AWS Assignment 1

Objective:

This assignment will help everyone understand and implement key AWS services, including IAM, EC2, VPC, Subnets, and Nginx. The goal is to set up a basic web server in a secure VPC environment.

Assignment Tasks:

1. IAM (Identity and Access Management)

- Create an IAM user with programmatic access and assign it to a custom IAM group with **EC2 and VPC Full Access**.
- Create a policy that allows the user to start, stop, and terminate EC2 instances but restricts access to other AWS services.

2. VPC (Virtual Private Cloud) and Subnets

- Create a **custom VPC** with CIDR block **192.168.0.0/16**.
- Inside the VPC, create two **subnets**:
    - **Public Subnet**: 192.168.1.0/24
    - **Private Subnet**: 192.168.2.0/24
- Set up an **Internet Gateway** and attach it to the VPC.
- Configure a **route table** to allow internet access only for the public subnet.

3. EC2 (Elastic Compute Cloud) Instance

- Launch an **EC2 instance** in the **public subnet** using Amazon Linux 2 or Ubuntu.
- Attach a **security group** that allows inbound SSH (port 22) and HTTP (port 80) access.

4. Install & Configure Nginx Web Server

- Connect to the EC2 instance via SSH.

Submission Requirements:

1. **Screenshots** of:
    - IAM User & Policy
    - VPC, Subnets, and Route Table Configuration
    - Running EC2 Instance with Public IP

    - Webpage running on Nginx
2. **Commands Used** (in a text file or PDF).
3. **Explanation** (brief write-up on what was learned).

Evaluation Criteria:

✔ Correct implementation of IAM, VPC, Subnet, EC2
✔ Successful Nginx installation and webpage hosting
✔ Clear documentation and screenshots

AWS Assignment 2

Below is an example assignment that combines AWS Amplify, API Gateway, Lambda, SNS, and DynamoDB to build a simple serverless web application. This assignment is designed to help all gain hands-on experience with AWS serverless technologies.

Objective

Build a serverless web application that allows users to submit data through a frontend application. The submitted data is processed by an API (via API Gateway and Lambda), stored in DynamoDB, and a notification is sent using SNS upon each successful data entry.

Assignment Tasks

1. AWS Amplify – Frontend Setup & Deployment

- **Create a New Amplify App:**
    - Initialize an Amplify project (using the Amplify CLI or Amplify Console).
    - Connect your Amplify project to a Git repository containing a simple static web app (e.g., built with React, Angular, or basic html css js).

- **Frontend Application:**
    - Develop a basic user interface with a form that collects sample user data (e.g., name, email, message).
    - Add functionality to call a REST API endpoint (to be created in Task 2) when the form is submitted.
- **Deployment:**
    - Deploy the frontend using Amplify Hosting.
    - Verify that the app is accessible via the Amplify-provided URL.

2. API Gateway – REST API Setup

- **Create a REST API:**
    - In the API Gateway console, create a new REST API.
    - Define a resource (e.g., `/submit`) with a POST method.
- **Integration with Lambda:**
    - Configure the POST method to trigger a Lambda function (created in Task 3).
    - Enable CORS on the API so that the Amplify-hosted frontend can call it.

3. AWS Lambda – Function Development

- **Create a Lambda Function:**
    - Develop a Lambda function in your preferred runtime (Node.js, Python, etc.).
    - The function should perform the following:
        - Parse the incoming JSON payload from API Gateway.
        - Insert the received data into a DynamoDB table (see Task 4).
        - Publish a notification to an SNS topic (see Task 5) confirming data receipt.
        - Return a suitable response (e.g., a success message and the stored data).
- **Permissions:**
    - Ensure the Lambda execution role has permissions to interact with DynamoDB and SNS.

4. DynamoDB – Data Storage

- **Create a DynamoDB Table:**
    - In the DynamoDB console, create a table (e.g., `UserSubmissions`) with an appropriate primary key (for example, `submissionId` as a UUID or timestamp).
- **Integrate with Lambda:**
    - Within your Lambda function, use the AWS SDK to insert new records into this table.
- **Data Validation:**
    - Test the table by manually inserting a sample record or using the Lambda function.

5. SNS – Notification Setup

- **Create an SNS Topic:**
  - In the SNS console, create a new topic (e.g., `SubmissionNotifications`).
  - Optionally, subscribe an email endpoint to the topic for real-time notifications.
- **Integrate with Lambda:**
  - Update your Lambda function to publish a message to the SNS topic after successfully inserting a record into DynamoDB.
  - The notification message should contain details about the new submission.

6. Testing & Validation

- **End-to-End Testing:**
  - Use the Amplify-hosted frontend to submit data.
  - Verify that:
    - The API Gateway correctly triggers the Lambda function.
    - The Lambda function stores the data in DynamoDB.
    - An SNS notification is published (and received if using email subscriptions).
    - A proper response is returned to the frontend, and the user sees confirmation.
- **Debugging:**
  - Check CloudWatch logs for Lambda and API Gateway if issues arise during testing.

Submission Requirements

1. **Documentation:**
   - A report describing your architecture and each component's role.
   - A diagram illustrating the flow from Amplify to API Gateway, Lambda, SNS, and DynamoDB.
2. **Screenshots:**
   - Amplify Console showing the deployed frontend.
   - API Gateway configuration.
   - Lambda function code and CloudWatch logs.
   - DynamoDB table with sample data.
   - SNS topic configuration and any notification received (if applicable).
3. **Source Code:**
   - Provide a link to the Git repository containing your frontend application and Lambda function code.
4. **Commands & Configurations:**
   - A text file or PDF listing important CLI commands (if used) and configuration settings.

Evaluation Criteria

- Correct integration of AWS Amplify, API Gateway, Lambda, SNS, and DynamoDB.
- Successful end-to-end functionality of the application.
- Clarity and completeness of documentation and code.
- Implementation of best practices for security (e.g., proper IAM roles and policies) and error handling.

AWS Assignment 3 - Brief

Below is an assignment that integrates Step Functions, CloudWatch, SES, EventBridge Scheduler, EventBridge, SNS, and SQS into a cohesive serverless order processing pipeline. This scenario will give hands-on experience with orchestrating workflows, event-driven architectures, and monitoring/logging across multiple AWS services.

Assignment: AWS Serverless Mini-Pipeline

Objective

Build a simple serverless workflow that:

- Receives an order message via SQS.
- Processes the message through a Step Functions workflow invoking a single Lambda function.
- Uses that Lambda function to log the event in CloudWatch, send a confirmation email via SES, and publish a notification via SNS.
- Uses an EventBridge Scheduler (or EventBridge rule) to trigger a heartbeat Lambda that logs a periodic message.

Assignment Tasks

1. SQS – Simple Message Queue

- **Create an SQS Queue:**
  - **Name:** `SimpleOrderQueue`
  - Use default settings to keep it simple.
- **Send a Test Message:**

Use the AWS Console or CLI to send a sample JSON message (e.g.,
{ "orderId": "001", "customerEmail": "raju@gmail.com" }
```).
```

2. Lambda Function – Process Order Message

- **Create a Lambda Function:**
  - **Name:** `ProcessOrderFunction`
  - **Runtime:** (e.g., Python, Node.js)
- **Function Tasks:**
  - **Log Event:** Write the incoming event details to CloudWatch.
  - **Send Email via SES:**
    - Use the SES API to send a simple confirmation email to the `customerEmail` received.
    - (Ensure that the sender email is verified in SES.)
  - **Publish SNS Notification:**
    - Publish a message to an SNS topic (created in Task 4) confirming order processing.
- **IAM Permissions:**
  - Ensure the Lambda execution role includes permissions for CloudWatch logs, SES, and SNS.

---

3. Step Functions – Minimal Workflow

- **Create a State Machine:**
  - **Name:** `SimpleOrderWorkflow`
  - **Definition:**
    - A single state that invokes the `ProcessOrderFunction` Lambda.
    - Pass the SQS message payload directly as input.
- **Triggering the Workflow:**
  - **Option A:** Manually start an execution with the test message.
  - **Option B:** (For extra practice) Configure an EventBridge rule to trigger the state machine on a schedule.

---

4. SNS – Notification Setup

- **Create an SNS Topic:**
  - **Name:** `SimpleOrderNotifications`
- **Subscribe to the Topic:**
  - Add an email subscription (verify the email if necessary).
- **Integration:**
  - The `ProcessOrderFunction` will publish a notification to this topic once the order is processed.

---

5. SES – Email Confirmation

- **Configure SES:**
  - **Verify Email:** Ensure your sender email is verified in SES (and the recipient if needed, due to sandbox restrictions).
- **Integration:**
  - Within `ProcessOrderFunction`, use SES to send an order confirmation email containing basic order details.

---

6. EventBridge Scheduler – Heartbeat Test

- **Create an EventBridge Rule or Scheduler:**
  - **Name:** `HeartbeatRule`
  - **Schedule:** Set a rule to trigger every 5–10 minutes (or a one-time test schedule).
- **Create a Heartbeat Lambda Function:**
  - **Name:** `HeartbeatFunction`
  - **Task:** Log a simple "Heartbeat – Scheduler Triggered" message to CloudWatch.
- **Integration:**
  - Configure the EventBridge rule to trigger `HeartbeatFunction` on schedule.

---

7. CloudWatch – Monitoring & Logging

- **Enable Logging:**
  - Ensure all Lambda functions and the Step Functions state machine output logs to CloudWatch.
- **Verification:**
  - Check CloudWatch logs to confirm that:
    - `ProcessOrderFunction` logs the received SQS message.
    - SES and SNS operations are logged.
    - `HeartbeatFunction` logs a scheduled heartbeat message.

Submission Requirements

1. **Documentation:**
   - A brief write-up explaining the purpose of each service and how they interact.
   - A simple diagram showing the flow:
     - SQS → Step Functions → Lambda → (SES & SNS)
     - EventBridge triggering the Heartbeat Lambda.
2. **Screenshots:**

    - SQS Queue configuration with a sample message.
    - Step Functions state machine definition and a sample execution.
    - Lambda function code (or snippet) with CloudWatch log outputs.
    - SNS topic configuration and a screenshot of the received email notification.
    - EventBridge rule/scheduler configuration and the Heartbeat Lambda's CloudWatch logs.
3. **Source Code:**
    - Provide code snippets for your Lambda functions.
4. **Deployment Instructions:**
    - A brief guide on how to deploy and test your solution.

Evaluation Criteria

- **Service Integration:** Correct creation and integration of SQS, Step Functions, Lambda, SNS, SES, EventBridge, and CloudWatch.
- **Functionality:** The pipeline should process a test message end-to-end, log events, send a confirmation email, and publish an SNS notification.
- **Simplicity:** Implementation should be achievable within approximately 2 hours.
- **Documentation:** Clear and concise documentation with diagrams and screenshots.

This assignment allows you to gain hands-on experience with key AWS serverless and event-driven services, Happy building!

AWS Assignment 4

Below is an assignment that combines Jenkins, Docker, GitHub, and shell scripting. This assignment is designed to help freshers understand how to create a CI/CD pipeline that builds and deploys a containerized application.

Assignment: CI/CD Pipeline with Jenkins, Docker, GitHub, and Shell Scripting

Objective

Create an automated CI/CD pipeline that:

- Hosts a simple web application in a GitHub repository.
- Uses a Dockerfile to containerize the application.
- Leverages Jenkins (running on an EC2 instance or your local machine) to trigger builds when changes are pushed to GitHub.
- Uses Jenkins to build the Docker image, run a container, and execute shell scripts that verify the deployment.

Assignment Tasks

1. GitHub Repository Setup

- **Create a Repository:**
  - Create a new GitHub repository (e.g., `simple-web-app`).
- **Develop a Simple Web Application:**
  - Create a minimal web application (e.g., using Node.js, Python Flask, or a static HTML page served by Nginx).
- **Dockerfile:**
  - Write a Dockerfile that:
    - Uses an appropriate base image.
    - Copies the web application code.
    - Installs necessary dependencies.
    - Exposes the appropriate port (e.g., 80 or 8080).
    - Specifies the command to run the application.

- **Push Code:**
    - Ensure the repository contains all the necessary code files, the Dockerfile, and a README with instructions.

---

2. Jenkins Setup

- **Jenkins Installation:**
    - Install Jenkins on an EC2 instance or your local machine.
- **Configure GitHub Integration:**
    - Set up a GitHub webhook so that Jenkins triggers a build when code is pushed.
- **Create a Pipeline Job:**
    - Configure a Jenkins Pipeline (or freestyle job) that performs the following steps:
        - **Clone Repository:** Pull code from your GitHub repository.
        - **Build Docker Image:** Run `docker build` to create an image from your Dockerfile.
        - **Run Docker Container:** Use `docker run` to start a container from the built image.
        - **Execute Verification Script:** Run a shell script (see Task 3) to verify that the container is running and serving the web application.

---

3. Docker & Shell Scripting

- **Docker Integration in Jenkins Pipeline:**
    - Ensure the Jenkins environment has Docker installed and the Jenkins user has permission to run Docker commands.
- **Create a Shell Script (e.g., `verify.sh`):**
    - **Purpose:** Verify that the Docker container is up and the web application is responding.
    - **Tasks:**
        - Check if the Docker container is running (e.g., using `docker ps`).
        - Use `curl` to fetch the web page served by the container.
        - Print a success or error message based on the HTTP response.
- **Integrate Shell Script:**
    - Include this shell script as a post-build step in your Jenkins Pipeline to confirm the deployment was successful.

---

4. (Optional Bonus) Docker Hub Integration

- **Push Docker Image:**

- Extend your pipeline to tag the Docker image and push it to Docker Hub.
- Ensure that your Docker Hub credentials are securely stored in Jenkins (e.g., using credentials plugins).

Submission Requirements

1. **Documentation:**
   - A README file in your GitHub repository that explains:
     - The web application functionality.
     - How to build and run the Docker container locally.
     - Instructions on how the Jenkins pipeline is configured.
   - A simple architecture diagram showing:
     - GitHub → Jenkins Pipeline → Docker (Build & Run) → Shell Script Verification.
2. **Screenshots:**
   - GitHub repository structure.
   - Jenkins pipeline configuration and build logs.
   - Output of the shell script (e.g., successful curl response and container status).
   - (Optional) Docker Hub repository view after pushing the image.
3. **Source Code:**
   - Provide links to your GitHub repository containing:
     - Web application code.
     - Dockerfile.
     - Shell script (`verify.sh`).
     - Jenkins Pipeline script (Jenkinsfile) or configuration details.
4. **Deployment Instructions:**
   - A step-by-step guide on how to set up the Jenkins job, configure the GitHub webhook, and run the pipeline.

Evaluation Criteria

- **Integration:** Proper integration of GitHub, Jenkins, Docker, and shell scripting.
- **Functionality:** The pipeline should successfully clone the repository, build the Docker image, run the container, and execute the verification script.
- **Clarity & Documentation:** Clear instructions, comments, and diagrams that explain the process.
- **Automation:** Effective use of Jenkins to automate the build and deployment process.
- **Bonus Enhancements:** If Docker Hub integration is implemented, proper handling of image tagging and pushing.

This assignment will help you gain practical experience in setting up CI/CD pipelines with modern DevOps tools. Happy building!

AWS Assignment 5

Below is a **small assignment** that combines Terraform, Linux, and Bash scripting. This task will have you provision infrastructure with Terraform, configure a Linux EC2 instance via a Bash user-data script, and then use an additional Bash script to verify and interact with the deployed instance.

Assignment: Automated Web Server Deployment with Terraform & Bash Scripting

Objective

Provision an AWS EC2 instance running Linux using Terraform. Automatically configure the instance to install and run a web server (Apache or Nginx) via a Bash user-data script, and then create a separate Bash script to SSH into the instance, verify system information, and check the web server's status.

Assignment Tasks

1. Terraform – Provision an EC2 Instance

- **Create Terraform Configuration Files:**
  - **main.tf:**
    - Configure the AWS provider.
    - Define an AWS EC2 instance resource with a suitable Linux AMI.
    - Create a security group that allows:
      - Inbound SSH (port 22)
      - Inbound HTTP (port 80)
  - **variables.tf:**
    - Define variables for instance type, region, key pair name, etc.
  - **outputs.tf:**
    - Output the public IP address of the EC2 instance.
- **Use a Bash User-Data Script:**
  - Embed a Bash script in the EC2 resource's `user_data` that:
    - Updates the system package index.
    - Installs a web server (choose either Apache or Nginx).

        - Creates or modifies the default index.html to display a custom welcome message.
        - Starts and enables the web server service.

---

2. Bash Scripting – Verification Script

- **Create a Bash Script (e.g., `verify.sh`):**
    - Use SSH (with the proper key) to connect to the newly provisioned EC2 instance.
    - Run commands on the instance to:
        - Display system information (e.g., hostname, uptime).
        - Check that the web server process is running.
        - Optionally, fetch the contents of the index.html (using `curl` or similar) to verify the welcome message.
- **Make sure the script is executable** and document any prerequisites (such as SSH key configuration).

---

3. Testing and Documentation

- **Deploy the Infrastructure:**
    - Initialize Terraform.
    - Run `terraform plan` to review changes.
    - Execute `terraform apply` to provision the EC2 instance.
- **Run the Verification Script:**
    - Once the EC2 instance is up, run your `verify.sh` script to check the system information and web server status.
- **Documentation & Diagram:**
    - Write a brief README that explains:
        - How to set up and run the Terraform configuration.
        - How the Bash user-data script configures the web server.
        - How to execute the verification Bash script.
    - Optionally, include a simple diagram showing:
        - Terraform provisioning → EC2 instance with user-data configuration → SSH verification.

Submission Requirements

1. **Terraform Code:**

    - Include all Terraform configuration files (`main.tf`, `variables.tf`, `outputs.tf`, etc.).
2. **Bash Scripts:**
    - Provide the user-data script (embedded in Terraform) and the separate `verify.sh` script.
3. **Documentation:**
    - A README file with deployment instructions and an explanation of your setup.
    - A diagram (can be a simple image or ASCII diagram) showing the architecture flow.
4. **Screenshots/Outputs:**
    - Screenshot of a successful `terraform apply` output.
    - Terminal output from running `verify.sh` showing system details and web server status.

Evaluation Criteria

- **Correct Terraform Configuration:**
    - Provisioning of an EC2 instance with the required security group and outputs.
- **Effective Bash Scripting:**
    - Proper installation and configuration of the web server via the user-data script.
    - A working verification script that successfully connects via SSH and checks the instance.
- **Documentation & Clarity:**
    - Clear instructions and a logical explanation of each component.
    - Inclusion of a diagram to illustrate the workflow.
- **End-to-End Functionality:**
    - The deployed instance should be accessible (via its public IP) with the custom web page, and the verification script should confirm the server's status.

This assignment will help you gain practical experience with infrastructure as code using Terraform, automate instance configuration with Bash scripting, and interact with Linux systems over SSH. Happy building!