficient.

This lab involves file editing. It's recommended to use the Lab Dashboard Code Editor and installing `Ansible` extension for VSCode

- Lab Diagram -



Insight 1. Ansible Connectivity

Ansible works by connecting to what you want automated and pushing programs that execute instructions that would have been done manually. These programs utilize Ansible modules that are written based on the specific expectations of the endpoint's connectivity, interface, and commands. Ansible then executes these modules (over standard SSH by default), and removes them when finished (if applicable). There are no additional servers, daemons, or databases required.

We differentiate the central node where ansible is running : `control node` from the node you are configuring the `managed hosts`

- Control node -

Unlike other configuration management utilities, ansible uses an agentless architecture (or 'temporary agent'). Ansible itself only needs to be installed on the host it will run referred as the `control node`.

Ansible installation on the control node requires Python 3.8 or newer installed. Ansible is packaged as a python library.

- Managed hosts -

Although you do not need a daemon on your managed nodes, you do need a way for Ansible to communicate with them. For most managed nodes, Ansible makes a connection over SSH and transfers modules using SFTP. If possible it's advisable for the managed host to run Python 3.5+.

Task 2. Ansible Installation

-Task 2.1-

Verify python version and install Ansible:

This commands are run on Code Editor terminal

```
python3 --version
```

Expected output

```
Python 3.9.2
```

-Task 2.2-

Install requirements

```
sudo apt install python3.9-venv python3-pip
```

Install pipx

```
python3 -m pip install pipx
```

Add pipx packages folder to PATH

```
echo 'export PATH=/home/coder/.local/bin:$PATH' >> ~/.bashrc
source ~/.bashrc
```

-Task 2.3-

Install ansible

```
pipx install ansible-core
```

-Task 2.4-

Verify ansible installation

```
ansible --version
```

Expected output

```

```

```
ansible [core 2.12.6]
  config file = None
  configured module search path =
['/home/ubuntu/.ansible/plugins/modules',
'/usr/share/ansible/plugins/modules']
  ansible python module location = /home/ubuntu/.local/lib/python3.8/site-
packages/ansible
  ansible collection location =
/home/ubuntu/.ansible/collections:/usr/share/ansible/collections
  executable location = /home/ubuntu/.local/bin/ansible
  python version = 3.8.10 (default, Mar 15 2022, 12:22:08) [GCC 9.4.0]
  jinja version = 3.1.2
  libyaml = True
```

Task 3. Ansible project folder structure

A well-organized Ansible project folder structure can greatly improve the maintainability and scalability of your automation efforts. An Ansible role is a reusable and modular unit of automation in Ansible. Roles are used to organize and encapsulate the configuration tasks and related files required to manage specific aspects of a system or application. Roles provide a structured way to manage and share automation code and make it easier to maintain and scale Ansible playbooks.

Generate a new empty role named `ansible-project` with the following command

```
ansible-galaxy role init ansible-project
```

You should get the following created structure :

```
ansible-project       # This hierarchy represents an Ansible project
├── README.md         # Documentation of your role
├── defaults          # Contains the default variables for the role.
│   └── main.yml
├── files             # Folder for static files
├── handlers          # Contains handlers, which are tasks that respond to
a notify directive.
│   └── main.yml
├── meta              # Metadata for the role, like author, support,
dependencies.
│   └── main.yml
├── tasks             # Generic tasks for playbooks
│   └── main.yml
├── templates         # Files for use with the template resource
├── tests             # Contains tests for the role.
│   ├── inventory
│   └── test.yml
└── vars              # Variables files to be used in playbooks
    └── main.yml
```

Insight 4. Inventory

A host inventory defines which hosts Ansible manages. Hosts may belong to groups which are typically used to identify the hosts role in the datacenter. A host can be a member of more than one group.

There are two ways in which host inventories can be defined. A static host inventory may be defined by a text file or a dynamic host inventory may be be generated from outside providers.

Note: it's a good practice to use SSH key-pair authentication from Control Node to Managed Hosts. And to enable passwordless sudo for the user ansible is using to connect to the Managed host.

- Dynamic host inventory -

Ansible host inventory information can also be dynamically generated. Source for dynamic inventory information include public/private cloud providers like AWS, GCP, Openstack or configuration management databases (CMDB).

A dynamic host inventory plugin is nothing but a python script that returns an Ansible formatted inventory structure.

- Static host inventory -

An ansible static host inventory is define in an ini-like text file or in YAML format. In INI format, each section defines one group of hosts (a host group). Each section starts with a host group name enclosed in square brackets (`[]`). Then host entries for each managed host in the group are listed, one host per line.

A valid host entry follow the following syntax:

```
[<alias>] (<IP> | <Hostname>) [<parameter>=<value>...]
```

Task 5. Create a static inventory

- Task 5.1 -

Inside your `ansible-project` folder create a new file named `inventory.ini` file with the following content to connect to your nginx server. **Update the ansible_host variable with your EC2 instance VM IP address and update the path to your SSH key**

```
[webservers]
nginx ansible_host=YOUR_NGINX_VM_IP_ADDRESS

[webservers:vars]
ansible_ssh_private_key_file=~/.ssh/my-private-key.pem
ansible_user=ubuntu
```

This inventory defines :

- `webeservers` : The group name
- `nginx` : An Ansible managed host
  - `ansible_host` : The IP address or DNS entry to reach the managed host
- `webservers:vars` : A list of variables for the `webserver` group
  - `ansible_ssh_private_key_file` : The SSH key used to connect to your instances
  - `ansible_user` : The user to connect to your instances

The full list of Ansible inventory parameters is available on Ansible documentation.

- Task 5.2 -

The command line `ansible-inventory` verifies your inventory and can display it in different formats.

Issue the following command to validate the syntax of your inventory and its content structure

```
ansible-inventory -i inventory.ini --list
```

You should get the following output :

```
{
    "_meta": {
        "hostvars": {
            "nginx": {
                "ansible_host": "REDACTED",
                "ansible_ssh_private_key_file": "~/.ssh/my-private-
key.pem",
                "ansible_user": "ubuntu"
            }
        }
    },
    "all": {
        "children": [
            "ungrouped",
            "webservers"
        ]
    },
    "webservers": {
        "hosts": [
            "nginx"
        ]
    }
}
```

- Task 5.3 -

You can validate your inventory by asking ansible to connect to `all` the managed hosts you defined using a builtin Ansible module named `ping`

```
ansible all -i inventory.ini --module-name ping
```

Expected output

```
nginx | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
```

Insight 6. YAML

Ansible playbooks are written using the YAML (Yet Another Markup Language) language. Therefore it's necessary to understand the basics of YAML syntax to compose Ansible playbook.

YAML was designed primarily for the representation of data structure such as lists and associative arrays in an essay to write, human-readable format. This design objective is accomplished primarily by abandoning traditional enclose syntax, such as brackets, braces (JSON) or opening and closing tags (XML). Instead YAML data hierarchy structures are maintained using outline indentation. There is no strict requirement regarding the number of space characters used for indentation other than elements must be further indented than their parents to indicate nested relationships.

Note : Indentation can only be performed using space character (no TAB). Indentation is very critical to the proper interpretation of YAML. **It's highly recommended to use an IDE (You will suffer)**. Users of VIM can alter the TAB key when editing Yaml files to perform 2 spaces instead.

```
autocmd FileType yaml setlocal ai ts=2 sw=2 et
```

- Strings -

string in YAML do not require enclosure in quotations. If desired, strings can enclosed in either double-quotes or single-quotes.

```
description: a valid string
```

```
description: 'a valid string'
```

```
description: "a valid string"
```

- List -

```
webservers:
  - web1
  - web2
  - web3
```

Python equivalent:

```
webeservers = ['web1','web2','web3']
```

- Dictionaries -

```
lab:
  name: devops
  subject: ansible
  date: 2023-09-01
```

Python equivalent:

```
lab = {
  'name' : 'devops',
  'subject' : 'ansible',
  'date' : '2023-09-01'
}
```

Insight 7. Ansible Modules and Playbook

Modules are programs that Ansible uses to perform operations on managed hosts. They are ready-to-use tools designed to perform specific operations.

There are three types of Ansible modules:

1. Core modules : Ansible built-in modules shipped with ansible an maintained by the Ansible team
2. Galaxy Modules : Modules created/maintained by the community or the industry. They are distributed via 'ansible packet manager' ansible-galaxy. Each ansible release often promote to core most used community modules.
3. Custom Modules : Your own modules.

Ansible automation strategy is to converge to the target intent configuration state. Some tasks might be expected to fail during your initial deployment. You should be able to re-run the all tasks until configuration state is reached.

In that consideration and because Ansible is stateless, modules *must* be idempotent. Each module *should* implement a logic to test if they need to be executed or not, modules must have a reliable predictive and consistent output. This is the main limitation of running Ansible in production. When using custom/community modules extensive testing of your automation plays must be done to prevent side effects on your infrastructure.

- All ansible modules list available here

- Modules usage -

Technically a module is nothing more than a python script that accepts a dictionary as parameter input and output an Ansible formatted dictionary structure.

A module is invoked in ansible using the following YAML syntax :

```
ansible.builtin.copy:          #module name
  src: /etc/foo.conf           #module parameters
  dest: /etc/bar.conf
```

- Ansible copy module parameters github

When using a module always refer to the module documentation and verify its parameters and usage.

Ansible modules can be used to perform operations on managed hosts using simple ad hoc commands (like the the ping module used earlier). While it is useful for simple operations ad hoc commands are not suited for the challenges of complex configuration management and declarative automation. Automating an infrastructure is the result of several `tasks`, performed by several `modules`. Several `tasks` are aggregated in a `playbook`.

Task 8. Create your first Playbook

- Task 8.1 -

A `Playbook` is a set of plays. A play must define a target inventory object (host or group) referred as `hosts` and a set of `tasks` performing that are a set of `modules`.

You are going to write a playbook to install a NGINX server on the target

Inside your ansible-nginx folder create a file named `nginx-install.yaml` with the following content :

```
---                                        # Conventionally YAML file
starts with '---'
- name: Simple nginx installation          # Descriptive play name
  gather_facts: yes                        # Gather `facts` on the target
host and store them in `ansible_facts` variable
  hosts: webservers                        # Target managed hosts, tasks
will run on/toward.

  tasks:                                   # List of tasks for the play
  - name: Install nginx package            # Name of the task
```

```
    ansible.builtin.apt:                    # Module name
      update_cache: yes
      pkg:
        - nginx
      state: present
    become: true                            # Privilege escalation for the
task
```

By default Ansible connect as the user defined in inventory/control node, for privilege escalation for a given task you can use the parameter `become: true` :

- Task 8.2 -

Execution of a playbook is done with the following command :

```
ansible-playbook -i inventory.ini nginx-install.yaml
```

Ansible treat each task atomically and will report a detailed report on success and status for each of them:

Expected output:

```
PLAY [Simple nginx installation]
**************************************************************************
*************************

TASK [Gathering Facts]
**************************************************************************
************************************
ok: [nginx]

TASK [Install nginx package]
**************************************************************************
***********
changed: [nginx]

PLAY RECAP
**************************************************************************
*************************************************
nginx                      : ok=2    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

Note:

- The command line parameter `-vvv` - very very verbose - can help with the troubleshooting.

Task 9. Verify NGINX installation

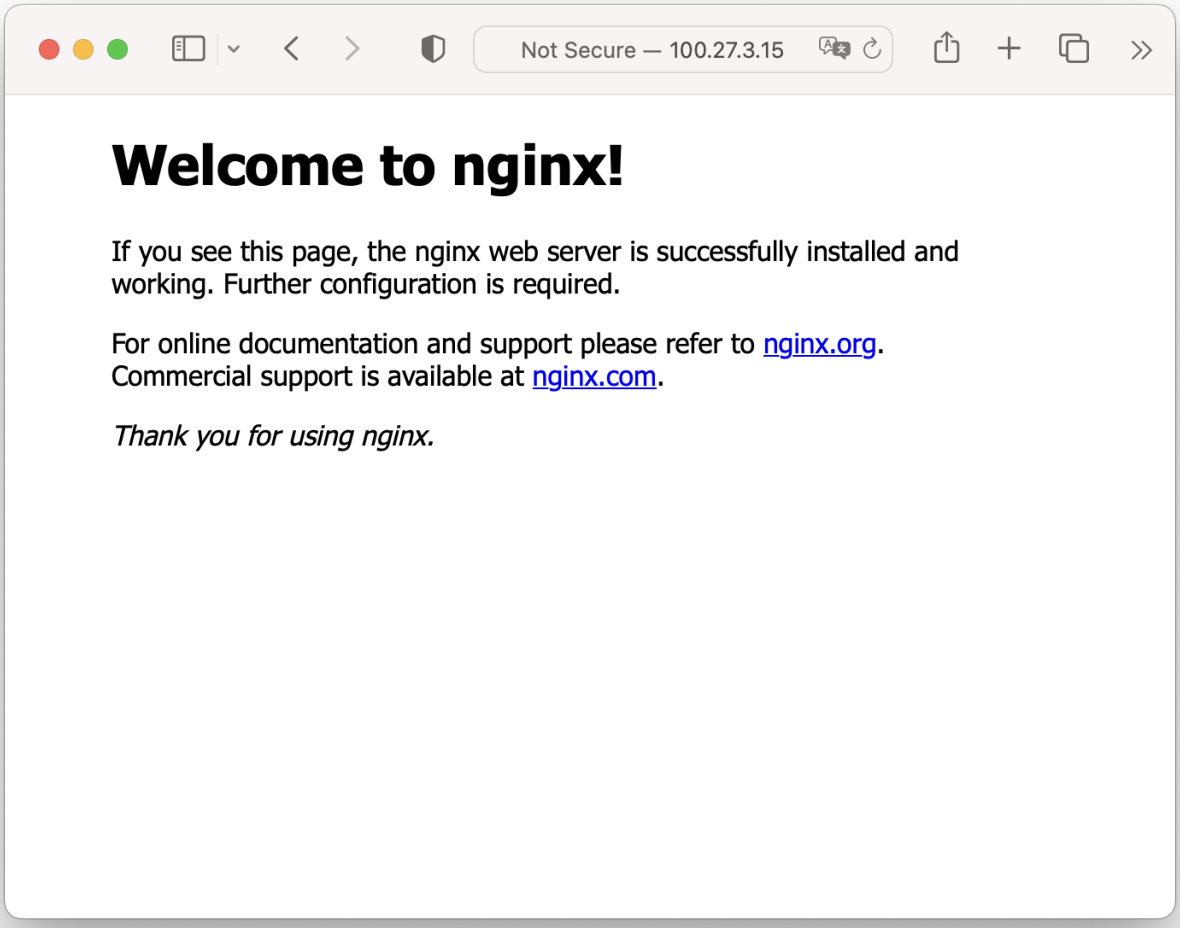
- Task 9.1 -

Nginx host a default static website on port 80. Make appropriate change to your VM to allow external connections on port 80.

- Task 9.2 -

Open a new tab on a web browser and use your VM instance IP address and port 80

If the installation was successful you should have this result :



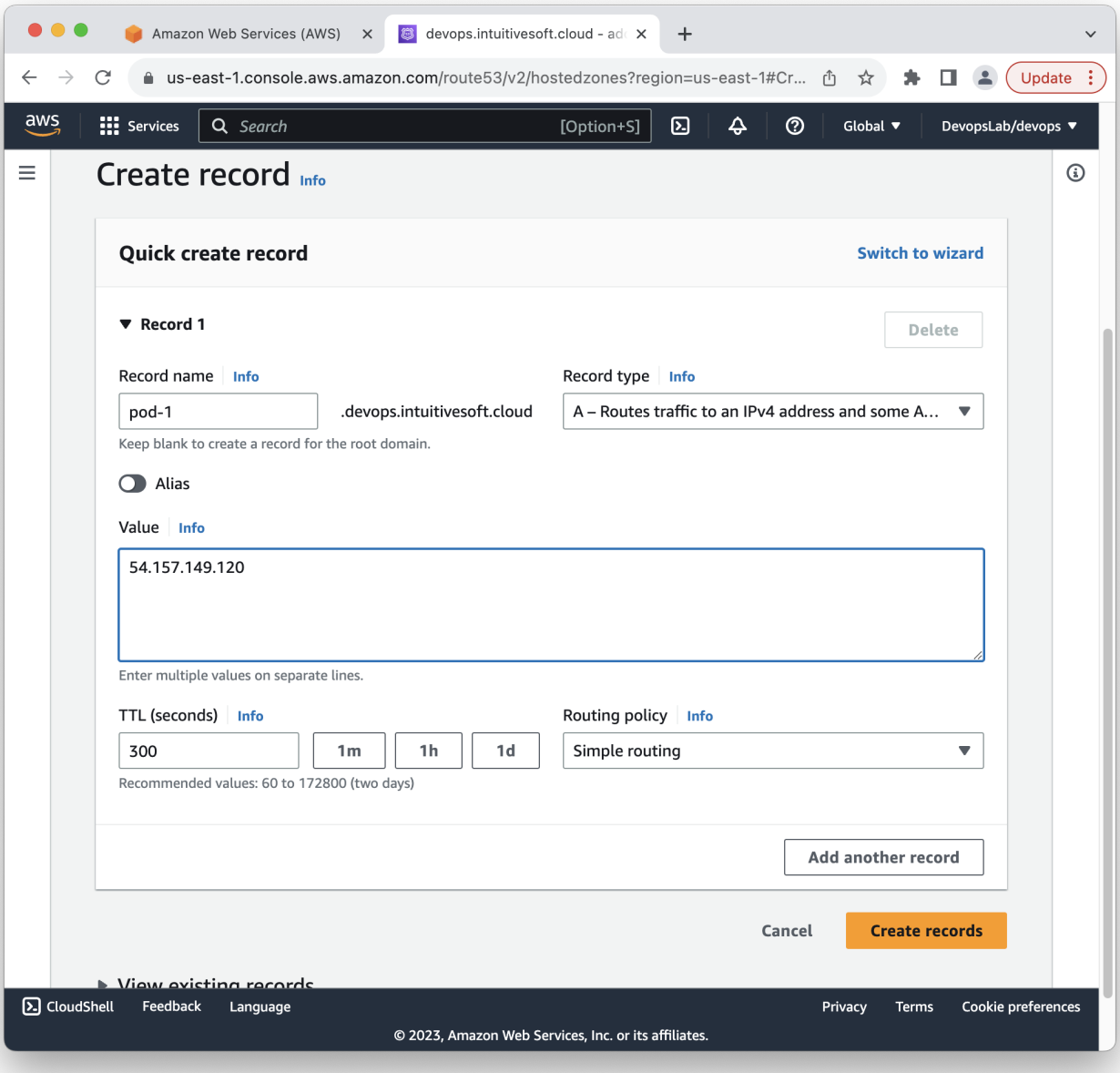
Task 10. Create a DNS entry for your NGINX VM

-Task 10.1-

Navigate to the `Route53 > Hosted zones` , or use the following link https://us-east-1.console.aws.amazon.com/route53/v2/hostedzones?region=us-east-1#ListRecordSets/Z03385262WMCAFO2KASN1

-Task 10.2-

Add a DNS record `A` to your EC2 instance public IP address



Insight 11. Variables

Ansible supports variables that can be used to store values that can reused throughout files in an entire Ansible project. Variables provide a convenient way to manage dynamic values for a given environment in your Ansible project.

To correctly interact with Ansible, you need to apprehend Ansible variable scope and precedence system.

- Scope -

Variables in Ansible can be defined in a bewildering variety of places. However this can be simplified to three basic scope levels:

- Global scope : Variables set from the command line or Ansible Configuration
- Play scope: Variables set in a play and related structure
- Host scope: Variables set on host groups and individual hosts by the inventory, fact gathering or tasks.

- Precedence -

At execution, your playbook will have access to the variables coming from the different scopes. If the same variable name is defined at more than one level, the higher wins.

At high level, here is the order of precedence from least to greatest :

1. inventory vars
2. playbook vars
3. command line vars

For reference, the full detailed precedence rules are available on ansible documentation

Task. 12 Define playbook variables

- Task. 12.1 -

Playbook variables can be defined in multiple ways. One of the simplest is to place it in a `vars` block at the beginning of a playbook. Create a new playbook file named `test-variables.yaml` inside your ansible project with the following content :

```
---
- name: A test on variables usage
  gather_facts: yes
  hosts: webservers

  vars:
    - user_name: devops
```

It defines a single variable named `user_name` with value `devops`.

- Task. 12.2 -

If you meant to reuse same group of variables in different plays, it's advised to define your variables in a dedicated file inside your project `vars` folder and import the file in the different plays:

Create a variable file named `generic.yaml` inside your project `vars` folder with the following content :

UPDATE THE DOMAIN NAME WITH YOUR VM DNS ENTRY

```
domain: YOUR-USERNAME.devops.intuitivesoft.cloud
```

- Task. 12.3 -

Then edit the `test-variables.yaml` playbook to add references of the generic.yml file :

```
---
- name:  A test on variables usage
```

```
  gather_facts: yes
  hosts: webservers
  become: true

  vars:
    - user_name: user-1

  vars_files:
    - vars/generic.yml
```

Task 13. Use variables in playbooks

- Task 13.1 -

Variables are referenced by placing the variable name in double curly braces.

Update the `test-variables.yaml` playbook and add a debug task to display information on the host

```
---
- name:  A test on variables usage
  gather_facts: yes
  hosts: webservers

  vars:
    - user_name: devops

  vars_files:
    - vars/generic.yaml

  tasks:
  - name: Print a message for debug
    ansible.builtin.debug:
      msg: "Connected to {{ domain }} - IP {{ ansible_host }} - For user
{{ user_name }}. Managed host is running {{ ansible_distribution }} {{
ansible_distribution_version }}"
```

Variables are defined :

- `user_name` is defined at the playbook level.
- `domain` is defined in the `vars/generic.yml` file.
- `ansible_host` is defined in the inventory.
- `ansible_distribution` and `ansible_distribution_version` are defined dynamically by the `gather_facts: yes` module.

Although Ansible is lenient, make sure variables are in a double quote string `"  "`, it will prevent errors later on. `{{ a_var }}` should be written `"{{ a_var }}"`.

You can have several variables in the same string.

- Task 13.2 -

Execute the playbook.

```
ansible-playbook -i inventory.ini test-variables.yaml
```

You should get the following output :

```
PLAY [A test on variables usage]
*******************************************************************************
*******

TASK [Gathering Facts]
*******************************************************************************
*****************
ok: [nginx]

TASK [Print a message for debug]
*******************************************************************************
*******
ok: [nginx] => {
    "msg": "Connected to user-1.devops.intuitivesoft.cloud - IP
100.27.3.15 - For user devops. Managed host is running Ubuntu 22.04"
}

PLAY RECAP
*******************************************************************************
*****************************
nginx                      : ok=2    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

Task 14. Create a virtual host Jinja2 template Nginx configuration file

Nginx virtual host, also known as server block, enables hosting multiple websites on a single server by directing incoming requests to different configurations based on domain names or IP addresses. Conventionally each host/site is defined in a dedicated configuration file.

You are going to dynamically generate the content of the configuration file using Jinja2.

Ansible uses the pythonic Jinja2 templating system to modify files before they are distributed to the managed hosts.

Note: Jinja2 is the key templating engine of Django to render HTML pages.

A jinja template is a text file with variables or logic expressions placed between tags or delimiters. At execution Jinja2 engine parses the text file for variables and renders it.

Variables are enclosed in double curly braces `{{ VAR }}`, much like variables in Ansible playbooks. While logic expressions are enclosed in `{% EXPR %}`.

- Task 14.1 -

Inside the `templates` folder create a file called `devops.conf.j2` with the following content :

```
server {
  listen 80 {{ 'default_server' if main_server }};
  listen [::]:80 {{ 'default_server' if main_server }};
  server_name {{ domain }};
  root /var/www/{{ domain }};
  location / {
    try_files $uri $uri/ =404;
  }
}
```

- Task 14.2 -

Create a new playbook file named `create-vhost.yaml` inside your ansible project with the following content :

```
---
- name: Configure a new virtualhost
  gather_facts: yes
  hosts: webservers
  become: true

  vars_files:
    - vars/generic.yaml

  vars:
    - main_server : true

  tasks:
  - name: "create www directory for your virtualhost"
    ansible.builtin.file:
      path: /var/www/{{ domain }}
      state: directory
      mode: '0775'
      owner: "{{ ansible_user }}"
      group: "{{ ansible_user }}"

  - name: delete default nginx site configuration
    ansible.builtin.file:
      path: /etc/nginx/sites-enabled/default
      state: absent
    notify: restart nginx

  - name: deploy new site configuration
    ansible.builtin.template:
      src: templates/devops.conf.j2
      dest: /etc/nginx/sites-enabled/{{ domain }}
      owner: root
      group: root
```

```
      mode: '0644'
    notify: restart nginx

  handlers:
    - name: restart nginx
      service:
        name: nginx
        state: restarted
```

Notes:

- The ordering of the content within a playbook is important, tasks are evaluated top to bottom.
- `handlers` define tasks that are played only if any playbook task `changed` and `notify` it.

- Task 14.3 -

Deploy your virtualhost

```
ansible-playbook -i inventory.ini create-vhost.yaml
```

```
PLAY [Configure a new virtualhost]
*********************************************************************************
*************************

TASK [Gathering Facts]
*********************************************************************************
************************************
ok: [nginx]

TASK [create www directory for your virtualhost]
*********************************************************************************
***********
changed: [nginx]

TASK [delete default nginx site]
*********************************************************************************
***************************
ok: [nginx]

TASK [copy nginx site.conf]
*********************************************************************************
*********************************
changed: [nginx]

RUNNING HANDLER [restart nginx]
*********************************************************************************
*****************************
changed: [nginx]
```

```
PLAY RECAP
*********************************************************************************
*************************************************
nginx                      : ok=5    changed=3    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

- Task 14.4 -

Verify that you can access to your VM DNS entry : http://USER-NAME.devops.intuitivesoft.cloud

No website yet exists access if forbidden, but the resolution is successful.

Task 15. Idempotency

In the context of automation and systems management, idempotency is a crucial concept. When designing scripts, configuration management systems, or automation tasks, ensuring idempotency means that running the same action multiple times will not lead to unexpected or undesirable outcomes. This property helps maintain consistent and predictable system states regardless of how many times the operation is executed.

Ansible modules *must* implement idempotency, let's verify this and re-run the previous playbook

```
ansible-playbook -i inventory.ini create-vhost.yaml
```

You should get the following output:

```
PLAY [Configure a new virtualhost]
*********************************************************************************
*****

TASK [Gathering Facts]
*********************************************************************************
*****************
ok: [nginx]

TASK [create www directory for your virtualhost]
*********************************************************************
ok: [nginx]

TASK [delete default nginx site]
*********************************************************************************
*******
ok: [nginx]

TASK [copy nginx site.conf]
*********************************************************************************
************
ok: [nginx]
```

```
PLAY RECAP
*********************************************************************************
*****************************
nginx                      : ok=4    changed=0    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

Note that no task has `changed` hence the `handler` task `restart nginx` has not been notified. Nothing happened !

LAB : Ansible NGINX SSL

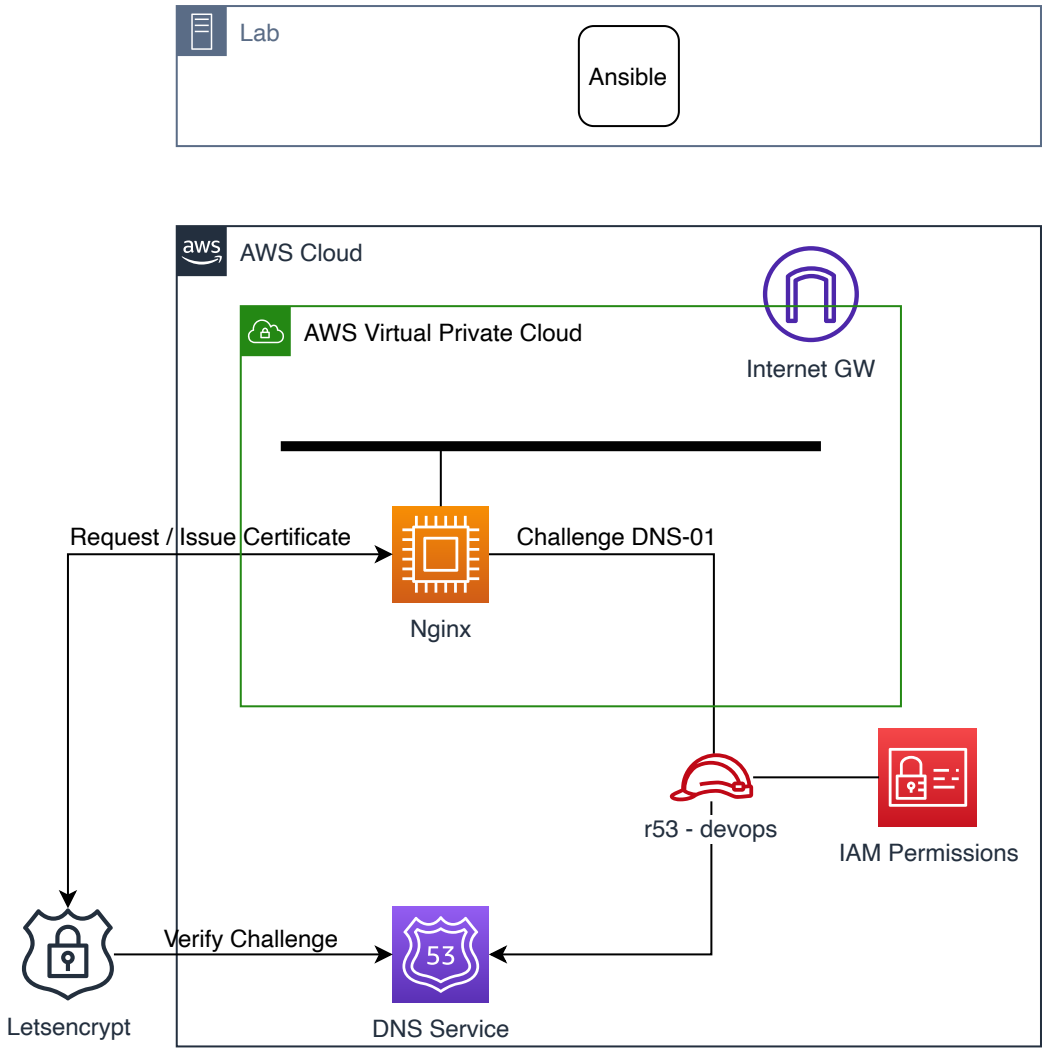
In the following lab you will automate the deployment of a simple website and its SSL certificate using Ansible and letsencrypt.

Certbot is a popular open-source tool developed by the Electronic Frontier Foundation (EFF) that simplifies the process of obtaining and managing SSL/TLS certificates from Let's Encrypt. Let's Encrypt is a free and automated certificate authority that provides SSL/TLS certificates to secure websites.

When setting up an SSL certificate for an Nginx web server, Certbot streamlines the otherwise complex process into a few straightforward steps. It automates the certificate issuance, renewal, and Nginx configuration process, ensuring your website remains secure without manual intervention.

For Letsencrypt to issue a new certificate you need to answer a challenge to verify domain ownership. The challenge method used in this lab will be DNS-01. During the certificate request process Letsencrypt awaits a TOKEN to be enter in a specific TXT entry under the domain you own.

Lab Diagram



Task 1. Create a playbook to deploy your website

Let's deploy a simple website that you will later on protect with HTTPS reverse-proxy.

- Task 1.1 -

Inside `files` folder create a `index.html` page with the following content :

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Hello World Page</title>
  <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.
css">
</head>
<body>
  <div class="container mt-5">
    <div class="row justify-content-center">
      <div class="col-md-6">
        <div class="card">
          <div class="card-body text-center">
```

```
            <h1 class="card-title">Hello, World!</h1>
            <p class="card-text">Welcome to my devops training.</p>
            <div class="alert alert-warning" role="alert" id="HttpsCheck">
              Warning: You are accessing this page over an insecure
connection (HTTP). Please consider using a secure connection (HTTPS) for
better security.
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
  <script>
    // JavaScript to change the site's color based on the protocol
    const isSecure = window.location.protocol === 'https:';
    const body = document.querySelector('body');
    const httpMessage = document.getElementById('HttpsCheck');
    if (isSecure) {
      body.style.backgroundColor = 'green';
      httpMessage.classList.add('d-none');
    } else {
      body.style.backgroundColor = 'red';
    }
  </script>
</body>
</html>
```

- Task 1.2 -

Create a new ansible playbook named `deploy-website.yml`

With the following content :

```
- name: Deploy devops website
  gather_facts: yes
  hosts: webservers

  vars_files:
    - vars/generic.yml

  tasks:
  - name: change index.html
    ansible.builtin.copy:
      src: files/index.html
      dest: /var/www/{{domain}}/index.html
      mode: '0775'
      owner: "{{ ansible_user }}"
      group: "{{ ansible_user }}"
    become: true
```

- Task 1.3 -

Execute your playbook:

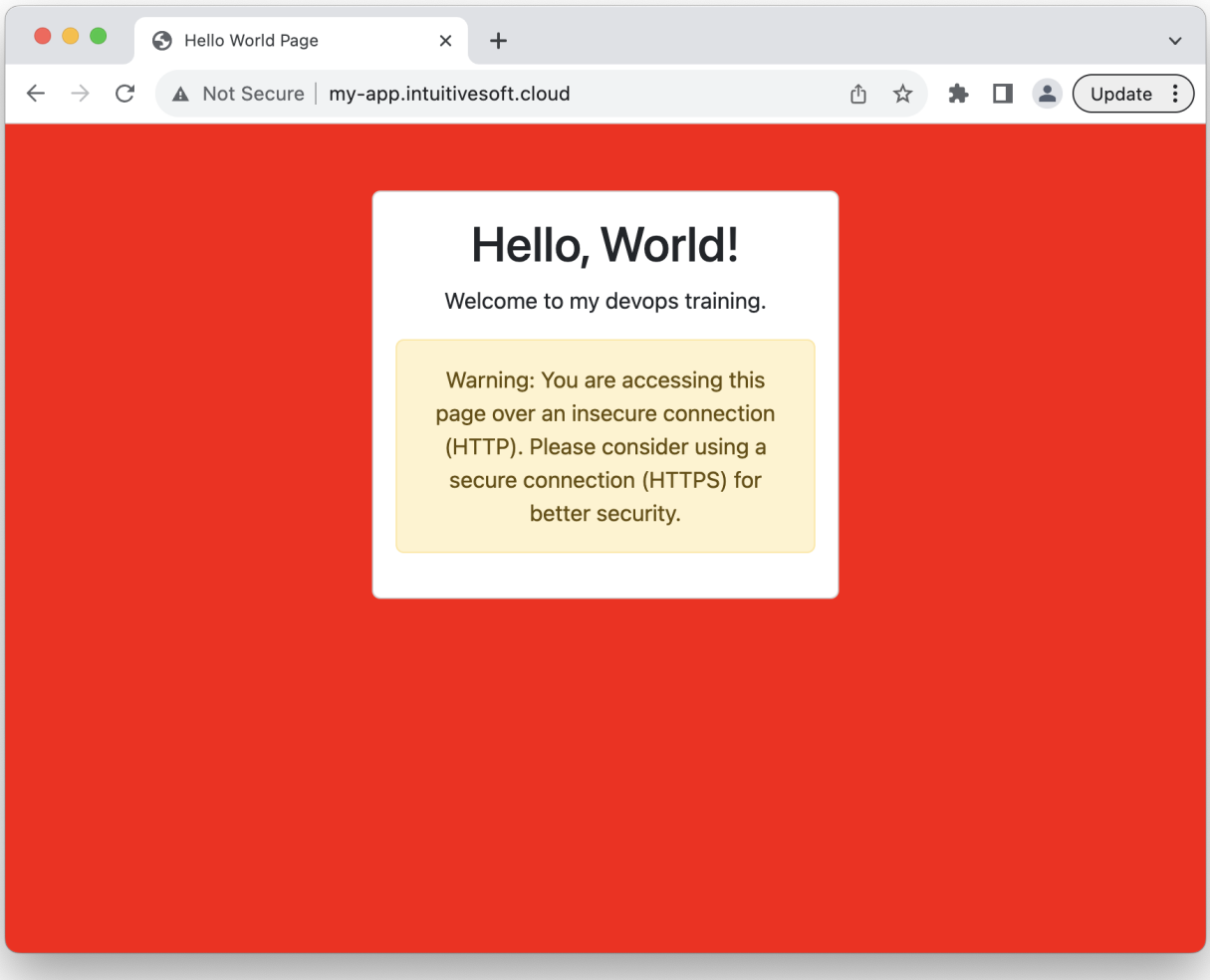
```
ansible-playbook -i inventory.ini deploy-website.yml
```

You should get the following output :

```
PLAY [Deploy devops website]
**********************************************************************************
*******************************

TASK [Gathering Facts]
**********************************************************************************
************************************
ok: [nginx]

TASK [change index.html]
**********************************************************************************
***********************************
changed: [nginx]

PLAY RECAP
**********************************************************************************
************************************************
nginx                      : ok=2    changed=1    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

- Task 1.4 -

Verify that your website is deployed by opening a web browser tab to your instance DNS entry.



Insight 2. EC2 IAM Role

An EC2 IAM role is a powerful tool that enables Amazon EC2 instances to securely access and interact with AWS services without needing to manage long-term credentials. When it comes to updating Route 53 DNS records, you can create an IAM role with the necessary permissions to modify Route 53 hosted zones and resource record sets.

By assigning this IAM role to an EC2 instance, you grant it the authority to make changes to Route 53 records on your behalf. This is particularly useful for scenarios like dynamic IP address updates or automated DNS record management.

With the IAM role attached, the EC2 instance can utilize AWS SDKs or command-line tools like the AWS CLI to make authorized requests to Route 53. The IAM role ensures that the instance follows the principle of least privilege, only having the specific permissions needed for updating DNS records.

This approach enhances both security and automation. You avoid exposing long-term access keys on instances, reducing the risk of unauthorized access. Plus, by centralizing permissions through IAM roles, you simplify the management of permissions across instances and services.

Task 3. Verify existing IAM EC2 role

- Task 3.1 -

An AWS EC2 IAM role named `r53-devops` already exist in your environnement let's visualize its content.

Using AWS CLI list the policies attached to the role :

```
aws iam list-attached-role-policies --role-name r53-devops
```

Expected output :

```
{
    "AttachedPolicies": [
        {
            "PolicyName": "Letsencrypt-devops.intuitivesoft.cloud",
            "PolicyArn": "arn:aws:iam::708113109960:policy/Letsencrypt-
devops.intuitivesoft.cloud"
        }
    ]
}
```

- Task 3.2 -

Visualize the content of the `Letsencrypt-devops.intuitivesoft.cloud` policy.

```
aws iam get-policy-version --policy-arn
arn:aws:iam::708113109960:policy/Letsencrypt-devops.intuitivesoft.cloud --
version-id v1
```

Expected output :

```
{
    "PolicyVersion": {
        "Document": {
            "Version": "2012-10-17",
            "Id": "certbot-dns-route53 sample policy",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Action": [
                        "route53:ListHostedZones",
                        "route53:GetChange"
                    ],
                    "Resource": [
                        "*"
                    ]
                },
                {
                    "Effect": "Allow",
```

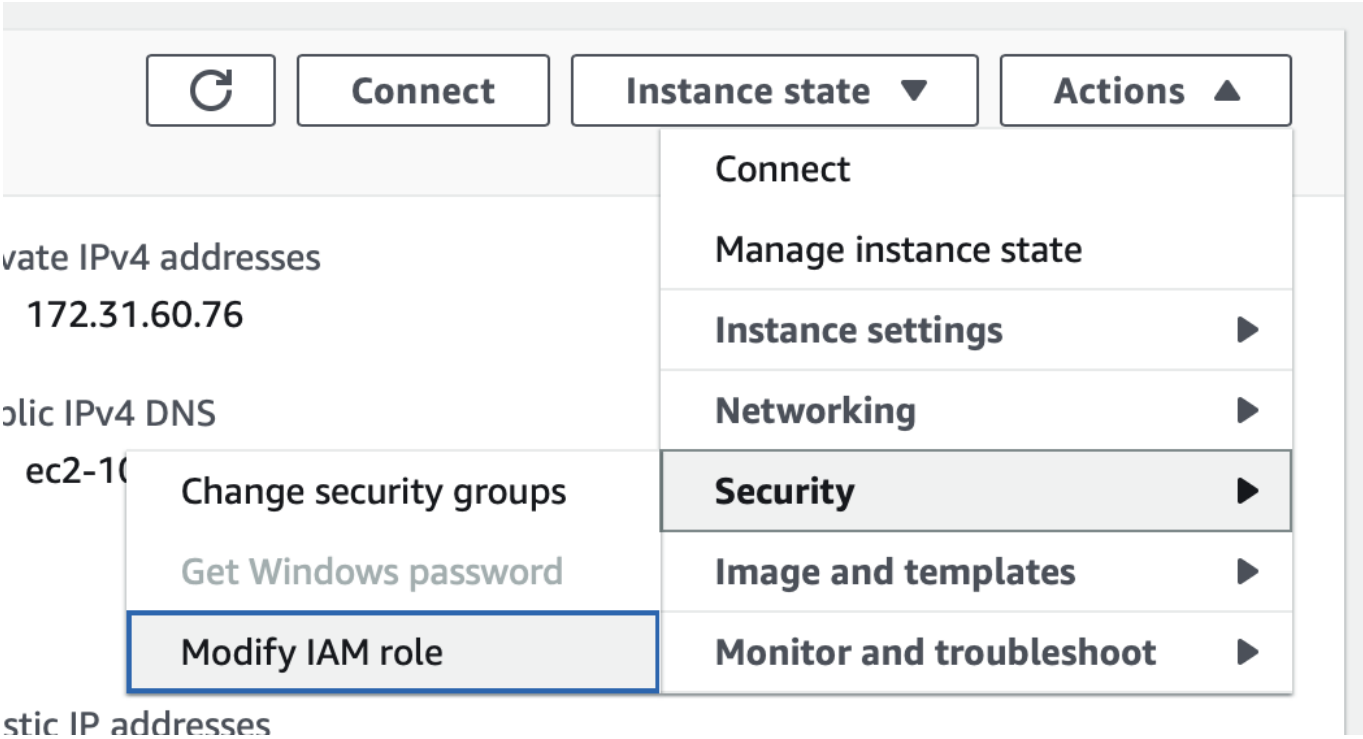
```
                    "Action": [
                        "route53:ChangeResourceRecordSets"
                    ],
                    "Resource": [

"arn:aws:route53:::hostedzone/Z03385262WMCAFO2KASN1"
                    ]
                }
            ]
        },
        "VersionId": "v1",
        "IsDefaultVersion": true,
        "CreateDate": "2023-05-12T14:20:15+00:00"
    }
}
```

- The policy allows `List/Get` actions on every hosted zone on AWS.
- It allows to modify DNS `records` for the specific zone ID `Z03385262WMCAFO2KASN1`

Task 4. Attach EC2 role to your instance

Web UI

Locate your EC2 instance click `Actions > Security > Modify IAM role`



From the dropdown menu select the `r53-devops` role an click `Update IAM role`

CLI

Update the value of INSTANCE_ID with the ID of your AWS EC2 instance

```
aws ec2 associate-iam-instance-profile --iam-instance-profile Name=r53-
devops --instance-id INSTANCE_ID
```

Task 5. Allow HTTPS connections

Make appropriate changes to your VM to allow external connections on port 443.

Task 6. Deploy certificate for your host

- Task 6.1 -

Create a new ansible playbook named `deploy-certificate.yml` with the following content :

```
---
- name: Activate SSL certificate
  gather_facts: yes
  hosts: webservers
  become: true

  vars_files:
    - vars/generic.yaml

  tasks:
  - name: install certbot
    community.general.snap:
      name:
        - certbot
      classic: true

  - name: set certbot privilege
    ansible.builtin.command: snap set certbot trust-plugin-with-root=ok

  - name: install certbot addons
    community.general.snap:
      name:
        - certbot-dns-route53
      classic: true

  - name: issue certificate
    ansible.builtin.command: certbot --non-interactive --redirect --agree-
tos --nginx -d {{domain}} -m devops@intuitivesoft.cloud
    notify: restart nginx

  handlers:
    - name: restart nginx
      service:
        name: nginx
        state: restarted
```

- Task 6.2 -

This playbook use a community library, you can download custom roles and collection using ansible-galaxy

```
ansible-galaxy collection install community.general
```

- Task 6.3 -

Finally deploy your certificate

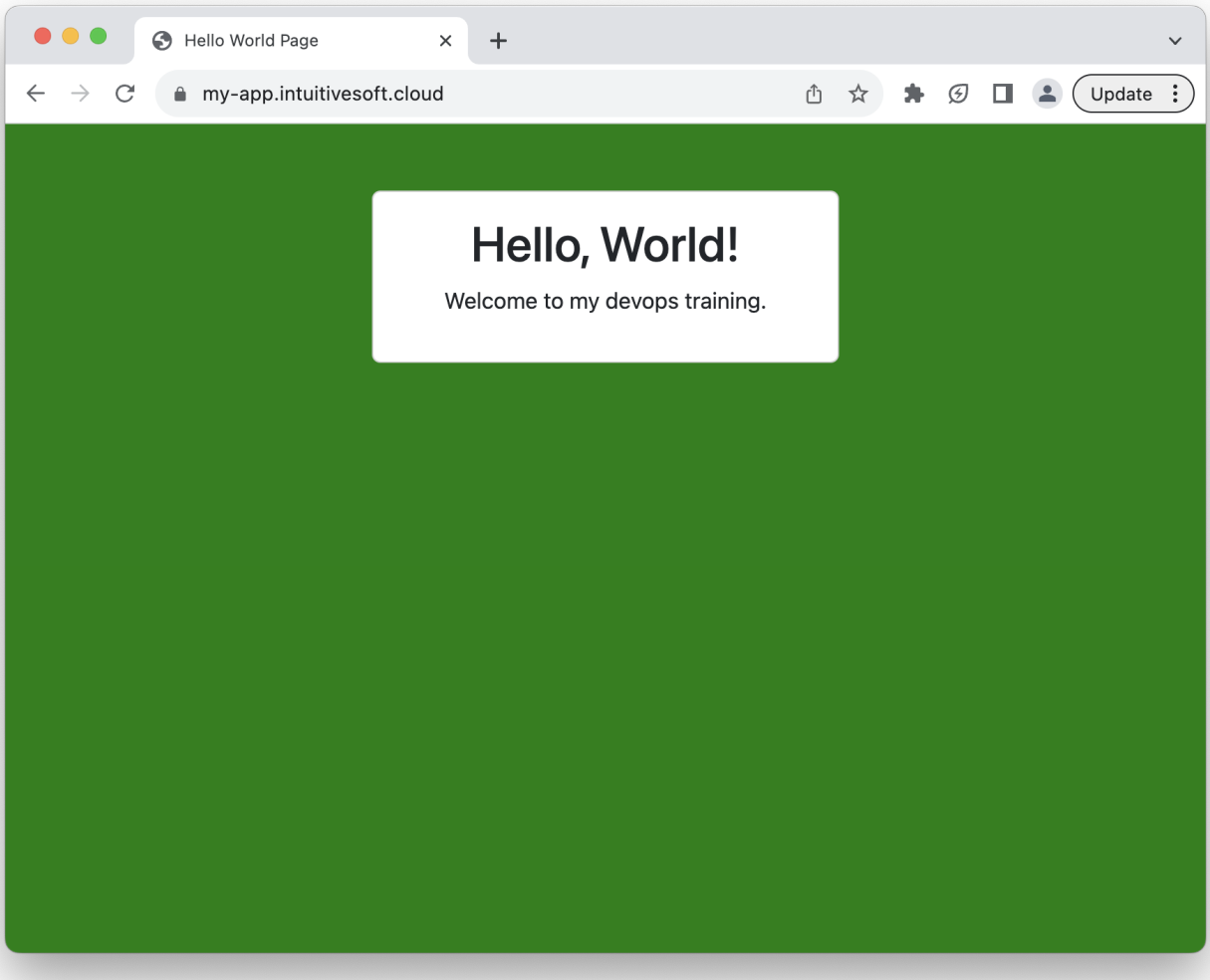
```
ansible-playbook -i inventory.ini deploy-certificate.yml
```

Expected output :

```
PLAY [Activate SSL certificate]
*********************************************************************

TASK [Gathering Facts]
*********************************************************************************
****
ok: [nginx]

TASK [install certbot and addons]
*********************************************************************
ok: [nginx]

TASK [issue certificate]
*********************************************************************************
**
changed: [nginx]

RUNNING HANDLER [restart nginx]
*********************************************************************
changed: [nginx]

PLAY RECAP
*********************************************************************************
****************
nginx                      : ok=4    changed=2    unreachable=0
failed=0    skipped=0    rescued=0    ignored=0
```

- Task 6.2 -

Verify that your website is now protected by a certificate and your are automatically redirected to HTTPS.



Task 7. Visualize the NGINX configuration file

Using Ansible ad-hoc commend retrieve the content of your NGINX configuration file. It has been automatically updated by certbot with the SSL configuration.

```
ansible nginx -i inventory.ini --module-name command -a "cat
/etc/nginx/sites-enabled/REDACTED.intuitivesoft.cloud"
```

Expected output:

```
nginx | CHANGED | rc=0 >>
server {
  server_name REDACTED.intuitivesoft.cloud;
  root /var/www/REDACTED.intuitivesoft.cloud;
  location / {
    try_files $uri $uri/ =404;
  }

    listen [::]:443 ssl ipv6only=on; # managed by Certbot
    listen 443 ssl; # managed by Certbot
```

```
    ssl_certificate /etc/letsencrypt/live/my-
REDACTED.intuitivesoft.cloud/fullchain.pem; # managed by Certbot
    ssl_certificate_key
/etc/letsencrypt/live/REDACTED.intuitivesoft.cloud/privkey.pem; # managed
by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot

}server {
    if ($host = REDACTED.intuitivesoft.cloud) {
        return 301 https://$host$request_uri;
    } # managed by Certbot


  listen 80 default_server;
  listen [::]:80 default_server;
  server_name REDACTED.intuitivesoft.cloud;
    return 404; # managed by Certbot
}
```

AWS Video Streaming App Automation project

Objectives

The project aims to develop an automation environment to deploy a Video Streaming service deployed on AWS public cloud.

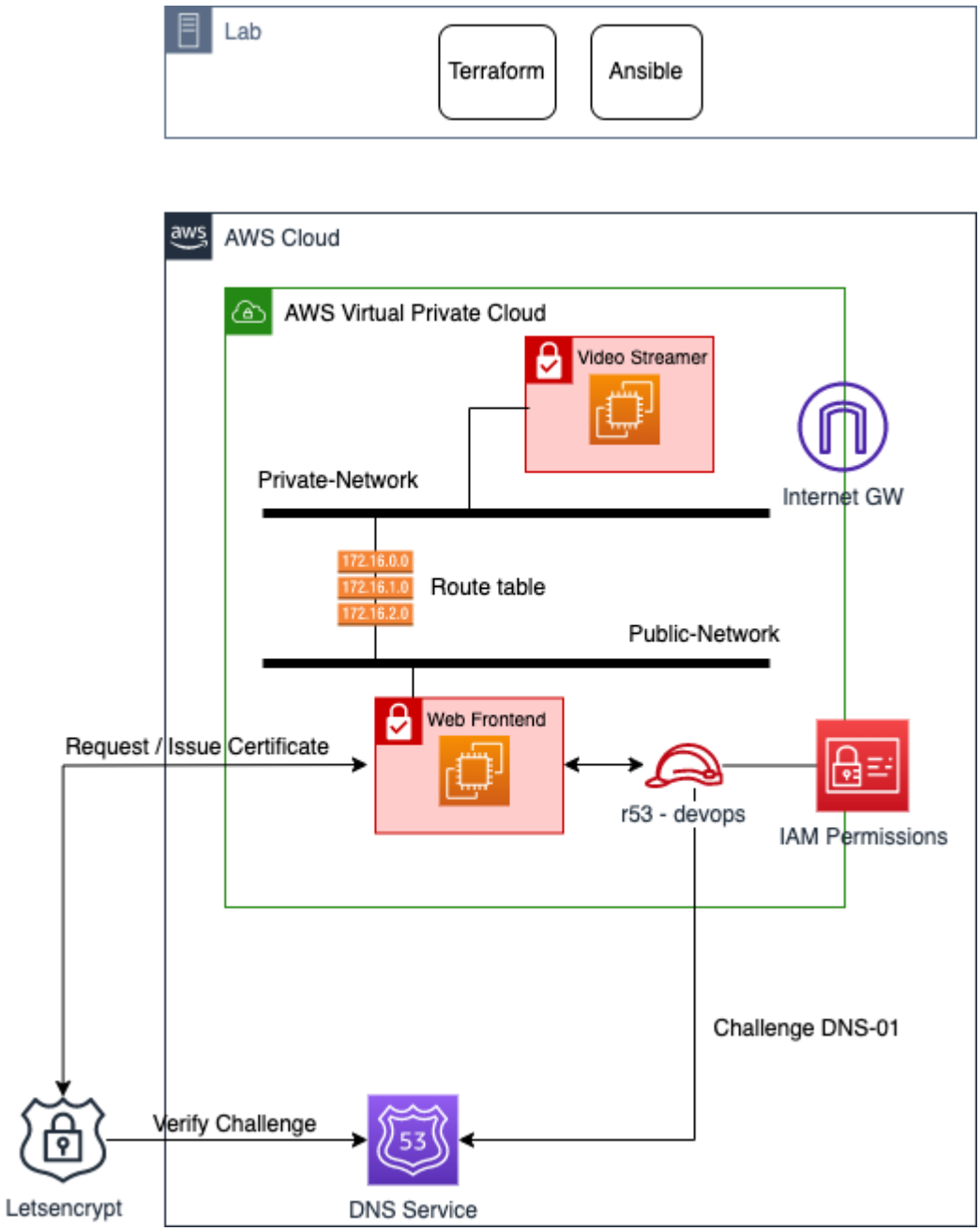
You will reuse the knowledge from the previous sessions to deploy the necessary network infrastructure (VPC, Internet Gateway, Subnets, Routing Table) and 2x VMs (EC2 instance) to run the video and web applications following a 2-tier design pattern.

One VM (Video Streamer) runs a Video Streamer application to stream video towards the other VM (Web Frontend). The Web Frontend VM ingests stream from the Video Streamer app and serves the video to the end users via HTTPS.

End users can only access the Web Frontend using HTTPS on port 443, and have no access to the Video Streamer. Web Frontend has a valid SSL certificated. Communication between Video Streamer and Web Frontend is restricted to the only necessary ports required for stream ingestion.

Application architecture

The diagram below highlights the target architecture which needs to be deployed and configured (through an automated process):



Automation steps

The different steps we foresse for the automation process are highlighted below:

1. TF AWS: the procedure includes the deployment of 2xEC2 instances (Video Streamer, Web Frontend) as well as VPC, Internet Gateway, Subnets, Routing Table or Routes. *It has been already done in a previous session during the Terraform lab.*
2. Ansible: the procedure includes the installation/configuration of the NGINX application on the Web Frontend EC2 instance and FFmpeg on the Video Streamer instance. *It has been almost done in a previous session during the Ansible lab.*

Project technical guidelines

On video Streamer

Linux Package dependencies

- ffmpeg

Creative Commons Attribution 3.0 sample videos : https://download.blender.org/demo/movies/BBB/

This command is used to stream a video file (BigBuckBunny_320x180.mp4) in a loop to an RTMP server

```
ffmpeg \
-stream_loop -1 \                      # Loops the input video infinitely
-re \                                   # Read input at native frame rate
(for real-time streaming)
-i /videos/BigBuckBunny_320x180.mp4 \   # Input video file (Big Buck Bunny
in this case)
-c:v libx264 \                          # Video codec: encode the video
using the H.264 codec
-c:a aac \                              # Audio codec: encode the audio
using the AAC codec
-f flv \                                # Output format: FLV (Flash
Video), common for RTMP streaming
rtmp://<rtmp-server-ip>:1935/live/stream      # RTMP server URL:
destination for the stream, targeting a live RTMP stream on the nginx-rtmp
server
```

On the Web Frontend

Linux Package dependencies

- nginx
- libnginx-mod-rtmp
- ffmpeg

Create Root folder `stream`, for DASH and HLS chunks.

```
mkdir /var/www/html/stream
```

Content of `/etc/nginx/nginx.conf` provided as reference :

```
worker_processes  1;
load_module "modules/ngx_rtmp_module.so";

events {
    worker_connections  1024;
}
```

```
http {
    sendfile        on;
    tcp_nopush      on;
    tcp_nodelay     on;
    keepalive_timeout  65;
    types_hash_max_size 2048;

    include         /etc/nginx/mime.types;
    include         /etc/nginx/sites-enabled/*;

    default_type    application/octet-stream;

    server {
        listen 8080;

        # rtmp control
        location /control {
            rtmp_control all;
        }
    }
}

rtmp {
    server {
        listen 1935;
        chunk_size 4096;

        application live {
            live on;
            record off;

            allow publish 127.0.0.0/16;
            allow publish 192.168.0.0/16;
            deny publish all;


            hls on;
            hls_path /var/www/html/stream/hls;
            hls_fragment 3;
            hls_playlist_length 60;

            dash on;
            dash_path /var/www/html/stream/dash;
            dash_fragment 15s;

        }
    }
}
```

Content of `/etc/nginx/sites-enabled/streamer` provided as reference :

```
server {
    listen 80;

    location /hls {
        add_header Access-Control-Allow-Origin *;
        root /var/www/html/stream;
    }

    autoindex_localtime on;

    location /dash {
        # Disable cache
        add_header 'Cache-Control' 'no-cache';

        # CORS setup
        add_header 'Access-Control-Allow-Origin' '*' always;
        add_header 'Access-Control-Expose-Headers' 'Content-Length';

        # Allow CORS preflight requests
        if ($request_method = 'OPTIONS') {
            add_header 'Access-Control-Allow-Origin' '*';
            add_header 'Access-Control-Max-Age' 1728000;
            add_header 'Content-Type' 'text/plain charset=UTF-8';
            add_header 'Content-Length' 0;
            return 204;
        }

        types {
            application/dash+xml mpd;
            video/mp4 mp4;
        }

        root /var/www/html/stream;
    }


}
```

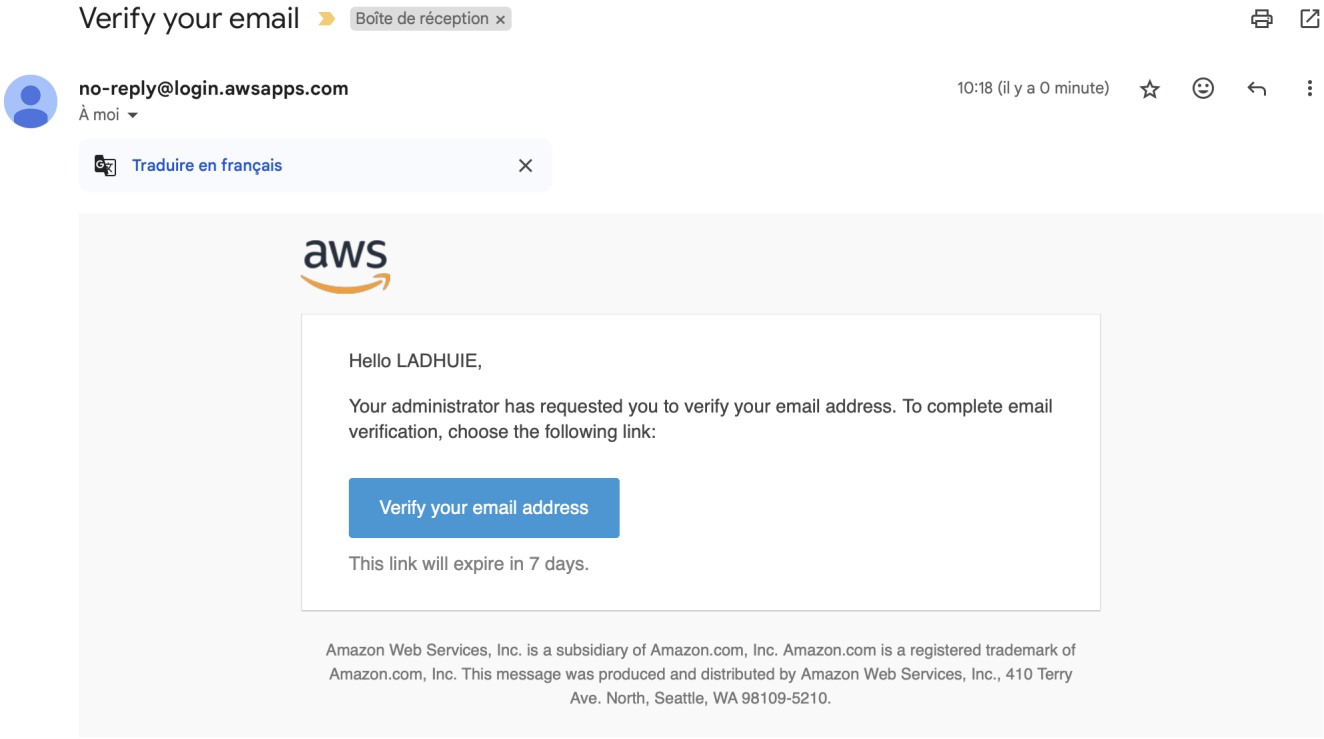
Connect to AWS

First connection

Task 1. Email Verification

For your first connection you should have received an email verification from AWS.

If you did not received the verification email reach out to an instructor.

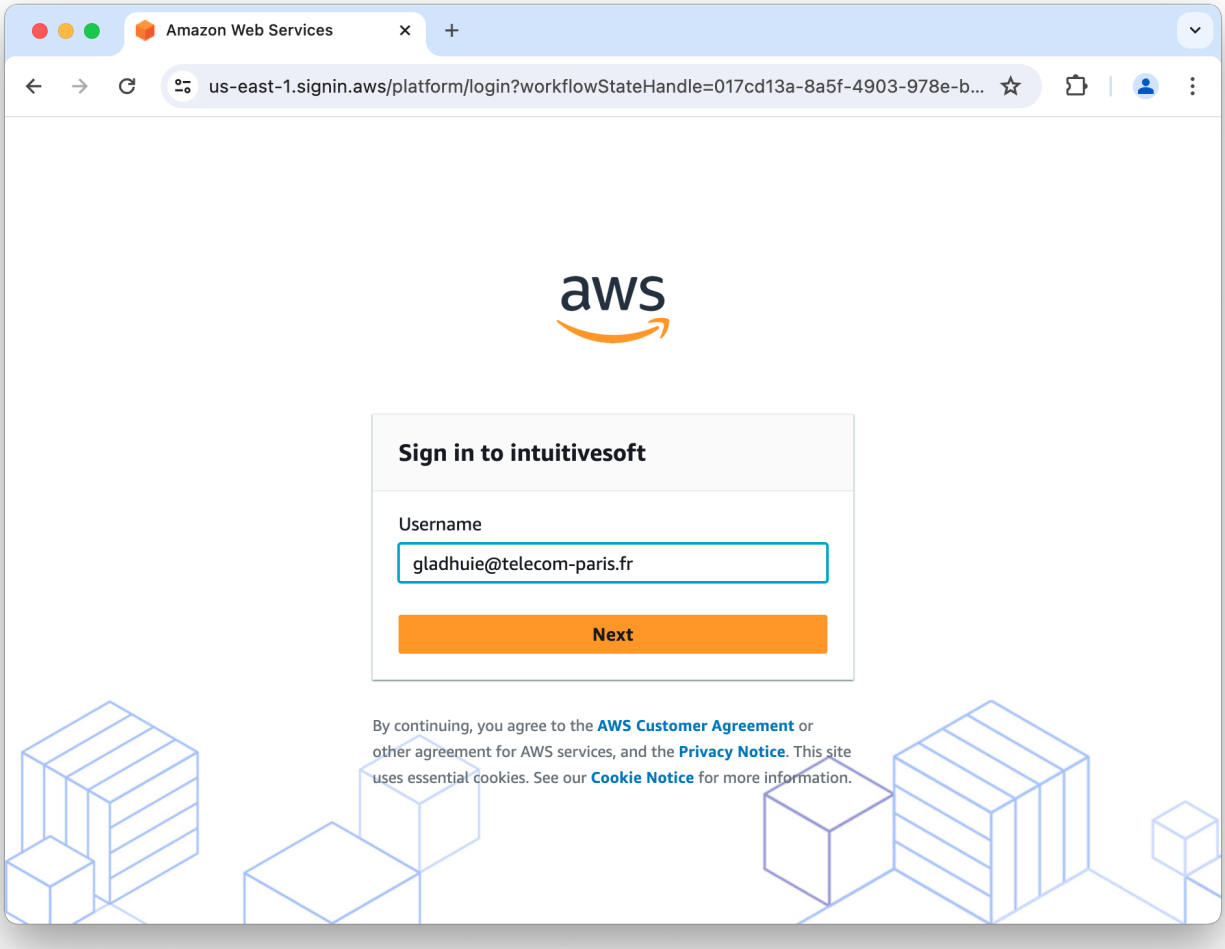


Task 2. First Login

Proceed to account login page or open the following URL : **https://intuitivesoft.awsapps.com/start#/**

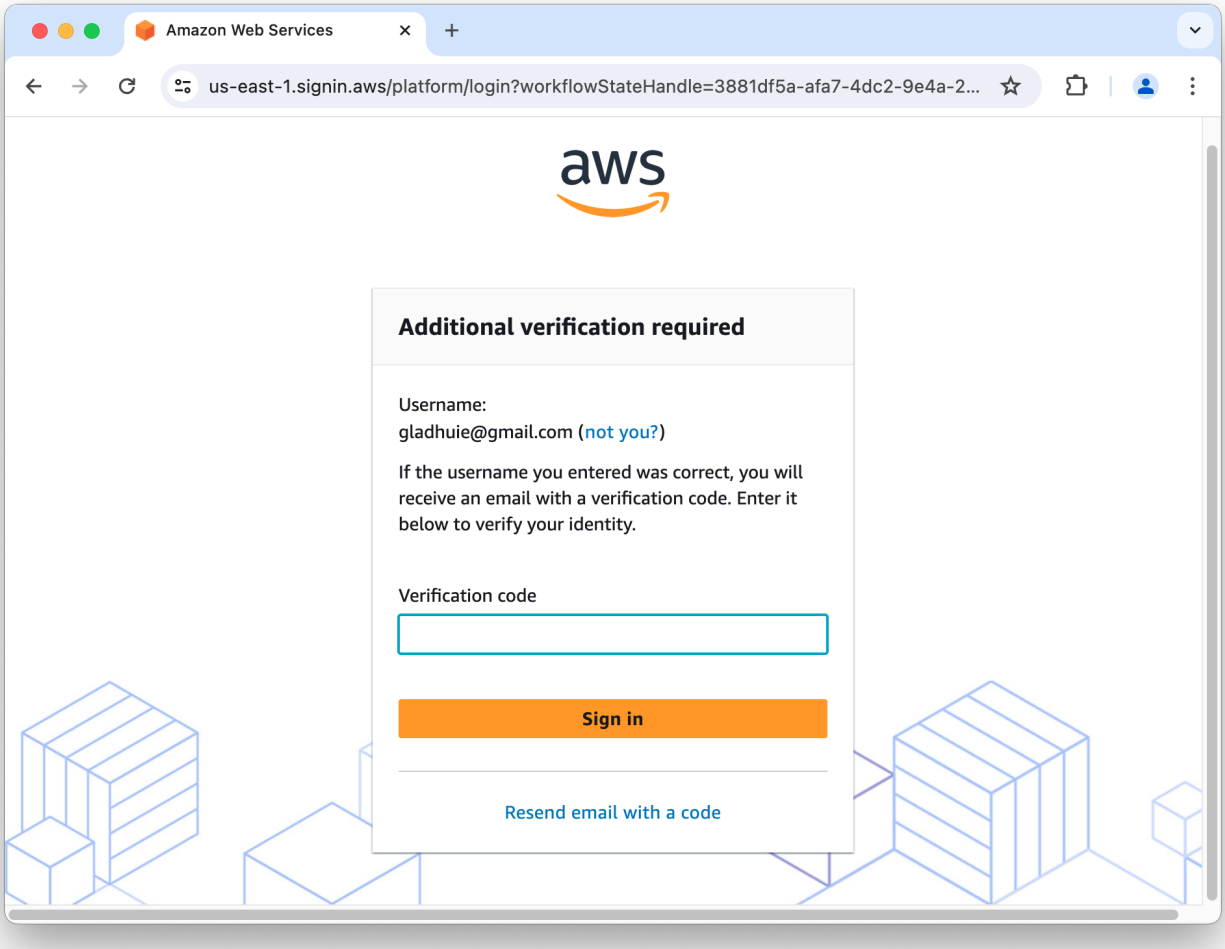
Enter your login username by default it's your email address.

If you are unsure about your login username please reach out to an instructor



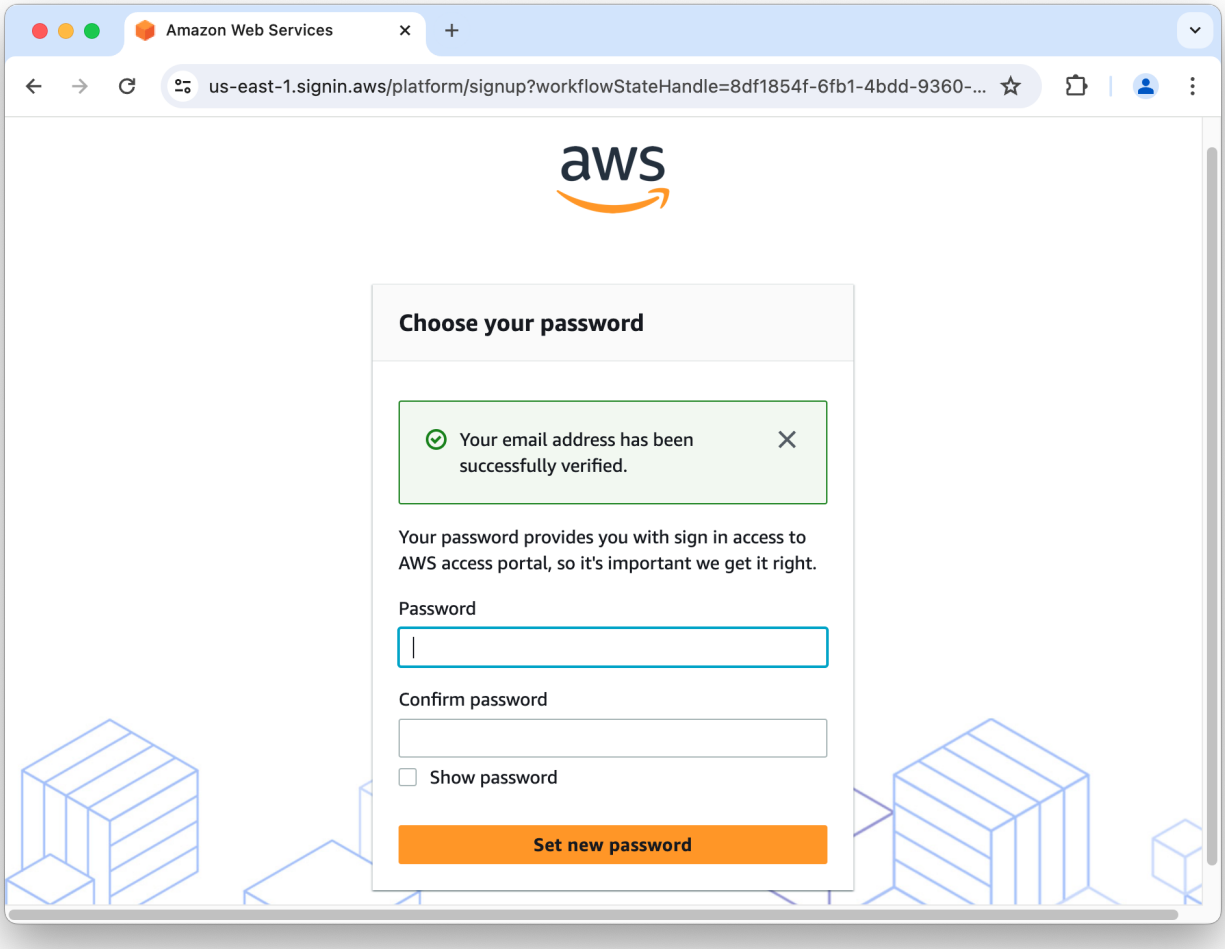
Task 3. Verification code

Once your account is verified, a Verification code will be sent to your contact email.



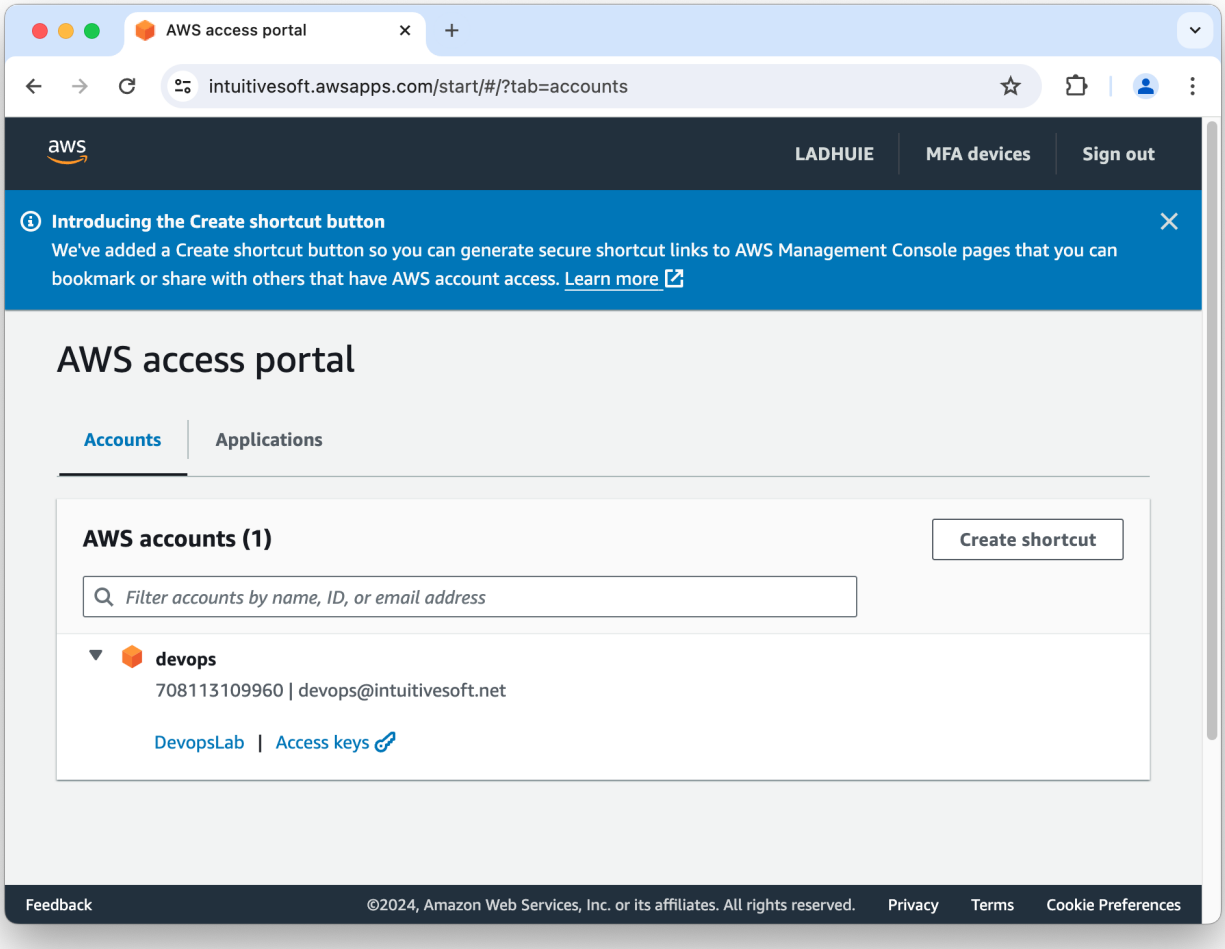
Task 4. New Password

Set a new compliant password for your account.



Task 5. AWS management Console Login

You can now connect to the AWS management console.

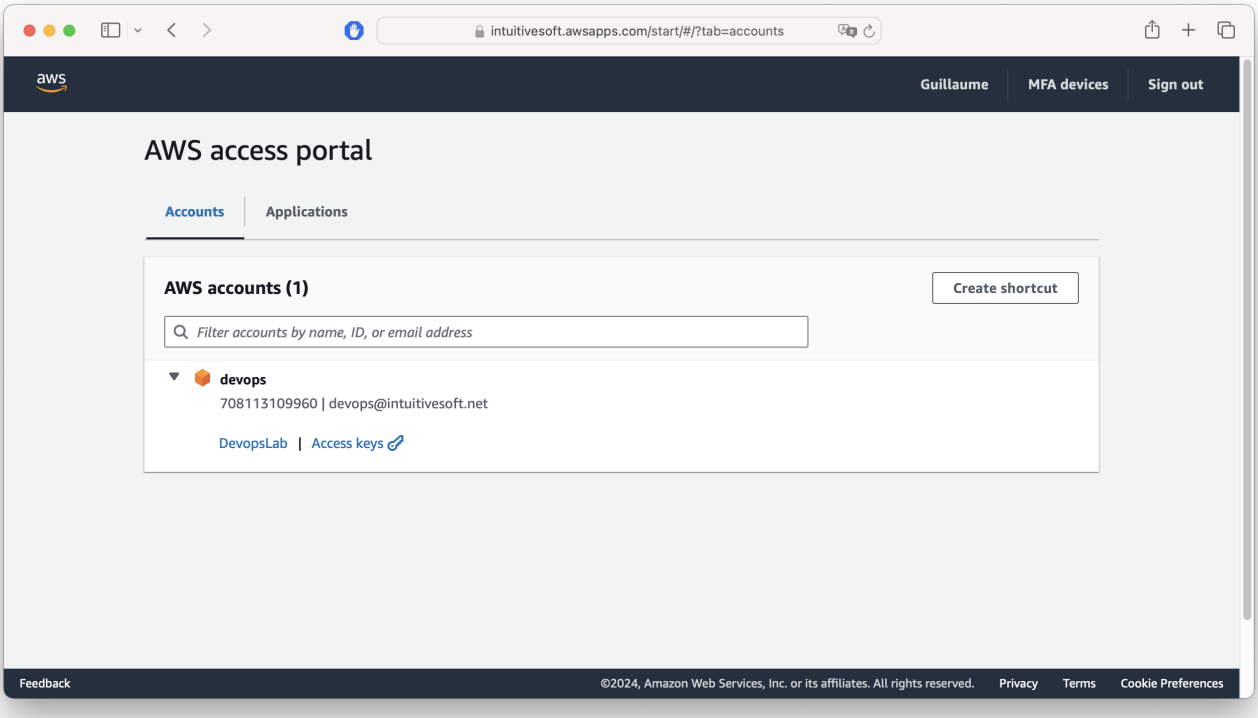


Connect to AWS Management console

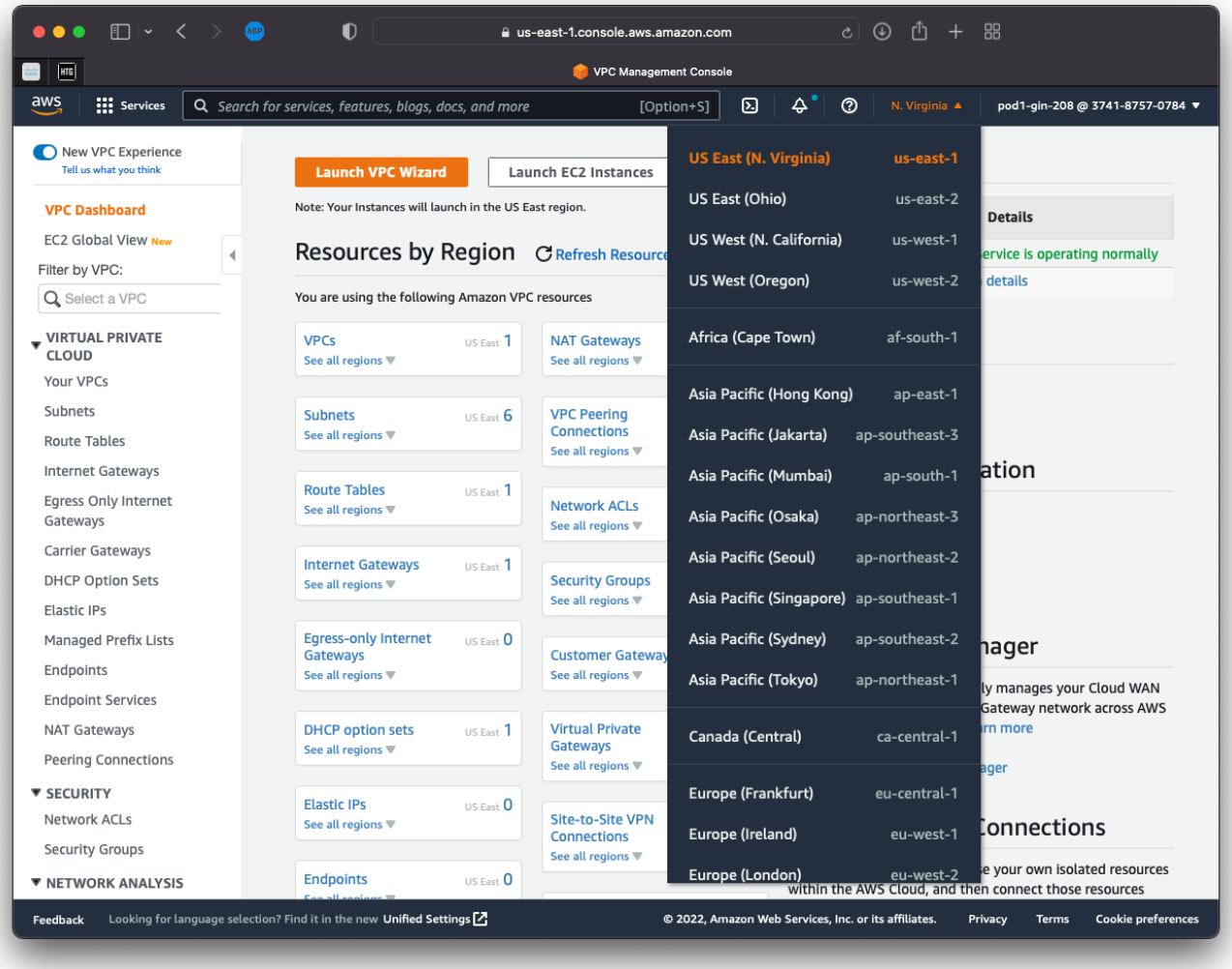
Connect to the AWS management console using the training SSO portal :

https://intuitivesoft.awsapps.com/start#/

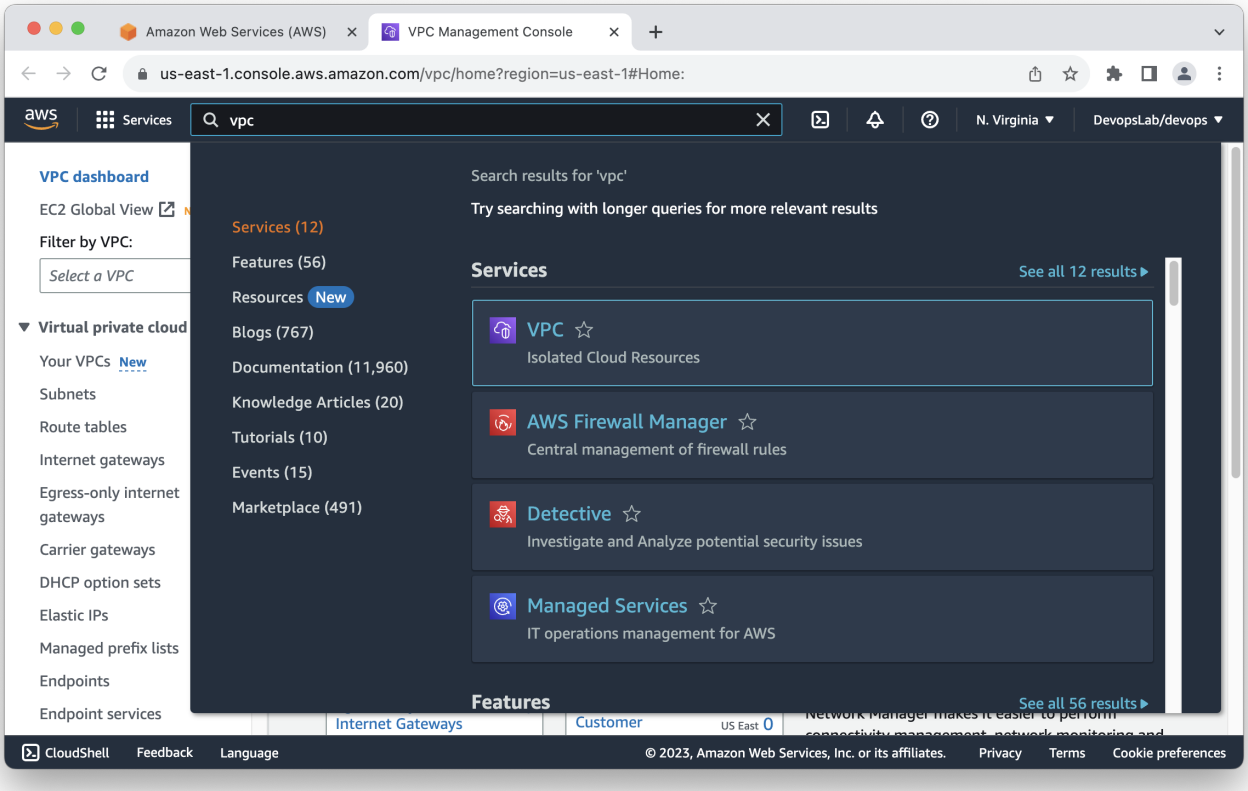
Select your training AWS account and click the link `DevopsLab` to get access to the account main page.



AWS Cloud infrastructure is segmented in administrative regions Once connected verify and switch to the lab region : *us-east-1*



The most convenient way to navigate in between AWS services is to use the research bar on the top side and type the name of the service, or feature inside a service, you'd like to access.



Annex : FAQ

Code-editor

```
sudo apt update
sudo apt install -y unzip jq less groff mandoc curl
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
"awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install --update
```

AWS

AWS CLI re-login

After long period of inactivity you will need to re-issue authentication toward AWS with the following command :

```
aws sso login
```

AWS CLI configuration

If you made a typo or want to edit a configuration of AWS CLI you can manage it by modifying the AWS configuration file located here :

```
~/.aws/config
```

File should contains

```
[default]
sso_session = devops
sso_account_id = 708113109960
sso_role_name = DevopsLab
region = us-east-1
output = json
[sso-session devops]
sso_start_url = https://intuitivesoft.awsapps.com/start#
sso_region = us-east-1
sso_registration_scopes = sso:account:access
```

Annex : Intuitive Lab Dashboard

Terminal

By default the lab dashboard layout offers a live shell terminal to interact with your live lab instance.

From the training instructions you will have the possibility to inject command lines directly to the shell prompt using the `Run in Terminal` box.

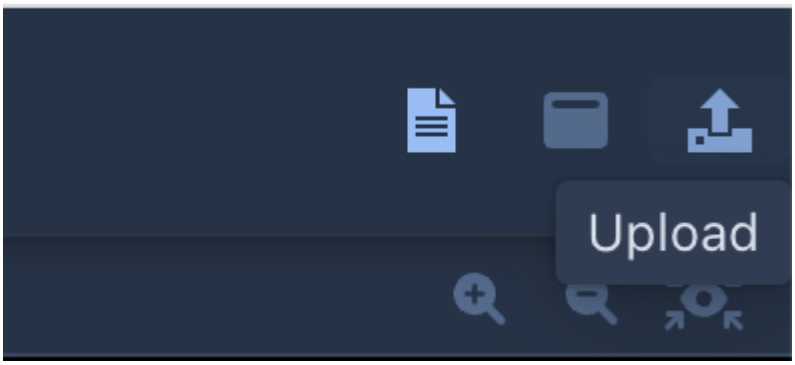
Note : This option is only available in `Terminal View`

Alternatively you can copy the content of box using the copy feature on the box top right corner.

```
aws sts get-caller-identity
```

Upload files to Lab Dashboard

You can upload files to your lab instance using the device browser `Upload` :



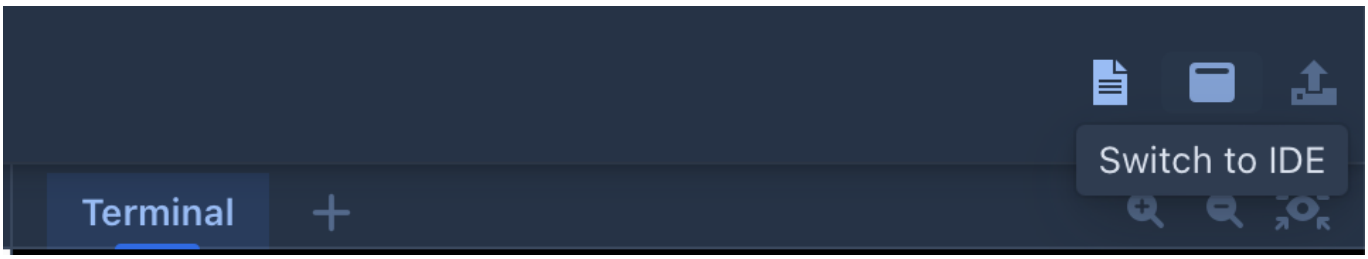
Files are stored in `~/files` folder.

Alternatively you can copy/paste or drag and drop files using Visual Studio Code `File Explorer`.

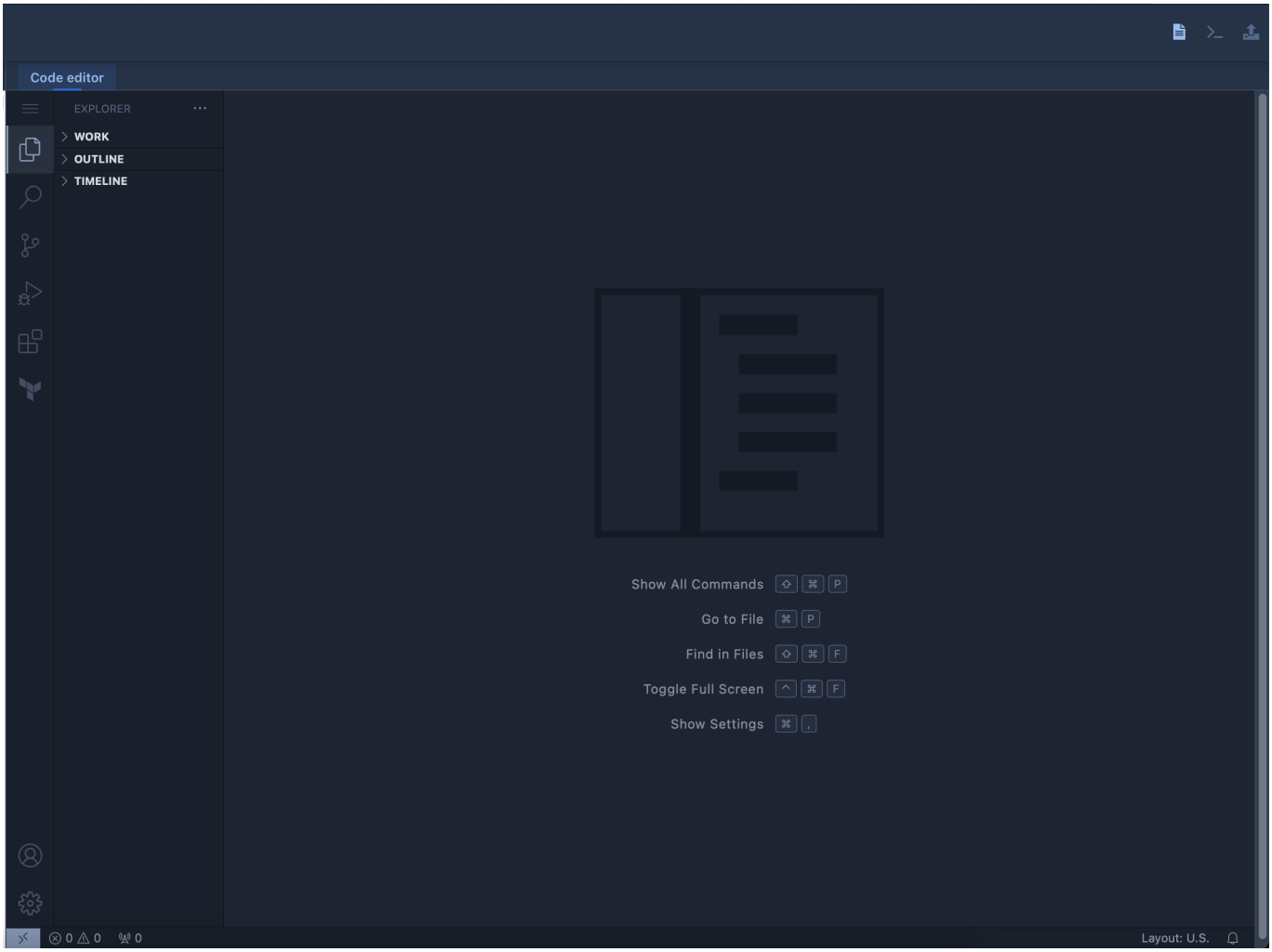
Visual Studio Code

From your lab environment a Visual Studio Code server is available to interact with your working VM.

From your lab dashboard you can switch from `Terminal` to `IDE` view.



It opens the live code editor.



1. File explorer
2. Extensions Manager

- File Explorer -

In the file explorer you can interact with the files and folder in workspace.

Your lab workspace is in `/home/ubuntu/work` folder.

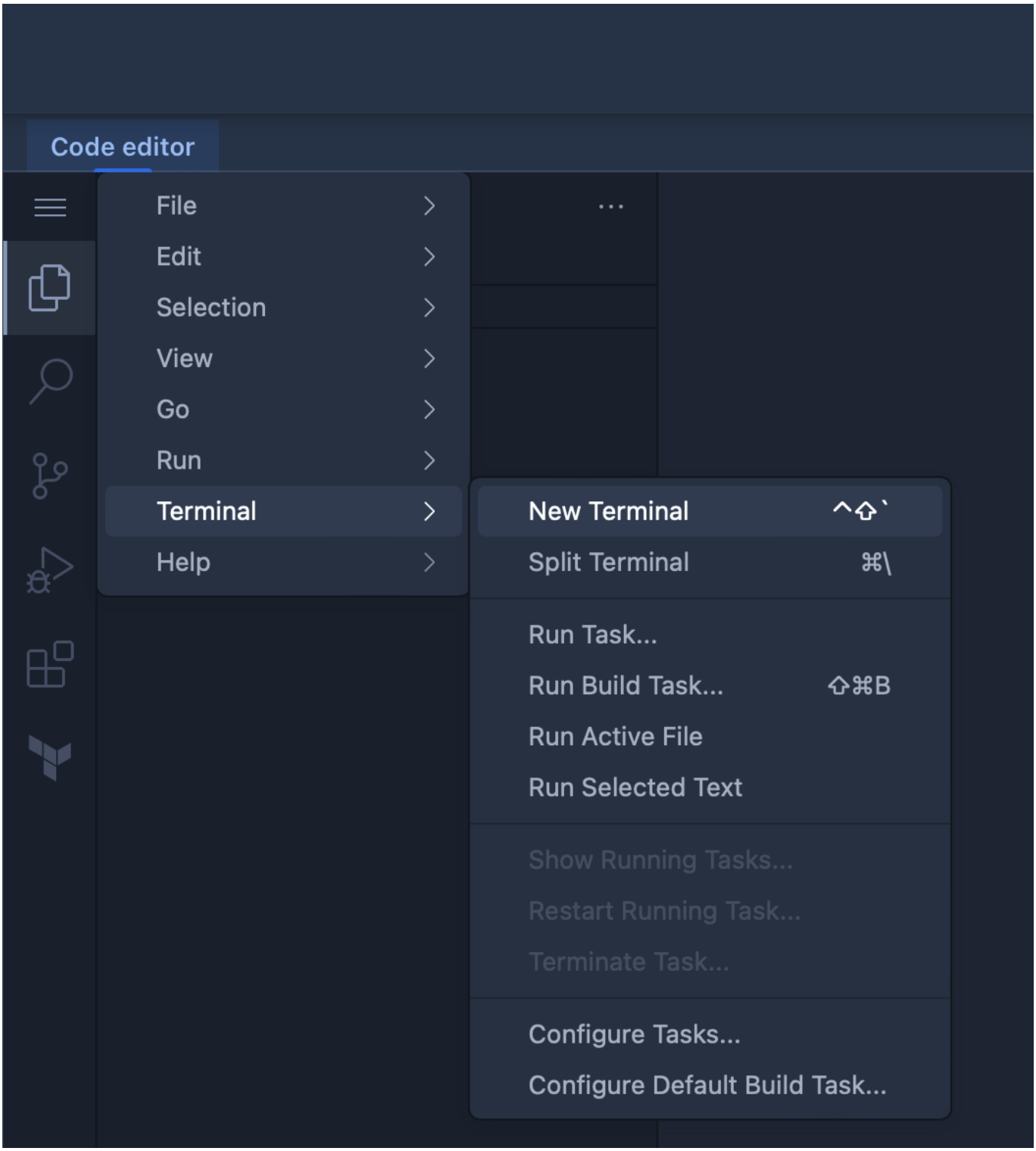
- Extensions Manager -

From the extensions manager menu you can install plugins to add functionality to VSCode based on the programming language or the solution your are working on.

The full list of compatible plugins is available here : https://open-vsx.org

- Terminal -

To open a new shell terminal to your working environment click `menu > Terminal > New Terminal`



Upload files to IDE

Just drag and drop files and folder