**Serverless Facial Recognition System for Employee and Visitor Identification Using AWS**

**Project Based Learning (PBL) Report**

**for the course**

**Cloud Computing- 20CS41003**

**BACHELOR OF TECHNOLOGY**

**IN**

**COMPUTER SCIENCE AND ENGINEERING**

By

**J. Sinduja(21R11A0574)**

**K. Mysanthosh(21R11A0579)**

**R. Naveen(21R11A0589)**

**Under the guidance of**

**Santhoshi**



**Department of Computer Science and Engineering**

**Accredited by NBA**

**Geethanjali College of Engineering and Technology**

**(UGC Autonomous)**

(Affiliated to J.N.T.U.H, Approved by AICTE, New Delhi)

Cheeryal (V), Keesara (M), Medchal.Dist.-501 301.

**December -2024**

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Contents** | **Page No** |
| **1** | **Introduction** | **1** |
|  | **1.1 About the Project** | **1** |
|  | **1.2 Project Outcomes and Objectives** | **1** |
| **2** | **System Design** | **2** |
|  | **2.1 System Architecture** | **2** |
|  | **2.2 Modules** | **2** |
| **3** | **Implementation** | **4** |
|  | **3.1 Modules Implementation** | **4** |
|  | **3.2 Sample Code** | **25** |
|  | **3**.**3 Results** | **28** |
| **4** | **Conclusion** | **29** |
| **5** | **References** | **30** |

1. **INTRODUCTION**
   1. **About the Project**

This project focuses on developing a serverless facial recognition system using AWS, designed to streamline the process of identifying employees and visitors. By uploading an image through the frontend, the system retrieves the name of the individual if they are recognized in the database. The serverless architecture leverages AWS services to ensure cost-effectiveness, scalability, and ease of maintenance, offering a modern approach to identity verification.

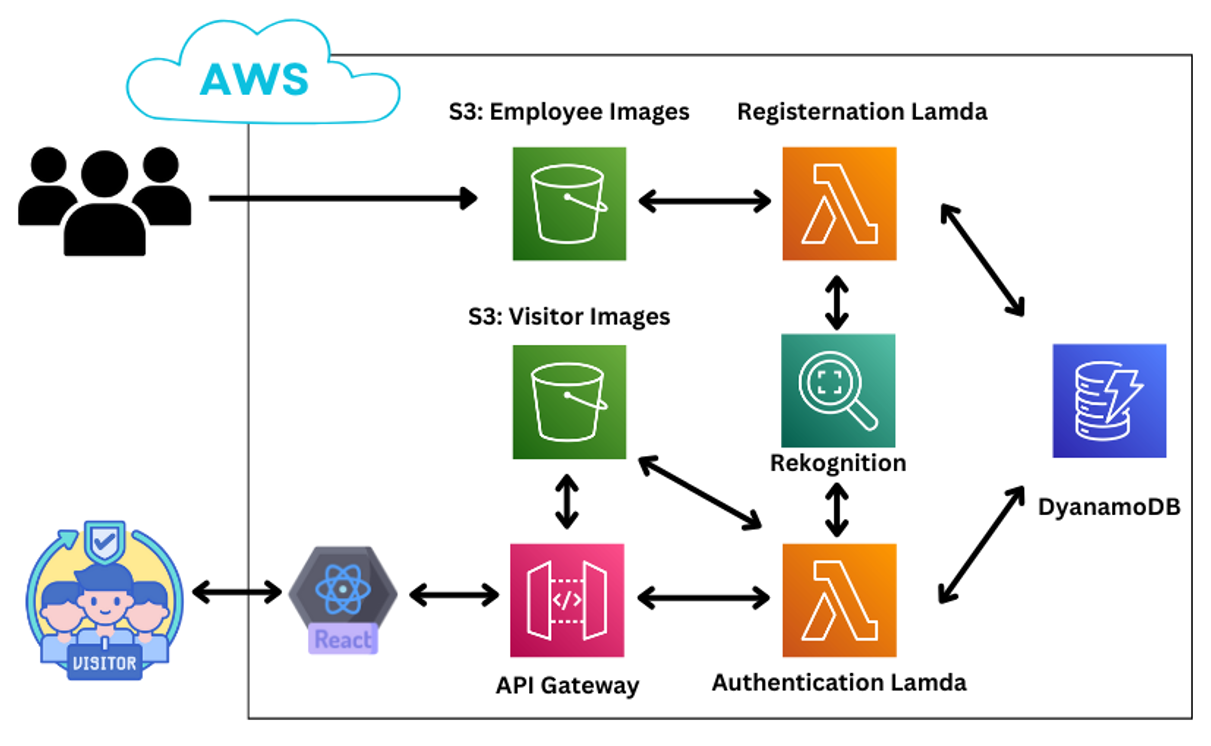
* 1. **Project Outcomes and Objectives**

The primary outcomes of this project include achieving accurate and efficient facial recognition for identifying employees and visitors. By employing a serverless architecture, the system eliminates the need for complex infrastructure management, making it cost-effective and highly scalable. Additionally, the solution provides a seamless user experience, enabling smooth interaction for image-to-name identification processes.

The objectives include:

* Developing a scalable system capable of handling dynamic user data.
* Ensuring high accuracy in facial recognition to minimize false matches.
* Integrating AWS services such as Lambda, Rekognition, and S3 to optimize performance.
* Enhancing security to protect sensitive user data and ensure compliance with privacy standards.
* Creating a user-friendly frontend for smooth interaction with the system.

1. **SYSTEM DESIGN**
   1. **System Architecture**

****

The system follows a serverless architecture built on AWS, ensuring scalability, cost-effectiveness, and efficient processing. It integrates various AWS services to handle user interactions, data storage, and facial recognition seamlessly. The architecture comprises:

* **Frontend:** A React-based user interface for uploading images or entering names.
* **Backend:** AWS services including API Gateway, Lambda functions, Rekognition, and DynamoDB.
* **Data Flow:** Images are stored in S3, processed by Rekognition, and matched against data in DynamoDB, with results displayed on the frontend.
* The architecture ensures minimal infrastructure management and high reliability by leveraging AWS’s serverless model.
  1. **Modules**

1. **Frontend**

* Developed using React to provide an interactive platform for users.
* Enables image uploads and displays results, ensuring a seamless user experience.

1. **API Gateway**

* Manages communication between the frontend and backend.
* Routes requests to the appropriate Lambda function.

1. **Amazon S3**

* Employee Images Bucket: Stores registered employee images for recognition.
* Visitor Images Bucket: Holds visitor images temporarily for processing.

1. **AWS Lambda**

* Registration Lambda: Registers new employees and stores their images in S3.
* Authentication Lambda: Authenticates requests and matches input to existing records.

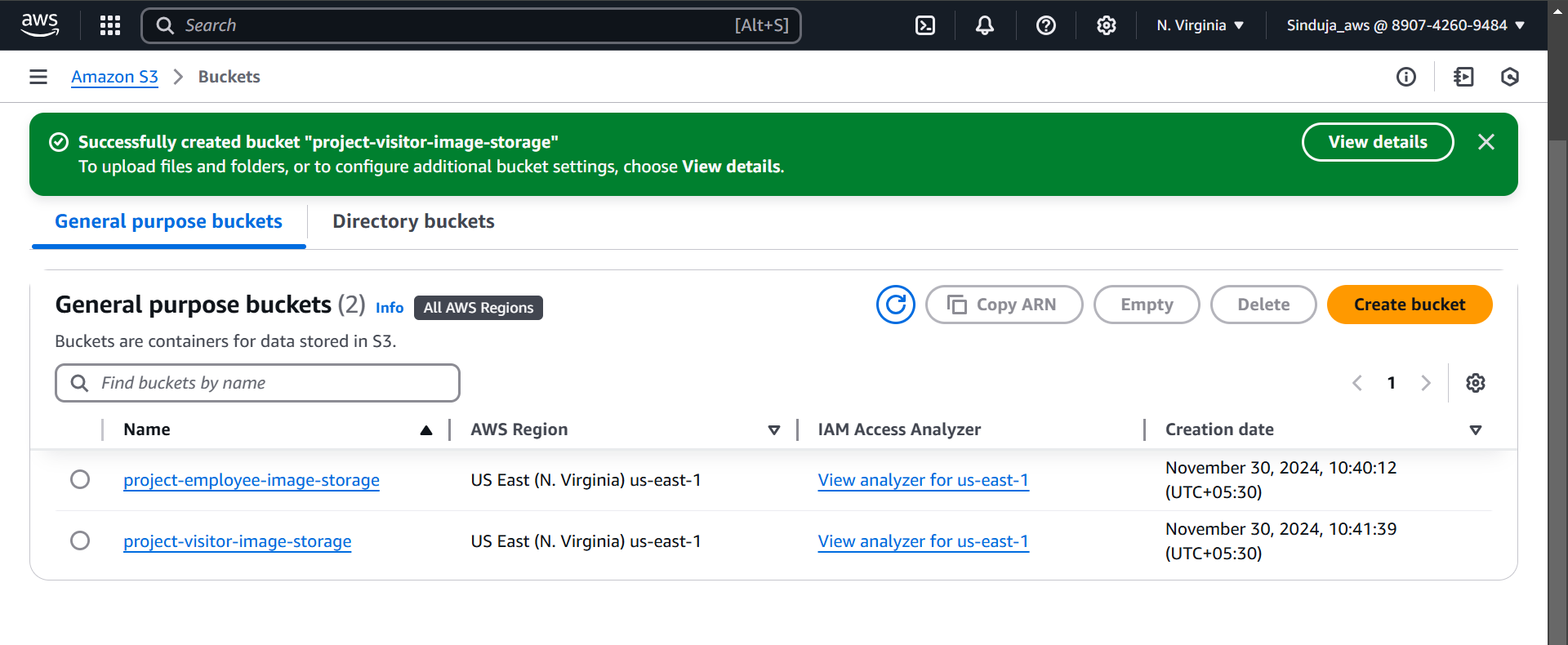
1. **Amazon Rekognition**

* Performs facial recognition and matches uploaded images against stored data.

1. **DynamoDB**

* Stores metadata like employee details and image references for efficient lookups.

1. **IMPLEMENTATION**
   1. **Modules Implementation**
      1. **Amazon S3**



In this project, two Amazon S3 buckets were created to efficiently store and manage images:

1. **project-employee-image-storage**

This bucket is designated for storing the images of registered employees. It serves as a permanent storage solution, allowing the system to access these images for facial recognition and identification purposes

1. **project-visitor-image-storage**

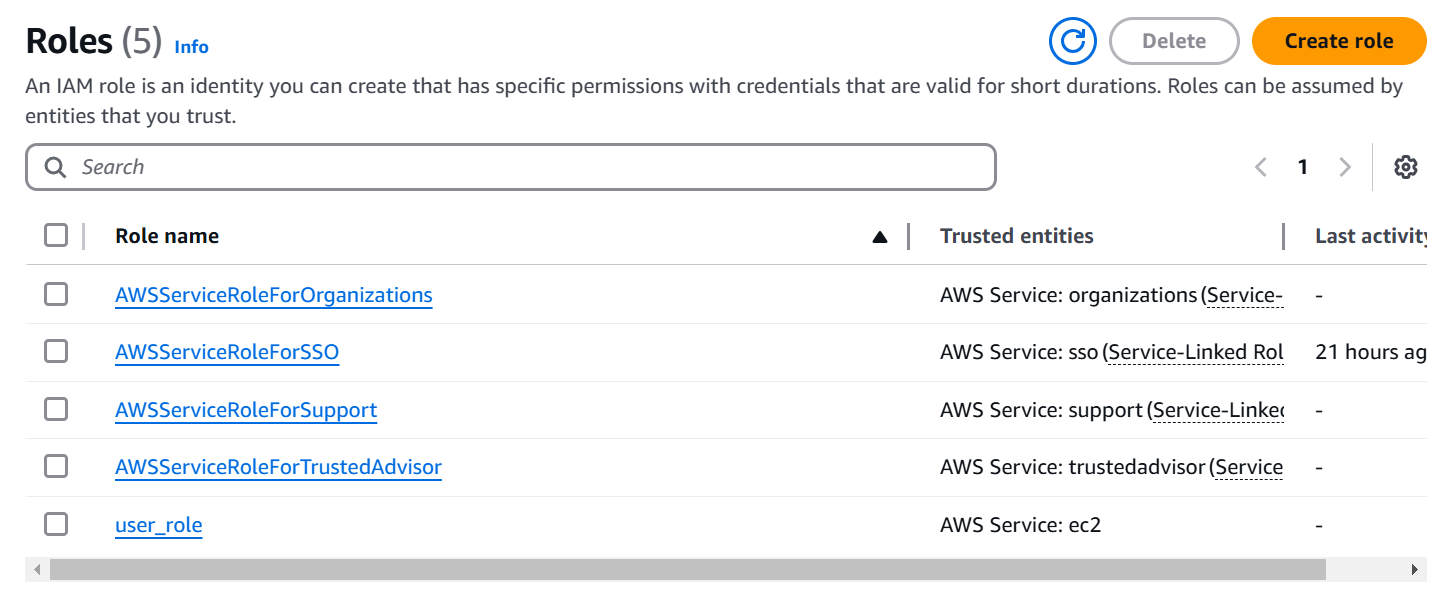
This bucket is designed to temporarily store the images of visitors uploaded for identification. Once the facial recognition process is completed and the visitor's data is matched or processed, the images can be retained or deleted, depending on the system's requirements and data retention policies.

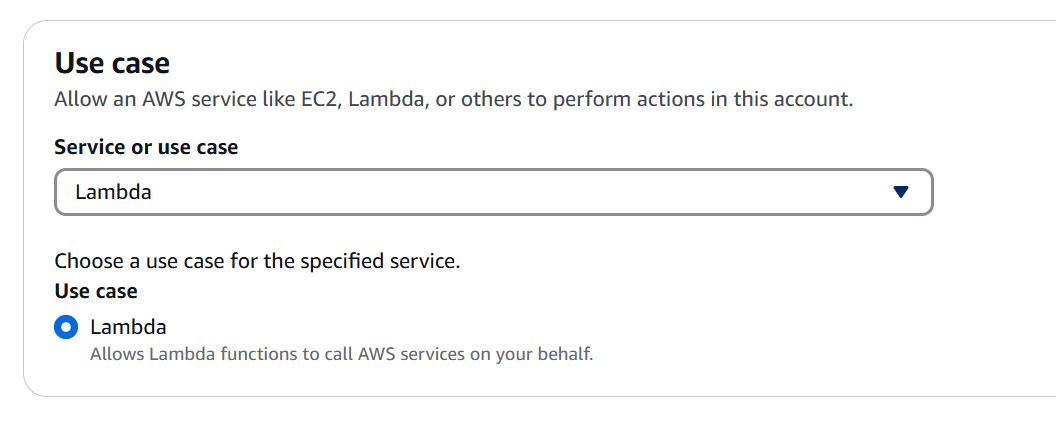
* + 1. **Project-Employee-Registration Lambda Function**

To set up the project-employee-registration Lambda function, the following steps were performed:

1. **Creating an IAM Role**

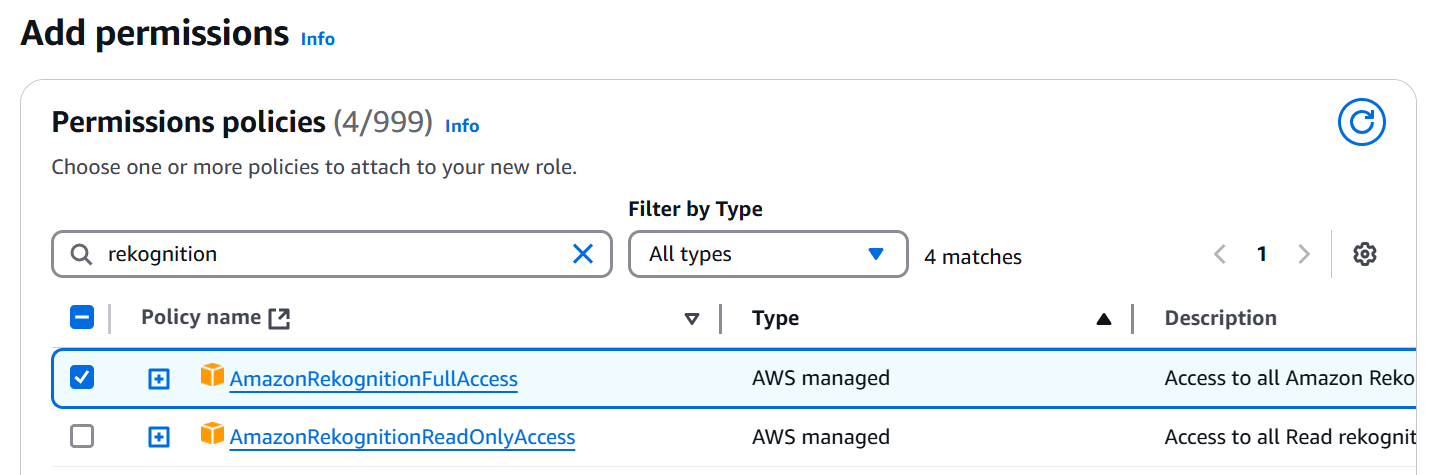
A dedicated IAM role was created to provide the Lambda function with the necessary permissions.

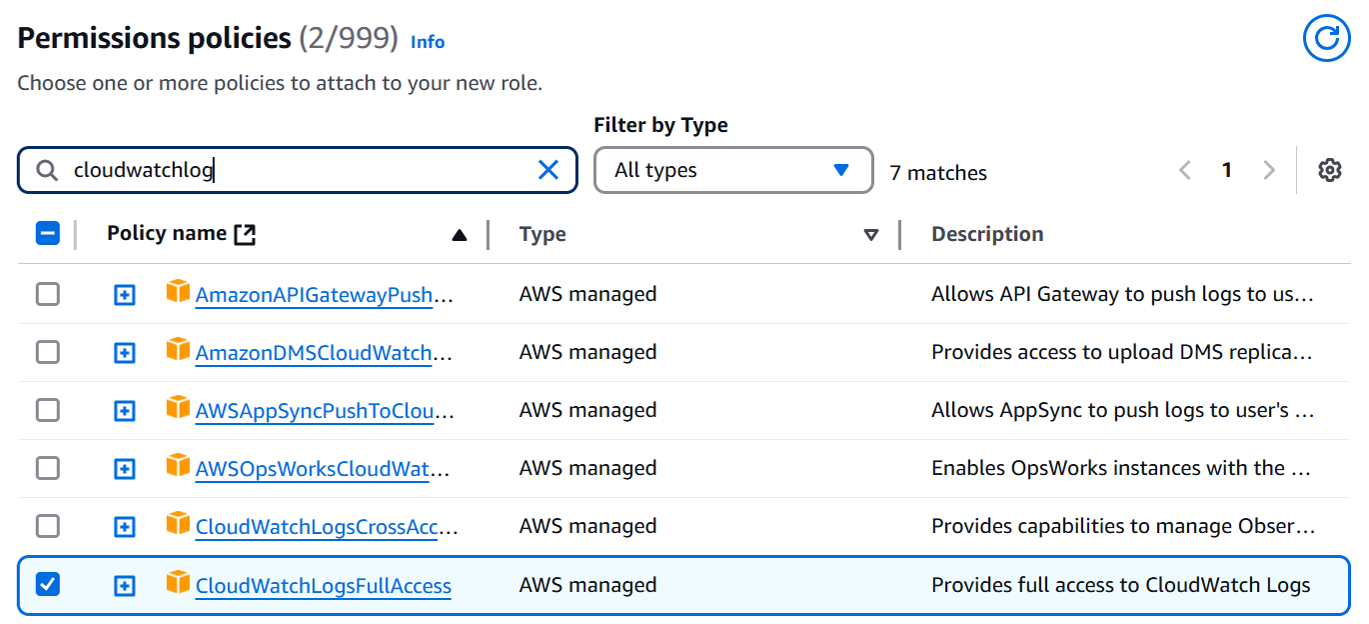


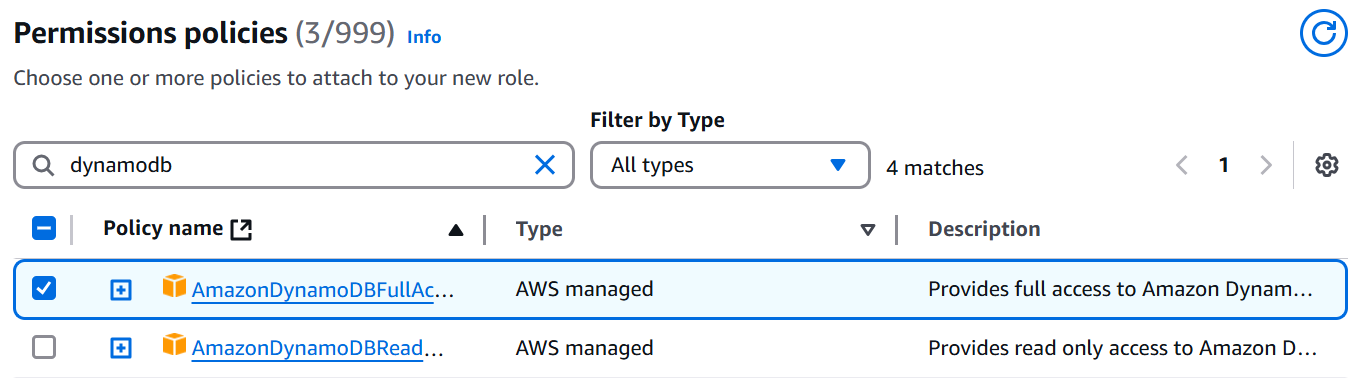


The role included full-access permissions for the following AWS services:

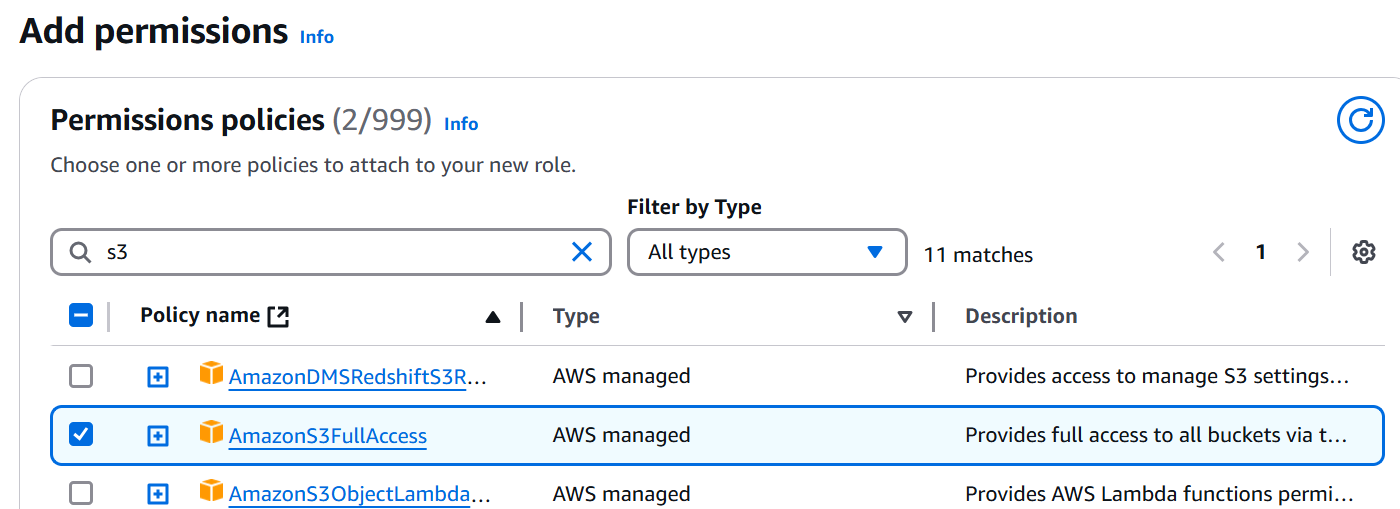
* **Rekognition:** For facial recognition and indexing.

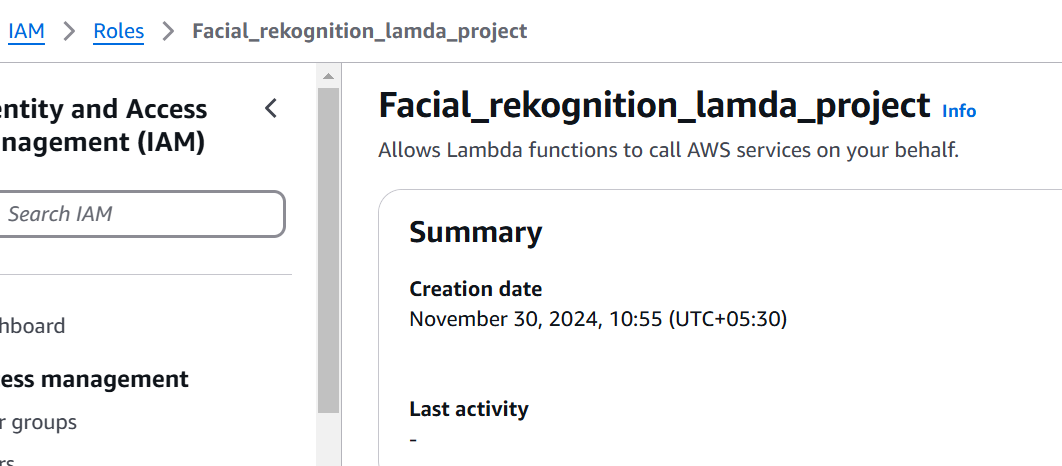


* **CloudWatch Logs:** To enable logging for monitoring and debugging. 
* **DynamoDB:** To store and retrieve employee details and metadata.



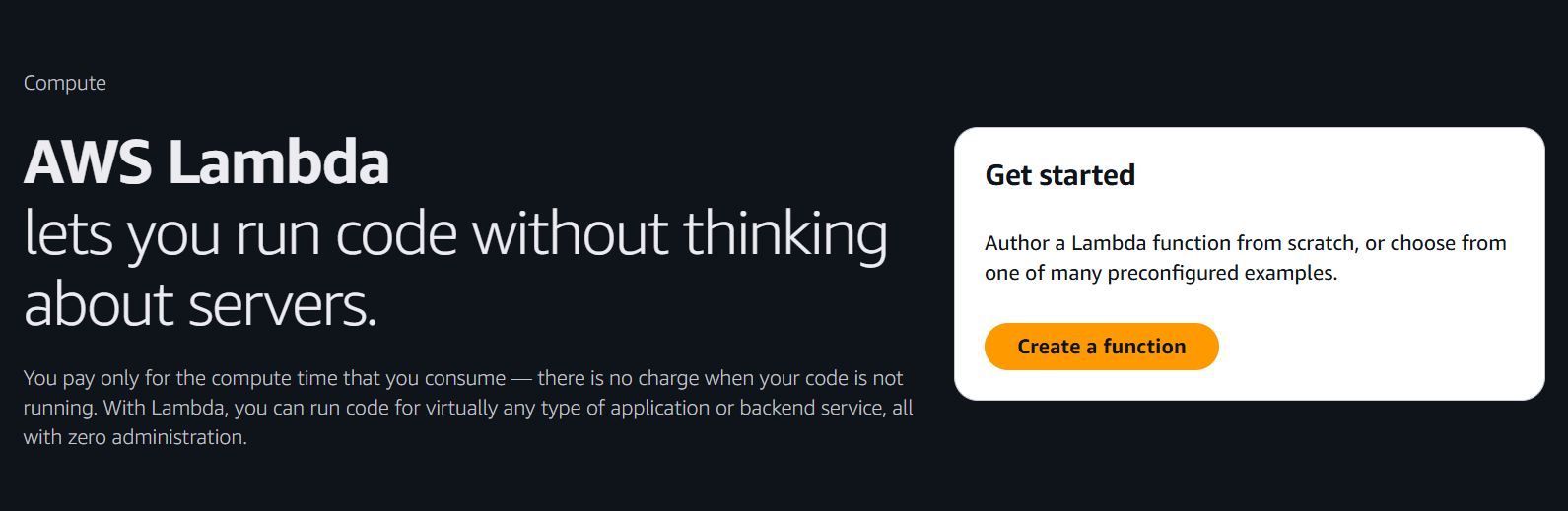
* **S3:** For storing employee images in the project-employee-image-storage bucket.

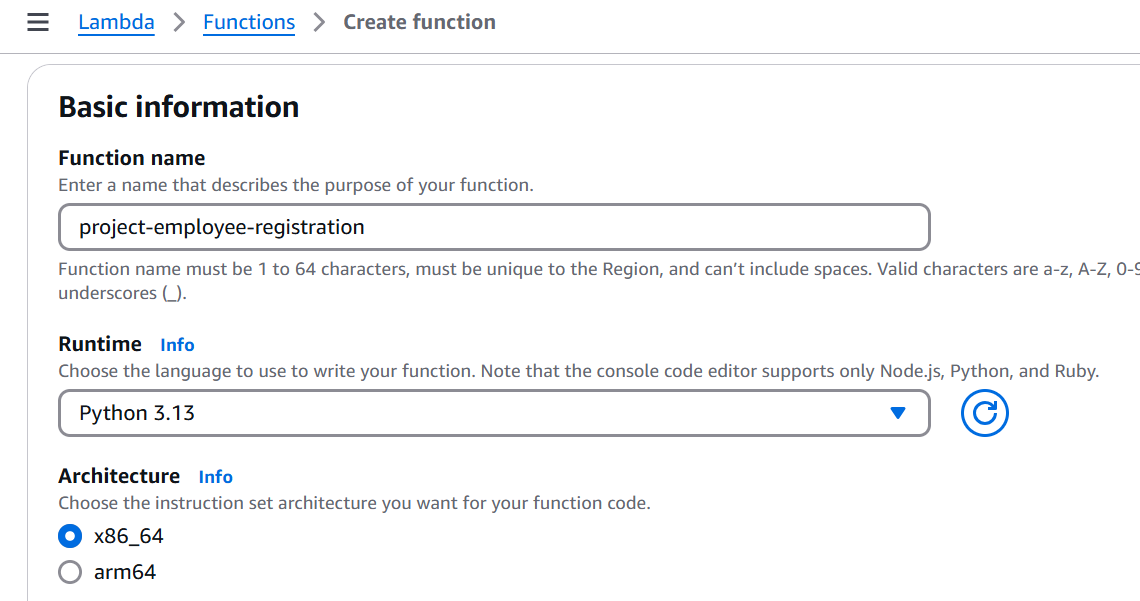


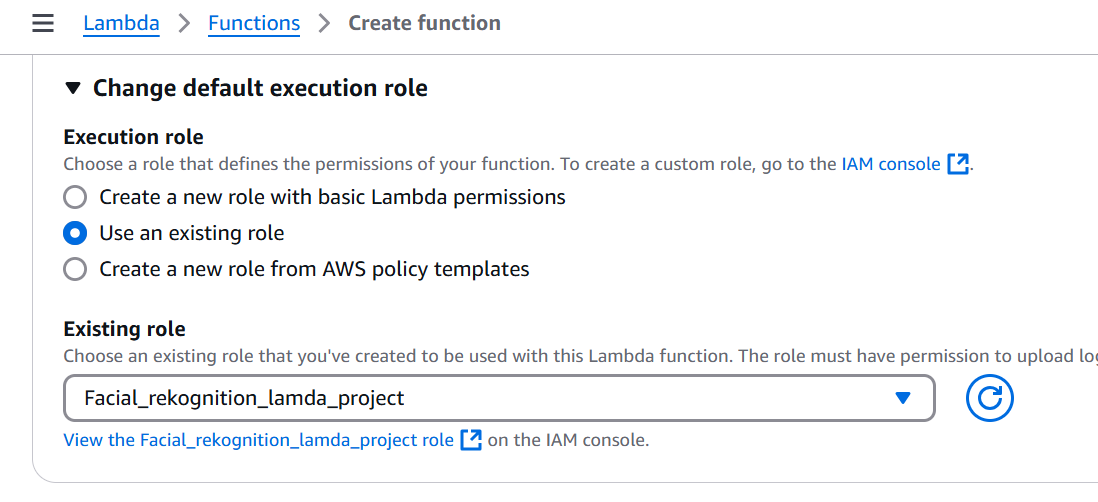


1. **Setting Up the Lambda Function**

The project-employee-registration Lambda function was created with the runtime set to Python 3.13 for compatibility and the latest features. Under the Execution Role section, the Use an existing role option was selected, linking the function to the recently created IAM role with full-access permissions for Rekognition, CloudWatch Logs, DynamoDB, and S3. This setup ensured seamless integration with AWS services for efficient employee registration.

****

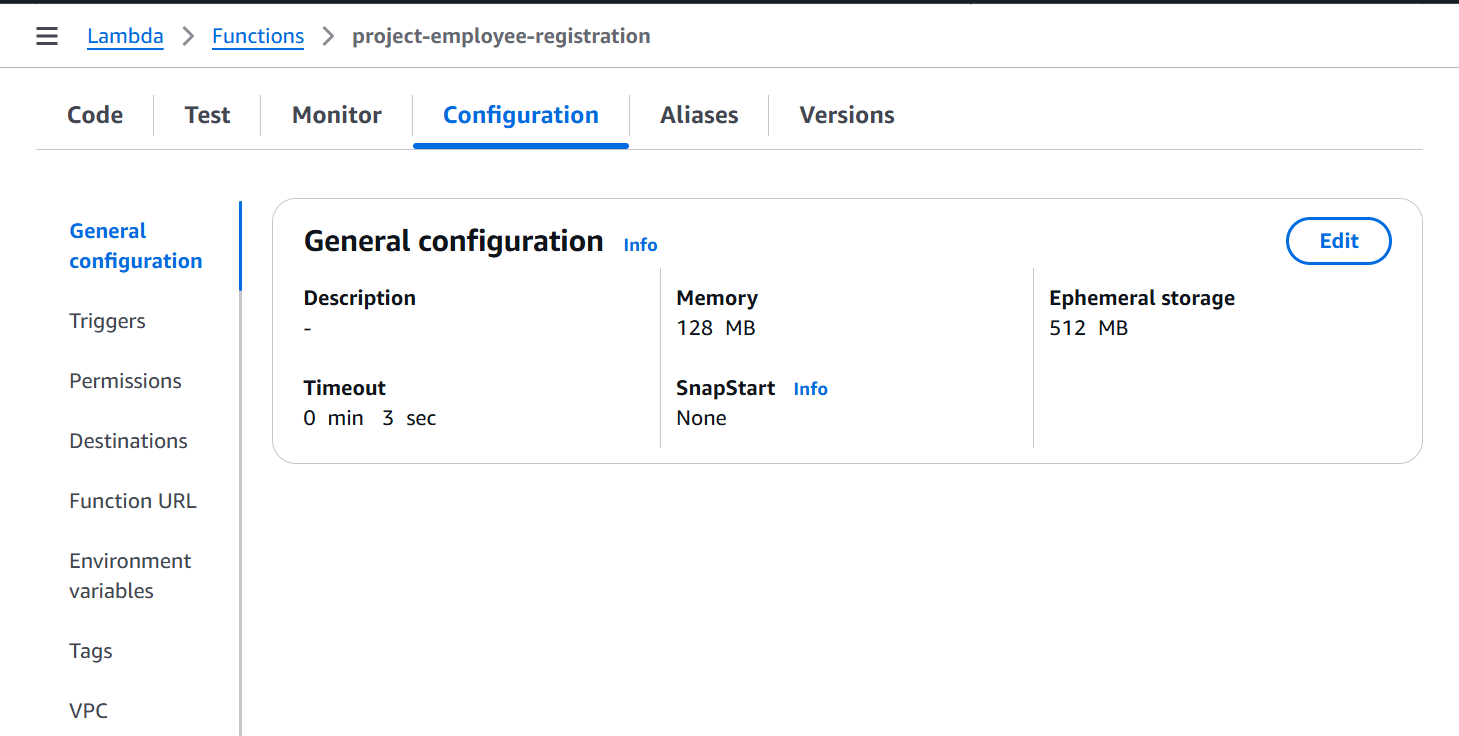
****

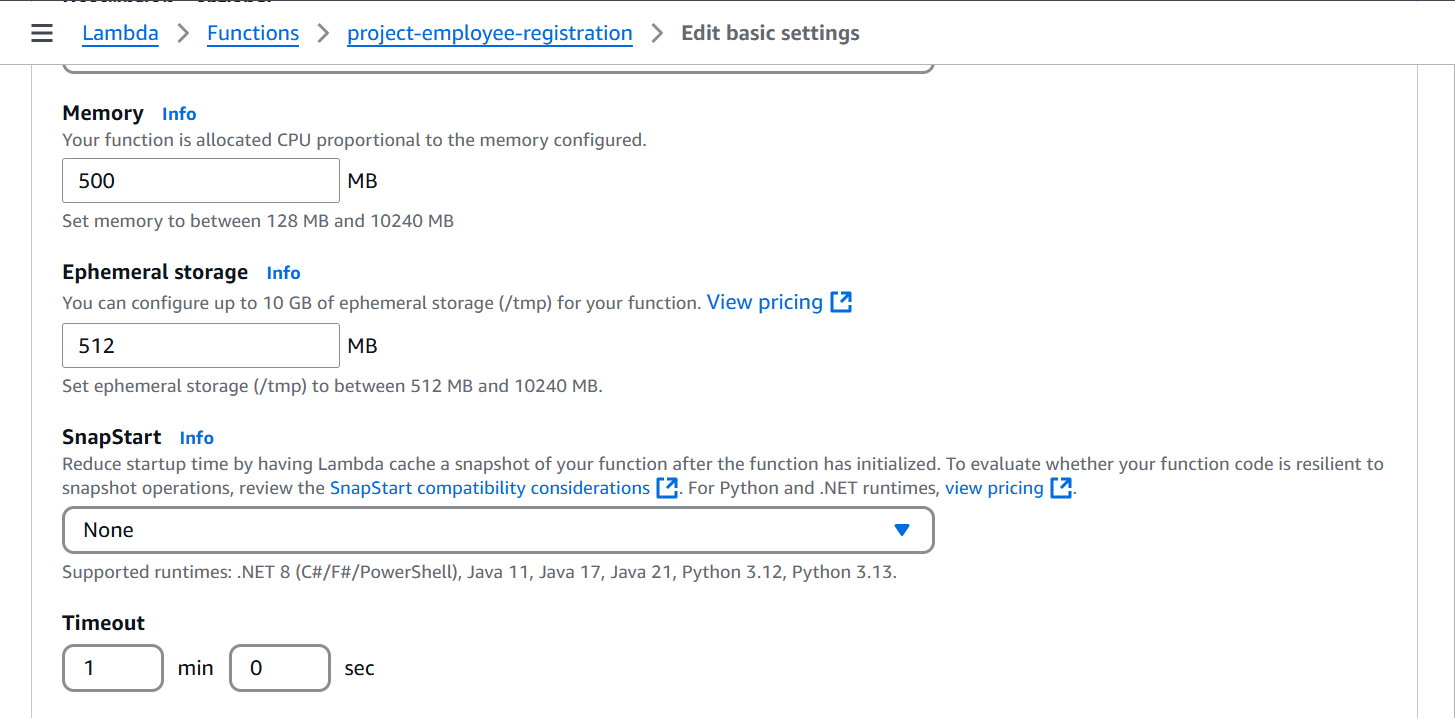
****

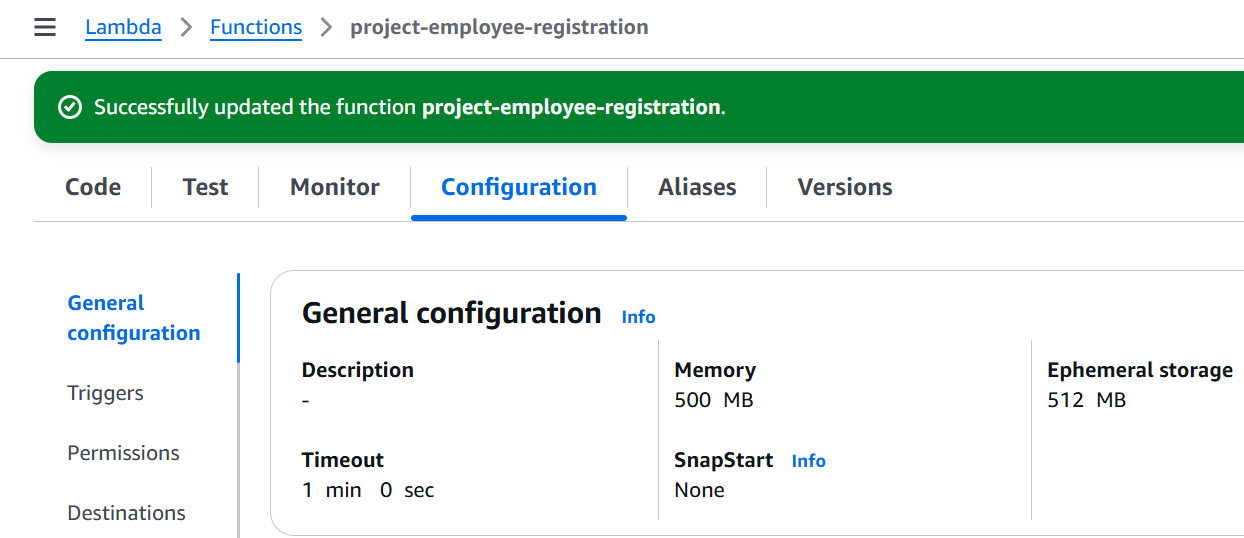
1. **Updating Lambda Configurations**

After creating the project-employee-registration Lambda function, we optimized its performance by modifying the Basic Settings under the Configuration tab:

* Memory: Increased to 500 MB to handle image processing tasks efficiently
* Timeout: Set to 1 second to ensure quick execution while avoiding unnecessary delays.

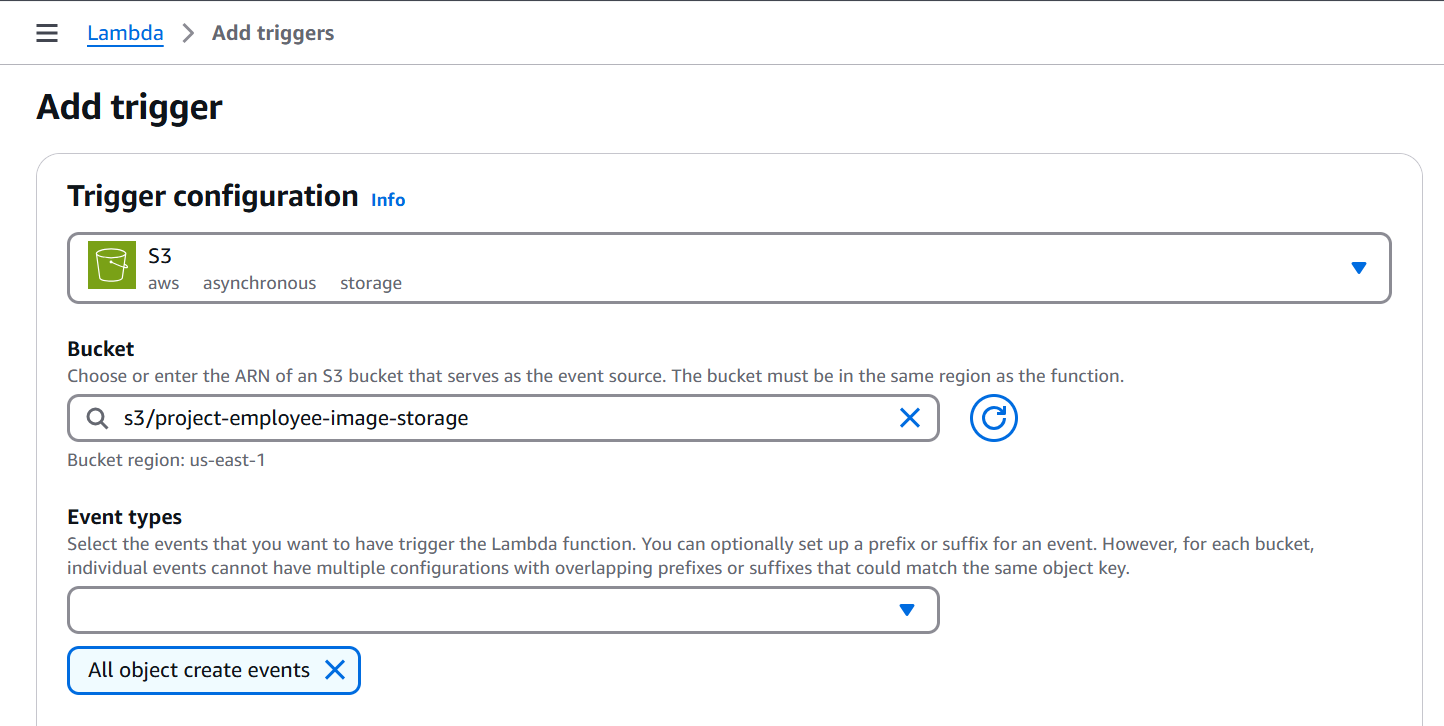


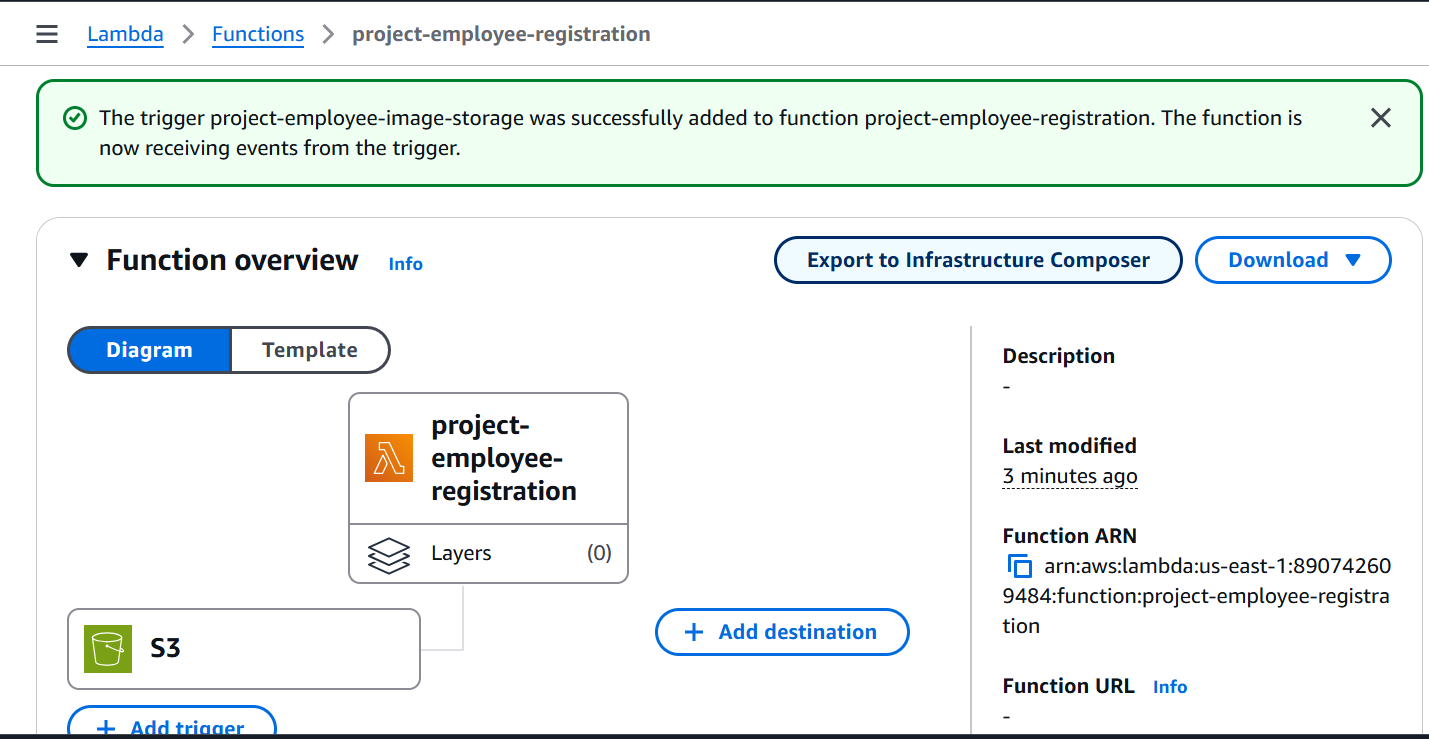




1. **Adding an S3 Trigger**

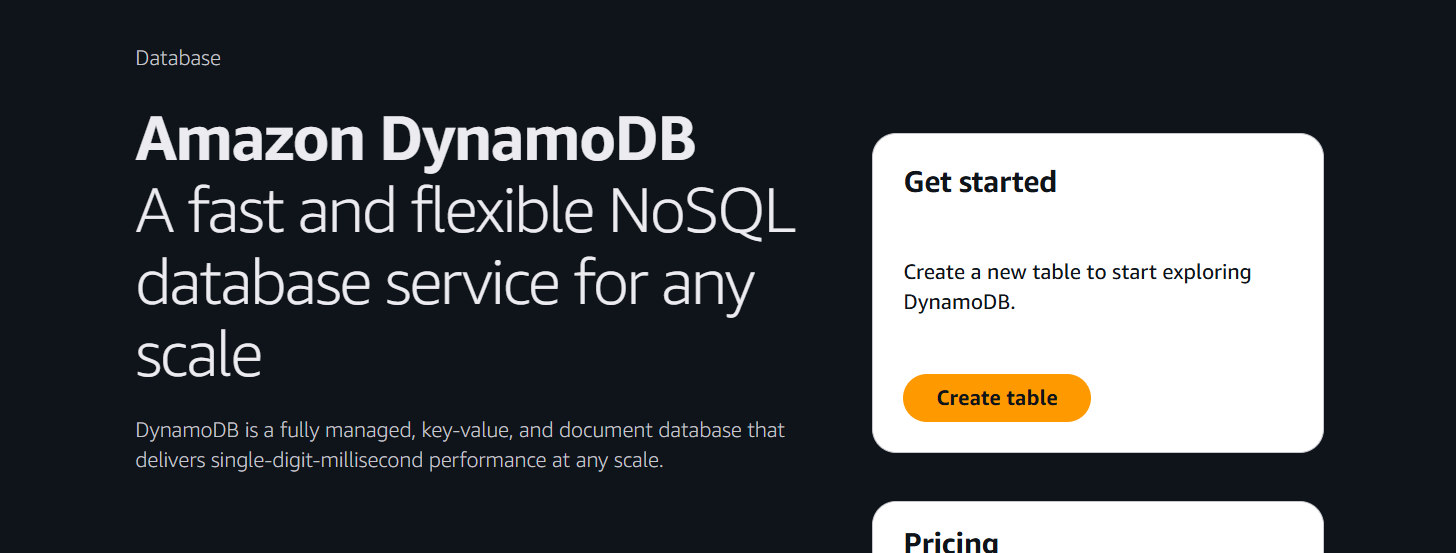
An S3 trigger was created for theproject-employee-registration Lambda function by linking it to the project-employee-image-storage bucket. This configuration ensures that the Lambda function is automatically invoked whenever a new image is uploaded to the bucket.

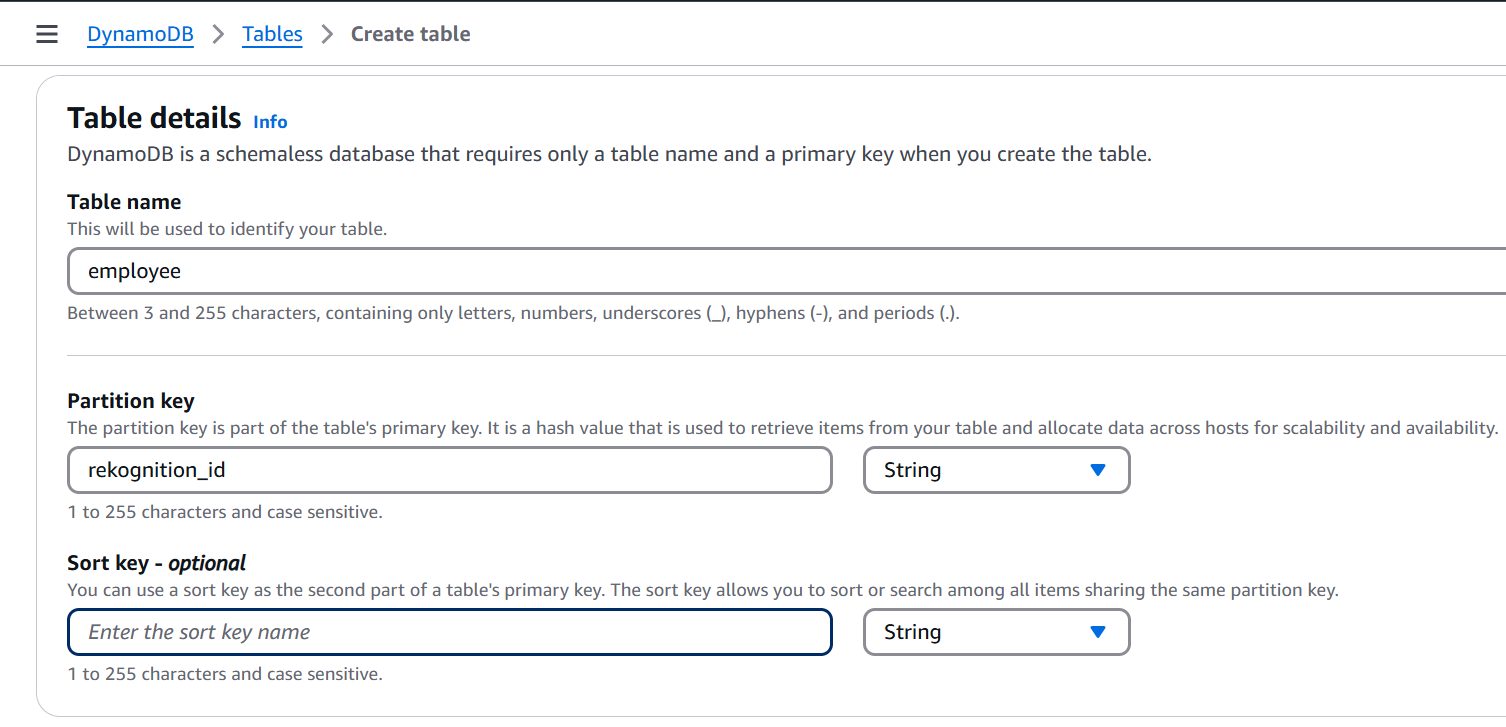


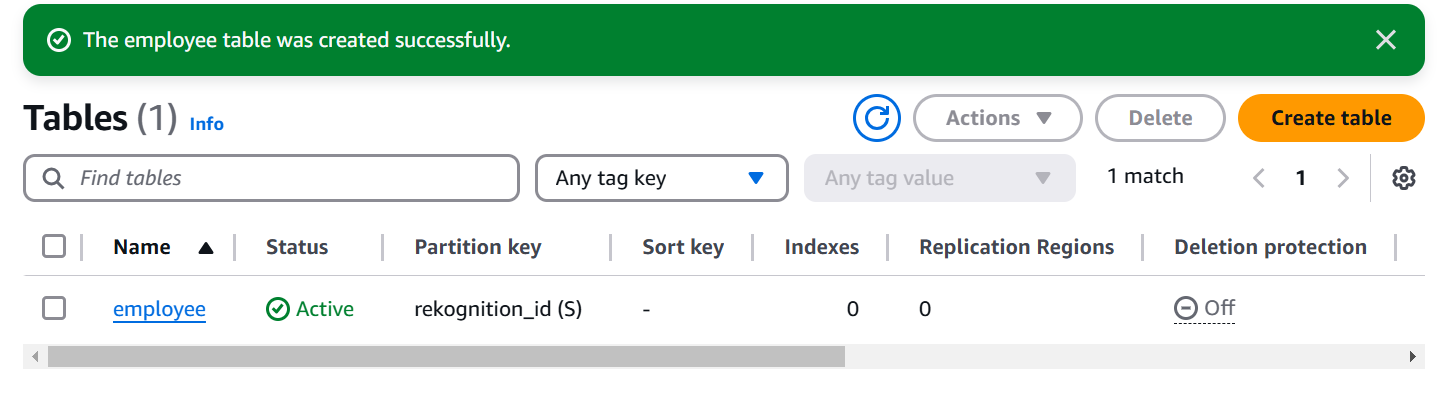


* + 1. **DynamoDB**

**Creating the DynamoDB Table**

****





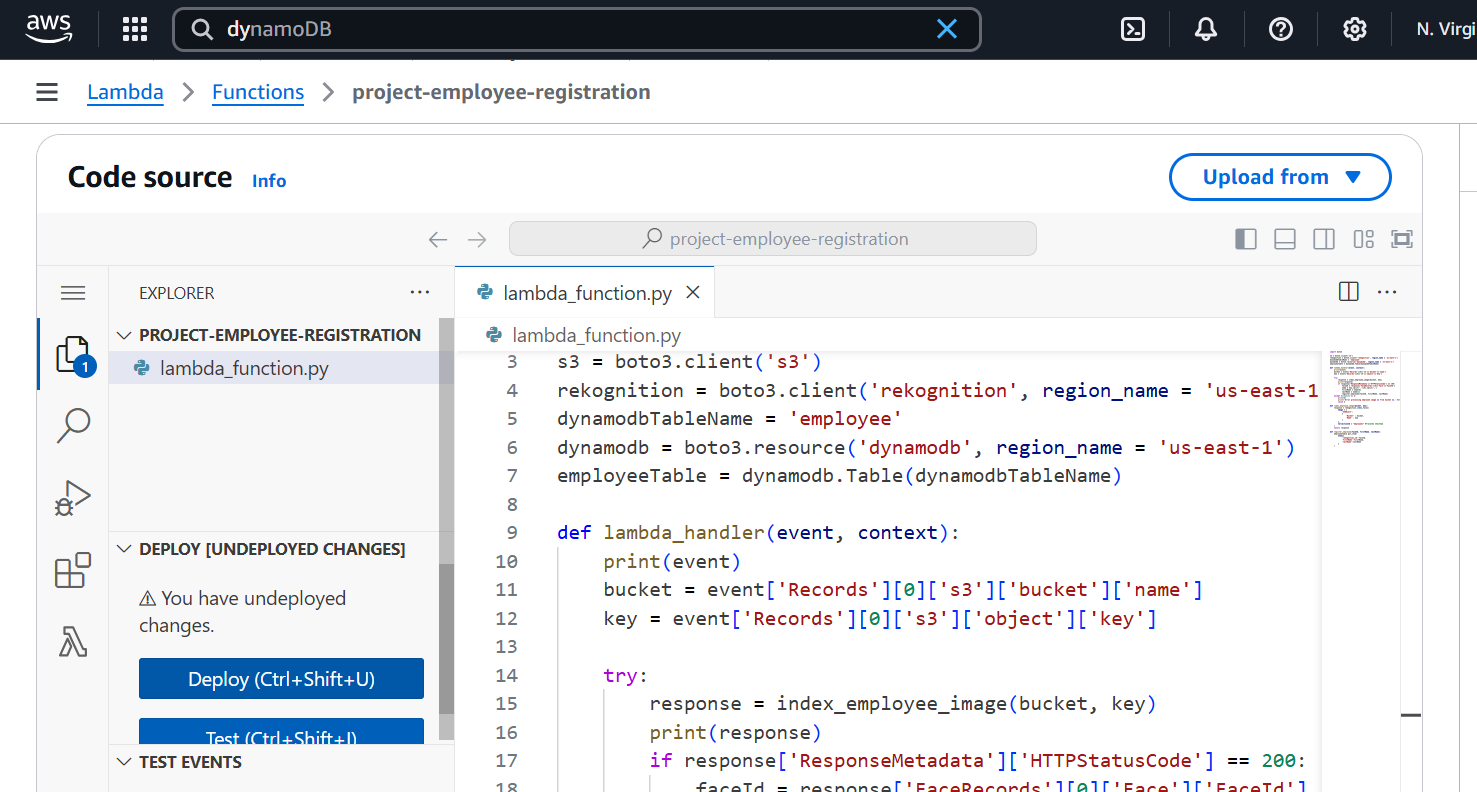
* + 1. **Writing and Deploying the Lambda Function Code**

1. **Writing the Code**

The code was written in Python and used the boto3 library to interact with AWS services such as S3, Rekognition, and DynamoDB.

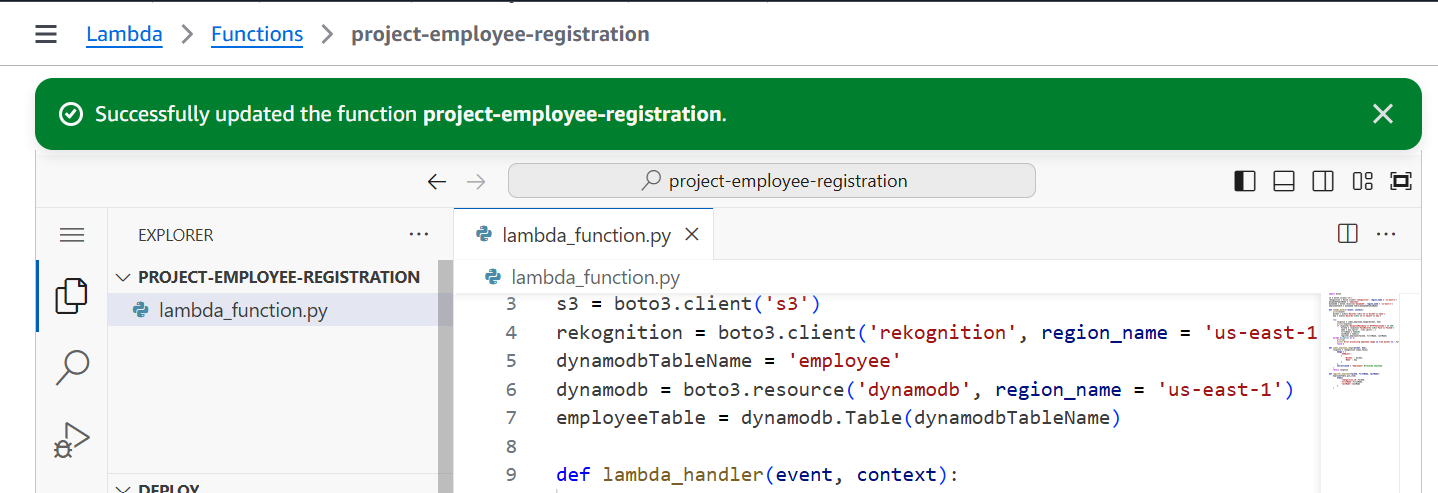
Key functionalities included:

* Extracting the bucket name and image name from the S3 trigger event.
* Indexing the employee image using Rekognition to generate a unique rekognition\_id (Face ID).
* Parsing the image file name to retrieve the employee's first and last name (e.g., FirstName\_LastName.jpg).
* Storing the employee details (Face ID, first name, last name) in the DynamoDB table.



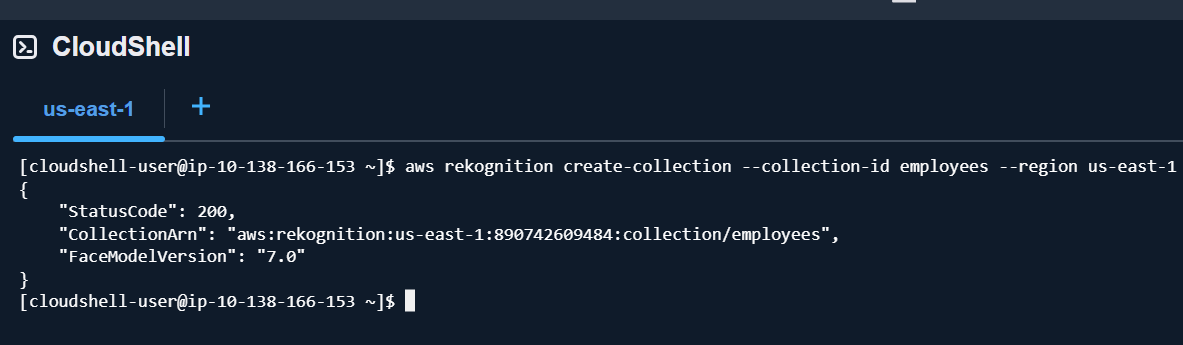
1. **Deploying the Lambda Function**

The Python code was uploaded to the Lambda console.



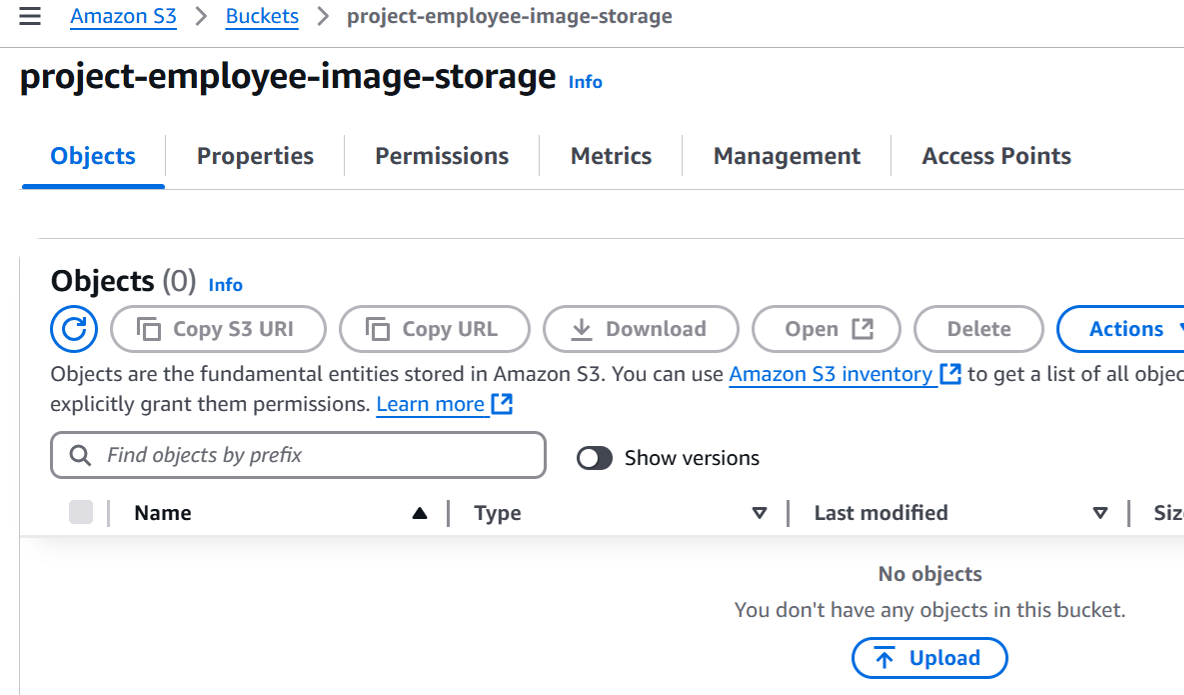
* + 1. **Creating a Rekognition Collection**

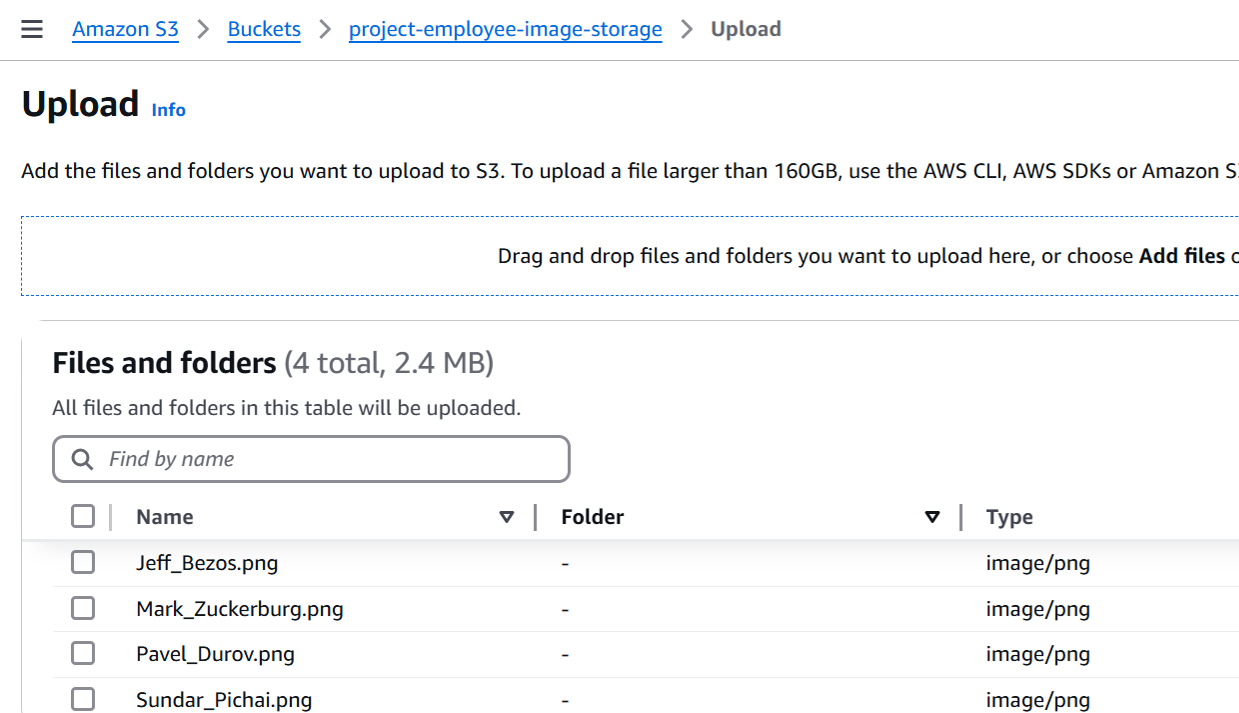
To store and manage facial data, we created a Rekognition collection named employees using the AWS CLI. This collection serves as a repository for indexed faces, enabling quick and accurate retrieval during image recognition.

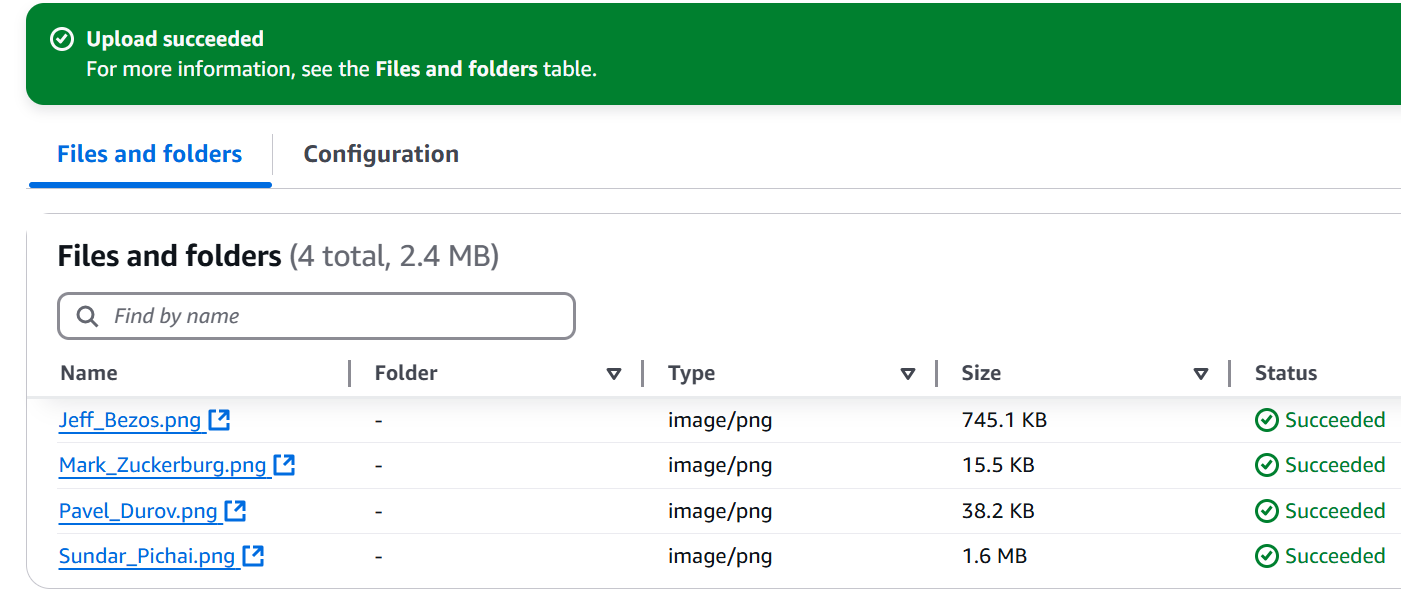


* + 1. **Testing the Registration Flow**

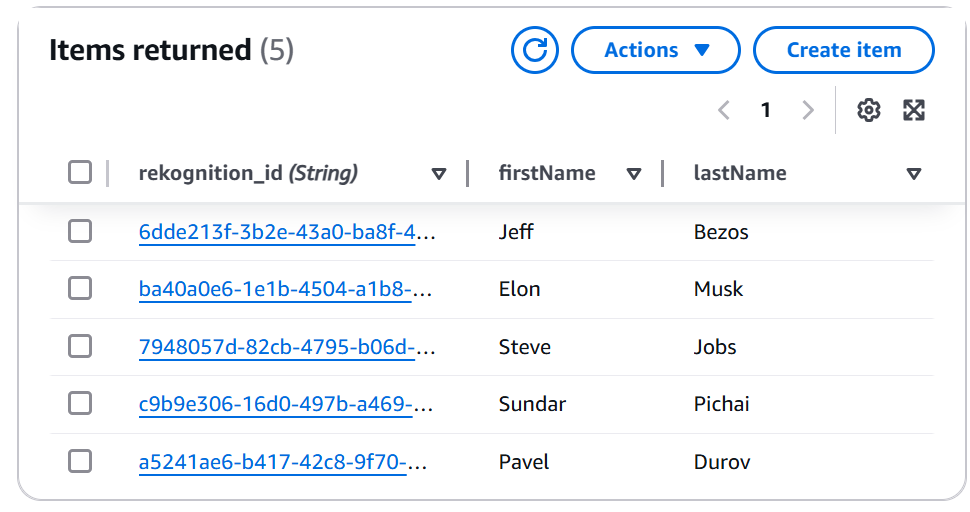
An employee image named in the format FirstName\_LastName.png was uploaded to the project-employee-image-storage S3 bucket.







The S3 trigger automatically invoked the project-employee-registration Lambda function whenever an image was uploaded to the S3 bucket. The function processed the image by indexing it in the Rekognition collection, generating a unique rekognition\_id (Face ID) to identify the employee. Finally, the employee's details, including the rekognition\_id, first name, and last name, were stored in the DynamoDB table for seamless retrieval and identification.



* + 1. **Project-Employee-Authentication Lambda Function**

1. **Project-Employee-Authentication creation**

To enable authentication, we created a Lambda function named project-employee-authentication. During the creation process:

* We selected Python 3.13 as the runtime.
* Used the "existing role" option to connect the function to the recently created IAM role with the necessary permissions.

1. **Configuration**

After the Lambda function was created, we updated its basic settings under the Configuration tab through General configuration, setting the memory size to 500 MB and timeout to 1 minute.

1. **Lambda Function Code Logic Overview**

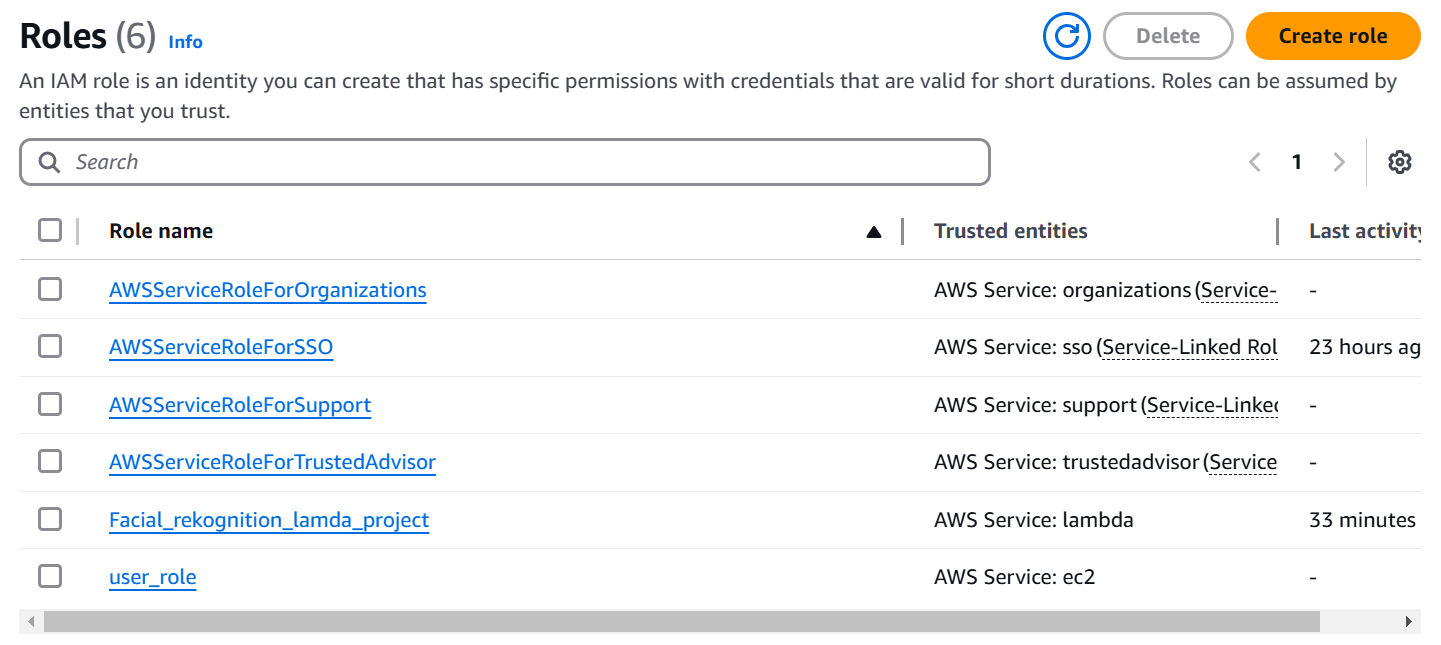
The authentication function interacts with the frontend for verifying visitors or employees. The following is how it works:

* The function receives a request from the frontend containing the image key of the visitor.
* It retrieves the image in binary format from the S3 bucket (project-visitor-image-storage) to ensure compatibility with Rekognition.
* Using Rekognition's SearchFacesByImage API, it matches the uploaded image with the indexed faces in the collection (employees).
* If a match is found, it retrieves the corresponding rekognition\_id from DynamoDB and fetches the employee’s details.
* It returns a response with the employee's first name and last name.
* If no match is found, “Not an Employee” response is returned to the client.
  + 1. **IAM Role Creation for API Gateway**

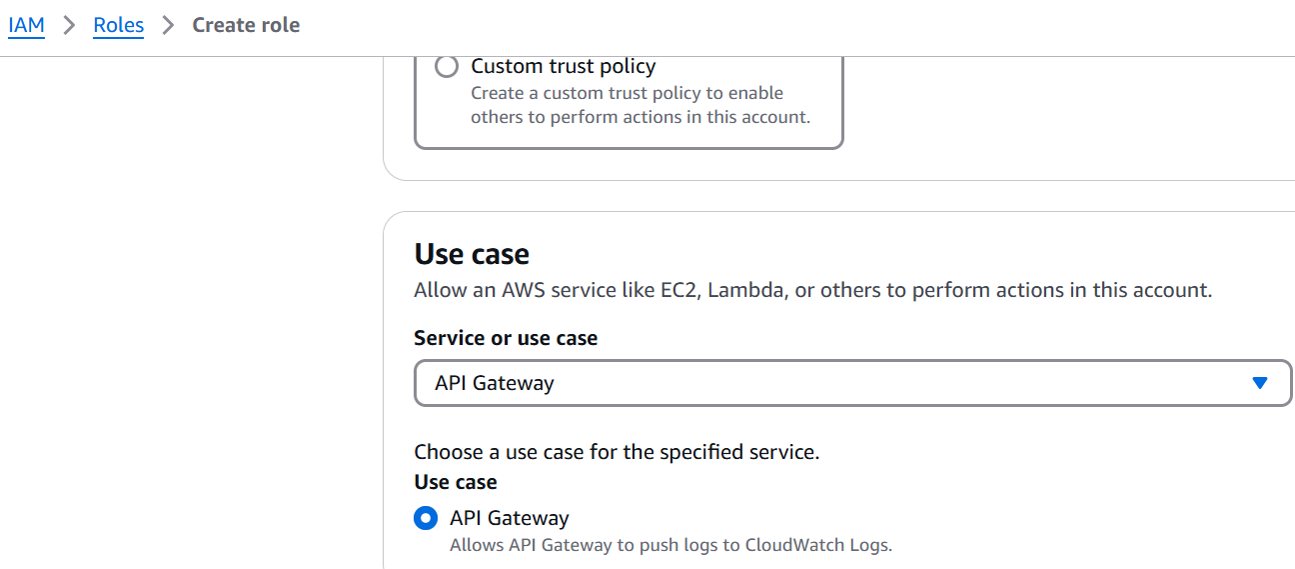
Before creating the API Gateway, an IAM Role was required to provide the necessary permissions for the Gateway to interact with the S3 bucket. The following steps were performed:

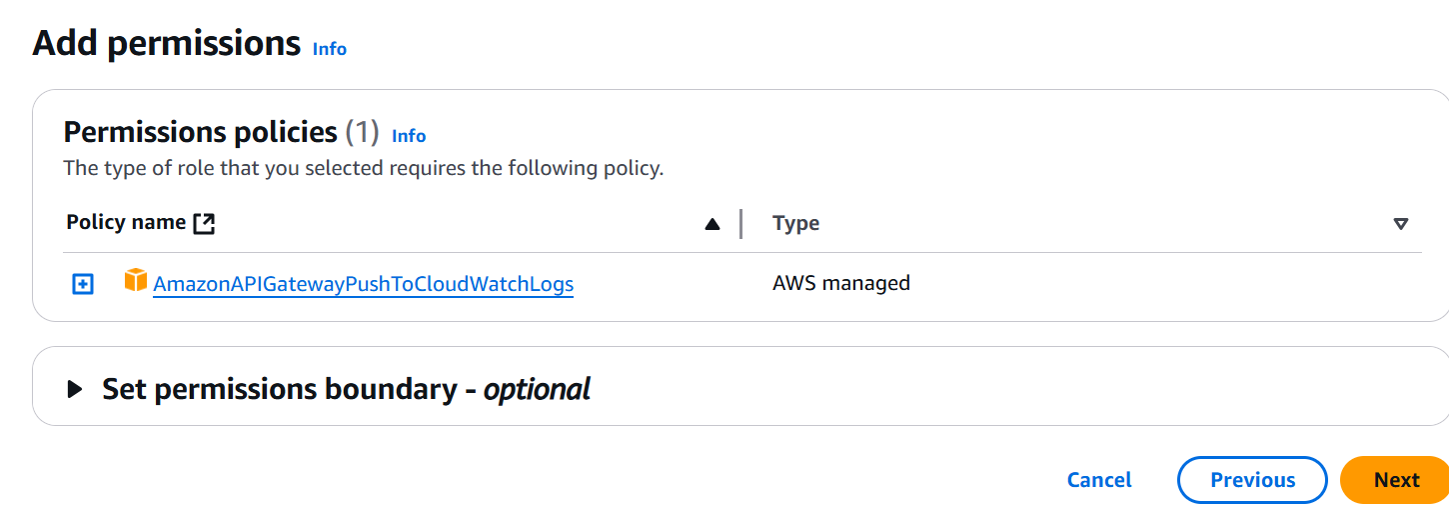
1. **Role Creation**

* We navigated to the IAM Console and created a new role for the API Gateway service.

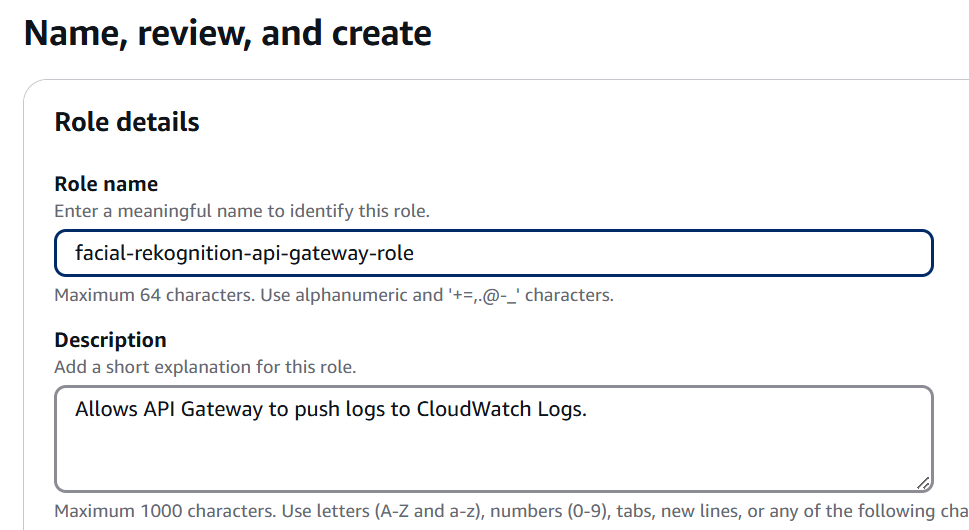


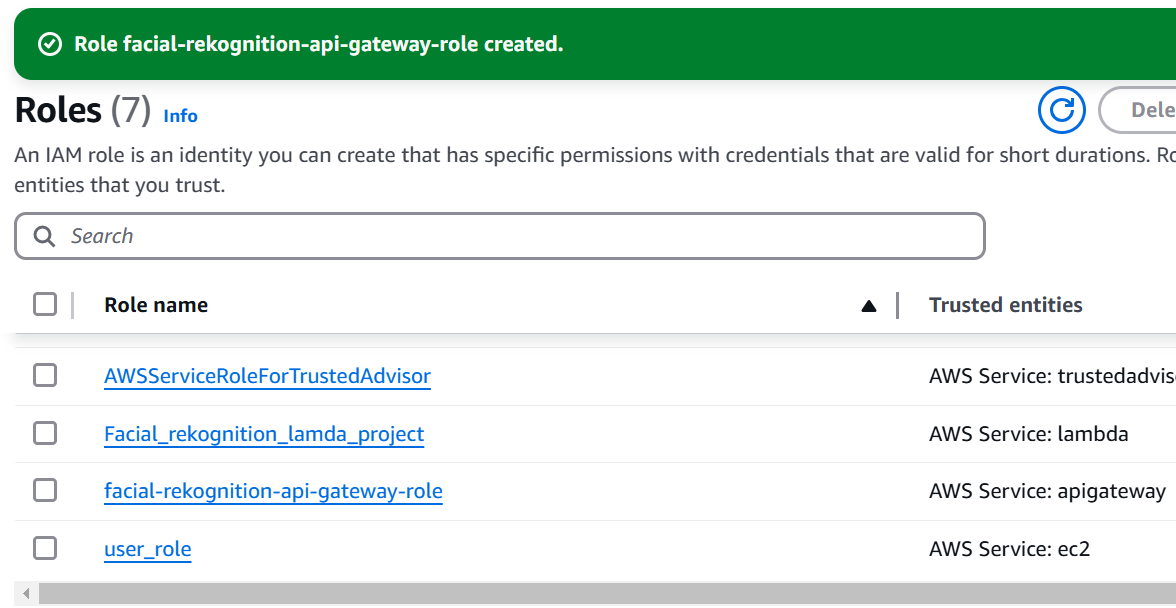
* API Gateway was chosen as the Use case and furthermore,AWS automatically attached the CloudWatch Logs permissions by default.





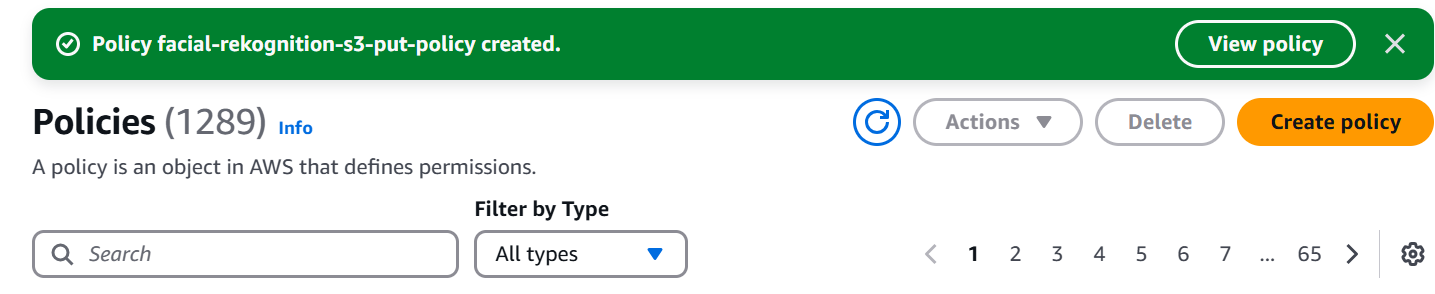
* The role was named facial-recognition-api-gateway-role.



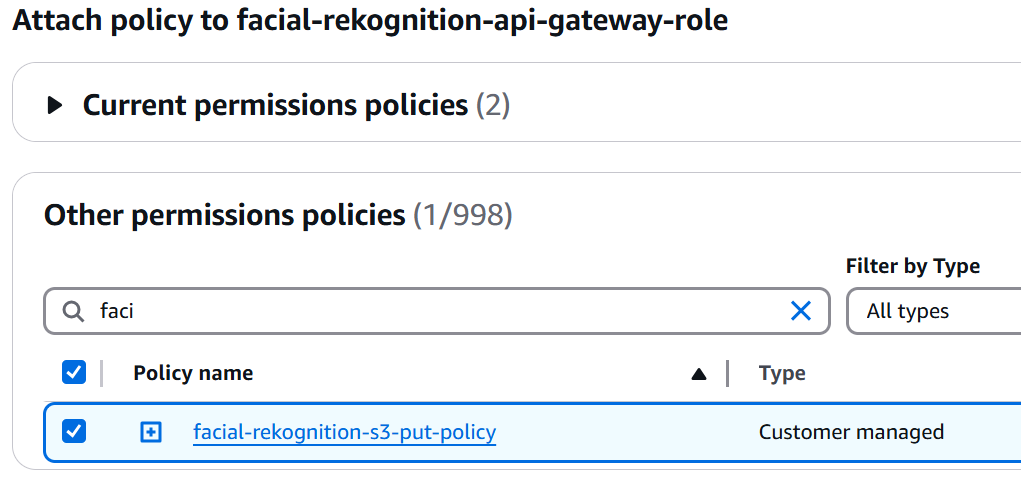


1. **Policy Attachment**

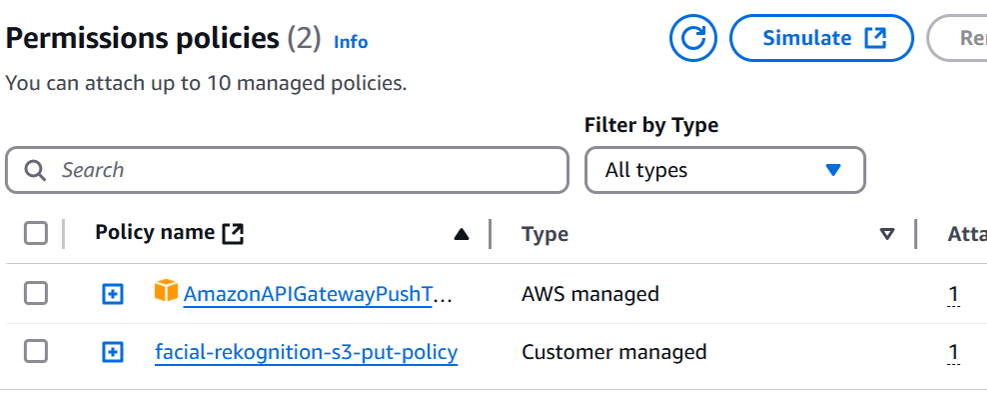
* Created a custom policy to grant PutObject permissions for the project-visitor-image-storage S3 bucket.



* Configured the policy to allow access only to this specific bucket for security.
* The policy was named facial-recognition-s3-put-policy and attached to the IAM role.



* Verified that the role now had two attached policies: CloudWatch Logs and the newly created S3 PutObject policy.

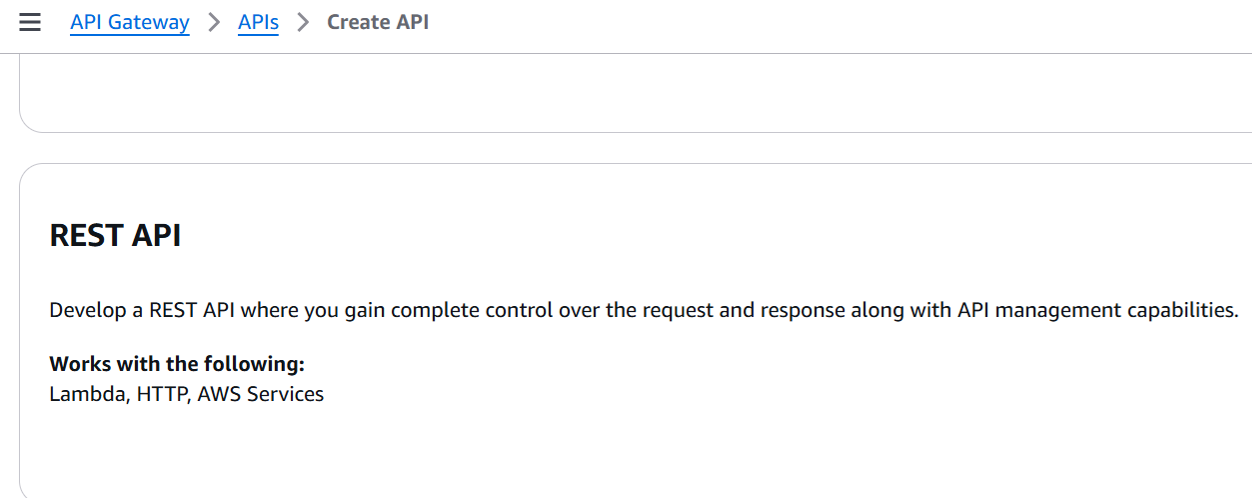


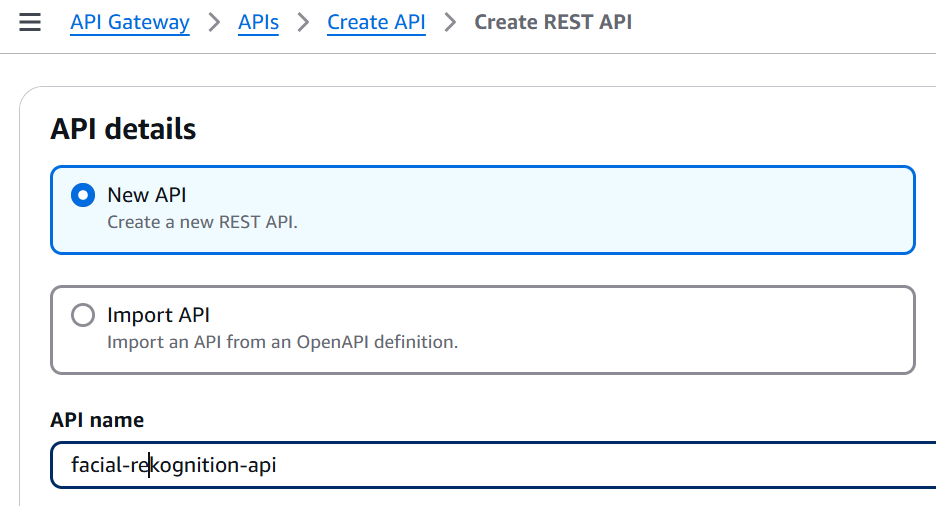
* + 1. **API Gateway**

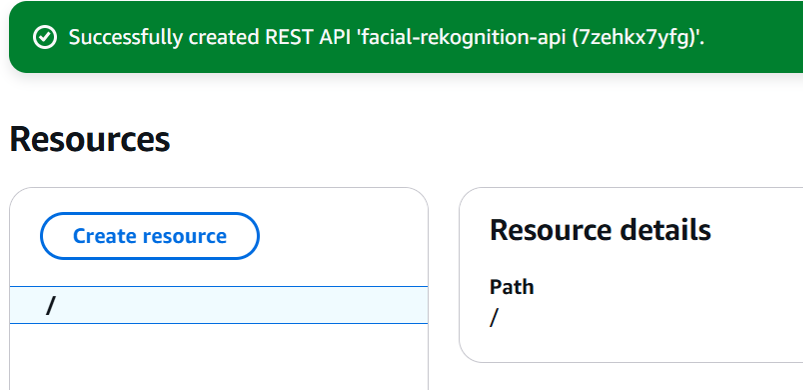
The API Gateway was designed to enable two key operations: uploading images to the S3 bucket and interfacing with the Lambda function for employee authentication.

1. **API Creation**

The REST API option was selected from the API Gateway console to set up the required interface. A new API was created and named "facial-recognition-api", with all default settings retained to streamline the process.

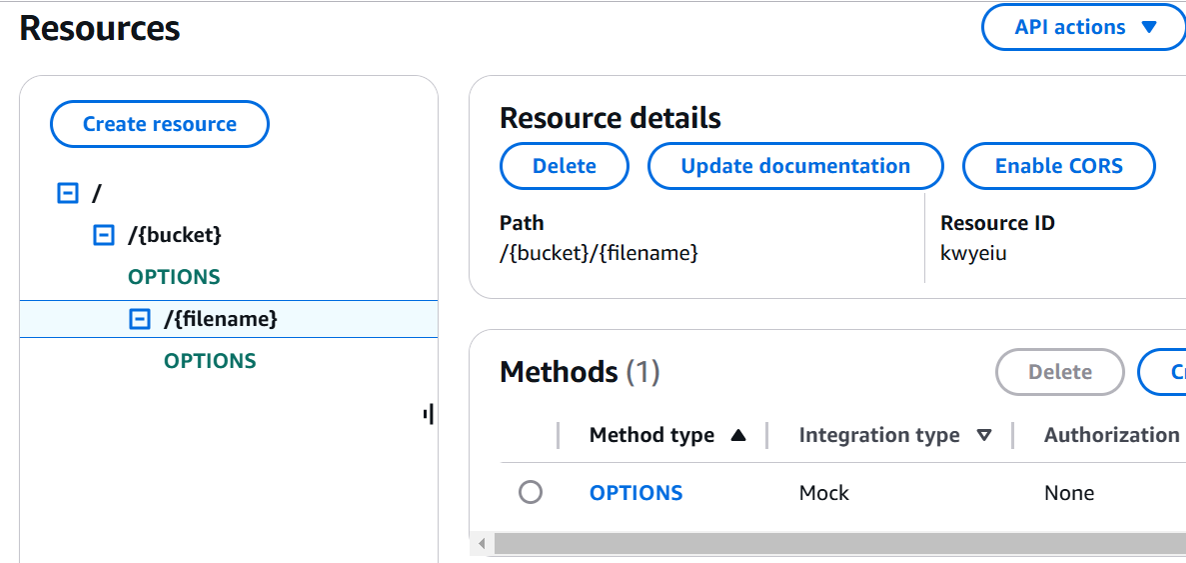




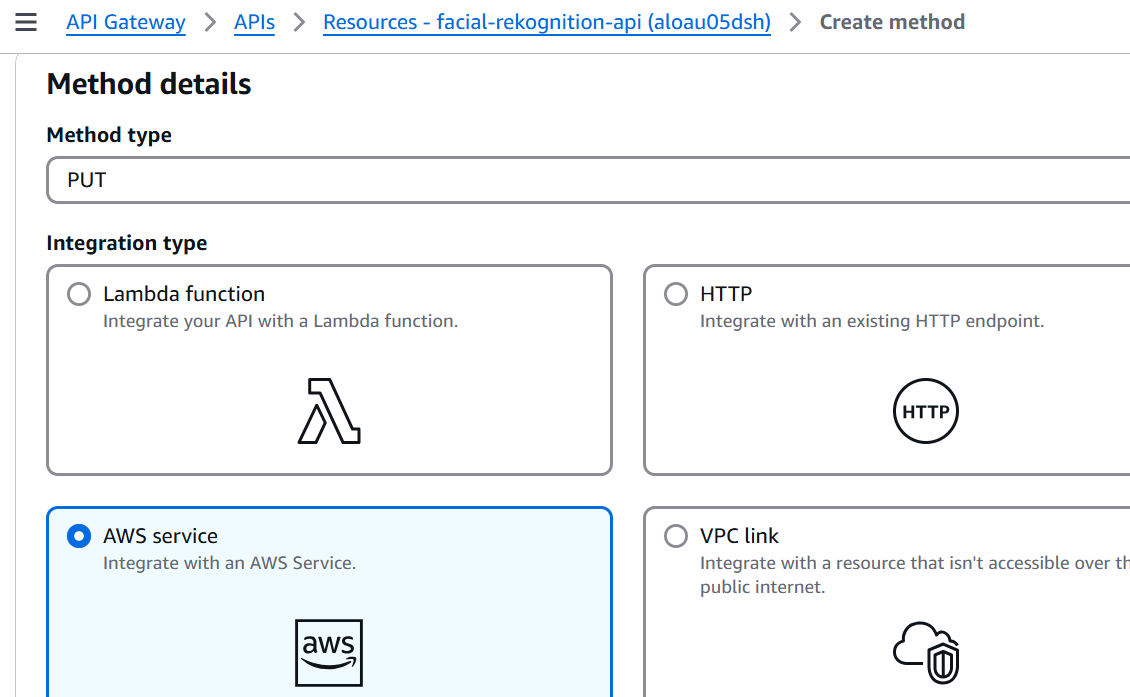


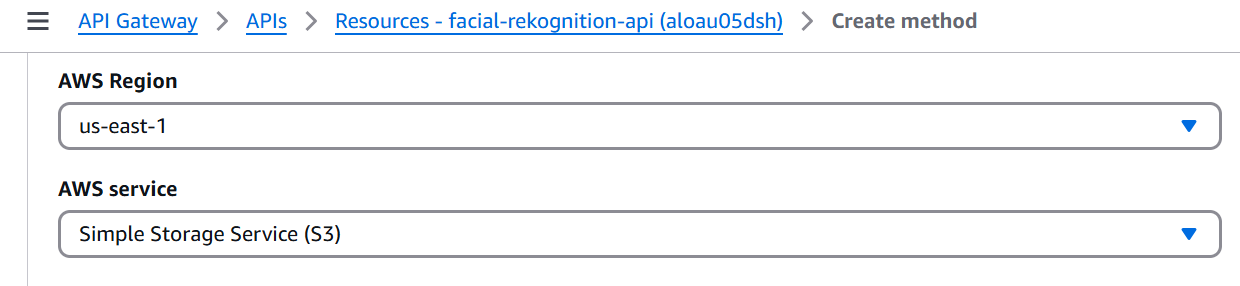
1. **EndPoint for Image Upload**

* Created a resource named bucket/{filename}, ensuring that Enable API Gateway CORS was checked to handle cross-origin requests effectively.

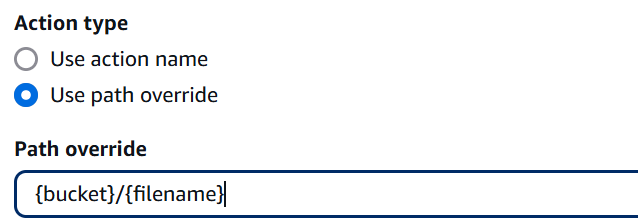


* Added a PUT method to the created resource to enable uploading of images to the S3 bucket and configured the method to use the AWS Service integration type, selecting S3 as the service for image uploads.

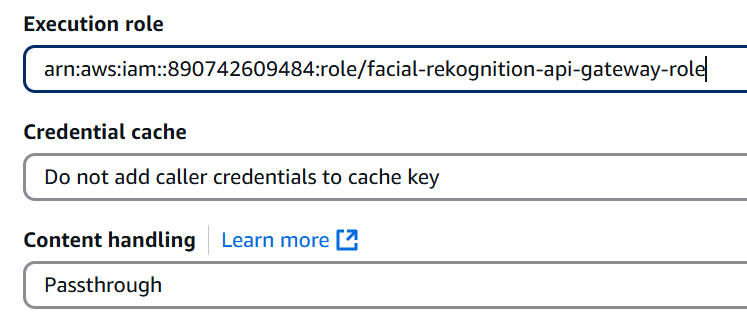


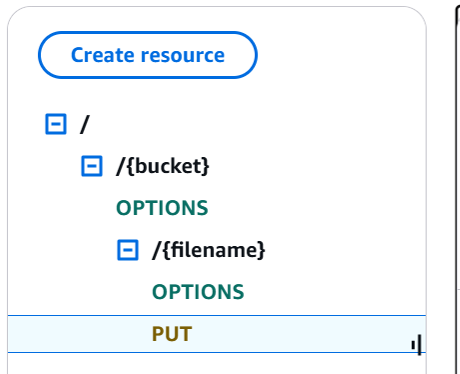


* Set up a path override to match the bucket and file name structure as bucket/{filename}, ensuring the correct mapping of uploaded files.

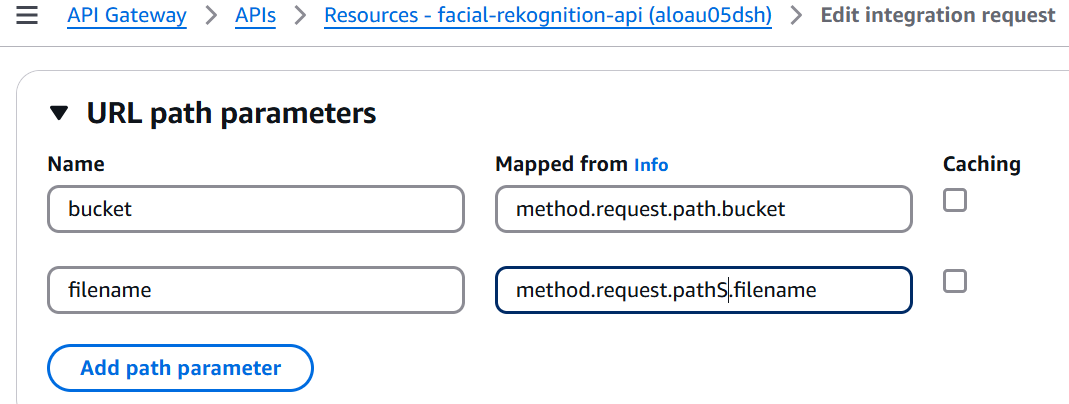


* Specified the IAM role created earlier as the execution role to provide secure access to the S3 bucket.

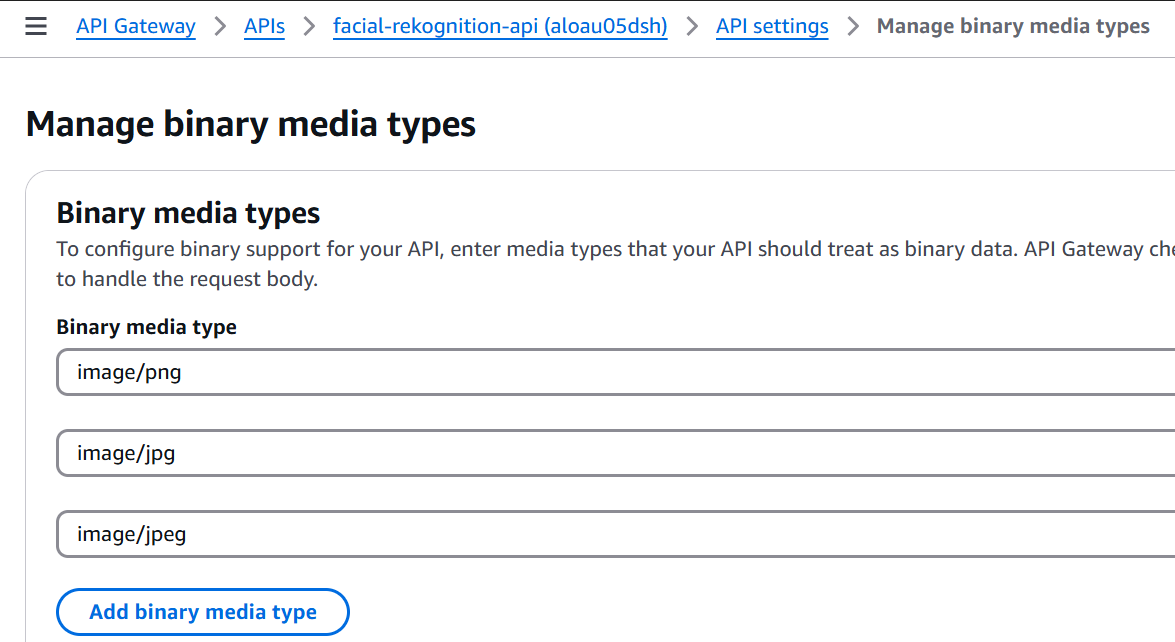




* Added path parameters for bucket and filename to ensure the API correctly interprets and processes the request parameters.

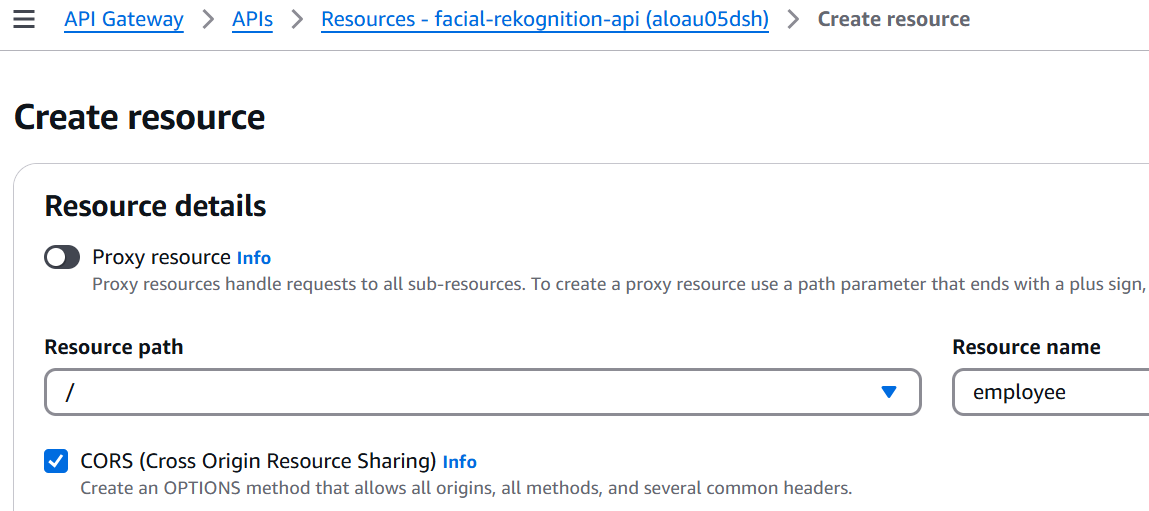


* Enabled binary support for image/jpeg and image/png formats by updating the configuration under Settings ,Binary Types, allowing these image formats to be processed seamlessly.

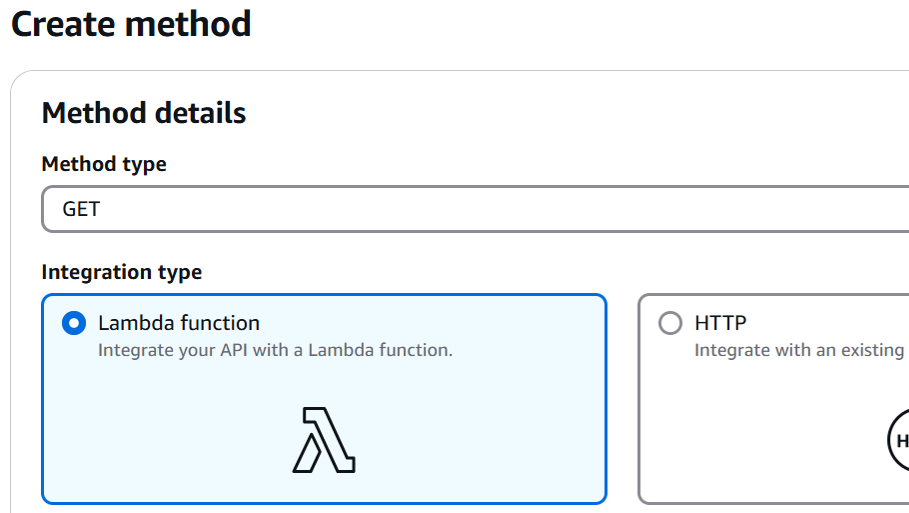


1. **Endpoint for employee Authentication**

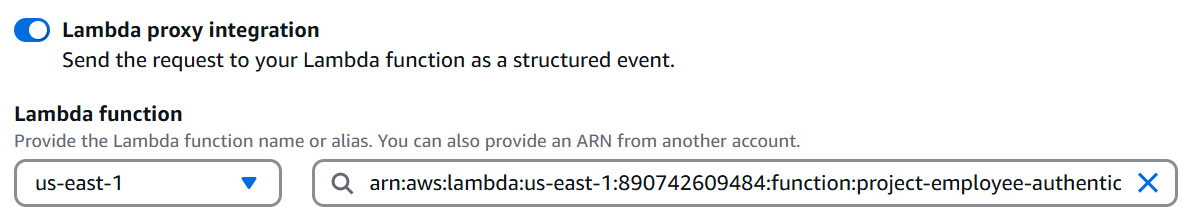
* Created another resource named employee to serve as the endpoint for employee authentication.



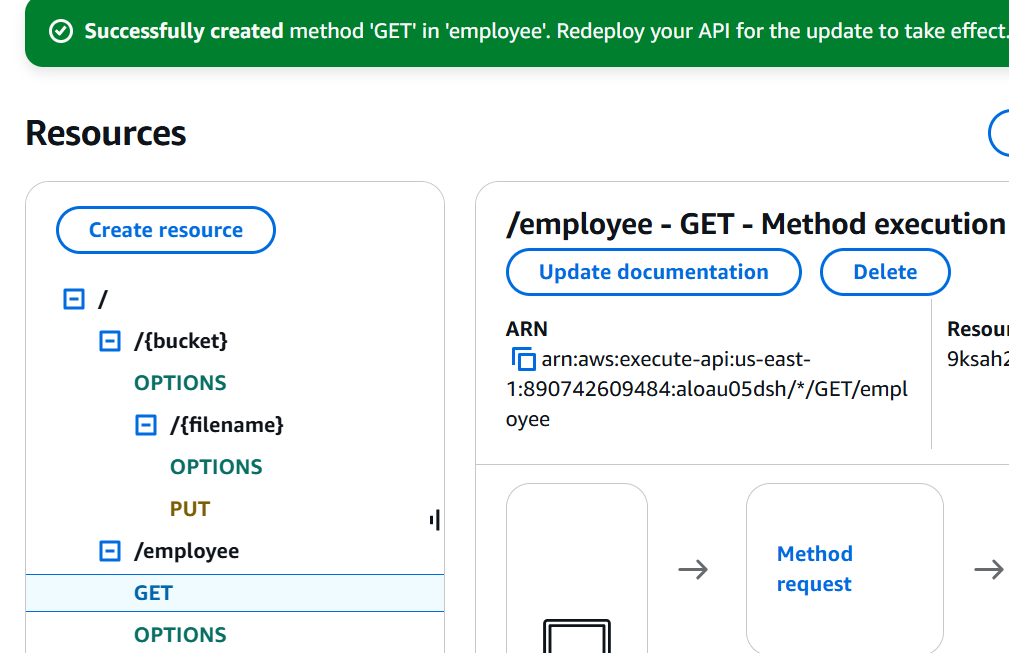
* Added a GET method to this resource, allowing it to retrieve employee authentication data. We also configured the method to use the Lambda Function integration type, connecting it to the employee-authentication Lambda function for processing authentication requests.



* Ensured that Lambda Proxy Integration was enabled to facilitate seamless integration between the API Gateway and the Lambda function.

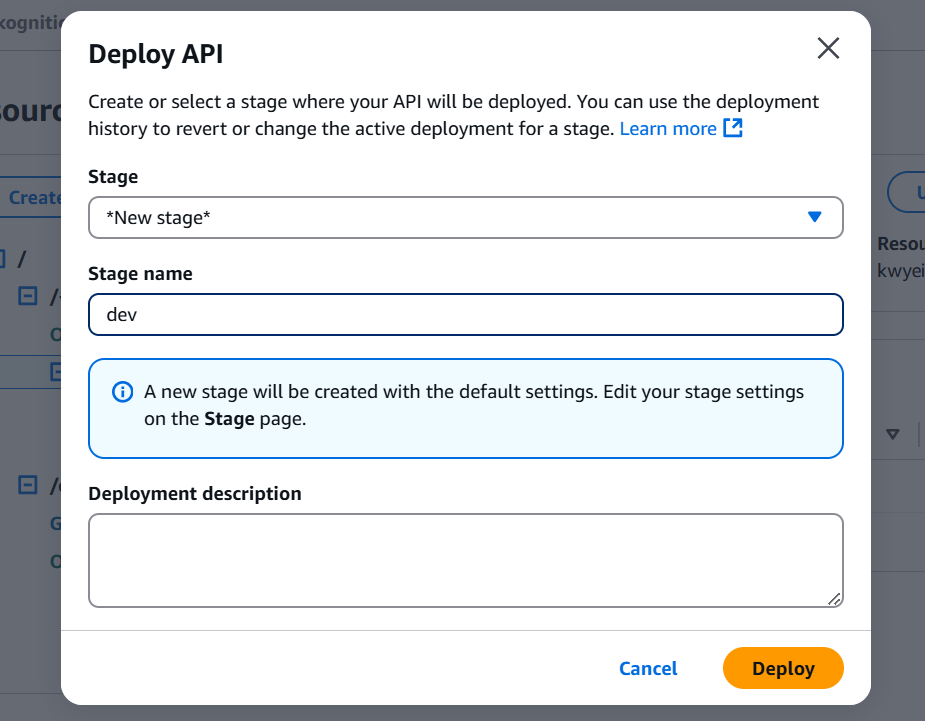


* We also Enabled CORS for this resource to handle cross-origin requests effectively, ensuring compatibility with frontend applications.
* Finally the API was deployed to a new stage for production use, making the endpoint ready for integration with the frontend.

****

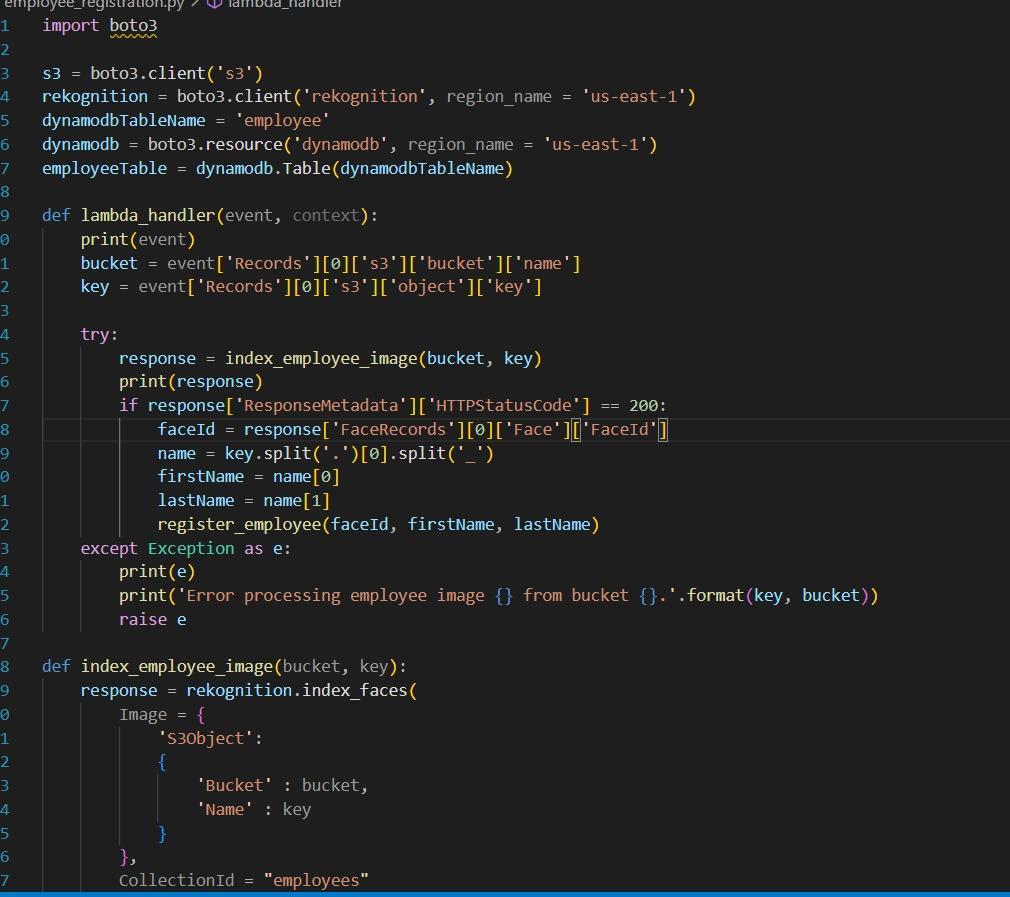
1. **Final API Gateway Deployment**

The API Gateway now includes two endpoints: a PUT endpoint for uploading images to the S3 bucket and a GET endpoint for interacting with the Lambda function to authenticate employees. Both endpoints were successfully deployed under the stage name dev, making them accessible for integration with the frontend application.



**3.2 Sample Code**

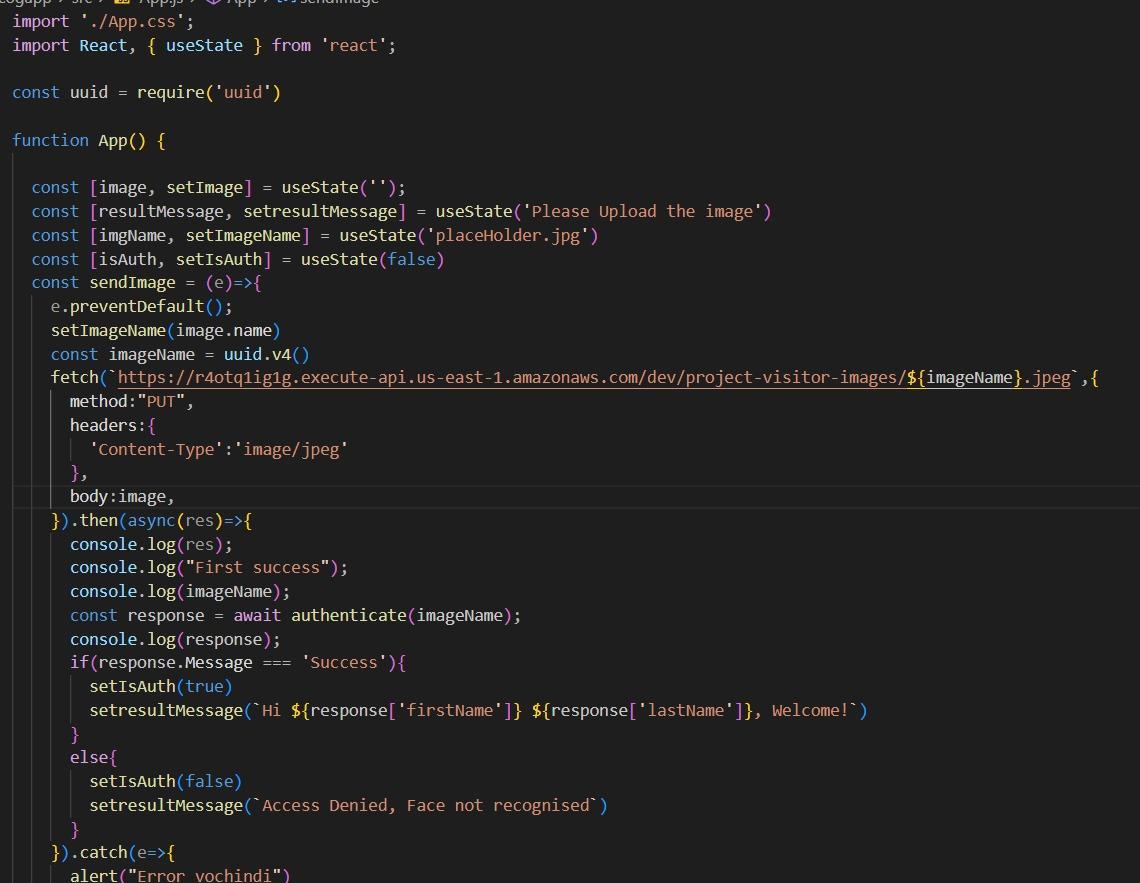
**3.2 1 Project-Employee-Registration Lambda Function**



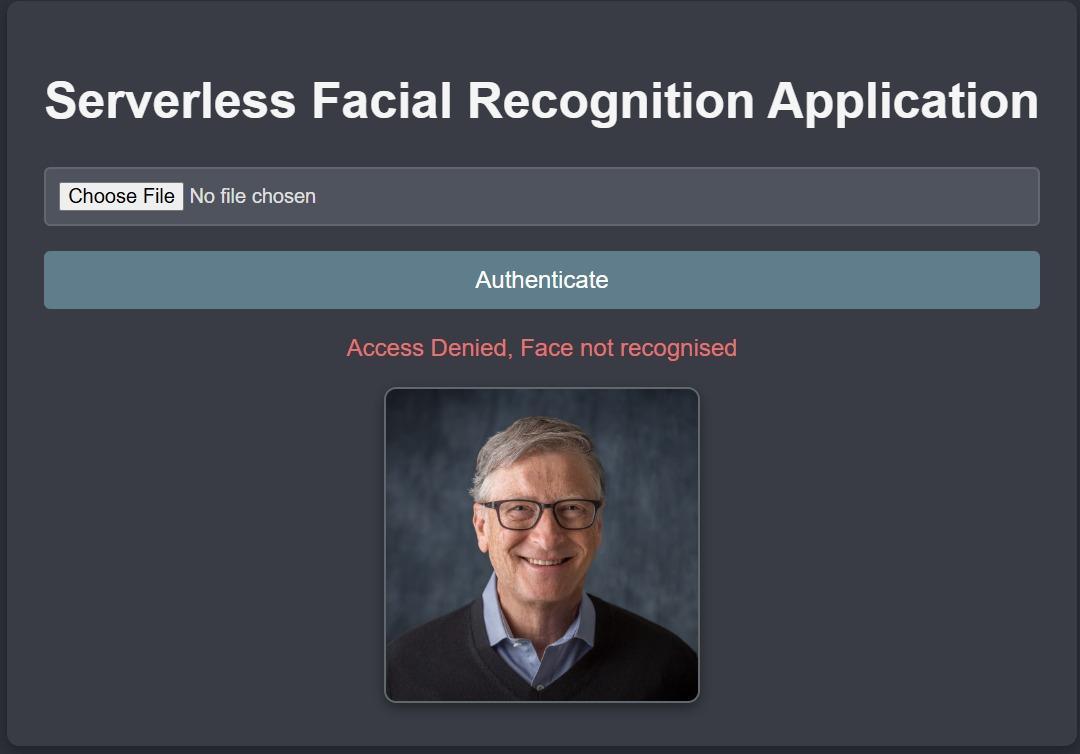
* + 1. **Project-Employee-Authentication Lambda Function**

****

* + 1. **Front End**

****

* 1. **Results**
     1. **Failure Case**

****

* + 1. **Success Case**

****

1. **CONCLUSION**

In conclusion, this project successfully implemented a serverless facial recognition system using AWS services to streamline employee registration and authentication processes. By integrating S3, Rekognition, DynamoDB, Lambda, and API Gateway, the system enables secure image indexing, efficient employee identification, and seamless API interactions. This scalable and cost-effective solution showcases the power of serverless architecture in building robust applications, laying a strong foundation for further enhancements, such as multi-factor authentication or real-time analytics, to meet evolving business needs.

1. **REFERENCES**
2. <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>
3. <https://docs.aws.amazon.com/rekognition/>
4. <https://docs.aws.amazon.com/s3/>
5. <https://docs.aws.amazon.com/apigateway/>
6. <https://react.dev/reference/react>
7. <https://aws.amazon.com/blogs/machine-learning/easily-perform-facial-analysis-on-live-feeds-by-creating-a-serverless-video-analytics-environment-with-amazon-rekognition-video-and-amazon-kinesis-video-streams/>