CAPSTONE PROJECT

Design, develop, and implement an Infrastructure-as-Code (IaC) solution on AWS, using CloudFormation or Terraform, integrating AWS Translate for automated language translation and Amazon S3 for object storage. The solution will use Python and Boto3 to process translation requests from JSON files and store both input and translated output in S3 buckets.

Tools & Technologies

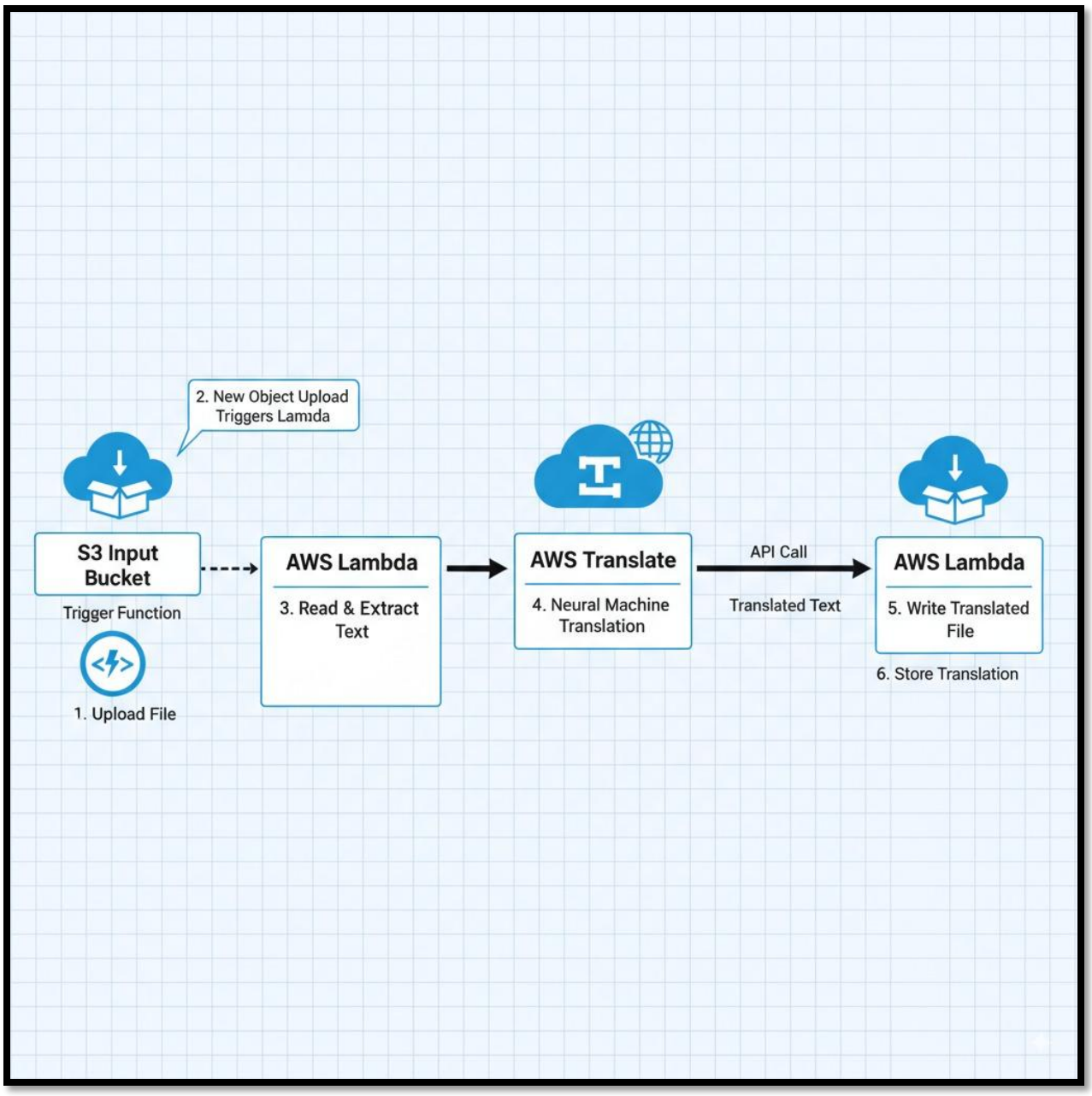
- **AWS Services:**
  - **Amazon S3 (object storage)**
  - **AWS Lambda (serverless compute)**
  - **Amazon Translate (NLP service)**
  - **AWS IAM (authentication & access policies)**
  - **AWS CloudFormation (IaC template management)**
  - **Amazon CloudWatch (monitoring & logging)**
- **Programming Language: Python 3.9**
- **Libraries: Boto3 SDK for AWS**

System Architecture

1. **User Request:** The process begins with a user who wants a document or text translated.
2. **Request S3 Bucket:** The user uploads the file containing the text to be translated into a designated "Request S3 Bucket." This bucket acts as the entry point for translation requests.
3. **S3 Event (Trigger):** The act of uploading a new file to the Request S3 Bucket generates an S3 Event. This event acts as a trigger for an AWS Lambda function.
4. **Lambda (Process Request):** An AWS Lambda function is invoked by the S3 event. This Lambda function reads the content of the newly uploaded file

from the Request S3 Bucket. It's responsible for extracting the text and preparing it for translation.

5. **Amazon Translate:** The Lambda function sends the extracted text to Amazon Translate, AWS's machine translation service. Amazon Translate processes the text and returns the translated version.
6. **Response S3 Bucket:** After receiving the translated text from Amazon Translate, the Lambda function writes this translated content into a new file and stores it in a "Response S3 Bucket." This bucket serves as the output location where the user can retrieve their translated documents.


2. New Object Upload Triggers Lamıda
S3 Input Bucket
Trigger Function
1. Upload File
AWS Lambda
3. Read & Extract Text
AWS Translate
4. Neural Machine Translation
API Call
Translated Text
AWS Lambda
5. Write Translated File
6. Store Translation

AWS CAPSTONE PROJECT – Step-by-Step Console Guide

Step 0 – AWS Account Readiness

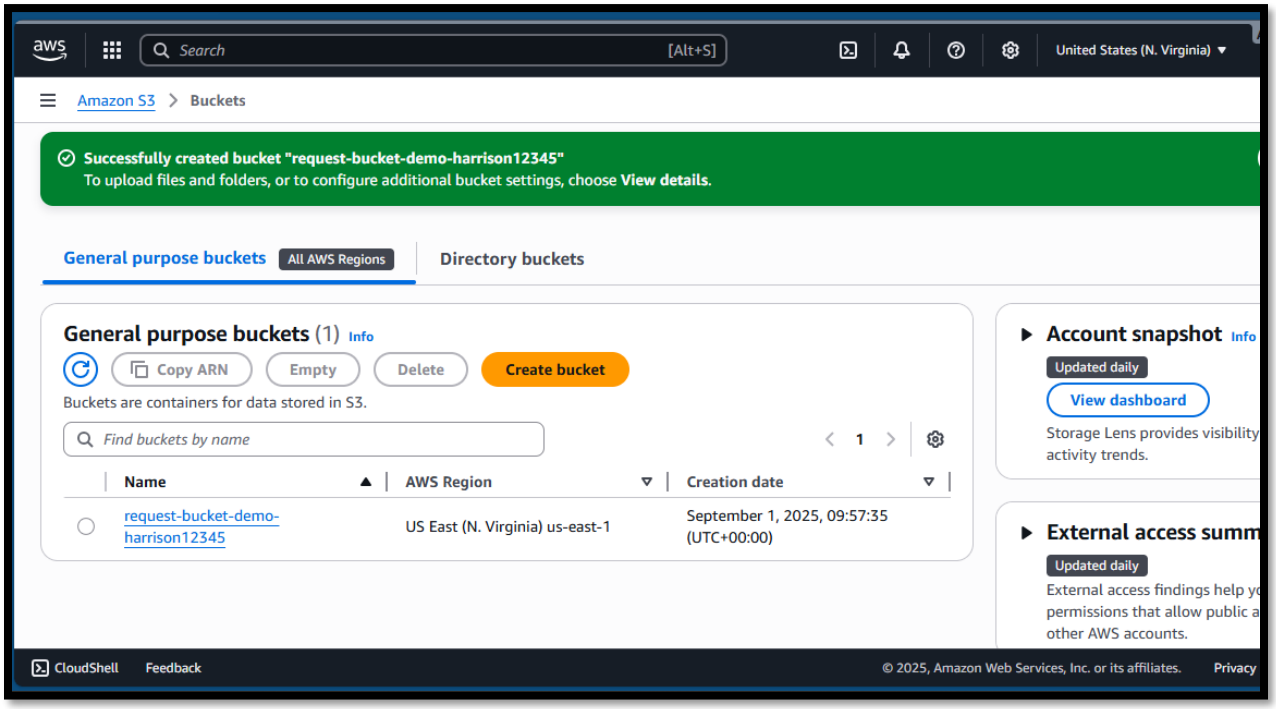
I logged into the AWS Management Console using my IAM admin user and confirmed that the selected region in the top-right corner was set to **N. Virginia (us-east-1)**, as required.

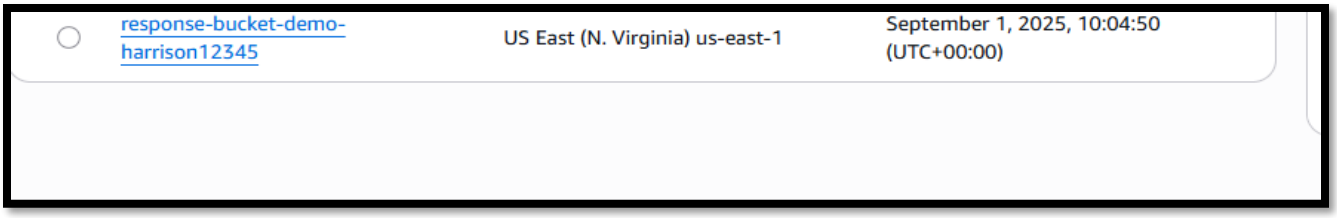
Step 1 – Creation of Two S3 Buckets

I created the two S3 buckets needed for this setup:

1. **request-bucket-demo-harrison1234**
    - Region: N. Virginia (us-east-1)
    - All other settings left at default
2. **response-bucket-demo-harrison1234**
    - Region: N. Virginia (us-east-1)
    - All other settings left at default

Both buckets were created successfully and are now visible and operational in the S3 console.





Step 2 – Create an IAM Role for Lambda

I set up an IAM role that allows Lambda to interact with the S3 buckets and call the Translate service. The steps I followed were:

1. In the AWS Management Console, I searched for **IAM**.
2. From the left-hand menu, I selected **Roles** and clicked **Create role**.
3. For the **Trusted entity**, I chose **AWS Service**, then selected **Lambda**, and clicked **Next**.
4. Under **Permissions**, I attached the following policies:
    - **AmazonS3FullAccess** – to allow read/write access to the buckets
    - **TranslateFullAccess** – to use AWS Translate
    - **AWSLambdaBasicExecutionRole** – to enable CloudWatch logging
5. I clicked **Next**, named the role **LambdaTranslateRole**, and completed the process by selecting **Create role**.

The role was created successfully and is now ready for use with Lambda.

Role LambdaTranslateRole Created

Now I have created a role Lambda can use;

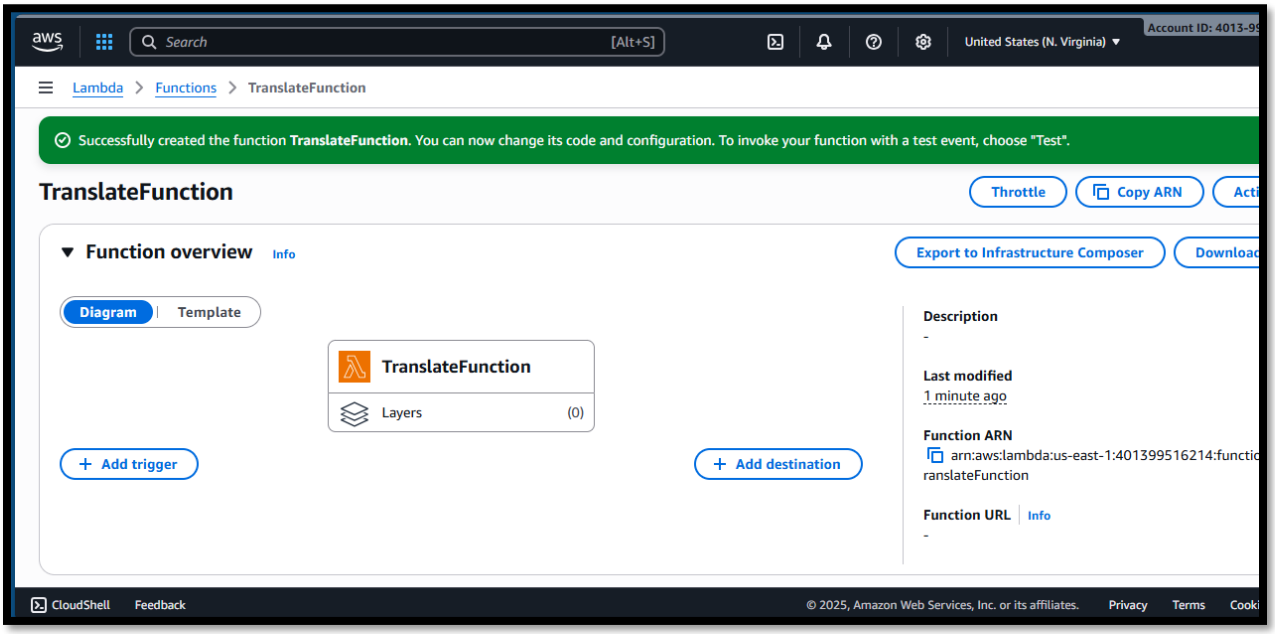
Step 3 – Create the Lambda Function

I created the Lambda function that will serve as the core logic of the solution. The steps were:

1. In the AWS Management Console, I searched for **Lambda** and opened the service.
2. I clicked **Create function** and selected **Author from scratch**.
    - **Function name:** `TranslateFunction`
    - **Runtime:** Python 3.9
    - **Execution role:** I chose **Use existing role** and selected the previously created **LambdaTranslateRole**.

3. I completed the setup by clicking **Create function**.

The function was successfully created and is now available in the Lambda console.



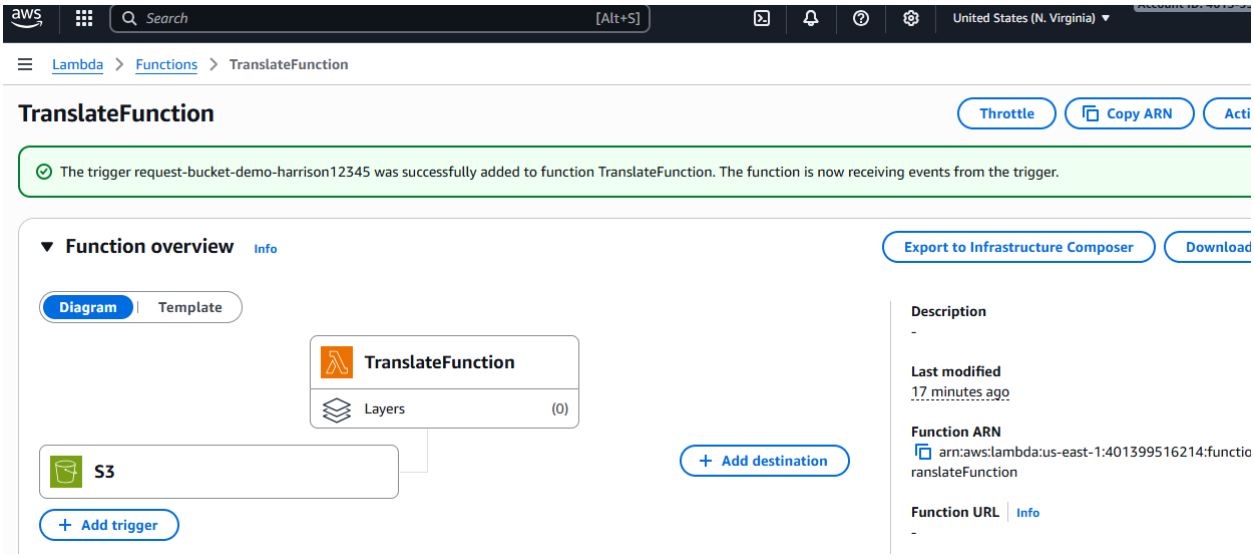
The default code was removed and replaced with my custom source code, shown below:

Step 4 – Add S3 Trigger

I configured an S3 trigger so the Lambda function runs automatically whenever a file is uploaded to the input bucket. The steps were:

1. On the **Lambda function page**, I scrolled to **Triggers** and clicked **Add trigger**.
2. I selected **S3** and set the following options:
    - **Bucket:** `request-bucket-demo-12345`
    - **Event type:** PUT (all object create events)
    - Checked the box to **Add permission to Lambda role**
3. I clicked **Add** to finish the setup.

The trigger was successfully attached, and the Lambda function now receives events from the input bucket.



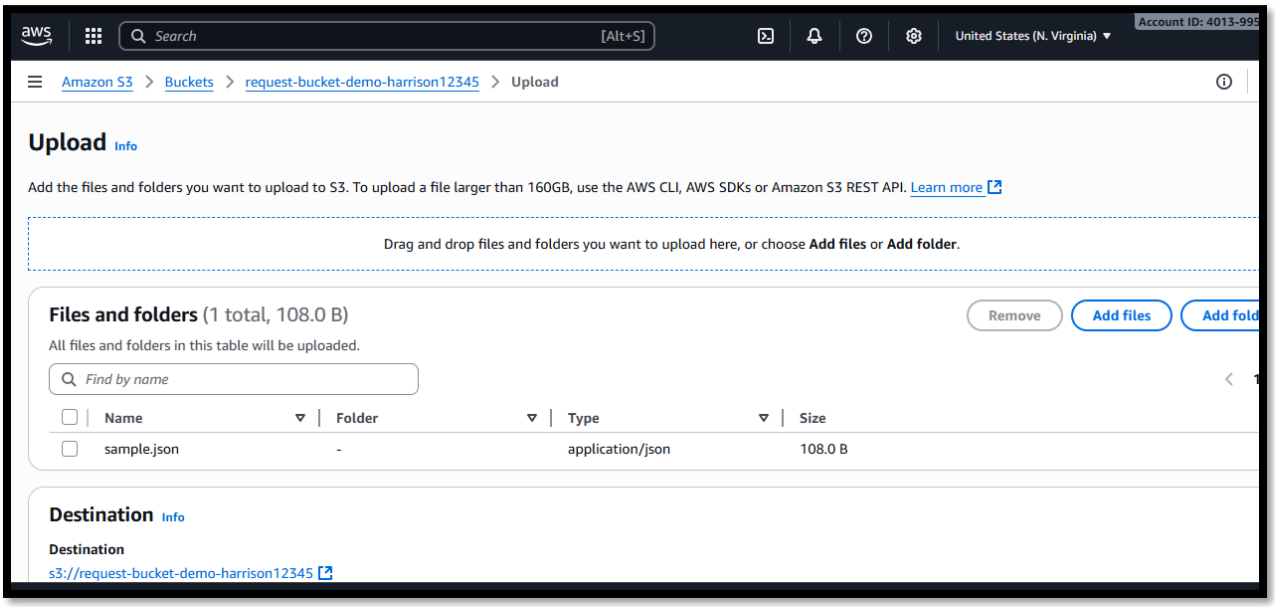
Now Lambda will be invoked when new files land in my input bucket.

Step 5 – Test End-to-End

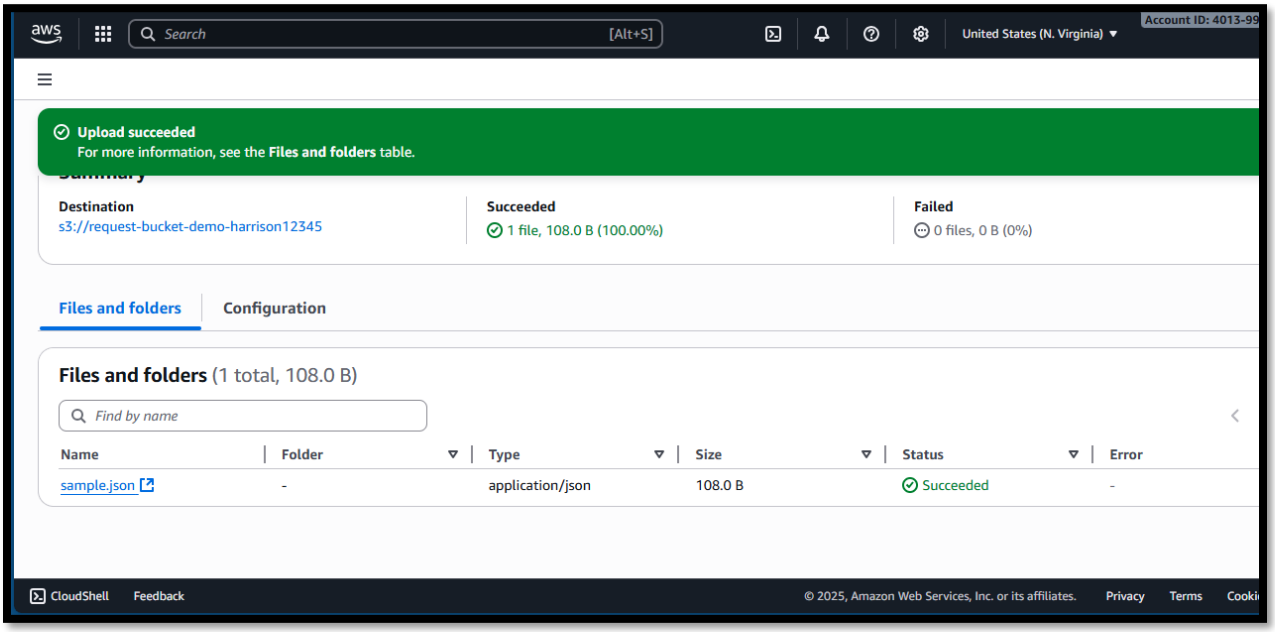
I performed an end-to-end test by uploading a sample file to the input bucket:

1. In the **S3 Console**, I opened the bucket `request-bucket-demo-12345`.
2. I clicked **Upload** and added the file **sample.json**.

This confirmed that the setup can process files end-to-end through the workflow.



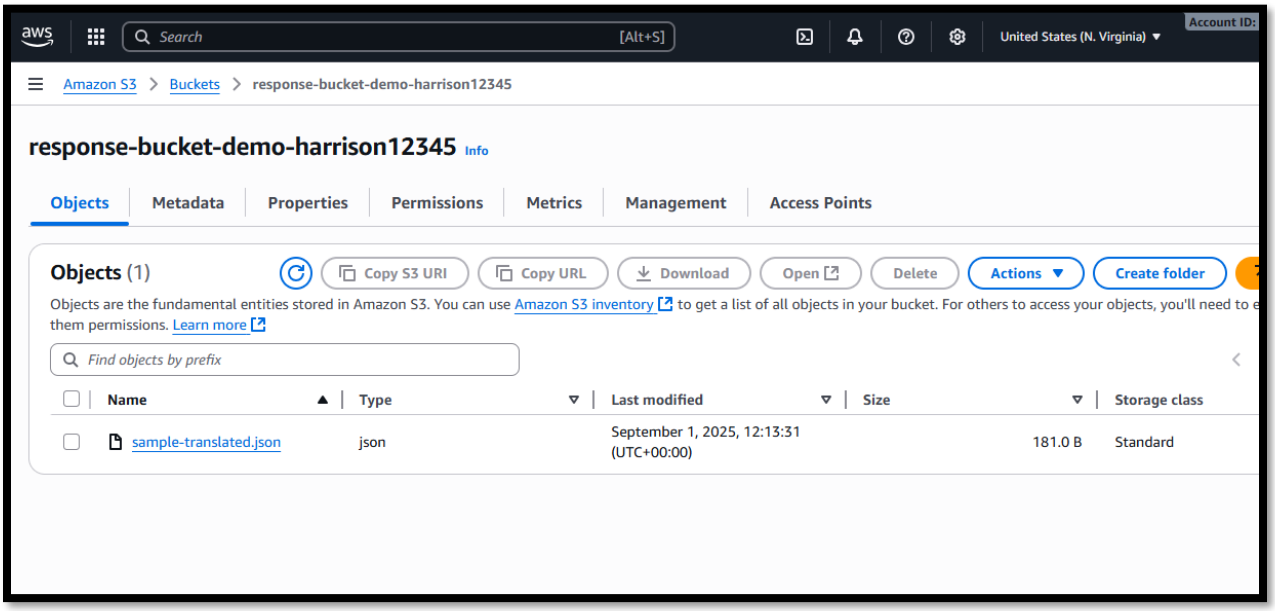
After the file was uploaded, the Lambda function was triggered automatically.



In the **S3 Console**, I opened the bucket response-bucket-demo-harrison12345.

- A file named **sample-translated.json** was present.

- Opening the file showed both the English original text and its Spanish translation.



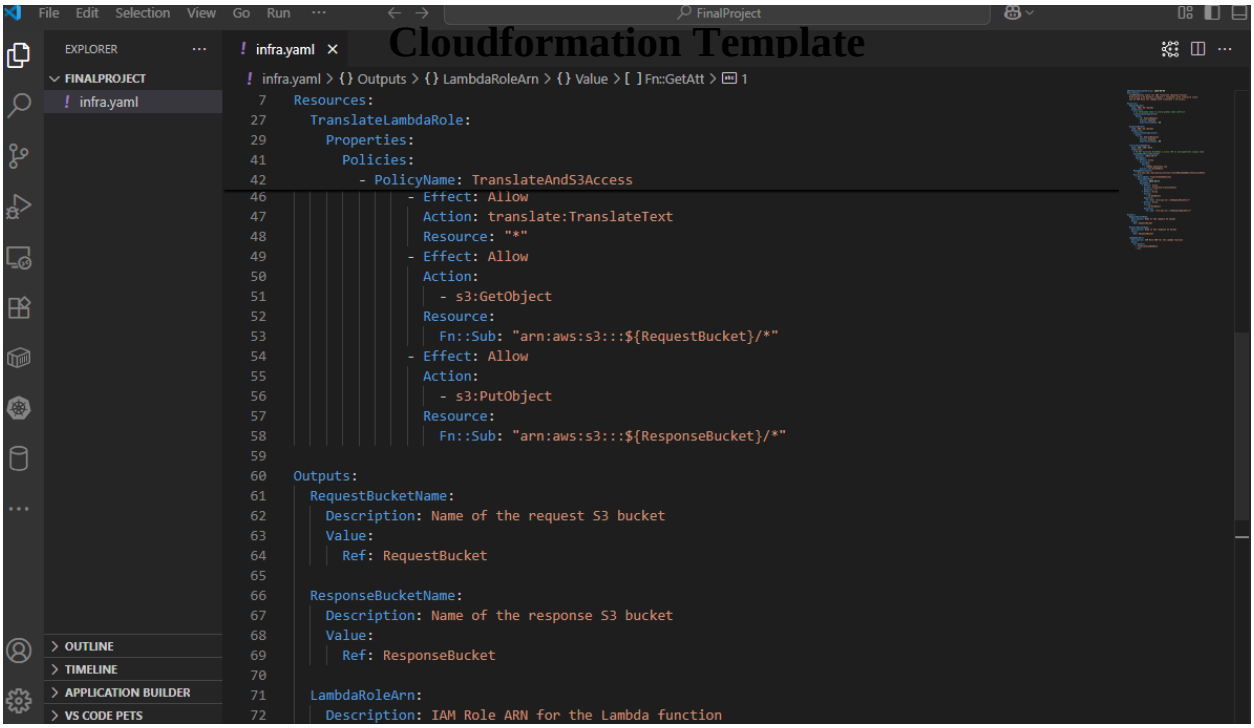
Phase 2 – Add IaC for Buckets and IAM Role

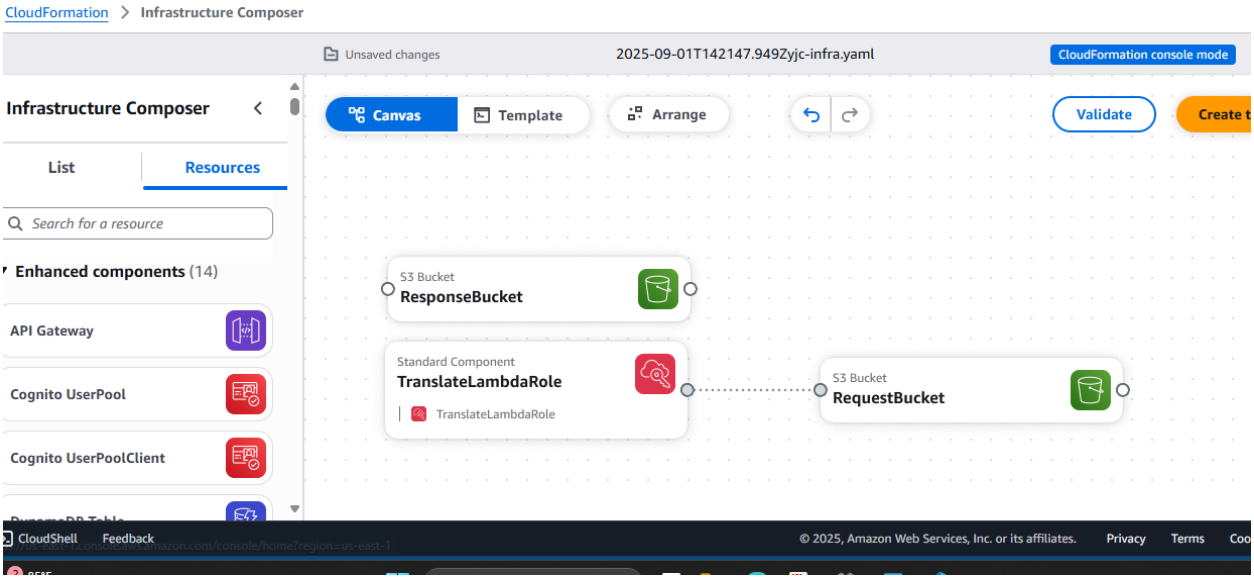
At this stage, I shifted to using **CloudFormation** to provision resources instead of creating them manually. The goal was to align with the project requirements and better understand how the infrastructure can be defined and managed as code. In this phase, both the S3 buckets and the IAM role were created through CloudFormation templates.

Cloudformation – Infrastructure Composer

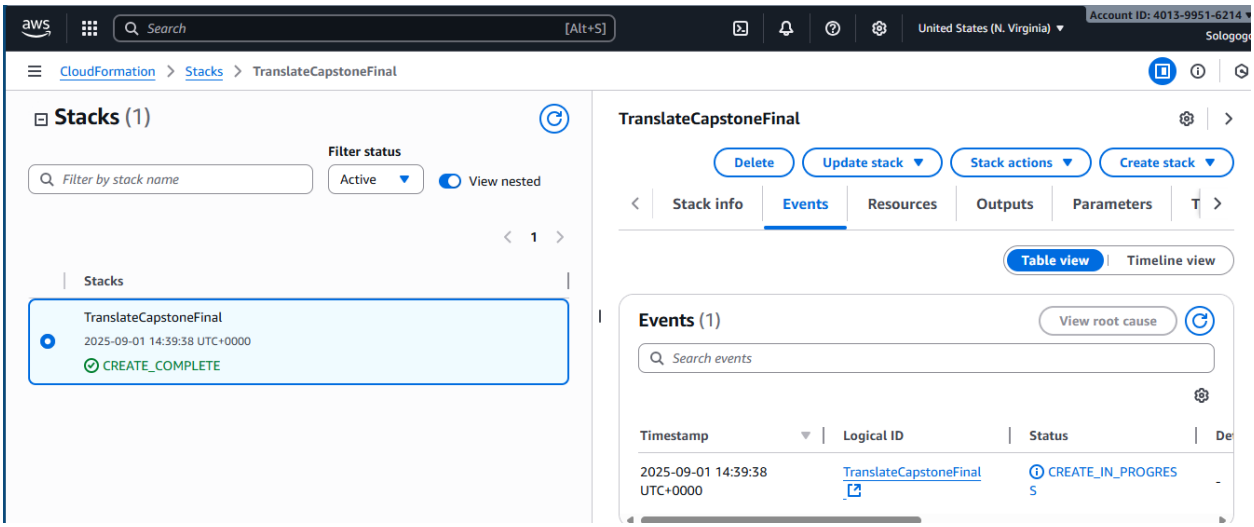
I deployed the resources using **infra.yaml**, which included:

- **S3 Buckets** – Request and response buckets with lifecycle policies applied
- **IAM Role for Lambda** – Configured with permissions for **Translate**, **S3**, and **CloudWatch**





Stack successfully created



Phase 3 – Lambda Function Development

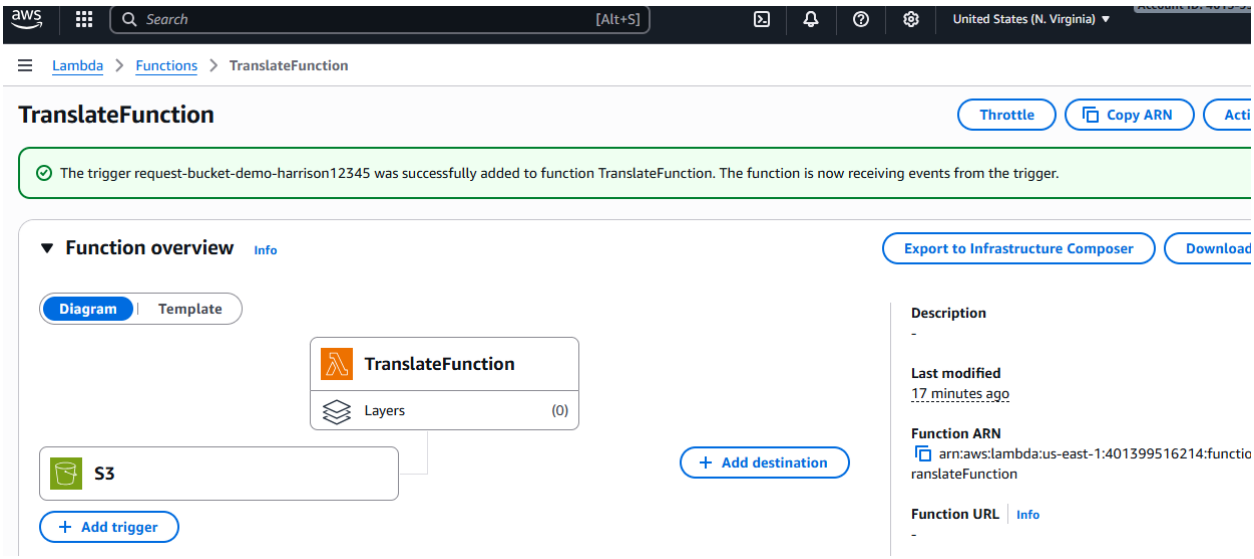
I developed the core Lambda function in **Python** (lambda_function.py) using **Boto3**. The function:

- Reads the uploaded file from the request bucket
- Uses Amazon Translate to translate the text
- Saves the translated output as a JSON file in the response bucket

To make the setup more flexible, I added **environment variables** for the bucket names, so they can be updated dynamically without changing the code.

Phase 4 – Event-Driven Automation

I configured an **S3 event trigger** on the request bucket so that whenever a new file is uploaded, the **Lambda function is invoked automatically**. This ensures the workflow runs in real time without requiring any manual intervention.

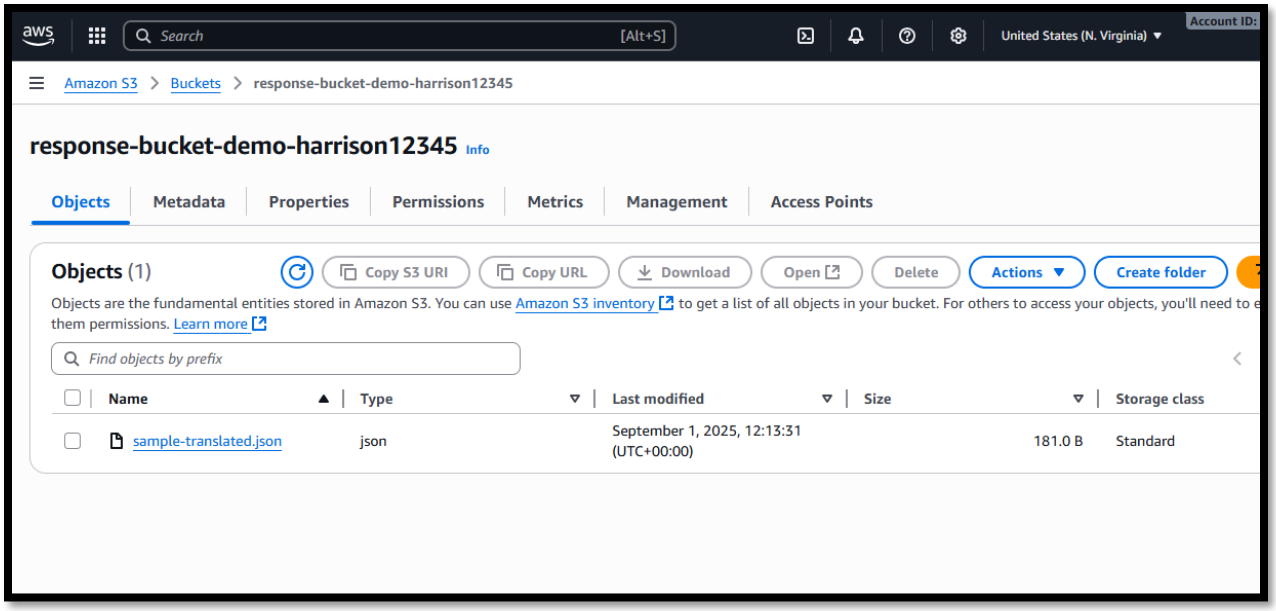


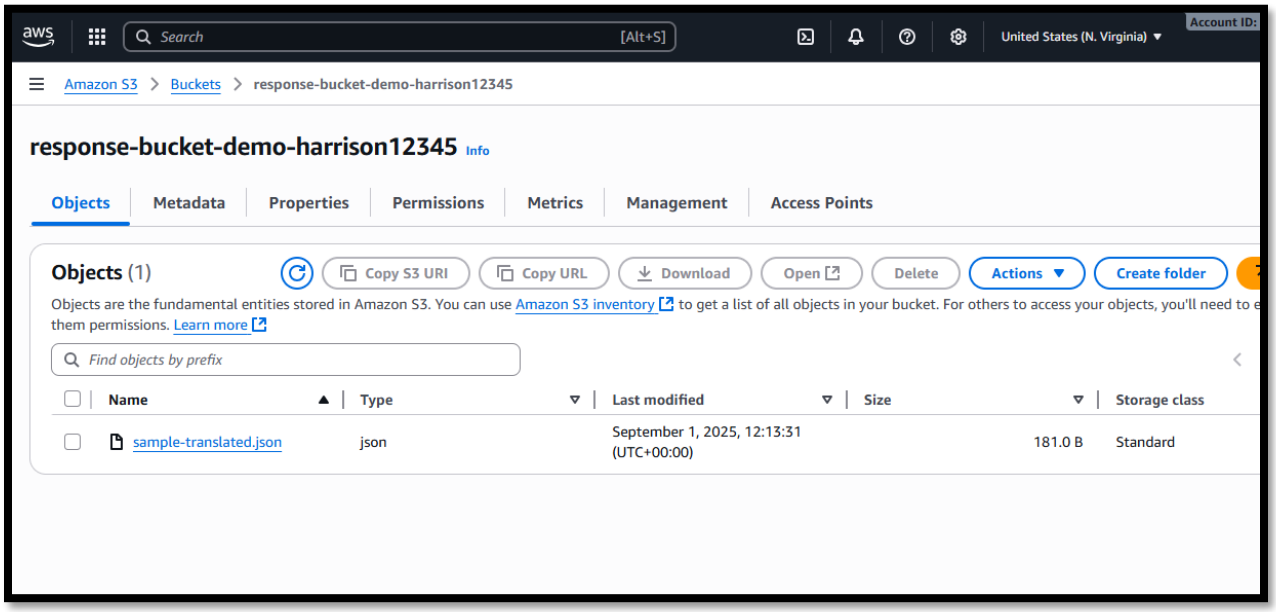
Phase 5 – Testing & Documentation

English

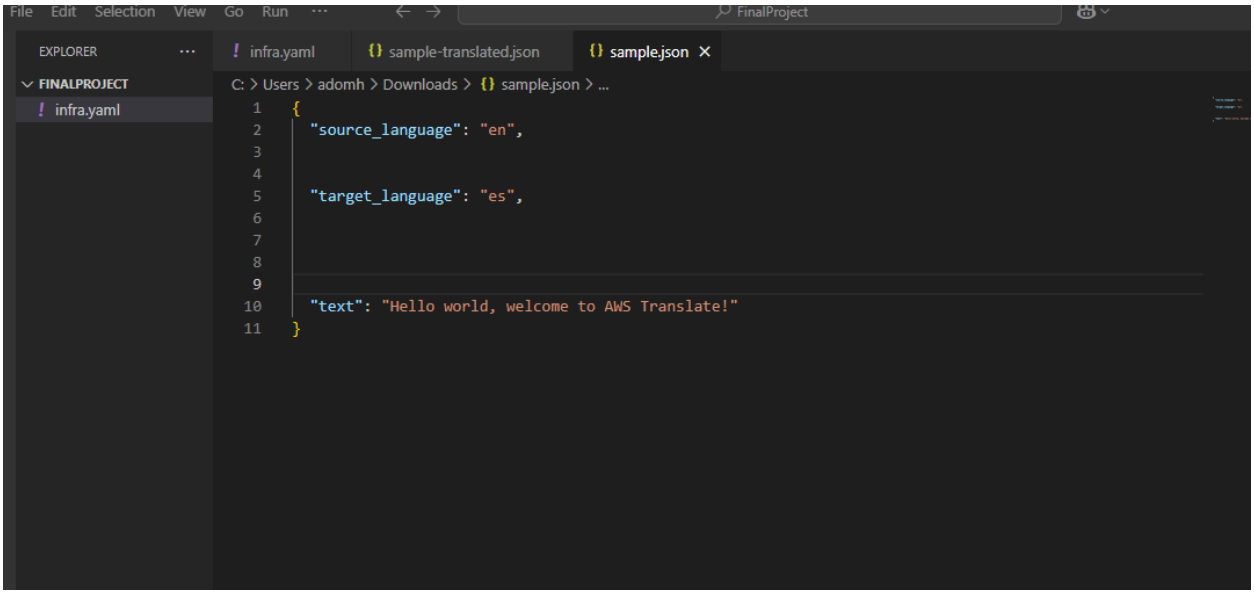
I carried out end-to-end testing to validate the solution:

- Ran translations with files in **English, Spanish.**
- Confirmed the **IAM role permissions** allowed Lambda to access S3, Translate, and CloudWatch as expected
- Documented the setup process, troubleshooting steps, and key lessons learned during the project

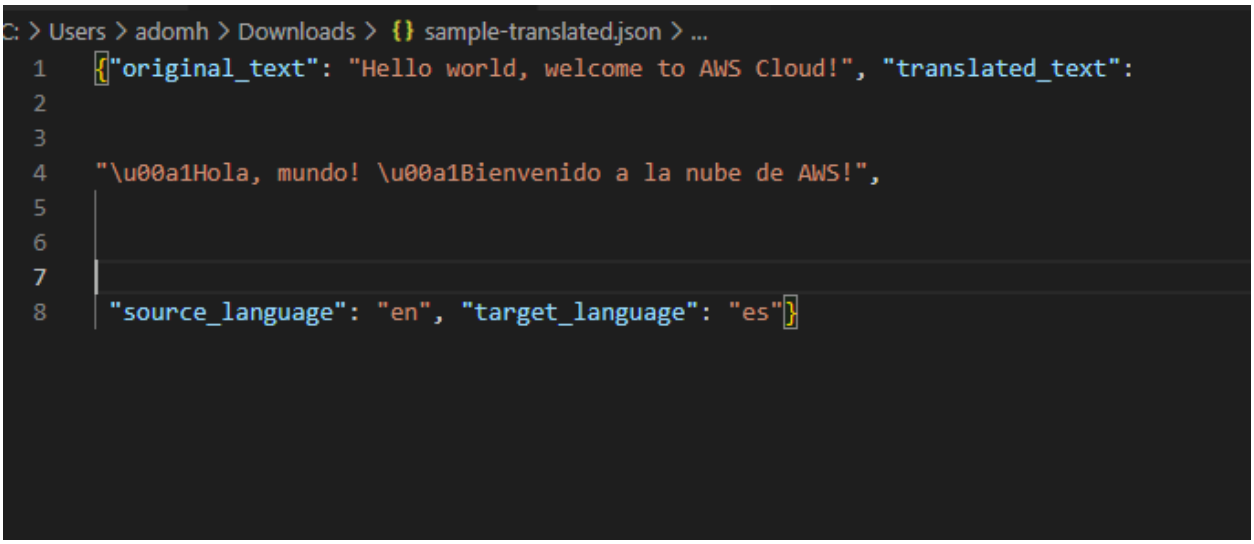




Input JSON (sample.json)



Output JSON (sample-translated.json)



Why I Used the Console First, then CloudFormation

As part of my learning path, I initially built the project resources using the AWS Console (the “click-ops” approach). This was intentional: by creating S3 buckets, IAM roles, and Lambda triggers manually, I was able to **see the relationships**

between services step by step, verify my understanding of access permissions, and debug basic issues in real time.

Once I was confident that I understood the workflow and had tested it successfully end-to-end, I transitioned to the **Infrastructure-as-Code (IaC) approach using AWS CloudFormation**. CloudFormation allowed me to automate the provisioning of the same infrastructure — S3 request/response buckets, lifecycle rules, and IAM roles — in a repeatable, version-controlled way.

This two-stage process not only helped me solidify the fundamentals but also demonstrated my ability to combine **hands-on console setup for learning** with **professional IaC practices for automation and scalability**, which is a critical skill in real-world cloud engineering and aligns with the project prerequisites.

7. Testing & Validation

- Tested multiple language pairs:
    - English → Spanish
    - Spanish → English
- Monitored Lambda executions through **CloudWatch Logs**
- Verified lifecycle policies:
    - Request files expire after **30 days**
    - Response files expire after **60 days**

8. Challenges & Solutions

- **Challenge:** Translate API initially failed with SubscriptionRequiredException
  ☑ **Solution:** Enabled Amazon Translate subscription in the account console
- **Challenge:** CloudFormation stack rollbacks due to hardcoded bucket names
  ☑ **Solution:** Allowed CloudFormation to auto-generate unique bucket names

- **Challenge:** Managing IAM permissions effectively
  ☑ **Solution:** Scoped IAM policies to only the required S3 buckets and Translate API

9. Security Considerations

- IAM roles designed with the **least privilege** principle
- No hardcoded credentials — Lambda uses its IAM execution role
- Lifecycle policies enforce data retention limits
- CloudWatch logging enabled for full visibility and accountability

10. Future Enhancements

- Add **API Gateway** for real-time translation requests via REST API
- Create a **CloudWatch Dashboard** for translation metrics and monitoring
- Configure **S3 Glacier** for automated archiving after retention periods
- Integrate with a **CI/CD pipeline (CodePipeline)** for automated Lambda updates

11. Conclusion

This project demonstrates a robust, serverless AWS architecture that delivers:

- Automated **multilingual text translation**
- A **scalable, event-driven** design
- **Infrastructure automation** with CloudFormation
- A secure, compliant solution that fits within the AWS Free Tier

It highlights practical skills in **AWS serverless services, Infrastructure-as-Code, IAM security, and Python development**.

12. Deliverables

- **Code & Templates**
  - infra.yaml – CloudFormation template
  - lambda_function.py – Translation logic
  - sample.json – Test input file

Lessons Learnt

1. **Start Manual, Then Automate** – Using the AWS Console first gave me a solid understanding of how S3, Lambda, IAM, and Translate connect before moving to IaC.
2. **Value of IaC** – CloudFormation made deployments reproducible, consistent, and easier to manage across environments.
3. **Debugging is Real Work** – Errors like `SubscriptionRequiredException` and stack rollbacks taught me to handle billing, permissions, and naming issues systematically.
4. **IAM Security** – Applying least-privilege policies showed me how critical correct permissions are for stability and security.
5. **Event-Driven Mindset** – S3 triggers introduced me to scalable, cost-efficient serverless design.
6. **Observability** – CloudWatch logs proved essential for debugging and validating the workflow from day one.
7. **Iterative Approach** – Breaking the project into milestones prevented major errors and made progress steady and testable.

Personal Takeaway

This project highlighted for me:

- The power of **AWS serverless services** in addressing real-world business needs
- How **Infrastructure-as-Code** and **event-driven pipelines** form the backbone of modern cloud-native architecture
- The importance of **professional skills** — documentation, troubleshooting, and presenting insights — alongside technical coding skill