Multilingual Document Translation Platform

Capstone project documentation
AWS Cloud Intensive Training

Prepared By:
Michael Aboagye
September 2025

GitHub Repository:

https://github.com/michaelaboagye76/multilingual-document-translation

Project Overview

For my capstone project in the AWS Cloud Intensive Training, I developed a cloud-based multilingual document translation solution. The purpose of the application is to allow users to upload JSON documents, translate the text into multiple languages using AWS Translate, and download the translated files from AWS S3. The project demonstrates the integration of AWS services with Python, along with front-end and back-end development, and infrastructure automation using Terraform.

Tools and Technologies Used

- AWS Services: S3, Lambda (optional), EC2, IAM
- Infrastructure as Code: Terraform
- Python Libraries: Flask, Boto3
- Frontend: HTML, Bootstrap
- Reverse Proxy: Nginx

Architecture

- **Frontend (Flask on EC2)**:
    - Users can upload JSON documents to the input S3 bucket (doc-uploads).
    - The application displays available translations from the output S3 bucket (doc-translated) using presigned URLs for secure download.
- **Backend (Python & Boto3)**:
    - Retrieves uploaded documents from S3.
    - Optionally triggers Lambda functions for translation.
    - Saves translated JSON back to the output S3 bucket.
- **Infrastructure**:
    - EC2 instance hosts the Flask application.

- Nginx reverse proxy routes HTTP traffic to Flask running on a non-privileged port (5000).
- IAM role attached to the EC2 instance grants S3 access without requiring hardcoded credentials.

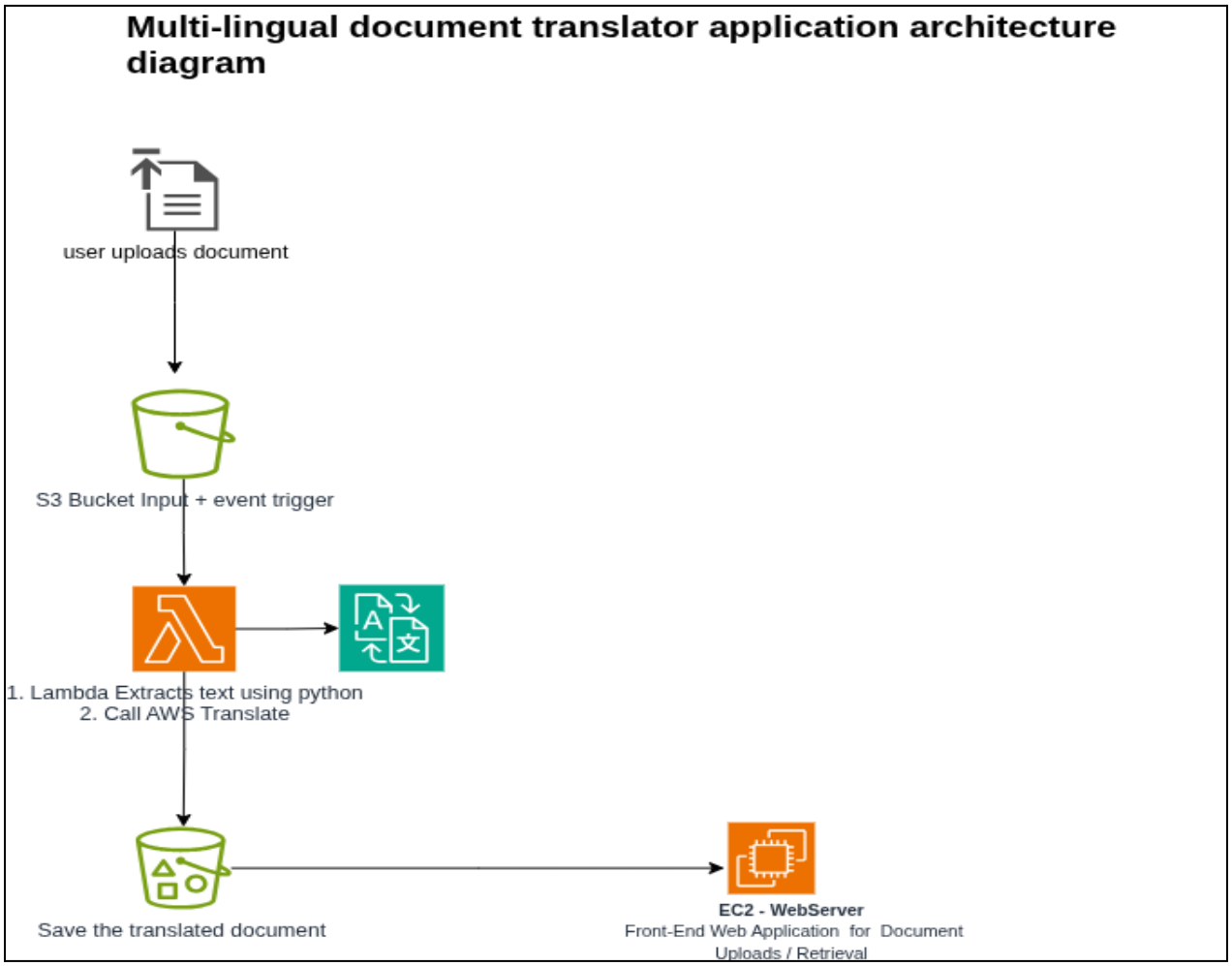

Fig: Architecture diagram showing the workflow of the application.

Implementation Steps

1. Developing the Flask Application

I began by creating a Flask application with routes to handle file uploads and display translated documents. The /upload route handled uploading JSON files to an input S3 bucket, while the / route listed translated files from the output S3 bucket using presigned URLs for secure access.

During testing, I encountered Internal Server Errors (HTTP 500). After inspecting the Flask logs, I discovered that the error was caused by improper Jinja2 template syntax in index.html, specifically a {% for %} loop that was not closed. I corrected the template by adding the missing {% endfor %} and included an {% if files %}…{% else %} block to handle cases when no files were present.

2. Resolving AWS Credentials Issues

When attempting to access S3 from the Flask app, I ran into the error: botocore.exceptions.NoCredentialsError: Unable to locate credentials. I realized that the EC2 instance did not have any AWS credentials configured. To resolve this, I attached an **IAM role** to the EC2 instance, granting read and write access to the relevant S3 buckets. This allowed the Flask app to access S3 securely without hardcoding credentials. I verified the configuration by running aws s3 ls from the EC2 instance and confirming that the buckets were accessible.

3. Deploying on EC2

I used Terraform to provision an EC2 instance. Initially, I attempted to run the Flask application with Gunicorn on port 80, but encountered permission errors because binding to ports below 1024 requires root privileges. To solve this, I removed Gunicorn and configured the Flask app to run on port 5000. I installed Nginx on the EC2 instance and set it up as a reverse proxy, forwarding requests from port 80 to Flask on port 5000. This allowed the application to be accessed through standard HTTP without running Flask as root.

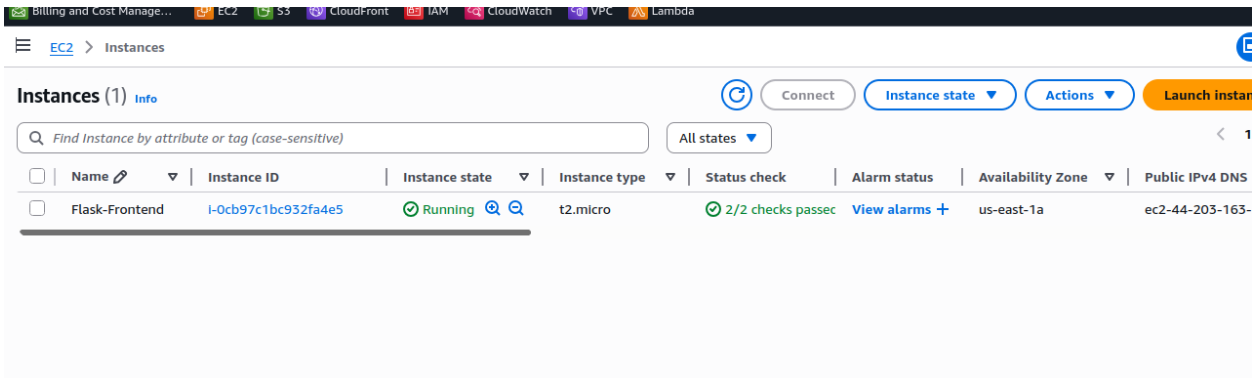

Fig 2. EC2 web server created

4. Automating EC2 Setup with User Data

I created a user_data script to automate the EC2 instance setup. The script:

- Installed Python3, pip, git, Flask, and Boto3
- Installed and started Nginx
- Cloned my GitHub repository containing the Flask app
- Started the Flask app in the background using nohup
- Configured Nginx to forward requests to the Flask application

This automation ensured that the EC2 instance could be deployed consistently and quickly.

```
#!/bin/bash
# Update packages
sudo yum update -y

# Install Python, pip, git
sudo yum install -y python3 python3-pip git

# Install Flask and Boto3
pip3 install flask boto3

# Install and start Nginx
sudo yum install -y nginx
sudo systemctl start nginx
sudo systemctl enable nginx

# Clone your repo
git clone https://github.com/michaelaboagye76/multilingual-document-translation.git
cd multilingual-document-translation/flask-app
# Start Flask app in the background on port 5000
nohup python3 app.py &
# Configure Nginx to proxy requests to Flask port 5000
sudo tee /etc/nginx/conf.d/flaskapp.conf <<'EOT'
server {
    listen 80;
    server_name _;
    location / {
        proxy_pass http://127.0.0.1:5000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
EOT
```

Fig 3. EC2 user data script.

5. Fixing Frontend Template Issues

I reviewed the index.html template and corrected syntax errors that had caused server crashes. I also improved the user interface to display a message when no translated files were available and ensured that download links for files were generated correctly using presigned S3 URLs.

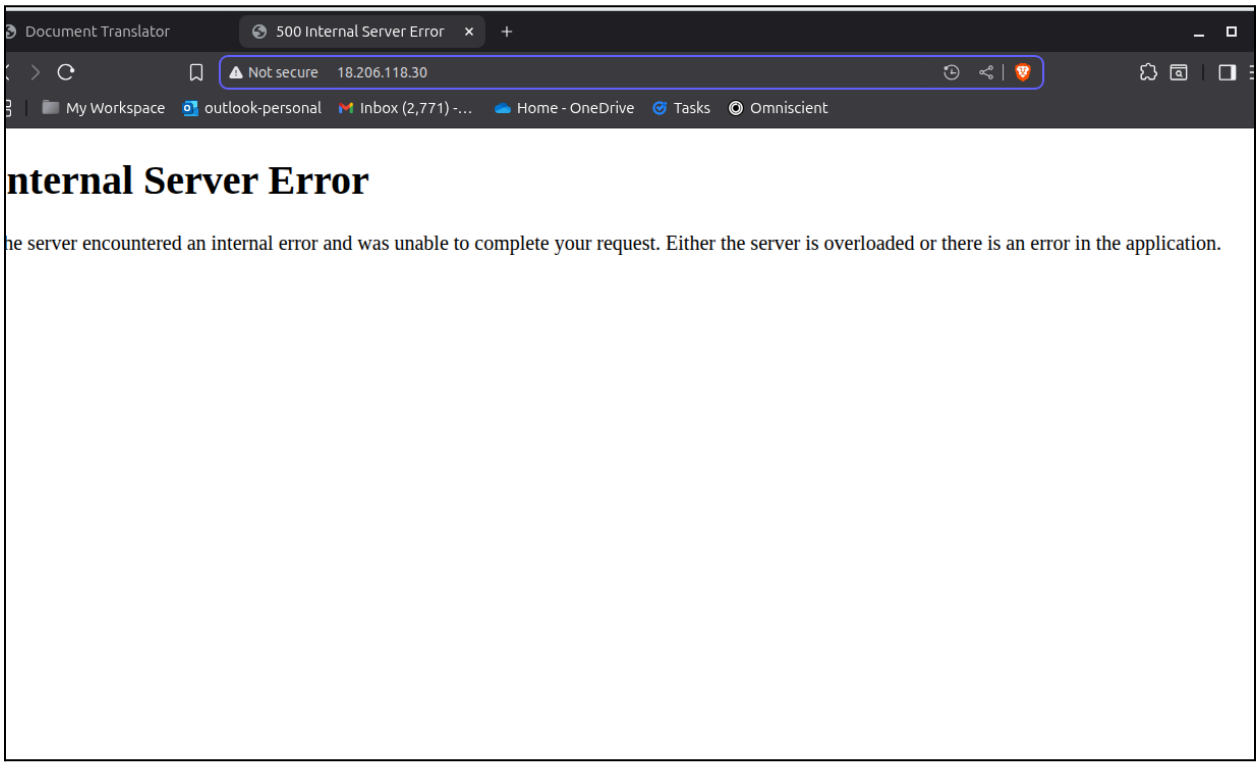

Fig 4. Error launching the flask app.

6. Security and Permissions

To ensure secure access, I attached an IAM role to the EC2 instance with permissions limited to the two S3 buckets used in the project (doc-uploads and doc-translated). The EC2 instance was also protected with a security group that allowed inbound HTTP (port 80) and SSH (port 22) access. This setup ensured that the application could interact with S3 securely without exposing sensitive credentials.

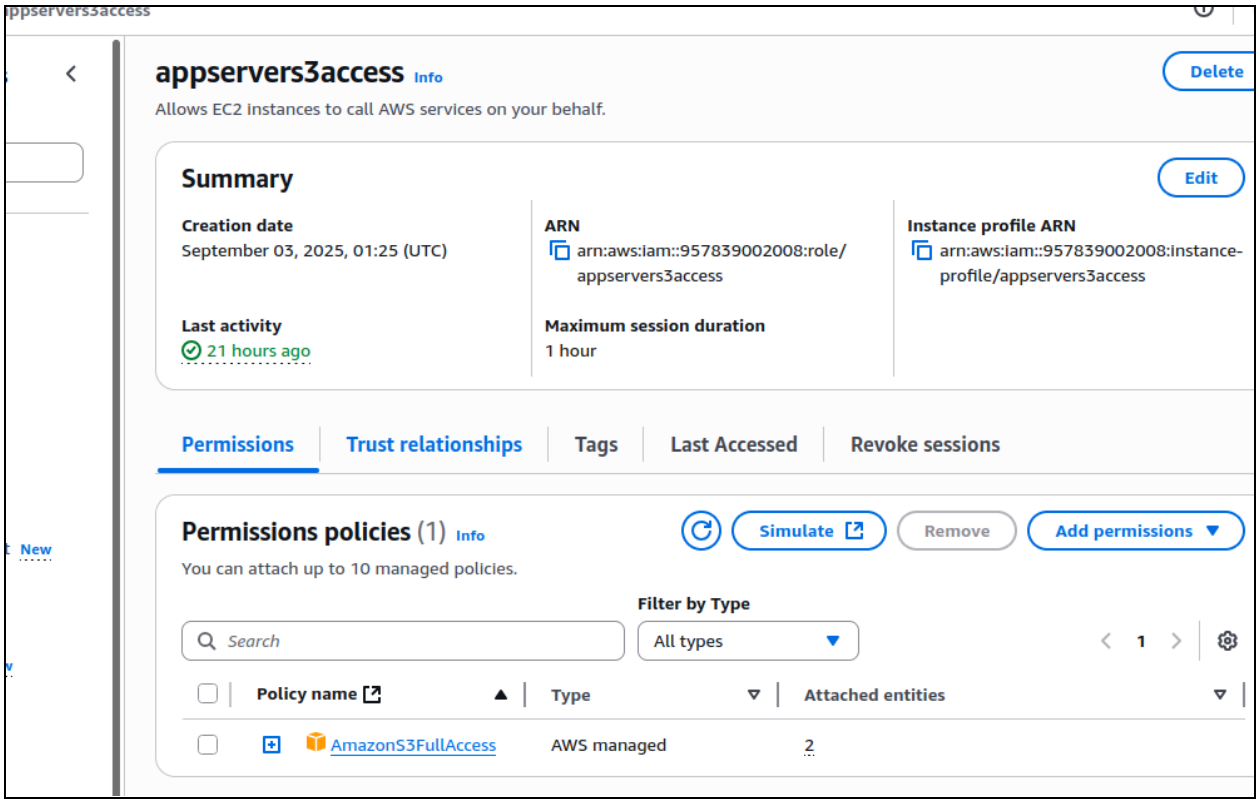

Fig 5. EC2 IAM role for S3 Access.

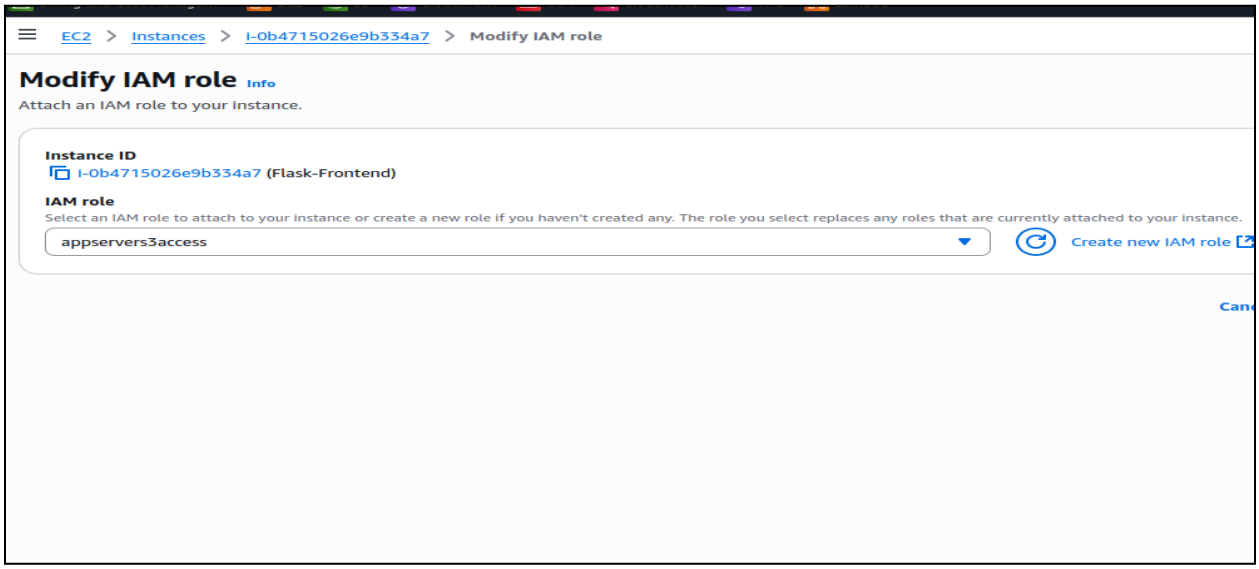

Fig 5.1. Attach IAM Role to EC2 Instance.

Challenges Encountered and Solutions

1. **AWS Credentials**: Initially missing, which caused access errors. Solved by attaching an IAM role to EC2.
2. **Template Syntax Errors**: Improperly closed Jinja2 blocks caused 500 errors. Fixed by closing loops and adding conditional rendering.
3. **Port Binding Errors**: Flask could not bind to port 80. Solved by running Flask on port 5000 and using Nginx as a reverse proxy.
4. **Automation**: Setting up dependencies and deployment manually was time-consuming. Solved by creating a user data script and using Terraform.
5. **Internal Server Errors**: Caused by missing S3 permissions and template syntax; fixed by IAM role and template correction.

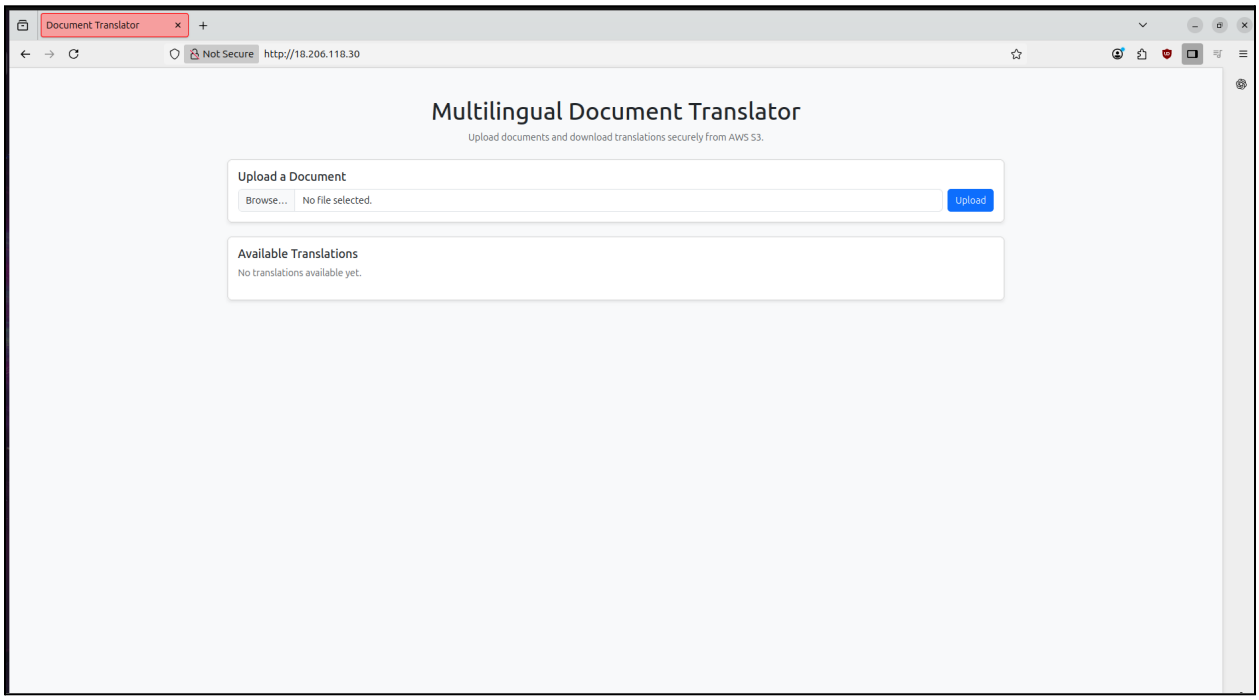

Fig 6. Flask App running.

Lessons Learned

- Using IAM roles for EC2 is a secure and efficient way to grant AWS service access.
- Jinja2 template syntax errors can easily cause server crashes; careful attention to {% for %} and {% if %} blocks is essential.
- Ports below 1024 require root; using a reverse proxy like Nginx avoids security risks.
- Automating infrastructure deployment through IaC reduces human error and improves reproducibility.

- Serverless functions can complement EC2 applications for scalable, event-driven processing.

Potential Improvements

- Integrate **AWS Lambda** to perform automatic translation whenever a new file is uploaded.
- Enable **HTTPS** using Let's Encrypt for secure web access.
- Display **JSON content previews** in the frontend so users can quickly see the translation before downloading.
- Replace the Flask development server with **Gunicorn + systemd** for production-grade performance.
- Expand IaC definitions to include Lambda functions and event triggers for a fully automated and serverless workflow.

Conclusion

This documentation captures the full workflow of my project, from development and debugging to deployment and automation, highlighting the importance of IaC and serverless components in building scalable, secure cloud applications.

Using Terraform for infrastructure provisioning was a critical part of this project. IaC allows for the **automation and reproducibility of cloud resources**, meaning that the EC2 instance, security groups, and S3 buckets can be recreated reliably across multiple environments without manual configuration.

This reduces human error, ensures consistency, and makes the deployment process faster and more scalable. By defining infrastructure in code, I was able to version-control the environment setup and apply changes systematically.

While the core application runs on EC2, integrating serverless components like **AWS Lambda** adds significant benefits. Lambda can automatically process uploaded JSON files, handle translation via AWS Translate, and write the results to S3 without requiring a continuously running server.

Serverless architectures reduce operational overhead, automatically scale with demand, and optimize costs by only charging for execution time. This project demonstrates how serverless functions could complement the EC2-hosted Flask application for real-time processing.