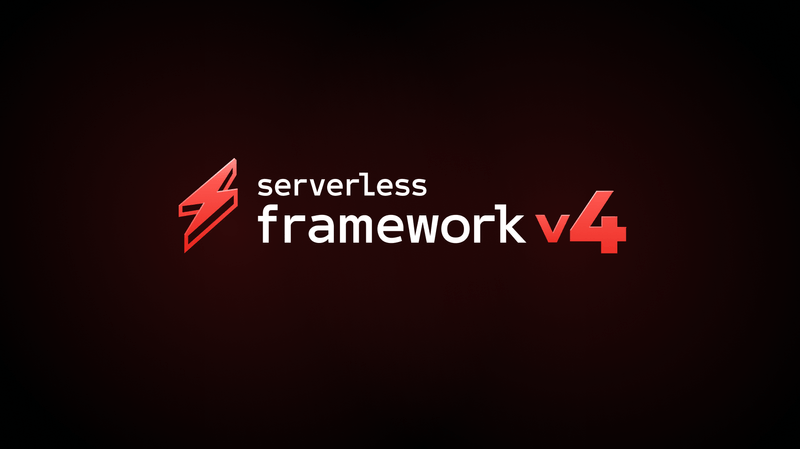
0️⃣ Activity 0: Overall Problem Statement



# **Overall Problem Statement**

Status In progress

Timing Sep 30, 2025 to Oct 3, 2025

Owners [Soumik Dutta](mailto:23m0826@iitb.ac.in)

Welcome! In this lab, you'll embark on a practical journey to master one of the most popular tools in modern cloud development: the **Serverless Framework**. We'll move beyond theory and build a real, event-driven application on AWS.

### Problem Statement

Imagine you're a backend developer at a fast-growing startup. Your team needs to build a new feature that can handle unpredictable traffic, scale automatically, and keep operational costs low. The traditional approach of provisioning servers, managing operating systems, and configuring load balancers is too slow and complex.

Your mission is to build this feature as a scalable, event-driven application using a serverless architecture. You will use the **Serverless Framework** as your primary tool. It will act as your Infrastructure as Code (IaC) command center, allowing you to define all your cloud resources—functions, APIs, databases, and message queues—in a single configuration file and deploy them with one command.

### Learning Objectives 🎯

By the end of this lab, you will be able to:

* Configure a professional development environment for serverless projects.
* Explain the core concepts of the Serverless Framework, such as services, providers, functions, events, and resources.
* Write a serverless.yml file to define and configure AWS Lambda, API Gateway, DynamoDB, SQS, and SNS.
* Deploy, test, and monitor a multi-service serverless application on AWS.
* Cleanly remove all cloud resources managed by the framework.

### Prerequisites ✅

Before you begin, please ensure you have the following:

* **An Active AWS Account:** You will need an account with administrative privileges to create and manage the resources in this lab. It is highly recommended to use a personal, development, or sandbox account, **not** a production environment.
* **Foundational AWS Knowledge:** You should have already completed introductory labs or possess a basic understanding of:
  + **IAM:** The difference between a user, a role, and a policy. You know what programmatic access keys are and why they are used.
  + **Lambda:** The basic concept of a Lambda function as a compute service that runs code in response to events.
* **Command Line Familiarity:** You should be comfortable opening a terminal or command prompt, navigating directories, and running basic commands.

### Lab Roadmap 🗺️

Here’s a quick overview of the activities we will complete:

* **Activity 1: Environment Setup:** First, we'll set up your local machine with all the necessary tools, including the AWS CLI and the Serverless Framework itself.
* **Activity 2: Core Concepts:** Next, we'll dive into the theory, understanding the "why" and "what" behind the Serverless Framework and its key abstractions.
* **Activity 3: Guided Project:** This is where the fun begins! You'll build a complete, multi-part serverless application with step-by-step guidance.
* **Activity 4: The Challenge:** Time to test your new skills. You will apply your knowledge to build a new service by writing your own serverless configuration.

═══ END OF DOCUMENT ═══

1️⃣ Activity 1: Environment Setup

### **Activity 1: Environment Setup 🔧**

Objective🎯

In this activity, we will prepare your local machine and AWS account, creating a professional development environment ready for building serverless applications. Follow these steps carefully.

### ⚠️ Important Note: This is an Ephemeral Activity

Please be aware that this activity is **ephemeral**.

If you exit the activity and come back later, **all previous states and progress made in your CLAB terminal will be lost**, and you will have to start over from the beginning.

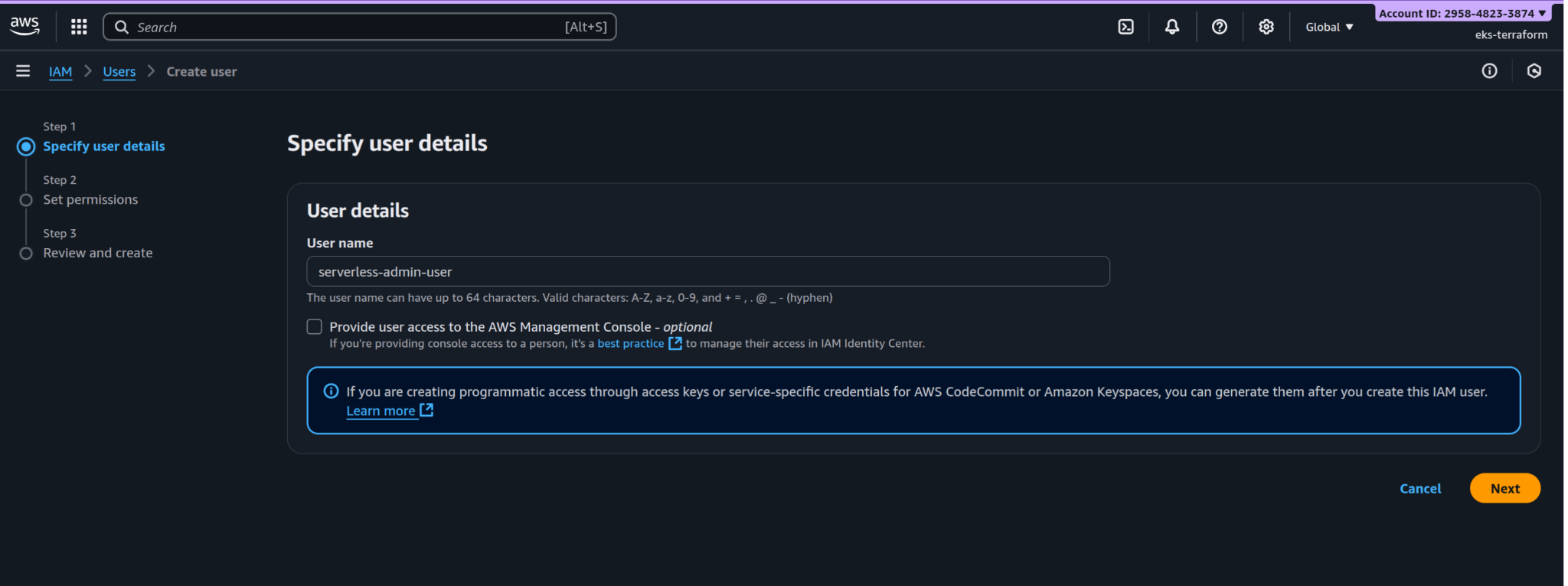
If you return, you must run the command below before starting again.

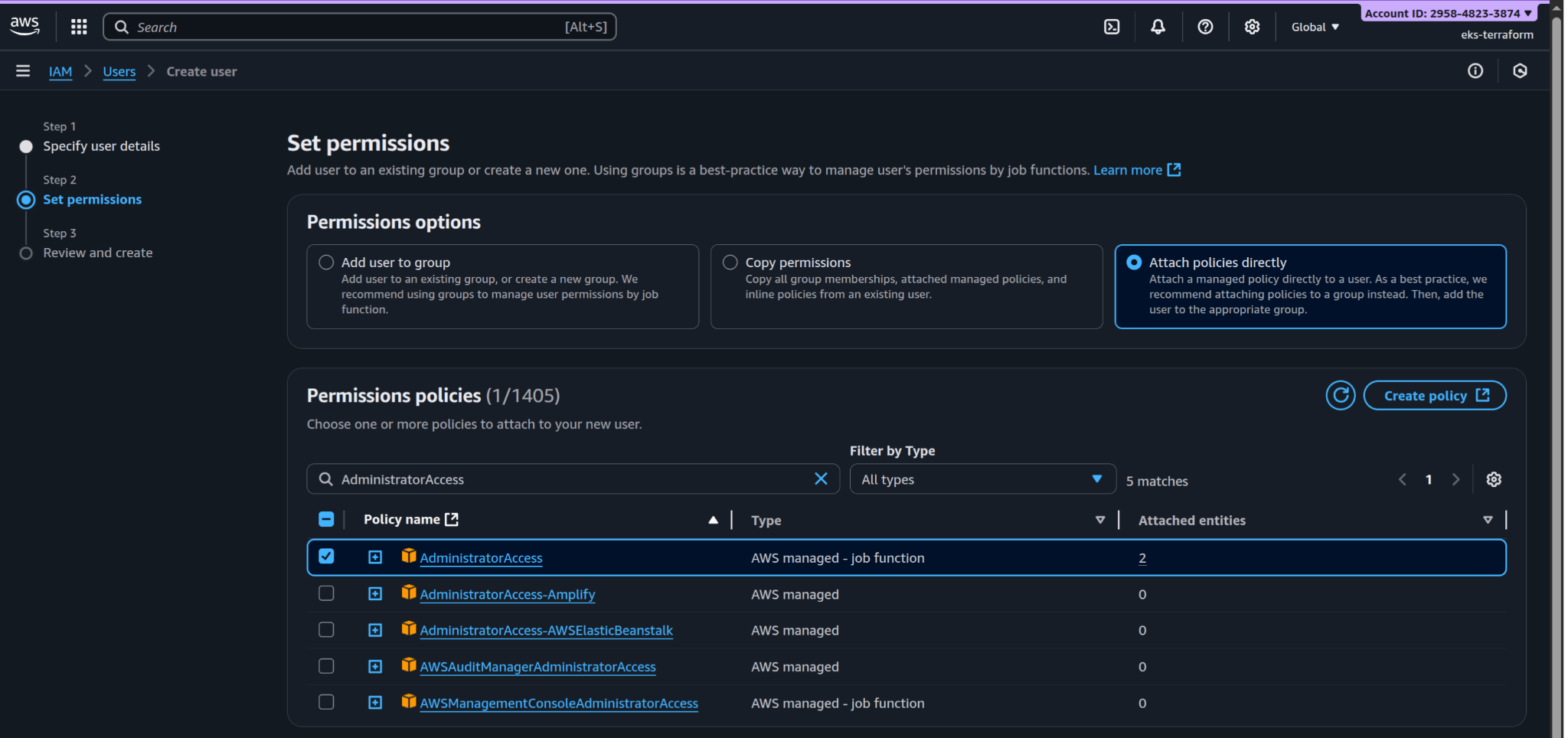
find . -maxdepth 1 ! -name 'problem\_statement.txt' ! -name 'task.txt' ! -name .git ! -name '.' -exec rm -rf {} +



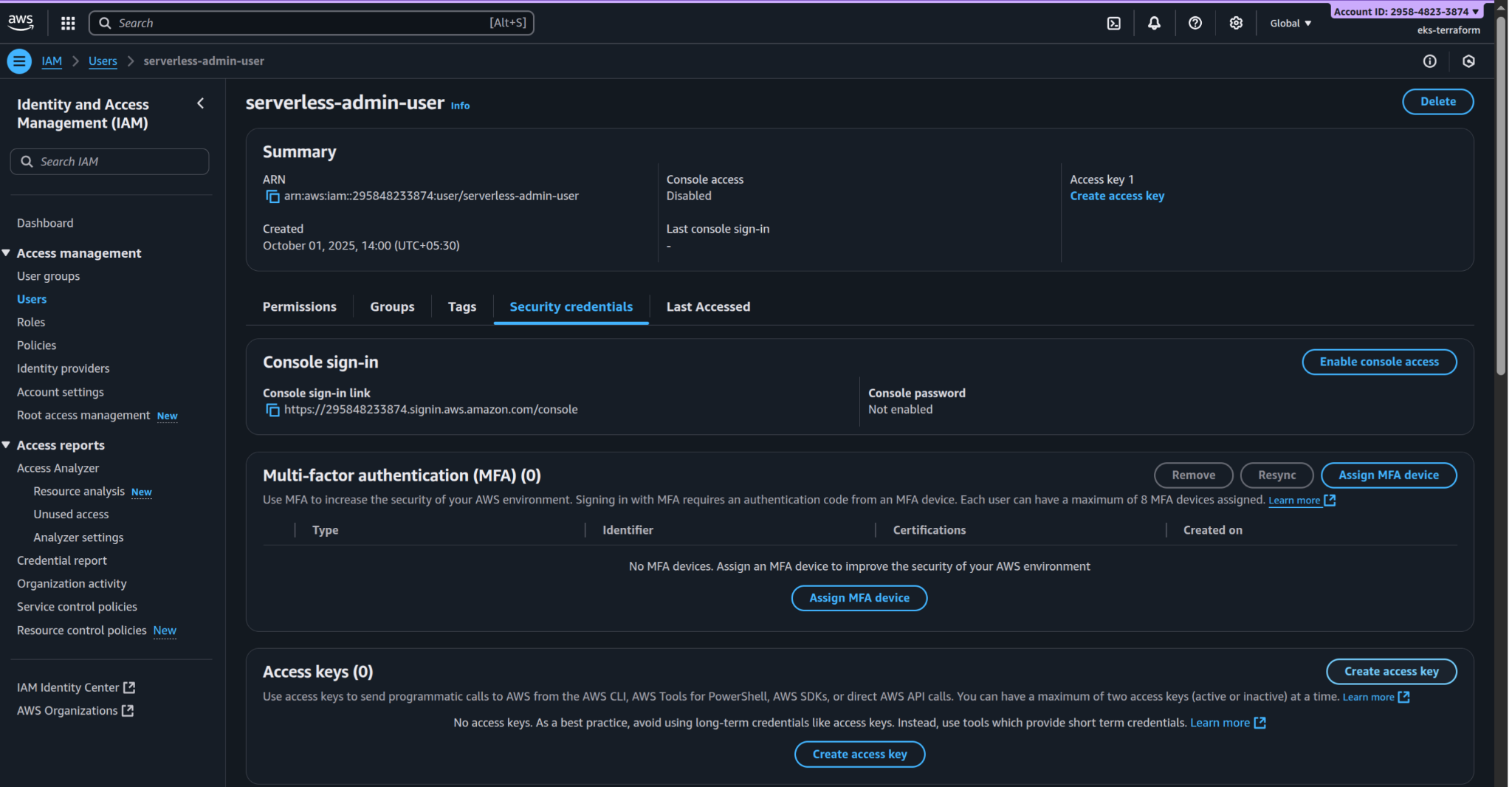
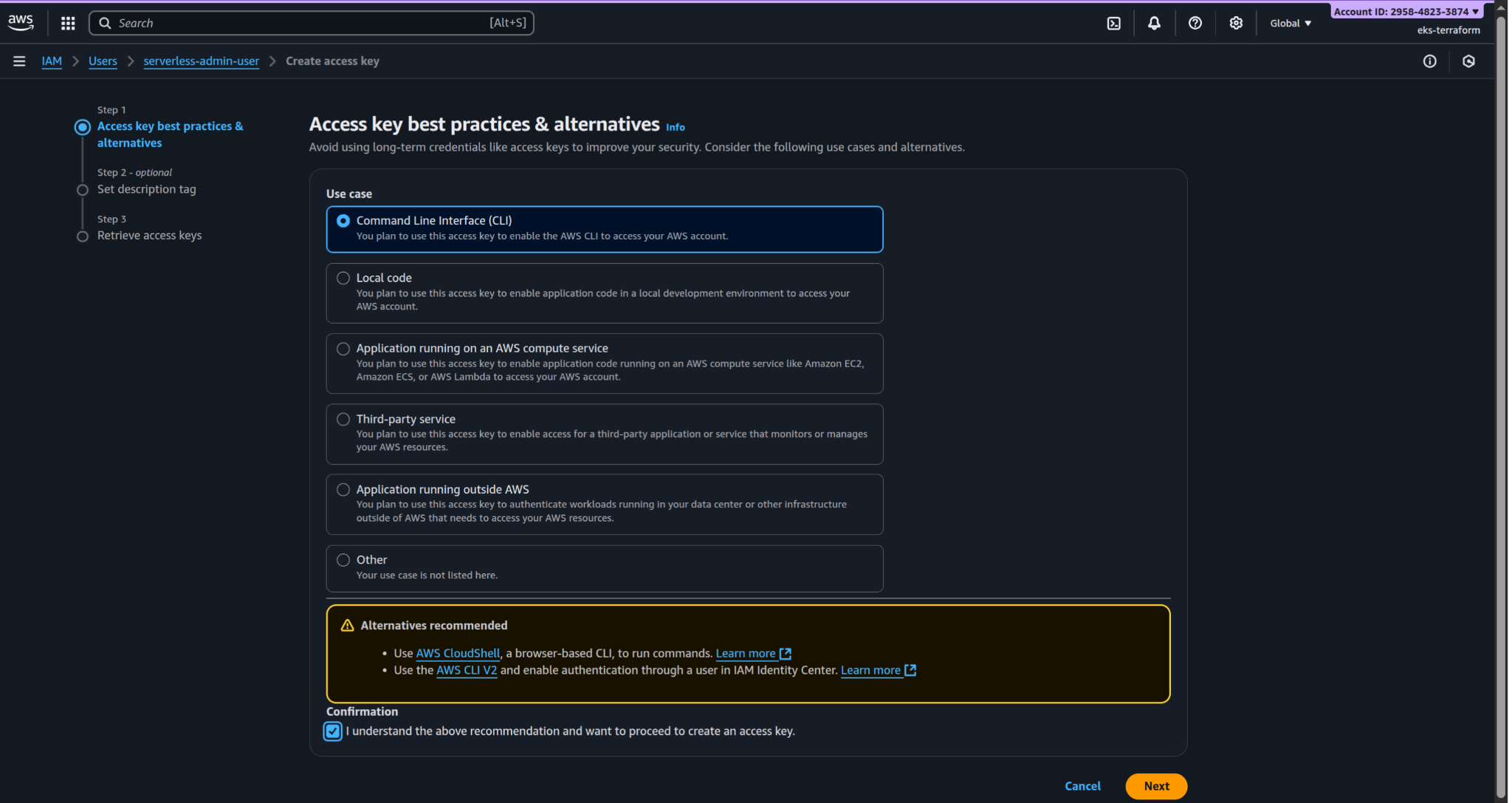
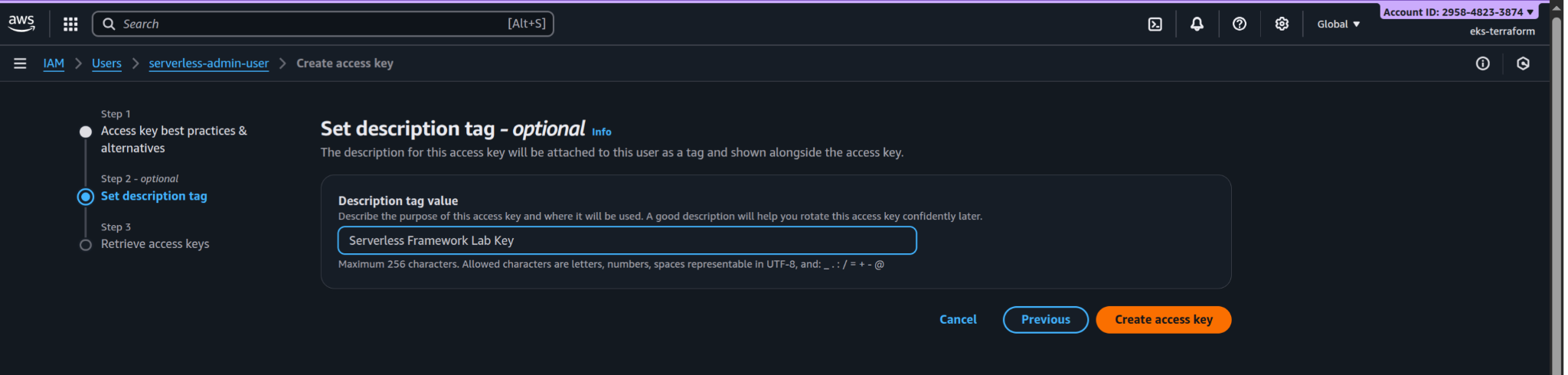
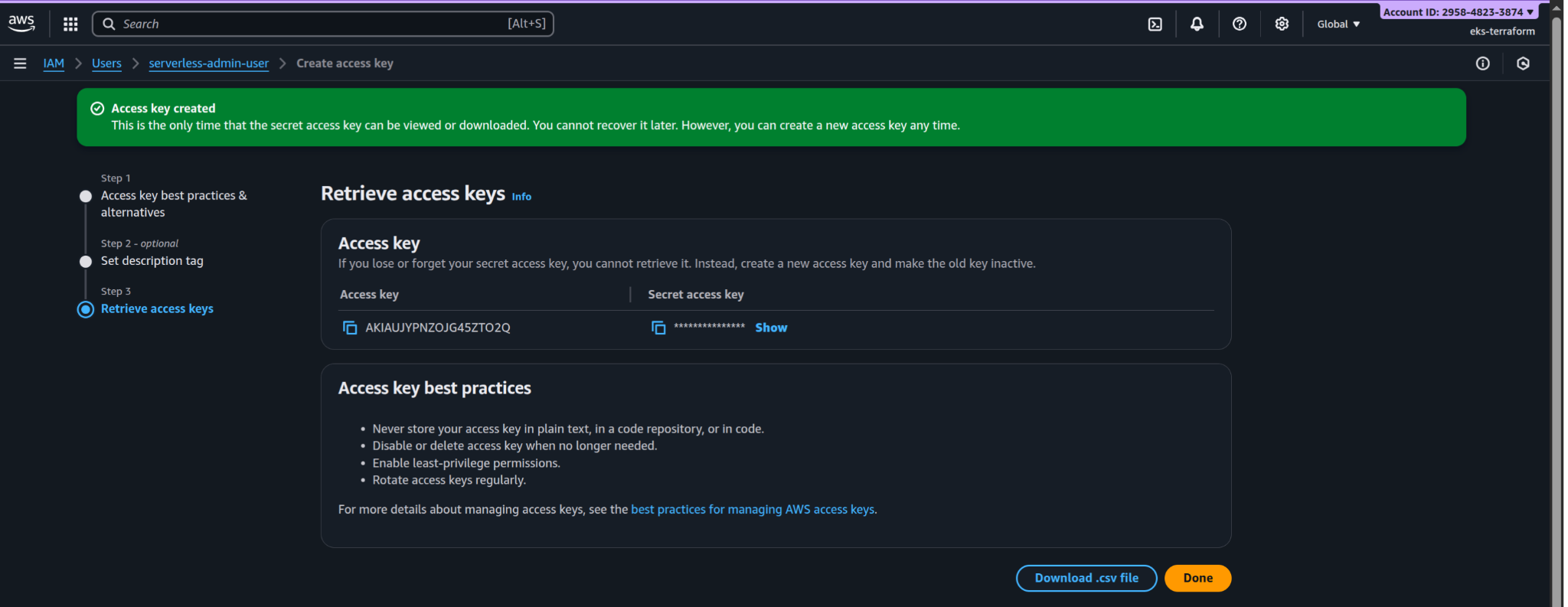
### Step 1: Create an AWS IAM User for Programmatic Access

The Serverless Framework needs programmatic permissions to create and manage resources in your AWS account. We will create a dedicated IAM user for this purpose.

1. **Navigate to the IAM Console:**
   * Sign in to your AWS Management Console.
   * In the search bar at the top, type IAM and select it from the services list.
2. **Create the User:**
   * In the IAM dashboard, click on **Users** in the left navigation pane, then click the **Create user** button.
   * **User name:** Enter a descriptive name, like serverless-admin-user.
   * Click **Next**.
3. **Set Permissions:**
   * Select **Attach policies directly**.
   * In the search box for policies, type AdministratorAccess.
   * Check the box next to the **AdministratorAccess** policy.
   * Click **Next**.



**Note:** For this educational lab, AdministratorAccess provides the simplest path to get started. In a real-world production environment, you should always follow the principle of least privilege and create a role with more restrictive, fine-grained permissions.

1. **Review and Create:**
   * Review the user details and click **Create user**.
2. **Retrieve Your Access Keys:**
   * After the user is created, click on the user's name in the list.
   * Go to the **Security credentials** tab.
   * Scroll down to the **Access keys** section and click **Create access key**.
   * Select **Command Line Interface (CLI)** as the use case.
   * Acknowledge the recommendation and click **Next**.
   * (Optional) Set a description tag, like Serverless Framework Lab Key.
   * Click **Create access key**.
   * **This is the only time you will see the Secret access key.** Copy both the **Access key ID** and the **Secret access key** and save them somewhere secure on your machine. We will need them in the next step.  
       
       
       
     

### Step 2: Install and Configure the AWS CLI

The AWS Command Line Interface (CLI) is a tool that allows you to interact with AWS services from your terminal.

1. **Install the AWS CLI:**
   * Open CLAB terminal.
   * Check if AWS CLI already installed with aws --version.
   * If you don't have it installed, follow the official instructions for your operating system: [Installing the AWS CLI version 2](https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html).
2. **Configure a CLI Profile:**
   * Open your terminal or command prompt.
   * We will create a named profile to keep our lab credentials separate. This is a best practice for managing multiple projects or accounts.
   * Run the following command:

aws configure --profile serverless-lab



* + The CLI will now prompt you for four pieces of information. Use the credentials you saved in the previous step.
    - **AWS Access Key ID:** Paste the Access key ID.
    - **AWS Secret Access Key:** Paste the Secret access key.
    - **Default region name:** Enter a region to work in, for example, ap-south-1.
    - **Default output format:** You can leave this blank or type json.

Your terminal should look something like this:

$ aws configure --profile serverless-lab

AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE

AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY

Default region name [None]: ap-south-1

Default output format [None]: json

# After setup verify

$ aws configure list --profile serverless-lab



Note: Aws saves all the configs in ~/.aws/config and ~/.aws/credentials files.

### Step 3: Install Python

Our guided project will use Python for the AWS Lambda functions, so you'll need a recent version of Python installed on your machine.

1. **Download and Install Python:**
   * Navigate to the official Python downloads page: [python.org/downloads/](https://www.python.org/downloads/).
   * Download and run the installer for a version of Python 3.9 or higher, as these are well-supported by AWS Lambda.
   * Or follow below instructions:

apt install -y python3 python3-pip python3-venv



1. **Best Practice: Python Virtual Environments:**
   * It is highly recommended to use a virtual environment for each of your Python projects to manage dependencies. While not strictly required for this lab to function, it's a critical skill. You can create one for our lab project later, but it's good to know the commands now:

# Create a virtual environment named 'venv' in your project folder

python3 -m venv venv

# Activate the virtual environment

source venv/bin/activate



### Step 4: Install Node.js and nvm

The Serverless Framework is a **Node.js** application, so we need Node.js installed. We'll use nvm (Node Version Manager) to install it, as this tool makes it easy to manage different Node.js versions.

1. **Install nvm:**
   * Open a new CLAB terminal.
   * Check nvn is already installed with nvm -v
   * Run the installation script from the official nvm repository. The exact command can be found here: [nvm GitHub Repository](https://github.com/nvm-sh/nvm). Typically, it's a curl or wget command.
   * After running the script, close and reopen your terminal.
2. **Install and Use Node.js:**
   * Now, use nvm to install the latest Long-Term Support (LTS) version of Node.js:

nvm install --lts



* + Tell nvm to use this version in your current shell:

nvm use --lts



### Step 5: Install the Serverless Framework

With Node.js and its package manager (npm) installed, we can now install the Serverless Framework.

1. **Install via npm:**
   * In your terminal, run the following command to install the framework:

npm install -g serverless



### Step 6: Connect to the Serverless Dashboard

The Serverless Dashboard provides a web interface to monitor, manage, and gain insights into your deployed services.

1. **Log in from the CLI:**
   * Run the login command in your terminal:

serverless login



* + This will automatically open a new tab in your web browser.
  + Choose to register or log in. It's recommended to sign up with GitHub for ease.
  + Once you've authenticated, you can return to your terminal.

### ✅ Step 7: Verification

Let's quickly verify that everything is installed and configured correctly.

1. **Check AWS CLI Configuration:**

aws sts get-caller-identity --profile serverless-lab



This command should return the UserId, Account, and Arn of the IAM user you created, confirming your credentials are correct.

**UserId** → The unique identifier for the IAM user or assumed role.

**Account** → The AWS account ID you’re operating under.

**Arn** → The Amazon Resource Name of the caller (could be a user, assumed role, etc.).

**STS** → AWS Security Token Service. It’s an AWS service that issues temporary, limited-privilege credentials for IAM users

**get-caller-identity** → a special STS API call that simply returns details about *who you are authenticated as* when making the call.

1. **Check Python Version:**

# On macOS, Linux, or WSL

python3 --version



This should show a version of 3.9 or higher.

1. **Check Node.js and npm Versions:**

node -v

npm -v



This should output the versions of Node.js and npm.

1. **Check Serverless Framework Version:**

serverless --version



This will display the framework version and confirm it's installed correctly.

Your environment is now fully configured and ready for building!

═══ END OF DOCUMENT ═══

2️⃣ Activity 2: Core Concepts

# Activity2 — Serverless Framework v4 Core Concepts 🧠

## Objective🎯

* Explain what the Serverless Framework v4 does and why we use it.
* Describe the main **keywords** used in serverless.yml (service, function, event, provider, resources, stages, variables, outputs, deployment bucket, artifacts) and when to use them.
* Run and interpret core Serverless **commands** (deploy, package, invoke, invoke local, logs, info, remove).
* Use essential plugins (install + configure) and run basic local development flows.
* Inspect the CloudFormation template that Serverless generates and know where artifacts are stored.

## Prerequisites (one-time per activity)

You must have completed Activity1. Before starting this activity, run:

aws configure --profile serverless-lab # set credentials for profile 'serverless-lab'

serverless login # link CLI with your Serverless.org account (dashboard)



Note: these two steps are required **once per activity** so Serverless can use your AWS profile and (optionally) the Serverless Dashboard.

## Lab setup (quick)

Create a small demo workspace we’ll use for hands-on snippets:

mkdir -p ~/serverless\_lab/Activity2\_demo

cd ~/serverless\_lab/Activity2\_demo



You can remove this folder later.

## 1) Introduction — what Serverless Framework is

**Serverless (idea)**: a way to build apps where you do not manage servers — AWS runs the compute (e.g., Lambda), you pay per-use.

**Serverless Framework (tool)**: a command-line tool that helps you describe serverless apps in one file (serverless.yml) and then deploy them as AWS (CloudFormation) stacks. Think of it like: *write a short recipe* → *the tool produces the long CloudFormation recipe and tells AWS to build everything for you*.

**Use cases**: small REST APIs, cron jobs, event-driven pipelines (S3 → Lambda), backends for single page apps, simple data processing.

## 2) Supported providers

Serverless Framework has plugins for multiple clouds (AWS, Azure, GCP, Cloudflare Workers, etc.). In this lab we focus on **AWS**.

## 3) Why use Serverless Framework

* You write a small config file instead of huge CloudFormation templates -> basically Infrastructure as Code (**IaC**).
* Framework auto-creates IAM roles and wiring for you (but you can override them).
* Easily manage multiple stages (dev/prod) with one repo.
* Big plugin ecosystem for packaging, local testing, canary releases.
* Works well with CI/CD pipelines.

## 4) Runtimes

Serverless supports many runtimes (Python, Node, Java, Go). We pick **python3.10** for all function code in this lab. That means your Lambda handlers are .py files and the runtime field in serverless.yml will be python3.10.

## 5) AWS services you can define with Serverless

* **Lambda** — the function code (compute). Use for request handlers or event processors.
* **API Gateway** — HTTP endpoints fronting Lambdas. Use for REST/HTTP APIs.
* **DynamoDB** — NoSQL table for storing items (session, user data).
* **S3** — object storage; trigger a Lambda on an upload.
* **SQS** — queues for decoupling and async processing.
* **SNS** — pub/sub fanout.
* **CloudWatch** — logs, metrics, alarms.
* **Step Functions** — orchestrate long-running workflows. Serverless expresses all of these either via functions (with events) or via resources.

## 6) Core keywords in serverless.yml — definition + example + use case

Below each keyword is explained in simple terms and followed by a minimal YAML snippet.

### **service**

**Meaning:** the name of your whole project / unit of deploy.

**Use:** organizes and names the CloudFormation stack.

service: todo-service



**Use case:** pick service as a short, unique name that describes this application.

### **provider**

**Meaning:** global settings for cloud provider (AWS), runtime, region, account details.

**Use:** set defaults used by all functions (runtime, region, IAM role behavior).

provider:

name: aws

runtime: python3.10

region: ap-south-1

profile: serverless-lab



**Important fields:**

* name — cloud provider (aws).
* runtime — default runtime for functions.
* region — AWS region for deployment.
* profile — AWS CLI profile to use.

### **functions**

**Meaning:** all Lambda functions; each entry defines the **handler**, **events**, and optional overrides.

**Use:** declare function code and triggers.

functions:

createTodo:

handler: handlers.api.create\_todo # file handlers/api.py function create\_todo

events:

- httpApi:

path: /todos

method: post



**Use case:** define REST endpoints, S3 triggers, SQS listeners etc. Each function maps to one Lambda in AWS.

### **handler** (within a function)

**Meaning:** the entry point to your function; format file.func (Python).

**Use case:** handlers.api.create\_todo means open handlers/api.py and call def create\_todo(event, context).

### **events**

**Meaning:** the triggers that cause a function to run (HTTP request, S3 event).

**Example events (brief):**

* httpApi / http → API Gateway routes.
* s3 → bucket event (upload).
* sqs → queue message.
* sns → topic message.
* schedule → cron-like scheduled events.

Example:

events:

- s3:

bucket: my-upload-bucket

event: s3:ObjectCreated:\*



### **resources**

**Meaning:** direct CloudFormation block; create anything CFN supports (DynamoDB table, S3 bucket).

**Use:** when you need extra infra that Serverless does not add automatically.

resources:

Resources:

TodoTable:

Type: AWS::DynamoDB::Table

Properties:

TableName: TodoTable

AttributeDefinitions: ...



**Note:**

In AWS, CFN is an abbreviation for AWS CloudFormation, a service that allows you to model and provision AWS resources by defining them in a template written in YAML or JSON.

resources are raw CloudFormation—be comfortable reading CFN syntax.

### **stages**

**Meaning:** named deployment environments (e.g., dev, prod).

**Use:** you deploy the same service under different stages to isolate them.

**Command usage**:

service: my-service

provider:

name: aws

runtime: python3.10

stage: ${opt:stage, 'dev'} # default = dev if no stage passed

region: ap-south-1

functions:

hello:

handler: handler.hello

environment:

TABLE\_NAME: ${self:service}-${self:provider.stage}-table

### How it works

* If you run:

sls deploy --stage dev



→ environment variable TABLE\_NAME = my-service-dev-table

* If you run:

sls deploy --stage prod



→ environment variable TABLE\_NAME = my-service-prod-table

This way, **different stages** get **separate resources** (good isolation between dev and prod).

### variables

**Meaning:** placeholders and lookups used in serverless.yml (self, env, ssm, file).

**Examples & use-cases:**

* ${self:provider.region} → refer to a value in the same file.
* ${env:MY\_VAR} → read an environment variable from your shell.
* ${ssm:/path/to/secret~true} → fetch secure parameter from SSM.
* ${file(./config.${opt:stage}.yml):dbTable} → include values from external file.

**Why use:** keep secrets out of repo, vary config by stage, reuse values.

### **outputs**

**Meaning:** Values that CloudFormation will export or display once the stack is deployed — such as **API endpoints** or **ARNs** of created resources.

**Use:**

* Shown when you run sls info.
* Useful for referencing resources across stacks.

**Note:** An **Amazon Resource Name (ARN)** is a globally unique identifier for an AWS resource (e.g., arn:aws:lambda:ap-south-1:123456789012:function:my-service-dev-hello).

#### Sample YAML (serverless.yml)

service: my-service

provider:

name: aws

runtime: python3.10

region: ap-south-1

functions:

hello:

handler: handler.hello

events:

- httpApi:

path: /hello

method: get

outputs:

ApiEndpoint:

Description: "Base API endpoint"

Value: !Sub "https://${HttpApi}.execute-api.${AWS::Region}.amazonaws.com"

#### Example command

sls info --stage dev

#### Sample output

Service Information

service: my-service

stage: dev

region: ap-south-1

stack: my-service-dev

resources: 10

api keys:

None

endpoints:

GET - https://abc123.execute-api.ap-south-1.amazonaws.com/hello

functions:

hello: my-service-dev-hello

outputs:

ApiEndpoint: https://abc123.execute-api.ap-south-1.amazonaws.com



This shows the **endpoint URL** under outputs, along with function names and other stack info.

### **deploymentBucket**

**Meaning:** The **deployment bucket** is an **S3 bucket** that the Serverless Framework uses internally during deployment.

* Every time you run sls deploy, Serverless packages your Lambda code (and dependencies) into a **zip file (artifact)**.
* That zip is uploaded to this **S3 bucket**.
* CloudFormation then uses that artifact from S3 to actually create/update the Lambda functions.

Without this bucket, AWS would not know where to fetch your function code from.

**Why S3 is used by Serverless:**

1. **Staging area for artifacts** → Lambda requires code to be in S3 before CloudFormation can deploy it.
2. **Version history** → Each deploy uploads a new zip file. You can roll back or inspect old versions.
3. **Scalability** → S3 is highly available, so artifacts are reliably stored during deployment.
4. **Organization** → You can control bucket name, region, encryption, lifecycle rules.
   * Example: auto-expire old artifacts after 7 days to save cost.
   * Example: enforce that artifacts are always in the same region as your Lambdas.

**Sample YAML (**serverless.yml**)**

provider:

name: aws

runtime: python3.10

region: ap-south-1

deploymentBucket:

name: my-serverless-deploy-bucket

blockPublicAccess: true # recommended: no public access

serverSideEncryption: AES256 # encrypt artifacts at rest

maxPreviousDeploymentArtifacts: 5 # keep only last 5 versions



**Hands-on check:**

1. Deploy a service:

sls deploy --stage dev



1. Go to **AWS Console → S3 → my-serverless-deploy-bucket**.
2. You’ll see a folder structure like:

serverless/my-service/dev/1664630509834-2022-10-01T12:15:09/my-service.zip



* + Each folder represents a deployment timestamp.
  + Inside: the zipped Lambda artifact uploaded by Serverless.

In short: the **deploymentBucket is the bridge** between your local machine and AWS CloudFormation. It makes sure AWS has the packaged code to deploy.

### **artifacts**

**Meaning:** the deployment bundles — zip files or container images that are uploaded.

**Where to look:** .serverless/ folder after sls package.

**Use:** CI pipelines publish these artifacts and feed them to CloudFormation.

## 7) Serverless Framework commands — detailed with hands-on examples

Below are the common commands you will use, what each does, and short hands-on steps.

For the examples assume current dir: ~/serverless\_lab/Activity2\_demo

### **sls**

**Meaning:** sls is the short form alias for serverless.

**What it does:** scaffolds a small Python service in demo/.

**Hands-on:**

sls

# Choose template as: AWS Python Simple function

# name project as: demo

cd demo

ls



**Check:** you should see a serverless.yml and a handler file. Open serverless.yml to inspect.

### **app**

**Meaning:** The app keyword in serverless.yml is used to link your service to the **Serverless Dashboard** (on [serverless.com](https://www.serverless.com)).

When you specify an app name, your deployments, metrics, and secrets can be managed in the Dashboard.

**Why use:**

* Centralized monitoring (function invocations, errors).
* Team collaboration (share deployments, CI/CD hooks).
* Secure secrets management.
* Easy rollback and deployment history.

**Sample YAML (**serverless.yml**)**

service: my-service

app: serverless-lab-app # app name in Serverless Dashboard

org: my-org-name # your org in Serverless Dashboard

provider:

name: aws

runtime: python3.10

region: ap-south-1



**Hands-on:**

1. Login to Serverless Dashboard:

serverless login



1. After deployment, open <https://app.serverless.com> → select your org → app → you’ll see:
   * deployed service,
   * stage,
   * endpoints,
   * logs & metrics.

Use app and org when you want **visibility and collaboration** through the Serverless Dashboard. For solo labs, it’s optional, but for real-world projects, it’s strongly recommended.

### **sls package**

**What it does:** packages your code and dependencies but **does not** deploy. It writes artifacts to .serverless/.

**Hands-on:**

sls package

ls -la .serverless

# you will see a zipped artifact and cloudformation template



**Why useful:** debug packaging problems locally; CI often runs sls package first. You can find the final AWS Cloudformation json there.

### **sls deploy**

**What it does:** packages, uploads artifacts to S3, generates a CloudFormation template, and runs CloudFormation to create/update resources.

**Hands-on (small demo):**

sls deploy --stage dev --region ap-south-1 --aws-profile serverless-lab

# watch the output: it lists created resources and endpoints



**After deploy:** note the API endpoint printed and the CloudFormation stack name in AWS Console.

**Common flags:** --stage, --region, --aws-profile (or --profile serverless-lab).

### **sls info**

**What it shows:** a summary of deployed stack: endpoints, functions, stack name, outputs.

**Hands-on:**

sls info --stage dev



**Use-case:** quickly find your API base URL or function ARNs.

### **sls invoke**

**What it does:** calls a deployed Lambda function with a payload.

**Hands-on:**

# sls invoke -f <function-name> --data '{"key":"value"}'

sls invoke -f createTodo --data '{"title":"buy milk"}' --stage dev

# prints function output JSON



**Use-case:** basic smoke test without using API Gateway.

### **sls invoke local**

**What it does:** runs your handler **locally** in your machine’s Python runtime. No AWS involved.

**Hands-on example:**

1. Create a simple handler handlers/api.py:

# handlers/api.py

def hello(event, context):

return {"statusCode": 200, "body": "hello from local"}



1. serverless.yml function:

functions:

hello:

handler: handlers.api.hello



1. Run:

sls invoke local -f hello --data '{}'

# Expected output printed in console



**Use-case:** quick unit-like run, faster iteration.

**Caveat:** local run won't catch **IAM** or **VPC** issues that only appear in AWS.

### **sls logs**

**What it does:** tails CloudWatch logs for a deployed function. **Hands-on:**

# sls logs -f <function> --tail

sls logs -f createTodo --stage dev --tail

# watch logs in real time while invoking API



**Use-case:** debug runtime errors that only appear when function executes in AWS.

### **sls remove**

**What it does:** removes the CloudFormation stack and deletes created resources. **Hands-on:**

sls remove --stage dev --region ap-south-1



**Important:** always run sls remove to avoid lingering costs from resources.

## 8) Plugins — how to install/configure and why

### What are plugins?

* **Plugins** are add-ons for the Serverless Framework.
* They extend or modify the default behavior of deployments (packaging, testing, monitoring, release strategies, etc.).
* Think of them like **“extensions”** in VSCode or Chrome — the core tool works without them, but plugins add powerful features.

### Are they installed by default?

❌ **No**.

* Serverless Framework comes with a **core set of commands** (deploy, invoke, logs, etc.).
* Any **extra capability** (like Python dependency packaging, local API simulation, canary rollouts) requires installing plugins.
* You install them using npm inside your service directory.

### How to install a plugin

Run this inside your project folder:

npm init -y # creates package.json if not already

npm install --save-dev serverless-python-requirements serverless-offline



This will add them to your devDependencies in package.json.

### How to enable a plugin

Add them to the plugins section of your serverless.yml:

plugins:

- serverless-python-requirements

- serverless-offline



### Common Plugins

#### 1. serverless-python-requirements

**Why:**

* By default, Serverless does not know how to package Python dependencies.
* This plugin reads your requirements.txt and bundles dependencies into the Lambda zip.
* It can use Docker (dockerizePip: true) to ensure compatibility with Lambda’s Linux runtime.

**How to configure:**

plugins:

- serverless-python-requirements

custom:

pythonRequirements:

dockerizePip: true # builds deps inside Docker (matches AWS runtime)

zip: true # compress dependencies

slim: true # remove unnecessary files



**Hands-on:**

1. Create requirements.txt with a small package, e.g.:

requests==2.31.0



1. Run deploy:

sls deploy



1. In .serverless/ you’ll see dependencies zipped along with your function.
2. Inside Lambda console → Code → check that requests is included.

#### 2. serverless-offline

**Why:**

* Testing every change by deploying to AWS is slow.
* This plugin lets you run your API Gateway + Lambda **locally** (http://localhost:3000).
* Speeds up development and debugging.

**How to configure:**

plugins:

- serverless-offline



**Hands-on:**

sls offline start



* Now you can hit your endpoint locally:

curl http://localhost:3000/hello



* Useful for frontend teams who want a working API without needing AWS access.

#### 3. serverless-deployment-bucket

**Why:**

* By default, Serverless auto-creates a random-named S3 bucket for artifacts.
* This plugin lets you **control the bucket name and settings** (encryption, lifecycle).
* Useful for compliance, cost control, and centralizing artifacts.

**How to configure:**

plugins:

- serverless-deployment-bucket

provider:

deploymentBucket:

name: my-serverless-deploy-bucket

blockPublicAccess: true

serverSideEncryption: AES256

lifecycleRules:

- expirationInDays: 7 # delete old artifacts after 7 days



**Hands-on:**

1. Deploy once.
2. Check S3 console → you’ll see all artifacts go into my-serverless-deploy-bucket.

#### 4. serverless-plugin-canary-deployments (advanced)

**Why:**

* Sometimes deploying a new version of Lambda directly is risky.
* Canary deployment = send **10% traffic** to new version → if no errors, then send 100%.
* Helps reduce risk of breaking production.

**How to configure:**

plugins:

- serverless-plugin-canary-deployments

functions:

hello:

handler: handler.hello

deploymentSettings:

type: Canary10Percent5Minutes

alias: Live



**Hands-on:**

1. Deploy function with canary settings.
2. Serverless will automatically configure Lambda Aliases to shift 10% traffic to new version.
3. After 5 minutes (if no rollback), 100% of traffic is shifted.

In short: **Plugins are not auto-installed**. You decide which ones you need, install them via npm, and enable in serverless.yml. Each plugin solves a specific problem — packaging, local testing, deployment control, or safe rollouts.

## 9) Local development vs deployed behavior — pitfalls & tips

**Local (fast feedback)**

* sls invoke local executes your code locally with your machine’s environment.
* sls offline simulates HTTP endpoints.

**Deployed (real)**

* AWS imposes IAM, network (VPC), cold-starts, and limits that local can’t simulate.
* Some issues only appear after deploy: missing IAM permissions, incorrect environment variables.

**Tips**

* Use dockerizePip: true in serverless-python-requirements when you depend on compiled libraries (ensures manylinux wheels).
* Use local DynamoDB (amazon/dynamodb-local) when you want a local copy of the DB (**docker** run provided below).
* Test both local and deployed — local for fast iteration, deployed for final verification.

**DynamoDB local quick start**

docker run -p 8000:8000 amazon/dynamodb-local

# in another shell:

aws dynamodb list-tables --endpoint-url http://localhost:8000



## 10) Internal process — what happens during sls deploy (step-by-step)

1. **Parse serverless.yml** and resolve variables.
2. **Build packages**: create zipped artifacts (or container images) per function or per service.
3. **Upload** artifacts to the deployment S3 bucket.
4. **Generate CloudFormation template** (written to .serverless/cloudformation-template-update-stack.json).
5. **Call CloudFormation** to create/update the stack. CloudFormation creates all AWS resources declared.
6. **Return outputs** and print endpoints.

**How to inspect generated template**

sls package

# then:

less .serverless/cloudformation-template-update-stack.json



Reading that file shows you the exact resources CloudFormation will create — helpful for debugging and for understanding what Serverless translates to.

### 12) Variables in serverless.yml — expanded with concrete use examples

Variables let you keep configuration DRY, pull secrets from secure stores, and change behavior per stage/region without editing the YAML. Below are the variable types we use in this lab, each with (A) how it looks in serverless.yml and (B) a short example that **shows the variable being referenced/used**.

#### **${self:}** — refer to values in the same serverless.yml

**Definition (declare a value):**

service: todo-service

provider:

stage: ${opt:stage, 'dev'} # default stage = dev



**Usage (where the resolved value is used):**

resources:

Resources:

TodoTable:

Type: AWS::DynamoDB::Table

Properties:

TableName: ${self:custom.tablePrefix}-table # e.g., todo-service-dev-table



**What happens:** if you run sls deploy --stage prod, ${self:custom.tablePrefix} becomes todo-service-prod.

#### **${env:VAR}** — read a shell environment variable

**Definition (use an env var in YAML):**

provider:

name: aws

runtime: python3.10

functions:

create:

handler: handlers.create.handler

environment:

TODOS\_TABLE: ${env:TODOS\_TABLE} # pulled from your shell env



**Usage (set and run):**

export TODOS\_TABLE="my-todos-dev"

sls deploy --stage dev

# In AWS Lambda, environment variable TODOS\_TABLE will be "my-todos-dev"



**Tip:** Good for local overrides or CI secrets injected by the pipeline.

#### **${ssm**:/path/to/param~true} — fetch secure string from SSM Parameter Store

**Definition (reference an SSM parameter in YAML):**

provider:

name: aws

functions:

worker:

handler: handlers.worker.handler

environment:

DB\_PASSWORD: ${ssm:/myapp/dev/db\_password~true} # ~true = decrypt SecureString



**Usage (how to create and then read):**

# create a SecureString in SSM (one-time, in AWS CLI)

aws ssm put-parameter --name /myapp/dev/db\_password --value "s3cr3t" --type SecureString --overwrite

# then deploy; Serverless will fetch and inject DB\_PASSWORD at deploy/runtime

sls deploy --stage dev



**What happens:** Serverless resolves the SSM value at deploy-time (or at runtime if configured) and injects it into the function env. Use ~true to tell Serverless to decrypt the SecureString.

#### **${file**(./config.${opt:stage}.yml):dbTable} — load from an external file

**External file config.dev.yml:**

dbTable: todo\_table\_dev

apiPrefix: /dev



serverless.yml **using the file (definition + usage):**

provider:

stage: ${opt:stage, 'dev'}

custom:

config: ${file(./config.${opt:stage}.yml)} # loads config.dev.yml for dev

functions:

list:

handler: handlers.list.handler

environment:

TODOS\_TABLE: ${self:custom.config.dbTable} # becomes todo\_table\_dev

events:

- httpApi:

path: ${self:custom.config.apiPrefix}/todos # /dev/todos

method: get



**Usage (switch stage):**

sls deploy --stage dev # loads config.dev.yml

sls deploy --stage prod # would load config.prod.yml if present



**Why:** lets you maintain stage-specific settings in small files (DB names, endpoints, limits).

### Quick summary / rules of thumb

* Use ${self:} for reusing values declared in the same YAML.
* Use ${env:} for per-machine or CI-provided values (secrets can be injected by pipelines).
* Use ${ssm:...~true} for secrets stored in AWS SSM SecureString (requires IAM permission to read).
* Use ${file(...)} to keep large or stage-specific configs outside the main serverless.yml.

## 12) Where to look for more info and how to debug

* Generated CloudFormation: .serverless/cloudformation-template-update-stack.json — shows what will be applied.
* CloudFormation Console → **change sets** / **events** — useful when deploy fails (errors reported here).
* **CloudWatch** Logs for function traces.
* sls logs -f <fn> and AWS Console for detailed debugging.
* sls package + inspect artifact to ensure all files are included.

## 13) Notes, best practices & small cautions (simple language)

* **Never** commit ~/.aws/credentials or secrets into Git. Use environment variables or SSM/Secrets Manager.
* Use serverless remove when done to avoid ongoing costs.
* Prefer **least-privilege IAM** for deploy accounts — don’t use AdministratorAccess for final tests.
* Use serverless-python-requirements with dockerizePip: true when your Python packages include native binaries.
* Local tests are helpful but always verify in AWS because of IAM/VPC differences.
* Keep function package sizes small to reduce cold starts. Consider Layers for shared libraries.

## Final note — evaluation

There is no evaluation for Activity2.

This activity is purely for learning so you are prepared for Activity3 (hand-held project).

Make sure you can run and explain the commands above — that knowledge is required for later hands-on work. Do attempt the quiz.

═══ END OF DOCUMENT ═══

3️⃣ Activity 3: Guided Project

# **Activity3 — Building a Serverless Application (Hand-held)** 🛠️

## **Objective 🎯**

In this activity, you will **build a complete serverless application step by step** using the Serverless Framework v4 on AWS.

By the end of this activity, you should be able to:

* Initialize a Serverless project with Python runtime.
* Create and deploy Lambda functions triggered by API Gateway.
* Define and use DynamoDB, SQS, SNS, and CloudWatch with Lambda.
* Attach IAM roles and permissions properly.
* Inspect the CloudFormation stack generated by Serverless.
* Clean up AWS resources safely.

## **Prerequisites 📝**

* Must have completed **Activity1 (Environment Setup)** and **Activity2 (Core Concepts)**.
* AWS CLI configured:

aws configure --profile serverless-lab



* Logged in to Serverless Dashboard:

serverless login



⚠️ **Note:** AWS CLI and Serverless Dashboard configuration are a **one-time job per activity**. You must do it once at the start of each new activity.

### **Expected Folder Structure**

Your project should look like this as we build it step by step:

activity3-app/

├── handlers/

│ ├── hello.py

│ ├── create\_todo.py

│ ├── publish\_message.py

│ └── consume\_message.py

├── requirements.txt # (optional, if using dependencies like boto3, requests etc.)

├── serverless.yml

└── .gitignore



## **Lab Setup ⚙️**

1. Initialize a new service:

sls



* + Choose template: **AWS - Python - Simple Function**
  + Name project: **activity3-app**
  + Choose Create new app: **serverless-app**
  + **Skip & Set Later (AWS SSO, ENV Vars)**

1. Move into the project folder:

cd activity3-app

ls



You should see serverless.yml and a sample handler.py.

## **Step 1 — Initialize project and configure provider**

Edit serverless.yml and set up:

service: activity3-app # Name of the service (used in stack/resource naming)

provider:

name: aws # Cloud provider (AWS in this case)

runtime: python3.10 # Lambda runtime environment

region: ap-south-1 # Default AWS region for deployment

stage: ${opt:stage, 'dev'} # Deployment stage (defaults to 'dev' if not passed via --stage)

profile: serverless-lab # AWS CLI profile to use for credentials



This ensures we are always deploying to ap-south-1 using our lab IAM profile.

Here’s a more descriptive write-up you can use for your **Step 2 — IAM Role setup** section. I’ve kept the same YAML, but added explanation so it’s clear what each block is doing and why:

### Step 2 — IAM Role setup

In AWS, every Lambda function needs an **execution role** (an IAM role that the function assumes at runtime). This role defines what actions the function is allowed to perform on AWS resources. By customizing the IAM role in serverless.yml, we can enforce **least-privilege access** — each function only gets the exact permissions it needs, nothing more.

Add the following under your provider: block:

provider:

iam:

role:

statements:

# ✅ CloudWatch Logs permissions

- Effect: Allow

Action:

- logs:CreateLogGroup

- logs:CreateLogStream

- logs:PutLogEvents

Resource: "arn:aws:logs:${self:provider.region}:\*:log-group:/aws/lambda/${self:service}-${self:provider.stage}-\*:\*"

# ✅ DynamoDB access (for TodoTable operations)

- Effect: Allow

Action:

- dynamodb:PutItem

- dynamodb:GetItem

- dynamodb:Scan

Resource:

- arn:aws:dynamodb:ap-south-1:\*:table/TodoTable

# ✅ SQS access (send & receive messages)

- Effect: Allow

Action:

- sqs:SendMessage

- sqs:ReceiveMessage

Resource: "\*"

# ✅ SNS publish access (so Lambda can push to our topic)

- Effect: Allow

Action:

- sns:Publish

Resource:

- Ref: MyTopic



### 🔎 Breakdown

* **CloudWatch Logs** Every Lambda writes logs by default. Without these permissions, your functions won’t be able to create log groups/streams or send log events. This block scopes access tightly to only the Lambda log groups of your service and stage.
* **DynamoDB** Only allows basic read/write operations (PutItem, GetItem, Scan) on the specific TodoTable. This prevents accidental access to other tables.
* **SQS** Allows sending and receiving messages. The resource is set to "\*" here, but ideally, you’d restrict this to the ARN of your own queue (e.g., !GetAtt MyQueue.Arn).
* **SNS** Grants sns:Publish specifically to the topic you created (MyTopic). Using Ref: MyTopic ensures CloudFormation injects the correct ARN at deploy time.

### ✅ Why this matters

* Without these, you’ll see runtime errors like AccessDenied when your function tries to log, insert into DynamoDB, or publish to SNS.
* By scoping each permission to just what’s required (least privilege), you reduce risk if your function is compromised.
* Defining IAM in serverless.yml keeps everything **as code** — easy to review and audit.

## **Step 3 — First Lambda function (Hello World)**

Create handlers/hello.py:

def handler(event, context):

return {

"statusCode": 200,

"body": "Hello from Activity3 app!"

}



Add it in serverless.yml:

functions:

hello: # Logical name of the function

handler: handlers/hello.handler # File + function to execute (handlers/hello.py → handler())

events: # Triggers for this Lambda

- httpApi: # HTTP API Gateway event

path: /hello # Endpoint path

method: get # HTTP method



Deploy:

sls deploy --stage dev



Test endpoint:

curl https://<api-id>.execute-api.ap-south-1.amazonaws.com/hello



### **After Step 3 — Verify Deployment & Check Resources**

Before moving to DynamoDB integration, let’s pause and **verify that everything created so far (till Step 3)** is working correctly.

#### 1) If Deployment Fails — How to Debug

If sls deploy did not finish successfully or curl fails:

1. Run the logs command for your function:

sls logs -f hello --stage dev --tail



* + This will stream the CloudWatch logs for your hello function.
  + Check for Python errors, missing handler issues, or permission problems.

1. If resources failed to create, go to **AWS Console → CloudFormation → Stacks → activity3-app-dev → Events**.
   * Look for **CREATE\_FAILED** messages.
   * The error message will indicate what went wrong (e.g., IAM permission denied, bucket already exists).

Only proceed once your stack shows status CREATE\_COMPLETE.

#### 2) Open AWS Console & Set Region

1. Go to [AWS Console](https://console.aws.amazon.com/).
2. In the top-right region selector, choose **Asia Pacific (Mumbai) — ap-south-1**.

#### 3) Inspect CloudFormation Stack

1. Services → **CloudFormation → Stacks**.
2. Find the stack **activity3-app-dev**.
3. Confirm status is **CREATE\_COMPLETE**.
4. Click on it → **Resources** tab. You should see:
   * AWS::Lambda::Function → hello function created.
   * AWS::ApiGatewayV2::Api → the HTTP API created.
   * AWS::ApiGatewayV2::Stage → stage (dev) created.
   * AWS::IAM::Role → execution role for the function.
   * AWS::LogGroup → CloudWatch log group for the function.

#### 4) Inspect Lambda Function

1. Services → **Lambda → Functions**.
2. Verify a function named like activity3-app-dev-hello exists.
3. Check:
   * Runtime = python3.10
   * Handler = handlers/hello.handler
   * Permissions tab → linked IAM role

#### 5) Inspect API Gateway (HTTP API)

1. Services → **API Gateway → HTTP APIs**.
2. Locate the API with name like activity3-app-dev.
3. Verify route:
   * GET /hello is present.
4. Copy **Invoke URL** and test with curl:

curl https://<api-id>.execute-api.ap-south-1.amazonaws.com/hello



#### 6) Inspect CloudWatch Logs

1. Services → **CloudWatch → Logs → Log groups**.
2. Find log group /aws/lambda/activity3-app-dev-hello.
3. Check the latest log stream for request logs (after your curl test).

## **Step 4 — Add DynamoDB integration**

1. Add DynamoDB table under resources:

resources:

Resources:

TodoTable: # Logical resource name

Type: AWS::DynamoDB::Table # Resource type (DynamoDB table)

Properties:

TableName: TodoTable # Actual table name in AWS

BillingMode: PAY\_PER\_REQUEST # On-demand billing (no capacity planning)

AttributeDefinitions: # Define attributes and their types

- AttributeName: id

AttributeType: S # 'S' = String

KeySchema: # Define primary key

- AttributeName: id

KeyType: HASH # Partition key (no sort key here)



1. Create function handlers/create\_todo.py:

import json, uuid, boto3

dynamodb = boto3.resource("dynamodb")

table = dynamodb.Table("TodoTable")

def handler(event, context):

body = json.loads(event["body"])

item = {"id": str(uuid.uuid4()), "task": body["task"]}

table.put\_item(Item=item)

return {"statusCode": 200, "body": json.dumps(item)}



1. Add function in serverless.yml:

functions:

createTodo:

handler: handlers/create\_todo.handler

events:

- httpApi:

path: /todos

method: post



Deploy and test:

curl -X POST https://<api-id>.execute-api.ap-south-1.amazonaws.com/todos \

-H "Content-Type: application/json" \

-d '{"task": "Finish Activity3"}'



Check **DynamoDB** table in AWS Console → **Explore** **Items**.

## **Step 5 — SQS & SNS:**

Before wiring services, first understand what they are and how they work.

### Conceptual overview

* **SNS (Simple Notification Service)**
  + Pub/Sub (publish–subscribe) messaging service.
  + A *publisher* sends a message to an **SNS topic**.
  + The topic *fans out* the message to all **subscribers** (HTTP endpoint, Lambda, SQS queue, email, SMS).
  + Use case: broadcast an event to multiple consumers.
* **SQS (Simple Queue Service)**
  + Durable message queue.
  + Producers send messages to a **queue**. Consumers poll the queue and process messages.
  + Guarantees at-least-once delivery (so make consumers idempotent).
  + Use case: decouple processing, buffer bursts, retry/backoff.
* **Common pattern (SNS → SQS → Lambda)**
  1. API or producer publishes to **SNS topic**.
  2. SNS forwards message to an **SQS queue** (subscription).
  3. A Lambda function is triggered by messages arriving in the SQS queue and processes them (scales independently).
  + This pattern provides fanout + durable, retryable processing.

### Integration — what we will create

* An **SNS topic** (MyTopic).
* An **SQS queue** (MyQueue).
* A CloudFormation **Subscription** that subscribes MyQueue to MyTopic (**SQS <- SNS**).
* A **publishMessage** HTTP endpoint which publishes to the SNS topic.
* A **consumeMessage** Lambda that is triggered by messages arriving in the SQS queue.

### 1) Add resources (serverless resources:)

Add these to serverless.yml under resources: Resources: (we include the subscription so SNS forwards to SQS and we expose the TopicArn via Outputs):

Resources:

MyQueue:

Type: AWS::SQS::Queue

Properties:

QueueName: my-queue

MyTopic:

Type: AWS::SNS::Topic

Properties:

TopicName: my-topic

MyTopicSubscription:

Type: AWS::SNS::Subscription

Properties:

Protocol: sqs

TopicArn:

Ref: MyTopic

Endpoint:

Fn::GetAtt:

- MyQueue

- Arn

# Allow SNS to send messages to the queue (policy attached to the queue)

MyQueuePolicy:

Type: AWS::SQS::QueuePolicy

Properties:

Queues:

- Ref: MyQueue

PolicyDocument:

Version: "2012-10-17"

Statement:

- Sid: Allow-SNS-SendMessage

Effect: Allow

Principal: "\*"

Action: "sqs:SendMessage"

Resource:

Fn::GetAtt:

- MyQueue

- Arn

Condition:

ArnEquals:

"aws:SourceArn":

Ref: MyTopic



**Why include MyQueuePolicy?** SNS must be allowed to send messages to the SQS queue; the queue policy by default restricts senders to send messages to its queue.

#### Broken into sub-parts and explained in detail:

#### A — MyQueue (SQS queue)

* **Purpose:** durable queue that buffers messages for asynchronous processing.
* **CFN Type:** AWS::SQS::Queue.
* **Properties explained:**
  + QueueName: friendly name. If omitted, CloudFormation generates a unique name (often recommended to avoid collisions).
* **Best-practices / enhancements to add in real projects:**
  + VisibilityTimeout — how long a message is hidden while being processed. Tune based on expected processing time.
  + MessageRetentionPeriod — how long messages persist (default 4 days). Lower to save cost if appropriate.
  + ReceiveMessageWaitTimeSeconds — long polling (recommended >0 to reduce empty receive calls).
  + RedrivePolicy — configure a Dead-Letter Queue (DLQ) to handle poison messages. Example:

RedrivePolicy:

deadLetterTargetArn: !GetAtt MyDeadLetterQueue.Arn

maxReceiveCount: 5



* + KmsMasterKeyId or SqsManagedSseEnabled — enable encryption at rest if needed.
* **Verification:** after deploy, check SQS console → queue attributes, note Queue URL and ARN.
* **Pitfalls:** hardcoding QueueName may cause create failures if name already exists in account/region. Consider using ${self:service}-${self:provider.stage}-my-queue pattern.

#### B — MyTopic (SNS topic)

* **Purpose:** pub/sub topic for broadcasting messages to one or many subscribers (HTTP, Lambda, SQS, email, SMS).
* **CFN Type:** AWS::SNS::Topic.
* **Properties explained:**
  + TopicName: friendly name for the topic. Like SQS, leaving it out lets CloudFormation generate a safe unique name.
* **Best-practices / enhancements:**
  + DisplayName for SMS-friendly name if sending SMS.
  + Subscription block can be declared inline here instead of separate AWS::SNS::Subscription.
  + Enable server-side encryption (use KMS) if messages contain sensitive data.
* **Verification:** Console → SNS → Topics → confirm Topic ARN.
* **Pitfalls:** topic names may collide across account/region if hardcoded — prefer stage-scoped names.

#### C — MyTopicSubscription (SNS → SQS subscription wiring)

* **Purpose:** connects the SNS topic to the SQS queue so that messages published to the topic are delivered to the queue. This enables durable fanout.
* **CFN Type:** AWS::SNS::Subscription.
* **Properties explained:**
  + Protocol: sqs — subscription protocol type (other types: lambda, http, https, email, etc.).
  + TopicArn: Ref: MyTopic — Ref returns the topic's logical reference (Topic ARN for AWS::SNS::Topic).
  + Endpoint: Fn::GetAtt: [MyQueue, Arn] — Fn::GetAtt gets the queue ARN (SQS requires ARN as endpoint for sqs protocol).
* **Behavior:** when subscription is created, SNS will attempt to deliver messages to the SQS endpoint. For SQS endpoints, no confirmation handshake is required (unlike HTTP/email).
* **Verification:** CloudFormation Resources should show MyTopicSubscription. In SNS console → Subscriptions you should see a subscription with protocol sqs and endpoint equal to SQS queue ARN.
* **Pitfalls:** subscription alone is not enough — SQS must allow SNS to send messages (see MyQueuePolicy); otherwise deliveries will be blocked.

#### D — MyQueuePolicy (SQS resource policy allowing SNS to send)

* **Purpose:** allow the SNS topic principal to call sqs:SendMessage on the SQS queue. Without this, SNS cannot deliver messages to SQS even if subscription exists.
* **CFN Type:** AWS::SQS::QueuePolicy.
* **Properties explained:**
  + Queues: - Ref: MyQueue — targets the physical queue(s) this policy applies to (you can attach same policy to multiple queues).
  + PolicyDocument — standard IAM policy document applied to the queue resource:
    - Version: policy version.
    - Statement: list of permission statements.
      * Sid: statement id (optional label).
      * Effect: Allow — allow the action.
      * Principal: "\*" — allows any principal, but the Condition below restricts who can actually send.
      * Action: sqs:SendMessage — only allow sending messages.
      * Resource: Fn::GetAtt: [MyQueue, Arn] — the ARN of the queue.
      * Condition: ArnEquals: "aws:SourceArn": Ref: MyTopic — restrict allowed sender to messages whose SourceArn equals this specific SNS Topic ARN.
* **Why this pattern:** SNS publishes to SQS on behalf of SNS’s service principal; the queue policy must explicitly permit that service principal when aws:SourceArn equals the topic ARN — this prevents other SNS topics or principals from sending messages to your queue.
* **Verification:** After deploy, in SQS console → Queue → Access Policy you should see a JSON policy that includes the Allow-SNS-SendMessage statement with the correct aws:SourceArn.
* **Pitfalls & security notes:**
  + Do **not** leave overly-broad policies (e.g., allow all sqs:SendMessage from Principal: "\*" without Condition), as this can let other accounts or services inject messages.
  + Ensure Condition matches the exact Ref: MyTopic (it resolves to the topic ARN).
  + If you have cross-account SNS topics, you may need a different policy allowing specific AWS account principals.

#### E — How these pieces interact (flow)

1. **Producer** publishes to MyTopic (SNS).
2. **SNS** looks at its subscriptions: it has a subscription where Protocol=sqs and Endpoint=MyQueue.Arn.
3. **SNS** delivers the message to the SQS queue by calling sqs:SendMessage.
4. **SQS** receives the message and stores it until a consumer (Lambda triggered by SQS) retrieves it.
5. **Queue policy** ensures only the configured SNS Topic (and no other source) can send messages.

#### F — Additional recommended production considerations

* **Dead-Letter Queue (DLQ):** define a separate SQS queue as DLQ and attach via RedrivePolicy to handle poison messages.
* **Encryption:** enable KMS for both SNS and SQS if sensitive data is involved.
* **FIFO queues:** if ordering is important use FifoQueue: true and topic-to-FIFO subscription rules (requires .fifo naming and MessageGroupId).
* **Monitoring:** enable CloudWatch alarms for SQS ApproximateAgeOfOldestMessage and SNS delivery failures.
* **Name collisions:** use stage/service prefix to avoid collisions: !Sub "${AWS::StackName}-my-queue".

#### G — Quick troubleshooting checklist if messages don’t arrive

* Confirm MyTopicSubscription exists in CloudFormation Resources.
* Check SQS queue policy for Allow-SNS-SendMessage with aws:SourceArn equal to topic ARN.
* In SNS console, view topic metrics for DeliveryFailures.
* Check CloudWatch logs for Lambda consumer errors.
* Use aws sns publish CLI to send test message and aws sqs receive-message to poll queue manually.

### 2) Publish function — handlers/publish\_message.py

Use the Topic ARN from CloudFormation outputs or resolve via environment variable. Prefer injecting the Topic ARN as an environment variable in serverless.yml (shown below).

handlers/publish\_message.py:

import json, boto3, os

sns = boto3.client("sns")

TOPIC\_ARN = os.environ.get("MY\_TOPIC\_ARN") # injected by serverless.yml

def handler(event, context):

body = json.loads(event.get("body", "{}"))

message = body.get("message", "hello from publishMessage")

resp = sns.publish(TopicArn=TOPIC\_ARN, Message=message)

return {"statusCode": 200, "body": json.dumps({"MessageId": resp.get("MessageId")})}



### 3) Consumer function — handlers/consume\_message.py

This function receives SQS event records (Lambda event format for SQS):

import json

def handler(event, context):

for record in event["Records"]:

# record['body'] contains the message published to SNS (string)

print("Received SQS message:", record["body"])

# TODO: parse/process the message and apply idempotency if needed

return {"statusCode": 200, "body": "Processed messages"}



**Note:** Processing must be idempotent because SQS + Lambda can deliver messages more than once.

### 4) Update functions: in serverless.yml (wiring + env var)

Add both functions, inject the Topic ARN into the publisher, and wire the SQS event for the consumer:

functions:

publishMessage:

handler: handlers/publish\_message.handler

environment:

MY\_TOPIC\_ARN:

Ref: MyTopic

events:

- httpApi:

path: /publish

method: post

consumeMessage:

handler: handlers/consume\_message.handler

events:

- sqs:

arn:

Fn::GetAtt: [MyQueue, Arn]

batchSize: 5 # process up to 5 messages per Lambda invocation

maximumBatchingWindow: 30 # seconds; optional

enabled: true



### 5) Outputs (optional — expose TopicArn and QueueArn)

Add to outputs: under resources: so you can easily see ARNs with sls info --stage dev --verbose:

Outputs:

MyTopicArn:

Description: "SNS Topic ARN"

Value: !Ref MyTopic

# Optional: export it for cross-stack usage

Export:

Name: ${self:service}-${self:provider.stage}-MyTopicArn

MyQueueArn:

Description: "SQS Queue ARN"

Value: !GetAtt MyQueue.Arn

Export:

Name: ${self:service}-${self:provider.stage}-MyQueueArn



### 6) Deploy

Deploy the updated service (packs, uploads, CloudFormation):

sls deploy --stage dev



Watch the output for created resources — confirm MyTopic, MyQueue, the subscription, and functions.

### 7) Test the flow (end-to-end)

1. **Publish via HTTP endpoint**:

curl -X POST https://<api-id>.execute-api.ap-south-1.amazonaws.com/publish \

-H "Content-Type: application/json" \

-d '{"message": "hello world"}'



Response should contain MessageId.

1. **Verify SQS received message**:
   * Console: SQS → my-queue → Monitoring / Messages available (may be zero if Lambda immediately consumed).
   * Or poll the queue manually (CLI) to peek:

aws sqs receive-message --queue-url $(aws sqs get-queue-url --queue-name my-queue --query QueueUrl --output text) --max-number-of-messages 1 --visibility-timeout 0



* + If Lambda consumes, check **CloudWatch Logs** for consumeMessage to see **printed messages**:

sls logs -f consumeMessage --stage dev --tail



1. **If consumer didn’t process** (messages remain), ensure:
   * consumeMessage Lambda exists and its event mapping to SQS is present.
   * IAM role has necessary sqs:ReceiveMessage/sqs:DeleteMessage permissions (we set Send/Receive earlier; refine as needed).

### 8) Troubleshooting tips

* **Subscription not visible**: check MyTopicSubscription in CloudFormation Resources and SQS queue policy.
* **Messages not delivered**: confirm MyQueuePolicy allows sqs:SendMessage from the SNS Topic ARN.
* **Lambda errors**: stream logs with sls logs -f consumeMessage --tail and inspect stack Events in CloudFormation for deployment failures.
* **Idempotency**: design consumeMessage to detect and ignore duplicate processing (store processed IDs in DynamoDB if needed).

### 9) Clean up (if you want to remove the entire stack)

sls remove --stage dev



## **Final serverless.yml (Full Stack)**

# "org" ensures this Service is used with the correct Serverless Framework Access Key.

org: sammagnet7

# "app" enables Serverless Framework Dashboard features and sharing them with other Services.

app: serverless-app

# "service" is the name of this project. This will also be added to your AWS resource names.

service: activity3-app

provider:

name: aws

runtime: python3.10

region: ap-south-1

stage: ${opt:stage, 'dev'}

profile: serverless-lab

iam:

role:

statements:

- Effect: Allow

Action:

- logs:CreateLogGroup

- logs:CreateLogStream

- logs:PutLogEvents

Resource: "arn:aws:logs:${self:provider.region}:\*:log-group:/aws/lambda/${self:service}-${self:provider.stage}-\*:\*"

- Effect: Allow

Action:

- dynamodb:PutItem

- dynamodb:GetItem

- dynamodb:Scan

Resource:

- arn:aws:dynamodb:ap-south-1:\*:table/TodoTable

- Effect: Allow

Action:

- sqs:SendMessage

- sqs:ReceiveMessage

Resource: "\*"

# <-- ADD THIS BLOCK FOR SNS-->

- Effect: Allow

Action:

- sns:Publish

Resource:

- Ref: MyTopic

functions:

hello:

handler: handlers/hello.handler

events:

- httpApi:

path: /hello

method: get

createTodo:

handler: handlers/create\_todo.handler

events:

- httpApi:

path: /todos

method: post

publishMessage:

handler: handlers/publish\_message.handler

environment:

MY\_TOPIC\_ARN:

Ref: MyTopic

events:

- httpApi:

path: /publish

method: post

consumeMessage:

handler: handlers/consume\_message.handler

events:

- sqs:

arn:

Fn::GetAtt: [MyQueue, Arn]

batchSize: 5 # process up to 5 messages per Lambda invocation

maximumBatchingWindow: 30 # seconds; optional

enabled: true

resources:

Resources:

TodoTable:

Type: AWS::DynamoDB::Table

Properties:

TableName: TodoTable

BillingMode: PAY\_PER\_REQUEST

AttributeDefinitions:

- AttributeName: id

AttributeType: S

KeySchema:

- AttributeName: id

KeyType: HASH

MyQueue:

Type: AWS::SQS::Queue

Properties:

QueueName: my-queue

MyTopic:

Type: AWS::SNS::Topic

Properties:

TopicName: my-topic

MyTopicSubscription:

Type: AWS::SNS::Subscription

Properties:

Protocol: sqs

TopicArn:

Ref: MyTopic

Endpoint:

Fn::GetAtt:

- MyQueue

- Arn

# Allow SNS to send messages to the queue (policy attached to the queue)

MyQueuePolicy:

Type: AWS::SQS::QueuePolicy

Properties:

Queues:

- Ref: MyQueue

PolicyDocument:

Version: "2012-10-17"

Statement:

- Sid: Allow-SNS-SendMessage

Effect: Allow

Principal: "\*"

Action: "sqs:SendMessage"

Resource:

Fn::GetAtt:

- MyQueue

- Arn

Condition:

ArnEquals:

"aws:SourceArn":

Ref: MyTopic

Outputs:

MyTopicArn:

Description: "SNS Topic ARN"

Value: !Ref MyTopic

# Optional: export it for cross-stack usage

Export:

Name: ${self:service}-${self:provider.stage}-MyTopicArn

MyQueueArn:

Description: "SQS Queue ARN"

Value: !GetAtt MyQueue.Arn

Export:

Name: ${self:service}-${self:provider.stage}-MyQueueArn



## **Verification Checklist ✅**

* Hello endpoint working via API Gateway.
* DynamoDB table created, items inserted via POST.
* SNS → SQS → Lambda flow works.
* Logs visible in CloudWatch.
* CloudFormation stack shows correct resources.

## **Notes & Cautions ⚠️**

* Always clean up (sls remove) to avoid charges.
* IAM roles must follow least privilege.
* Keep region consistent (ap-south-1).
* Ensure you update TopicArn correctly in publish\_message.py.

## **Final Note 📌**

This activity is not evaluated.

Students must perform all hands-on steps for self-learning. Do attempt the quiz.

In the next activity, you will be asked to create a similar stack from scratch — and that will be evaluated.

═══ END OF DOCUMENT ═══

4️⃣ Activity 4: Exercise

# **Activity4 — Exercise: Image Processing Pipeline (Student Assignment)**

## **Objective 🎯**

In this activity, you will **independently build a complete serverless image processing pipeline** using Serverless Framework v4 on AWS.

By the end of this assignment you should be able to:

* Write a full serverless.yml that wires together AWS Lambda, S3, DynamoDB, and SNS.
* Configure IAM roles and policies for least privilege.
* Deploy and verify a multi-step workflow where:
  1. A user uploads an image to S3.
  2. A Lambda resizes it, stores metadata in DynamoDB, and publishes an SNS notification.
  3. Another Lambda consumes SNS and logs structured analytics data into an S3 bucket.

This exercise is **evaluated** — you must implement everything end-to-end and submit your work.

## **Prerequisites**

Before starting this activity, make sure you have:

1. Completed **Activity1** to **Activity3**.
2. Configured AWS CLI with the profile serverless-lab:

aws configure --profile serverless-lab



1. Logged in to Serverless Framework Dashboard:

serverless login



⚠️ **Note:** AWS CLI and Serverless Dashboard login are **one-time setup per activity**. You must do it at the start of each new activity.

## **Plugins Required**

This activity requires the following Serverless Framework plugin:

* serverless-python-requirements → to bundle Python dependencies (like Pillow for image resizing).

### Installation Steps

Run this in your activity4-image-pipeline/ project folder:

npm init -y # if package.json does not exist

npm install --save-dev serverless-python-requirements



Enable it in your serverless.yml:

plugins:

- serverless-python-requirements

custom:

pythonRequirements:

dockerizePip: true

zip: true

slim: true



This ensures dependencies listed in requirements.txt (like boto3, pillow) are packaged inside your Lambda zip correctly.

## **Problem Statement 📝**

We want to build a **Serverless Image Processing Pipeline** that automates:

1. **Uploads** → User uploads image to a private S3 bucket.
2. **Processing** → A Lambda function resizes the image, stores metadata in DynamoDB, and publishes result to SNS.
3. **Analytics** → Another Lambda subscribes to SNS and stores structured logs into a separate Analytics S3 bucket.

The flow:

**Start → Upload → Process → Store Metadata → Publish → Log Analytics → End.**

## **Folder Structure 📂**

Your workspace should look like this:

/home/labDirectory/activity4-image-pipeline

├── data.json

├── handlers

│ ├── analytics\_logger.py

│ └── image\_processor.py

├── instructor\_policy.json

├── problem\_statement.txt

├── requirements.txt

├── LEARN.jpg

└── serverless.yml



## **Lambda Functions (Provided) 🐍**

You are given two Lambda functions inside handlers/.

### 🔹 image\_processor.py — Image Processor Lambda

* Triggered automatically when a new object is uploaded to the Uploads bucket.
* Steps:
  1. Downloads image from Uploads bucket.
  2. Creates a 128x128 thumbnail using Pillow (PIL).
  3. Uploads thumbnail to the Thumbnails bucket (public-read).
  4. Stores metadata (jobId, keys, status, timestamp) in DynamoDB table.
  5. Publishes success/failure notification to SNS Topic.
* Uses **logging** for every step.

### 🔹 analytics\_logger.py — Analytics Logger Lambda

* Subscribed to the SNS Topic.
* Steps:
  1. Receives notification from SNS.
  2. Formats into structured JSON log.
  3. Saves log to Analytics S3 bucket.
* This bucket acts like a **data lake**.

### 🔹 requirements.txt

boto3>=1.26.0

Pillow>=9.0.0



## **Your Task (Student Assignment)** 🛠️

You must write the **serverless.yml** file that ties everything together:

1. **Buckets**
   * UploadsBucket (private)
   * ThumbnailsBucket (public-read)
   * AnalyticsBucket (private)
2. **DynamoDB Table**
   * JobsTable with primary key jobId (string).
3. **SNS Topic**
   * ImageEventsTopic.
4. **Functions**
   * imageProcessor triggered by S3 uploads bucket.
   * analyticsLogger subscribed to SNS topic.
   * Inject environment variables (bucket names, table, topic ARN).
5. **IAM Role**
   * CloudWatch Logs permissions.
   * S3 read/write (scoped to relevant buckets).
   * DynamoDB write for JobsTable.
   * SNS publish (for processor).
   * SNS subscribe/invoke (for logger).
6. **Outputs**
   * All resource names (buckets, table, topic ARN).

📌 **Note:** The folder structure is already provided in the **clab directory workspace**. You only need to **fill in** the following files:

* serverless.yml → define resources, functions, IAM, outputs.
* data.json → fill with required values after deployment.

## **Self-Testing Before Submission ✅**

Before submission, run the following commands in order to fully test your stack.

# 1) Deploy stack with verbose output (shows detailed CFN events)

sls deploy --stage dev --region ap-south-1 --aws-profile serverless-lab --verbose

# 2) Get deployed resource info (bucket names, table, topic ARN)

sls info --stage dev --region ap-south-1 --aws-profile serverless-lab --verbose

aws cloudformation describe-stack-resources --stack-name activity4-image-pipeline-dev --region ap-south-1 --profile serverless-lab

aws cloudformation describe-stacks --stack-name activity4-image-pipeline-dev --region ap-south-1 --profile serverless-lab

# 3) Upload a test image(kept in CLAB workspace) to the Uploads bucket (replace <uploads-bucket-name>)

aws s3 cp LEARN.jpg s3://<uploads-bucket-name>/ --profile serverless-lab

# 4) Check DynamoDB table for job entry (replace <jobs-table-name>)

aws dynamodb scan --table-name <jobs-table-name> --profile serverless-lab

# 5) Tail logs of Image Processor Lambda (see resize, DynamoDB, SNS publish actions)

sls logs -f imageProcessor --stage dev --tail

sls logs -f imageProcessor --stage dev --startTime 5h

sls logs -f imageProcessor --stage dev --startTime 2d

# 6) Tail logs of Analytics Logger Lambda (see SNS → S3 log writes)

sls logs -f analyticsLogger --stage dev --tail

sls logs -f analyticsLogger --stage dev --startTime 5h

sls logs -f analyticsLogger --stage dev --startTime 2d

# 7) (Optional) Receive a message directly from SQS queue (debug SNS→SQS if added)

aws sqs receive-message --queue-url <queue-url> --profile serverless-lab

# 8) (Optional) Remove stack to avoid charges

sls remove --stage dev --region ap-south-1 --aws-profile serverless-lab



✅ If the test image appears in the **Thumbnails bucket**, a record shows up in **DynamoDB**, and a log JSON file is written in the **Analytics bucket**, then your stack is working correctly.

## **Evaluation 📤**

This activity **will be tested using an automated script**.

You must:

1. Deploy your stack successfully (sls deploy --stage dev).
2. Create a temporary **Instructor IAM User** with the policy below.
3. Provide credentials and stack details in data.json.

### 🔹 Instructor Policy (Save as instructor\_policy.json)

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"cloudformation:\*",

"s3:\*",

"lambda:\*",

"iam:PassRole",

"iam:CreateRole",

"iam:GetRole",

"iam:DeleteRole",

"iam:AttachRolePolicy",

"iam:DetachRolePolicy",

"iam:DeleteRolePolicy",

"iam:PutRolePolicy",

"apigateway:\*",

"dynamodb:\*",

"sqs:\*",

"sns:\*",

"logs:\*",

"iam:ListRolePolicies",

"iam:GetRolePolicy",

"iam:ListAttachedRolePolicies",

"iam:GetPolicy",

"iam:GetPolicyVersion",

"iam:ListRoles",

"iam:ListPolicies"

],

"Resource": "\*"

}

]

}



⚠️ Note: This user must be **deleted after grading**.

### 🔹 data.json Format

{

"instructor\_iam\_username": "<IAM > Users (username you created for instructor)>",

"instructor\_access\_key\_id": "<IAM > Security credentials (Access key)>",

"instructor\_secret\_access\_key": "<IAM > Security credentials (Secret key, only shown at creation)>",

"aws\_region": "<Region where you deployed stack (e.g., ap-south-1)>",

"stack\_name": "<From 'sls info' → stack name (e.g., activity4-app-dev)>",

"uploads\_bucket": "<From 'sls info' → S3 Uploads bucket name>",

"thumbnails\_bucket": "<From 'sls info' → S3 Thumbnails bucket name>",

"analytics\_bucket": "<From 'sls info' → S3 Analytics bucket name>",

"jobs\_table": "<From 'sls info' → DynamoDB table name>",

"sns\_topic\_arn": "<From 'sls info' or CloudFormation Outputs → SNS Topic ARN>"

}



## **Submission Checklist ✅**

* serverless.yml complete and correct.
* instructor\_policy.json applied to Instructor IAM user.
* data.json filled.
* Stack deploys, processes image uploads, produces thumbnails, stores metadata, publishes SNS, and writes analytics logs.
* The stack is running live in AWS.

## **Final Note 📌**

This assignment **is evaluated**.

Be precise with YAML, environment variables, and IAM.

After the evaluation is done, ensure resources are cleaned up (sls remove) unless instructed otherwise.

## 🏆 Congratulations: You’ve completed the Lab

═══ END OF DOCUMENT ═══