System Design Document: Invoice OCR Processing Solution

Objective

Design and implement a solution that allows users to upload an invoice image to a webpage, automatically extract text from the image using OCR (Optical Character Recognition), and display a structured summary of the extracted data to the user.

Requirements

1. **Frontend**: A webpage for image upload and extracted data visualization.
2. **Backend**: Services to handle image uploads, OCR processing, and database management.
3. **Cloud Resources**: AWS services for hosting, processing, and storage.
4. **Database Modeling**: PostgreSQL with Prisma ORM.
5. **Authentication and Permissions**: User authentication with GAuth.

Architecture

1. Frontend

- **Framework**: Next.js
- **Functionality**:
    - User authentication;
    - Image upload functionality;
    - Invoices uploaded list;
    - Invoice detail with summary of extracted data.

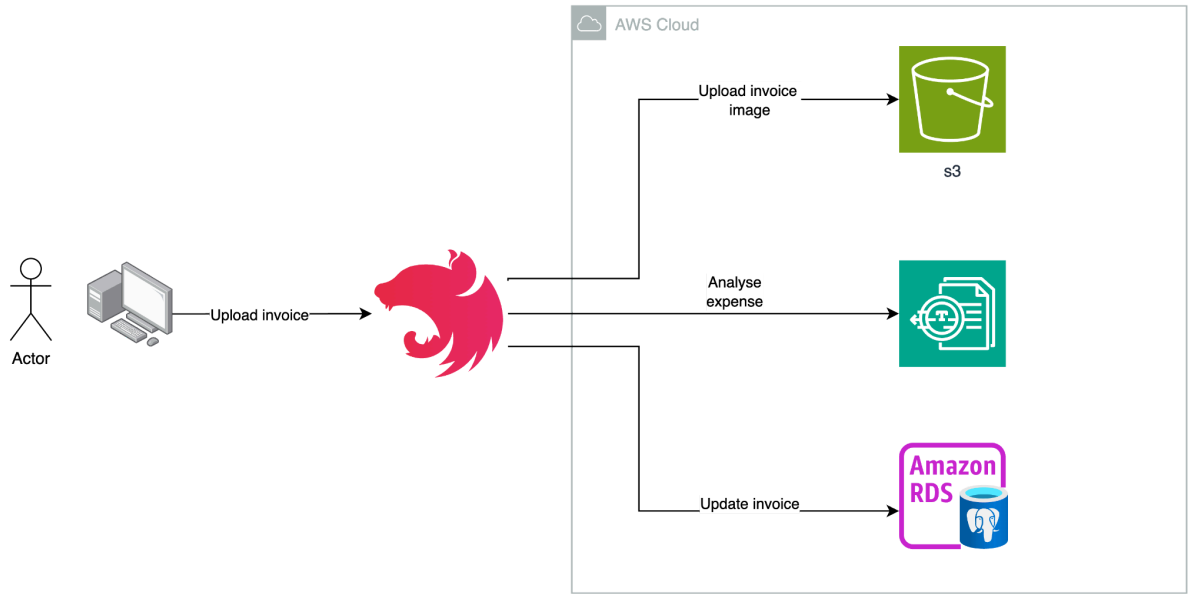
2. Backend

- **Framework**: NestJS
- **Functionality**:
    - User authentication using GAuth;
    - Image upload handling;
    - OCR processing;
    - Storing and retrieving OCR results from the database.

3. Cloud Resources (AWS)

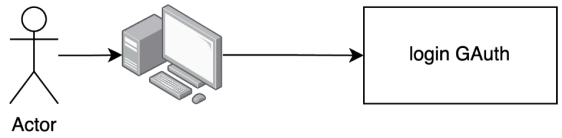
- **S3**: Storage for uploaded images.
- **Textract**: AWS service for OCR.
- **RDS**: PostgreSQL database for storing extracted data.
- **Lambda**: Serverless function to trigger OCR processing.
- **API Gateway**: Routing API requests to the appropriate services.

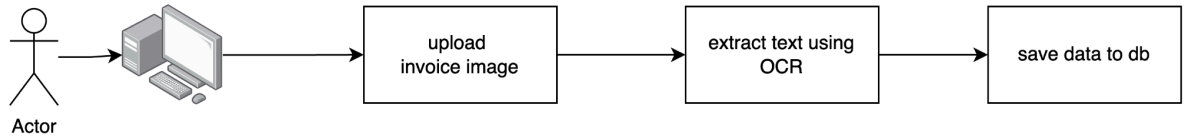
Architecture Diagram

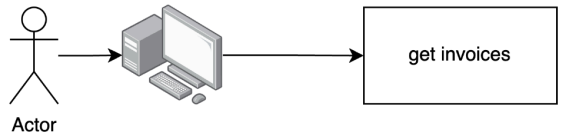


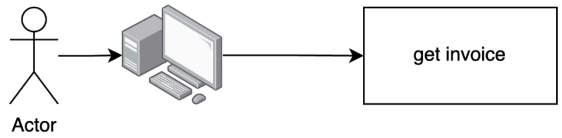
Component Interactions

1. **User Interaction**:
    - User login in via GAuth.
    - User uploads invoice image.
    - User views invoice screen.
    - User views the details of an invoice.
2. **Frontend**:
    - Send the image to the backend.
    - Retrieve invoice data.
3. **Backend**:
    - Stores the image in S3.
    - Sends the image to Textract for OCR processing.
    - Receives structured data from Textract.
    - Stores the structured data on PostgreSQL.
4. **User Access**:
    - Authenticated users can view their processed invoice data.









AWS Resources

- **Amazon S3**: To store uploaded images.
- **Amazon Textract**: For OCR processing.

Modeling

Entities

1. **User**:
    - `id`: UUID
    - `email`: String
    - `name`: String
    - `invoice`: Invoice[ ]
    - `created_at`: DateTime
    - `updated_at`: DateTime
    - `deleted_at`: DateTime?

2. **Invoice**:
    - `id`: UUID
    - `image_url`: String
    - `file_name`: String
    - `extract_raw`: Json?
    - `status`: String
    - `created_at`: DateTime
    - `updated_at`: DateTime
    - `deleted_at`: DateTime?
    - `userId`: UUID (Foreign Key)

Frontend

Pages

1. **Login Page**: For GAuth login.
2. **Upload Page**: To upload invoice images.
3. **Invoices Page**: To display a list of invoices uploaded.
4. **Detail Page:** To display extracted invoice data.

Backend

Controllers

1. **AuthController**: To handle authentication with GAuth.
2. **InvoiceController**: To manage invoice uploads and retrieval.

Services

1. **AuthService**: To manage authentication logic.
2. **InvoicesService**: To handle invoice-related operations.
3. **UsersService**: To handle invoice-related operations.

Authentication and Permissions

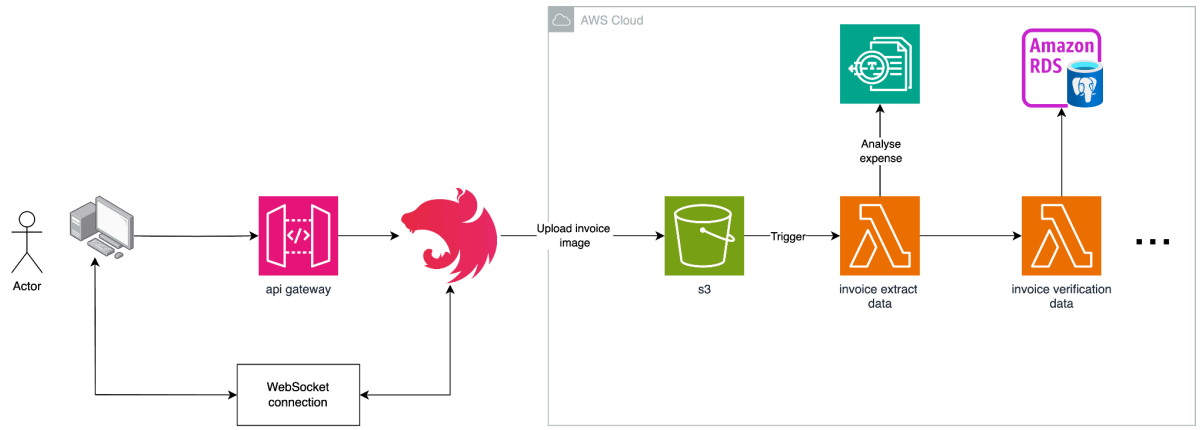
- **Authentication**: Using GAuth.
- **Authorization**: Only the user who uploaded an invoice can access the extracted data.

Conclusion

This document outlines the system design for an invoice OCR processing solution created. It includes the architecture, database modeling, and key components required to implement

the solution using Next.js, NestJS, and AWS resources. The system ensures secure user authentication and provides a structured summary of the extracted data to the user.

Not all planned structures were possible to be created, the final decision of the planned architecture was as per the image below with more independence in extracting invoice data from the NestJs application, making it more robust with the execution of the lambda as a trigger after the completion of the upload process, using lambda to process the image would also bring greater scalability and facilitate the creation of new chains of events at the end of each step.



The deployment in the AWS environment was also not carried out, the initial deployment method would be done through the integration of Github actions for the code deployment in AWS. Nest would be deployed in a dockerized manner in the ECR repository and made available by ECS.