Mastering Environment Variables and Infrastructure Environments in Cloud Scripting

(Two-Phase Implementation: Beginner → Advanced)

Author: Oluwaseun Osunsola
Environment and Tools: AWS, Ubuntu VM, Nano
Project Link: https://github.com/Oluwaseunoa/DevOps-Projects/tree/main

Introduction

In modern software development and cloud operations, the ability to **dynamically configure applications across multiple environments** is a foundational skill. This mini project, titled **"Mastering Environment Variables and Infrastructure Environments in Cloud Scripting"**, explores the **critical distinction** between **infrastructure environments** (distinct deployment stages such as local, testing, and production) and **environment variables** (dynamic key-value pairs used to configure behavior without code changes).

Using a **real-world FinTech product scenario**, this project demonstrates how a single Bash script — `aws_cloud_manager.sh` — can securely and flexibly manage AWS resources across **three isolated environments**:

- **Local (Development)**: VMware Workstation + Ubuntu
- **Testing (AWS Account 1)**: Simulated pre-production
- **Production (AWS Account 2)**: Live customer-facing deployment

The implementation progresses in **two phases**:

1. **Beginner Phase**: Focus on scripting fundamentals — conditionals, `export`, hard-coding pitfalls, and positional parameters
2. **Advanced Phase**: Full AWS integration using IAM users, CLI profiles, and real EC2 provisioning

Every step is **visually documented** with descriptive screenshots, ensuring **reproducibility**, **clarity**, and **alignment with DevOps best practices**.

Definition of Terms

Term	Definition
Infrastructure Environment	A fully isolated deployment stage (e.g., local, testing, production) with dedicated compute, network, and data resources to prevent interference and ensure safe progression through the software lifecycle.
Environment Variable	A named value (e.g., `DB_URL=localhost`) injected into a process at runtime to control configuration. Enables the same code to behave differently across environments.
Positional Parameter	Command-line arguments passed to a script (`$1`, `$2`, etc.) that provide input at execution time, replacing hard-coded values for greater flexibility.

Term	Definition
Hard-Coding	Embedding fixed values directly in code (e.g., `ENVIRONMENT="testing"`). An **anti-pattern** that reduces reusability and increases maintenance risk.
AWS CLI Profile	A named configuration in `~/.aws/credentials` and `~/.aws/config` that stores access keys, region, and output format for secure, account-specific operations.
IAM User	An AWS identity with granular permissions. Used to generate access keys for programmatic access (e.g., via AWS CLI) without using root credentials.
Exit Code	An integer (`0` = success, `1+` = error) returned by a script to indicate execution outcome. Enables robust error handling and automation.
Least Privilege	Security principle granting only the permissions required for a task (e.g., `AmazonEC2FullAccess` instead of admin rights).

Objectives

This project was designed to achieve the following **learning and technical objectives**:

1. **Clarify the difference** between infrastructure environments and environment variables
2. **Demonstrate progression** from hard-coded anti-patterns to dynamic configuration using `export` and positional parameters
3. **Implement robust input validation** and error handling with meaningful exit codes
4. **Create secure IAM users** with least-privilege policies in **two separate AWS accounts**
5. **Configure AWS CLI profiles** (`testing`, `production`) for isolated, secure access
6. **Develop a modular Bash script** that:
    - Accepts `<environment> <number_of_instances>`
    - Validates arguments
    - Checks AWS CLI and profile availability
    - Provisions real EC2 instances using `aws ec2 run-instances --profile`
    - Simulates operations locally via log file creation
7. **Verify all operations** in the AWS Management Console
8. **Apply DevOps principles**: reusability, security, automation, documentation, and incremental development

---

Project Overview

This comprehensive mini project demonstrates the **evolution from beginner-level environment variable handling to advanced AWS multi-account automation**. The implementation progresses through two distinct phases:

Phase 1: Beginner (No Users - Basic Scripting)

- Basic environment variable concepts
- Positional parameters and validation
- Local testing with mock operations
- Script structure and error handling fundamentals

Phase 2: Advanced (IAM Users + AWS Integration)

- Secure IAM user creation with least-privilege policies
- AWS CLI profile configuration for multi-account management
- Real EC2 instance provisioning across testing/production
- Environment-specific resource configurations

The final script `aws_cloud_manager.sh` accepts two arguments: `<environment>` and `<number_of_instances>`, supporting local log file creation or AWS EC2 provisioning via dedicated profiles.

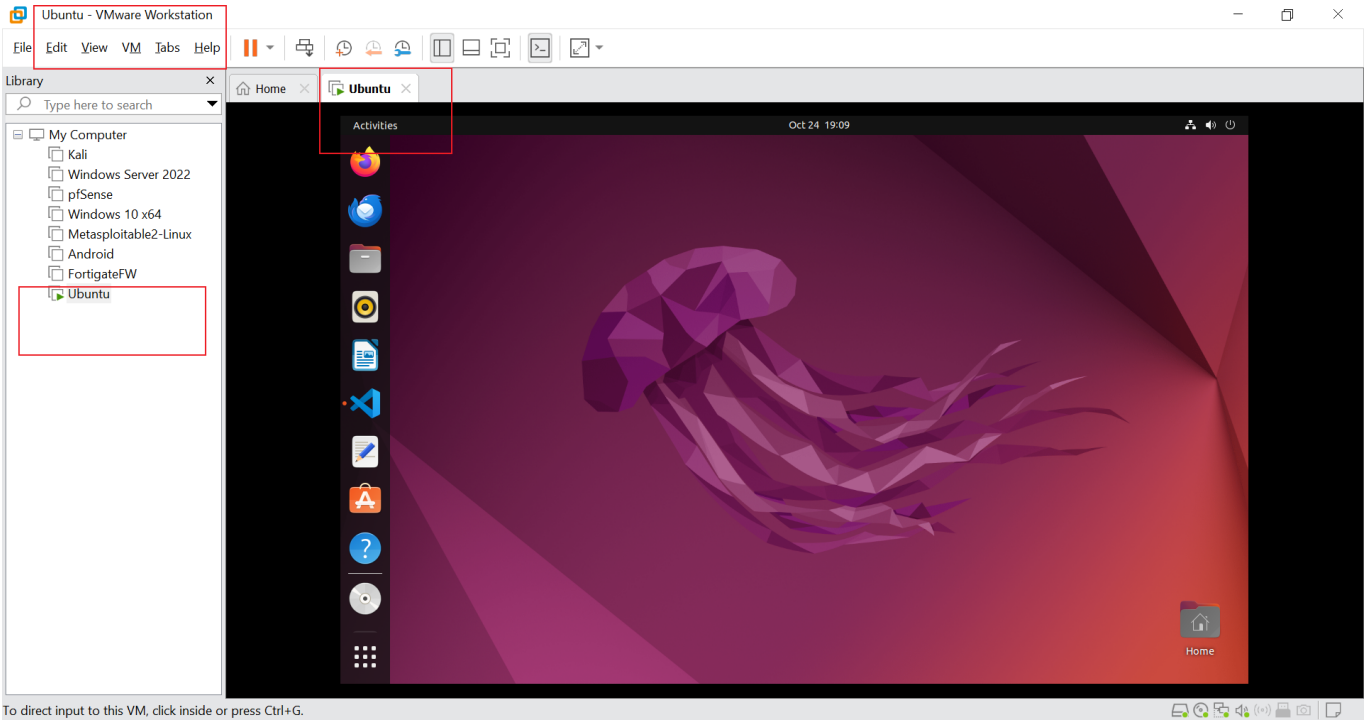
Phase 1: Beginner Implementation (No AWS Users Required)

This phase focuses on **scripting fundamentals** using **environment variables**, **conditionals**, **positional parameters**, and **input validation** — all without requiring AWS credentials or IAM users.

We begin by setting up isolated **infrastructure environments** and evolve a simple script from **static logic** to **dynamic, reusable configuration**.

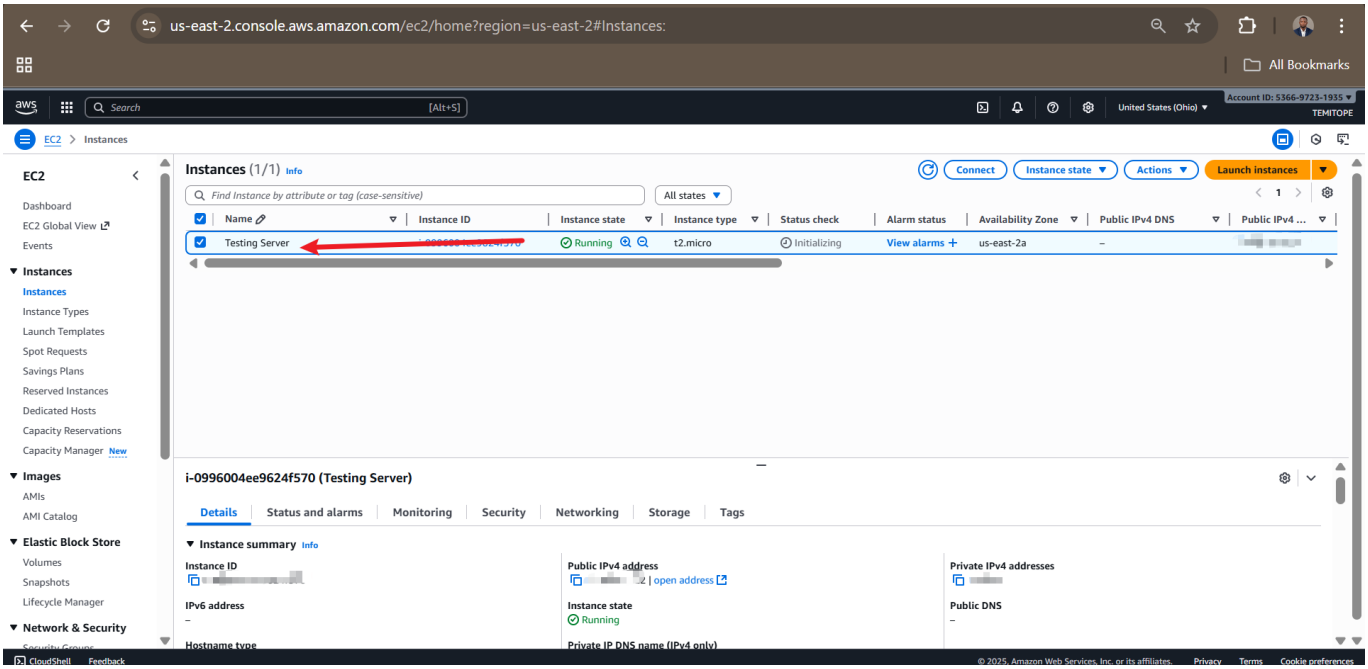
Section 1: Infrastructure Environments Setup

Step 1: Launch Ubuntu VM (Local Development)



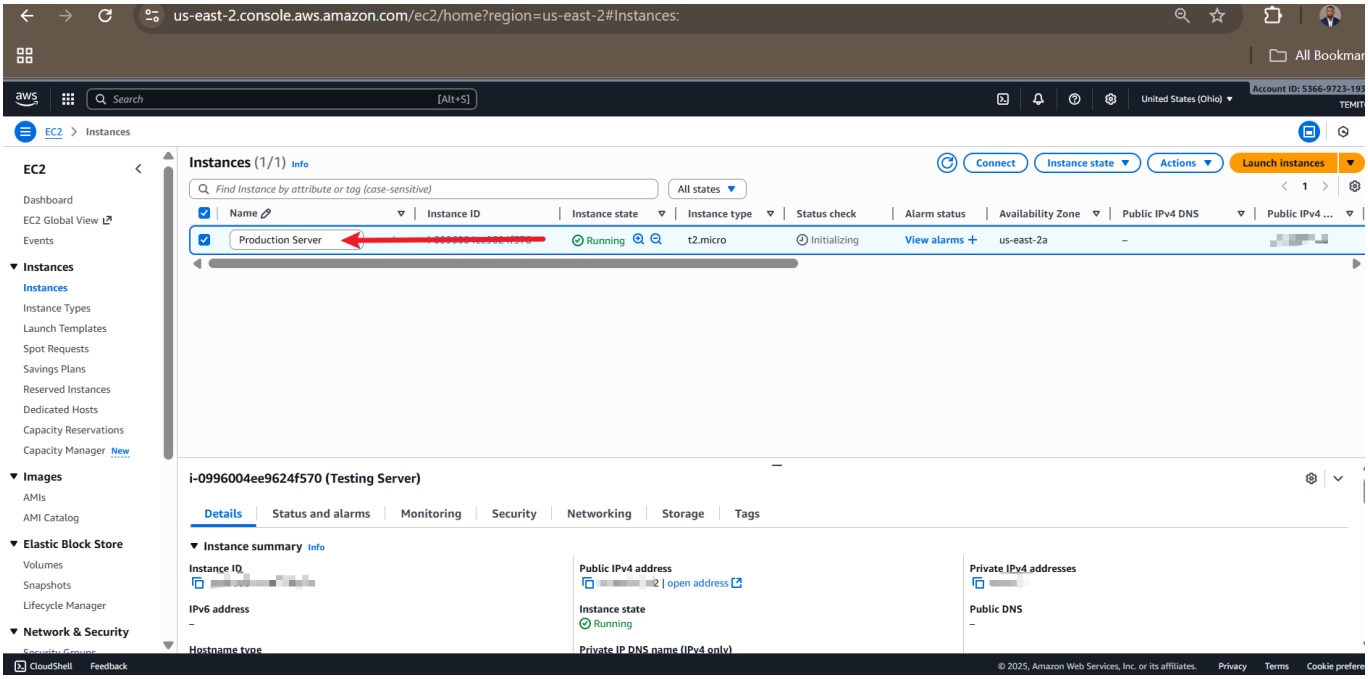
> *Ubuntu 22.04 VM — our development sandbox*

Step 2: Launch Testing EC2 (AWS Account 1)



> *Isolated testing environment — no customer impact*
> **Note:** The existing EC2 server will be deleted and a new one will be created by the automation script.

Step 3: Launch Production EC2 (AWS Account 2)



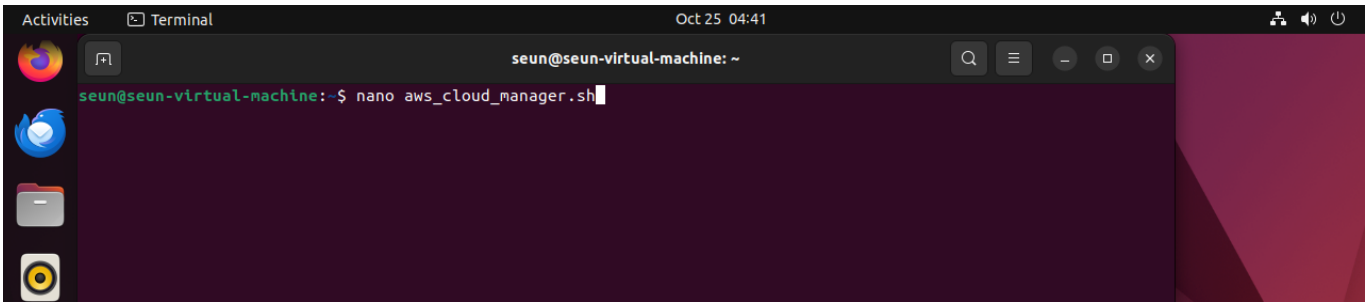
> *Live environment — separate account for security* **Note:** The existing EC2 server will be deleted and a new one will be created by the automation script.

1.2 Creating the Initial Script

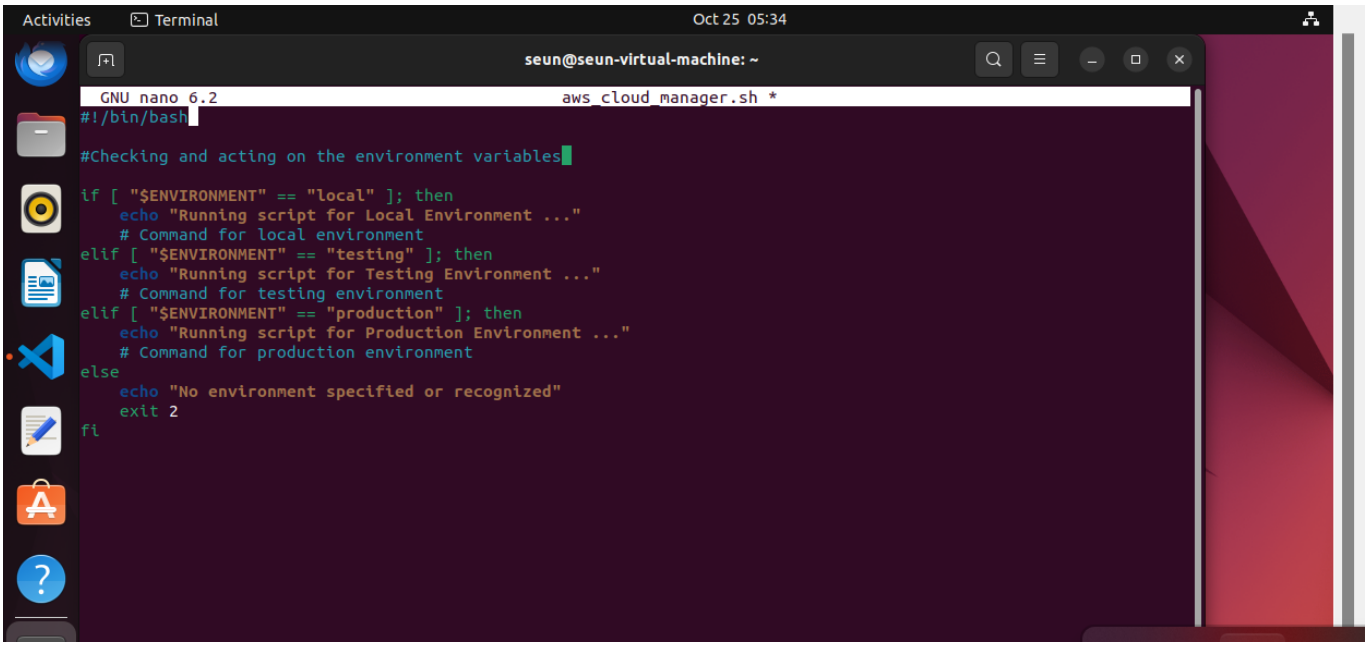
2.1 Script Creation

Step 1: Open nano to create the script

```
nano aws_cloud_manager.sh
```



Step 2: Add basic conditional logic



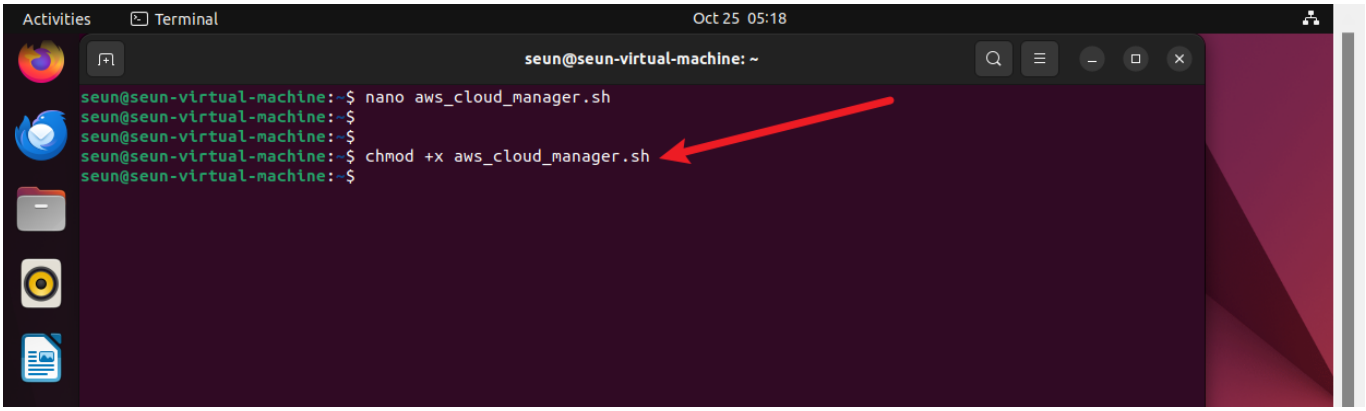

```
#!/bin/bash
# Checking and acting on the environment variable
if [ "$ENVIRONMENT" == "local" ]; then
    echo "Running script for Local Environment..."
elif [ "$ENVIRONMENT" == "testing" ]; then
    echo "Running script for Testing Environment..."
elif [ "$ENVIRONMENT" == "production" ]; then
    echo "Running script for Production Environment..."
else
    echo "No environment specified or recognized."
    exit 2
fi
```

> **Note:** This version relies on the **global environment variable** `$ENVIRONMENT`.

1.3 Making the Script Executable

Step: Grant execution permissions

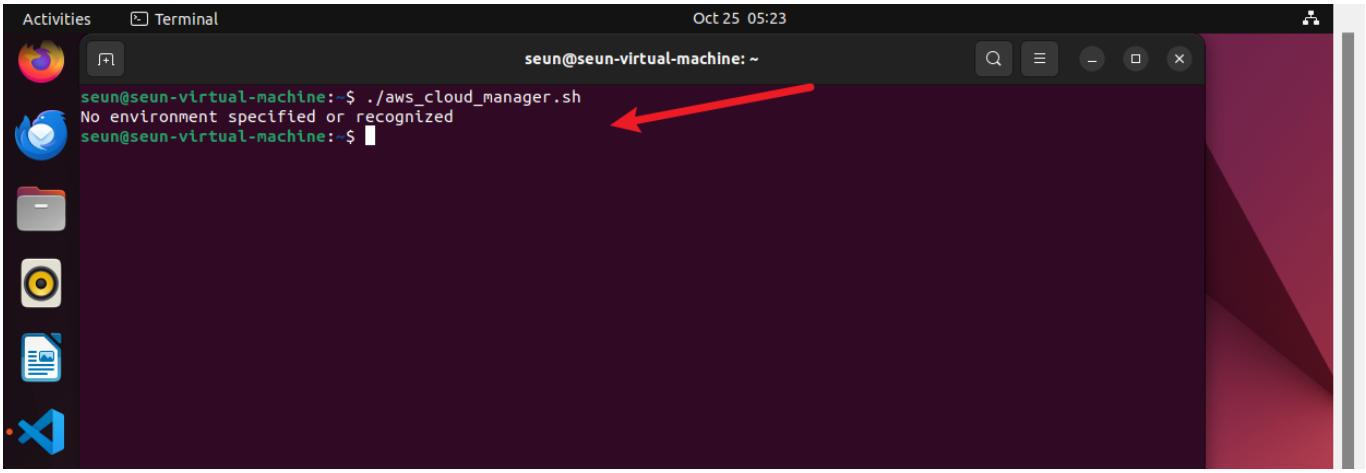
```
sudo chmod +x aws_cloud_manager.sh
```



1.4 Testing with Environment Variables

1.4.1 Run Without Setting `ENVIRONMENT`

```
./aws_cloud_manager.sh
```



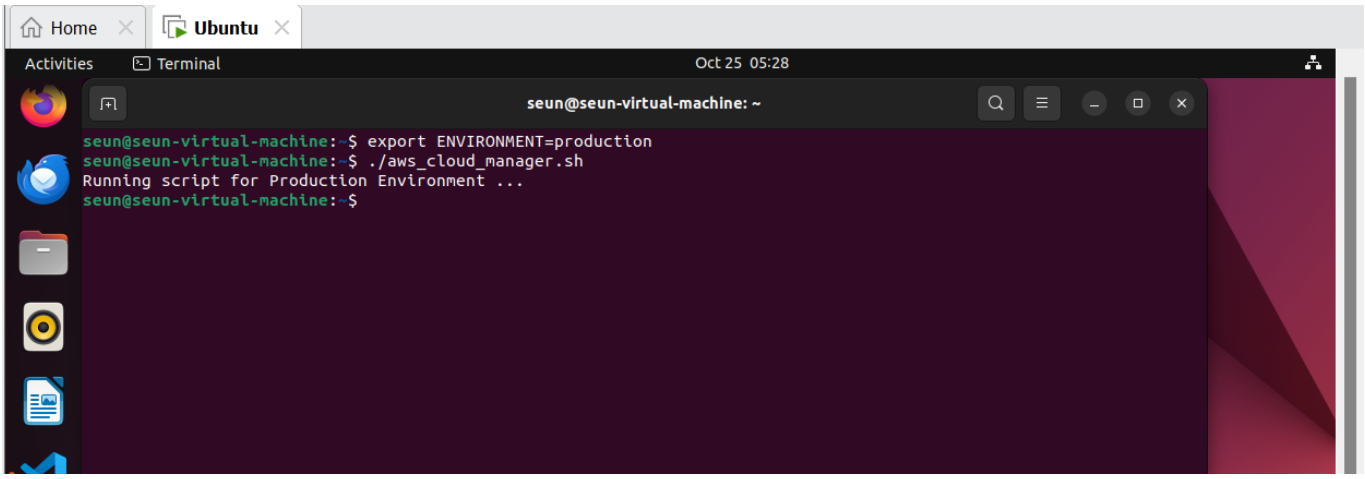
Output:

```
No environment specified or recognized.
```

Exit Code: `2` → Script terminates in `else` block

1.4.2 Use `export` to Set Environment Dynamically

```
export ENVIRONMENT=production
./aws_cloud_manager.sh
```



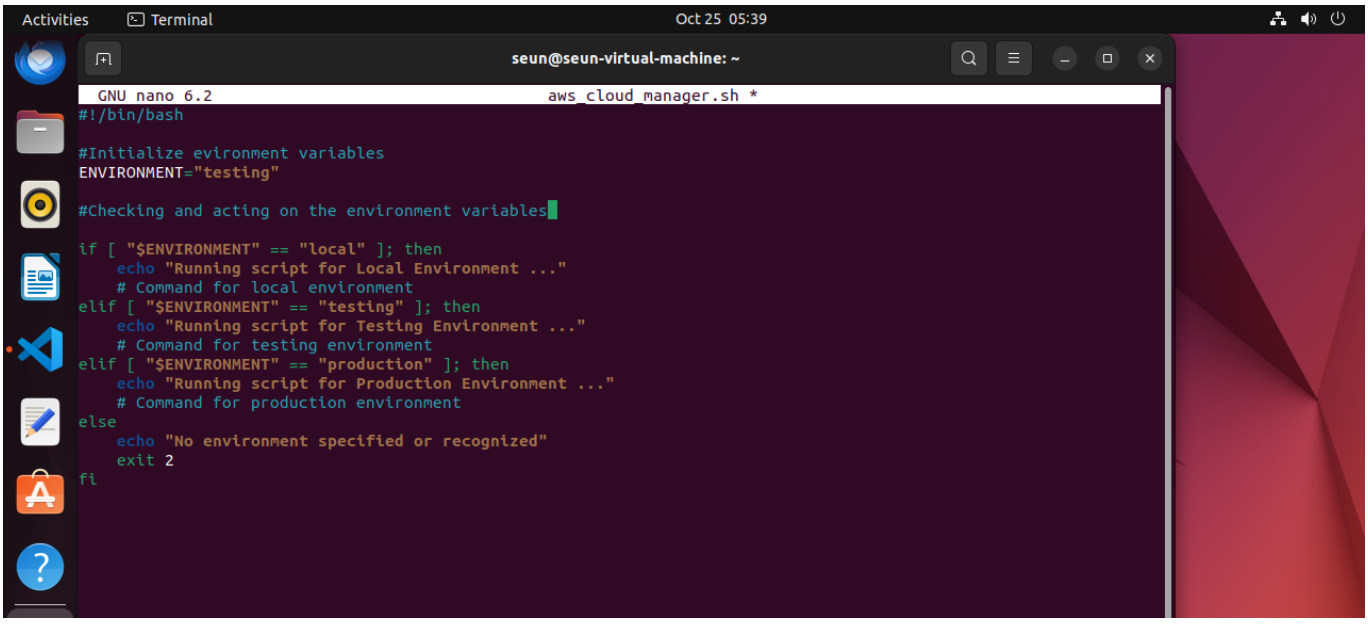
Output:

```
Running script for Production Environment...
```

> **Key Insight:**
> `export` makes the variable available to child processes (like our script). This enables **dynamic configuration** without code changes.

1.5 Hard-Coded Version (Educational Anti-Pattern)

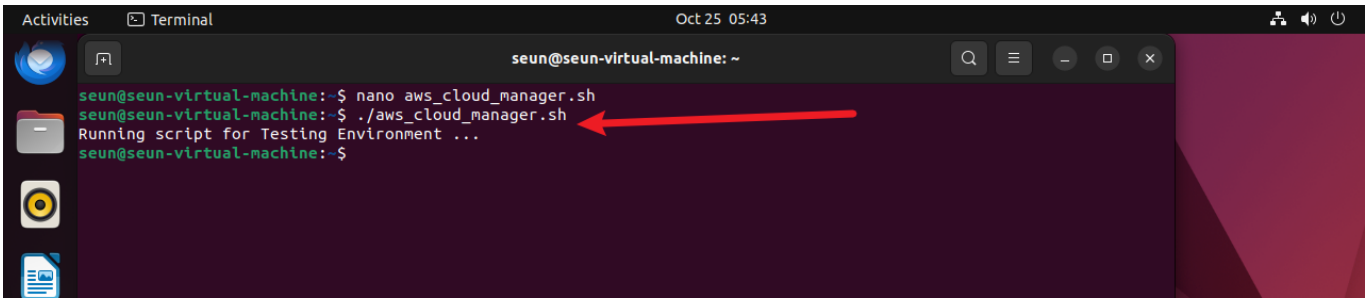
Step: Modify script to hard-code the environment




```
ENVIRONMENT="testing"  # Hard-coded — not flexible
```

Execute:

```
./aws_cloud_manager.sh
```



Output:

```
Running script for Testing Environment...
```

> **Problem:**
>
> - No flexibility
> - Requires code edit per environment
> - Violates **reusability** and **separation of config from code**

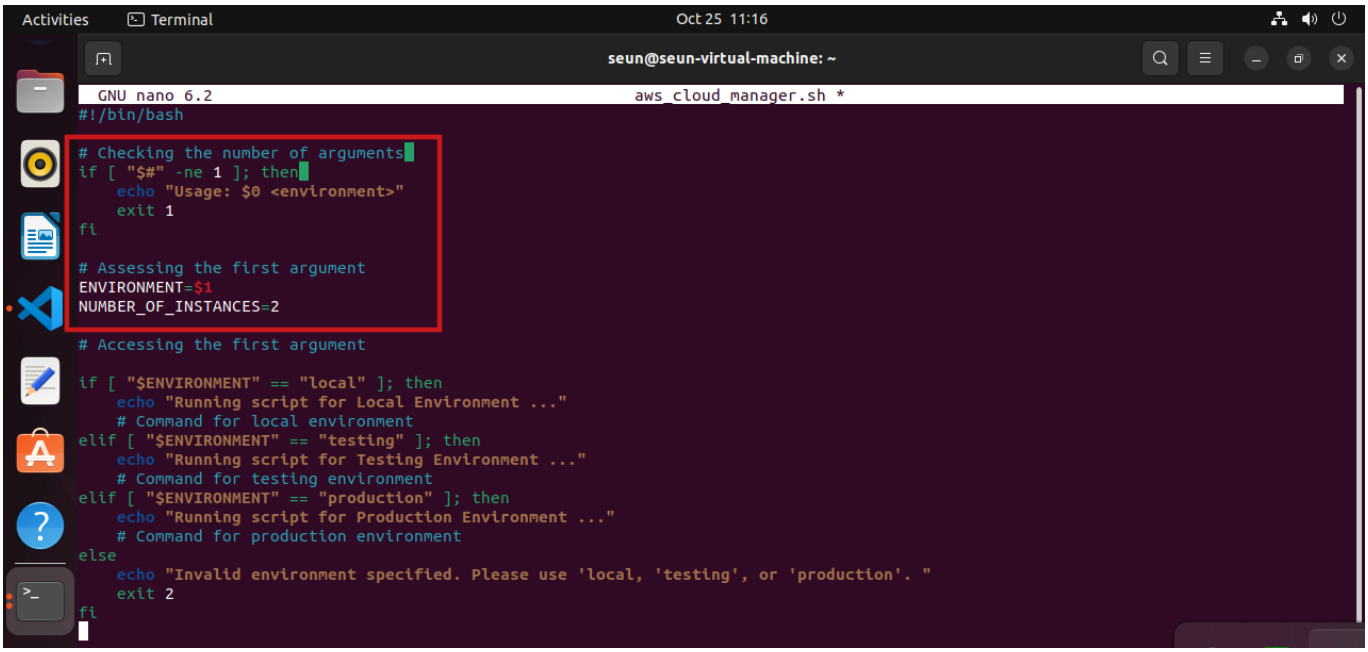
> **Best Practice:** Use **environment variables** or **command-line arguments**

1.6 Introducing Positional Parameters

Goal: Replace hard-coding with **runtime arguments**

```
./aws_cloud_manager.sh testing 5
```

1.6.1 Add Argument Count Validation



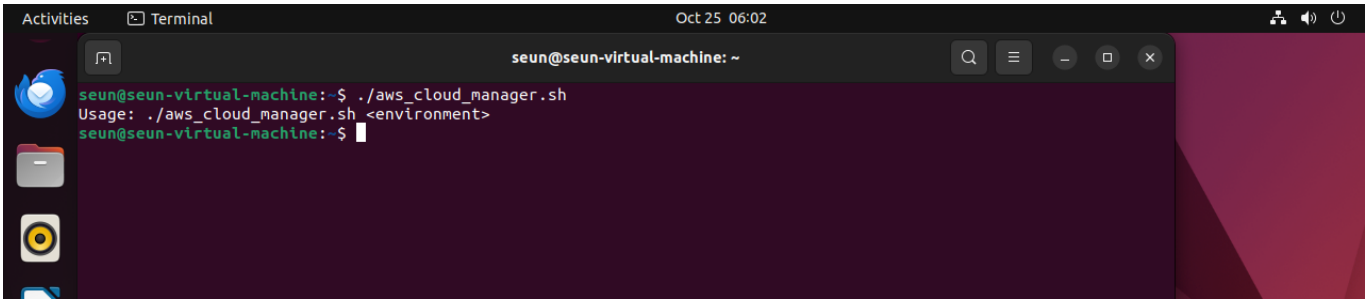

```
if [ "$#" -ne 1 ]; then
    echo "Usage: $0 <environment>"
    exit 1
fi

ENVIRONMENT=$1
```

> $# = number of arguments
> $0 = script name
> $1 = first argument

1.6.2 Test Error Cases

No arguments:

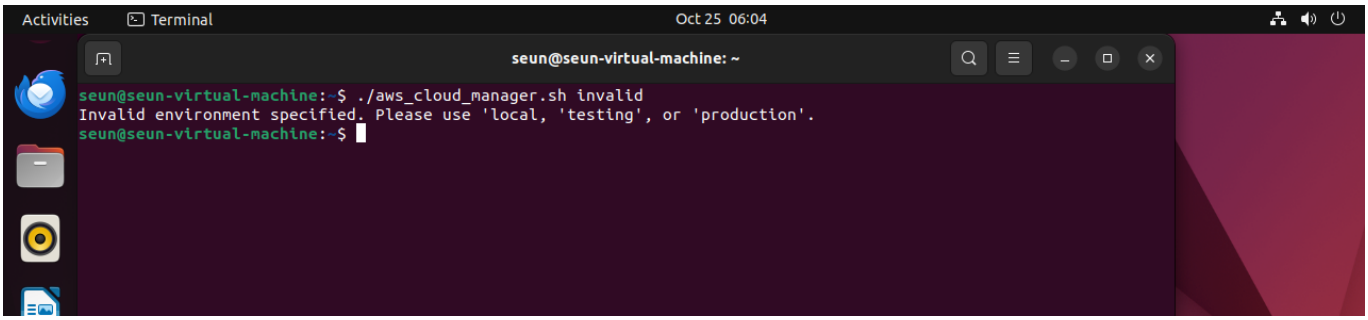
```
./aws_cloud_manager.sh
```



Output: `Usage: ./aws_cloud_manager.sh <environment>`

Invalid environment:

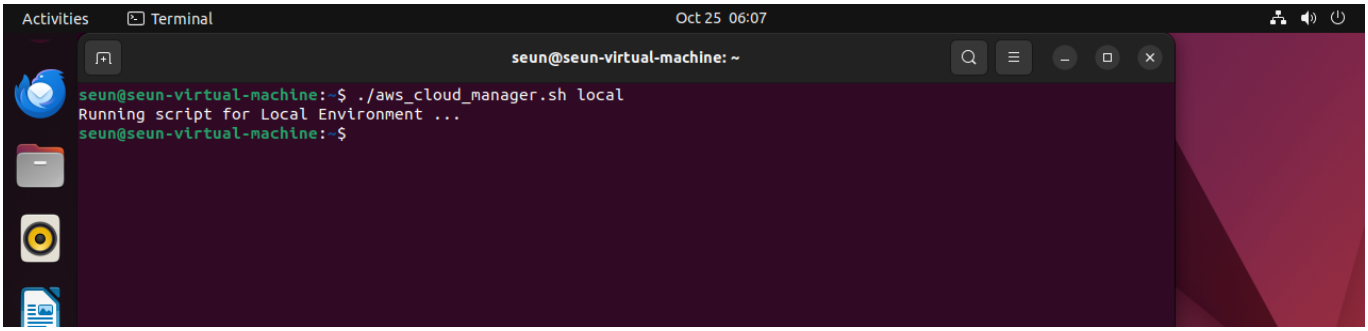
```
./aws_cloud_manager.sh invalid
```



Output: `Invalid environment specified. Please use 'local', 'testing', or 'production`

Valid environment (local):

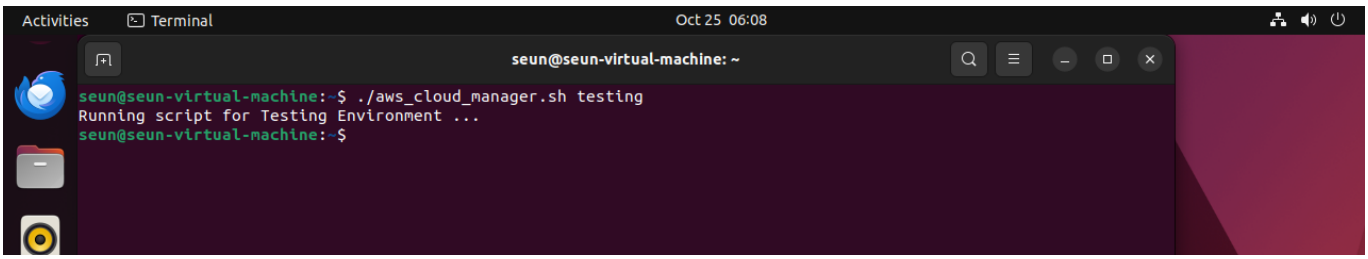
```
./aws_cloud_manager.sh local
```



Output: `Running script for Local Environment...`

Valid environment (testing):

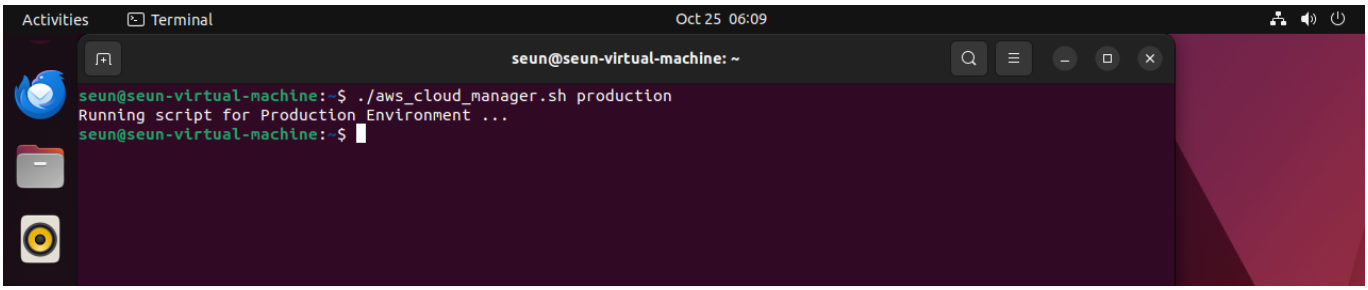
```
./aws_cloud_manager.sh testing
```



Output: `Running script for Testing Environment...`

Valid environment (production):

```
./aws_cloud_manager.sh production
```



Output: `Running script for Testing Environment...`

1.6.3 Final Beginner Script (With Validation)

```
#!/bin/bash

# Validate number of arguments
if [ "$#" -ne 1 ]; then
    echo "Usage: $0 <environment>"
    exit 1
fi

ENVIRONMENT=$1

# Act based on environment
if [ "$ENVIRONMENT" == "local" ]; then
    echo "Running script for Local Environment..."
elif [ "$ENVIRONMENT" == "testing" ]; then
    echo "Running script for Testing Environment..."
elif [ "$ENVIRONMENT" == "production" ]; then
    echo "Running script for Production Environment..."
else
    echo "Invalid environment specified. Please use 'local', 'testing', or
'production'."
    exit 2
fi
```

Phase 1 Summary

Concept	Learned
Infrastructure Environments	Isolated stages (local, testing, production)
Environment Variables	Dynamic config via `export`
Hard-Coding	Anti-pattern — reduces reusability
Positional Parameters	`$1`, `$#` for runtime input
Input Validation	Prevent bugs with argument checks

Concept	Learned
Exit Codes	`exit 1` (usage), `exit 2` (invalid input)

Next: In **Phase 2**, we extend this script to accept **two arguments** (`environment` + `number_of_instances`), integrate **AWS CLI**, and provision **real EC2 instances** using **IAM profiles**.

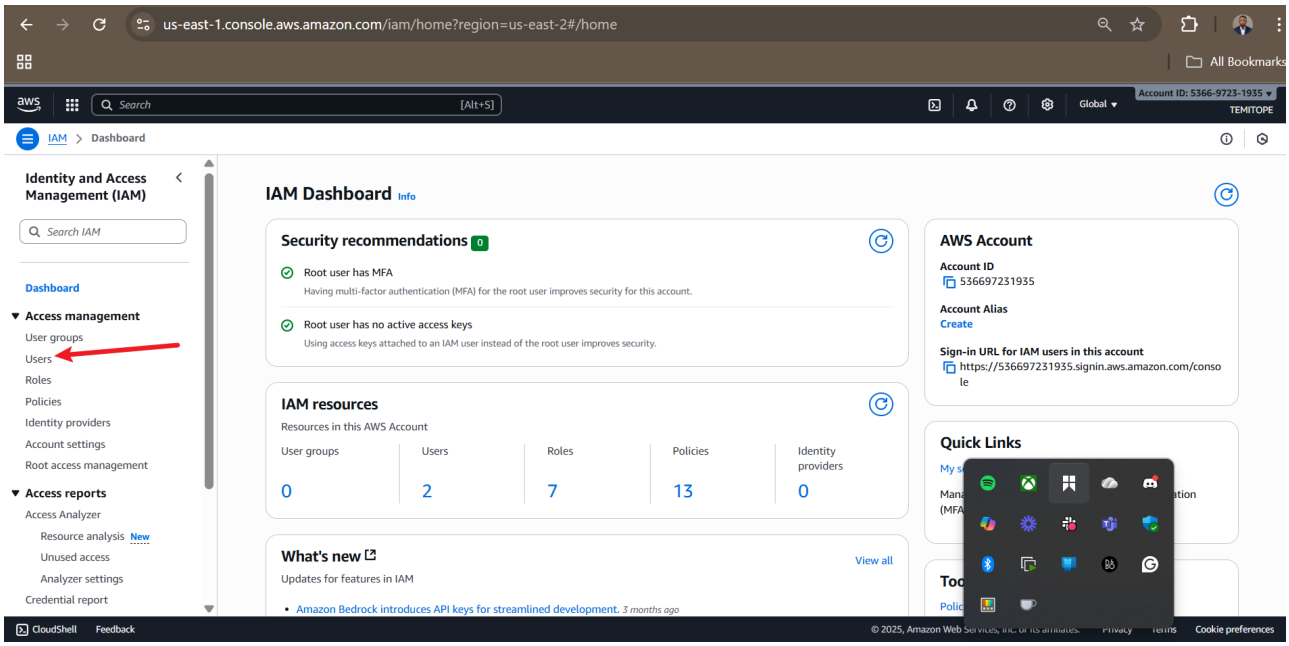
Phase 2: Advanced Implementation (IAM Users + AWS Integration)

Section 2: Secure AWS Account Setup

3.1 Testing Account IAM User Creation (AWS Account 1)

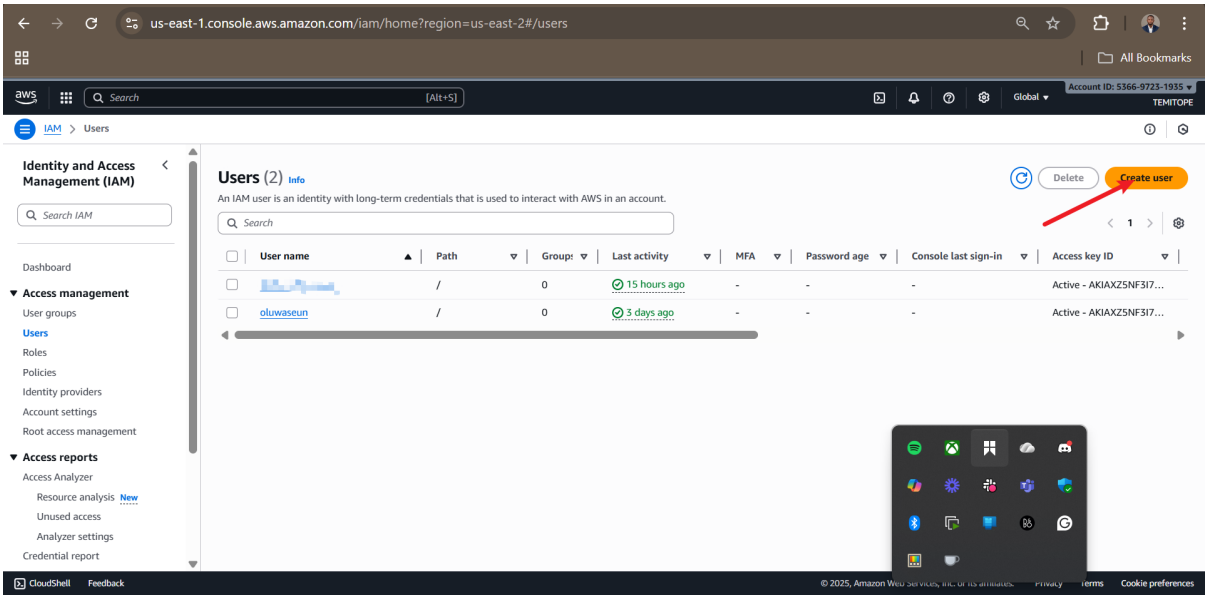
Step 1: Navigate to IAM Dashboard and Click Users

- Login to AWS Account 1 (Another account you have), navigate to IAM dashboard and click on users



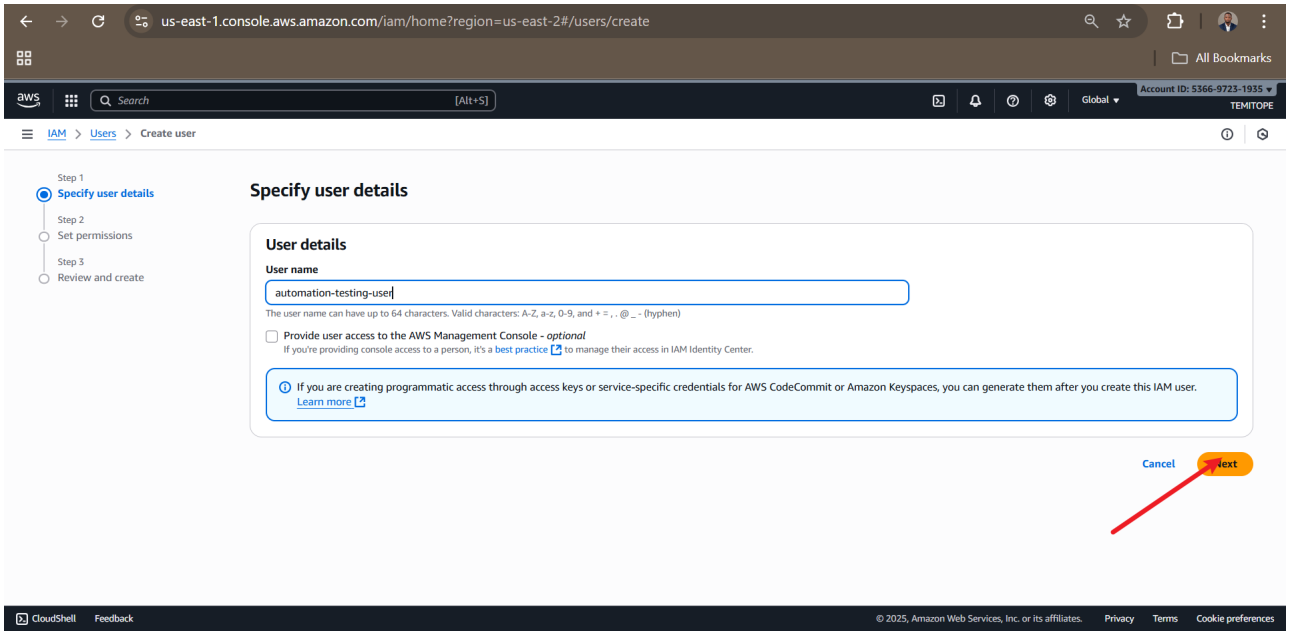
2. **Click Create User**

- On Users page click on create users



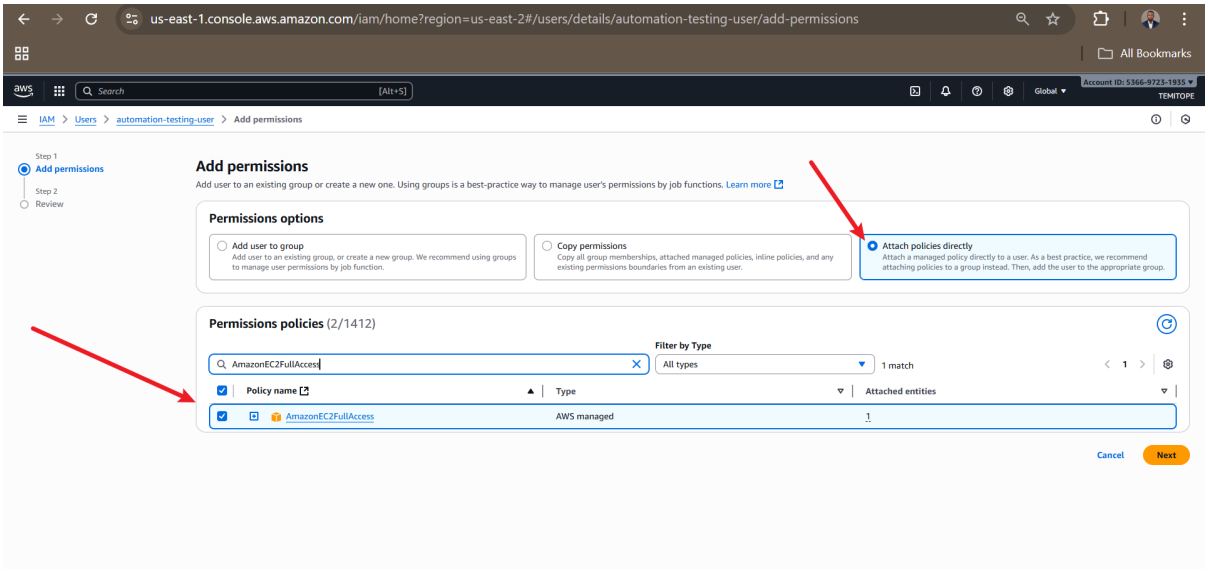
3. **Enter User Name:**

- Enter User Name and Click Next

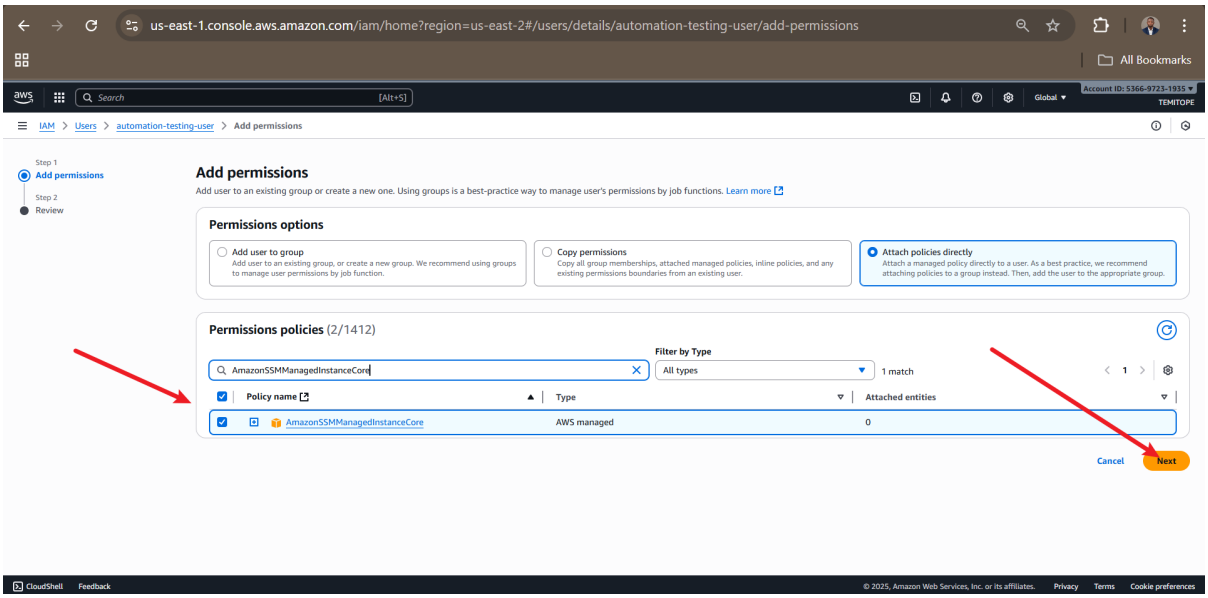


1. **Attach Policies Directly**

- Select "Attach Policies Directly", search and select AmazonEC2FullAccess

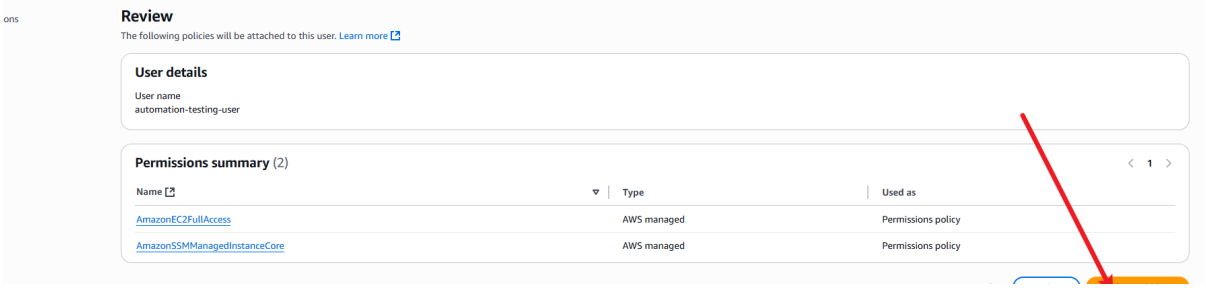


- Also Search and select AmazonSSMManagedInstanceCore



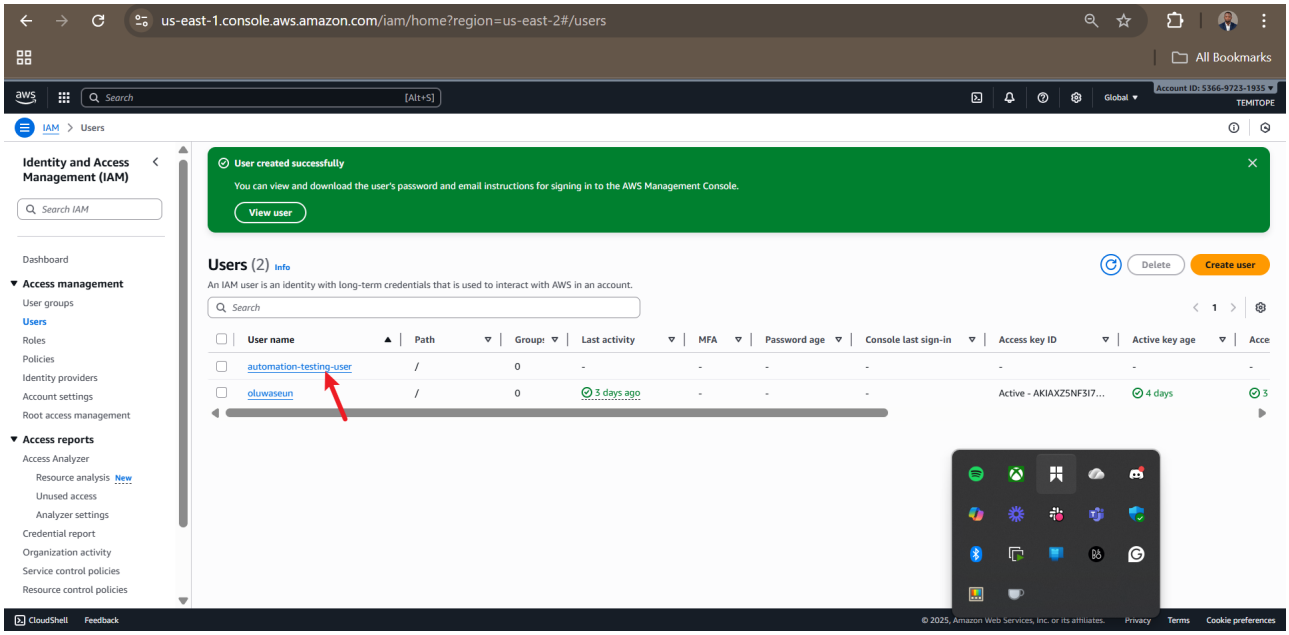
2. **Review & Create**

- Review all the details set and click create user if everything is perfect.



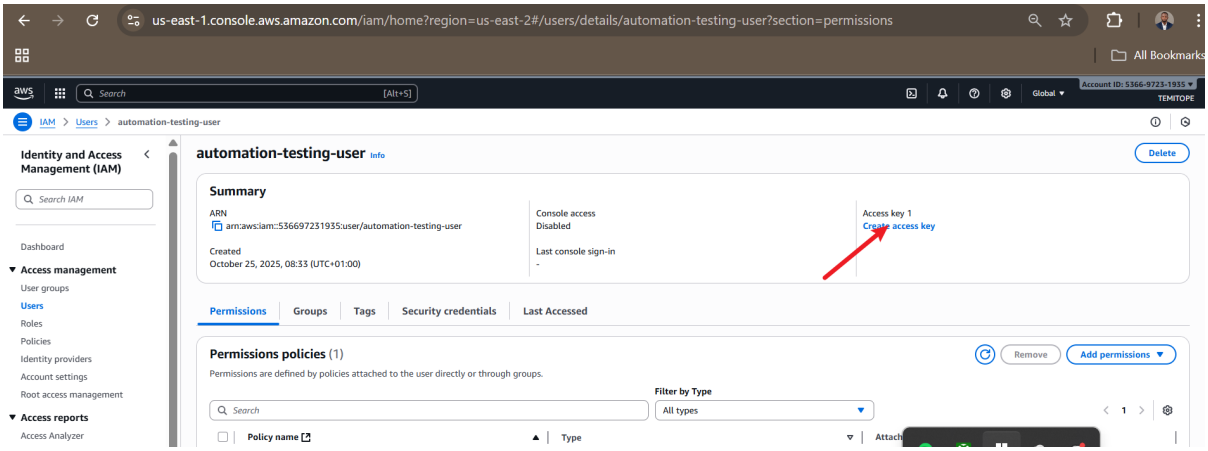
3. **User Created Successfully**

- automation-testing-user created successfuly click on automation-testing-user in the Users list.

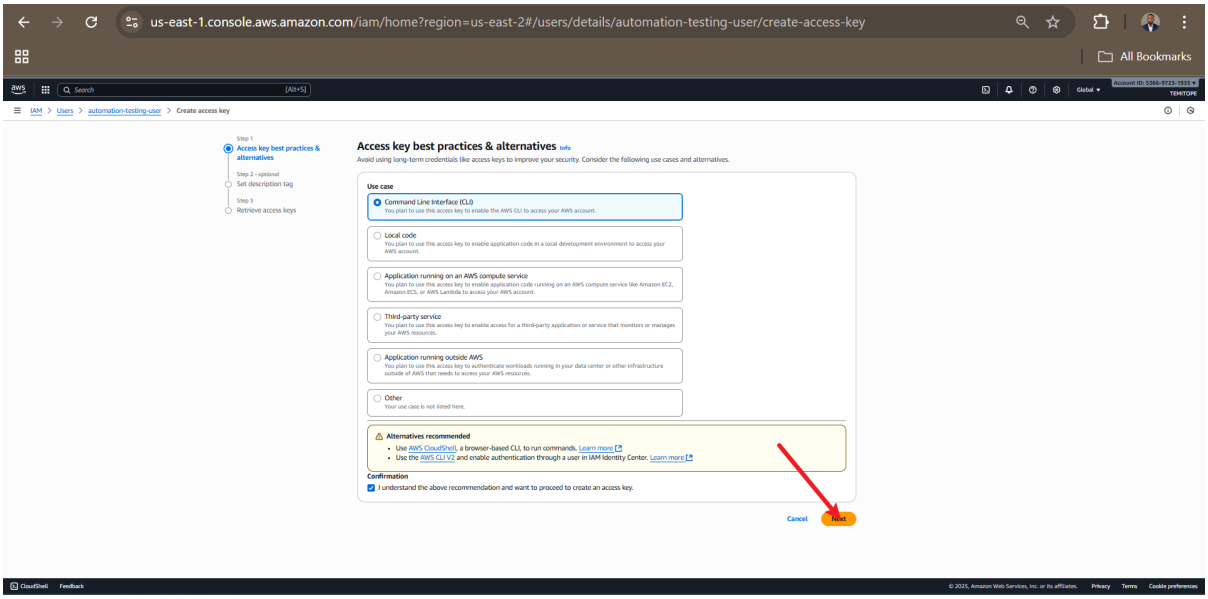


1. **Click create Access Key**

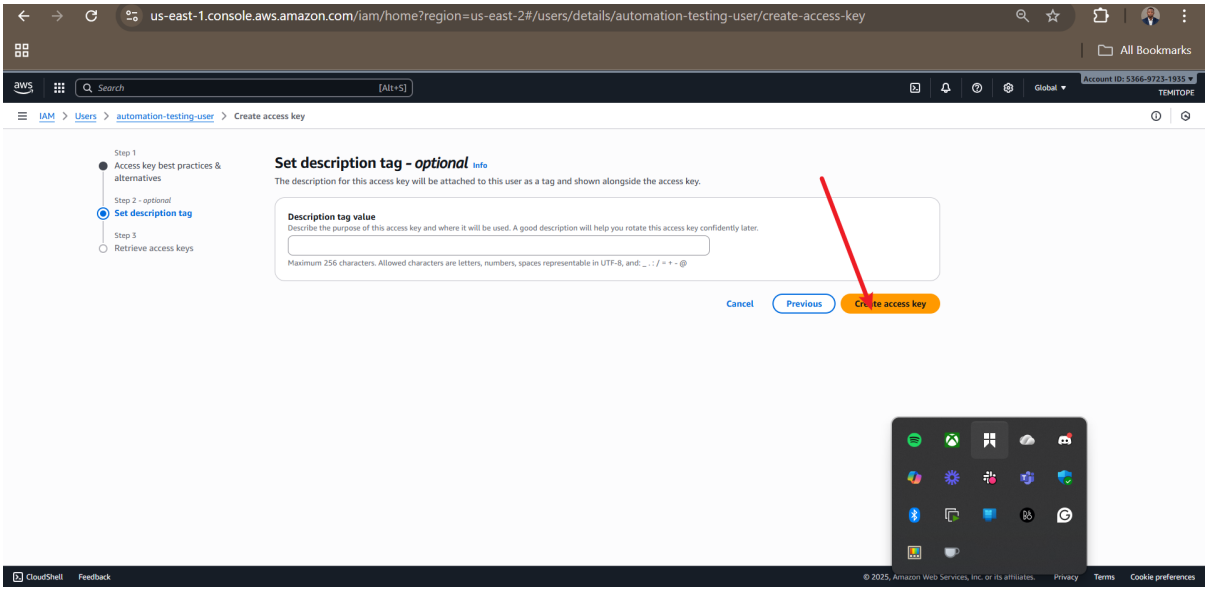
   - On automation-testing-user dashboard, click create access key.



   - Select Command Line Interface (CLI) and check the box to acknowledge understanding the recommendation and click next.

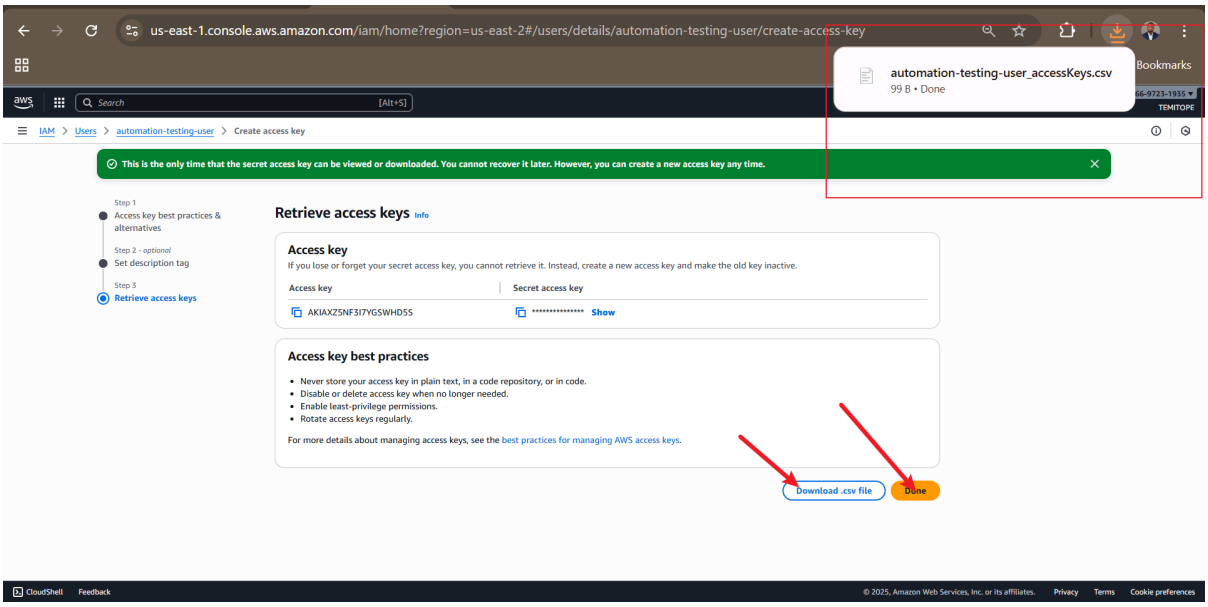


- Skip tag description and click create access key.

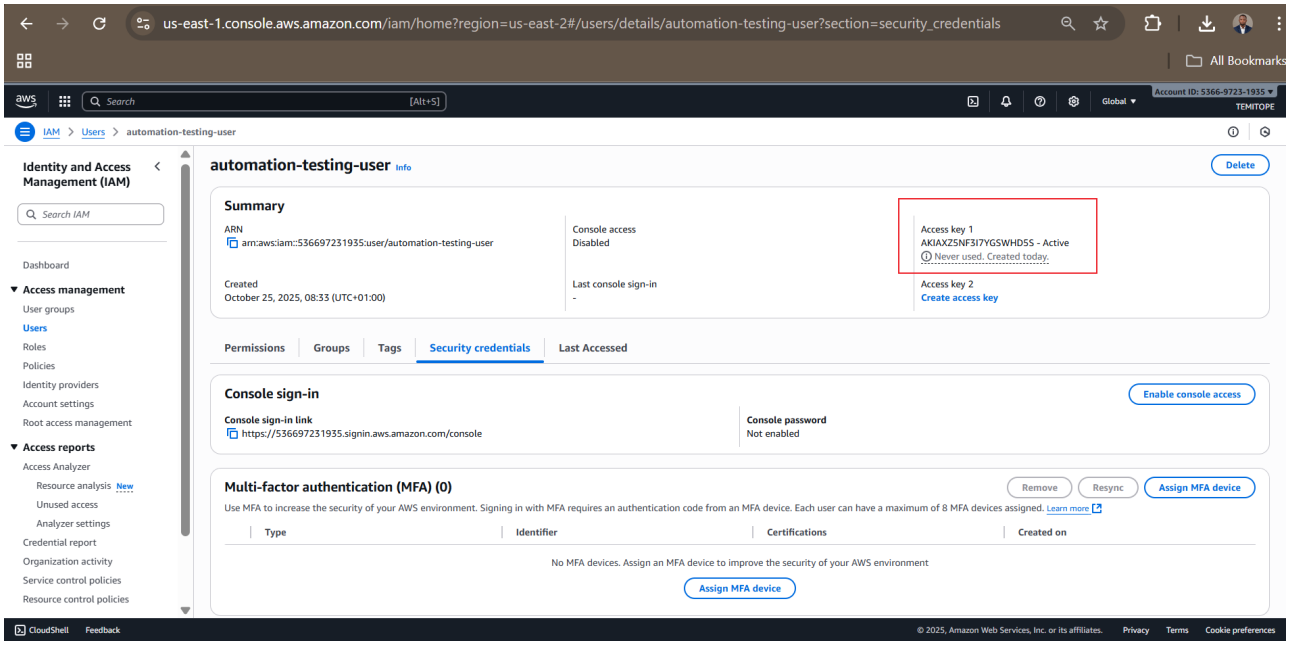


2. **Download Credentials**

- Access key created now download and click, Done.



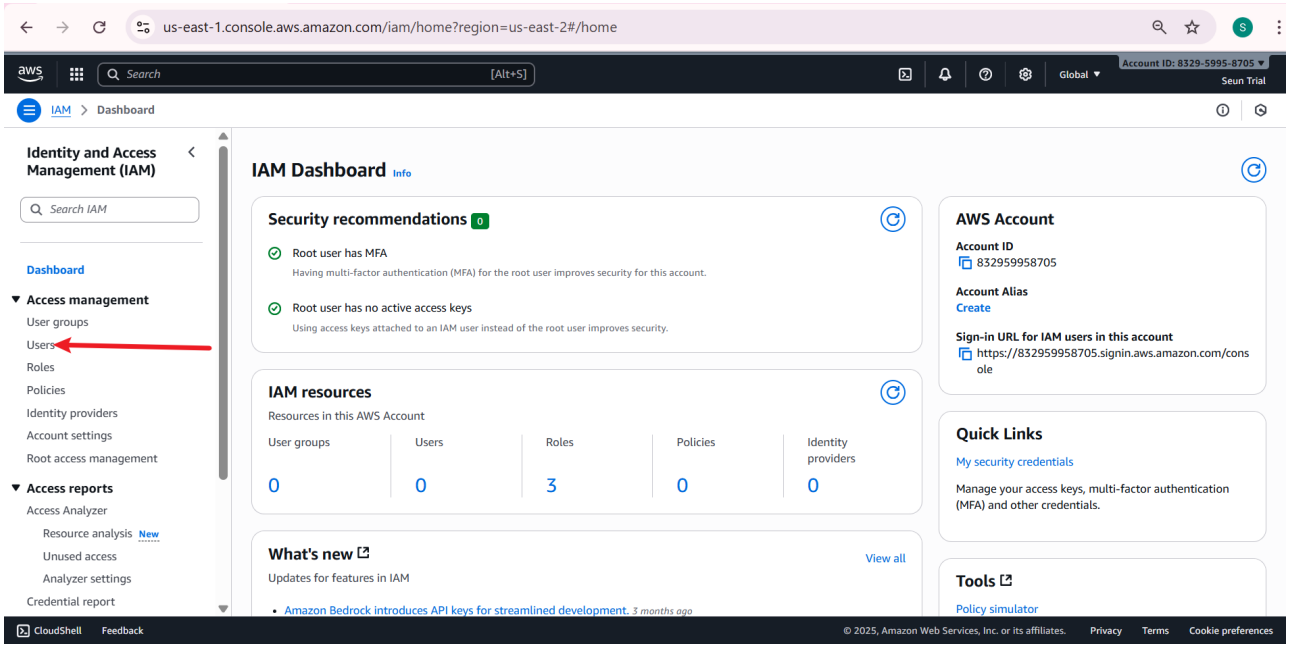
- Notice on the dashboard that access key isnow created but not used.



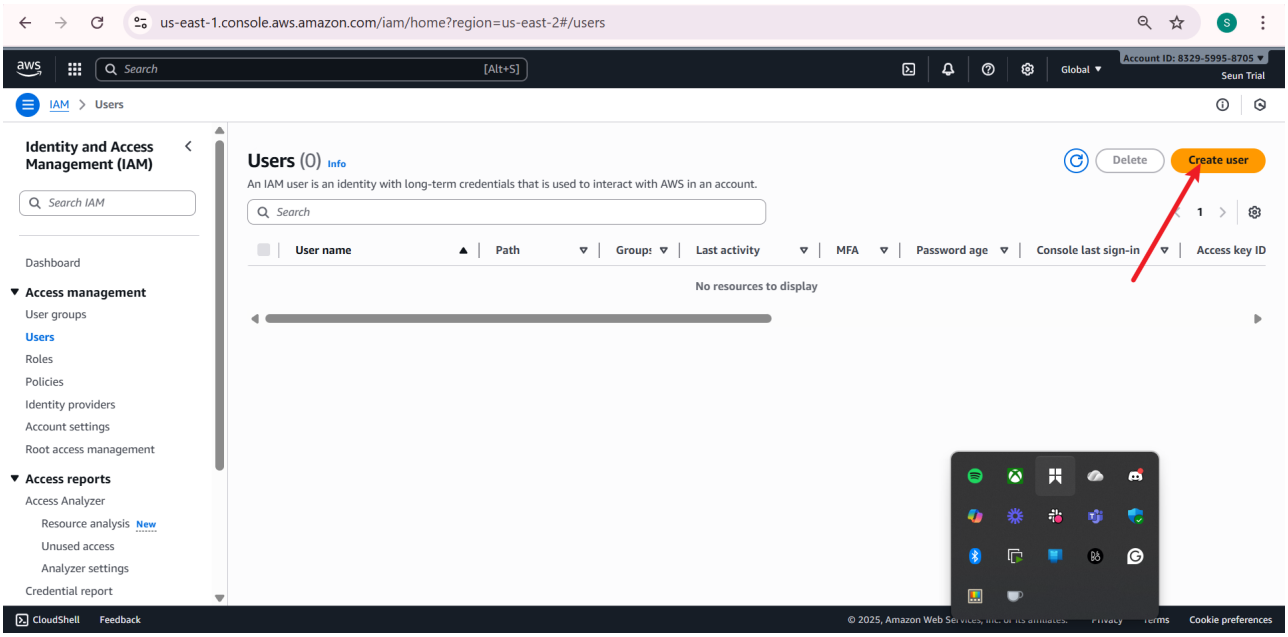
3.2 Production Account IAM User Creation (AWS Account 2)

Same process for `automation-production-user`:

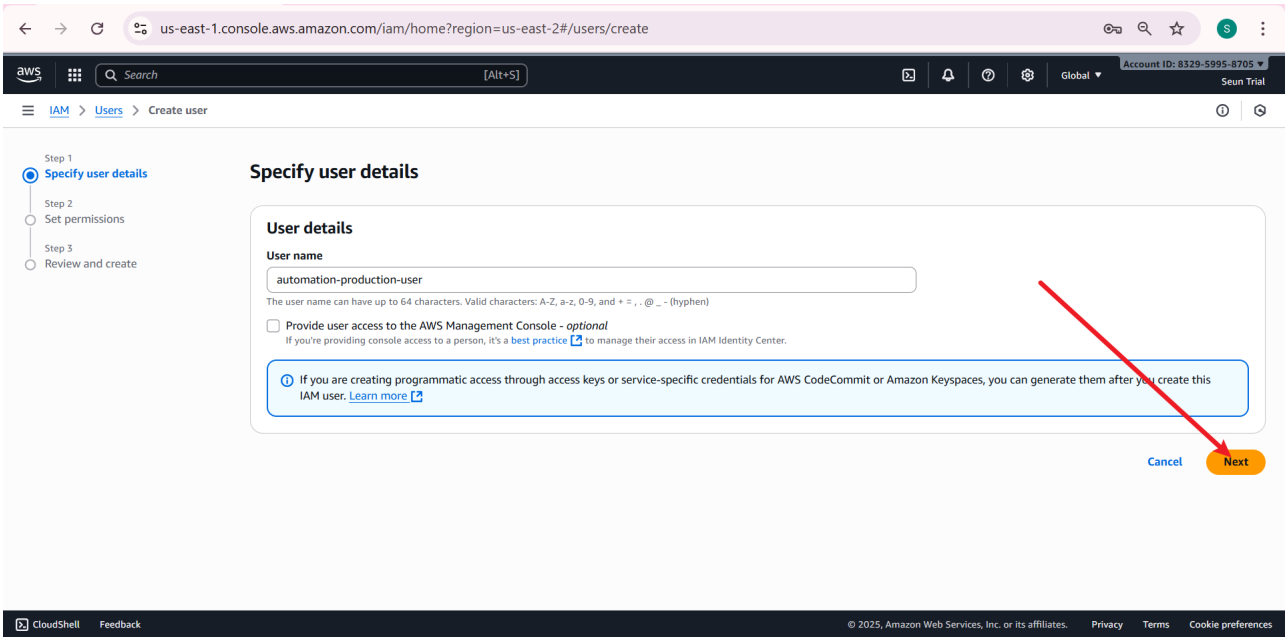
- Switch to AWS Account 2 (Another account you have), navigate to IAM dashboard and click on users



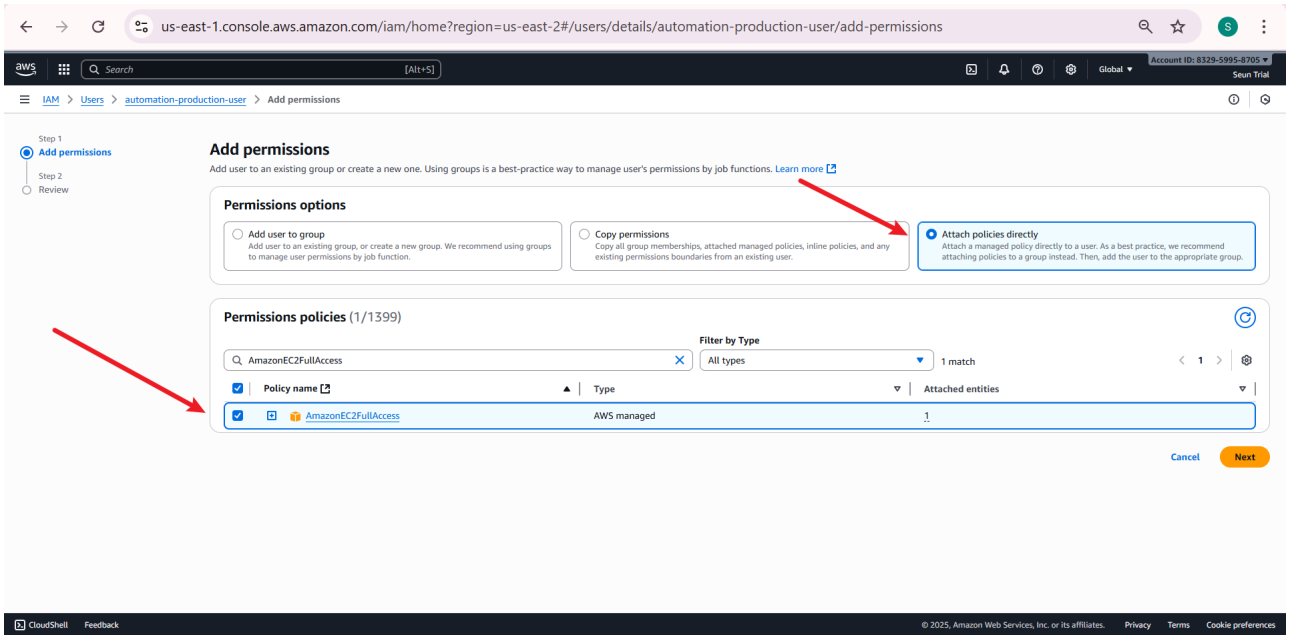
- On users dashboard, click on create user.



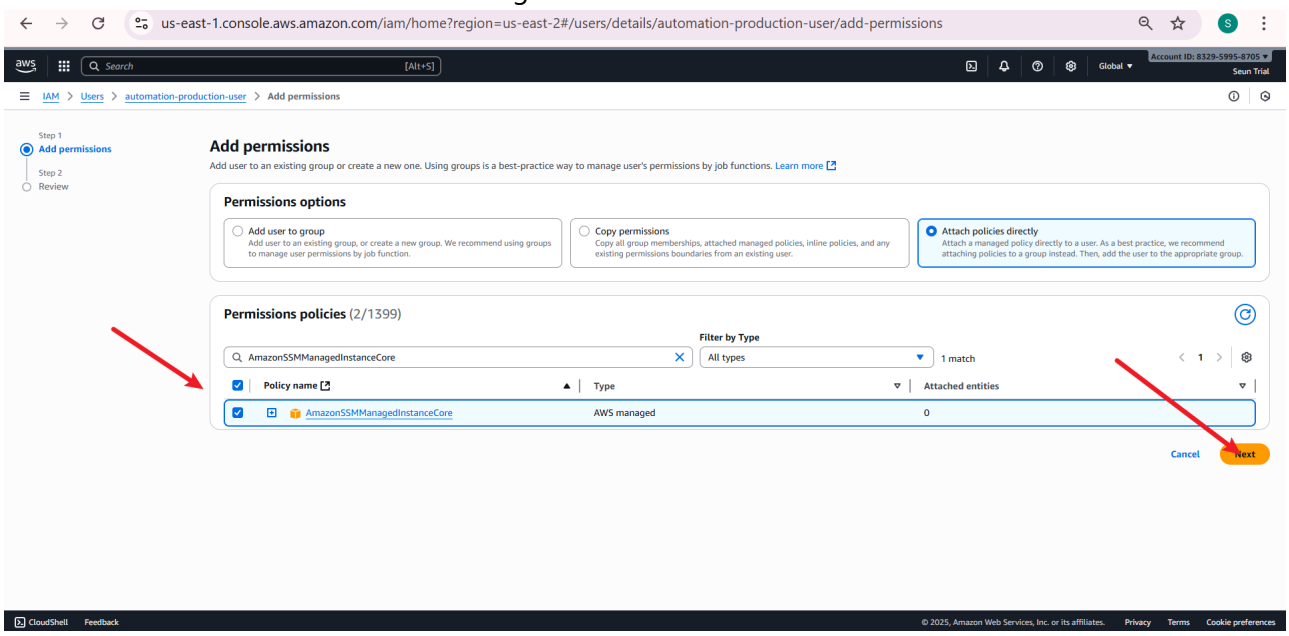
- Name user automation-production-user and click next



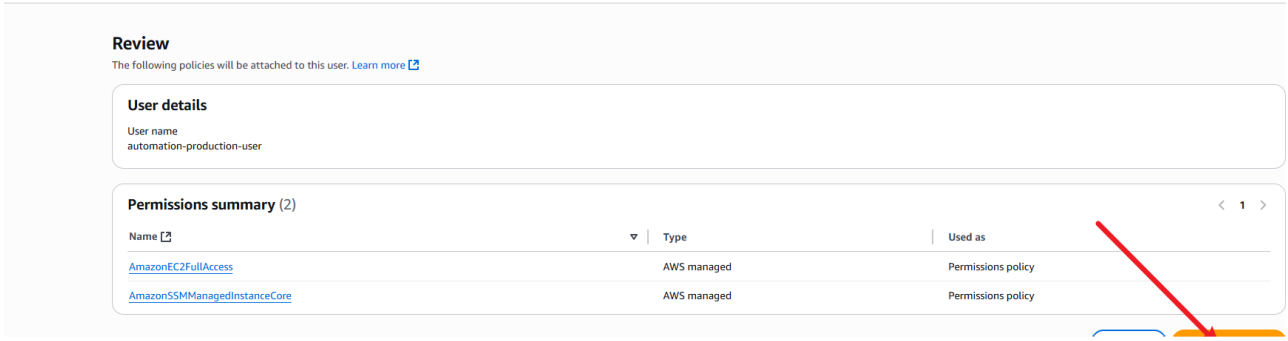
- Select "Attach Policies Directly", search and select AmazonEC2FullAccess



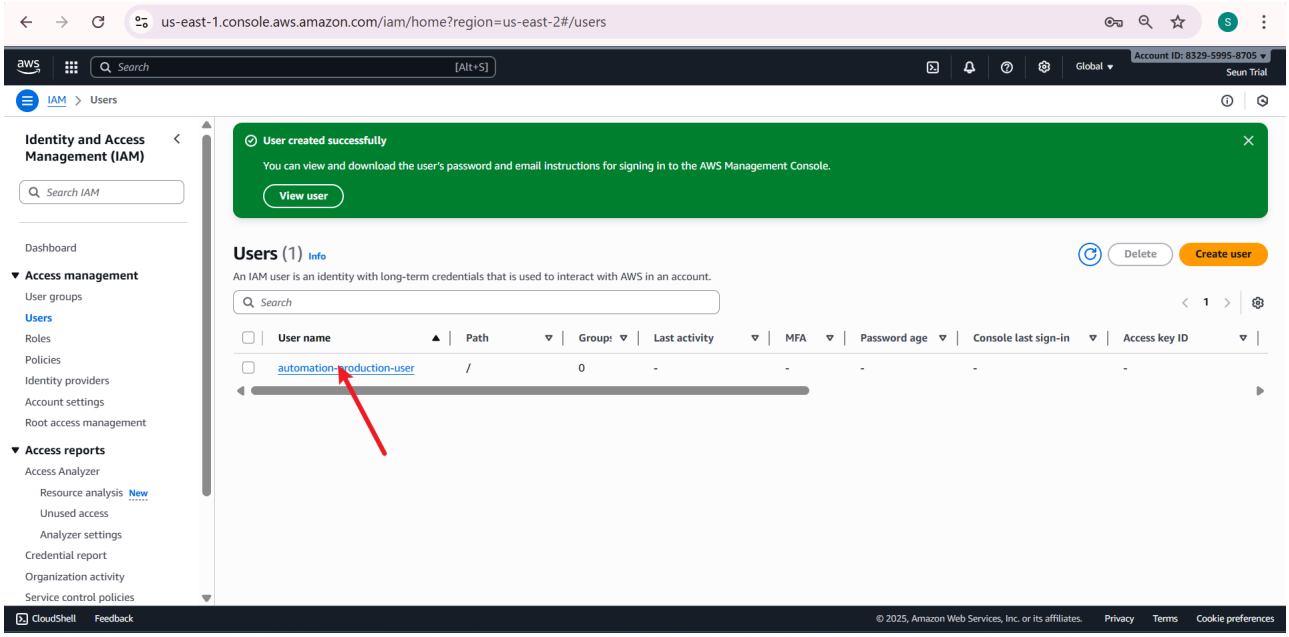
- Also Search and select AmazonSSMManagedInstanceCore then click next



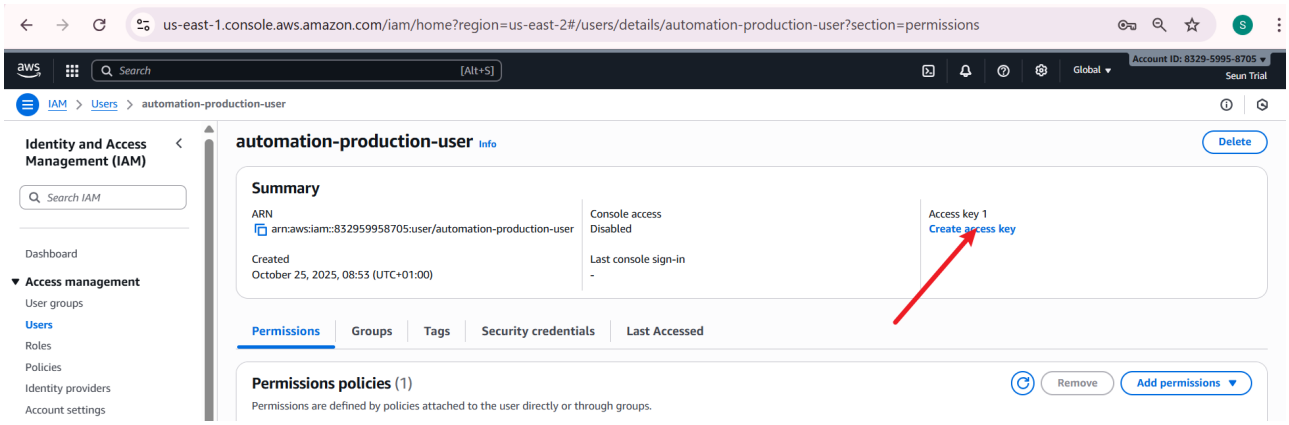
- Review all the details set and click create user if everything is perfect.



- automation-production-user created successfuly click on automation-production-user in the Users list.

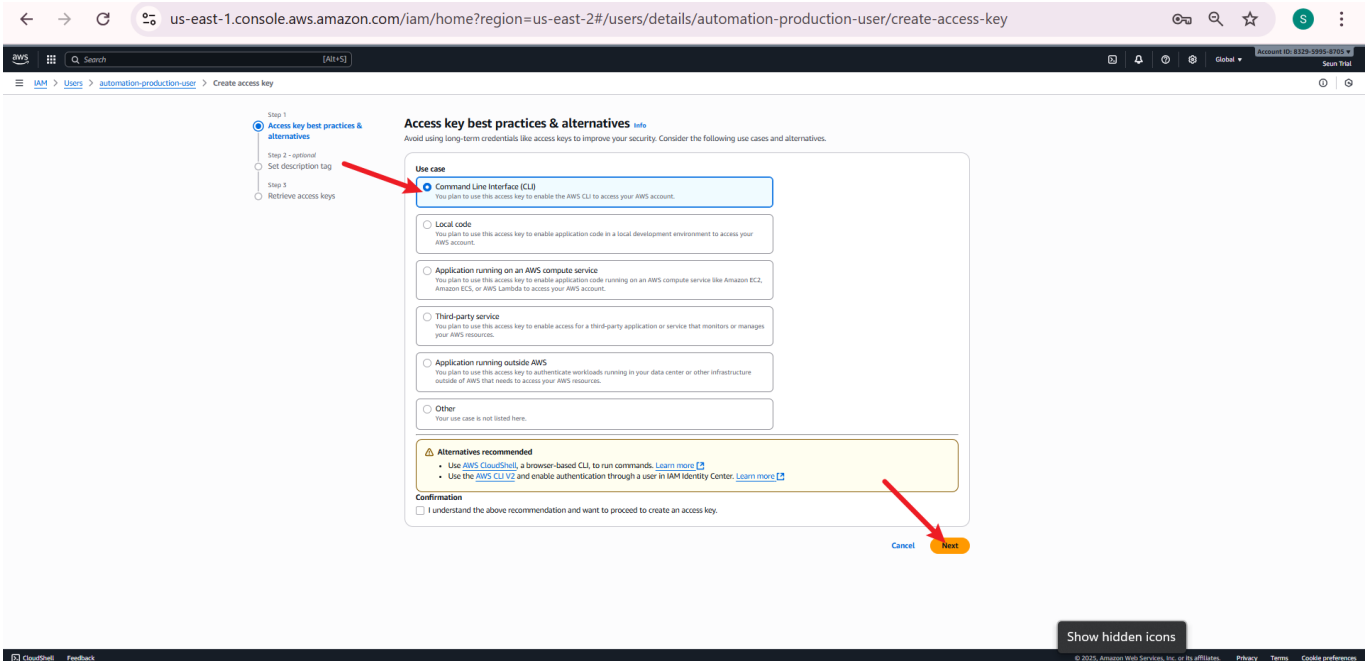


- On automation-production-user click create access key

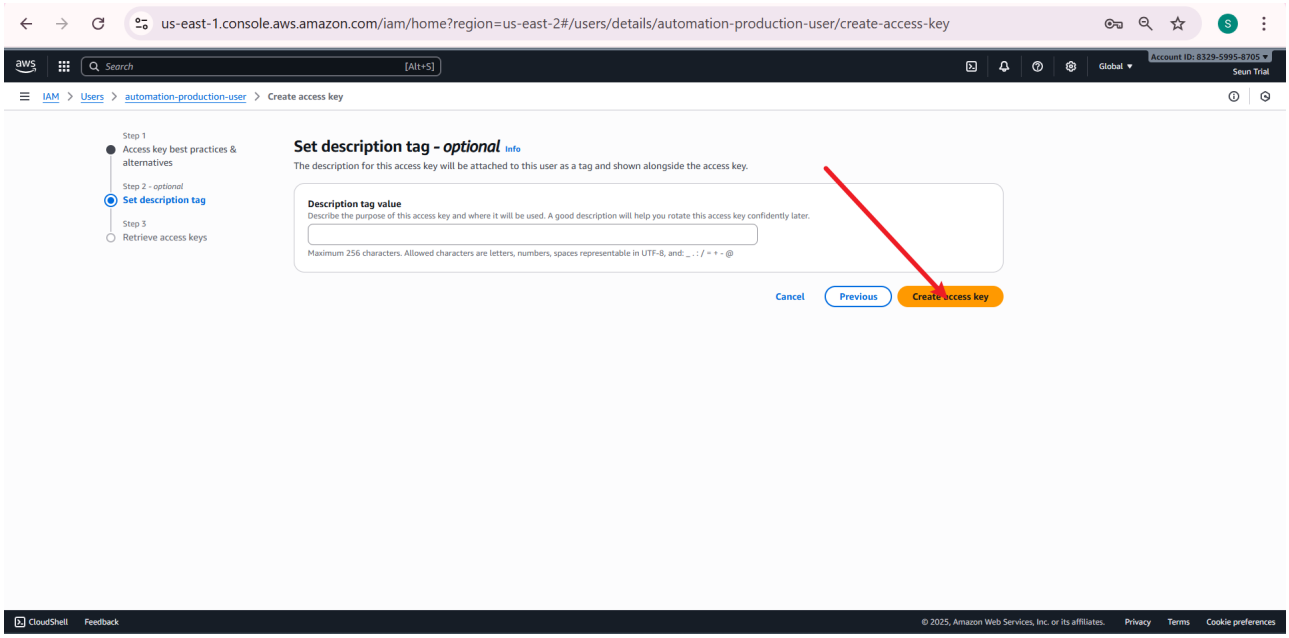


- Select Command Line Interface (CLI) and check the box to acknowledge understanding the recommendation and click next.

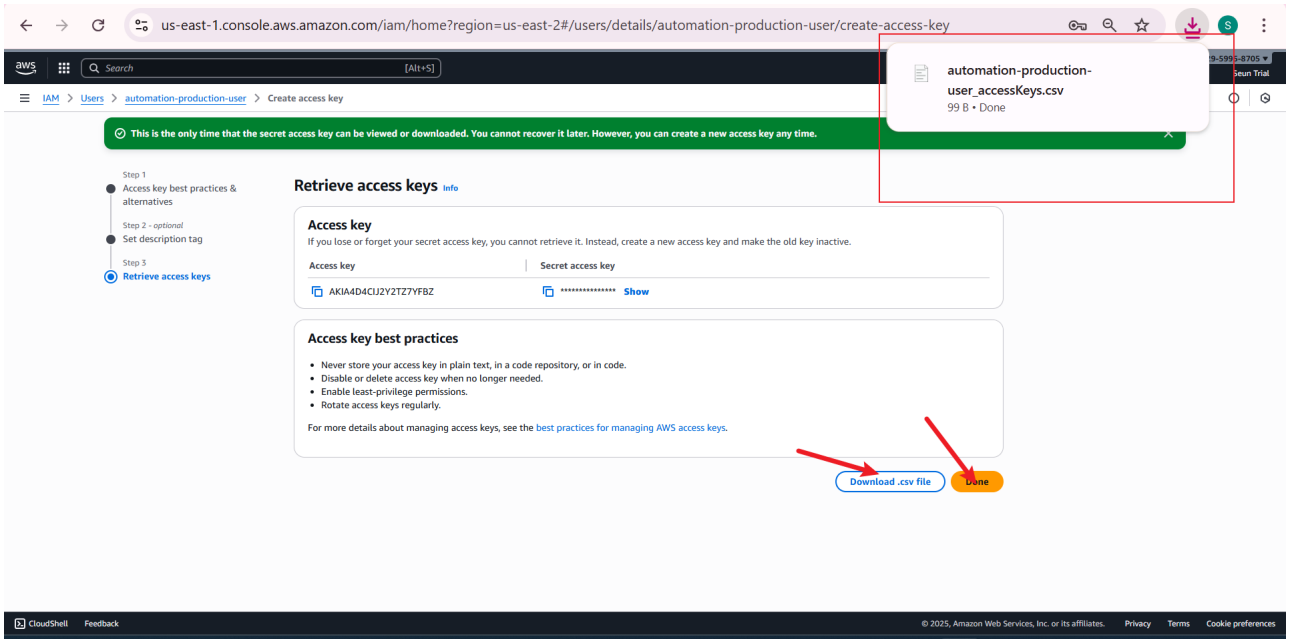
-



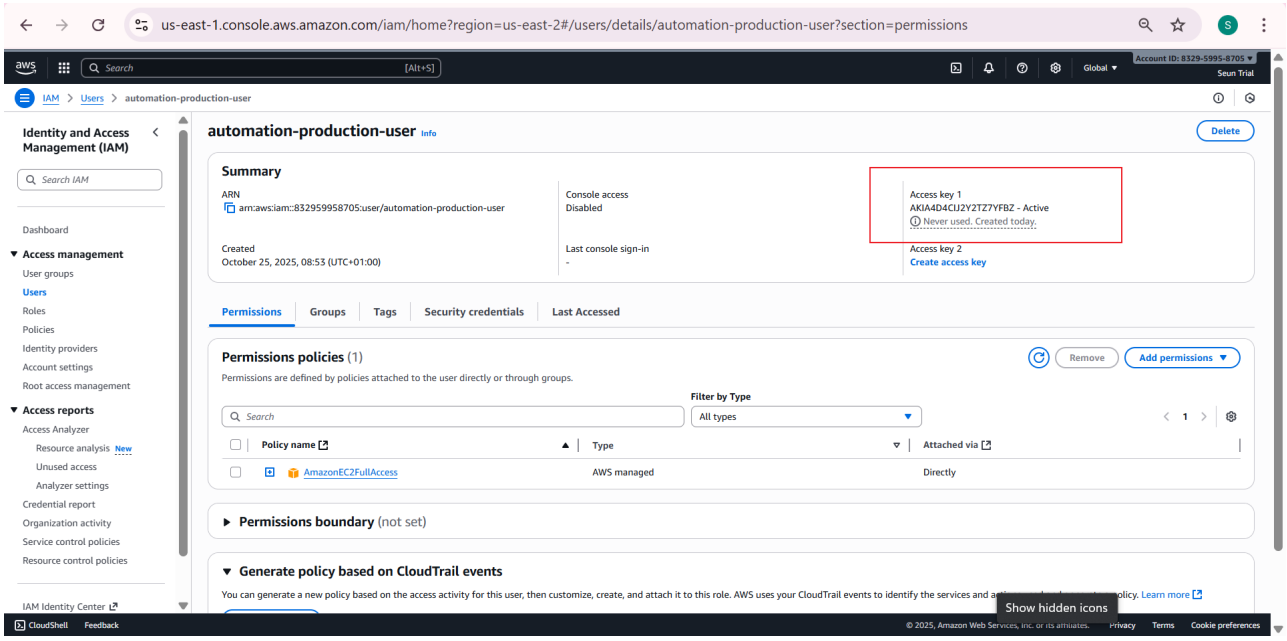
- Skip tag description and click create access key.



- Access key created now download and click, Done.



- Notice on the dashboard that access key isnow created but not used.

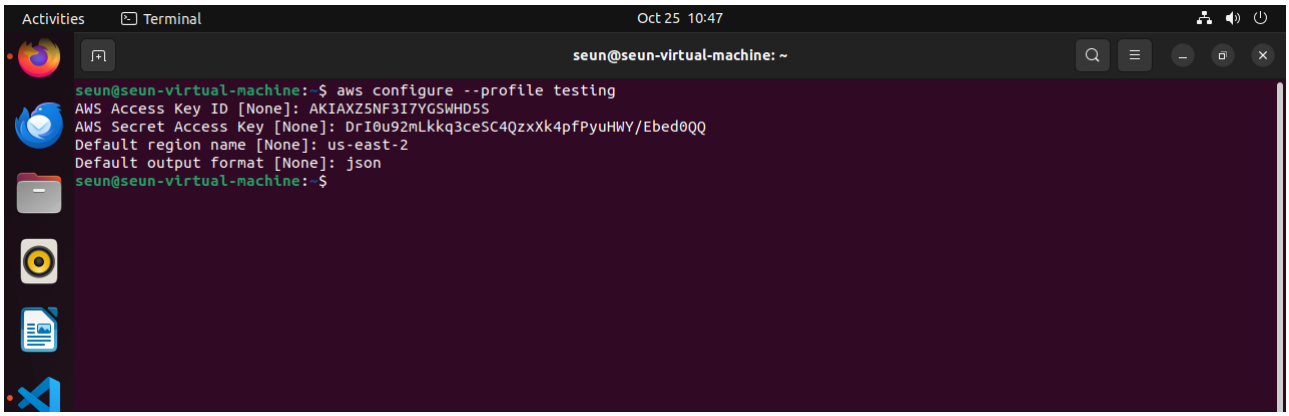


3.3 AWS CLI Profile Configuration

Testing Profile:

```
aws configure --profile testing
```

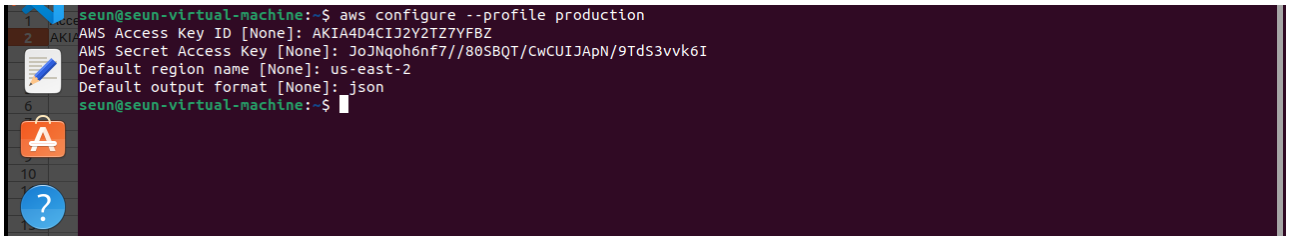
- The response will be a prompt asking for the create Access key and Secret Key as well as default region and default format. Once everything



Production Profile:

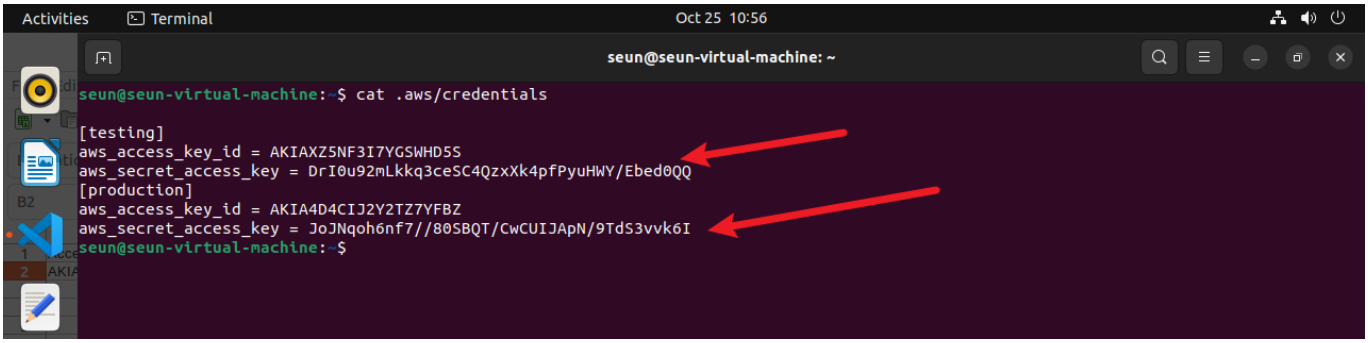
```
aws configure --profile production
```

- The response will be a prompt asking for the create Access key and Secret Key as well as default region and default format. Once everything



Verify Configuration:

```
cat ~/.aws/credentials
```



Section 3: Advanced Script Implementation

4.1 Enhanced Script Development and Explanation

Core Functions: The script has now been enhanced with functions. Some logic that previously wasn't organized into functions has now been refactored into separate ones. Hence, we have the following functions:

```
check_num_of_args
activate_infra_environment
```

```
check_aws_cli
check_aws_profile
create_local_testing_or_prod_resources
```

> **Note**: Functions that have not yet been utilized or described will be explained in subsequent parts of this section.

AWS CLI & Profile Checks:

```
check_aws_cli() {
    if ! command -v aws &> /dev/null; then
        echo "AWS CLI is not installed. Please install it before proceeding."
        exit 1
    fi
}

# Function to check if AWS profile is set
check_aws_profile() {
    if [ -z "$AWS_PROFILE" ] && [ "$ENVIRONMENT" != "local" ]; then
        echo "AWS profile environment variable is not set for $ENVIRONMENT
environment."
        exit 1
    fi
}
```

> **Note** The above code is not the complete script.

check_aws_cli(): Verifies that the AWS CLI is installed on the system. If not found, it displays an error message and exits the script.

check_aws_profile(): Ensures the `AWS_PROFILE` environment variable is set for non-local environments. If missing, it alerts the user and exits to prevent unauthorized or failed AWS operations.

Resource Creation Logic:

```
# Function to create local log files or EC2 instances
create_local_testing_or_prod_resources() {
    # Validate number_of_instances is a positive integer
    if ! [[ "$NUMBER_OF_INSTANCES" =~ ^[1-9][0-9]*$ ]]; then
        echo "Invalid number_of_instances: Must be a positive integer."
        exit 2
    fi

    if [ "$ENVIRONMENT" == "local" ]; then
        echo "Creating $NUMBER_OF_INSTANCES sample log files locally..."
        for ((i=1; i<=NUMBER_OF_INSTANCES; i++)); do
            echo "Log entry $i for local simulation" > "log_file_$i.txt"
            if [ $? -eq 0 ]; then
                echo "☑ Created log_file_$i.txt successfully."
            else
                echo "❌ Failed to create log_file_$i.txt."
            fi
        done
    else

        # Common parameters
        ...
```

> **Note** The above code snippet if incomplete. It won't work

create_local_testing_or_prod_resources(): This function creates resources based on the selected environment.

- It first checks that `NUMBER_OF_INSTANCES` is a valid positive integer.
- If the environment is **local**, it simulates resource creation by generating sample log files (`log_file_1.txt`, `log_file_2.txt`, etc.).
- If the environment is **testing** or **production**, it prepares configuration parameters (like instance type, AMI ID, region, and key pairs) needed to launch EC2 instances using AWS CLI.
- It ensures the setup process matches the environment — lightweight for local testing and actual cloud provisioning for AWS environments.

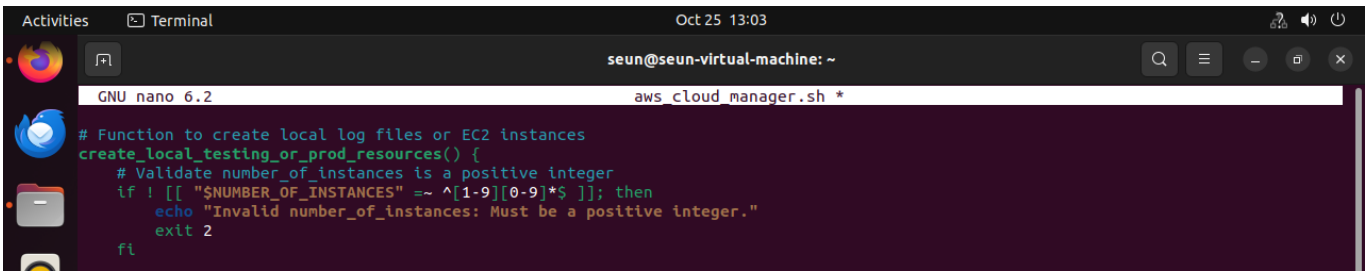
Input Validation:

```
if ! [[ "$NUMBER_OF_INSTANCES" =~ ^[1-9][0-9]*$ ]]; then
        echo "Invalid number_of_instances: Must be a positive integer."
        exit 2
    fi
```

> **Note** The above code snippet if incomplete.

This block checks whether `NUMBER_OF_INSTANCES` is a **positive integer**.

- The expression `[[ "$NUMBER_OF_INSTANCES" =~ ^[1-9][0-9]*$ ]]` uses a **regular expression** to match numbers starting from 1 and containing only digits.
- The `!` negates the test, meaning *"if the value does **not** match this pattern."*
- If invalid, it prints an error message and exits the script with **status code 2** to stop execution.



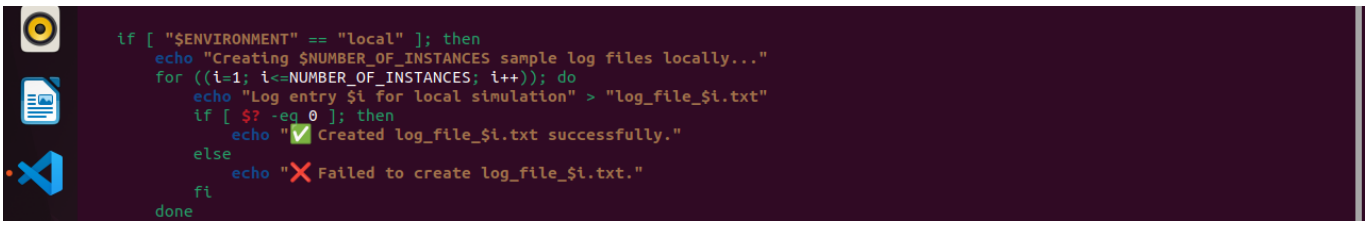
Local Environment:

```
if [ "$ENVIRONMENT" == "local" ]; then
        echo "Creating $NUMBER_OF_INSTANCES sample log files locally..."
        for ((i=1; i<=NUMBER_OF_INSTANCES; i++)); do
            echo "Log entry $i for local simulation" > "log_file_$i.txt"
            if [ $? -eq 0 ]; then
                echo "☑ Created log_file_$i.txt successfully."
            else
                echo "❌ Failed to create log_file_$i.txt."
            fi
        done
    else
```

> **Note** The above code snippet if incomplete. It won't work.

This block handles the **local environment setup**.

- It checks if `ENVIRONMENT` equals `"local"`.
- If true, it creates the specified number of sample log files (`log_file_1.txt`, `log_file_2.txt`, etc.).
- Each file contains a simple line of text for simulation (`"Log entry i for local simulation"`).
- After each creation, it checks the command's exit status (`$?`) — printing a success ☑ or failure ❌ message accordingly.
- If the environment isn't local, control passes to the `else` block for cloud (AWS) resource creation.



AWS Parameters:

```
        # Common parameters
        instance_type="t2.micro"    # for testing environment
        instance_type2="t3.micro"   #for production environment
        ami_id="resolve:ssm:/aws/service/ami-amazon-linux-latest/al2023-ami-
kernel-default-x86_64"
        region="us-east-2"
        # No explicit subnet assignment (let AWS choose default)
        subnet_id=None   # or simply omit this parameter when creating the
instance
        # Create a new Security Group instead of using an existing one
        create_security_group=True  # flag to indicate a new SG should be created
        security_group_name="auto-created-sg"
        security_group_description="Security group automatically created for this
instance"
        key_name="my-key-new-pair" # for testing environment
        key_name2="MyKeyPair1"      #for production environment
```

This section defines **common configuration parameters** for launching AWS EC2 instances.

- `instance_type` and `instance_type2` specify instance sizes — `t2.micro` for **testing** and `t3.micro` for **production**.
- `ami_id` references the **latest Amazon Linux 2023 image** via AWS Systems Manager (SSM).
- `region` sets the deployment location (`us-east-2`).
- `subnet_id` is set to `None`, letting AWS automatically choose a default subnet.
- `create_security_group=True` indicates a new **security group** should be created with a name and description provided.
- `key_name` and `key_name2` define the **SSH key pairs** for connecting to instances in testing and production respectively.

In short, these variables standardize AWS deployment settings across environments.

```
# Common parameters
instance_type="t2.micro"    # for testing environment
instance_type2="t3.micro"   #for production environment
ami_id="resolve:ssm:/aws/service/ami-amazon-linux-latest/al2023-ami-kernel-default-x86_64"
region="us-east-2"
# No explicit subnet assignment (let AWS choose default)
subnet_id=None   # or simply omit this parameter when creating the instance
# Create a new Security Group instead of using an existing one
create_security_group=True  # flag to indicate a new SG should be created
security_group_name="auto-created-sg"
security_group_description="Security group automatically created for this instance"
key_name="my-key-new-pair" # for testing environment
key_name2="MyKeyPair1"      #for production environment

echo "Launching $NUMBER_OF_INSTANCES EC2 instances in $ENVIRONMENT subnet: $subnet_id..."
```

Testing EC2 Launch:

```
        echo "Launching $NUMBER_OF_INSTANCES EC2 instances in $ENVIRONMENT subnet:
$subnet_id..."

        # Launch EC2 instances with profile based on environment
        if [ "$ENVIRONMENT" == "testing" ]; then
            aws ec2 run-instances --profile testing \
                --image-id "$ami_id" \
                --instance-type "$instance_type" \
                --count "$NUMBER_OF_INSTANCES" \
                --key-name "$key_name" \
                --region "$region" \
                --associate-public-ip-address \
                --tag-specifications "ResourceType=instance,Tags=
[{Key=Environment,Value=$ENVIRONMENT}]"
```

> **Note** The above code snippet if incomplete. It won't work.

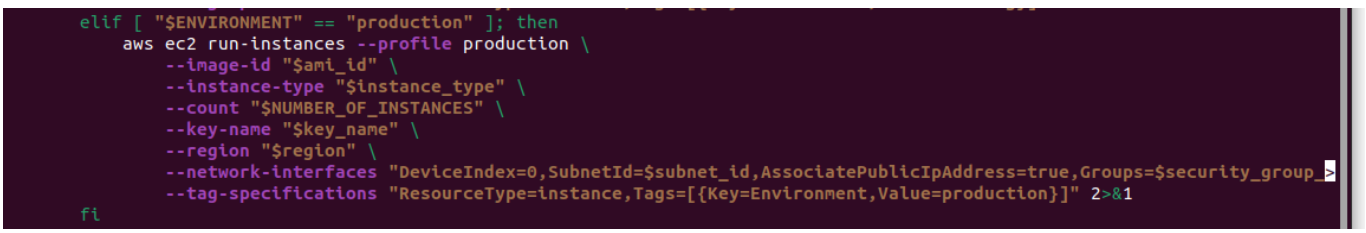
This section launches EC2 instances based on the selected environment. It uses parameters like `--profile` to choose the correct AWS credentials, `--image-id` for the base AMI, `--instance-type` for hardware specs, `--count` for the number of instances, `--key-name` for SSH access, `--region` for deployment location, `--associate-public-ip-address` to enable internet access, and `--tag-specifications` to label instances with environment metadata.

Production EC2 Launch:

```
        elif [ "$ENVIRONMENT" == "production" ]; then
            aws ec2 run-instances --profile production \
                --image-id "$ami_id" \
                --instance-type "$instance_type2" \
                --count "$NUMBER_OF_INSTANCES" \
                --key-name "$key_name2" \
                --region "$region" \
                --associate-public-ip-address \
                --tag-specifications "ResourceType=instance,Tags=
[{Key=Environment,Value=$ENVIRONMENT}]"
        fi
```

> **Note** The above code snippet if incomplete. It won't work. This block handles **production environment deployment**.

- The `elif` checks if the environment is **production**.
- It runs `aws ec2 run-instances` with the **production profile** to ensure the correct AWS account and permissions are used.
- It specifies parameters such as:
  - `--image-id`: defines the AMI to use, same as testing.
  - `--instance-type`: uses `t3.micro`, slightly more powerful than `t2.micro`.
  - `--count`: number of instances to create.
  - `--key-name`: production key pair for secure access.
  - `--region`: AWS region for deployment.
  - `--associate-public-ip-address`: assigns public IPs for external access.
  - `--tag-specifications`: labels instances for easy identification by environment.



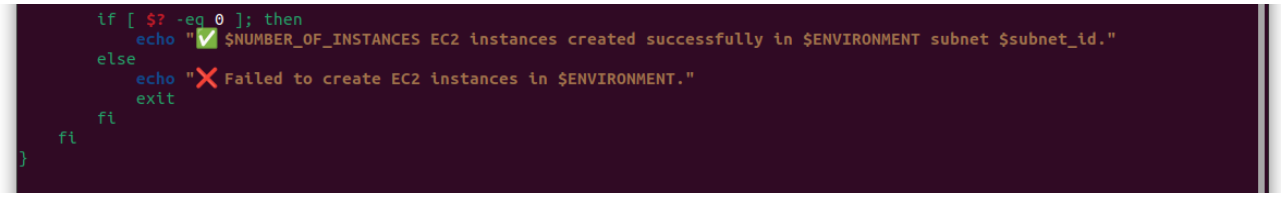
Error Handling:

```
if [ $? -eq 0 ]; then
            echo "✅ $NUMBER_OF_INSTANCES EC2 instances created successfully in
$ENVIRONMENT subnet $subnet_id."
        else
            echo "❌ Failed to create EC2 instances in $ENVIRONMENT."
            exit
        fi
    fi
}
```

> **Note** The above code snippet if incomplete. It won't work

This block **verifies whether the EC2 instance creation command succeeded or failed**:

- `$?` stores the exit status of the last executed command (`0` means success).
- If the status equals `0`, it prints a success message showing how many instances were created and in which environment/subnet.
- If not, it prints an error message indicating failure and exits the script to stop further execution.



Function Execution:

```
check_num_of_args "$@"
activate_infra_environment
check_aws_cli
check_aws_profile
create_local_testing_or_prod_resources
```

> **Note** The above code snippet if incomplete. It won't work

This sequence defines the **main execution flow** of the script — it runs each function in order to ensure everything is set before creating resources:

1. `check_num_of_args "$@"` – Verifies that the correct number of command-line arguments were passed (e.g., environment name, number of instances).
2. `activate_infra_environment` – Sets up or activates the environment (local, testing, or production) before resource creation.
3. `check_aws_cli` – Ensures that AWS CLI is installed on the system.
4. `check_aws_profile` – Confirms that a valid AWS profile is configured (unless in local mode).
5. `create_local_testing_or_prod_resources` – Executes the main task: creating local files (for local mode) or launching EC2 instances (for testing/production).

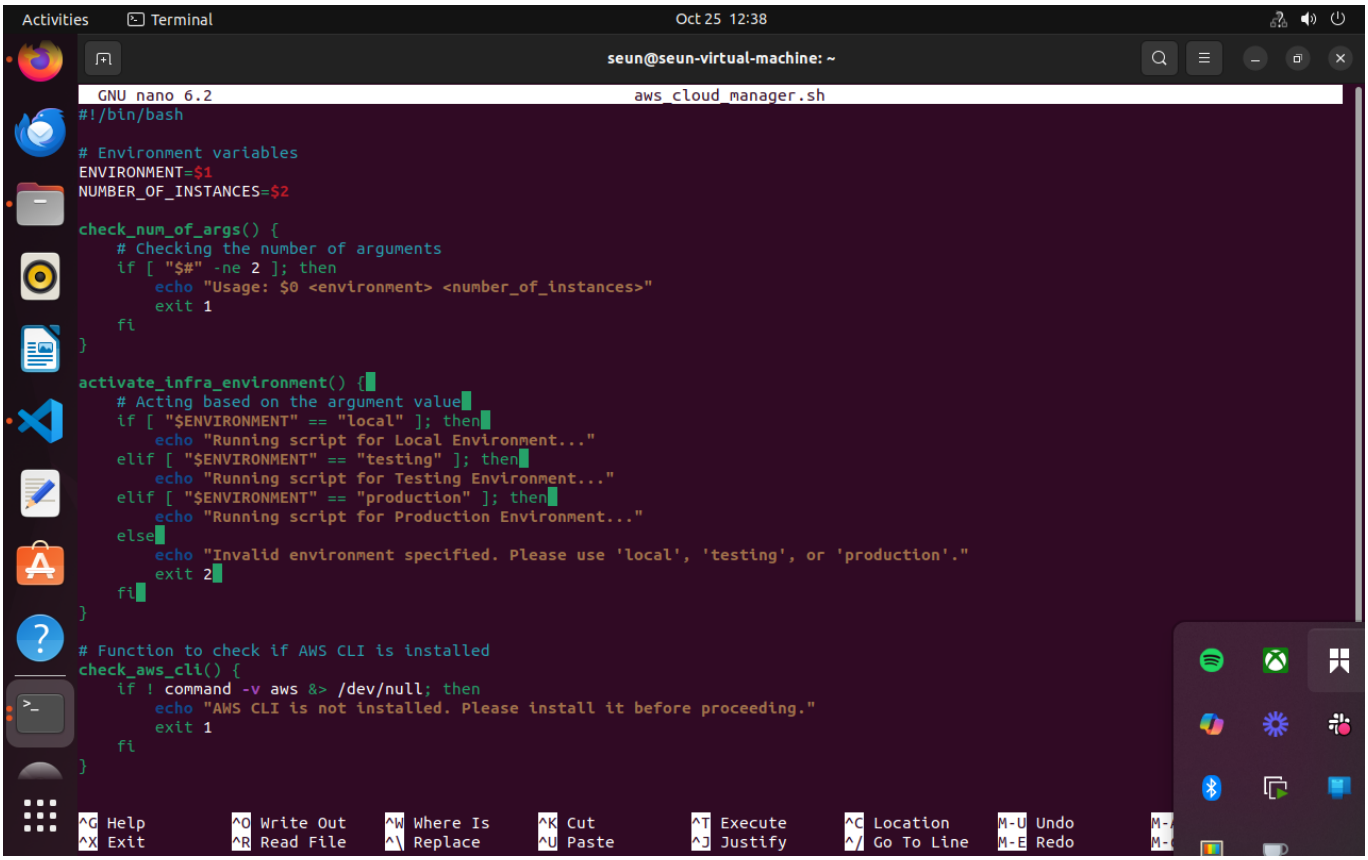
```
# Execute functions
check_num_of_args "$@"
activate_infra_environment
check_aws_cli
check_aws_profile
create_local_testing_or_prod_resources
```

Full Script:

```
#!/bin/bash

# Environment variables
ENVIRONMENT=$1
NUMBER_OF_INSTANCES=$2

check_num_of_args() {
    # Checking the number of arguments
    if [ "$#" -ne 2 ]; then
        echo "Usage: $0 <environment> <number_of_instances>"
        exit 1
    fi
}

activate_infra_environment() {
    # Acting based on the argument value
    if [ "$ENVIRONMENT" == "local" ]; then
        echo "Running script for Local Environment..."
    elif [ "$ENVIRONMENT" == "testing" ]; then
        echo "Running script for Testing Environment..."
    elif [ "$ENVIRONMENT" == "production" ]; then
        echo "Running script for Production Environment..."
    else
        echo "Invalid environment specified. Please use 'local', 'testing', or
'production'."
        exit 2
    fi
}


# Function to check if AWS CLI is installed
check_aws_cli() {
    if ! command -v aws &> /dev/null; then
        echo "AWS CLI is not installed. Please install it before proceeding."
        exit 1
    fi
}

# Function to check if AWS profile is set
check_aws_profile() {
    if [ -z "$AWS_PROFILE" ] && [ "$ENVIRONMENT" != "local" ]; then
        echo "AWS profile environment variable is not set for $ENVIRONMENT
environment."
```

```
        exit 1
    fi
}

# Function to create local log files or EC2 instances
create_local_testing_or_prod_resources() {
    # Validate number_of_instances is a positive integer
    if ! [[ "$NUMBER_OF_INSTANCES" =~ ^[1-9][0-9]*$ ]]; then
        echo "Invalid number_of_instances: Must be a positive integer."
        exit 2
    fi

    if [ "$ENVIRONMENT" == "local" ]; then
        echo "Creating $NUMBER_OF_INSTANCES sample log files locally..."
        for ((i=1; i<=NUMBER_OF_INSTANCES; i++)); do
            echo "Log entry $i for local simulation" > "log_file_$i.txt"
            if [ $? -eq 0 ]; then
                echo "☑ Created log_file_$i.txt successfully."
            else
                echo "❌ Failed to create log_file_$i.txt."
            fi
        done
    else

        # Common parameters
        instance_type="t2.micro"    # for testing environment
        instance_type2="t3.micro"   #for production environment
        ami_id="resolve:ssm:/aws/service/ami-amazon-linux-latest/al2023-ami-
kernel-default-x86_64"
        region="us-east-2"
        # No explicit subnet assignment (let AWS choose default)
        subnet_id=None   # or simply omit this parameter when creating the
instance
        # Create a new Security Group instead of using an existing one
        create_security_group=True  # flag to indicate a new SG should be created
        security_group_name="auto-created-sg"
        security_group_description="Security group automatically created for this
instance"
        key_name="my-key-new-pair" # for testing environment
        key_name2="MyKeyPair1"      #for production environment

        echo "Launching $NUMBER_OF_INSTANCES EC2 instances in $ENVIRONMENT subnet:
$subnet_id..."

        # Launch EC2 instances with profile based on environment
        if [ "$ENVIRONMENT" == "testing" ]; then
            aws ec2 run-instances --profile testing \
                --image-id "$ami_id" \
                --instance-type "$instance_type" \
                --count "$NUMBER_OF_INSTANCES" \
                --key-name "$key_name" \
                --region "$region" \
                --associate-public-ip-address \
                --tag-specifications "ResourceType=instance,Tags=
```

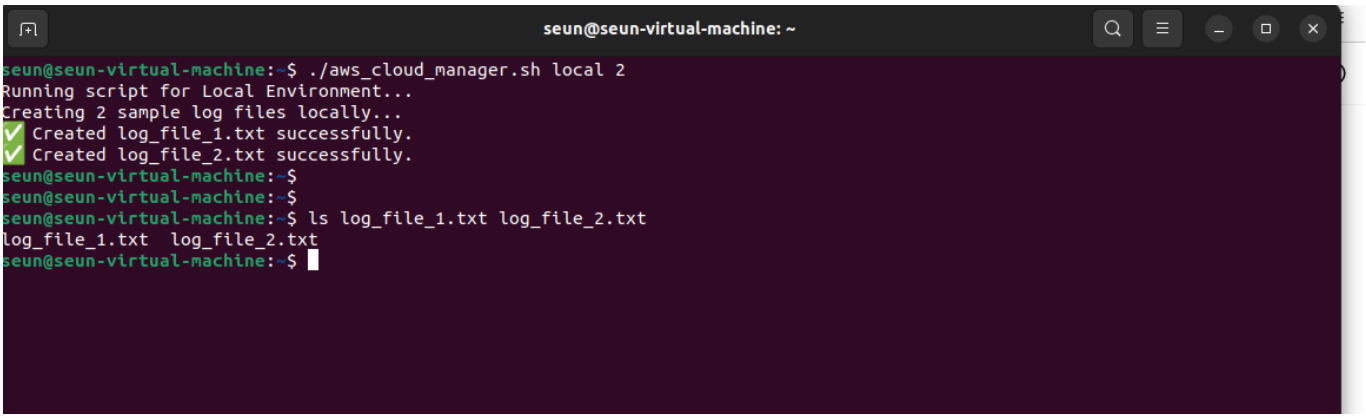
```
[{Key=Environment,Value=$ENVIRONMENT}]"
        elif [ "$ENVIRONMENT" == "production" ]; then
            aws ec2 run-instances --profile production \
                --image-id "$ami_id" \
                --instance-type "$instance_type2" \
                --count "$NUMBER_OF_INSTANCES" \
                --key-name "$key_name2" \
                --region "$region" \
                --associate-public-ip-address \
                --tag-specifications "ResourceType=instance,Tags=
[{Key=Environment,Value=$ENVIRONMENT}]"
        fi

        if [ $? -eq 0 ]; then
            echo "☑ $NUMBER_OF_INSTANCES EC2 instances created successfully in
$ENVIRONMENT subnet $subnet_id."
        else
            echo "❌ Failed to create EC2 instances in $ENVIRONMENT."
            exit
        fi
    fi
}


# Execute functions
check_num_of_args "$@"
activate_infra_environment
check_aws_cli
check_aws_profile
create_local_testing_or_prod_resources
```



Section 4: Comprehensive Testing

5.1 Local Environment Test

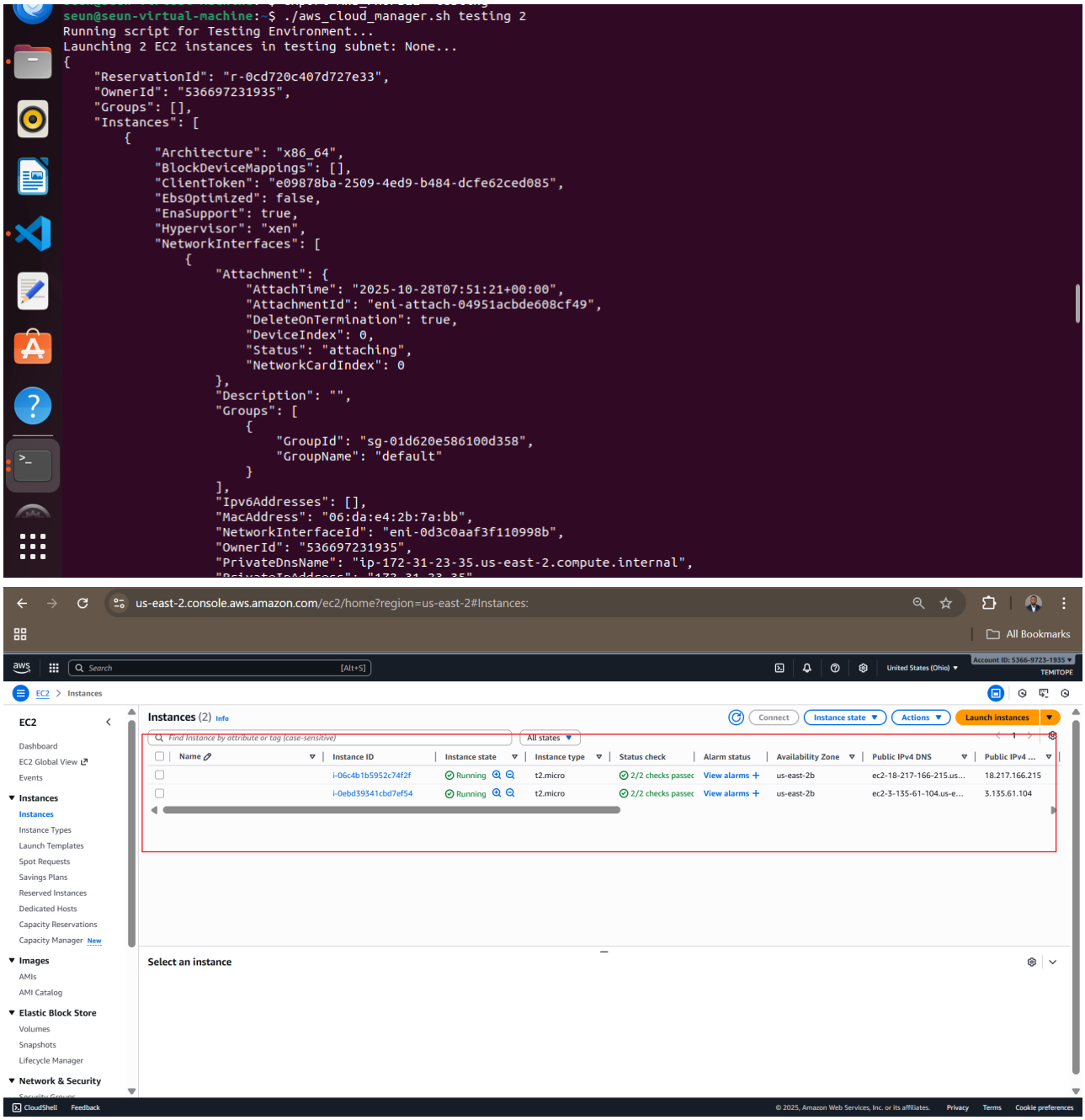
```
./aws_cloud_manager.sh local 2
```



5.2 Testing Environment Test

```
./aws_cloud_manager.sh testing 2
```

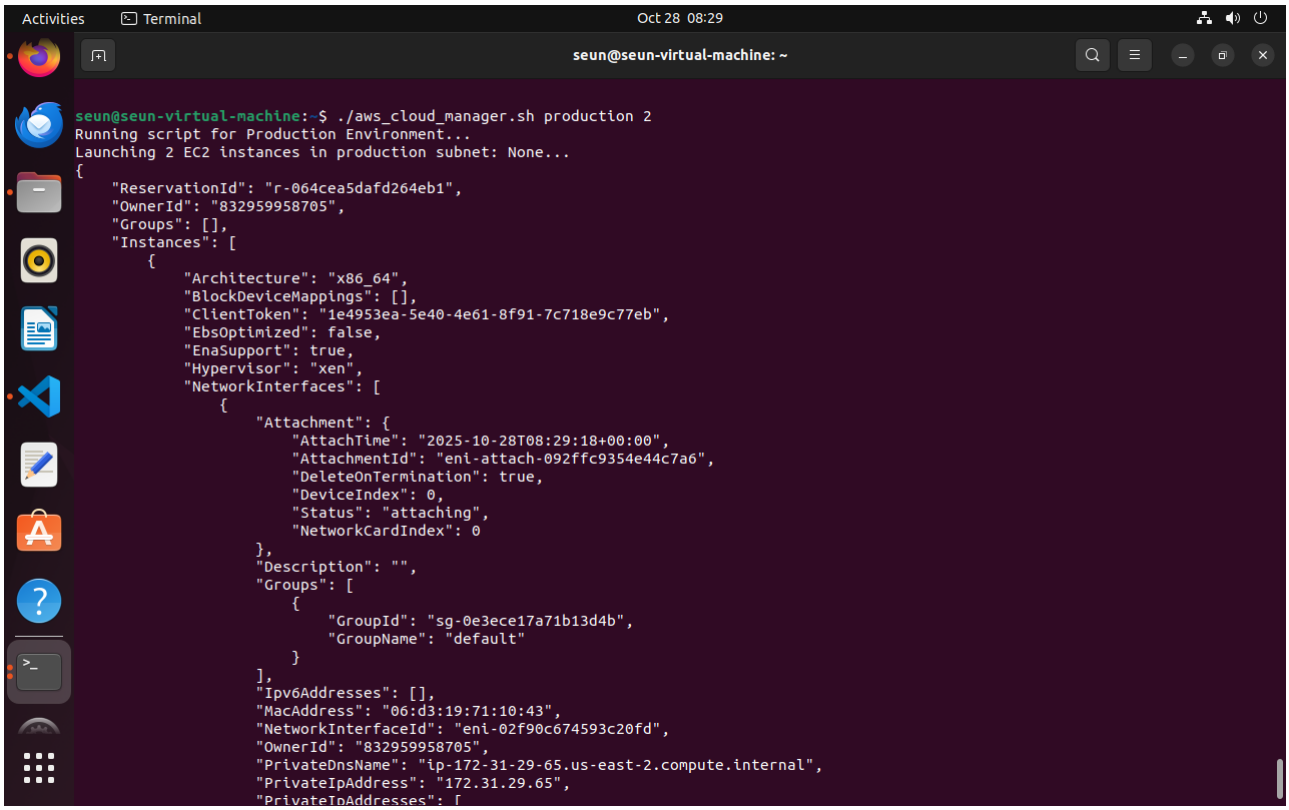
- After execution, two EC2 Instances were successfully created in testing environment (AWS Account 1) as instructed by the script.

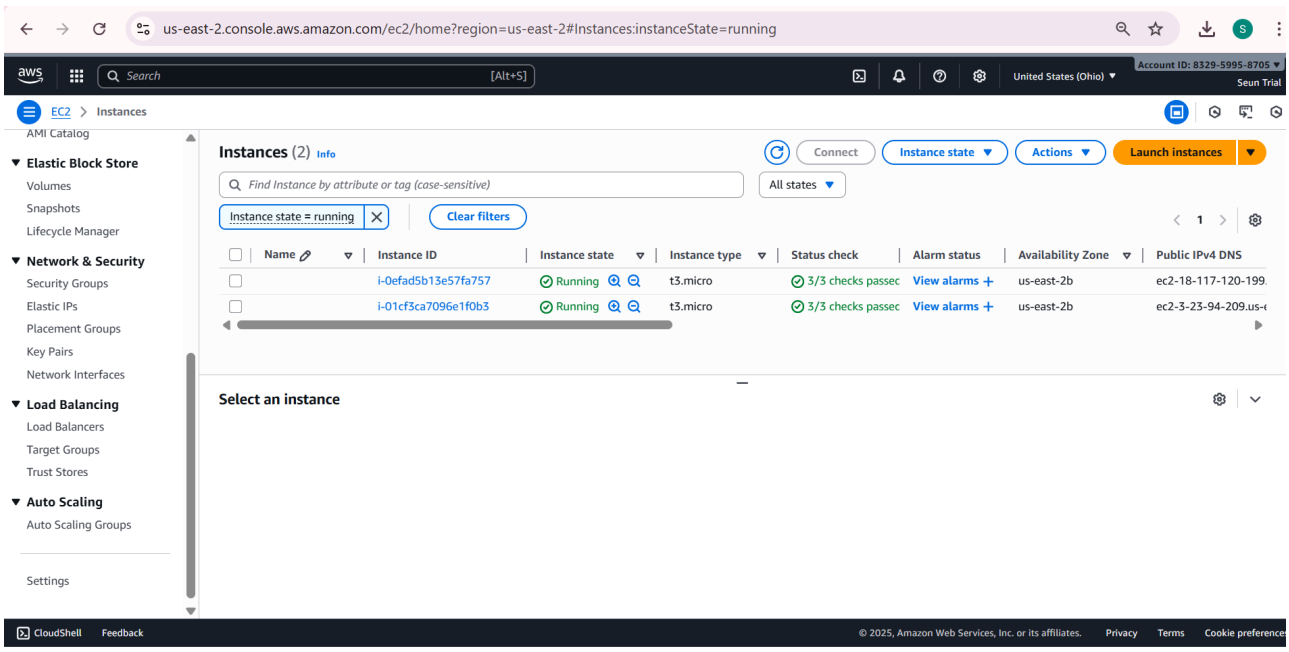


5.3 Production Environment Test

```
./aws_cloud_manager.sh production 1
```

- After execution, two EC2 Instances were successfully created in production environment (AWS Account 2) as instructed by the script.





Key Learnings & Feedback Addressed

Original Feedback	How Addressed
No AWS CLI Integration	Full `run-instances` implementation with profiles
Missing NUMBER_OF_INSTANCES	`$2` parameter with regex validation
Placeholder Logic	Real EC2 provisioning + local file operations
Error Handling	Exit codes 1-3, input validation, AWS checks
Testing Gaps	Comprehensive testing across all environments

Conclusion

This project successfully demonstrated the **full lifecycle of environment-aware cloud scripting**, from conceptual understanding to production-grade automation. By progressing through **two structured phases**, we evolved a basic conditional script into a **secure, reusable, and robust cloud management tool**.

Key achievements include:

- **Eliminated hard-coding** in favor of positional parameters and AWS CLI profiles
- **Implemented multi-account security** using dedicated IAM users and isolated profiles
- **Automated real EC2 provisioning** with environment-specific configurations (`t2.micro` for testing, `t3.micro` for production)
- **Added comprehensive validation** for arguments, AWS CLI, and profile existence
- **Verified functionality** across **local, testing, and production environments** with AWS Console evidence

The final script `aws_cloud_manager.sh` is **modular**, **well-documented**, and **ready for integration** into CI/CD pipelines or infrastructure-as-code frameworks. This implementation serves as a **strong portfolio piece** showcasing **Bash scripting**, **AWS automation**, **security best practices**, and **DevOps methodology**.

Recommendations

To extend this project into a **production-ready automation framework**, the following enhancements are recommended:

1. **Adopt Terraform or AWS CDK**
   Replace imperative `aws ec2 run-instances` calls with declarative infrastructure-as-code for idempotency and version control.

2. **Integrate with CI/CD**
   Trigger the script via **GitHub Actions** on code merge to testing/production branches.

3. **Add Cleanup Functionality**
   Implement a `--destroy` flag to terminate instances and delete log files for cost control.

4. **Use AWS Systems Manager Parameter Store**
   Store `NUMBER_OF_INSTANCES`, AMIs, and subnet IDs securely instead of hard-coding.

5. **Enhance Logging**
   Redirect output to structured logs (e.g., JSON) and send to CloudWatch.

6. **Add Dry-Run Mode**
   Simulate actions without executing AWS API calls for safer testing.

7. **Implement Idempotency**
   Check for existing resources (by tag) before creating new ones.

8. **Containerize the Script**
   Package in a Docker image for consistent execution across environments.

These improvements would transform this educational project into a **scalable, enterprise-grade automation tool**.

> **Let's connect on LinkedIn to discuss DevOps, automation, and cloud engineering!**
> linkedin.com