

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.

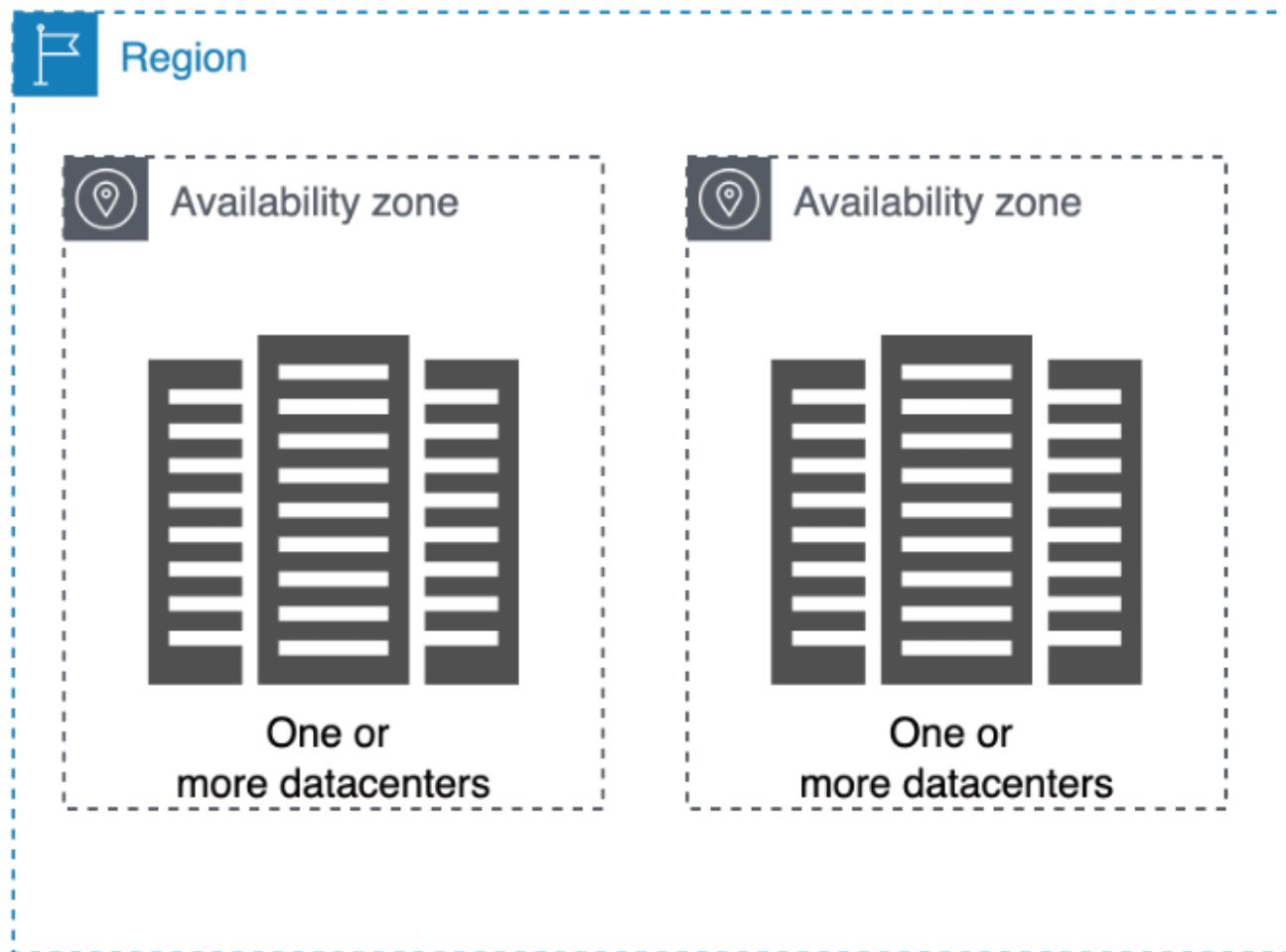
Insight 1. AWS Cloud Architecture

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

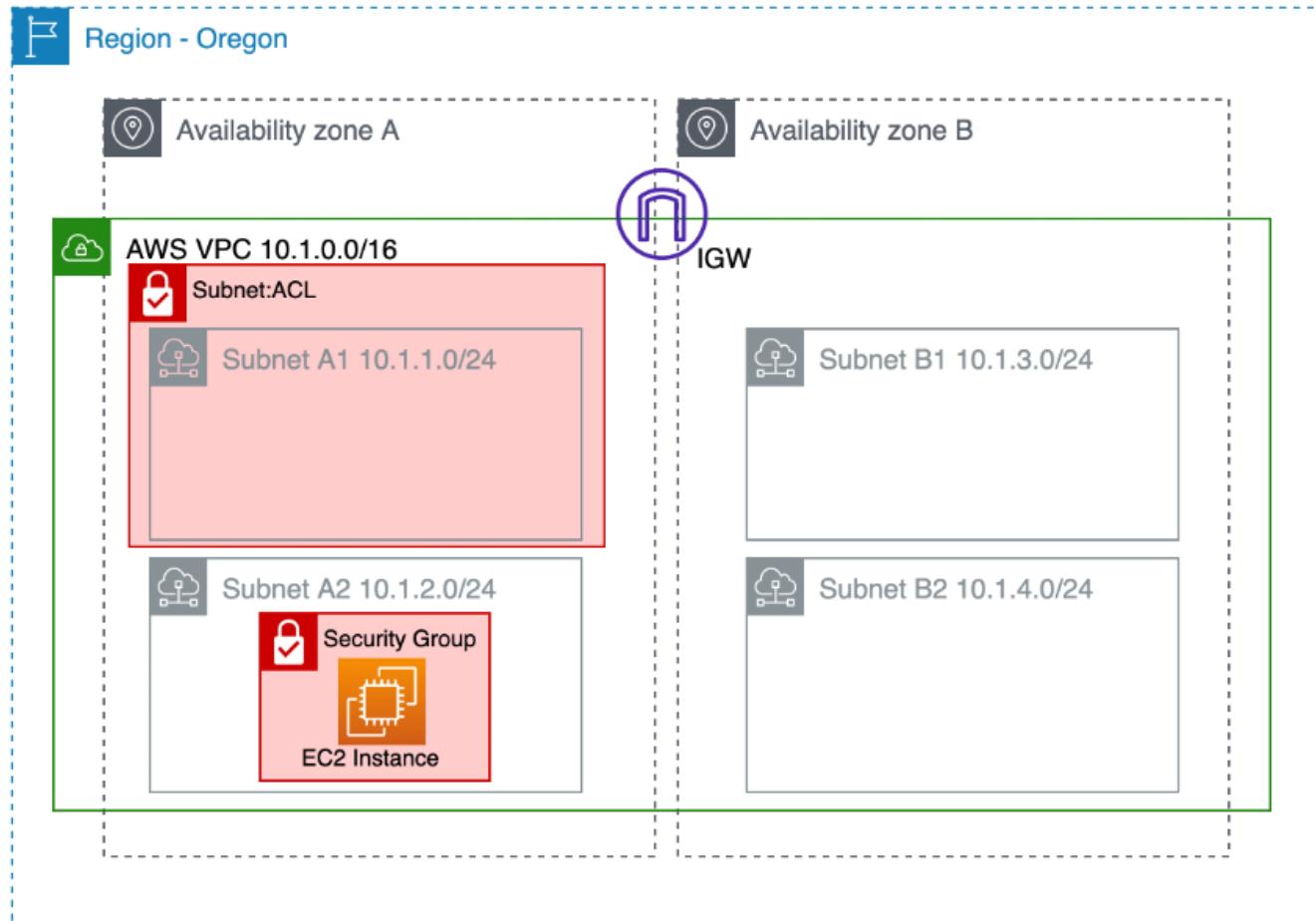
In summary, while AWS ensures the security and reliability of its infrastructure, customers are responsible for securing the applications, data, and workloads they run on AWS services. The Shared Responsibility Model emphasizes collaboration between AWS and its customers to create a secure and compliant 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

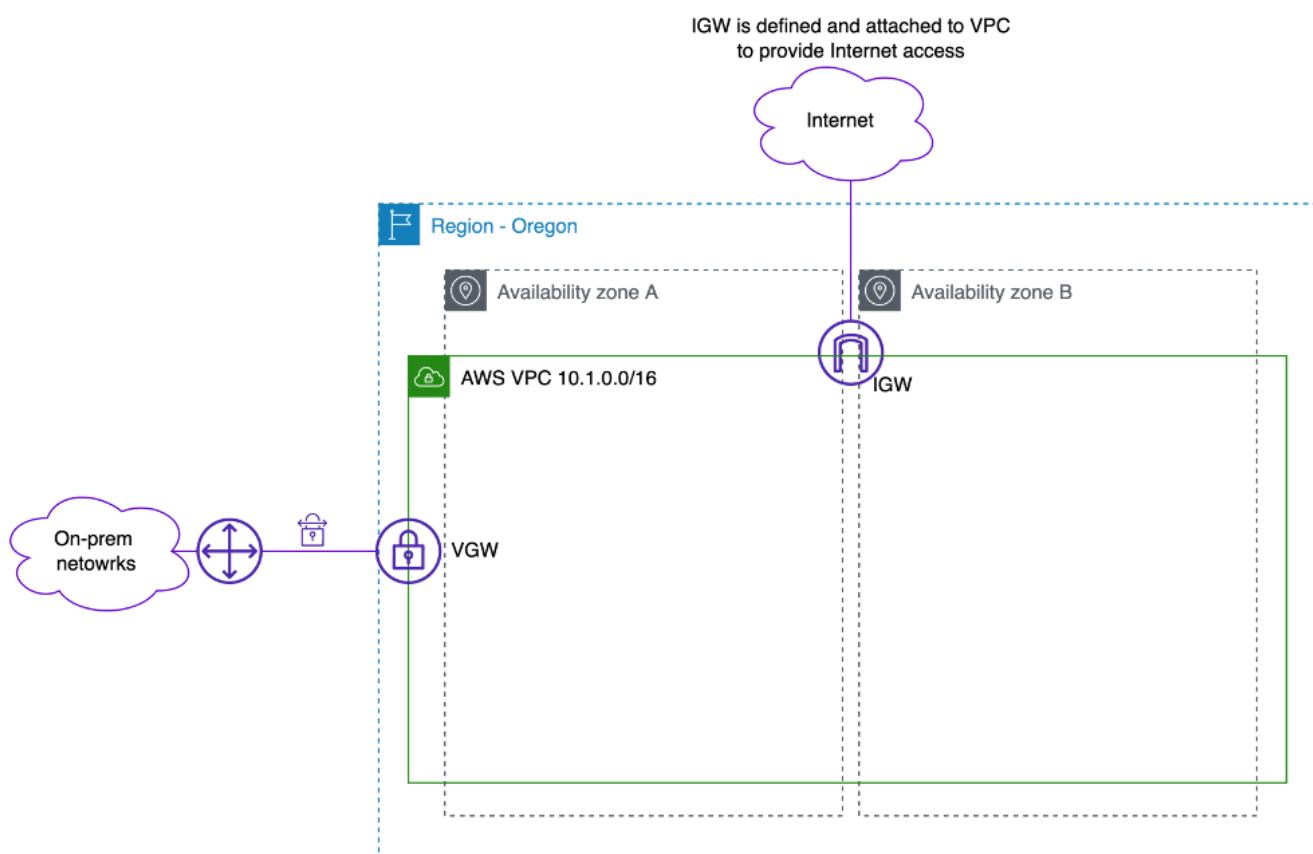
Amazon Virtual Private Cloud (Amazon VPC) is a networking service provided by Amazon Web Services (AWS) that allows you to create and manage a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network environment. This virtual network closely resembles a traditional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

VPC, Subnet and Security Group:

Security Groups and Network ACL



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

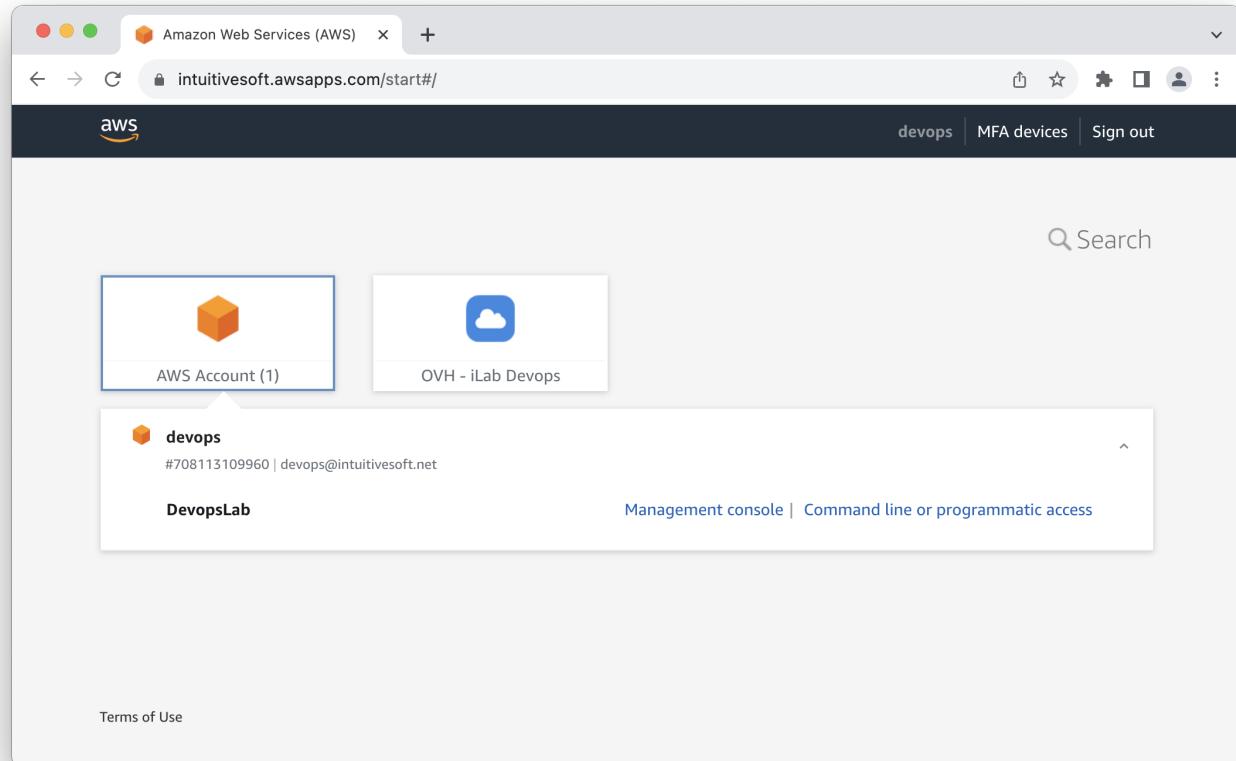
Refer to documentation AWS first connect

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 **Management console** 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*

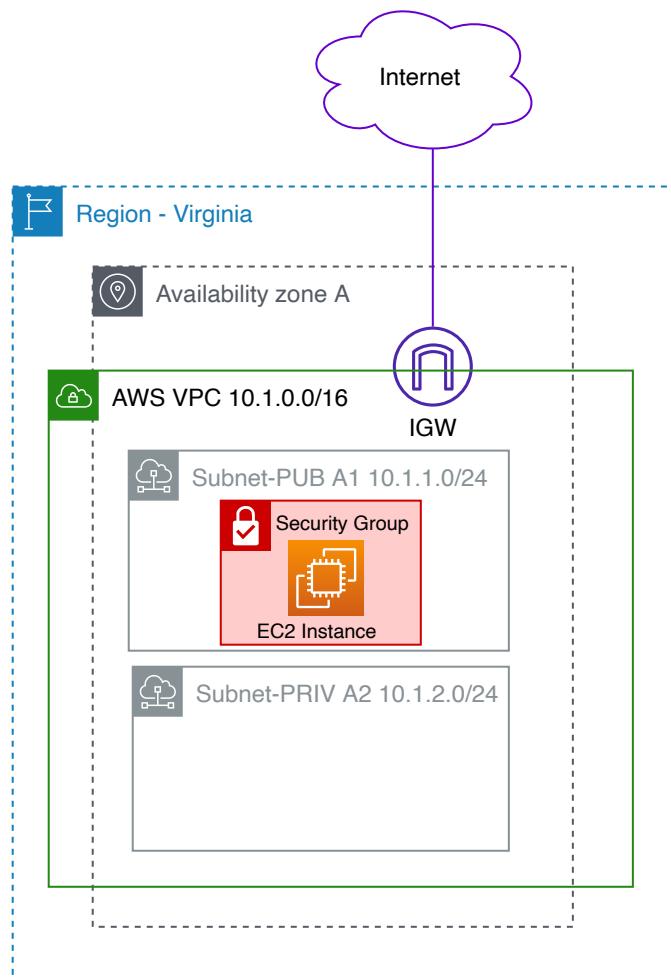
The screenshot shows the AWS VPC Management Console interface. On the left, there's a sidebar with navigation links for VPC Dashboard, EC2 Global View, Filter by VPC, VIRTUAL PRIVATE CLOUD (Your VPCs, Subnets, Route Tables, Internet Gateways, Egress Only Internet Gateways, Carrier Gateways, DHCP Option Sets, Elastic IPs, Managed Prefix Lists, Endpoints, Endpoint Services, NAT Gateways, Peering Connections), SECURITY (Network ACLs, Security Groups), and NETWORK ANALYSIS. The main content area displays 'Resources by Region' with a note that instances will launch in the US East region. It lists various Amazon VPC resources: VPCs (1), Subnets (6), Route Tables (1), Internet Gateways (1), Egress-only Internet Gateways (0), DHCP option sets (1), Elastic IPs (0), and Endpoints (0). To the right, a large table lists AWS regions and their corresponding AWS IDs. A tooltip for the 'Details' column indicates the service is operating normally. The footer includes links for Feedback, Unified Settings, Privacy, Terms, and Cookie preferences.

Region	AWS ID
US East (N. Virginia)	us-east-1
US East (Ohio)	us-east-2
US West (N. California)	us-west-1
US West (Oregon)	us-west-2
Africa (Cape Town)	af-south-1
Asia Pacific (Hong Kong)	ap-east-1
Asia Pacific (Jakarta)	ap-southeast-3
Asia Pacific (Mumbai)	ap-south-1
Asia Pacific (Osaka)	ap-northeast-3
Asia Pacific (Seoul)	ap-northeast-2
Asia Pacific (Singapore)	ap-southeast-1
Asia Pacific (Sydney)	ap-southeast-2
Asia Pacific (Tokyo)	ap-northeast-1
Canada (Central)	ca-central-1
Europe (Frankfurt)	eu-central-1
Europe (Ireland)	eu-west-1
Europe (London)	eu-west-2

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.

The screenshot shows the AWS VPC Management Console interface. In the top navigation bar, the URL is `us-east-1.console.aws.amazon.com/vpc/home?region=us-east-1#Home:`. The search bar contains the query `vpc`. The left sidebar has sections for **VPC dashboard**, **EC2 Global View**, **Filter by VPC:** (with a dropdown menu for `Select a VPC`), and **Virtual private cloud** (with links for `Your VPCs`, `Subnets`, `Route tables`, `Internet gateways`, `Egress-only internet gateways`, `Carrier gateways`, `DHCP option sets`, `Elastic IPs`, `Managed prefix lists`, `Endpoints`, and `Endpoint services`). The main content area displays search results for 'vpc'. It includes a section for **Services (12)** with links to `Blogs (767)`, `Documentation (11,960)`, `Knowledge Articles (20)`, `Tutorials (10)`, `Events (15)`, and `Marketplace (491)`. Below this is a section for **Features** with links to `VPC` (Isolated Cloud Resources), `AWS Firewall Manager` (Central management of firewall rules), `Detective` (Investigate and Analyze potential security issues), and `Managed Services` (IT operations management for AWS). At the bottom of the page, there are links for `Internet Gateways`, `Customer`, and `US East (0)`. A note about `Network Manager` is present. The footer includes links for `cloudShell`, `Feedback`, `Language`, `© 2023, Amazon Web Services, Inc. or its affiliates.`, `Privacy`, `Terms`, and `Cookie preferences`.

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 your tag convention. Use it as a best practice to differentiate from other lab environments.

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) operations on any AWS resources. You can then assign policies to an IAM role. IAM roles are temporary identities 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

You 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_XXXXXXXXXXXXXX** 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**

Policy details

Type	Customer managed	Creation time	August 13, 2023, 18:07 (UTC+02:00)	Edited time	August 13, 2023, 18:18 (UTC+02:00)	ARN
						arn:aws:iam::708113109960:policy/iSoft_Poweruser

Permissions Entities attached Tags Policy versions Access Advisor

Permissions defined in this policy Info

Permissions defined in this policy document specify which actions are allowed or denied. To define permissions for an IAM identity (user, user group, or role), attach a policy to it.

Service	Access level	Resource	Request condition
IAM	Full: List Limited: Read, Write	All resources	None
IAM Identity Center (successor to AWS SSO)	Full access	All resources	None
RDS IAM Authentication	Full access	All resources	None
IAM Identity Center (successor to AWS SSO) directory	Full access	All resources	None
IAM Roles Anywhere	Full access	All resources	None

Allow (384 of 384 services) Show remaining 0 services

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",
        "organizations:ListAccounts"
      ]
    }
  ]
}
```

```
        "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.

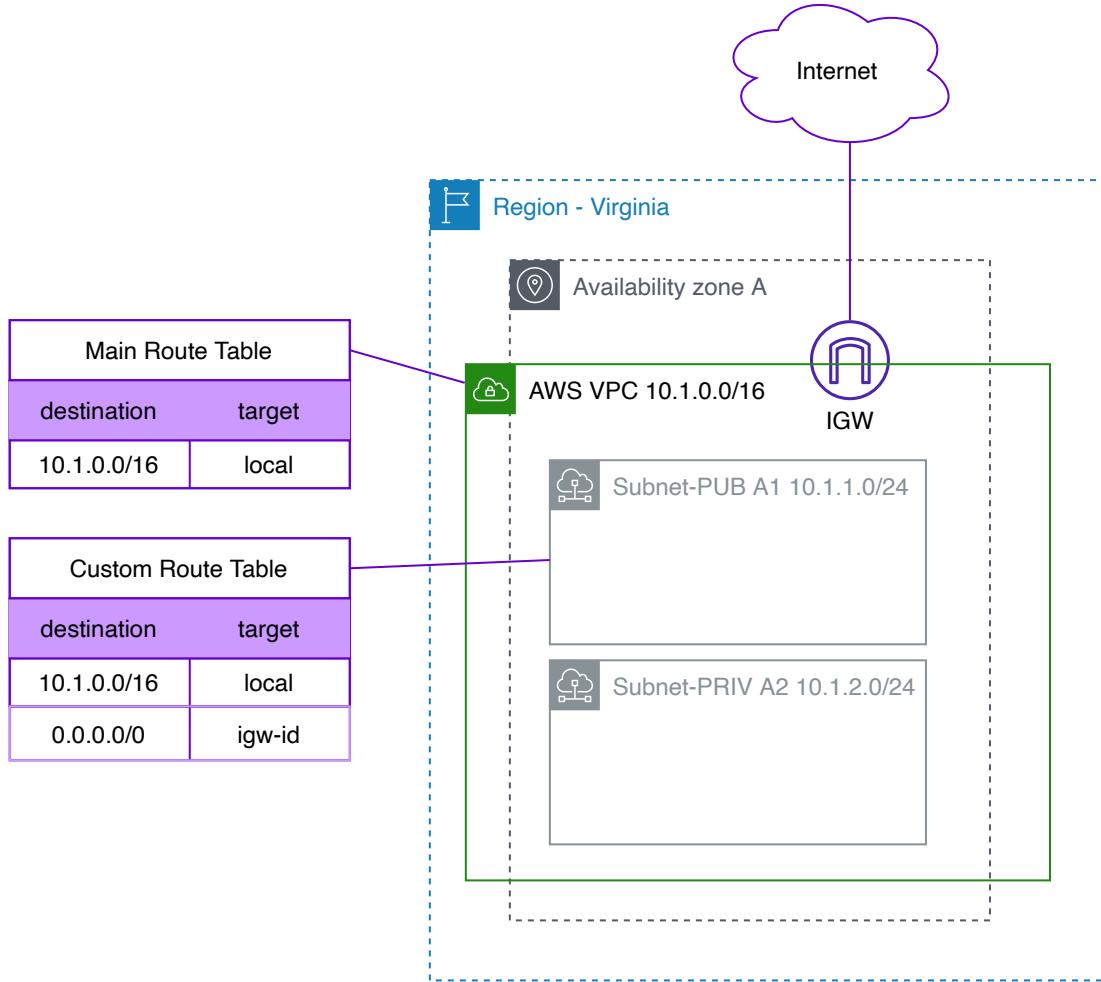
Resource: Specify the exact resource on which the policy applies. The asterisk (*) as the resource value indicates that this denial applies to all resources.

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.

The screenshot shows the AWS VPC Management Console interface on a Mac OS X desktop. The window title is "VPC Management Console". The URL in the address bar is "us-east-1.console.aws.amazon.com". The main content area is titled "Create VPC" with a "Info" link. A sub-header states: "A VPC is an isolated portion of the AWS Cloud populated by AWS objects, such as Amazon EC2 instances." Below this, there's a section titled "VPC settings" with a sub-section "Resources to create" containing two options: "VPC only" (selected) and "VPC, subnets, etc.". There are also fields for "Name tag - optional" (containing "pod1-vpc"), "IPv4 CIDR block" (containing "192.168.0.0/16"), "IPv6 CIDR block" (containing "No IPv6 CIDR block selected"), and "Tenancy" (containing "Default"). At the bottom of the page, there are links for "Feedback", "Language selection", "Privacy", "Terms", and "Cookie preferences".

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.

The screenshot shows the AWS VPC Management Console with the following details:

- VPC ID:** vpc-0ed2cf7555c1d240
- State:** Available
- Tenancy:** Default
- DHCP option set:** dopt-009751154b8f2f3e4
- Default VPC:** No
- IPv4 CIDR:** 192.168.0.0/16
- Network Address Usage metrics:** Disabled
- DNS hostnames:** Disabled
- Main route table:** rtb-09532f2438c4f7b3d
- IPv6 pool:** -
- Route 53 Resolver DNS Firewall rule groups:** -
- Owner ID:** 708113109960
- DNS resolution:** Enabled
- Main network ACL:** acl-0eb5eb4658620c42c
- IPv6 CIDR (Network border group):** -

Resource map:

- VPC:** pod1-vpc
- Subnets (0):** Subnets within this VPC
- Route tables (1):** Route network traffic to resources

Introducing the VPC resource map

- **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:

The screenshot shows the AWS VPC Dashboard. On the left, there's a sidebar with links like 'VPC Dashboard', 'EC2 Global View New', 'Filter by VPC:', 'Select a VPC', and 'Your VPCs'. Under 'Your VPCs', there are 'Subnets' and a collapsed section for 'Virtual Private Cloud'. The main area is titled 'Your VPCs (1/2)' and has an 'Info' link. A search bar at the top right contains 'Search for services, features, blogs, docs, and more...'. Below the search bar, a search result for 'lab-id:' is shown, with 'lab-id values' and 'lab-id: pod1' listed. A list of VPCs follows, with 'pod1-vpc' highlighted and checked.

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 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 And 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 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 -

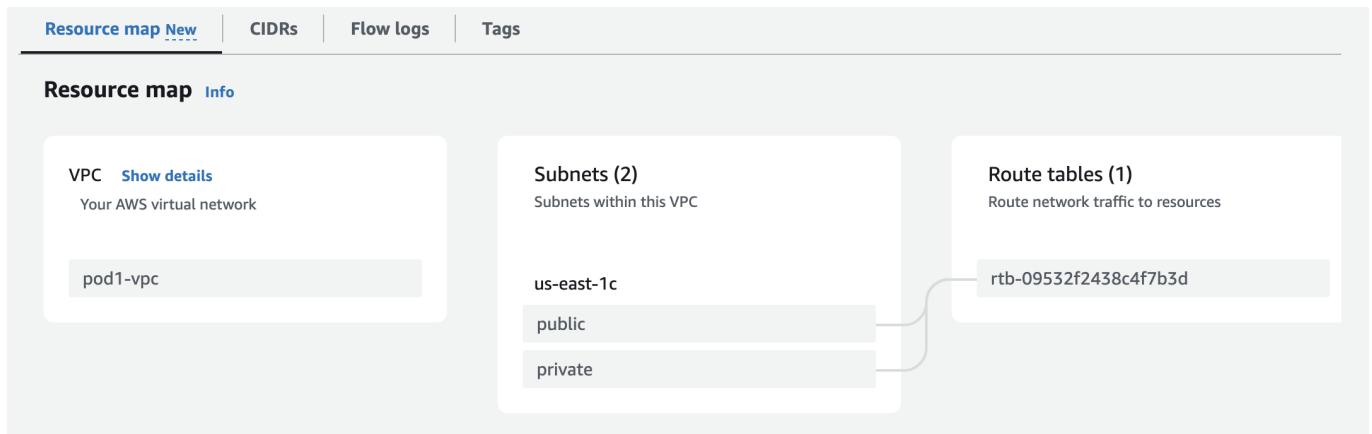
In the AWS VPC resource create two new subnet from the VPC CIDR one public and an other private.

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.

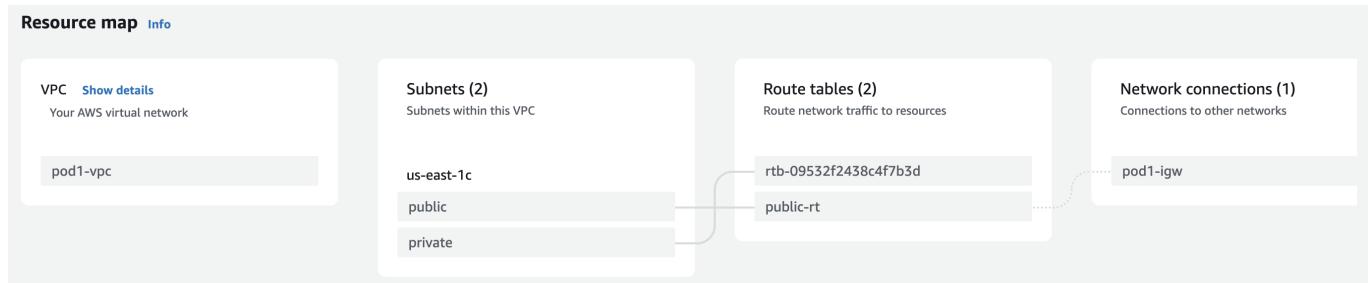
- Task 12.3 -

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 :

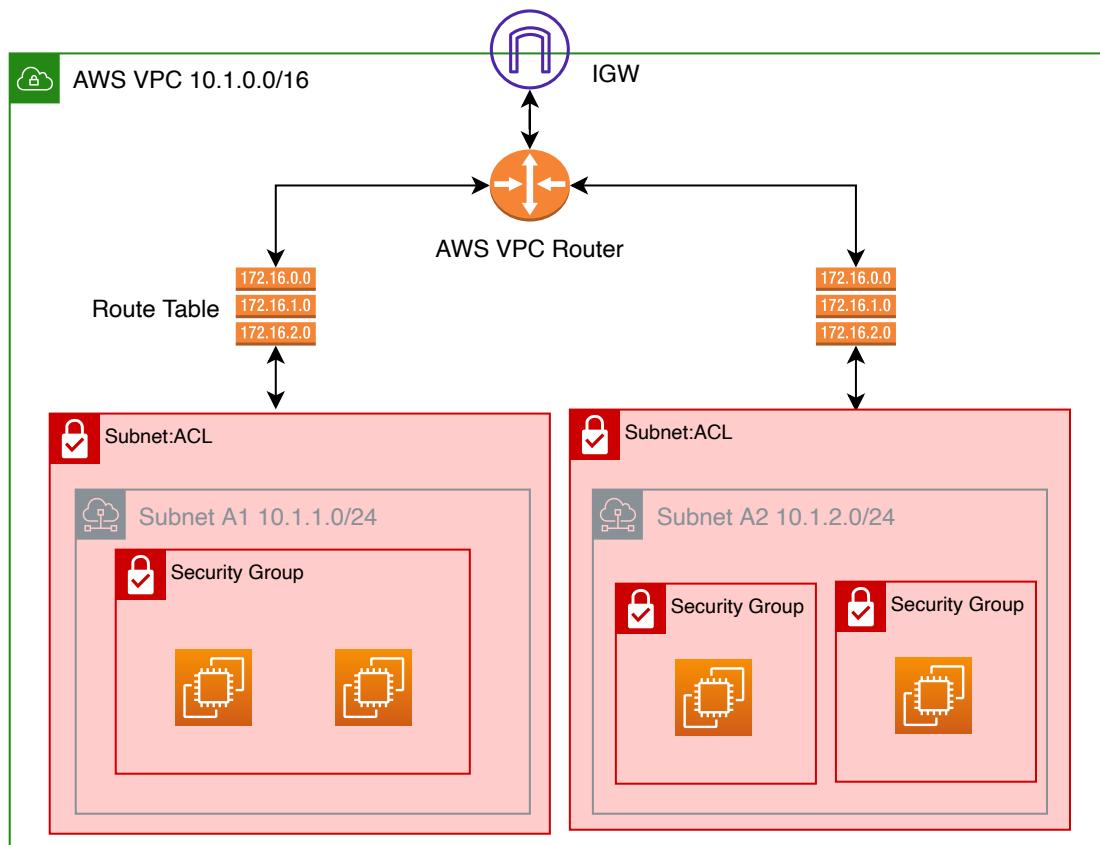


Task 13. Security

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.

▼ Network settings [Info](#)

VPC - required [Info](#)

vpc-0d491aef9bbd6972e (pod1-vpc)
192.168.0.0/16

Subnet [Info](#)

subnet-04b956c9dbe8854dc public
VPC: vpc-0d491aef9bbd6972e Owner: 708113109960
Availability Zone: us-east-1a IP addresses available: 251 CIDR: 192.168.1.0/24)

Create new subnet [\[+\]](#)

Auto-assign public IP [Info](#)

Enable

Firewall (security groups) [Info](#)

A security group is a set of firewall rules that control the traffic for your instance. Add rules to allow specific traffic to reach your instance.

Create security group Select existing security group

Common security groups [Info](#)

Select security groups

pod1-sg sg-05a36e0b28cc78d99 X
VPC: vpc-0d491aef9bbd6972e

Compare security group rules [\[+\]](#)

Security groups that you add or remove here will be added to or removed from all your network interfaces.

► Advanced network configuration

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:wbABDsF3oune0A0B2o9bHuIGnzikZa2qn4HXrIUVaB0.
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.

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

```
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 your 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

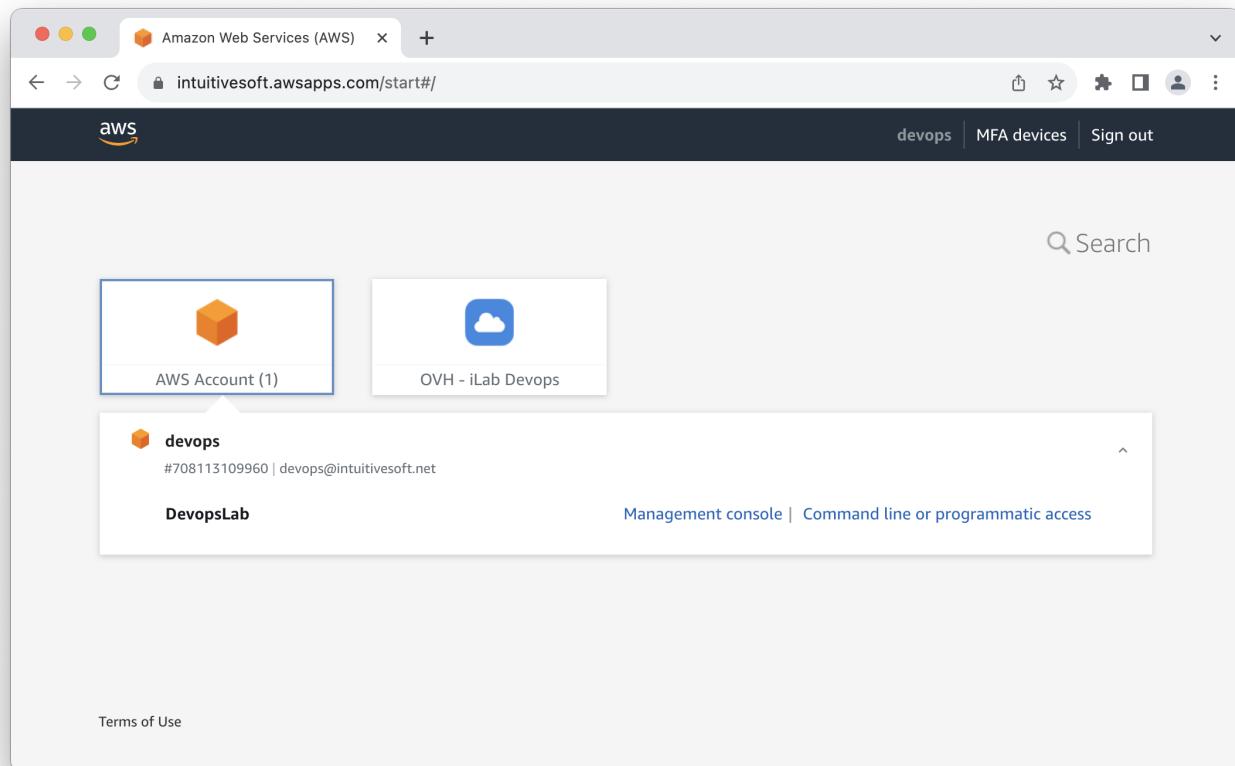
You need to specify application credentials for authentication and permissions on the AWS API.

- 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.

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": [  
        {  
            "VpcId": "vpc-0000000000000000",  
            "CidrBlock": "10.0.0.0/16",  
            "IsDefault": true,  
            "OwnerId": "708113109960",  
            "Status": "available",  
            "SubnetIds": ["subnet-0000000000000000"],  
            "Tags": [{"Key": "Name", "Value": "Default"}]  
        }  
    ]  
}
```

```

"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": [
    {
      "Instances": [
        {
          "Metadata": {
            "AWS-RunTime-Config": {
              "Configurations": [
                {
                  "Name": "aws-runconfig-1",
                  "Value": "value1"
                }
              ]
            }
          }
        }
      ]
    }
  ]
}
```

```
{  
    "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-0ed2cf7555c1d240",  
            "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": {  
                        "DeviceIndex": 0,  
                        "DeleteWithVolume": true,  
                        "Status": "attached",  
                        "VolumeId": "vol-04068f16a20c27007"  
                    }  
                }  
            ]  
        }  
    ]  
}
```

```
        "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-0ed2cf7555c1d240",
    "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 environment 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 environment 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
```

Task 13. [Optional] AWS CloudShell

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

The screenshot shows the AWS VPC Management Console. On the left, there's a sidebar for 'Virtual private cloud' with options like 'Your VPCs', 'Subnets', 'Route tables', etc. The main area has sections for 'Create VPC' and 'Launch EC2 Instances'. Below that is 'Resources by Region' with counts for VPCs, Subnets, Route Tables, Internet Gateways, NAT Gateways, VPC Peering Connections, Network ACLs, and Security Groups. To the right are 'Service Health' (with a link to 'View complete service health details'), 'Settings' (including 'Zones' and 'Console Experiments'), 'Additional Information' (links to 'VPC Documentation', 'All VPC Resources', 'Forums', and 'Report an Issue'), and 'AWS Network Manager'. At the bottom, there's a CloudShell window titled 'us-east-1' showing a command-line session:

```
[cloudshell-user@ip-10-4-80-39 ~]$ aws ec2 describe-vpcs
{
  "Vpcs": [
    {
      "CidrBlock": "192.168.0.0/16",
      "DhcpOptionsId": "dopt-009751154b8f2f3e4",
      "State": "available",
      "VpcId": "vpc-0ed2c7d7555c1d240",
      "OwnerId": "708113109960",
      "InstanceTenancy": "default",
      "CidrBlockAssociationSet": [
        {
          "AssociationId": "vpc-cidr-assoc-081f857b1b0248c2b",
          "CidrBlock": "192.168.0.0/16",
          "CidrBlockState": {
            "State": "associated"
          }
        }
      ]
    }
  ]
}
```

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...).

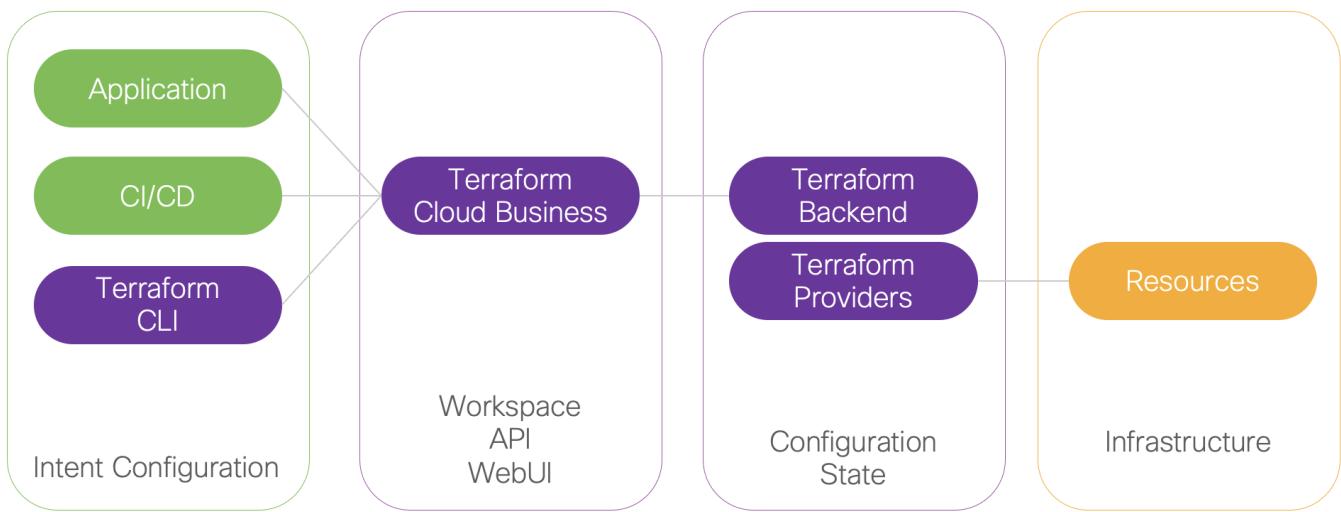
The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

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 keeps configuration states (real view of resources configuration) and figures 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.

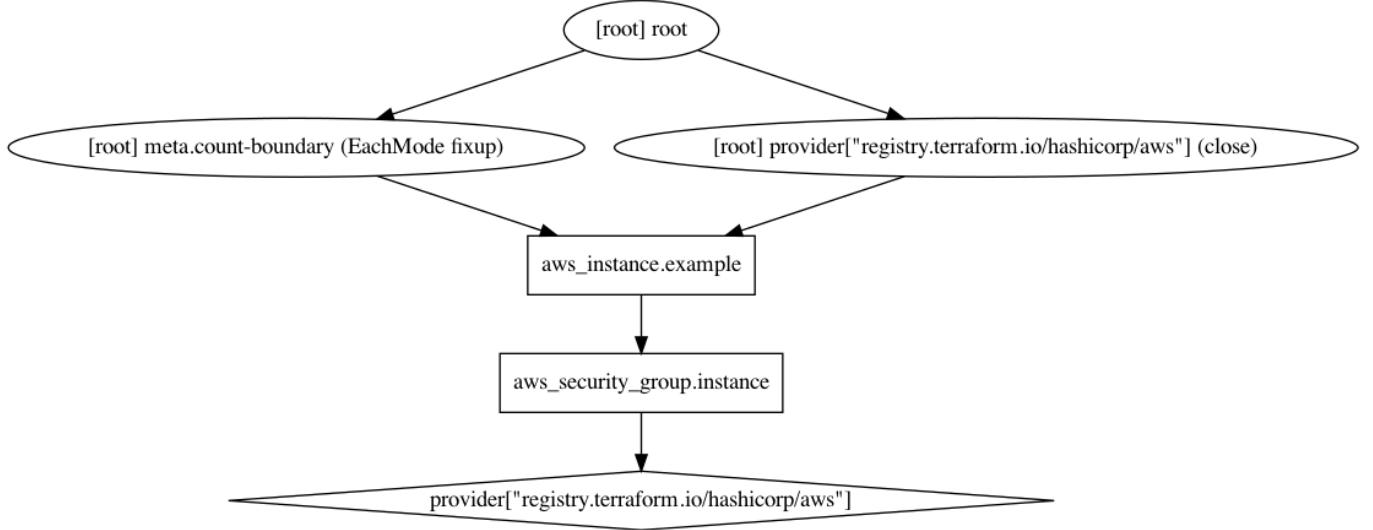
Insight 4. Workflow

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

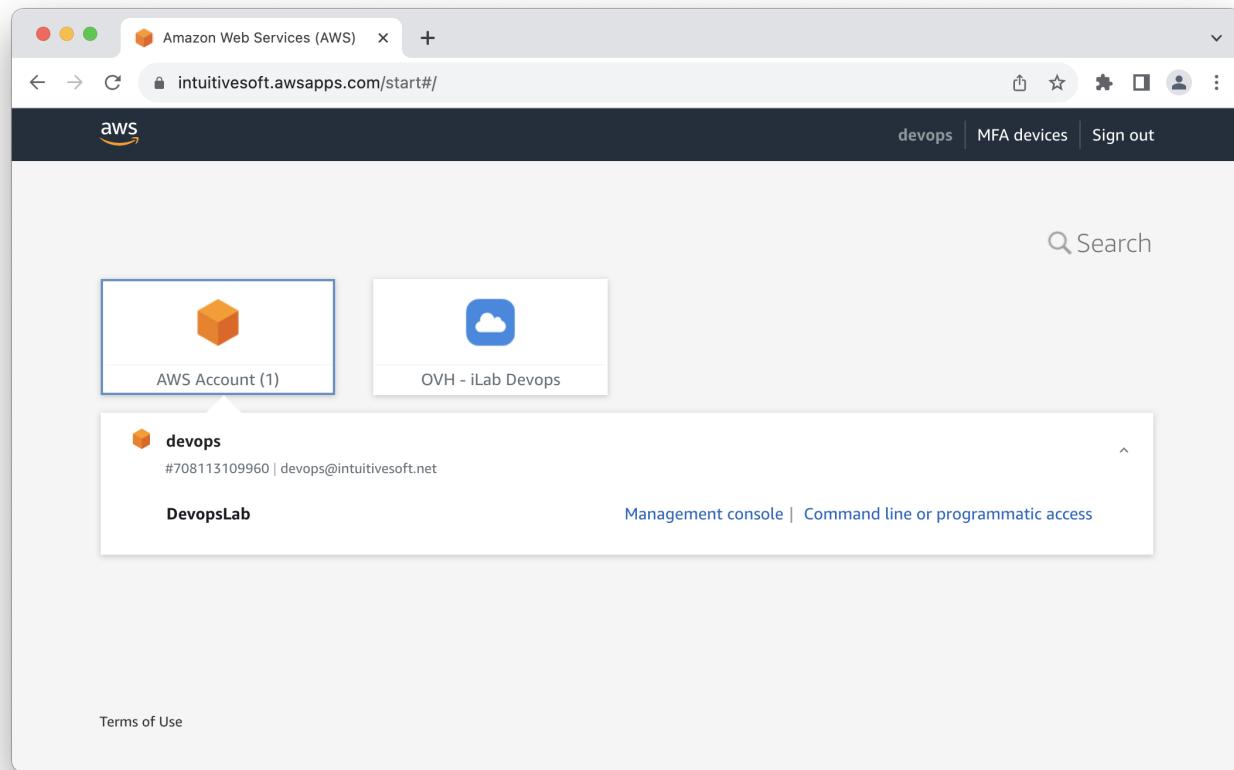
You need to specify application credentials for authentication and permissions on the AWS API.

- 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 keeps 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 = "podox-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 = "podox-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" = "podox-public-subnet"
    }
    + tags_all                            = {
        + "Name" = "podox-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" = "podox-public-subnet"
    }
    + tags_all                               = {
        + "Name" = "podox-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:

The screenshot shows the AWS VPC Subnets page. On the left sidebar, under the 'VIRTUAL PRIVATE CLOUD' section, 'Subnets' is selected. The main content area displays a table titled 'Subnets (1/1)'. The table has columns: Name, Subnet ID, State, and VPC. There is one row for 'pdx-subnet-public' with the details: subnet-0d98197a7dbb84d10, Available, and vpc-00bfb9fd3a6691af | podx. A search bar at the top is set to 'Name: pdx-subnet-public'.

Name	Subnet ID	State	VPC
pdx-subnet-public	subnet-0d98197a7dbb84d10	Available	vpc-00bfb9fd3a6691af podx

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.

The screenshot shows the AWS VPC dashboard. In the center, there's a table titled 'Subnets (1/1) Info' with one item. The item has a 'Name' field containing 'pod1-public-subnet-test'. A modal window is open over this row, with the 'Edit Name' field set to 'wrong-name'. At the bottom of the modal are 'Cancel' and 'Save' buttons. The status bar at the top right indicates 'Available'.

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: Refreshing state... [id=subnet-0783f8621ad659e83]
~ 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 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
# 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 13. 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 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
    null
        - availability_zone_id = "use1-az4" -> null
    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
        null
            - private_dns_hostname_type_on_launch = "ip-name" -> null
            - tags =
                - "Name" = "podox-public-subnet"
            } -> null
        - tags_all =
            - "Name" = "podox-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:

The screenshot shows the AWS VPC Subnets page. On the left sidebar, under 'Your VPCs', 'Subnets' is selected. The main area displays a table titled 'Subnets' with one row. The row contains the following data:

Name	Subnet ID	State	VPC
podx-subnet-public			

Below the table, there is a search bar labeled 'Filter subnets' and a button labeled 'Clear filters'.

Task 14. Create Terraform variables

-Task 14.1-

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

-Task 14.2-

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 = "pox-public-subnet"

```

Task 15. Re-write your script with variable references

The new `main.tf` file is re-written to leverage the variables definition file as they are now 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 `var` 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 16. 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 17. 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 18. 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 19. 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 20. 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 21. 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 = "podox-public-subnet"
 - cidr_block = "192.168.1.0/24"
- subnet2:
 - tag:Name = "podox-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 22. 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:

Name	Subnet ID	State	VPC	IPv4 CIDR
podox-public-subnet	subnet-00d8e57de905b925f	Available	vpc-00fb9fd3a6691af4 pod...	192.168.1.0/24
podox-private-subnet	subnet-0764c65210848260c	Available	vpc-00fb9fd3a6691af4 pod...	192.168.2.0/24

Task 23. 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 = "podox_public_route_table"
- associate the route table with already provisioned **podox-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**

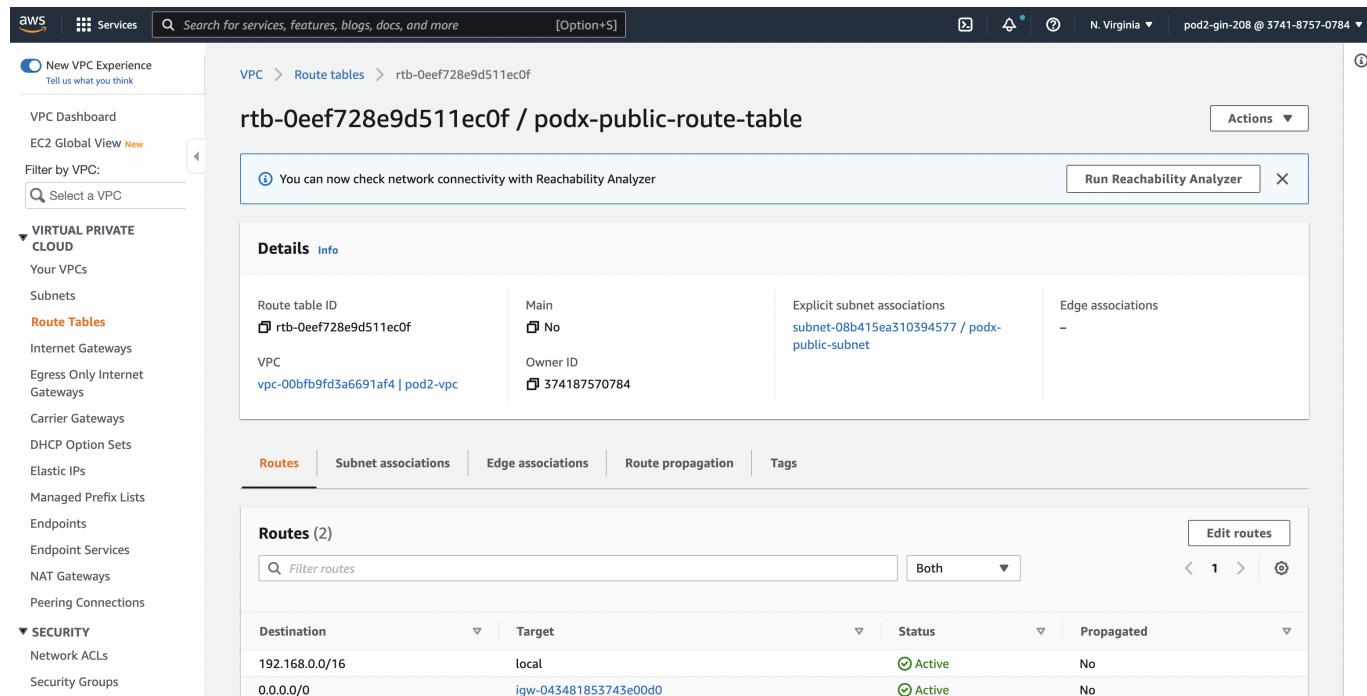
Ideally create a folder named **create-route-table** to place your script and other files.

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 24. 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:



The screenshot shows the AWS VPC Route Tables page. The route table ID is `rtb-0eef728e9d511ec0f`, which is associated with the VPC `pod2-vpc`. It has two routes: one to the local subnet and another to the internet gateway (`igw-043481853743e00d0`). The table is explicitly associated with the subnet `subnet-08b415ea310394577`.

Destination	Target	Status	Propagated
<code>192.168.0.0/16</code>	<code>local</code>	<code>Active</code>	No
<code>0.0.0.0/0</code>	<code>igw-043481853743e00d0</code>	<code>Active</code>	No

Task 25. 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 = "podox-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 26. 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:

Security group name	Security group ID	Description	VPC ID
sg-035051cc8b8f16fb9 - podx-ssh-security-group	sg-035051cc8b8f16fb9	Managed by Terraform	vpc-00fb9fd3a6691af4

Owner	Inbound rules count	Outbound rules count
374187570784	1 Permission entry	1 Permission entry

Inbound rules (1/1)

Name	Security group rule...	IP version	Type	Protocol	Port range
-	sgr-06a2bf212b720801a	IPv4	SSH	TCP	22

Task 27. 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 = "podox-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 28. 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:

	Name	Type	Created	Fingerprint	ID
<input type="checkbox"/>	podox-vm-key	rsa	2022/05/06 08:50 GMT+2	a3:88:97:92:29:94:ca:b6:4c:b0:53:4a:f...	key-029da9f7c9078108c

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 29. 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 = `podox-vm`
- Image = `ubuntu-focal-20.04-amd64-server-20211129`
- Subnet = `podox-public-subnet`
- Security group = `podox-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 30. 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:

The screenshot shows the AWS EC2 Instances page. The left sidebar has links for New EC2 Experience, EC2 Dashboard, EC2 Global View, Events, Tags, and Limits. The main area shows a table with one row for an instance named 'podox-vm' with ID 'i-01f2a4d2f777b57b1'. The instance is listed as 'Running' with a Public IPv4 of '34.201.76.90'. It is associated with the security group 'podox-ssh-security-group' and key pair 'podox-vm-key'. There are buttons for 'Connect', 'Actions', and 'Launch instances'.

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.
```

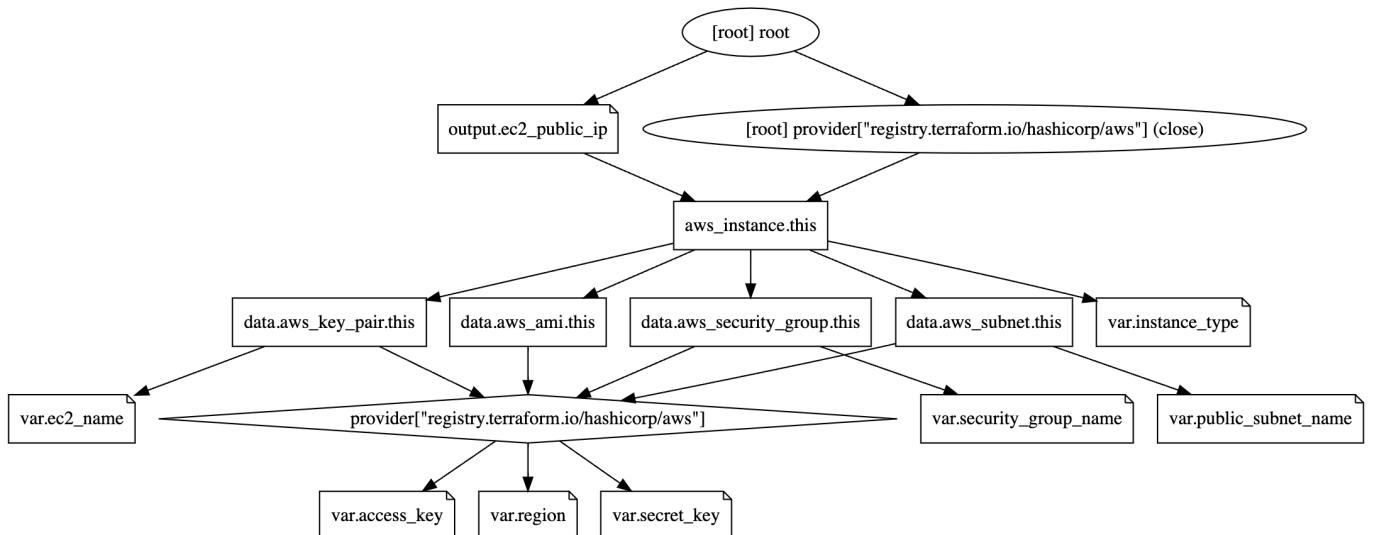
```
ubuntu@ip-192-168-1-46:~$
```

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

Task 31. 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:



AWS First Connect

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.

The screenshot shows an email from 'no-reply@login.awsapps.com' to 'À moi'. The subject is 'Boîte de réception'. The email body contains the AWS logo and a message to 'Hello LADHUIE,'. It states that an administrator has requested verification and provides a blue button to 'Verify your email address'. A note below says the link expires in 7 days. At the bottom, there is a small legal notice about Amazon Web Services.

Hello LADHUIE,

Your administrator has requested you to verify your email address. To complete email verification, choose the following link:

[Verify your email address](#)

This link will expire in 7 days.

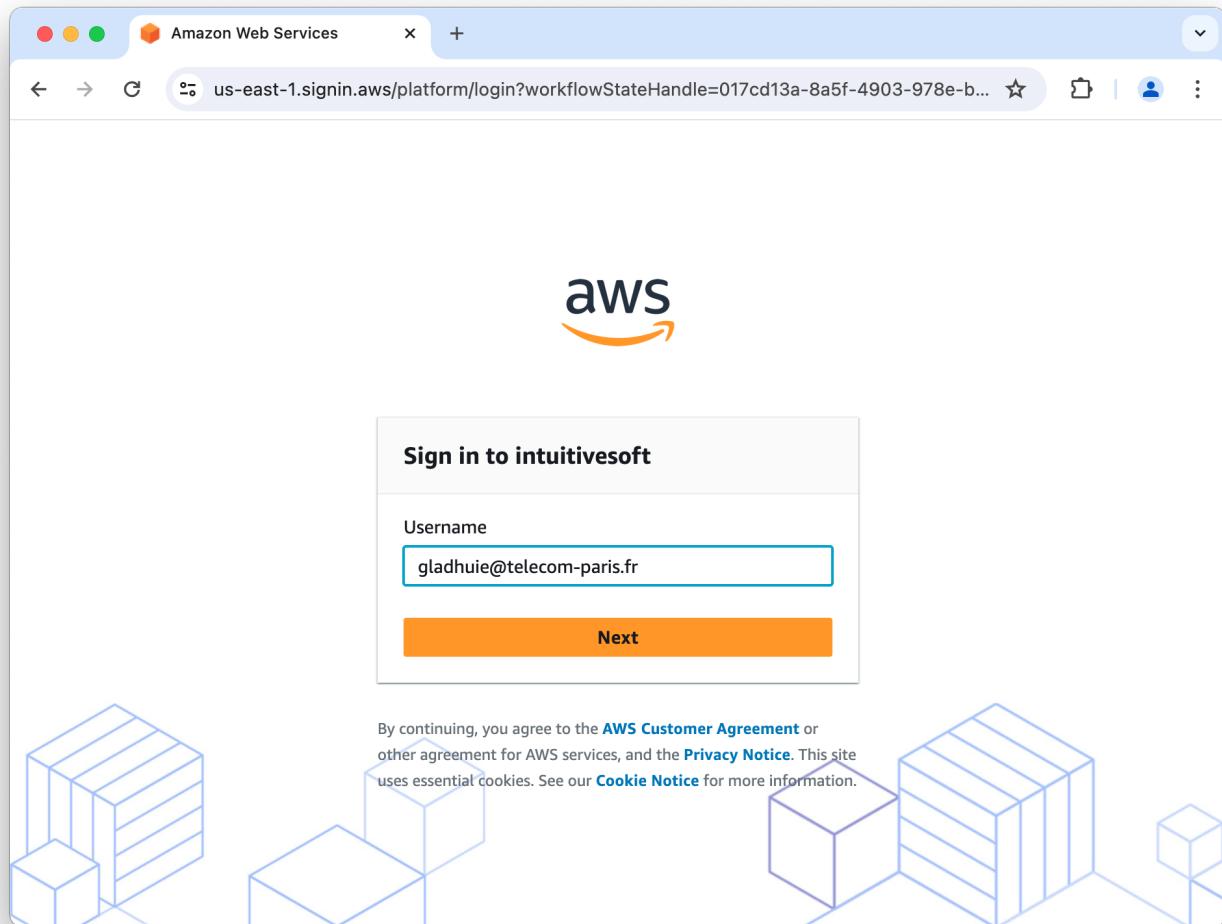
Amazon Web Services, Inc. is a subsidiary of Amazon.com, Inc. Amazon.com is a registered trademark of Amazon.com, Inc. This message was produced and distributed by Amazon Web Services, Inc., 410 Terry Ave. North, Seattle, WA 98109-5210.

Task 2. First Login

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

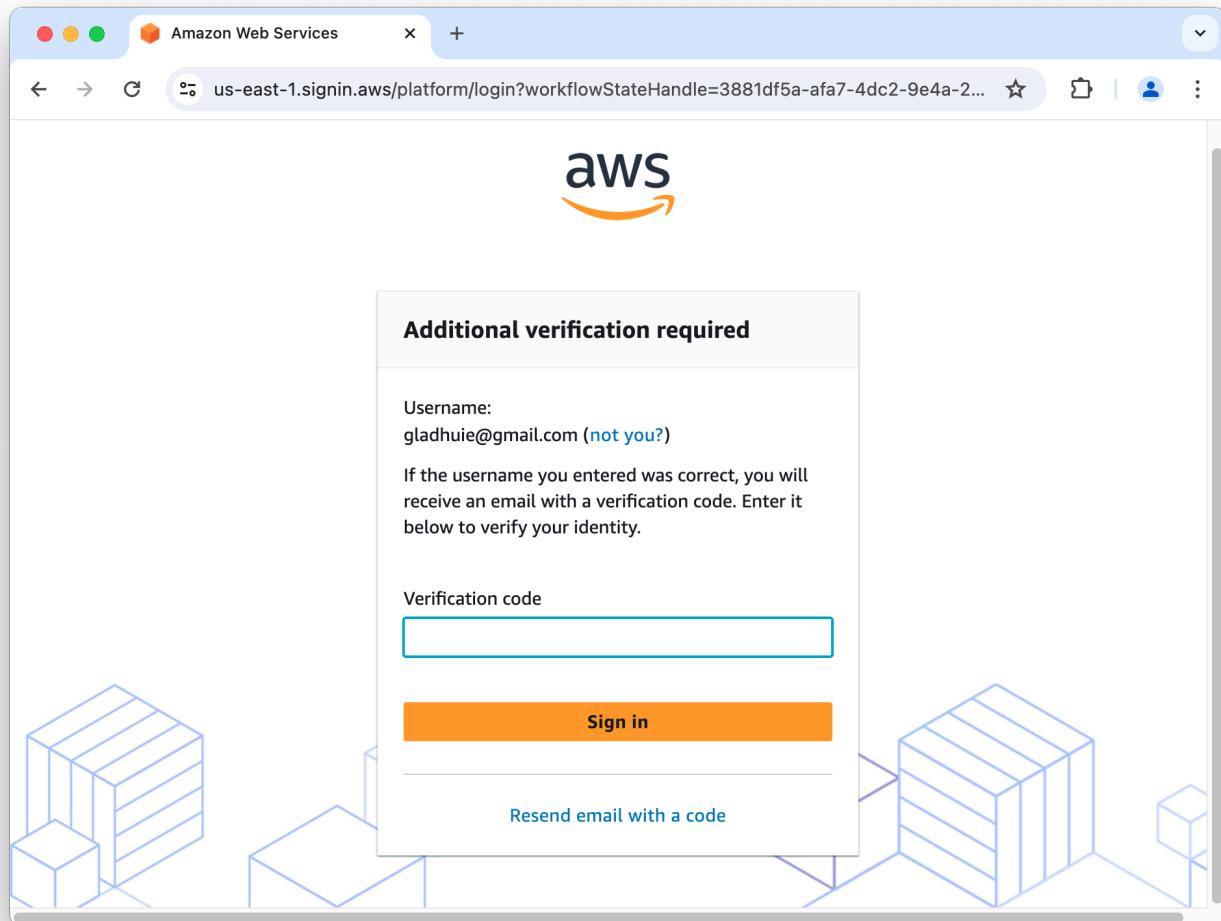
Enter your login username by default it's your email address.

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



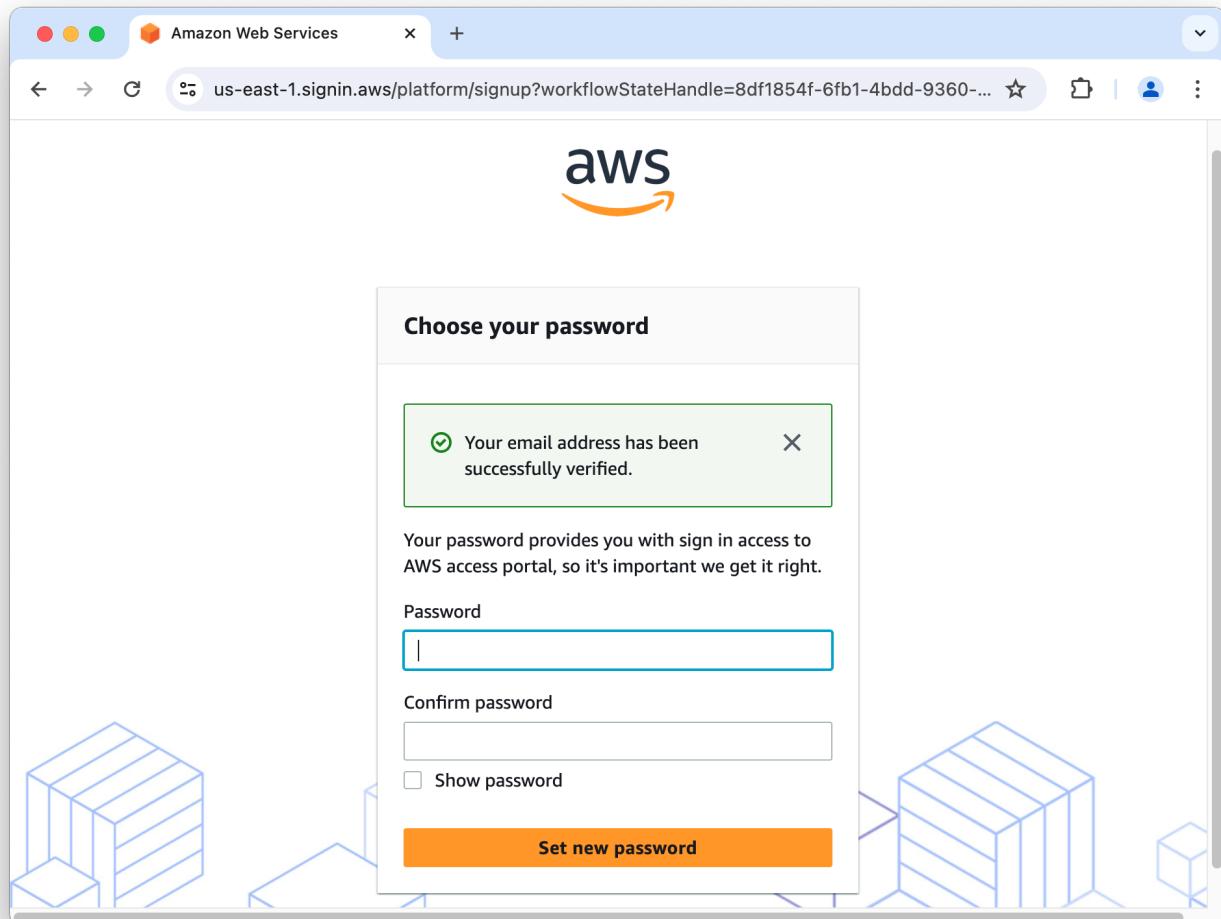
Task 3. Verification code

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.

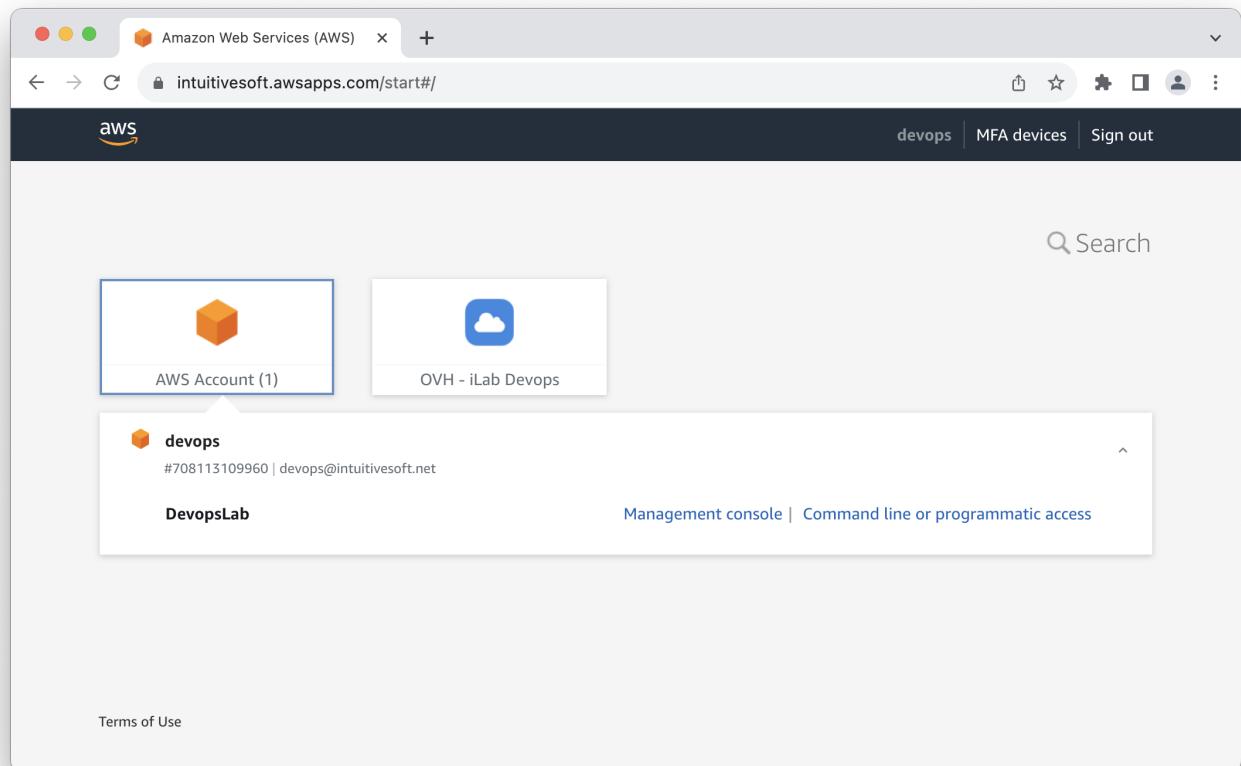
The screenshot shows a web browser window titled "AWS access portal" with the URL "intuitivesoft.awsapps.com/start#/?tab=accounts". The interface is dark-themed. At the top, there's a navigation bar with the AWS logo, the user name "LADHUIE", "MFA devices", and a "Sign out" button. A blue banner at the top displays a message about the "Create shortcut" button, which allows users to generate secure shortcut links to AWS Management Console pages. Below the banner, the main title "AWS access portal" is centered. Underneath, there are two tabs: "Accounts" (which is selected) and "Applications". The "Accounts" section has a heading "AWS accounts (1)" and a search bar with the placeholder "Filter accounts by name, ID, or email address". A single account entry is listed: "devops" (ID: 708113109960 | Email: devops@intuitivesoft.net). To the right of this entry is a "Create shortcut" button. At the bottom of the page, there are links for "Feedback", "©2024, Amazon Web Services, Inc. or its affiliates. All rights reserved.", "Privacy", "Terms", and "Cookie Preferences".

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 **Management console** 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 screenshot shows the AWS VPC Management Console interface. The top navigation bar includes the AWS logo, a search bar, and tabs for 'Services' and 'VPC Management Console'. Below the search bar is a 'Launch VPC Wizard' button and a 'Launch EC2 Instances' button. A note states: 'Note: Your Instances will launch in the US East region.' On the left, a sidebar lists categories like 'Virtual Private Cloud', 'Security', and 'Network Analysis'. The main content area is titled 'Resources by Region' and displays a grid of Amazon VPC resources across various AWS regions. The grid includes columns for resource type, count, and region. A tooltip for the 'Details' column indicates 'Service is operating normally'. The regions listed are:

Region	Region ID
US East (N. Virginia)	us-east-1
US East (Ohio)	us-east-2
US West (N. California)	us-west-1
US West (Oregon)	us-west-2
Africa (Cape Town)	af-south-1
Asia Pacific (Hong Kong)	ap-east-1
Asia Pacific (Jakarta)	ap-southeast-3
Asia Pacific (Mumbai)	ap-south-1
Asia Pacific (Osaka)	ap-northeast-3
Asia Pacific (Seoul)	ap-northeast-2
Asia Pacific (Singapore)	ap-southeast-1
Asia Pacific (Sydney)	ap-southeast-2
Asia Pacific (Tokyo)	ap-northeast-1
Canada (Central)	ca-central-1
Europe (Frankfurt)	eu-central-1
Europe (Ireland)	eu-west-1
Europe (London)	eu-west-2

At the bottom, there are links for 'Feedback', 'Language selection', 'Unified Settings', and copyright information: '© 2022, Amazon Web Services, Inc. or its affiliates.'

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.

The screenshot shows the AWS VPC Management Console interface. In the top navigation bar, the URL is `us-east-1.console.aws.amazon.com/vpc/home?region=us-east-1#Home:`. A search bar contains the query `vpc`. The left sidebar has sections for **VPC dashboard**, **EC2 Global View**, **Filter by VPC:** (with a dropdown menu for `Select a VPC`), and **Virtual private cloud** (with sub-options like `Your VPCs`, `Subnets`, `Route tables`, etc.). The main content area displays search results for 'vpc'. It includes a heading 'Search results for 'vpc'' and a note 'Try searching with longer queries for more relevant results'. The results are categorized into **Services** (12) and **Features** (56). Under **Services**, there are four items: **VPC** (Isolated Cloud Resources), **AWS Firewall Manager** (Central management of firewall rules), **Detective** (Investigate and Analyze potential security issues), and **Managed Services** (IT operations management for AWS). Under **Features**, there is a link to 'See all 56 results'. At the bottom of the page, there are links for `cloudShell`, `Feedback`, `Language`, and footer information including copyright notice (© 2023, Amazon Web Services, Inc. or its affiliates.), privacy policy, terms of service, and cookie preferences.

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

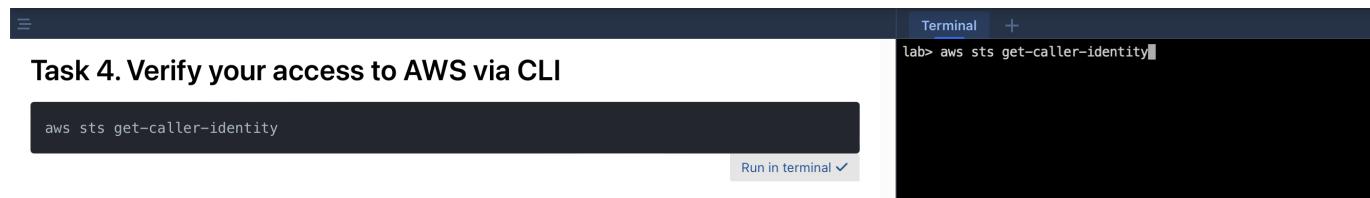
```
[default]
sso_session = devops
sso_account_id = 708113109960
sso_role_name = DevopsLab
region = us-east-1
output = json
[sso-session devops]
sso_start_url = https://intuitivesoft.awsapps.com/start#
sso_region = us-east-1
sso_registration_scopes = sso:account:access
```

Annex : Intuitive Lab Dashboard

Terminal

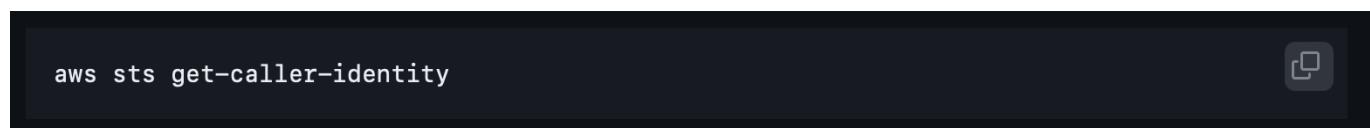
By default the lab dashboard layout offers a live shell terminal to interact with your live lab instance.

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



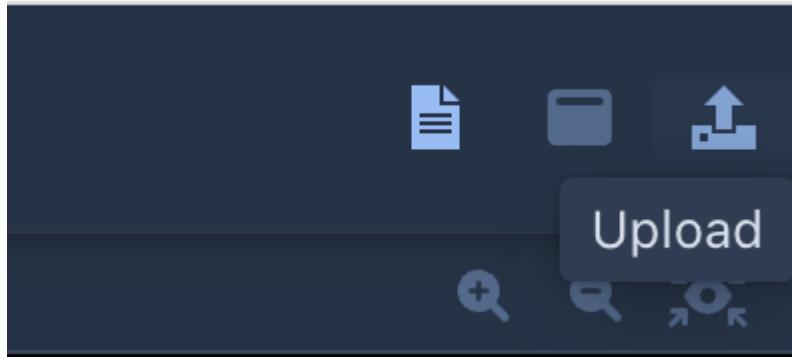
Note : This option is only available in **Terminal View**

Alternatively you can copy the content of box using the copy feature on the box top right corner.



Upload files to Lab Dashboard

You can upload files to your lab instance using the device browser **Upload** :



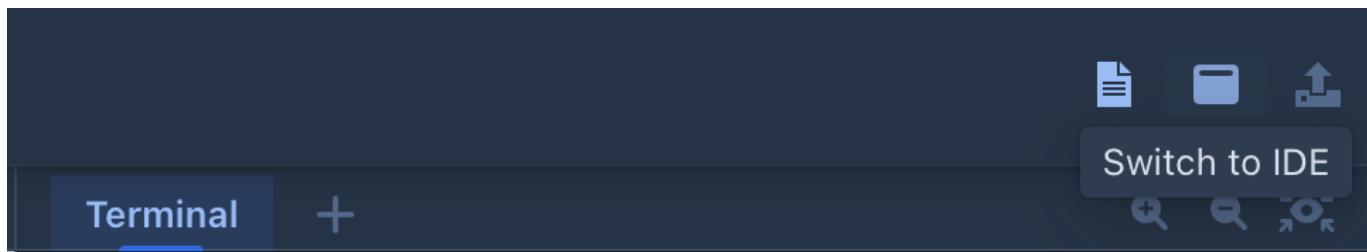
Files are stored in `~/files` folder.

Alternatively you can copy/paste or drag and drop files using Visual Studio Code [File Explorer](#).

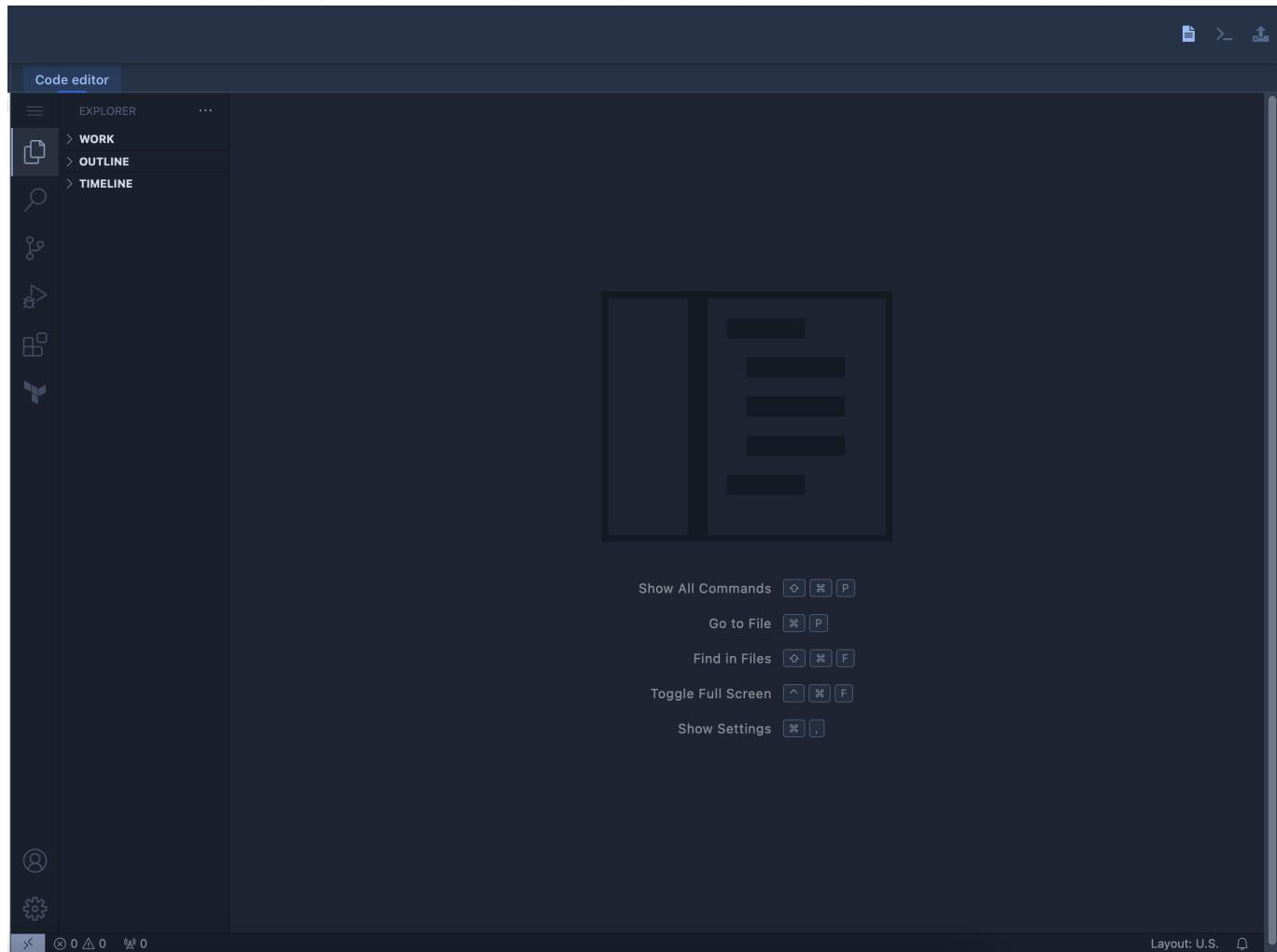
Visual Studio Code

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.

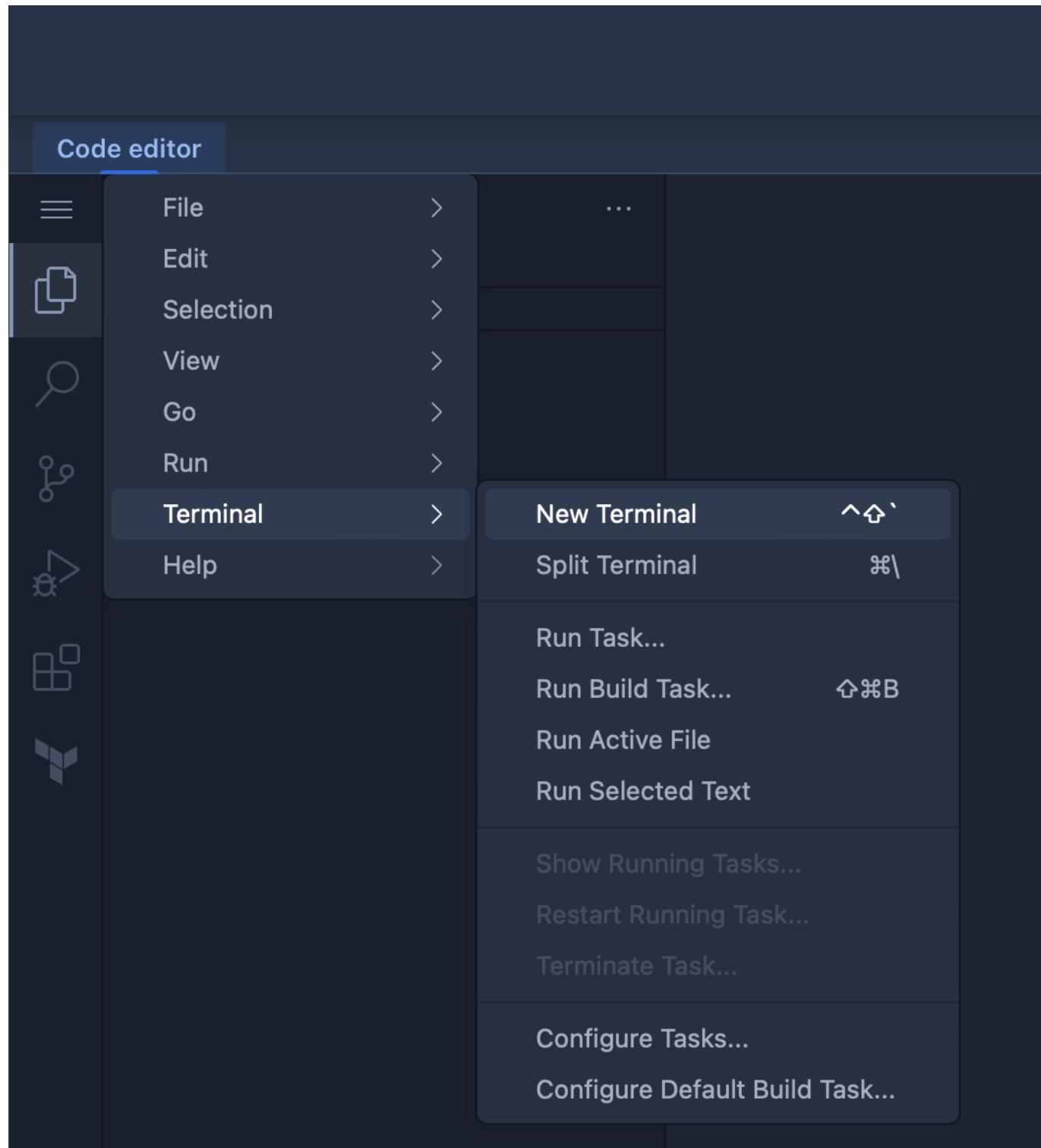
- Extensions Manager -

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-vscode.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

gin208