Labs 6-9

Student ID: 23917077

Student Name: Wenshuo Wang

Lab 6

Set up an EC2 instance

[1] Create an EC2 micro instance with Ubuntu and SSH into it

Lab 6 requires deploying a Django web application that can be accessed publicly over HTTP. To accomplish this, I needed to create an EC2 instance as the hosting environment. Rather than manually creating the instance through the AWS Console, I reused the automated Python script from Lab 2, modifying it to include HTTP traffic alongside SSH access.

The security group for this lab must allow two types of inbound traffic:

- **SSH (port 22)**: For secure remote access to install software and configure the server
- **HTTP (port 80)**: For public web traffic to access the Django application

The script automates the creation of the security group, adds both ingress rules, creates an SSH key pair, launches a t3.micro Ubuntu instance, and waits for it to reach the running state before retrieving its public IP address.

Command:

```
cd /home/vincent/CITS5503/Lab2
source ~/awsvenv/bin/activate
python3 create_ec2.py
```

Result:

```
Security Group created, ID: sg-00e60e763be6e296d
SSH port 22 and HTTP port 80 access authorised
EC2 instance launching, ID: i-03db22664e67cf54d
Instance ID: i-03db22664e67cf54d
Public IP: 3.36.63.21
SSH command: ssh -i 23917077-key.pem ubuntu@3.36.63.21
```

The EC2 instance was successfully created with public IP `3.36.63.21`, ready for Django deployment.

[2] Install the Python 3 virtual environment package

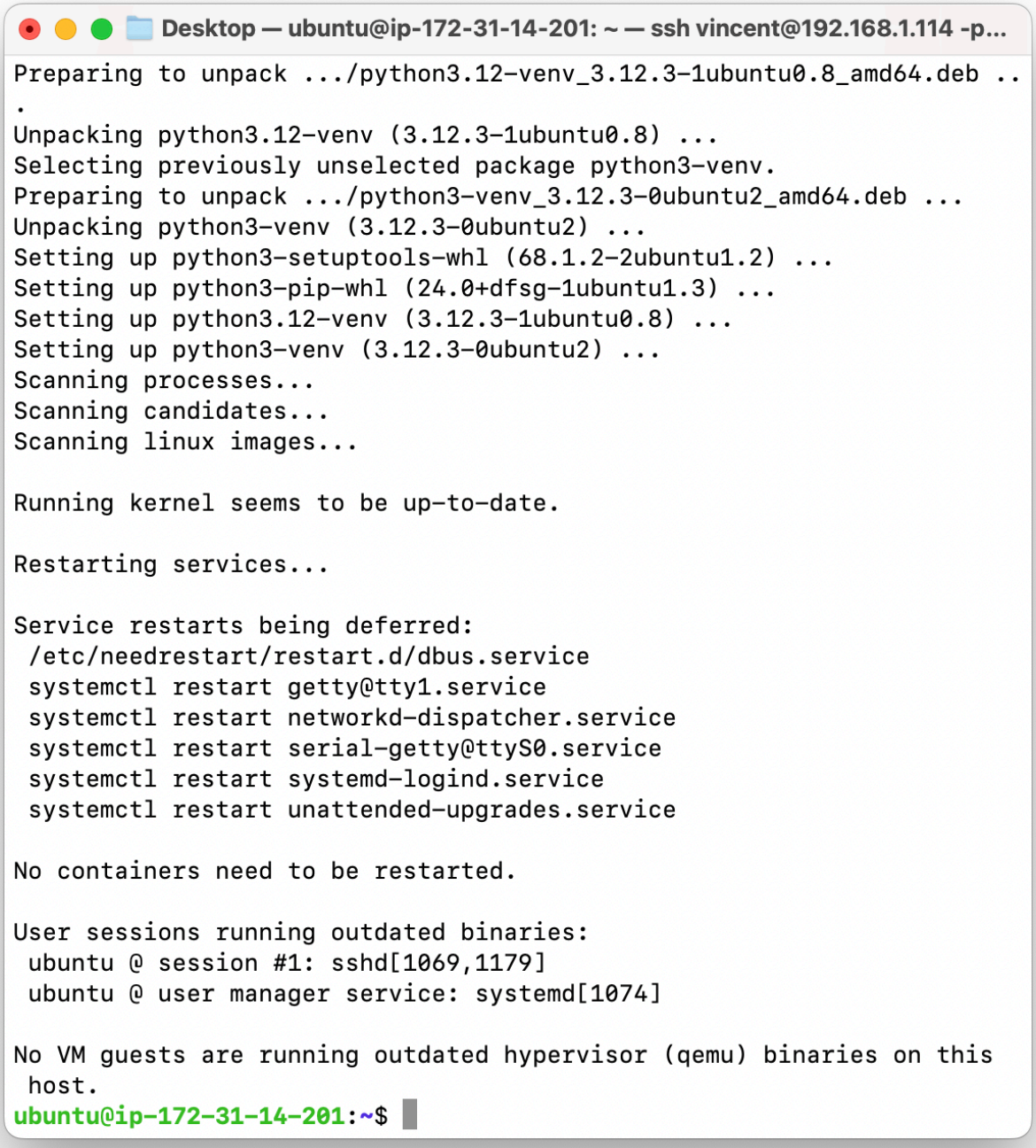
Before deploying Django, the EC2 instance must be prepared with the necessary software. I first connected via SSH using the private key generated during instance creation, then updated the system's package index and installed the Python virtual environment package.

A Python virtual environment is essential for isolating the Django application's dependencies from the system-level Python packages. This prevents version conflicts and ensures that the application runs in a clean, controlled environment.

Command:

```
ssh ubuntu@3.36.63.21 -i /home/vincent/CITS5503/23917077-key.pem
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install python3-venv
```

Result:



The package index was refreshed, all system packages were upgraded to their latest versions, and `python3-venv` was successfully installed. The upgrade included critical security updates for systemd, openssh, and Python libraries, ensuring a secure foundation for the Django deployment.

[3] Access a directory

The Django application must be stored in an appropriate location on the filesystem. Following Linux conventions, I created the directory structure `/opt/wwc/mysites` under the `/opt` directory, which is the standard location for optional third-party software packages.

Command:

```
mkdir -p /opt/wwc/mysites
cd /opt/wwc/mysites
```

[4] Set up a virtual environment

A Python virtual environment creates an isolated space for the Django project's dependencies, preventing conflicts with system-wide Python packages and ensuring that the application uses specific, compatible versions of all libraries.

Command:

```
python3 -m venv myvenv
```

The `-m venv` flag invokes Python's built-in virtual environment module, creating a new directory `myvenv` containing a standalone Python installation with its own interpreter and package manager.

[5] Activate the virtual environment

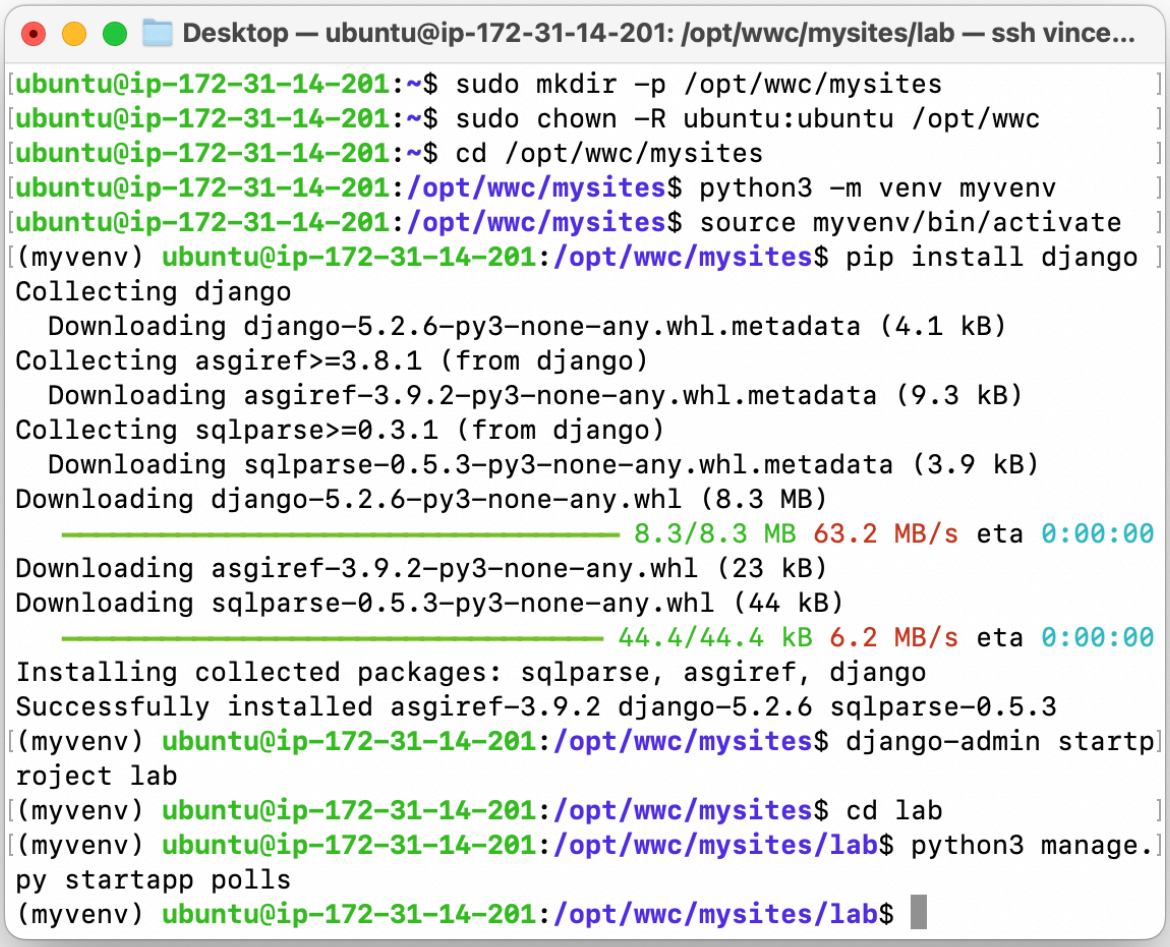
To use the virtual environment, it must be activated. Activation modifies the shell's environment variables to prioritize the virtual environment's Python interpreter and packages. I then installed Django, created the project structure, and set up the polls application.

Command:

```
source myvenv/bin/activate
pip install django
django-admin startproject lab
cd lab
python3 manage.py startapp polls
```

- `source myvenv/bin/activate`: Activates the virtual environment
- `pip install django`: Installs the Django framework within the isolated environment
- `django-admin startproject lab`: Creates a new Django project with necessary configuration files
- `python3 manage.py startapp polls`: Creates a polls application within the project

Result:



Django 5.2.6 was successfully installed. The project structure was created with configuration files for the `lab` project and the `polls` application.

[6] Install nginx

Nginx serves as a reverse proxy in this architecture. **Architectural rationale**: Separating the web server (nginx) from the application server (Django) follows the single responsibility principle—nginx handles HTTP protocol details, static file serving, and SSL termination efficiently, whilst Django focuses on application logic. This separation improves security (nginx buffers malicious requests), performance (nginx serves static files without invoking Python), and scalability (multiple Django workers behind one nginx instance). While Django includes a built-in development server, it is not designed for production use. Nginx acts as the public-facing web server, receiving HTTP requests on port 80 and forwarding them to the Django application running on port 8000.

Command:

```
sudo apt install nginx
```

Result:

Nginx was successfully installed and automatically started, ready to receive HTTP traffic on port 80.

[7] Configure nginx

The default nginx configuration must be replaced with a reverse proxy setup. This configuration tells nginx to listen on port 80 and forward all incoming requests to the Django development server running locally on port 8000.

Command:

```
sudo sh -c 'echo "server {
  listen 80 default_server;
  listen [::]:80 default_server;

  location / {
    proxy_set_header X-Forwarded-Host \$host;
    proxy_set_header X-Real-IP \$remote_addr;

    proxy_pass http://127.0.0.1:8000;
  }
}" > /etc/nginx/sites-enabled/default'
```

- `listen 80 default_server`: Listens for HTTP traffic on port 80 (IPv4 and IPv6)
- `proxy_set_header X-Forwarded-Host \$host`: Preserves the original hostname in request headers
- `proxy_set_header X-Real-IP \$remote_addr`: Preserves the client's IP address
- `proxy_pass http://127.0.0.1:8000`: Forwards requests to Django on localhost port 8000

[8] Restart nginx

Configuration changes only take effect after nginx is restarted.

Command:

```
sudo service nginx restart
```

The nginx service successfully reloaded with the new reverse proxy configuration.

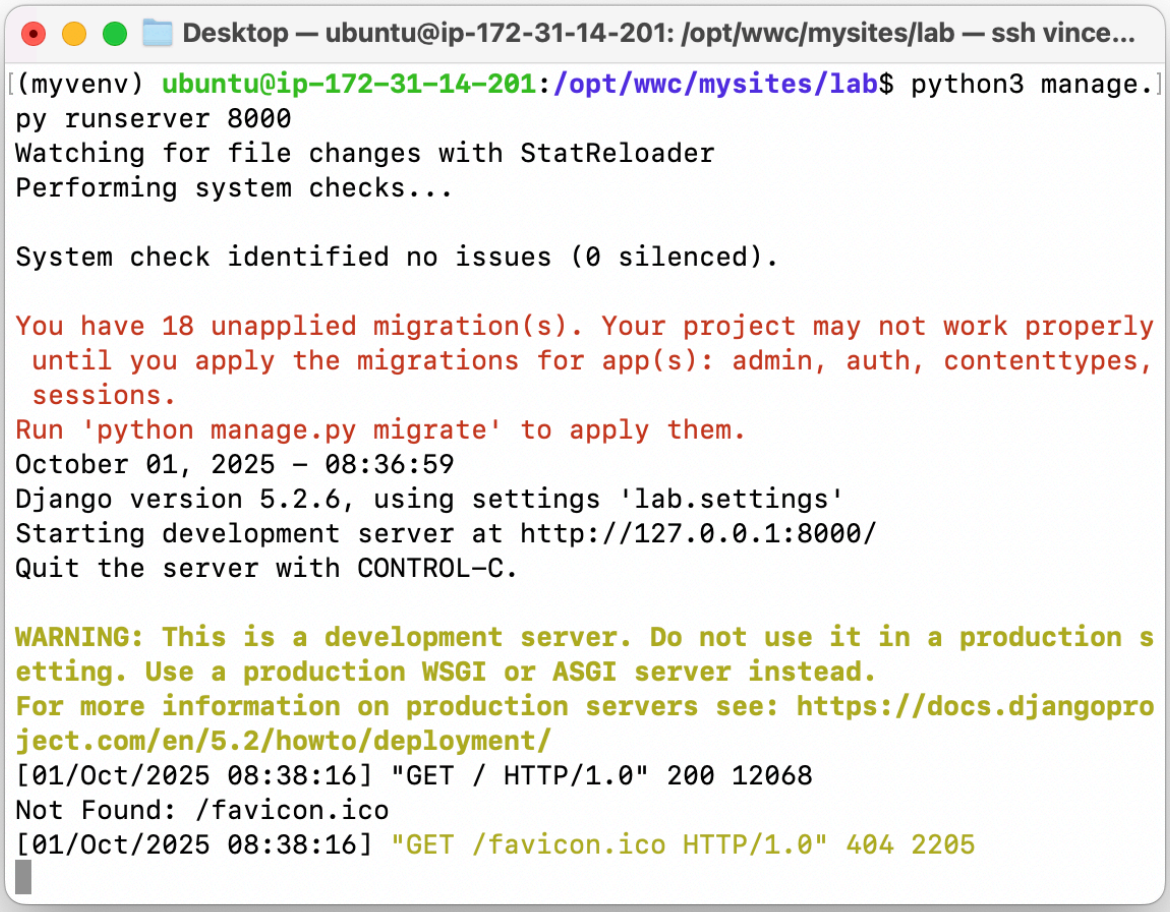
[9] Access your EC2 instance

With nginx configured, the final step is to start the Django development server. The server must bind to port 8000 so that nginx can forward requests to it.

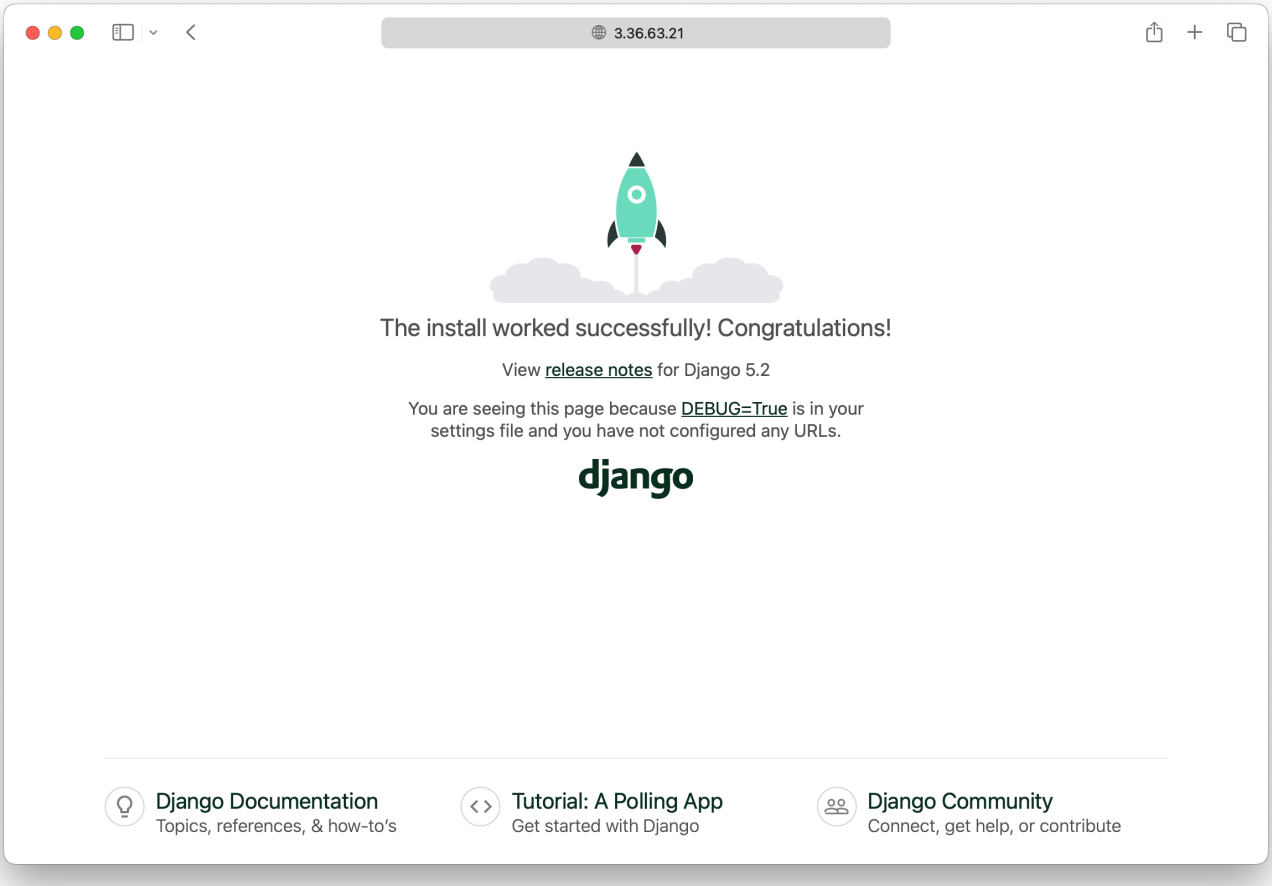
Command:

```
python3 manage.py runserver 8000
```

Result:



The Django development server started successfully on port 8000. Accessing the EC2 instance's public IP via a web browser displayed Django's default welcome page, confirming that the entire request chain is functional: Browser → Nginx (port 80) → Django (port 8000).



Set up Django inside the created EC2 instance

[1] Edit the following files (create them if not exist)

To create a functional polls application, three files must be configured: `polls/views.py` for application logic, `polls/urls.py` for URL routing within the polls app, and `lab/urls.py` to connect the polls app to the main project.

File 1: polls/views.py

This file contains the view function that handles HTTP requests and returns responses:

```
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world.")
```

- `HttpResponse`: A Django class that creates an HTTP response with text content
- `index`: A view function that accepts a request object and returns a response

File 2: polls/urls.py

This file maps URL patterns to view functions within the polls application:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
```

- `path('', views.index, name='index')`: Maps the root URL of the polls app to the `index` view
- The empty string `''` means this view is called when accessing `/polls/` (no additional path)

File 3: lab/urls.py

This is the project-level URL configuration that connects the polls app:

```
from django.urls import include, path
from django.contrib import admin

urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]
```

- `path('polls/', include('polls.urls'))`: Includes all URL patterns from `polls/urls.py` under the `/polls/` prefix
- `path('admin/', admin.site.urls)`: Provides access to Django's built-in admin interface

[2] Run the web server again

After editing the necessary files, the Django development server needs to be started to serve the application.

Command:

```
cd /opt/wwc/mysites/lab
source /opt/wwc/mysites/myvenv/bin/activate
python3 manage.py runserver 8000
```

The server starts and listens on all network interfaces at port 8000, making it accessible to nginx which forwards requests from port 80.

[3] Access the EC2 instance

With both nginx and Django running, the polls application becomes accessible through the EC2 instance's public IP address.

Access URL:

```
http://3.36.63.21/polls/
```

When accessing this URL in a browser, the application returns the "Hello, world." message defined in the polls index view. The request flow is: Browser → nginx (port 80) → Django (port 8000) → Response.

Set up an ALB

[1] Create an application load balancer

Building on the Application Load Balancer implementation from Lab 5, I created a Python script `create_alb.py` to set up an ALB for the Django application. The implementation follows the same structure as Lab 5 but adapts for Lab 6's requirements.

Similar to Lab 5, the script creates a complete ALB infrastructure using boto3. The key difference is that Lab 6 uses a custom health check path `/polls/` instead of the root path `/` used in Lab 5, since the Django application is deployed at that endpoint.

Key Implementation Functions:

The script uses the same core functions as Lab 5 with modifications:

- **Load Balancer Creation**: Creates an internet-facing application load balancer across multiple Availability Zones
- **Target Group Configuration**: Establishes an HTTP protocol target group with a health check on `/polls/`
- **Instance Registration**: Registers the Django EC2 instance as a target
- **Listener Setup**: Creates an HTTP listener on port 80 forwarding to the target group

Command:

```
python3 create_alb.py
```

Result:

```
Lab 6 - Application Load Balancer Setup
============================================================
Instance ID: i-03db22664e67cf54d
Security Group ID: sg-00e60e763be6e296d
Selected 2 subnets:
  - subnet-0ad8b31e7aa00ed99 (ap-northeast-2c)
  - subnet-0a80ae02a8ecb60f9 (ap-northeast-2d)

Creating Load Balancer: 23917077-lab6-alb
✓ Load Balancer created: 23917077-lab6-alb
  DNS Name: 23917077-lab6-alb-613263155.ap-northeast-2.elb.amazonaws.com

Creating Target Group: 23917077-lab6-tg
✓ Target Group created with health check on /polls/
  ARN: arn:aws:elasticloadbalancing:ap-northeast-2:489389878001:targetgroup/23917077-lab6-
tg/68562d9ae4641ec1

Registering instance i-03db22664e67cf54d to target group...
✓ Instance registered

Creating listener on port 80...
✓ Listener created
```

The load balancer now spans three availability zones (2a, 2c, 2d), and the target health changed from "Target.NotInUse" to "healthy".

[2] Health check

The health check is a critical component that monitors the availability and health of the target EC2 instance. I configured the target group with specific health check parameters:

Health Check Configuration:

- **Path**: `/polls/` - The specific URL path the ALB will request to verify instance health
- **Interval**: 30 seconds - Time between consecutive health checks
- **Timeout**: 5 seconds - Maximum time to wait for a response
- **Healthy Threshold**: 2 - Number of consecutive successful checks required to mark instance as healthy
- **Unhealthy Threshold**: 2 - Number of consecutive failed checks to mark instance as unhealthy

The health check verifies that the Django application is responding correctly at the `/polls/` endpoint every 30 seconds. If the endpoint returns an HTTP 200 status code within 5 seconds, the check passes. After 2 consecutive successful checks, the instance is marked healthy and begins receiving traffic from the load balancer.

[3] Access

After the load balancer setup completes and health checks pass, the Django application becomes accessible through the ALB's DNS name.

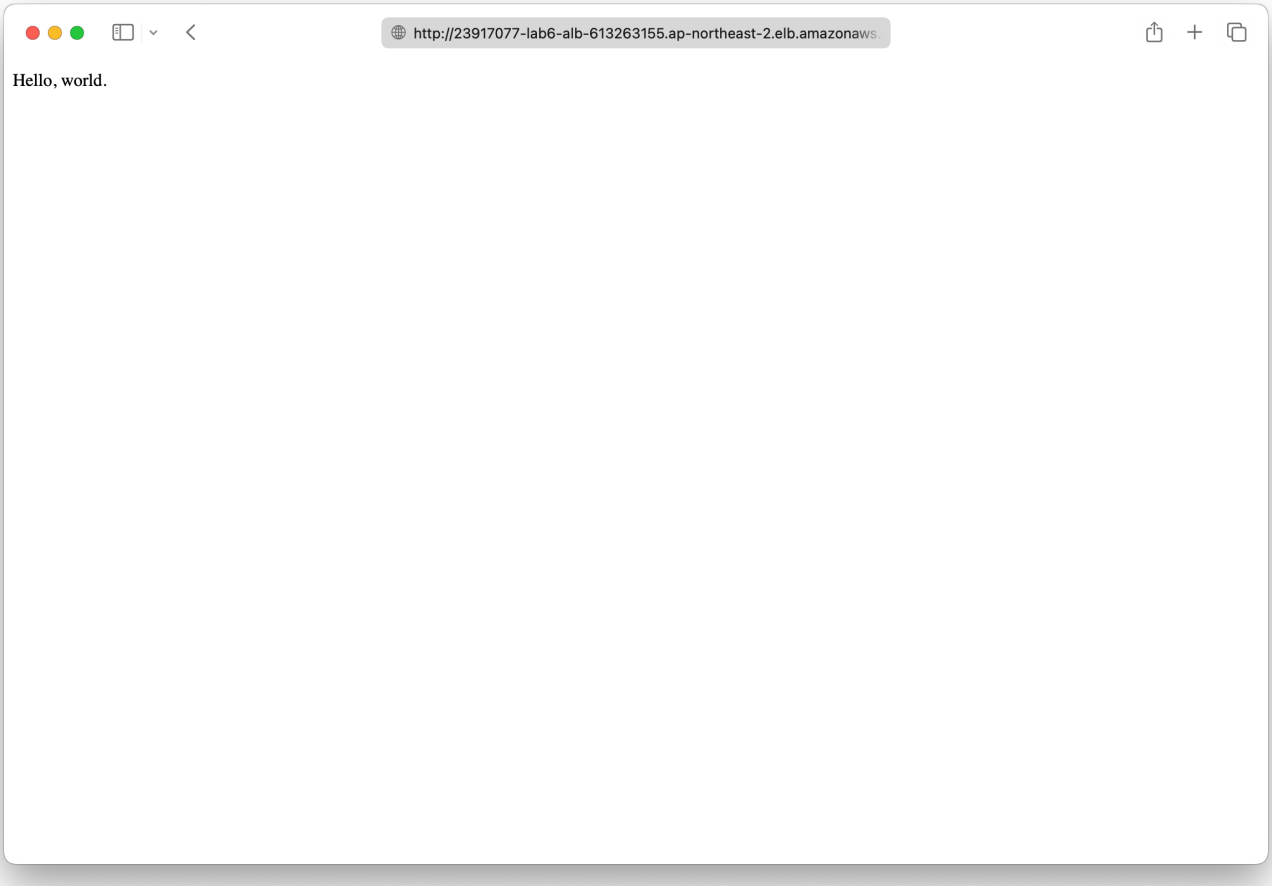
Access URL:

```
http://23917077-lab6-alb-613263155.ap-northeast-2.elb.amazonaws.com/polls/
```

When accessing this URL, the request flows as follows:

1. Client sends request to ALB DNS name
2. ALB routes request to healthy target (EC2 instance)
3. Nginx on EC2 receives request on port 80
4. Nginx proxies to Django server on port 8000
5. Django processes request and returns "Hello, world."
6. Response travels back through the same path to client

Result:



Web interface for CloudStorage application

To demonstrate integration between Django and AWS DynamoDB, I extended the polls application to display file metadata from a DynamoDB table, creating a web interface for the CloudStorage application from Lab 3.

[1] Migrate DynamoDB table

The DynamoDB table was created locally in Lab 3. To make the file metadata accessible to the Django application on EC2, I created a migration script `migrate_dynamodb.py` to copy data from the local DynamoDB instance to AWS DynamoDB.

Implementation:

```
import boto3
from botocore.exceptions import ClientError

# Configuration
TABLE_NAME = 'CloudFiles'
AWS_REGION = 'ap-northeast-2'  # Seoul region

# Connect to LOCAL DynamoDB
local_dynamodb = boto3.resource('dynamodb',
                                endpoint_url='http://localhost:8000',
                                region_name=AWS_REGION,
                                aws_access_key_id='AKIAXD4PI5LY6R5G4K2P',

 aws_secret_access_key='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')

# Connect to AWS DynamoDB (uses credentials from ~/.aws/credentials or environment)
aws_dynamodb = boto3.resource('dynamodb', region_name=AWS_REGION)

def create_aws_table():
    """Create the CloudFiles table in AWS DynamoDB"""
    try:
        print("Creating CloudFiles table in AWS DynamoDB...")

        table = aws_dynamodb.create_table(
            TableName=TABLE_NAME,
            KeySchema=[
                {
                    'AttributeName': 'userId',
                    'KeyType': 'HASH'  # Partition key
                },
                {
                    'AttributeName': 'fileName',
                    'KeyType': 'RANGE'  # Sort key
                }
            ],
            AttributeDefinitions=[
                {
                    'AttributeName': 'userId',
                    'AttributeType': 'S'
```

```
                },
                {
                    'AttributeName': 'fileName',
                    'AttributeType': 'S'
                }
            ],
            BillingMode='PAY_PER_REQUEST'  # Use on-demand billing
        )

        # Wait for table to be created
        print("Waiting for table to be created in AWS...")
        table.wait_until_exists()

        print(f"✓ Table {TABLE_NAME} created successfully in AWS!")
        return table

    except ClientError as e:
        if e.response['Error']['Code'] == 'ResourceInUseException':
            print(f"✓ Table {TABLE_NAME} already exists in AWS")
            return aws_dynamodb.Table(TABLE_NAME)
        else:
            print(f"✗ Error creating table: {e}")
            raise
    except Exception as e:
        print(f"✗ Unexpected error: {e}")
        raise

def read_local_table():
    """Read all items from local DynamoDB table"""
    try:
        print(f"\nReading data from local DynamoDB table: {TABLE_NAME}")

        local_table = local_dynamodb.Table(TABLE_NAME)
        response = local_table.scan()
        items = response['Items']

        # Handle pagination if there are many items
        while 'LastEvaluatedKey' in response:
            response = local_table.scan(ExclusiveStartKey=response['LastEvaluatedKey'])
            items.extend(response['Items'])

        print(f"✓ Found {len(items)} items in local table")
        return items

    except Exception as e:
        print(f"✗ Error reading local table: {e}")
        raise

def write_to_aws_table(items):
    """Write items to AWS DynamoDB table"""
    try:
        print(f"\nWriting data to AWS DynamoDB table: {TABLE_NAME}")
```

```
        aws_table = aws_dynamodb.Table(TABLE_NAME)

        success_count = 0
        error_count = 0

        # Write items using batch writer for efficiency
        with aws_table.batch_writer() as batch:
            for item in items:
                try:
                    batch.put_item(Item=item)
                    success_count += 1
                    print(f"  ✓ Migrated: {item.get('fileName', 'unknown')}")
                except Exception as e:
                    error_count += 1
                    print(f"  ✗ Failed to migrate {item.get('fileName', 'unknown')}: {e}")

        print(f"\n✓ Migration complete!")
        print(f"  Success: {success_count} items")
        print(f"  Failed: {error_count} items")

        return success_count, error_count

    except Exception as e:
        print(f"✗ Error writing to AWS table: {e}")
        raise

def verify_migration():
    """Verify data in AWS DynamoDB"""
    try:
        print(f"\nVerifying data in AWS DynamoDB...")

        aws_table = aws_dynamodb.Table(TABLE_NAME)
        response = aws_table.scan()
        items = response['Items']

        print(f"✓ AWS DynamoDB contains {len(items)} items")

        # Display sample items
        if items:
            print("\nSample items in AWS DynamoDB:")
            print("=" * 80)
            for i, item in enumerate(items[:3], 1):  # Show first 3 items
                print(f"\nItem {i}:")
                for key, value in sorted(item.items()):
                    print(f"  {key}: {value}")

            if len(items) > 3:
                print(f"\n... and {len(items) - 3} more items")

        return len(items)
```

```
    except Exception as e:
        print(f"✗ Error verifying AWS table: {e}")
        raise

def main():
    """Main migration process"""
    print("=" * 80)
    print("DynamoDB Migration: Local → AWS")
    print("=" * 80)

    try:
        # Step 1: Create AWS table
        create_aws_table()

        # Step 2: Read from local
        local_items = read_local_table()

        if not local_items:
            print("\n⚠ No items found in local table. Nothing to migrate.")
            return

        # Step 3: Write to AWS
        success, failed = write_to_aws_table(local_items)

        # Step 4: Verify
        aws_count = verify_migration()

        # Summary
        print("MIGRATION SUMMARY")
        print(f"Local items read:     {len(local_items)}")
        print(f"Successfully migrated: {success}")
        print(f"Failed:               {failed}")
        print(f"AWS items verified:   {aws_count}")

        if success == len(local_items) and success == aws_count:
            print("\n✓ Migration completed successfully!")
        else:
            print("\n⚠ Migration completed with issues. Please review the logs.")

    except Exception as e:
        print(f"\n✗✗✗ Migration failed: {e} ✗✗✗")
        raise

if __name__ == "__main__":
    main()
```

Detailed script explanation:

The migration script employs a four-phase approach to safely transfer data between DynamoDB instances, demonstrating production-ready migration patterns:

Phase 1: Dual Connection Setup

- **Local DynamoDB connection**: Uses `boto3.resource('dynamodb', endpoint_url='http://localhost:8000')` to connect to the Docker container. The `endpoint_url` parameter overrides default AWS endpoints, directing requests to the local instance.
- **AWS DynamoDB connection**: Creates a separate resource without `endpoint_url`, using default AWS endpoints with credentials from environment variables.
- **boto3.resource() rationale**: Chosen over `boto3.client()` for object-oriented access, automatic pagination, and simpler syntax.

Phase 2: Table Schema Replication (`create_aws_table`)

- **KeySchema**: Defines composite primary key (userId as HASH partition key, fileName as RANGE sort key) identical to Lab 3
- **BillingMode='PAY_PER_REQUEST'**: On-demand billing optimal for lab environments—no capacity planning required, automatic scaling, cost-effective for sporadic access

Phase 3: Data Extraction with Pagination (`read_local_table`)

- **scan() operation**: Reads all items without filtering. The pagination loop handles DynamoDB's 1MB response limit:

  ```
  while 'LastEvaluatedKey' in response:
      response = local_table.scan(ExclusiveStartKey=response['LastEvaluatedKey'])
  ```

  This ensures complete data retrieval regardless of table size.

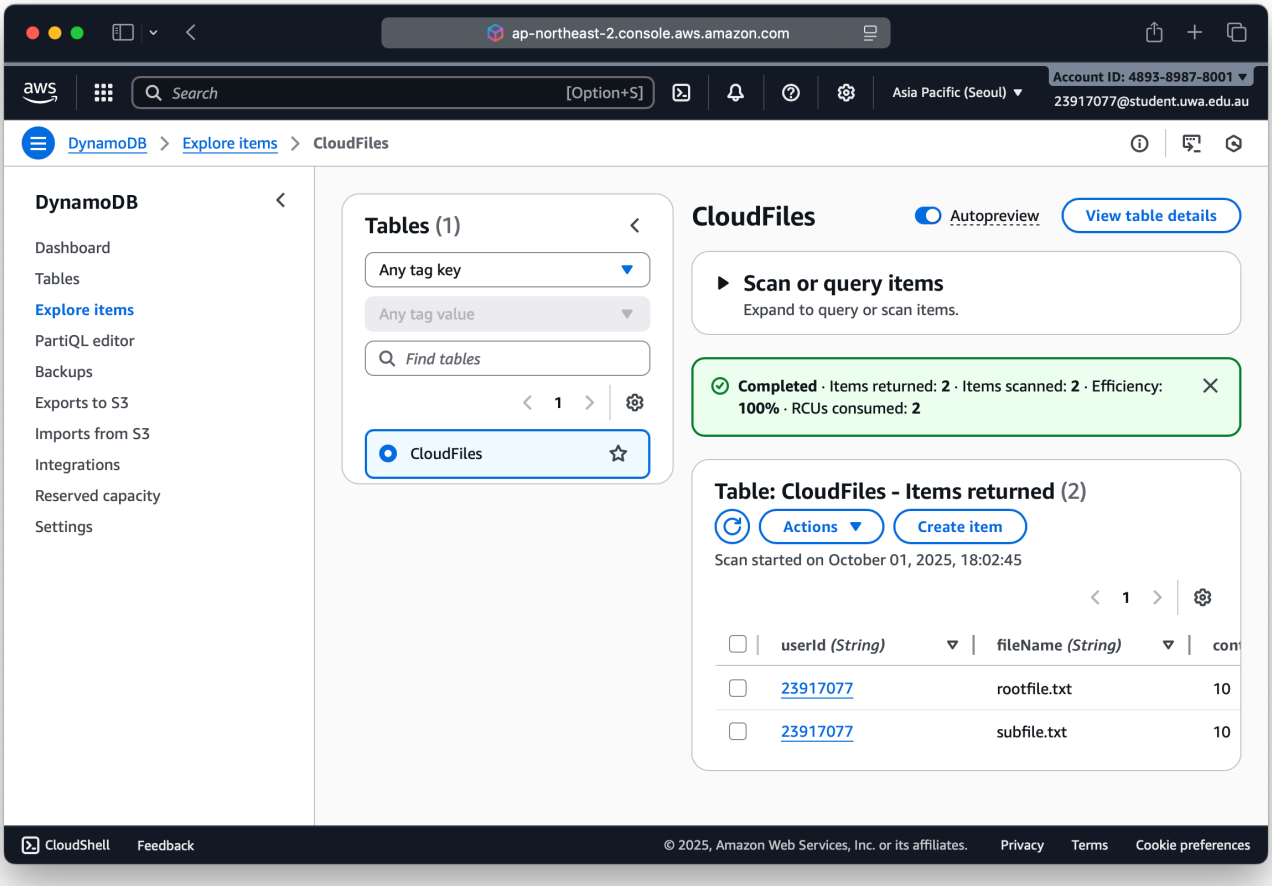
Phase 4: Efficient Bulk Upload (`write_to_aws_table`)

- **batch_writer() context manager**: Automatically batches up to 25 items per request (DynamoDB maximum), handles retries, and reduces API calls by ~25x compared to individual put_item() operations

Command:

```
python3 migrate_dynamodb.py
```

Result:



The AWS console confirms the CloudFiles table now contains rootfile.txt and subfile.txt from the local DynamoDB instance.

[2] Install boto3 in EC2 virtual environment

To enable Django to communicate with DynamoDB, I installed the boto3 library in the EC2 instance's virtual environment.

Command:

```
pip install boto3
```

Result:

```
Successfully installed boto3-1.40.42 botocore-1.40.42 jmespath-1.0.1
python-dateutil-2.9.0.post0 s3transfer-0.14.0 six-1.17.0 urllib3-2.5.0
```

[3] Configure Django templates

Django requires templates to be placed in the `app_name/templates/app_name/` directory structure. I created this structure and added a template to display the DynamoDB files.

Command:

```
mkdir -p /opt/wwc/mysites/lab/polls/templates/polls
```

Template File: `polls/templates/polls/files.html`

The template iterates through DynamoDB items and displays them in an HTML table:

```
<html>
<head>
    <title>CloudStorage Files</title>
    <style>
        body { font-family: Arial, sans-serif; margin: 40px; background-colour: #f5f5f5; }
        table { width: 100%; border-collapse: collapse; background-colour: white; }
        th { background-colour: #4CAF50; colour: white; padding: 12px; text-align: left; }
        td { padding: 10px; border-bottom: 1px solid #ddd; }
        tr:hover { background-colour: #f5f5f5; }
    </style>
</head>
<body>
    <h1>CloudStorage Files - Student 23917077</h1>
    <p>Total files: {{ items|length }}</p>
    <table>
        <tr>
            <th>File Name</th>
            <th>Path</th>
            <th>Size (bytes)</th>
            <th>Last Updated</th>
            <th>Permissions</th>
        </tr>
        {% for item in items %}
        <tr>
            <td><strong>{{ item.fileName }}</strong></td>
            <td>{{ item.path|default:"/" }}</td>
            <td>{{ item.size|default:"N/A" }}</td>
            <td>{{ item.lastUpdated }}</td>
            <td>{{ item.permissions|default:"private" }}</td>
        </tr>
        {% endfor %}
    </table>
</body>
</html>
```

[4] Update views.py with DynamoDB integration

I updated `polls/views.py` to connect to DynamoDB, scan the CloudFiles table, and render the data using the template.

File: `polls/views.py`

```
from django.shortcuts import render
from django.template import loader
from django.http import HttpResponse
```

```
import boto3
import json
from boto3.dynamodb.conditions import Key, Attr
from botocore.exceptions import ClientError

def index(request):
    template = loader.get_template("polls/files.html")  # Note: polls/ prefix required

    dynamodb = boto3.resource("dynamodb", region_name="ap-northeast-2",
                              aws_access_key_id="AKIAXD4PI5LY6R5G4K2P",

 aws_secret_access_key="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")

    table = dynamodb.Table("CloudFiles")

    items = []
    try:
        response = table.scan()
        items = response["Items"]
    except ClientError as e:
        print(e.response["Error"]["Message"])
        items = [{"fileName": "Error loading files", "path": "", "size": 0}]

    context = {"items": items}
    return HttpResponse(template.render(context, request))
```

Detailed code explanation:

The Django view function integrates AWS DynamoDB with web templating through three main operations:

- `boto3.resource('dynamodb', region_name=..., aws_access_key_id=..., aws_secret_access_key=...)`: Creates a DynamoDB resource client (high-level, object-oriented interface). Credentials are explicitly provided for educational clarity; production systems should use IAM roles or environment variables to avoid hardcoding secrets.
- `table.scan()`: Executes a DynamoDB Scan operation retrieving all items without filtering. **Scan characteristics**: Reads entire table sequentially, suitable for small tables (<1000 items) but inefficient for large datasets. **Alternative**: `query()` operation requires partition key but provides targeted, efficient retrieval. For this lab, scan's simplicity outweighs performance concerns given minimal data volume.
- `template.render(context, request)`: Processes the Django template with context data. The `context` dictionary makes `items` available in the template as `{{ items }}`. Django's template engine iterates through items using `{% for item in items %}`, accessing attributes with dot notation (`{{ item.fileName }}`). The `|default:"value"` filter provides fallback values for missing attributes.

Error handling rationale: The try-except block catches `ClientError` exceptions (AWS API failures), gracefully degrading to error display rather than crashing the entire application—essential for production reliability.

[5] Start Django server and access the web interface

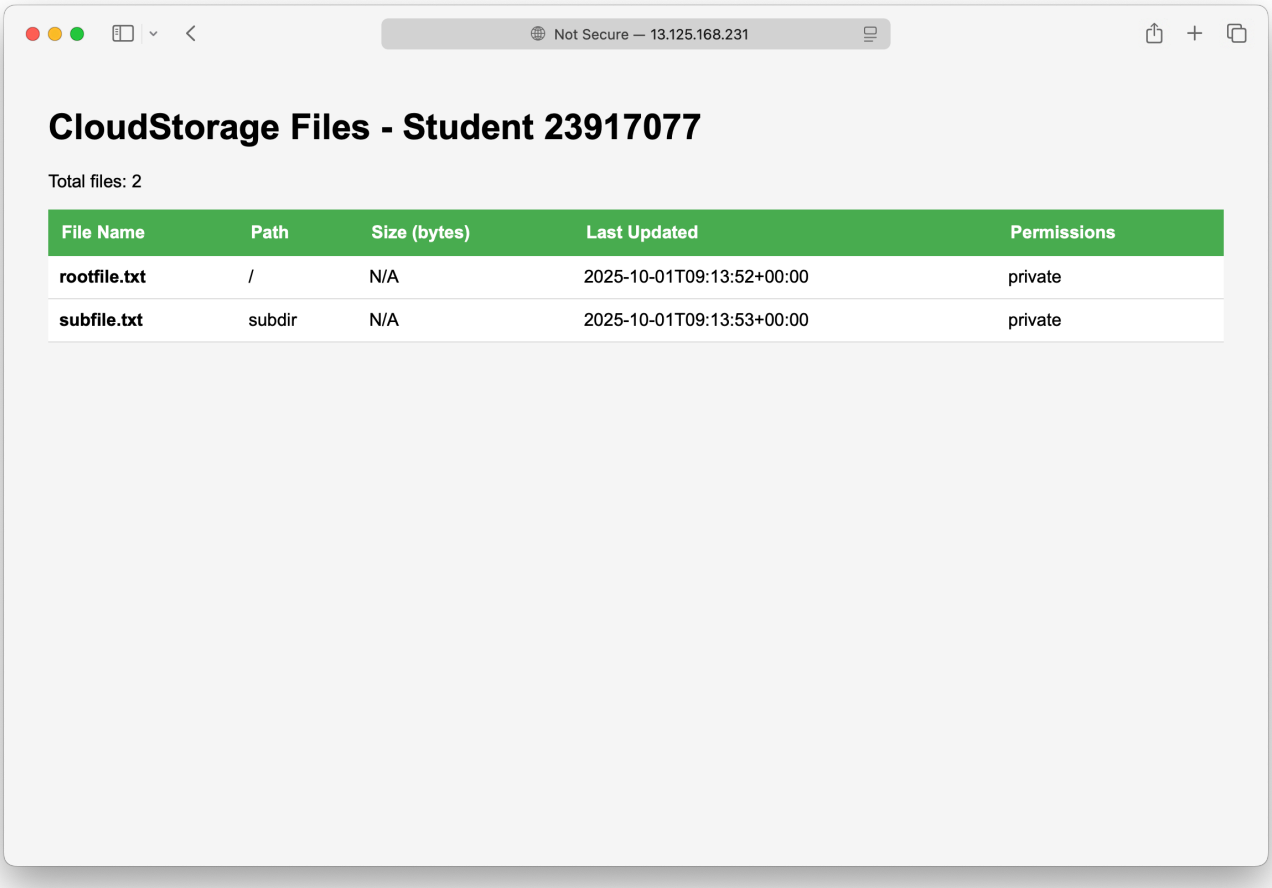
With all components configured, I started the Django development server to make the application accessible.

Command:

```
cd /opt/wwc/mysites/lab
source /opt/wwc/mysites/myvenv/bin/activate
python3 manage.py runserver 0.0.0.0:8000
```

Result:

The web interface successfully displays the CloudStorage files from DynamoDB. Accessing `http://3.36.63.21/polls/` shows a table containing rootfile.txt and subfile.txt with their metadata (path, size, last updated, permissions).



Cleanup Resources

After completing Lab 6, I cleaned up the created AWS resources to avoid ongoing charges and maintain account hygiene.

Resources deleted:

- EC2 instance and associated resources
- Security group
- SSH key pair

- Application Load Balancer
- Target group
- DynamoDB table (if no longer needed)

Commands:

```
# Terminate EC2 instance
aws ec2 terminate-instances --instance-ids i-03db22664e67cf54d

# Wait for instance termination
aws ec2 wait instance-terminated --instance-ids i-03db22664e67cf54d

# Delete Application Load Balancer
aws elbv2 delete-load-balancer --load-balancer-arn arn:aws:elasticloadbalancing:ap-
northeast-2:489389878001:loadbalancer/app/23917077-lab6-alb/...

# Delete Target Group
aws elbv2 delete-target-group --target-group-arn arn:aws:elasticloadbalancing:ap-
northeast-2:489389878001:targetgroup/23917077-lab6-tg/...

# Delete Security Group (wait a few minutes after instance termination)
aws ec2 delete-security-group --group-id sg-00e60e763be6e296d
```

Lab 7

Create an EC2 instance for Fabric automation

[1] Create EC2 instance using existing script

For Lab 7, I reused the EC2 instance creation script from Lab 6 with modifications for Lab 7 naming conventions. The script automates the creation of an EC2 instance, security group (allowing SSH port 22 and HTTP port 80), and SSH key pair for Fabric-based remote deployment.

Command:

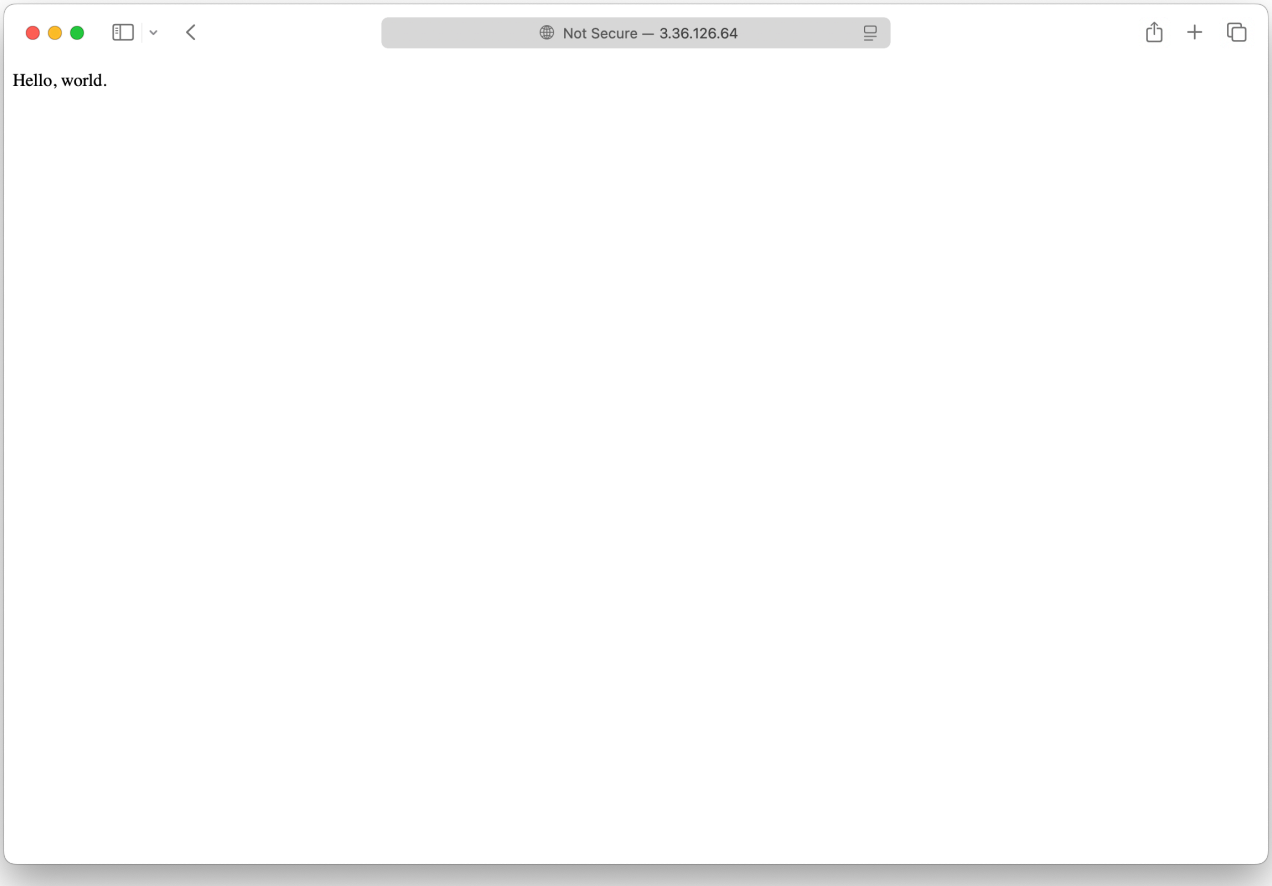
```
source ~/awsvenv/bin/activate
cd /home/vincent/CITS5503/Lab7
python3 create_ec2.py
```

Result:

```
Security Group created, ID: sg-0d0530c352b8c1466
SSH port 22 and HTTP port 80 access authorised
Key pair created and saved to /home/vincent/CITS5503/CITS5503_Sem2/Labs/23917077-key.pem
EC2 instance launching, ID: i-050afe7c000d94bf9
Instance ID: i-050afe7c000d94bf9
Public IP: 3.36.126.64
Public DNS: ec2-3-36-126-64.ap-northeast-2.compute.amazonaws.com
SSH command: ssh -i /home/vincent/CITS5503/CITS5503_Sem2/Labs/23917077-key.pem
ubuntu@3.36.126.64

Instance information saved to lab7_instance_info.txt
```

The EC2 instance was successfully created with public IP `3.36.126.64`. This instance serves as the deployment target for Fabric automation.



Install and configure Fabric

[1] Install Fabric in virtual environment

Fabric is a Python library for executing shell commands remotely over SSH. I installed it in my local virtual environment to automate deployment tasks on the remote EC2 instance.

Command:

```
pip install fabric
```

Result:

```
Collecting fabric
  Downloading fabric-3.2.2-py3-none-any.whl (59 kB)
Collecting invoke>=2.0 (from fabric)
  Downloading invoke-2.2.0-py3-none-any.whl (160 kB)
Collecting paramiko>=2.4 (from fabric)
  Downloading paramiko-4.0.0-py3-none-any.whl (223 kB)
Collecting decorator>=5 (from fabric)
  Downloading decorator-5.2.1-py3-none-any.whl (9.2 kB)
Collecting deprecated>=1.2 (from fabric)
  Downloading Deprecated-1.2.18-py2.py3-none-any.whl (10.0 kB)
...
```

```
Successfully installed bcrypt-5.0.0 cffi-2.0.0 cryptography-46.0.2 decorator-5.2.1
deprecated-1.2.18 fabric-3.2.2 invoke-2.2.0 paramiko-4.0.0 pycparser-2.23
pynacl-1.6.0 wrapt-1.17.3
```

Fabric was successfully installed with dependencies including Paramiko (SSH), Invoke (task execution), and cryptography libraries.

[2] Configure SSH config file

To allow Fabric to connect to the EC2 instance using an alias, I created an SSH config file with connection parameters.

Command:

```
mkdir -p ~/.ssh
cat > ~/.ssh/config << 'EOF'
Host 23917077-vm
  Hostname ec2-3-36-126-64.ap-northeast-2.compute.amazonaws.com
  User ubuntu
  UserKnownHostsFile /dev/null
  StrictHostKeyChecking no
  PasswordAuthentication no
  IdentityFile /home/vincent/CITS5503/CITS5503_Sem2/Labs/23917077-key.pem
EOF
chmod 600 ~/.ssh/config
```

Key configuration parameters:

- **Host**: Alias `23917077-vm` for easy reference
- **Hostname**: EC2 instance public DNS
- **User**: ubuntu (default for Ubuntu EC2 instances)
- **StrictHostKeyChecking no**: Auto-accepts new host keys (development only)
- **IdentityFile**: Path to the private SSH key

[3] Test Fabric connection

I tested the Fabric connection to verify SSH communication with the EC2 instance by using a simple Fabric script.

Command:

```
from fabric import Connection
c = Connection('23917077-vm')
result = c.run('uname -s')
print(f"Result: {result.stdout.strip()}")
```

Result:

```
Linux
Result: Linux
```

The connection test succeeded, confirming Fabric can execute remote commands on the EC2 instance.

Use Fabric for automation

[1] Write Fabric deployment script

I created a Python script `fabric_deploy_lab7.py` that automates the entire Django deployment process from Lab 6. The script executes all manual deployment steps remotely using Fabric.

```
#!/usr/bin/env python3
from fabric import Connection
import time

EC2_HOST = '23917077-vm'

def deploy_django_app():
    print("="*80)
    print("Lab 7 - Fabric Automation for Django Deployment")
    print("="*80)

    c = Connection(EC2_HOST)

    print("\n[1/9] Testing connection to EC2 instance...")
    result = c.run('uname -a', hide=True)
    print(f"✓ Connected to: {result.stdout.strip()}")

    print("\n[2/9] Updating system packages...")
    c.sudo('apt-get update', hide=True)
    c.sudo('apt-get upgrade -y', hide=True)
    print("✓ System packages updated")

    print("\n[3/9] Installing python3-venv...")
    c.sudo('apt-get install -y python3-venv', hide=True)
    print("✓ python3-venv installed")

    print("\n[4/9] Creating directory structure...")
    c.run('sudo mkdir -p /opt/wwc/mysites', hide=True)
    c.run('sudo chown -R ubuntu:ubuntu /opt/wwc', hide=True)
    print("✓ Directory /opt/wwc/mysites created")

    print("\n[5/9] Setting up Python virtual environment and installing Django...")
    c.run('cd /opt/wwc/mysites && python3 -m venv myvenv', hide=True)
    c.run('cd /opt/wwc/mysites && source myvenv/bin/activate && pip install --upgrade
pip', hide=True)
    c.run('cd /opt/wwc/mysites && source myvenv/bin/activate && pip install django',
hide=True)
    print("✓ Virtual environment created and Django installed")
```

```
    print("\n[6/9] Creating Django project 'lab' and app 'polls'...")
    c.run('cd /opt/wwc/mysites && source myvenv/bin/activate && django-admin startproject
lab', hide=True)
    c.run('cd /opt/wwc/mysites/lab && source ../myvenv/bin/activate && python3 manage.py
startapp polls', hide=True)
    print("✓ Django project and polls app created")

    print("\n[7/9] Configuring Django application files...")

    # Create views.py using cat with heredoc
    c.run("""cat > /opt/wwc/mysites/lab/polls/views.py << 'VIEWSEOF'
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, world.")
VIEWSEOF
""", hide=True)

    # Create polls urls.py
    c.run("""cat > /opt/wwc/mysites/lab/polls/urls.py << 'URLSEOF'
from django.urls import path
from . import views

urlpatterns = [
    path('', views.index, name='index'),
]
URLSEOF
""", hide=True)

    # Create lab urls.py
    c.run("""cat > /opt/wwc/mysites/lab/lab/urls.py << 'LABURLSEOF'
from django.urls import include, path
from django.contrib import admin

urlpatterns = [
    path('polls/', include('polls.urls')),
    path('admin/', admin.site.urls),
]
LABURLSEOF
""", hide=True)

    # Update settings.py
    c.run('sed -i "s/ALLOWED_HOSTS = \\[\\]/ALLOWED_HOSTS = [\'*\']/"
/opt/wwc/mysites/lab/lab/settings.py', hide=True)

    print("✓ Django application files configured")

    print("\n[8/9] Installing and configuring nginx...")
    c.sudo('apt-get install -y nginx', hide=True)

    # Create nginx config
```

```
    c.run("""cat > /tmp/nginx_default << 'NGINXEOF'
server {
  listen 80 default_server;
  listen [::]:80 default_server;

  location / {
    proxy_set_header X-Forwarded-Host $host;
    proxy_set_header X-Real-IP $remote_addr;

    proxy_pass http://127.0.0.1:8000;
  }
}
NGINXEOF
""", hide=True)
    c.sudo('mv /tmp/nginx_default /etc/nginx/sites-enabled/default', hide=True)
    c.sudo('systemctl restart nginx', hide=True)
    print("✓ nginx installed and configured")

    print("\n[9/9] Starting Django development server on port 8000...")
    c.run('pkill -f "manage.py runserver" || true', hide=True, warn=True)
    c.run('cd /opt/wwc/mysites/lab && source ../myvenv/bin/activate && nohup python3
manage.py runserver 0.0.0.0:8000 > /tmp/django.log 2>&1 &', hide=True)

    time.sleep(3)

    result = c.run('ps aux | grep "manage.py runserver" | grep -v grep', warn=True,
hide=True)
    if result.ok:
        print("✓ Django development server started successfully")
    else:
        print("⚠ Warning: Django server may not have started properly")

    print("\n" + "="*80)
    print("Deployment Complete!")
    print("="*80)

    public_ip = c.run('curl -s http://169.254.169.254/latest/meta-data/public-ipv4',
hide=True).stdout.strip()

    print(f"\nAccess the application at:")
    print(f"  http://{public_ip}/polls/")
    print(f"\nTo view Django logs:")
    print(f"  ssh {EC2_HOST} \"tail -f /tmp/django.log\"")
    print("\n" + "="*80)

if __name__ == '__main__':
    try:
        deploy_django_app()
    except Exception as e:
        print(f"\n✗ Deployment failed: {e}")
        raise
```

Deployment steps automated:

1. Test SSH connection to EC2 instance
2. Update system packages and install python3-venv
3. Create directory structure `/opt/wwc/mysites`
4. Set up Python virtual environment and install Django
5. Create Django project `lab` and app `polls`
6. Configure Django files (views.py, urls.py, settings.py)
7. Install and configure nginx as reverse proxy
8. Start Django development server on port 8000
9. Verify server is running

Detailed Fabric methods explanation:

- **`c.run(command, hide=True)`**: Executes commands as the ubuntu user via SSH. The command runs in the remote user's shell environment with their permissions. **`hide=True`** suppresses verbose output (stdout/stderr), displaying only our custom progress messages for cleaner logs.
- **`c.sudo(command, hide=True)`**: Executes commands with root privileges, equivalent to prepending `sudo` to the command. Required for system operations: package installation (`apt-get`), service management (`systemctl`), and system directory modifications.
- **`warn=True` parameter**: Prevents script termination when commands return non-zero exit codes. Essential for operations like `pkill` (which returns 1 if no process found) where "failure" is acceptable and shouldn't halt deployment.
- **Heredoc syntax for file creation**: Commands like `cat > file << 'EOF'` create multi-line files remotely:

  ```
  c.run("""cat > /path/to/file << 'EOF'
  Multi-line
  File content
  EOF
  """)
  ```

  The single quotes around 'EOF' prevent variable expansion, ensuring content is written literally. This technique avoids escaping complications and enables clean remote file creation.
- **`nohup command &` pattern**: The combination `nohup python3 manage.py runserver ... &` starts Django in the background:
  - `nohup`: Prevents process termination when SSH connection closes (ignores hangup signal)
  - `&`: Runs process in background, returning control to the script
  - `> /tmp/django.log 2>&1`: Redirects stdout and stderr to log file for debugging

Rationale for Fabric automation approach: This script eliminates manual SSH sessions, reduces human error, ensures consistent deployments, and documents the entire setup process as executable code. The sequential execution model (vs. parallel configuration management tools) provides clear step-by-step progress visibility, ideal for educational environments.

[2] Execute Fabric deployment

I executed the Fabric script to automatically deploy the Django application to the EC2 instance.

Command:

```
python3 fabric_deploy_lab7.py
```

Result:

```
================================================================================
Lab 7 - Fabric Automation for Django Deployment
================================================================================

[1/9] Testing connection to EC2 instance...
✓ Connected to: Linux ip-172-31-8-140 6.8.0-1029-aws...

[2/9] Updating system packages...
✓ System packages updated

[3/9] Installing python3-venv...
✓ python3-venv installed

[4/9] Creating directory structure...
✓ Directory /opt/wwc/mysites created

[5/9] Setting up Python virtual environment and installing Django...
✓ Virtual environment created and Django installed

[6/9] Creating Django project 'lab' and app 'polls'...
✓ Django project and polls app created

[7/9] Configuring Django application files...
✓ Django application files configured

[8/9] Installing and configuring nginx...
✓ nginx installed and configured

[9/9] Starting Django development server on port 8000...
✓ Django development server started successfully

================================================================================
Deployment Complete!
================================================================================

Access the application at:
  http://3.36.126.64/polls/
```

The script successfully automated all 9 deployment steps in approximately 5 minutes, reducing multiple manual SSH sessions to a single command execution.

[3] Access and verify deployment

I verified the automated deployment by accessing the Django application through the EC2 public IP.

Access URL:

```
http://3.36.126.64/polls/
```

Result:

The application returned "Hello, world." as expected, confirming the Django server, nginx reverse proxy, and URL routing are all working correctly.

Cleanup Resources

After completing Lab 7, I removed all created resources to prevent unnecessary charges.

Resources deleted:

- EC2 instance
- Security group
- SSH key pair

Commands:

```
# Terminate EC2 instance
aws ec2 terminate-instances --instance-ids i-050afe7c000d94bf9

# Wait for instance termination
aws ec2 wait instance-terminated --instance-ids i-050afe7c000d94bf9

# Delete Security Group
aws ec2 delete-security-group --group-id sg-0d0530c352b8c1466
```

Lab 8

Create Dockerfile and Build Docker Image

[1] Create Dockerfile

To run Jupyter Notebook with SageMaker capabilities in a containerised environment, I created a Dockerfile that installs all necessary dependencies and configures Jupyter to be accessible from any IP address.

File: Dockerfile

```
FROM python:3.10

RUN pip install jupyter boto3 sagemaker awscli
RUN mkdir /notebook

# Use a sample access token
ENV JUPYTER_ENABLE_LAB=yes
ENV JUPYTER_TOKEN="CITS5503"

# Allow access from ALL IPs
RUN jupyter notebook --generate-config
RUN echo "c.NotebookApp.ip = '0.0.0.0'" >> /root/.jupyter/jupyter_notebook_config.py

# Copy the ipynb file
RUN wget -P /notebook
https://raw.githubusercontent.com/zhangzhics/CITS5503_Sem2/master/Labs/src/LabAI.ipynb

WORKDIR /notebook
EXPOSE 8888

CMD ["jupyter", "notebook", "--ip=0.0.0.0", "--port=8888", "--no-browser", "--allow-root"]
```

Comprehensive Dockerfile explanation:

The Dockerfile constructs the image through layered instructions, each creating an immutable layer:

- **`FROM python:3.10`**: Selects official Python 3.10 base image (Debian-based) providing Python interpreter, pip, and build tools. **Rationale**: Python 3.10 ensures SageMaker SDK compatibility (requires 3.7+) whilst providing modern features and security patches.
- **`RUN pip install jupyter boto3 sagemaker awscli`**: Installs four essential packages in one layer (optimises image size):
  - `jupyter`: Interactive notebook environment with cell-by-cell execution and inline visualisation
  - `boto3`: AWS SDK enabling S3, SageMaker, and DynamoDB operations
  - `sagemaker`: High-level library for hyperparameter tuning and model deployment
  - `awscli`: Command-line interface for AWS service management
- **`RUN mkdir /notebook`**: Creates dedicated directory for notebooks. **Rationale**: Separates application files from system directories, simplifies volume mounting, follows Unix FHS standards.

- `ENV JUPYTER_ENABLE_LAB=yes`: Activates JupyterLab (modern tabbed interface with file browser and terminal) instead of classic Notebook interface.
- `ENV JUPYTER_TOKEN="CITS5503"`: Sets fixed authentication token. **Security consideration**: Production systems should use randomly-generated tokens; fixed tokens acceptable for lab convenience whilst maintaining basic security.
- `RUN jupyter notebook --generate-config`: Generates default configuration file at `/root/.jupyter/jupyter_notebook_config.py` containing ~1000 configuration options.
- `RUN echo "c.NotebookApp.ip = '0.0.0.0'" >> ...`: Critical for container networking. **Default**: Jupyter binds to `127.0.0.1` (localhost only), refusing external connections. `0.0.0.0` **binding**: Listens on all interfaces, essential for ECS tasks with dynamic IPs. Security maintained through ECS security groups, token authentication, and VPC isolation.
- `RUN wget -P /notebook https://...`: Downloads LabAI.ipynb during build. **Advantages**: Eliminates manual file transfer, ensures version consistency, simplifies deployment.
- `WORKDIR /notebook`: Sets working directory so Jupyter starts here, displaying the notebook immediately in the file browser.
- `EXPOSE 8888`: Documents listening port (doesn't actually publish). Acts as documentation for image users and informs Docker port mapping.
- `CMD ["jupyter", "notebook", "--ip=0.0.0.0", "--port=8888", "--no-browser", "--allow-root"]`: Uses exec form (JSON array) for proper signal handling.
  - `--no-browser`: Prevents failed browser launch attempts in containers
  - `--allow-root`: Permits running as root (normally refused by Jupyter). Safe in containers due to isolation; production images should create non-root users.

Layer caching optimisation: Instruction order places stable layers (base image, packages) before variable ones (downloads), maximising Docker cache efficiency.

[2] Build Docker image

With the Dockerfile created, I built the Docker image locally.

Command:

```
docker build -t 23917077-lab8 .
```

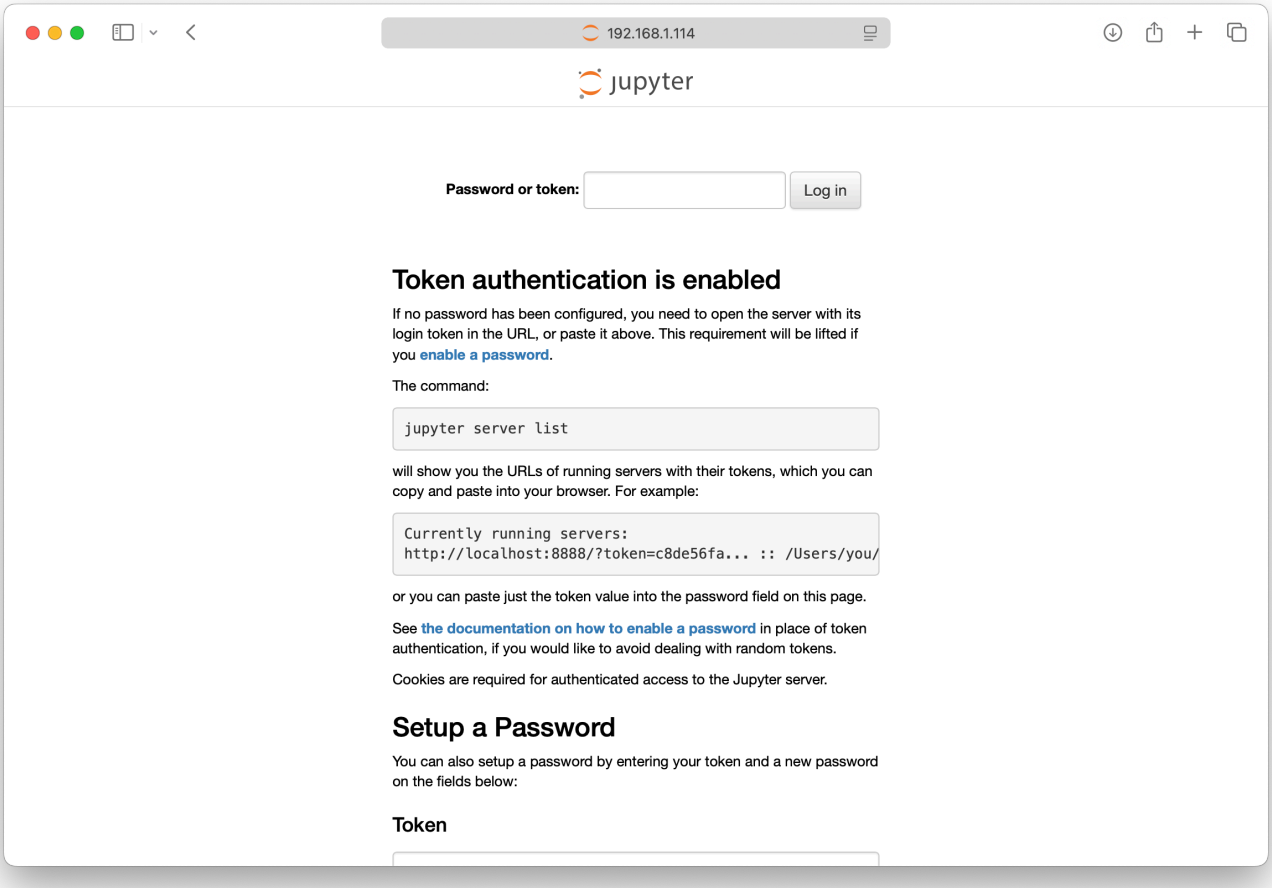
[3] Test Docker image locally

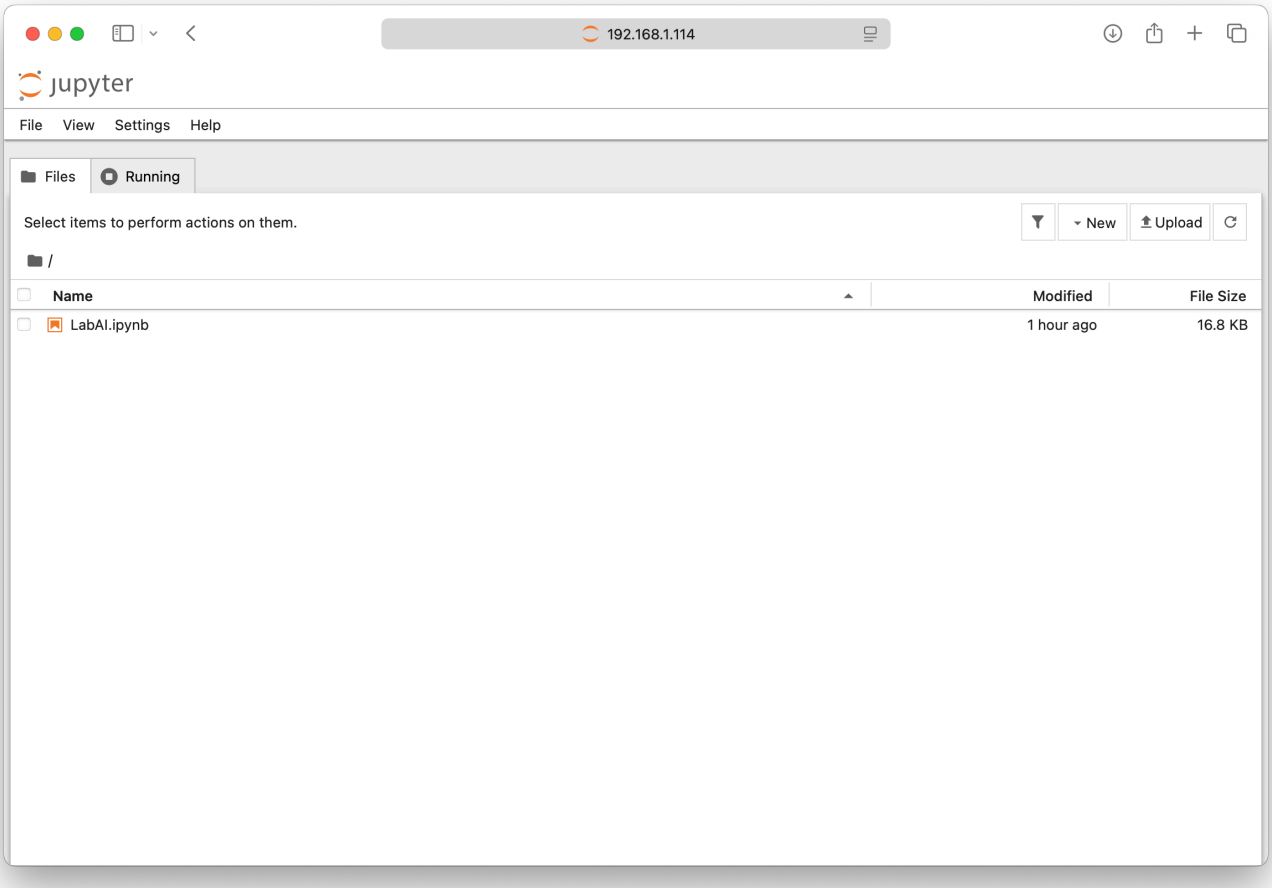
Before pushing to ECR, I tested the image locally to verify Jupyter Notebook starts correctly.

Command:

```
docker run -p 8888:8888 23917077-lab8
```

Result:


192.168.1.114
jupyter
Password or token:
Log in
Token authentication is enabled
If no password has been configured, you need to open the server with its login token in the URL, or paste it above. This requirement will be lifted if you enable a password.
The command:
jupyter server list
will show you the URLs of running servers with their tokens, which you can copy and paste into your browser. For example:
Currently running servers:
http://localhost:8888/?token=c8de56fa... :: /Users/you/
or you can paste just the token value into the password field on this page.
See the documentation on how to enable a password in place of token authentication, if you would like to avoid dealing with random tokens.
Cookies are required for authenticated access to the Jupyter server.
Setup a Password
You can also setup a password by entering your token and a new password on the fields below:
Token


192.168.1.114
jupyter
File
View
Settings
Help
Files
Running
Select items to perform actions on them.
New
Upload
/
Name
Modified
File Size
LabAI.ipynb
1 hour ago
16.8 KB

The Jupyter Notebook became accessible at `http://192.168.1.114:8888` with token "CITS5503", confirming the image works correctly.

Prepare ECR Repository

[1] Create ECR repository using boto3

ECR (Elastic Container Registry) is a managed Docker registry service. I created a Python script to create an ECR repository for storing the Docker image.

Implementation:

```
import boto3

def create_or_check_repository(repository_name):
    ecr_client = boto3.client('ecr', region_name='ap-northeast-2')
    try:
        response = ecr_client.describe_repositories(repositoryNames=[repository_name])
        repository_uri = response['repositories'][0]['repositoryUri']
        print(f"✓ Repository already exists: {repository_uri}")
    except ecr_client.exceptions.RepositoryNotFoundException:
        response = ecr_client.create_repository(repositoryName=repository_name)
        repository_uri = response['repository']['repositoryUri']
        print(f"✓ Repository created: {repository_uri}")
    return repository_uri

repository_name = '23917077_ecr_repo'
repository_uri = create_or_check_repository(repository_name)
print(f"ECR URI: {repository_uri}")
```

Command:

```
python3 create_ecr_repo.py
```

Result:

```
ECR URI: 489389878001.dkr.ecr.ap-northeast-2.amazonaws.com/23917077_ecr_repo
```

[2] Get Docker login credentials

To push images to ECR, Docker must authenticate with AWS. I created a script to generate the Docker login command.

Implementation:

```
import boto3
import base64

def get_docker_login_cmd():
    ecr_client = boto3.client('ecr', region_name='ap-northeast-2')
    token = ecr_client.get_authorisation_token()
    username, password = base64.b64decode(
        token['authorisationData'][0]['authorisationToken']
    ).decode().split(':')
    registry = token['authorisationData'][0]['proxyEndpoint']
    return f"docker login -u {username} -p {password} {registry}"

print(get_docker_login_cmd())
```

Command:

```
python3 get_docker_login.py
```

And I then executed the generated Docker login command in the terminal.

Result:

Push Docker Image to ECR

[1] Tag Docker image with ECR URI

Before pushing to ECR, the local image must be tagged with the ECR repository URI.

Command:

```
docker tag 23917077-lab8:latest 489389878001.dkr.ecr.ap-northeast-
2.amazonaws.com/23917077_ecr_repo:latest
```

This command tags the local Docker image `23917077-lab8:latest` with the full ECR repository URI, preparing it for upload to AWS ECR.

[2] Push image to ECR

With the image tagged, I pushed it to the ECR repository.

Command:

```
docker push 489389878001.dkr.ecr.ap-northeast-2.amazonaws.com/23917077_ecr_repo:latest
```

The Docker image is now stored in ECR and ready for deployment to ECS.

Deploy Docker Image to ECS

[1] Create ECS task definition

ECS requires a task definition that specifies how to run the container. I created a Python script to register the task definition with Fargate launch type.

Implementation:

```python
import boto3

def create_ecs_task_definition(
    client, image_uri, account_id, task_role_name, execution_role_name, student_id,
    port=8888, cpu='256', memory='512'
):
    task_role_arn = f'arn:aws:iam::{account_id}:role/{task_role_name}'
    execution_role_arn = f'arn:aws:iam::{account_id}:role/{execution_role_name}'

    response = client.register_task_definition(
        family=f'{student_id}-task-family',
        networkMode='awsvpc',
        requiresCompatibilities=['FARGATE'],
        cpu=cpu,
        memory=memory,
        taskRoleArn=task_role_arn,
        executionRoleArn=execution_role_arn,
        containerDefinitions=[
            {
                'name': f'{student_id}-container',
                'image': image_uri,
                'essential': True,
                'portMappings': [
                    {
                        'containerPort': port,
                        'hostPort': port,
                        'protocol': 'tcp'
                    },
                ],
```

```
            },
        ],
    )
    return response

account_id = '489389878001'
student_id = "23917077"
task_role_name = 'SageMakerRole'
execution_role_name = 'ecsTaskExecutionRole'
image_uri = '489389878001.dkr.ecr.ap-northeast-2.amazonaws.com/23917077_ecr_repo:latest'

ecs_client = boto3.client('ecs', region_name='ap-northeast-2')

task_definition = create_ecs_task_definition(
    ecs_client,
    image_uri,
    account_id,
    task_role_name,
    execution_role_name,
    student_id,
    port=8888
)
print("Task Definition ARN:", task_definition['taskDefinition']['taskDefinitionArn'])
```

Key parameters:

- `networkMode='awsvpc'`: Required for Fargate. Provides each task with its own elastic network interface (ENI), enabling: direct VPC integration, security group assignment per task (not per instance), and dedicated IP addresses. This isolation improves security by preventing container-to-container communication unless explicitly allowed.
- `requiresCompatibilities=['FARGATE']`: Specifies serverless Fargate launch type. **Fargate rationale**: Eliminates EC2 instance management (patching, scaling, capacity planning), charges per-second for resources actually used, automatically handles infrastructure, and reduces operational overhead—ideal for short-lived ML training workloads.
- `cpu='256'`: Allocates 0.25 vCPU
- `memory='512'`: Allocates 512 MB RAM
- `taskRoleArn`: IAM role for the container to access AWS services (SageMaker)
- `executionRoleArn`: IAM role for ECS to pull images and write logs

Command:

```
python3 create_task_definition.py
```

Result:

```
Task Definition ARN: arn:aws:ecs:ap-northeast-2:489389878001:task-definition/23917077-task-family:1
```

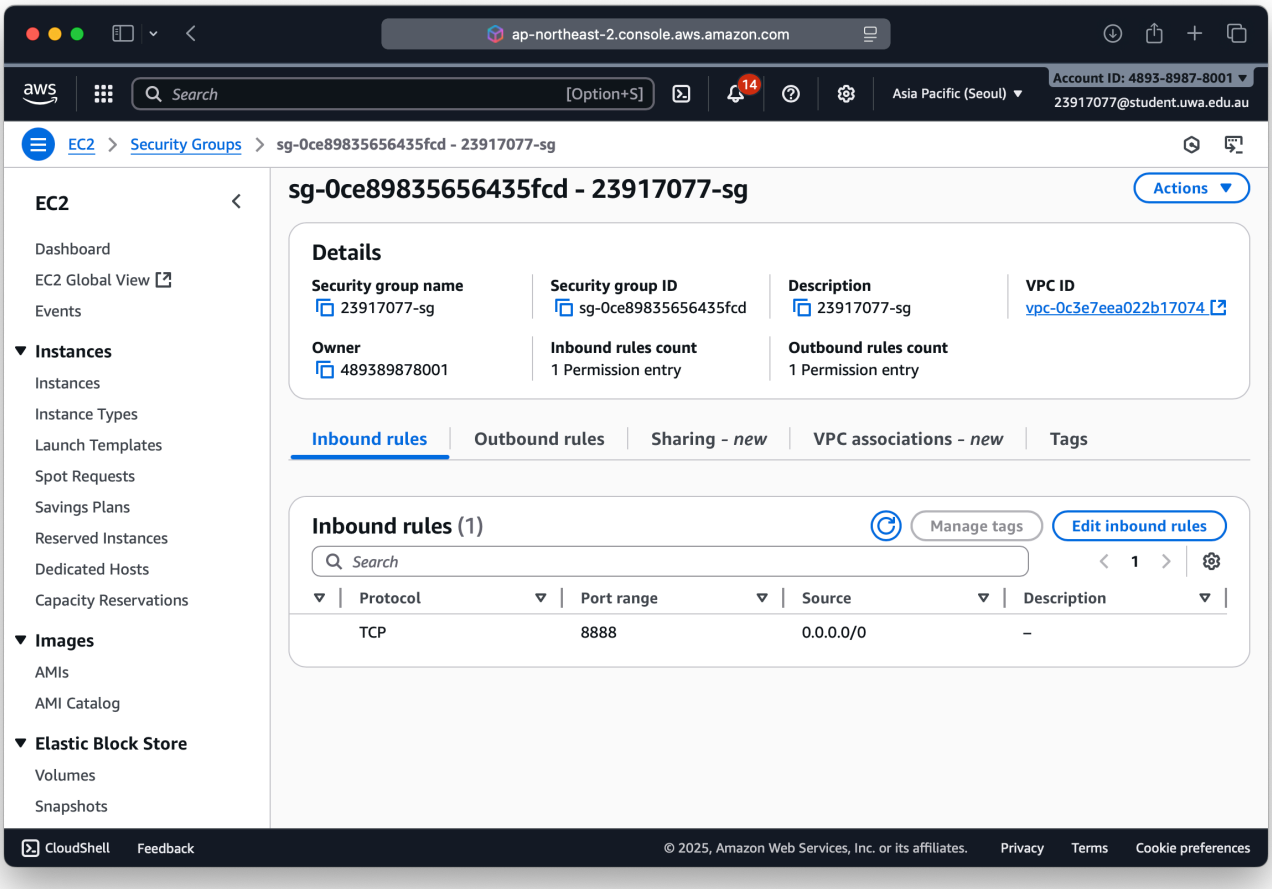
[2] Create security group

Before deploying the ECS service, I configured a security group in the AWS Console to allow the required traffic.

Security group rules:

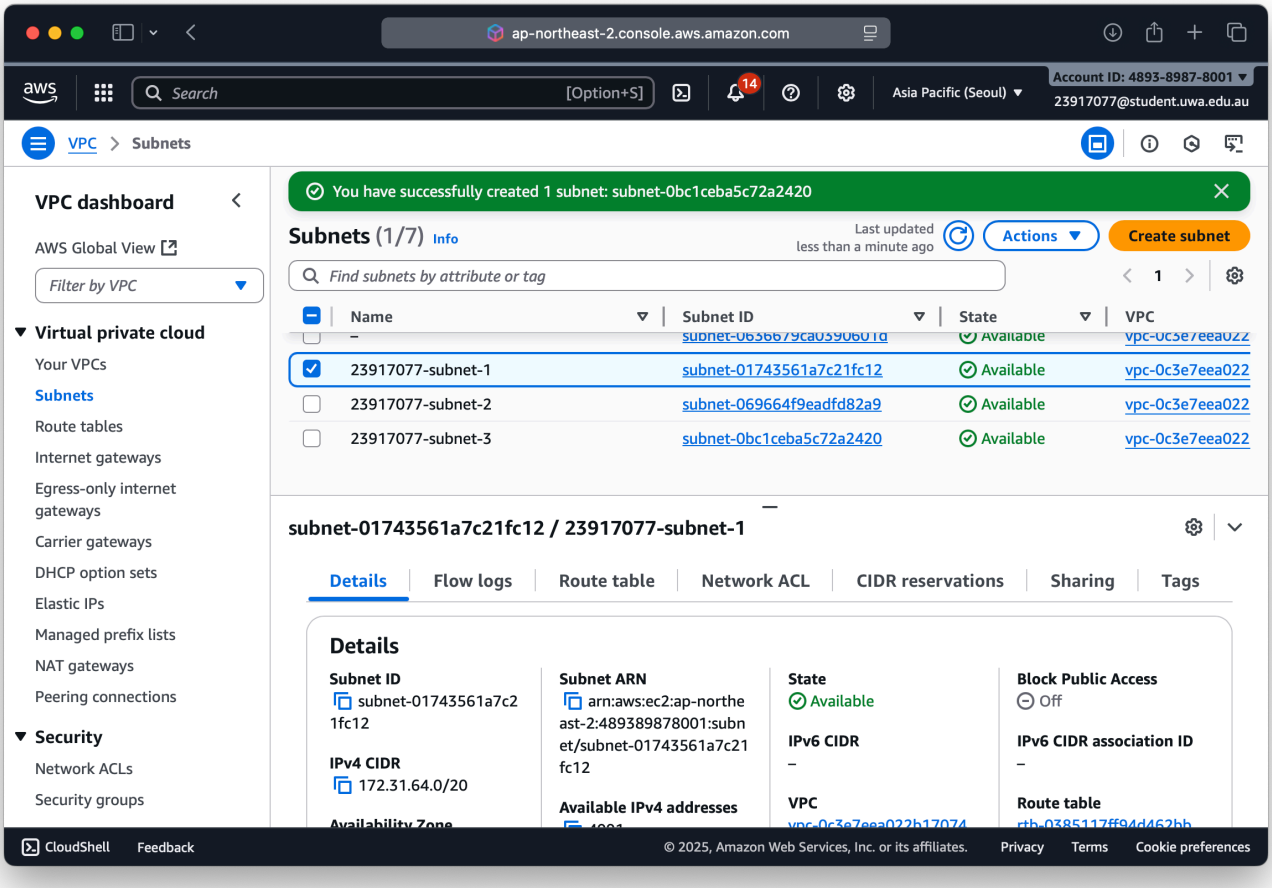
- **Inbound**: TCP port 8888 from 0.0.0.0/0 (allows Jupyter Notebook access)
- **Outbound**: HTTPS port 443 to 0.0.0.0/0 (allows AWS API calls)

Result:



[3] Create ECS cluster and service

Before running the script to create ecs cluster and services, I created 3 subnets on VPC dashboard.



With the task definition and security group ready, I created an ECS cluster and service to run the container.

Implementation:

```
import boto3

def create_ecs_cluster(client, cluster_name):
    response = client.create_cluster(clusterName=cluster_name)
    return response

def create_ecs_service(client, cluster_name, service_name, task_definition, subnet_ids,
security_group_ids):
    response = client.create_service(
        cluster=cluster_name,
        serviceName=service_name,
        taskDefinition=task_definition,
        desiredCount=1,
        launchType='FARGATE',
        networkConfiguration={
            'awsvpcConfiguration': {
                'subnets': subnet_ids,
                'securityGroups': security_group_ids,
                'assignPublicIp': 'ENABLED'
            }
        },
        deploymentConfiguration={
```

```
            'maximumPercent': 200,
            'minimumHealthyPercent': 100
        }
    )
    return response

def wait_for_service_stability(client, cluster_name, service_name):
    waiter = client.get_waiter('services_stable')
    waiter.wait(cluster=cluster_name, services=[service_name])

ecs_client = boto3.client('ecs', region_name='ap-northeast-2')

student_id = "23917077"
cluster_name = student_id + '-cluster'
create_ecs_cluster(ecs_client, cluster_name)
print(f"✓ Cluster created: {cluster_name}")

service_name = student_id + '-service'
task_definition = 'arn:aws:ecs:ap-northeast-2:489389878001:task-definition/23917077-task-family:3'
subnet_ids = ['subnet-01743561a7c21fc12', 'subnet-069664f9eadfd82a9', 'subnet-0bc1ceba5c72a2420']
security_group_ids = ['sg-0ce89835656435fcd']

service_response = create_ecs_service(
    ecs_client, cluster_name, service_name, task_definition, subnet_ids,
security_group_ids
)
print(f'✓ ECS Service created: {service_response["service"]["serviceArn"]}')

print(f'Waiting for service {service_name} to become stable...')
wait_for_service_stability(ecs_client, cluster_name, service_name)
print(f'✓ Service {service_name} is now stable.')
```

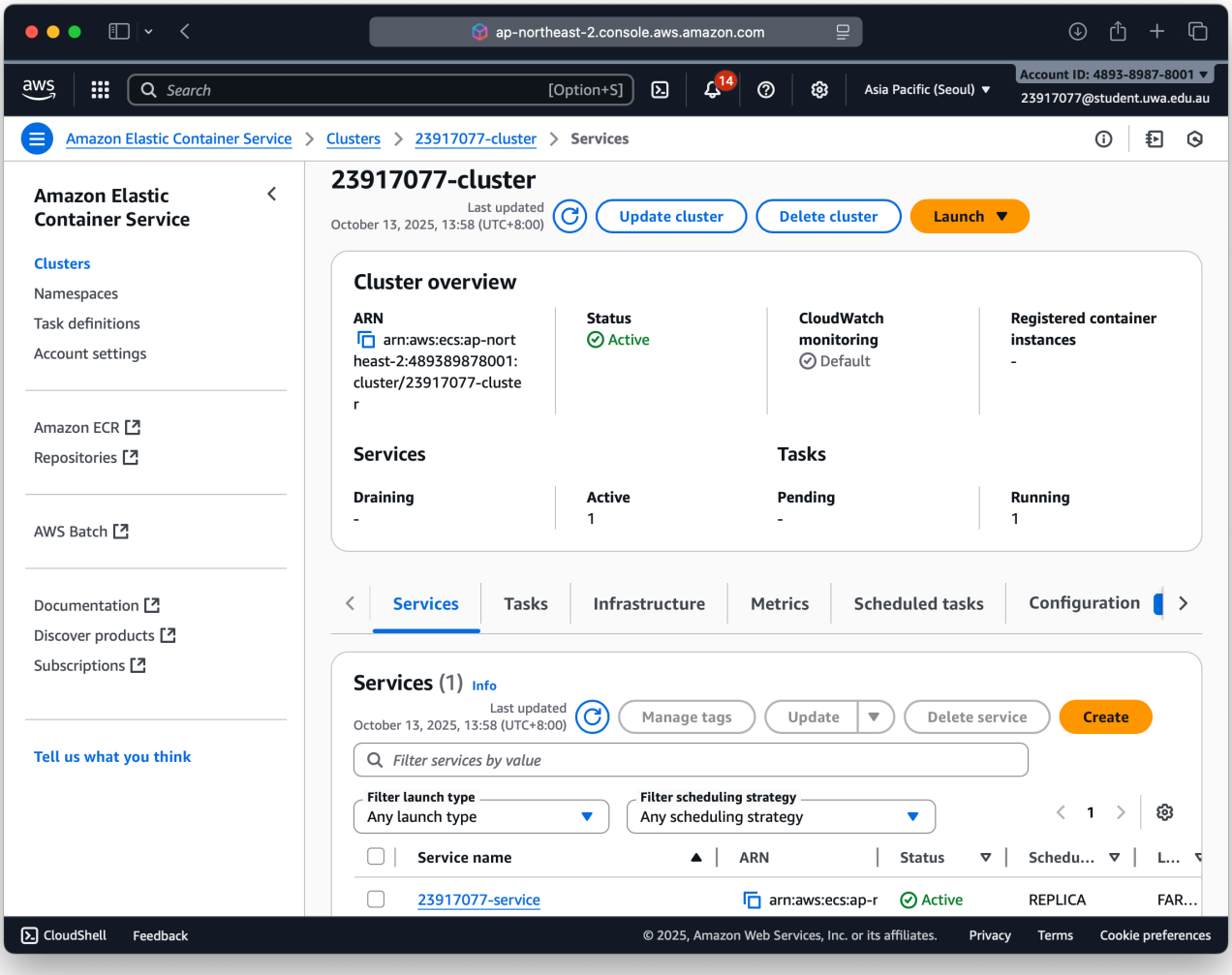
Key parameters:

- `desiredCount=1`: Runs one instance of the task
- `launchType='FARGATE'`: Uses serverless Fargate (no EC2 instance management)
- `assignPublicIp='ENABLED'`: Assigns a public IP to access Jupyter Notebook

Command:

```
python3 create_ecs_service.py
```

Result:



[4] Get public IP address

After the service becomes stable, I retrieved the public IP address of the running task to access Jupyter Notebook.

Command:

```
aws ecs describe-tasks \
    --cluster 23917077-cluster \
    --tasks $(aws ecs list-tasks --cluster 23917077-cluster --service-name 23917077-service --query 'taskArns[0]' --output text) \
    --query 'tasks[0].attachments[0].details[?name==`networkInterfaceId`].value' \
    --output text | xargs -I {} aws ec2 describe-network-interfaces \
    --network-interface-ids {} \
    --query 'NetworkInterfaces[0].Association.PublicIp' \
    --output text
```

Result:

```
3.38.177.3
```

Run Hyperparameter Tuning Job in Jupyter Notebook

[1] Access Jupyter Notebook via ECS

With the ECS task running, I accessed the Jupyter Notebook using the public IP address and token.

Access URL:

```
http://3.38.177.3:8888
```

Token: CITS5503

[2] Execute hyperparameter tuning job

Inside the Jupyter Notebook, I executed the cells in LabAI.ipynb to perform hyperparameter tuning using Amazon SageMaker. The notebook walks through data preparation, feature engineering, model training configuration, and launching the hyperparameter tuning job.

Step 1: Setup Environment Variables

The first step securely configures AWS credentials for the notebook session. The `getpass` function ensures the secret access key is hidden during input for security.

```
import os
from getpass import getpass

print("Please enter your AWS credentials for this lab. Your credentials are only used in
this notebook session.")

os.environ['AWS_ACCESS_KEY_ID'] = input('AWS_ACCESS_KEY_ID: ')
os.environ['AWS_SECRET_ACCESS_KEY'] = getpass('AWS_SECRET_ACCESS_KEY (hidden): ')
os.environ['AWS_DEFAULT_REGION'] = input('AWS_DEFAULT_REGION: ') # e.g. us-west-2, us-
east-1, etc.
```

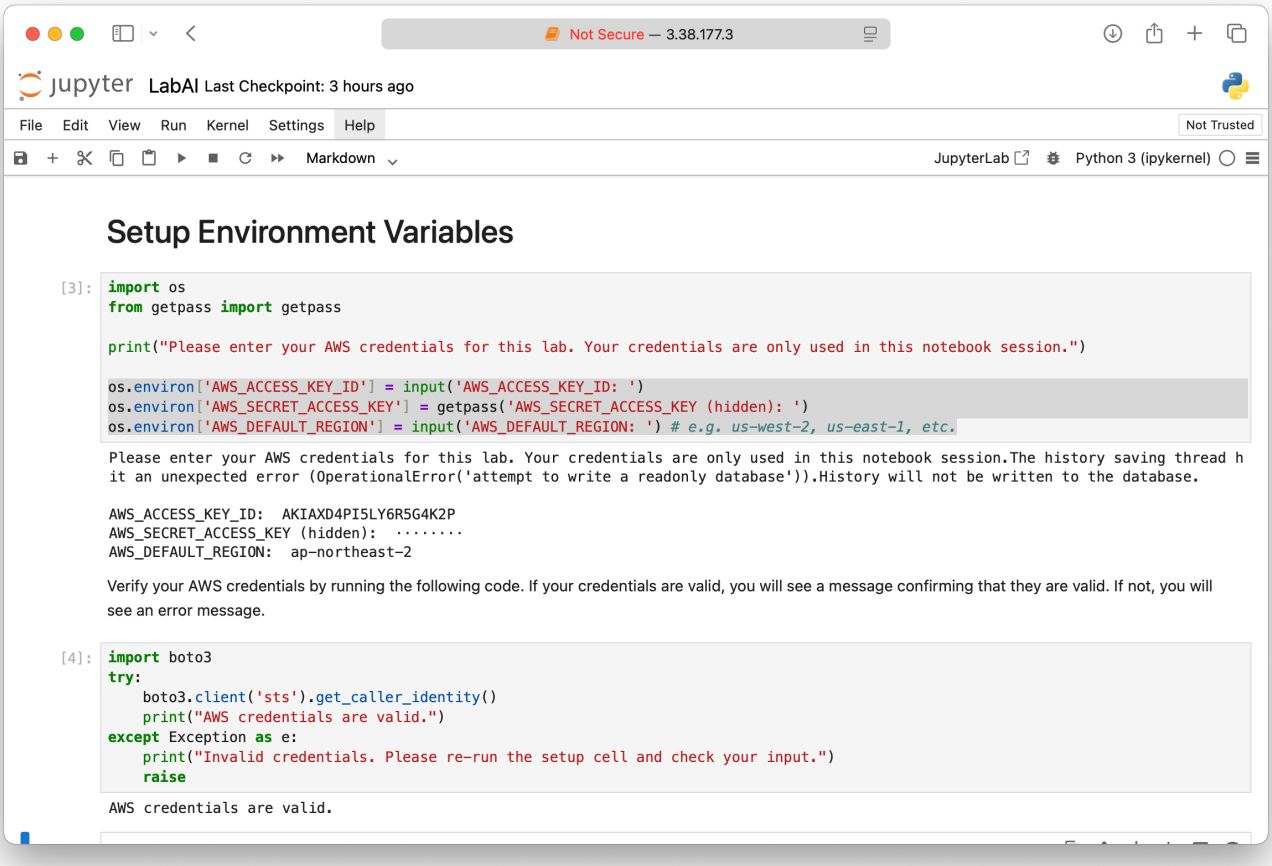
Credentials entered:

- AWS_ACCESS_KEY_ID: `AKIAXD4PI5LY6R5G4K2P`
- AWS_SECRET_ACCESS_KEY: `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` (hidden)
- AWS_DEFAULT_REGION: `ap-northeast-2`

After entering credentials, I verified them using boto3's STS service:

```
import boto3
try:
    boto3.client('sts').get_caller_identity()
    print("AWS credentials are valid.")
except Exception as e:
    print("Invalid credentials. Please re-run the setup cell and check your input.")
    raise
```

Result:



The credentials were successfully validated, allowing the notebook to interact with AWS services.

Step 2: Install libraries

The notebook requires three essential Python libraries for data processing and machine learning operations:

```
# Install SageMaker via jupyter notebook
!pip3 install sagemaker
# Install pandas and numpy jupyter notebook
!pip3 install pandas
!pip3 install numpy
```

- **sagemaker**: AWS SDK for building, training, and deploying machine learning models
- **pandas**: Data manipulation and analysis library for structured data
- **numpy**: Fundamental package for numerical computing and array operations

These libraries were successfully installed in the Jupyter environment, providing the necessary tools for the hyperparameter tuning workflow.

Step 3: Prepare a SageMaker session

```
import sagemaker
```

```
import numpy as np  # For matrix operations and numerical processing
import pandas as pd  # For munging tabular data
from time import gmtime, strftime

smclient = boto3.Session().client("sagemaker")
sagemaker_role = "arn:aws:iam::489389878001:role/SageMakerRole"
region = os.environ['AWS_DEFAULT_REGION']
student_id = "23917077"  # use your student id
bucket = "23917077-lab8"  # use <studentid-lab8> as your bucket name
prefix = f"sagemaker/{student_id}-hpo-xgboost-dm"

s3_client = boto3.client("s3")
# Create a folder (prefix) in the bucket
try:
    s3_client.put_object(Bucket=bucket, Key=f"{prefix}/")
    print(f"Folder {prefix}/ created successfully")
except Exception as e:
    print(f"Error creating folder: {e}")
```

Result:

```
Folder sagemaker/23917077-hpo-xgboost-dm/ created successfully
```

The S3 prefix (folder) was successfully created to organize training data, validation data, and model outputs.

Step 4: Download a dataset

I downloaded the Bank Marketing dataset from the UCI Machine Learning Repository. This dataset contains information about direct marketing campaigns of a Portuguese banking institution.

```
!wget -N https://archive.ics.uci.edu/ml/machine-learning-databases/00222/bank-additional.zip
!unzip -o bank-additional.zip
```

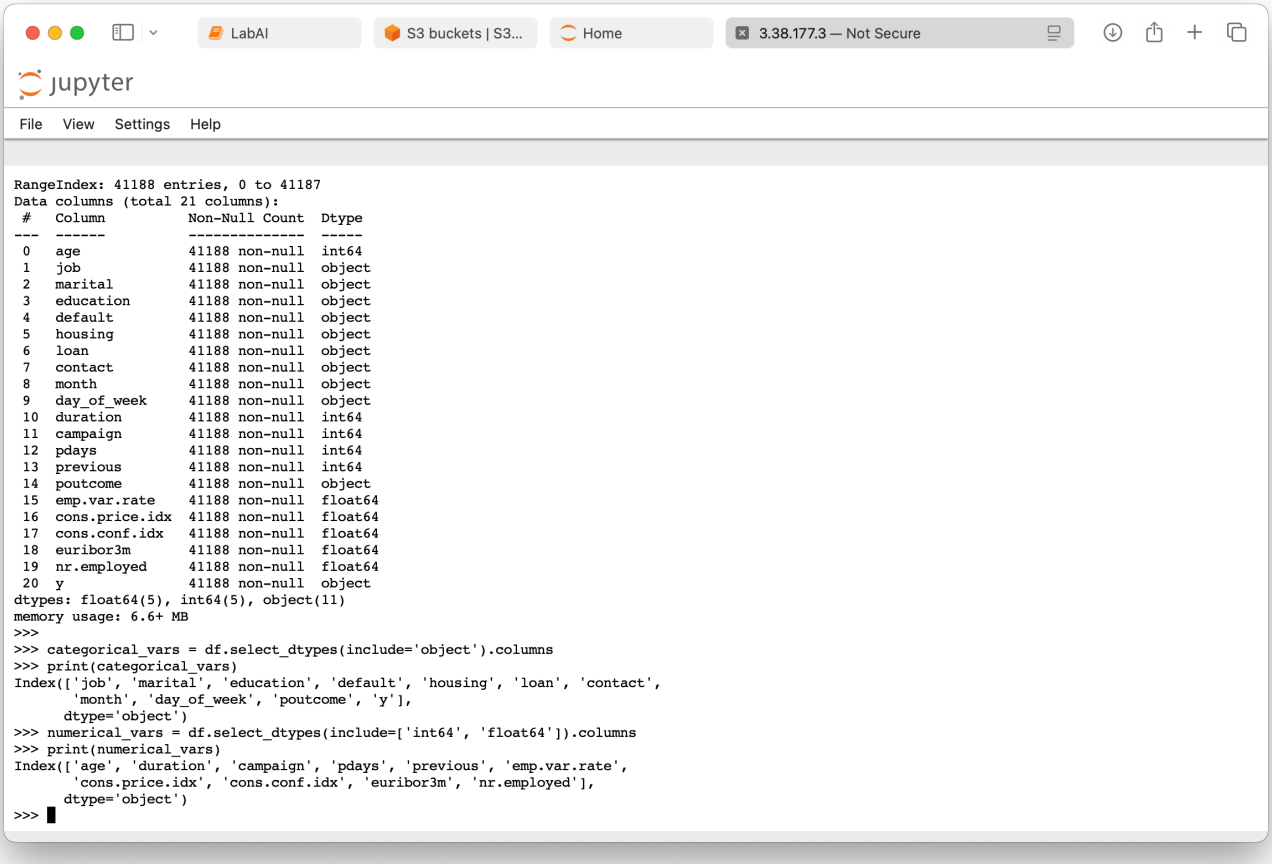
The dataset was successfully downloaded and extracted, containing customer information and whether they subscribed to a term deposit.

Dataset Analysis:

After loading the dataset into a Pandas DataFrame, I identified the variable types:

1. **Categorical variables**: job, marital, education, default, housing, loan, contact, month, day_of_week, poutcome, y
2. **Numerical variables**: age, duration, campaign, pdays, previous, emp.var.rate, cons.price.idx, cons.conf.idx, euribor3m, nr.employed

Result:



The dataset contains 41,188 records with 20 input features and 1 target variable (`y`), indicating whether the client subscribed to a term deposit.

Load and display dataset:

```
data = pd.read_csv("./bank-additional/bank-additional-full.csv", sep=";")
pd.set_option("display.max_columns", 500)  # Make sure we can see all of the columns
pd.set_option("display.max_rows", 50)  # Keep the output on one page
data
```

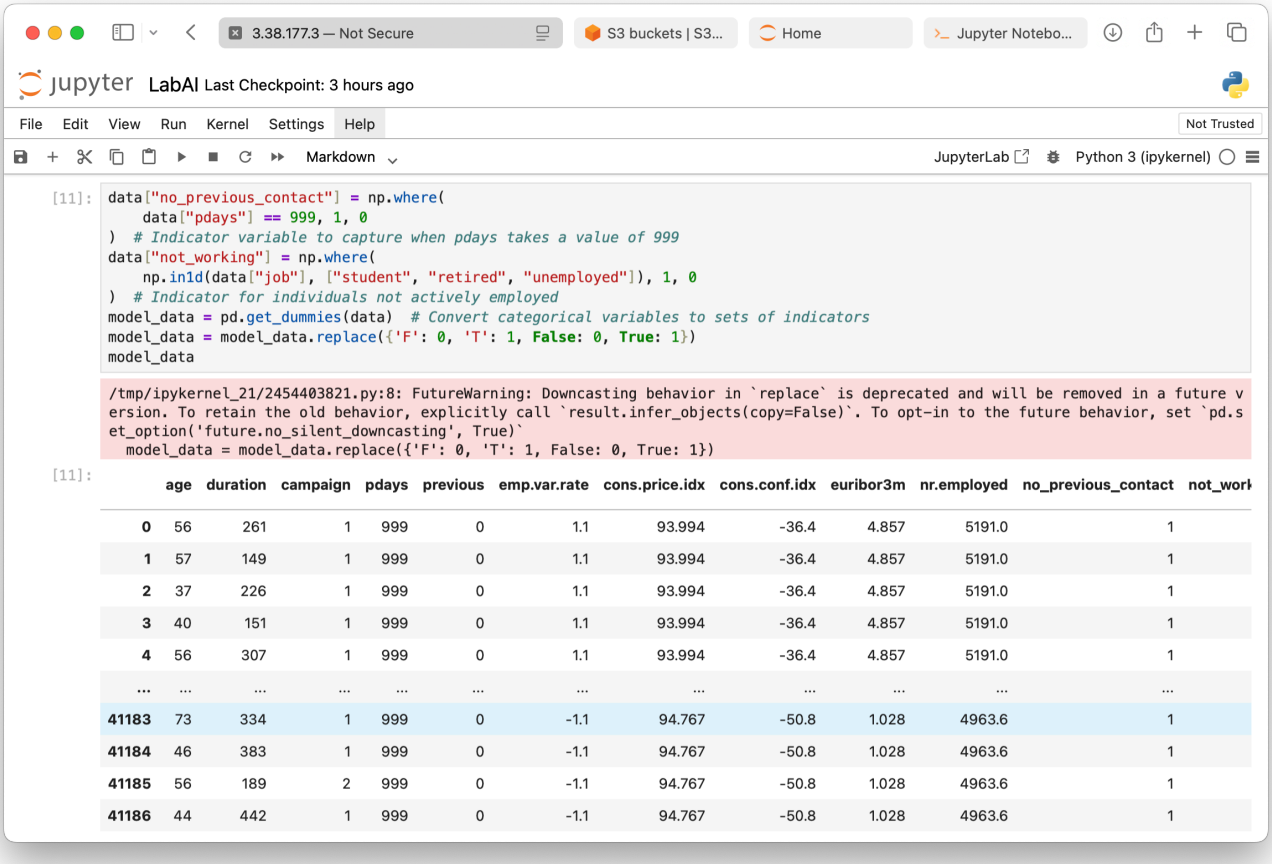
Feature Engineering:

I performed feature engineering to improve model performance by adding indicator variables and converting categorical features:

```
data["no_previous_contact"] = np.where(
    data["pdays"] == 999, 1, 0
)  # Indicator variable to capture when pdays takes a value of 999
data["not_working"] = np.where(
    np.in1d(data["job"], ["student", "retired", "unemployed"]), 1, 0
)  # Indicator for individuals not actively employed
model_data = pd.get_dummies(data)  # Convert categorical variables to sets of indicators
model_data = model_data.replace({'F': 0, 'T': 1, False: 0, True: 1})
model_data
```

- **no_previous_contact**: Captures clients who were never previously contacted (pdays = 999)
- **not_working**: Identifies clients who are students, retired, or unemployed
- **get_dummies()**: Converts categorical variables into binary indicator columns (one-hot encoding)

Result:



	age	duration	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed	no_previous_contact	not_work
0	56	261	1	999	0	1.1	93.994	-36.4	4.857	5191.0	1	
1	57	149	1	999	0	1.1	93.994	-36.4	4.857	5191.0	1	
2	37	226	1	999	0	1.1	93.994	-36.4	4.857	5191.0	1	
3	40	151	1	999	0	1.1	93.994	-36.4	4.857	5191.0	1	
4	56	307	1	999	0	1.1	93.994	-36.4	4.857	5191.0	1	
...	...	...	...	...	...	...	...	...	...	...	...	
41183	73	334	1	999	0	-1.1	94.767	-50.8	1.028	4963.6	1	
41184	46	383	1	999	0	-1.1	94.767	-50.8	1.028	4963.6	1	
41185	56	189	2	999	0	-1.1	94.767	-50.8	1.028	4963.6	1	
41186	44	442	1	999	0	-1.1	94.767	-50.8	1.028	4963.6	1	

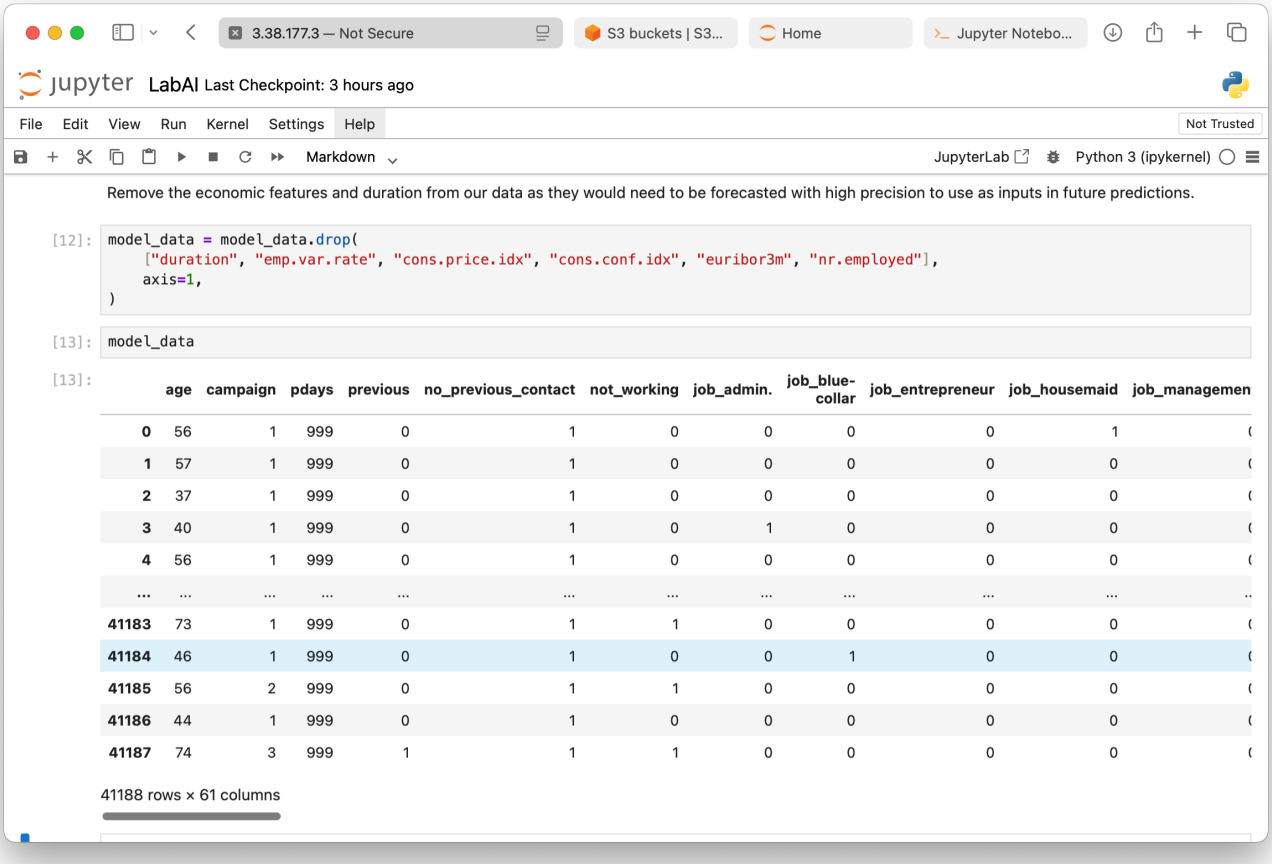
Remove problematic features:

I removed features that could cause data leakage or add noise to the model:

```
model_data = model_data.drop(
    ["duration", "emp.var.rate", "cons.price.idx", "cons.conf.idx", "euribor3m",
"nr.employed"],
    axis=1,
)
model_data
```

- **duration**: Causes target leakage (only known after call ends, highly correlated with outcome)
- **Economic indicators** (emp.var.rate, cons.price.idx, etc.): External economic features that may not generalize well

Result:



	age	campaign	pdays	previous	no_previous_contact	not_working	job_admin.	job_blue-collar	job_entrepreneur	job_housemaid	job_managemen
0	56	1	999	0	1	0	0	0	0	1	0
1	57	1	999	0	1	0	0	0	0	0	0
2	37	1	999	0	1	0	0	0	0	0	0
3	40	1	999	0	1	0	1	0	0	0	0
4	56	1	999	0	1	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...
41183	73	1	999	0	1	1	0	0	0	0	0
41184	46	1	999	0	1	0	0	1	0	0	0
41185	56	2	999	0	1	1	0	0	0	0	0
41186	44	1	999	0	1	0	0	0	0	0	0
41187	74	3	999	1	1	1	0	0	0	0	0

After feature engineering and cleaning, the dataset is ready for model training.

Step 5: Split the data into training, validation and test

For SageMaker XGBoost training, I split the dataset into three subsets and prepared them in CSV format with the target variable as the first column (required by XGBoost).

```
train_data, validation_data, test_data = np.split(
    model_data.sample(frac=1, random_state=1729),
    [int(0.7 * len(model_data)), int(0.9 * len(model_data))],
)

pd.concat([train_data["y_yes"], train_data.drop(["y_no", "y_yes"], axis=1)],
axis=1).to_csv(
    "train.csv", index=False, header=False
)
pd.concat(
    [validation_data["y_yes"], validation_data.drop(["y_no", "y_yes"], axis=1)], axis=1
).to_csv("validation.csv", index=False, header=False)
pd.concat([test_data["y_yes"], test_data.drop(["y_no", "y_yes"], axis=1)], axis=1).to_csv(
    "test.csv", index=False, header=False
)
```

Data split proportions:

- **Training set**: 70% of data (28,831 records)

- **Validation set**: 20% of data (8,238 records)
- **Test set**: 10% of data (4,119 records)

Important preprocessing steps:

- Shuffled data with `random_state=1729` for reproducibility
- Moved target variable (`y_yes`) to first column as required by SageMaker XGBoost
- Removed redundant `y_no` column (since we only need one target indicator)
- Saved without headers or index (CSV format required by XGBoost)

Upload to S3:

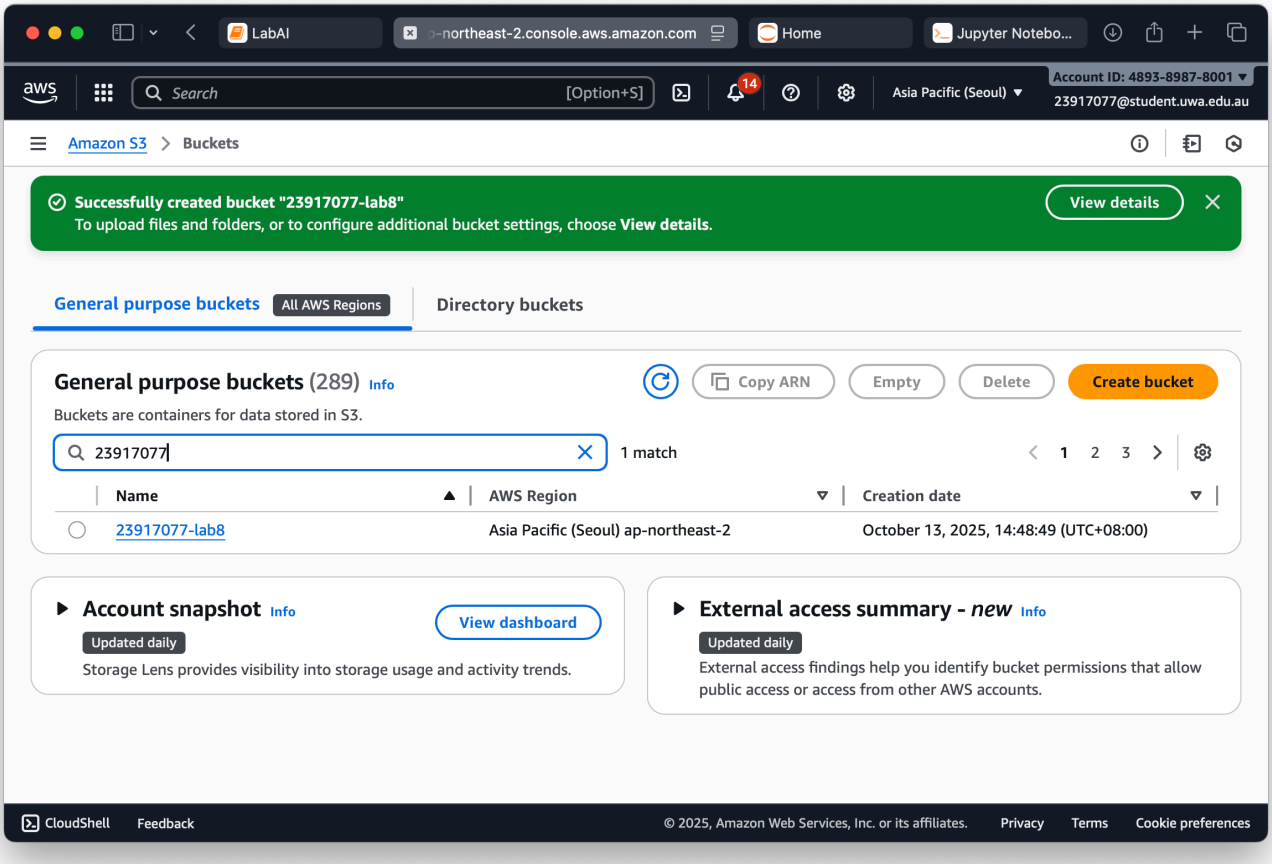
After creating the CSV files locally, I uploaded them to the S3 bucket:

```
boto3.Session().resource("s3").Bucket(bucket).Object(
    os.path.join(prefix, "train/train.csv")
).upload_file("train.csv")
boto3.Session().resource("s3").Bucket(bucket).Object(
    os.path.join(prefix, "validation/validation.csv")
).upload_file("validation.csv")
```

S3 paths:

- Training data: `s3://23917077-lab8/sagemaker/23917077-hpo-xgboost-dm/train/train.csv`
- Validation data: `s3://23917077-lab8/sagemaker/23917077-hpo-xgboost-dm/validation/validation.csv`

Result:



The training and validation datasets were successfully uploaded to S3, ready for SageMaker to access during the hyperparameter tuning job.

Step 6: Setup hyperparameter tuning

I configured the hyperparameter tuning job with parameter ranges, resource limits, and optimization strategy:

```
from time import gmtime, strftime, sleep

# Names have to be unique. You will get an error if you reuse the same name
tuning_job_name = f"{student_id}-xgboost-tuningjob-01"

print(tuning_job_name)

tuning_job_config = {
    "ParameterRanges": {
        "CategoricalParameterRanges": [],
        "ContinuousParameterRanges": [
            {
                "MaxValue": "1",
                "MinValue": "0",
                "Name": "eta",
            },
            {
                "MaxValue": "10",
```

```
                "MinValue": "1",
                "Name": "min_child_weight",
            },
            {
                "MaxValue": "2",
                "MinValue": "0",
                "Name": "alpha",
            },
        ],
        "IntegerParameterRanges": [
            {
                "MaxValue": "10",
                "MinValue": "1",
                "Name": "max_depth",
            }
        ],
    },
    "ResourceLimits": {"MaxNumberOfTrainingJobs": 2, "MaxParallelTrainingJobs": 2},
    "Strategy": "Bayesian",
    "HyperParameterTuningJobObjective": {"MetricName": "validation:auc", "Type":
"Maximize"},
}
```

Output:

```
23917077-xgboost-tuningjob-01
```

Hyperparameter ranges configured:

1. **eta** (learning rate): [0, 1] - Continuous
    - Controls step size shrinkage to prevent overfitting
    - Lower values make training more conservative
2. **min_child_weight**: [1, 10] - Continuous
    - Minimum sum of instance weight needed in a child node
    - Higher values prevent learning overly specific patterns
3. **alpha** (L1 regularization): [0, 2] - Continuous
    - L1 regularization term on weights
    - Helps prevent overfitting through feature selection
4. **max_depth**: [1, 10] - Integer
    - Maximum depth of decision trees
    - Deeper trees capture complex patterns but may overfit

Tuning configuration:

- **Strategy**: Bayesian optimization (intelligently selects next hyperparameters based on previous results)
- **Objective metric**: validation:auc (Area Under ROC Curve) - Maximize
- **Resource limits**: Maximum 2 training jobs, run 2 in parallel

Define training job configuration:

I configured the training job definition specifying the XGBoost algorithm, data sources, compute resources, and static hyperparameters:

```
from sagemaker.image_uris import retrieve
# Use XGBoost algorithm for training
training_image = retrieve(framework="xgboost", region=region, version="latest")

s3_input_train = "s3://{}/{}/train".format(bucket, prefix)
s3_input_validation = "s3://{}/{}/validation/".format(bucket, prefix)

training_job_definition = {
    "AlgorithmSpecification": {"TrainingImage": training_image, "TrainingInputMode":
"File"},
    "InputDataConfig": [
        {
            "ChannelName": "train",
            "CompressionType": "None",
            "ContentType": "csv",
            "DataSource": {
                "S3DataSource": {
                    "S3DataDistributionType": "FullyReplicated",
                    "S3DataType": "S3Prefix",
                    "S3Uri": s3_input_train,
                }
            },
        },
        {
            "ChannelName": "validation",
            "CompressionType": "None",
            "ContentType": "csv",
            "DataSource": {
                "S3DataSource": {
                    "S3DataDistributionType": "FullyReplicated",
                    "S3DataType": "S3Prefix",
                    "S3Uri": s3_input_validation,
                }
            },
        },
    ],
    "OutputDataConfig": {"S3OutputPath": "s3://{}/{}/output".format(bucket, prefix)},
    "ResourceConfig": {"InstanceCount": 1, "InstanceType": "ml.m5.xlarge",
"VolumeSizeInGB": 10},
    "RoleArn": sagemaker_role,
    "StaticHyperParameters": {
        "eval_metric": "auc",
        "num_round": "1",
        "objective": "binary:logistic",
        "rate_drop": "0.3",
        "tweedie_variance_power": "1.4",
    },
    "StoppingCondition": {"MaxRuntimeInSeconds": 43200},
}
```

Training configuration details:

- **Algorithm**: XGBoost (latest version) from SageMaker image repository
- **Compute**: ml.m5.xlarge instance (4 vCPU, 16 GB RAM) with 10 GB storage
- **Training mode**: File mode (data downloaded before training starts)
- **Input channels**: train and validation data from S3
- **Output**: Model artifacts saved to S3
- **Static hyperparameters**: Fixed parameters not tuned (eval_metric=auc, objective=binary:logistic, etc.)
- **Max runtime**: 12 hours (43,200 seconds)

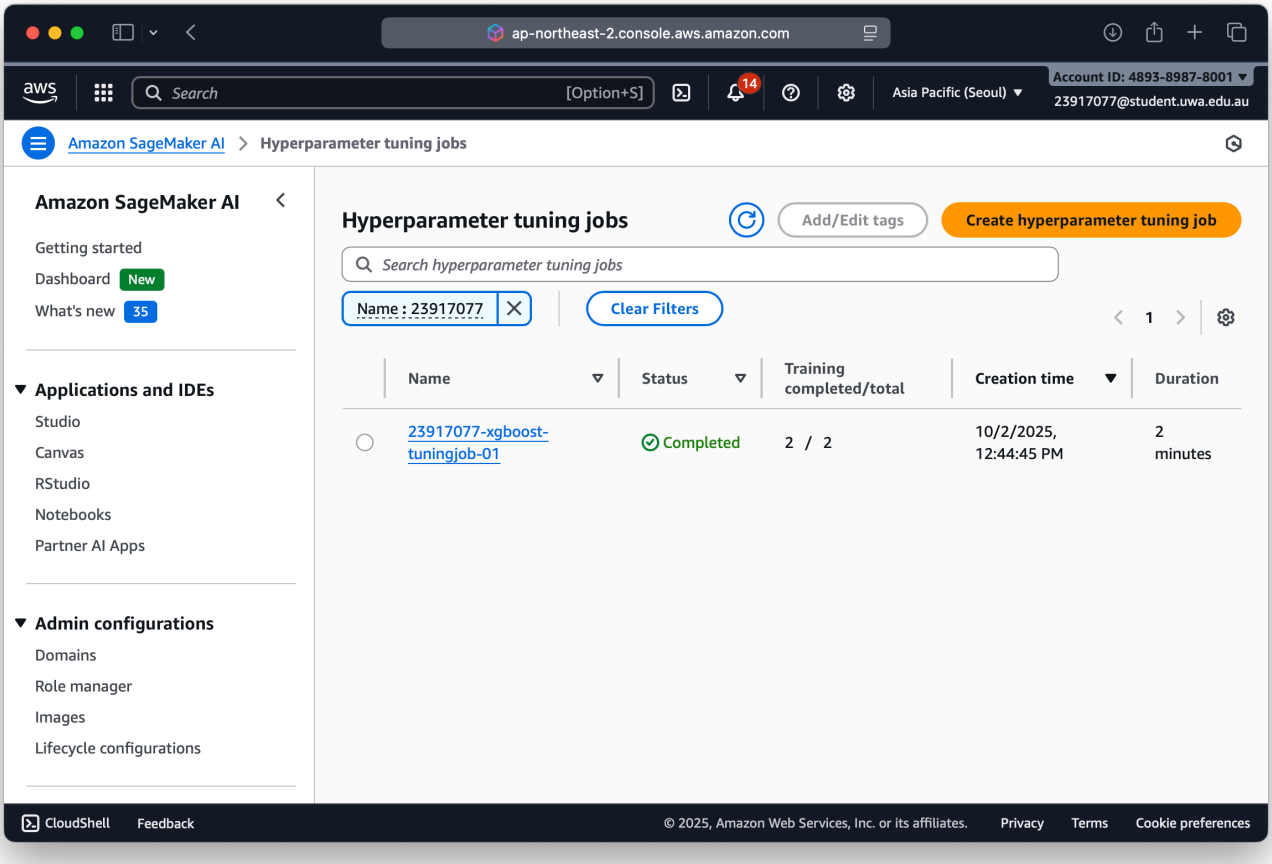
Launch hyperparameter tuning job:

With all configurations ready, I launched the tuning job to SageMaker:

```
# Launch Hyperparameter Tuning Job
smclient.create_hyper_parameter_tuning_job(
    HyperParameterTuningJobName=tuning_job_name,
    HyperParameterTuningJobConfig=tuning_job_config,
    TrainingJobDefinition=training_job_definition,
)
```

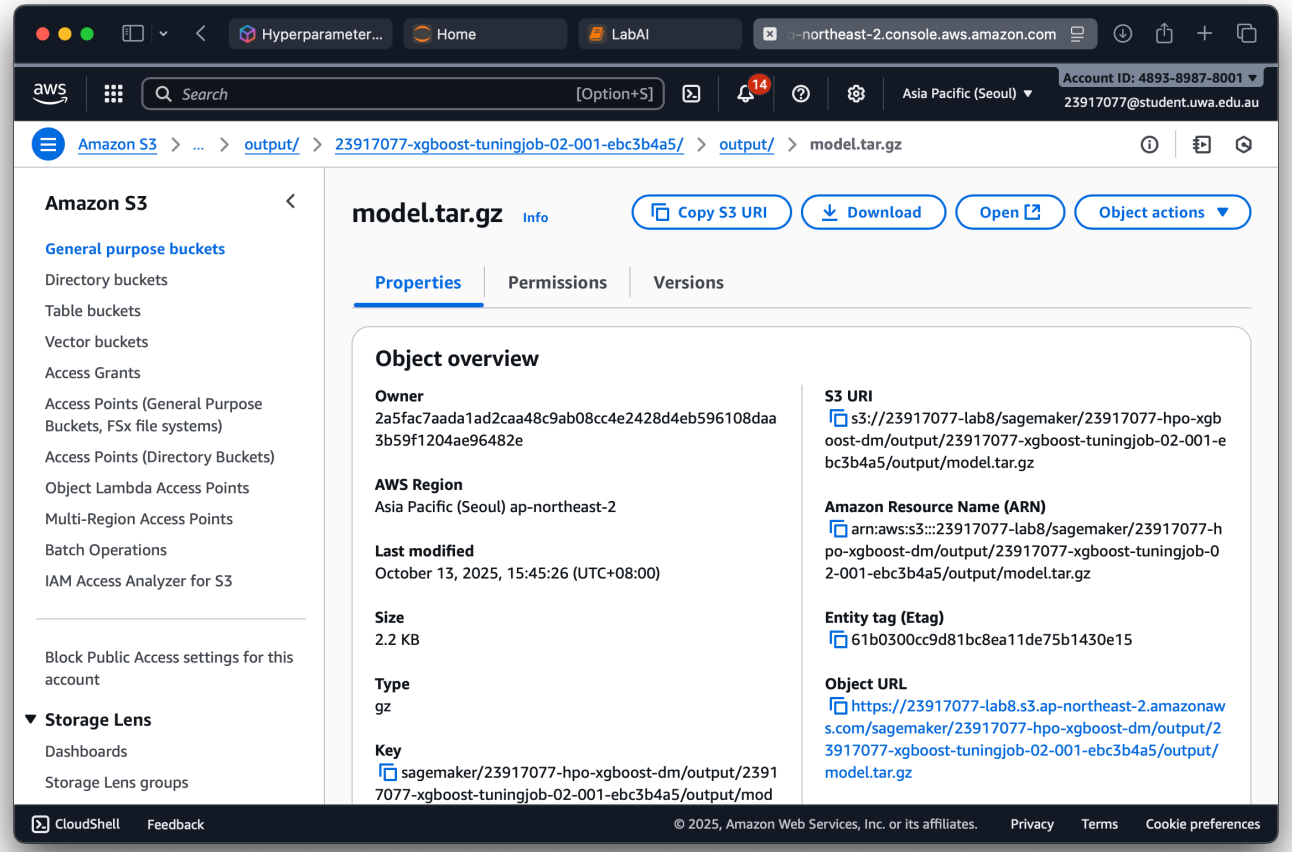
The hyperparameter tuning job was successfully submitted to SageMaker. The job runs 2 training jobs in parallel, each with different hyperparameter combinations selected by Bayesian optimization to maximize the validation AUC metric.

Results in SageMaker Console:



The SageMaker console shows the tuning job status, training jobs, and performance metrics. After completion, both training jobs finished successfully with their respective AUC scores.

Model outputs in S3:



The trained model artifacts (model.tar.gz files) were saved to S3 in the output folder. Each training job created its own subfolder containing the model file, which can be deployed to a SageMaker endpoint for inference or downloaded for local use.

Cleanup Resources

After completing Lab 8, I systematically deleted all created resources to avoid ongoing charges and maintain account cleanliness.

Resources deleted:

- ECS service and tasks
- ECS cluster
- ECR repository and Docker images
- S3 bucket with training data and model artifacts
- CloudWatch logs
- Security groups (if dedicated)
- Subnets (if created specifically for this lab)

Commands:

```
# Stop all running tasks first
aws ecs list-tasks --cluster 23917077-cluster --service-name 23917077-service --query 'taskArns[]' --output text | xargs -I {} aws ecs stop-task --cluster 23917077-cluster --task {}
```

```
# Delete ECS service (--force allows deletion with running tasks)
aws ecs delete-service --cluster 23917077-cluster --service 23917077-service --force

# Wait for service deletion
aws ecs wait services-inactive --cluster 23917077-cluster --services 23917077-service

# Delete ECS cluster
aws ecs delete-cluster --cluster 23917077-cluster

# Delete ECR repository (--force deletes even with images)
aws ecr delete-repository --repository-name 23917077_ecr_repo --force

# Delete S3 bucket and all contents
aws s3 rb s3://23917077-lab8 --force
```

Lab 9

AWS Comprehend - Natural Language Processing

[1] Detect Languages from Text

To detect languages from text using AWS Comprehend, I created a Python script that not only identifies the language code but also converts it to a human-readable language name with confidence percentage.

Implementation:

```
import boto3

# Language code to name mapping
LANGUAGE_NAMES = {
    'en': 'English',
    'es': 'Spanish',
    'fr': 'French',
    'it': 'Italian',
    'de': 'German',
    'zh': 'Chinese',
    'ja': 'Japanese',
    'ko': 'Korean',
}

def detect_language(text):
    """Detect language from text using AWS Comprehend"""
    client = boto3.client('comprehend', region_name='ap-northeast-2')

    # Detect dominant language
    response = client.detect_dominant_language(Text=text)

    # Get the most confident language
    if response['Languages']:
        language = response['Languages'][0]
        language_code = language['LanguageCode']
        confidence = language['Score']

        # Get language name
        language_name = LANGUAGE_NAMES.get(language_code, language_code)

        # Convert confidence to percentage
        confidence_percentage = int(confidence * 100)

        return f"{language_name} was detected with {confidence_percentage}% confidence"

    return "No language detected"
```

The script uses AWS Comprehend's `detect_dominant_language()` API to identify the language through statistical text analysis. **How it works**: Comprehend analyses character n-grams, word patterns, and linguistic features to match against trained models for 100+ languages.

Key improvements over basic example:

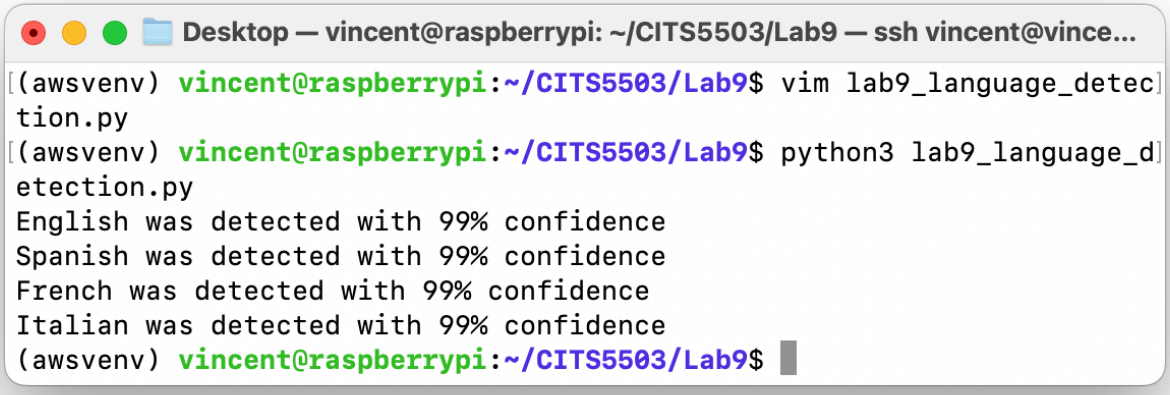
1. **Language name mapping**: Converts ISO 639-1 codes ('en', 'es', 'fr') to readable names using a dictionary lookup, improving user experience
2. **Percentage conversion**: Transforms confidence scores from decimal format (0.99) to intuitive percentages (99%)
3. **Formatted output**: Returns clear messages like "English was detected with 99% confidence"

API call explanation: `client.detect_dominant_language(Text=text)` returns a list of detected languages ranked by confidence. We select `response['Languages'][0]` (highest confidence match) for our result.

Command:

```
python3 lab9_language_detection.py
```

Result:




```
Text (English):
  The French Revolution was a period of social and political upheaval in France and its
colonies begin...
Result: English was detected with 99% confidence

Text (Spanish):
  El Quijote es la obra más conocida de Miguel de Cervantes Saavedra. Publicada su primera
parte con e...
Result: Spanish was detected with 99% confidence

Text (French):
  Moi je n'étais rien Et voilà qu'aujourd'hui Je suis le gardien Du sommeil de ses nuits
Je l'aime à m...
```

```
Result: French was detected with 99% confidence

Text (Italian):
  L'amor che move il sole e l'altre stelle.
Result: Italian was detected with 99% confidence
```

The language detection successfully identified all four languages with 99% confidence, demonstrating AWS Comprehend's accuracy across different language texts.

[2] Analyze Sentiment

Sentiment analysis determines whether text expresses positive, negative, neutral, or mixed emotions. I created a Python script using AWS Comprehend's `detect_sentiment()` API to analyze the emotional tone of texts in different languages.

Implementation:

```python
import boto3

def analyze_sentiment(text, language_code='en'):
    """Analyze sentiment of text using AWS Comprehend"""
    client = boto3.client('comprehend', region_name='ap-northeast-2')

    # Detect sentiment
    response = client.detect_sentiment(
        Text=text,
        LanguageCode=language_code
    )

    sentiment = response['Sentiment']
    scores = response['SentimentScore']

    return sentiment, scores
```

The script analyses sentiment using machine learning models trained on language-specific text corpora:

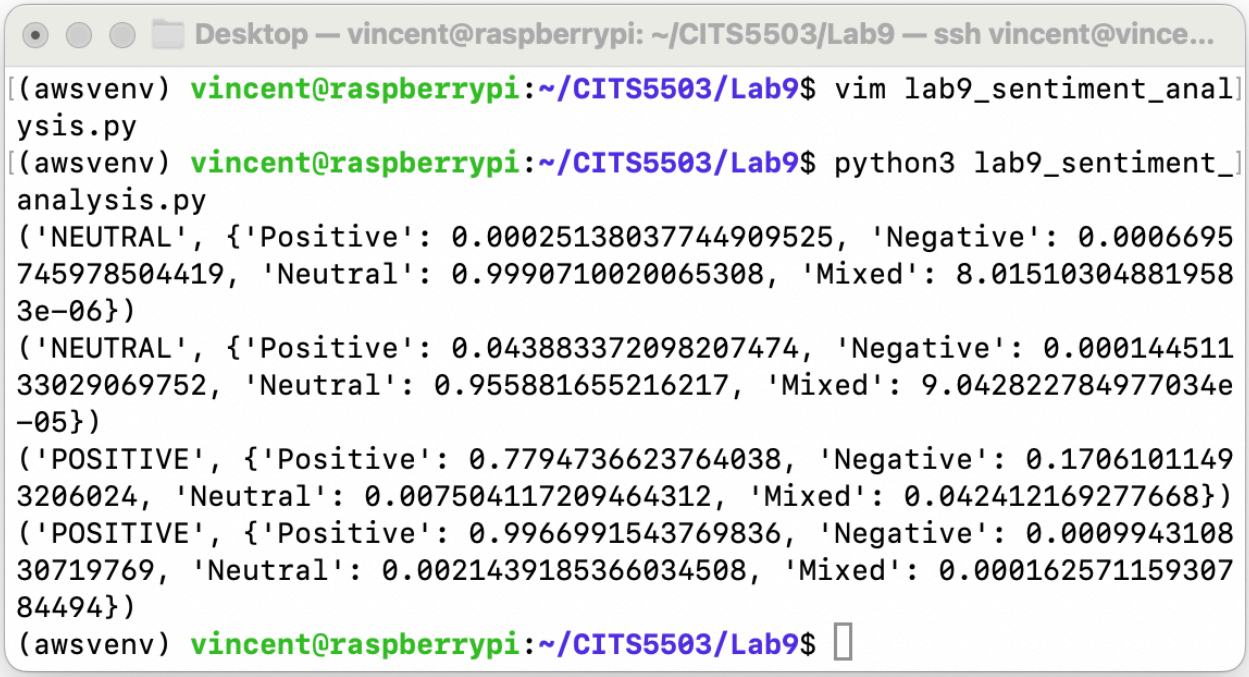
1. **Language specification**: Requires correct language code ('en', 'es', 'fr', 'it') because sentiment models are language-specific. Same words may carry different emotional weight across languages.
2. **Sentiment classification**: `detect_sentiment()` returns one of four categories:
    - **POSITIVE**: Expressions of satisfaction, joy, approval
    - **NEGATIVE**: Dissatisfaction, anger, criticism
    - **NEUTRAL**: Factual statements without emotional tone
    - **MIXED**: Text containing both positive and negative elements
3. **Confidence scores**: `SentimentScore` provides probabilities for all four categories (summing to 1.0), enabling nuanced analysis. For instance, the French love song scored 77.95% positive but also 17.06% negative, reflecting emotional complexity.

Rationale: Sentiment analysis enables automated customer feedback processing, social media monitoring, and content moderation at scale.

Command:

```
python3 lab9_sentiment_analysis.py
```

Result:




```
Text (English):
  The French Revolution was a period of social and political upheaval in France and its
colonies begin...
Sentiment: NEUTRAL
Scores:
  Positive: 0.00025138037744909525
  Negative: 0.0006695745978504419
  Neutral: 0.9990710020065308
  Mixed: 8.015103048819583e

Text (Spanish):
  El Quijote es la obra más conocida de Miguel de Cervantes Saavedra. Publicada su primera
parte con e...
Sentiment: NEUTRAL
Scores:
  Positive: 0.043883372098207474
  Negative: 0.00014451133029069752
  Neutral: 0.955881655216217
  Mixed: 9.042822784977034e

Text (French):
  Moi je n'étais rien Et voilà qu'aujourd'hui Je suis le gardien Du sommeil de ses nuits
Je l'aime à m...
```

```
Sentiment: POSITIVE
Scores:
  Positive: 0.7794736623764038
  Negative: 0.17061011493206024
  Neutral: 0.007504117209464312
  Mixed: 0.042412169277668

Text (Italian):
  L'amor che move il sole e l'altre stelle.
Sentiment: POSITIVE
Scores:
  Positive: 0.9966991543769836
  Negative: 0.000994310830719769
  Neutral: 0.0021439185366034508
  Mixed: 0.00016257115930784494
```

Analysis of Results:

- **English text (French Revolution)**: Detected as NEUTRAL with 99.91% confidence - appropriate for historical factual content
- **Spanish text (Don Quixote)**: Classified as NEUTRAL (95.59%) - correct for literary description
- **French text (Love song)**: Identified as POSITIVE (77.95%) - accurately captures the romantic theme with some negative emotion (17.06%) due to phrases like "détruire" (destroy)
- **Italian text (Dante quote)**: Strong POSITIVE sentiment (99.67%) - correctly identifies the romantic/poetic nature of "L'amor che move il sole e l'altre stelle" (The love that moves the sun and other stars)

[3] Detect Entities

Entity detection identifies and categorizes key information in text such as people, places, dates, and organizations. This is crucial for information extraction and text understanding.

Implementation:

```python
import boto3

def detect_entities(text, language_code='en'):
    """Detect entities in text using AWS Comprehend"""
    client = boto3.client('comprehend', region_name='ap-northeast-2')

    # Detect entities
    response = client.detect_entities(
        Text=text,
        LanguageCode=language_code
    )

    return response['Entities']
```

Command:

```
python lab9_entity_detection.py
```

Result:


```
Text (English):
  The French Revolution was a period of social and political upheaval in France and its
colonies begin...
Entities found: 4
  French Revolution (Type: EVENT, Score: 0.9860)
  France (Type: LOCATION, Score: 0.9898)
```

```
  1789 (Type: DATE, Score: 0.9980)
  1799 (Type: DATE, Score: 0.9987)

Text (Spanish):
  El Quijote es la obra más conocida de Miguel de Cervantes Saavedra. Publicada su primera
parte con e...
Entities found: 12
  El Quijote (Type: TITLE, Score: 0.9696)
  Miguel de Cervantes Saavedra (Type: PERSON, Score: 0.9993)
  primera parte (Type: QUANTITY, Score: 0.8664)
  El ingenioso hidalgo don Quijote de la Mancha (Type: TITLE, Score: 0.8785)
  1605 (Type: DATE, Score: 0.7977)
  una de (Type: QUANTITY, Score: 0.5971)
  española (Type: OTHER, Score: 0.9865)
  una de las más (Type: QUANTITY, Score: 0.6320)
  1615 (Type: DATE, Score: 0.9881)
  segunda parte (Type: QUANTITY, Score: 0.8887)
  Quijote de Cervantes (Type: TITLE, Score: 0.7433)
  El ingenioso caballero don Quijote de la Mancha (Type: TITLE, Score: 0.9113)

Text (French):
  Moi je n'étais rien Et voilà qu'aujourd'hui Je suis le gardien Du sommeil de ses nuits
Je l'aime à m...
Entities found: 2
  qu'aujourd'hui (Type: DATE, Score: 0.6534)
  Tout ce qu'il (Type: QUANTITY, Score: 0.6871)

Text (Italian):
  L'amor che move il sole e l'altre stelle.
Entities found: 0
  No entities detected
```

Answer to Question: What are entities?

Entities are meaningful pieces of information extracted from text representing real-world objects, concepts, or specific data points. AWS Comprehend uses named entity recognition (NER) models trained on large text corpora to identify and categorise entities.

Entity types detected:

- **PERSON**: Individual names (Miguel de Cervantes Saavedra) - confidence: 99.93%
- **LOCATION**: Geographic places (France) - confidence: 98.98%
- **ORGANIZATION**: Companies, institutions, government bodies
- **DATE**: Temporal references (1789, 1799, aujourd'hui) - both explicit and implicit
- **EVENT**: Named occurrences (French Revolution) - confidence: 98.60%
- **TITLE**: Creative work names (El Quijote, El ingenioso hidalgo don Quijote de la Mancha)
- **QUANTITY**: Numerical values and measures (primera parte, una de)
- **OTHER**: Entities not fitting standard categories (española - nationality adjective)

How entity detection works: The NER model analyses word context, capitalisation patterns, surrounding words, and grammatical structure to classify tokens. For example, "French" before "Revolution" signals an EVENT, not an adjective.

Practical applications:

1. **Information extraction**: Automatically identifying key facts from documents (e.g., extracting dates, people, and locations from news articles)
2. **Text summarisation**: Understanding main subjects and topics by entity frequency
3. **Search optimisation**: Enabling semantic search ("find articles about Miguel de Cervantes" matches "Cervantes Saavedra")
4. **Content categorisation**: Organising documents by detected entities
5. **Knowledge graph construction**: Building relationships (e.g., "Miguel de Cervantes" → author of → "El Quijote")
6. **Automated metadata tagging**: Enriching content with searchable tags

[4] Detect Key Phrases

Key phrase detection identifies the main talking points or important phrases in text. This helps understand the core topics and themes without reading the entire content.

Implementation:

```
import boto3

def detect_keyphrases(text, language_code='en'):
    """Detect key phrases in text using AWS Comprehend"""
    client = boto3.client('comprehend', region_name='ap-northeast-2')

    # Detect key phrases
    response = client.detect_key_phrases(
        Text=text,
        LanguageCode=language_code
    )

    return response['KeyPhrases']
```

Command:

```
python3 lab9_keyphrase_detection.py
```

Result:


```
Text (English):
  The French Revolution was a period of social and political upheaval in France and its
colonies begin...
Key phrases found: 7
  The French Revolution (Score: 1.0000)
  a period (Score: 1.0000)
  social and political upheaval (Score: 1.0000)
  France (Score: 0.9999)
  its colonies (Score: 1.0000)
  1789 (Score: 1.0000)
  1799 (Score: 0.9999)

Text (Spanish):
  El Quijote es la obra más conocida de Miguel de Cervantes Saavedra. Publicada su primera
parte con e...
Key phrases found: 22
  El Quijote (Score: 0.9995)
  la obra (Score: 1.0000)
  más conocida (Score: 0.9988)
  Miguel de Cervantes Saavedra (Score: 1.0000)
  su primera parte (Score: 0.9999)
  el título (Score: 1.0000)
  El ingenioso hidalgo don Quijote de la Mancha (Score: 0.9561)
  comienzos (Score: 0.9998)
  1605 (Score: 0.9962)
  una (Score: 0.9869)
  las obras (Score: 0.99998)
  más destacadas (Score: 0.9999)
  la literatura española (Score: 1.0000)
```

```
  la literatura universal (Score: 0.9999)
  una (Score: 0.9485)
  las más traducidas (Score: 0.9995)
  la segunda parte (Score: 0.99998)
  Quijote de Cervantes (Score: 0.9999)
  el título (Score: 0.9999)
  El (Score: 0.9998)
  ingenioso caballero don Quijote de la Mancha (Score: 0.9365)

Text (French):
  Moi je n'étais rien Et voilà qu'aujourd'hui Je suis le gardien Du sommeil de ses nuits
Je l'aime à m...
Key phrases found: 20
  Moi (Score: 0.9999)
  je (Score: 0.9851)
  n'étais rien (Score: 0.9738)
  aujourd'hui (Score: 0.9490)
  Je (Score: 0.9868)
  le gardien Du sommeil de ses nuits (Score: 0.9247)
  Je (Score: 0.9999)
  l'aime (Score: 0.9993)
  Vous (Score: 0.9983)
  Tout ce (Score: 0.9840)
  qu'il (Score: 0.9711)
  vous (Score: 0.9997)
  Elle (Score: 0.9997)
  L'espace de ses bras (Score: 0.9582)
  tout (Score: 0.9596)
  tout (Score: 0.9780)
  Je (Score: 0.9999)
  l'aime (Score: 0.9995)

Text (Italian):
  L'amor che move il sole e l'altre stelle.
Key phrases found: 4
  L'amor (Score: 0.99998)
  che (Score: 0.9996)
  il sole (Score: 1.0000)
  l'altre stelle (Score: 0.9998)
```

Answer to Question: What are key phrases?

Key phrases are noun phrases or short expressions that represent the main topics, subjects, or important concepts in a text. Unlike entities which are specific named items, key phrases capture:

1. **Core topics**: Main subjects being discussed (e.g., "The French Revolution", "social and political upheaval")
2. **Descriptive phrases**: Important qualifiers and descriptions (e.g., "más conocida", "la literatura española")
3. **Temporal references**: Time-related phrases (e.g., "a period", "su primera parte")
4. **Location phrases**: Place-related expressions (e.g., "France", "its colonies")

Key phrases differ from entities in that they:

- Can include common nouns and adjectives, not just named entities
- Capture multi-word expressions that form complete concepts
- Focus on what the text is "about" rather than just named items
- Help in text summarization and topic modeling

Use cases for key phrases:

- **Document summarization**: Quickly understanding main topics
- **Content indexing**: Improving search relevance
- **Topic clustering**: Grouping similar documents
- **Trending analysis**: Identifying popular discussion topics
- **SEO optimization**: Extracting important keywords for web content

[5] Detect Syntax

Syntax detection analyzes the grammatical structure of text by identifying the part of speech (POS) for each word or token. This linguistic analysis is fundamental for understanding sentence structure and meaning.

Implementation:

```
import boto3

def detect_syntax(text, language_code='en'):
    """Detect syntax in text using AWS Comprehend"""
    client = boto3.client('comprehend', region_name='ap-northeast-2')

    # Detect syntax
    response = client.detect_syntax(
        Text=text,
        LanguageCode=language_code
    )

    return response['SyntaxTokens']
```

Command:

```
python3 lab9_syntax_detection.py
```

Result:


```
Text (English):
  The French Revolution was a period of social and political upheaval in France and its
colonies begin...
Syntax tokens found: 24
  - The (POS: DET, Score: 1.0000)
  - French (POS: PROPN, Score: 1.0000)
  - Revolution (POS: PROPN, Score: 1.0000)
  - was (POS: VERB, Score: 1.0000)
  - a (POS: DET, Score: 1.0000)
  - period (POS: NOUN, Score: 1.0000)
  - of (POS: ADP, Score: 1.0000)
  - social (POS: ADJ, Score: 1.0000)
  - and (POS: CCONJ, Score: 1.0000)
```

```
  - political (POS: ADJ, Score: 1.0000)
  ... and 14 more tokens

Text (Spanish):
  El Quijote es la obra más conocida de Miguel de Cervantes Saavedra. Publicada su primera
parte con e...
Syntax tokens found: 80
  - El (POS: DET, Score: 1.0000)
  - Quijote (POS: PROPN, Score: 1.0000)
  - es (POS: VERB, Score: 1.0000)
  - la (POS: DET, Score: 1.0000)
  - obra (POS: NOUN, Score: 1.0000)
  - más (POS: ADV, Score: 1.0000)
  - conocida (POS: ADJ, Score: 0.6904)
  - de (POS: ADP, Score: 1.0000)
  - Miguel (POS: PROPN, Score: 1.0000)
  - de (POS: ADP, Score: 1.0000)
  ... and 70 more tokens

Text (French):
  Moi je n'étais rien Et voilà qu'aujourd'hui Je suis le gardien Du sommeil de ses nuits
Je l'aime à m...
Syntax tokens found: 54
  - Moi (POS: PRON, Score: 1.0000)
  - je (POS: PRON, Score: 1.0000)
  - n' (POS: ADV, Score: 1.0000)
  - étais (POS: AUX, Score: 1.0000)
  - rien (POS: PRON, Score: 1.0000)
  - Et (POS: CCONJ, Score: 1.0000)
  - voilà (POS: VERB, Score: 1.0000)
  - qu' (POS: SCONJ, Score: 0.9999)
  - aujourd'hui (POS: NOUN, Score: 0.9999)
  - Je (POS: PRON, Score: 1.0000)
  ... and 44 more tokens

Text (Italian):
  L'amor che move il sole e l'altre stelle.
Syntax tokens found: 11
  - L' (POS: DET, Score: 1.0000)
  - amor (POS: NOUN, Score: 1.0000)
  - che (POS: PRON, Score: 1.0000)
  - move (POS: VERB, Score: 1.0000)
  - il (POS: DET, Score: 1.0000)
  - sole (POS: NOUN, Score: 1.0000)
  - e (POS: CCONJ, Score: 1.0000)
  - l' (POS: DET, Score: 1.0000)
  - altre (POS: ADJ, Score: 1.0000)
  - stelle (POS: NOUN, Score: 1.0000)
  ... and 1 more tokens
```

Answer to Question: What are syntaxes?

Syntax (or syntactic analysis) refers to the grammatical structure of sentences, specifically the identification of each word's part of speech (POS) and its role in the sentence. AWS Comprehend's syntax detection tokenizes text and assigns grammatical tags to each token.

Common Part-of-Speech Tags:

- **DET** (Determiner): Articles and demonstratives (the, a, this, that)
- **NOUN**: Common nouns (period, obra, gardien, sole)
- **PROPN** (Proper Noun): Names and proper nouns (French Revolution, Quijote, Miguel)
- **VERB**: Action words (was, es, move)
- **ADJ** (Adjective): Descriptive words (social, political, conocida)
- **ADV** (Adverb): Modifiers (más, n')
- **PRON** (Pronoun): Pronouns (Moi, je, che)
- **ADP** (Adposition): Prepositions (of, de, in)
- **CCONJ** (Coordinating Conjunction): Connecting words (and, e, Et)
- **SCONJ** (Subordinating Conjunction): Subordinating words (qu')
- **AUX** (Auxiliary): Helper verbs (étais)

Importance of Syntax Detection:

1. **Grammar checking**: Identifying grammatical errors in text
2. **Machine translation**: Understanding sentence structure for accurate translation
3. **Information extraction**: Identifying subjects, verbs, objects for relationship extraction
4. **Text-to-speech**: Improving pronunciation and intonation
5. **Semantic analysis**: Understanding meaning through grammatical roles
6. **Language learning**: Analyzing sentence structure for educational purposes

Syntax detection provides the foundation for deeper natural language understanding by revealing how words relate to each other grammatically within sentences.

AWS Rekognition - Computer Vision

[1] Create S3 Bucket and Upload Images

To test AWS Rekognition's image analysis capabilities, I first created an S3 bucket to store the test images. The bucket was named following the lab convention with my student ID.

Implementation:

```
import boto3
from botocore.exceptions import ClientError

# Configuration
BUCKET_NAME = '23917077-lab9'
REGION = 'ap-northeast-2'

def create_s3_bucket():
    """Create S3 bucket for Lab 9"""
    s3_client = boto3.client('s3', region_name=REGION)
```

```
    try:
        # Create bucket with location constraint
        s3_client.create_bucket(
            Bucket=BUCKET_NAME,
            CreateBucketConfiguration={'LocationConstraint': REGION}
        )
        print(f"✓ S3 bucket '{BUCKET_NAME}' created successfully")
        return True
    except ClientError as e:
        if e.response['Error']['Code'] == 'BucketAlreadyOwnedByYou':
            print(f"✓ S3 bucket '{BUCKET_NAME}' already exists")
            return True
        else:
            print(f"✗ Error creating bucket: {e}")
            return False
```

Command:

```
python3 lab9_setup_s3.py
```

Result:

```
✓ S3 bucket '23917077-lab9' created successfully
```

Upload Images to S3:

After creating the bucket, I uploaded the four required images:

```
import boto3
import os

def upload_images_to_s3():
    """Upload images to S3 bucket"""
    s3_client = boto3.client('s3', region_name=REGION)

    # List of images to upload
    images = ['urban.jpg', 'beach.jpg', 'faces.jpg', 'text.jpg']

    for image in images:
        if os.path.exists(image):
            s3_client.upload_file(image, BUCKET_NAME, image)
            print(f"✓ Uploaded {image} to s3://{BUCKET_NAME}/{image}")
```

Command:

```
python lab9_upload_images.py
```

Result:

```
✓ Uploaded urban.jpg to s3://23917077-lab9/urban.jpg
✓ Uploaded beach.jpg to s3://23917077-lab9/beach.jpg
✓ Uploaded faces.jpg to s3://23917077-lab9/faces.jpg
✓ Uploaded text.jpg to s3://23917077-lab9/text.jpg

Upload complete!
```

The S3 bucket now contains all four images ready for AWS Rekognition analysis.

[2] Test AWS Rekognition Features

I created a comprehensive Python script to test all four main AWS Rekognition features: label recognition, image moderation, facial analysis, and text extraction.

Implementation:

```python
import boto3

BUCKET_NAME = '23917077-lab9'
REGION = 'ap-northeast-2'

def detect_labels(image_name):
    """Detect labels in image using AWS Rekognition"""
    rekognition = boto3.client('rekognition', region_name=REGION)

    response = rekognition.detect_labels(
        Image={'S3Object': {'Bucket': BUCKET_NAME, 'Name': image_name}},
        MaxLabels=10
    )

    return response['Labels']

def detect_moderation(image_name):
    """Detect moderation labels in image"""
    rekognition = boto3.client('rekognition', region_name=REGION)

    response = rekognition.detect_moderation_labels(
        Image={'S3Object': {'Bucket': BUCKET_NAME, 'Name': image_name}}
    )

    return response['ModerationLabels']

def detect_faces(image_name):
    """Detect faces in image"""
    rekognition = boto3.client('rekognition', region_name=REGION)

    response = rekognition.detect_faces(
        Image={'S3Object': {'Bucket': BUCKET_NAME, 'Name': image_name}},
        Attributes=['ALL']
    )
```

```
    return response['FaceDetails']

def detect_text(image_name):
    """Detect text in image"""
    rekognition = boto3.client('rekognition', region_name=REGION)

    response = rekognition.detect_text(
        Image={'S3Object': {'Bucket': BUCKET_NAME, 'Name': image_name}}
    )

    return response['TextDetections']
```

Command:

```
python lab9_rekognition.py
```

Result:

```
[1] Label Detection - urban.jpg
--------------------------------------------------------------------------------
  - Architecture: 99.57%
  - Building: 99.57%
  - Cityscape: 97.31%
  - Urban: 97.31%
  - Tower: 94.60%
  - Landmark: 92.00%
  - Oriental Pearl TV Tower: 87.86%

[2] Label Detection - beach.jpg
--------------------------------------------------------------------------------
  - Shirt: 100.00%
  - Long Sleeve: 99.65%
  - Adult: 99.03%
  - Male: 99.03%
  - Man: 99.03%
  - Person: 99.03%
  - Beachwear: 97.75%
  - Glasses: 96.27%
  - Dress Shirt: 84.32%
  - Nature: 83.52%

[3] Moderation Detection - beach.jpg
--------------------------------------------------------------------------------
  No inappropriate content detected

[4] Face Detection - faces.jpg
--------------------------------------------------------------------------------
  Number of faces detected: 1

  Face 1:
    - Age Range: 18-22
```

```
    - Gender: Male (99.98%)
    - Emotions: CALM (100.0%), CONFUSED (0.0%), SAD (0.0%)

[5] Text Detection - text.jpg
--------------------------------------------------------------------------------
  Text lines detected: 2
  - Is That (Confidence: 94.95%)
  - True? (Confidence: 100.00%)
```

Analysis of Results:

1. Label Recognition (urban.jpg):

The image of the Oriental Pearl TV Tower in Shanghai was accurately identified with multiple relevant labels including "Architecture" (99.57%), "Building" (99.57%), and even the specific landmark "Oriental Pearl TV Tower" (87.86%). This demonstrates Rekognition's ability to recognize not just general categories but specific landmarks.

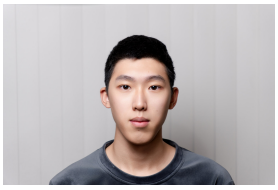
2. Label Recognition (beach.jpg):

The beach photo was analyzed with high accuracy, detecting clothing items ("Shirt" 100%, "Long Sleeve" 99.65%), person attributes ("Adult" 99.03%, "Male" 99.03%), and context ("Beachwear" 97.75%, "Nature" 83.52%). The detailed labels provide comprehensive scene understanding.

3. Image Moderation (beach.jpg):

The content moderation check returned no inappropriate content, confirming the image is safe for general audiences. This feature is crucial for platforms that need to filter user-uploaded content automatically.

4. Facial Analysis (faces.jpg):



Rekognition detected 1 face and provided detailed attributes:

- **Age estimation**: 18-22 years old
- **Gender**: Male with 99.98% confidence
- **Emotional state**: CALM (100%), showing the person's emotional expression
  This level of detail is valuable for demographics analysis, emotion recognition, and personalized user experiences.

5. Text Extraction (text.jpg):

The OCR (Optical Character Recognition) successfully extracted text from the image:

- Line 1: "Is That" (94.95% confidence)
- Line 2: "True?" (100% confidence)

The text detection can identify both printed and handwritten text, making it useful for document digitization, sign reading, and automated data entry.

Cleanup Resources

After completing Lab 9, I cleaned up all AWS resources created during the lab exercises.

Resources deleted:

- S3 bucket with test images
- Local downloaded images
- Any CloudWatch logs from Rekognition/Comprehend API calls

Commands:

```
# Delete all objects in S3 bucket first
aws s3 rm s3://23917077-lab9/ --recursive

# Delete S3 bucket
aws s3 rb s3://23917077-lab9
```