Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

Class Based Project Report
BCSE355L- Cloud Architecture Design
Faculty: Dr. Mohanraj Gopal

Review- Guidelines (100 Marks) Date: 07 NOV 2025

Cloud-Powered Smart Notes Repository

Project By: Vedansh Gupta (23BIT0236) Kushagra Mukhija (23BIT0424)

1.0 Problem Identification

In today's academic and professional environments, students and researchers accumulate a massive volume of notes in various digital formats, including PDFs, lecture slides, and images (e.g., photos of a whiteboard). The primary challenge is that this data is "unstructured" and "dumb." It is impossible to search for keywords or concepts *inside* these documents without opening them one by one.

For example, finding a single topic like "Amazon SQS" or a specific registration number like "23BIT0424" across hundreds of files is a manual, time-consuming, and inefficient process.

Our project, the **Cloud-Powered Smart Notes Repository**, solves this problem. We have built an intelligent cloud-native application that ingests any PDF or image, uses Optical Character Recognition (OCR) to "read" its contents, and stores this text in a high-speed, searchable database. This makes every document in the repository instantly and fully searchable through a simple, modern web interface.

2.0 AWS Architecture and Services

This is the core of your report. You will show your architecture first, then explain the services.

2.1 Workflow Diagram

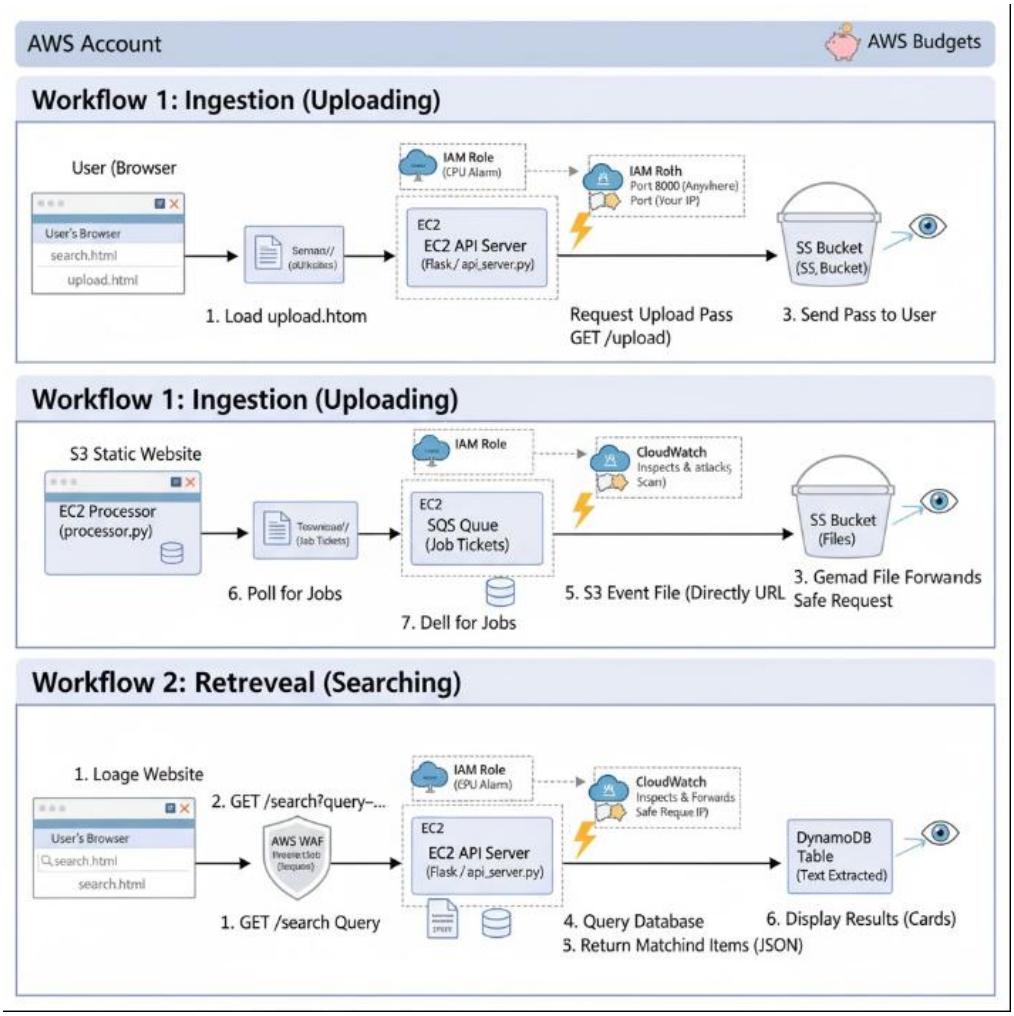


Fig 1: The decoupled, event-driven architecture of the Smart Notes Repository.

Our application is built on a decoupled, event-driven architecture, ensuring it is scalable, fault-tolerant, and secure. The workflow is split into two independent processes:

1. **The Ingestion (Upload) Workflow:** A user uploads a file from our S3-hosted website. The browser gets a secure, temporary presigned URL from our API to upload the file directly to an S3 bucket. This PUT event in S3 automatically sends a 'job message' to an **SQS queue**. Our **EC2 'Processor' script** (worker) is polling this queue, receives the message, downloads the file from S3, uses OCR to extract the text, and writes that text to the **DynamoDB** table.
2. **The Retrieval (Search) Workflow:** A user visits our website (hosted on **S3 Static Website Hosting**). A JavaScript fetch request is sent to our API. This request is first inspected by **AWS WAF** (Web Application Firewall) to block attacks. The safe request hits our **EC2 'API Server' script** (a Flask app), which queries **DynamoDB** for the text. The results are returned as JSON and displayed on the webpage.

2.2 Table of AWS Services Used

Service	Service Name	Purpose in Project (Brief Summary)	Syllabus Module
S3	Simple Storage Service	1. Stores original PDF/image files. 2. Hosts the public-facing static HTML/JS/CSS website.	2
EC2	Elastic Compute Cloud	A t3.micro instance running our two Python backend scripts: the Flask api_server.py and the processor.py worker.	2
SQS	Simple Queue Service	The "to-do list" that decouples the upload from the processing. This makes the system scalable and fault-tolerant.	7
DynamoDB	DynamoDB	A NoSQL database that acts as our "search index." We store the extracted text here for high-speed queries.	4
IAM	Identity & Access Mngmt.	Used to create an EC2 Role (EC2-Notes-Processor-Role) so our server can securely access other services without secret keys.	1
WAF	Web Application Firewall	Secures our public-facing API by inspecting requests and blocking common web attacks (e.g., SQL injection, XSS).	6
CloudWatch	CloudWatch	Used to create an alarm that monitors our EC2 instance's CPU Utilization, alerting us if the server is overworked.	5
Budgets	AWS Budgets	Used to set a $10 monthly budget with a *forecasted* alert to prevent any accidental or surprise costs.	5

2.3 Detailed Service Explanation

- **Amazon S3 (Simple Storage Service):** "S3 is the foundation of our storage. We use it to host our search.html, upload.html, and dashboard.html files as a public static website. We also use it as the primary storage location for all uploaded documents."

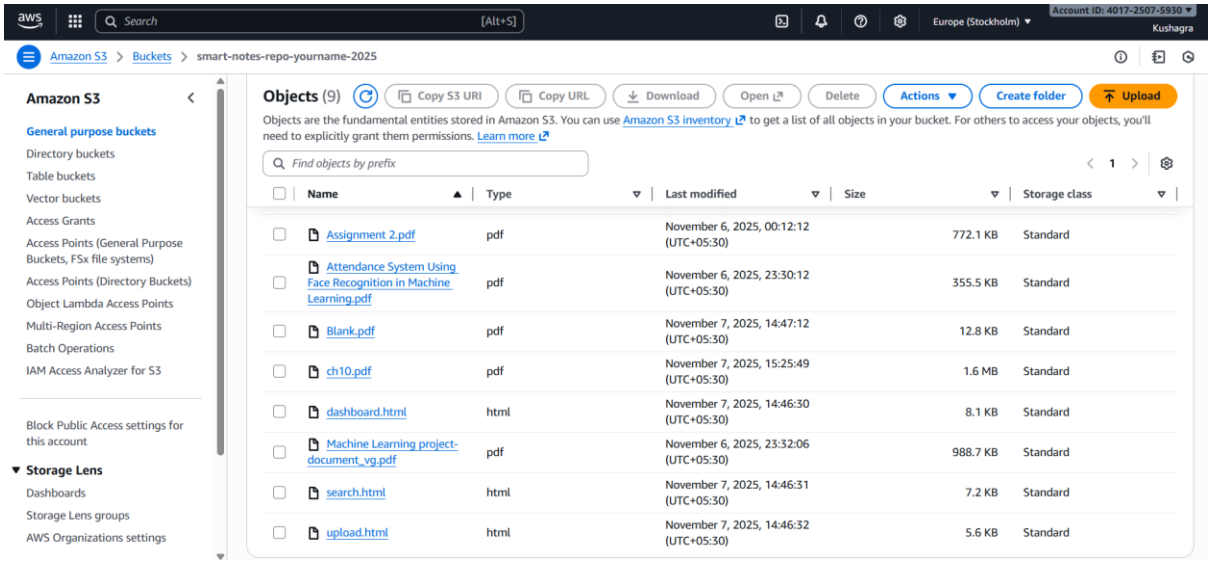


Fig 2: S3 bucket smart-notes-repo-yourname-2025 showing hosted HTML files and uploaded user documents.

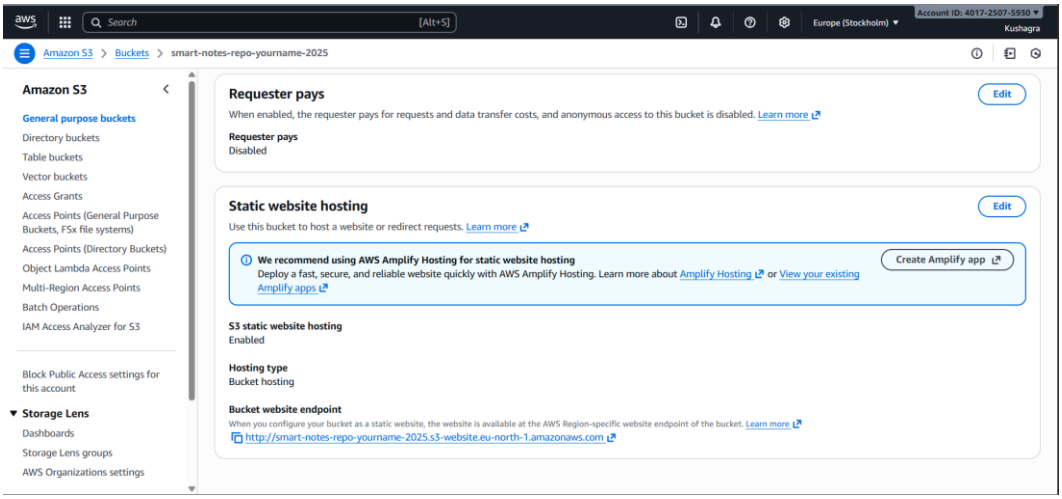


Fig 3: S3 Static Website Hosting enabled, with search.html as the index document.

- **Amazon EC2 (Elastic Compute Cloud):** "A t3.micro EC2 instance running Ubuntu is our application's 'compute brain.' It runs our two persistent Python scripts: the api_server.py and the processor.py worker."

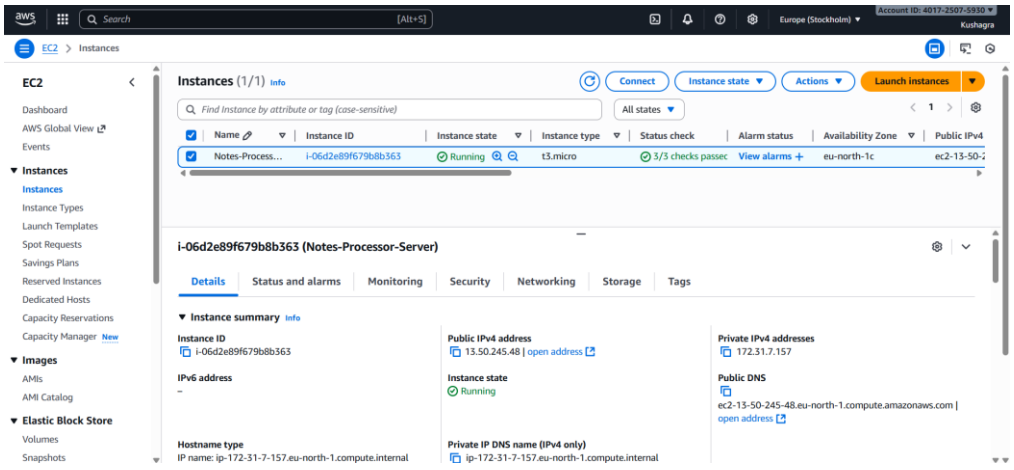


Fig 4: The Notes-Processor-Server (t3.micro) in the "Running" state in the eu-north-1 region.

- **Amazon SQS & DynamoDB (The Pipeline):** "SQS and DynamoDB work together. SQS provides the message queue for reliability, while DynamoDB provides the fast, indexed storage for our extracted text, making searches nearly instantaneous."

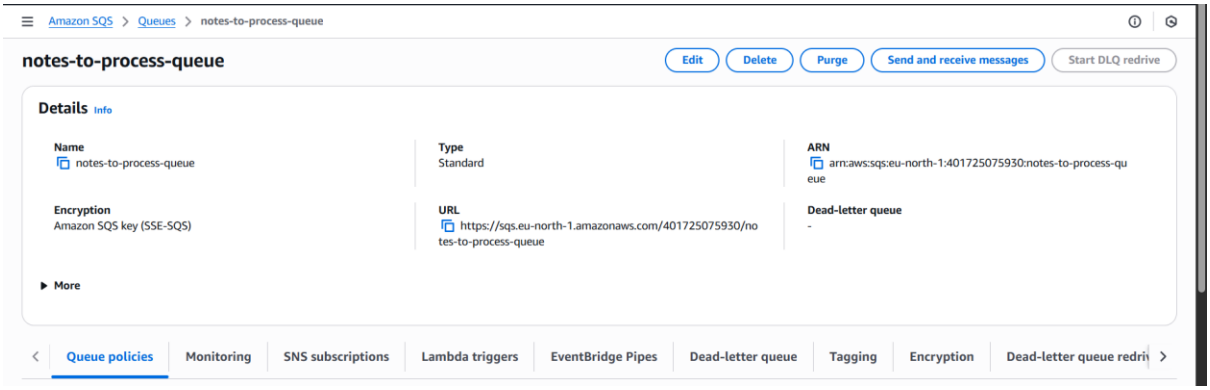


Fig 5: The notes-to-process-queue configuration details.

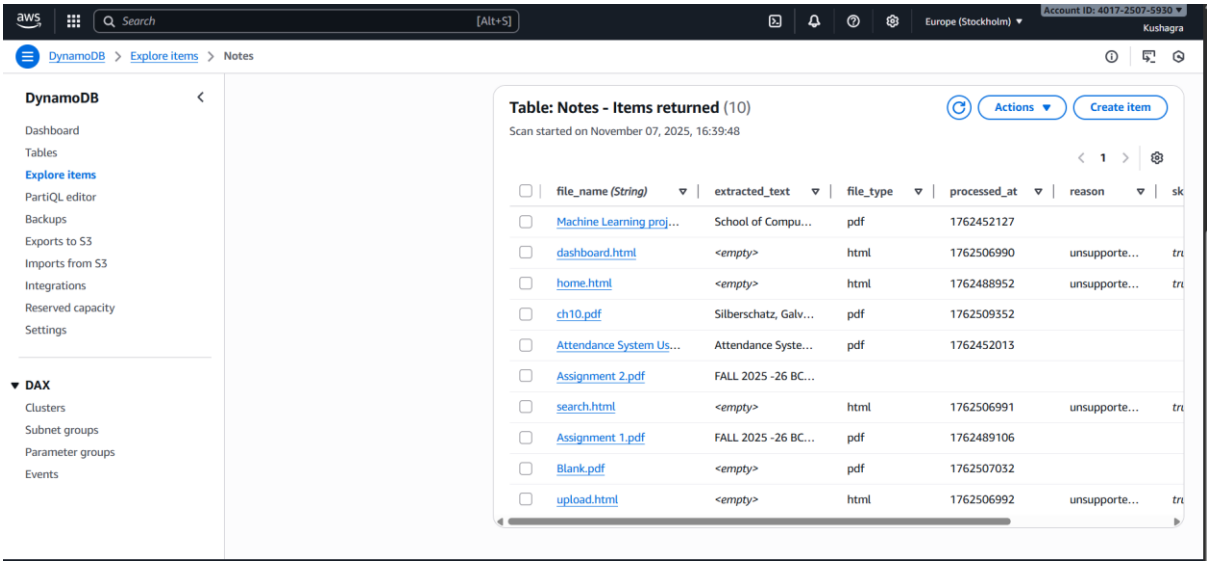


Fig 6: An item in the Notes table showing the file_name and its fully extracted text content.

- **IAM, WAF, & Security Groups (Security):** "We implemented a multi-layer security strategy. The **IAM Role** provides secure credentials to our EC2 instance. The **Security Group** acts as a basic firewall, and **AWS WAF** provides advanced, managed protection against web attacks."

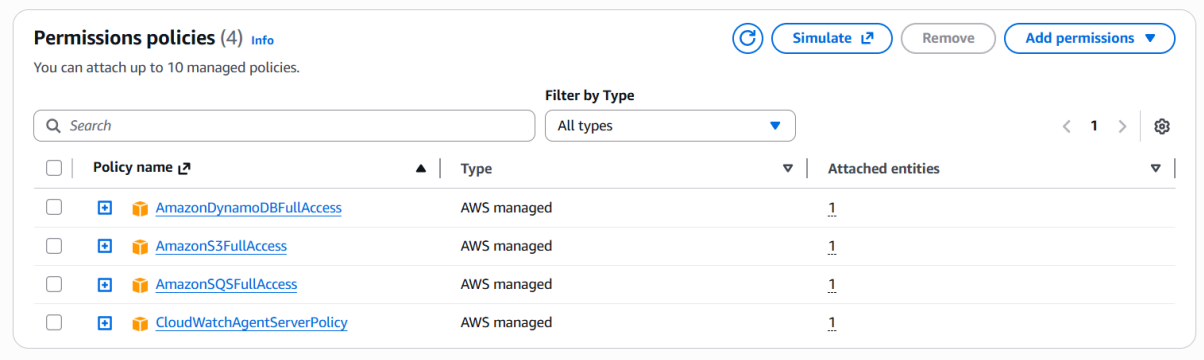
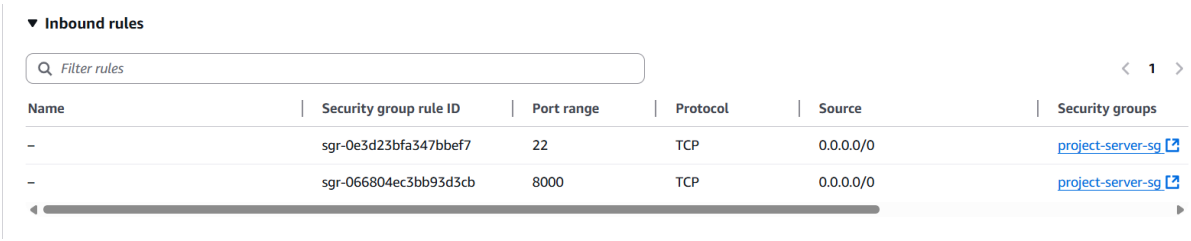


Fig 7: The EC2-Notes-Processor-Role with policies attached to allow access to S3, SQS, and DynamoDB.



Caption: *Fig 8: Inbound rules configured to allow SSH (Port 22) only from our personal IP and HTTP (Port 8000) from anywhere for the API.*

- **AWS Budgets (Cost Management):** "From the beginning, cost management was a priority. We set a $10 monthly budget with a *forecasted* alert. This ensures we are notified *before* we ever incur significant costs, which is a critical best practice.

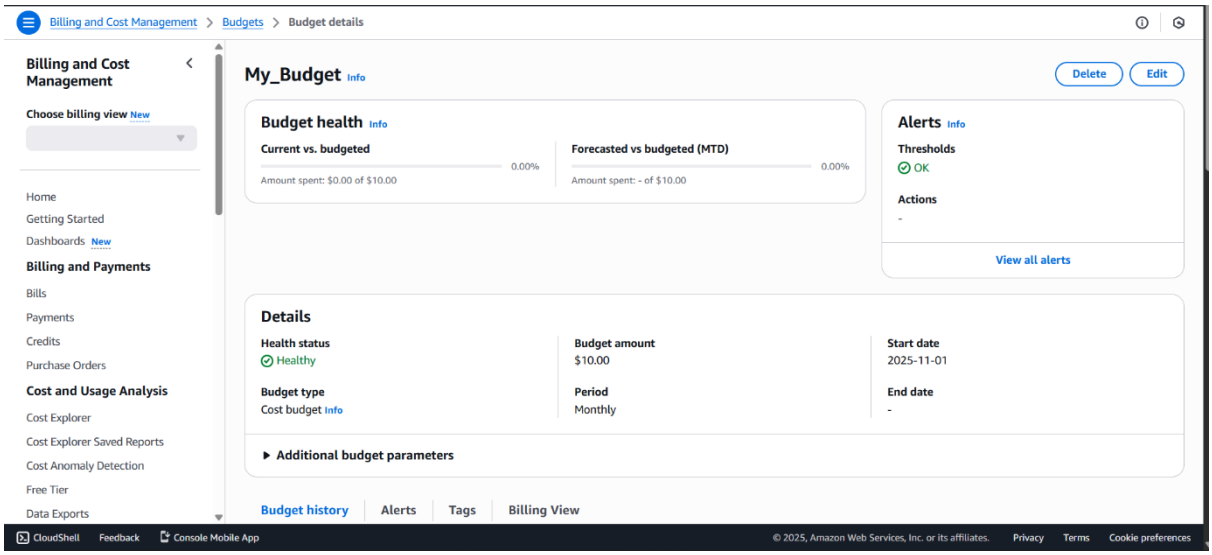


Fig 9: The $10 monthly cost budget with a forecasted alert threshold.

- **AWS CloudFront:** From the beginning, cost management was a priority. We set a 70% alarm on CPU utilization. This ensures we are notified *before* we ever incur significant CPU, which is a critical best practice.

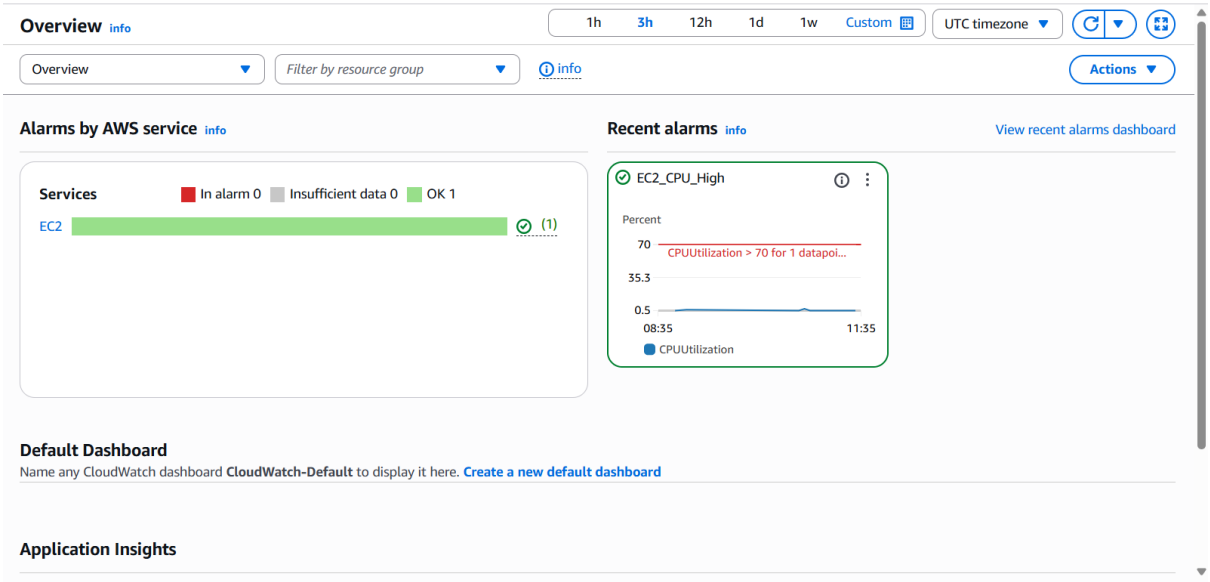


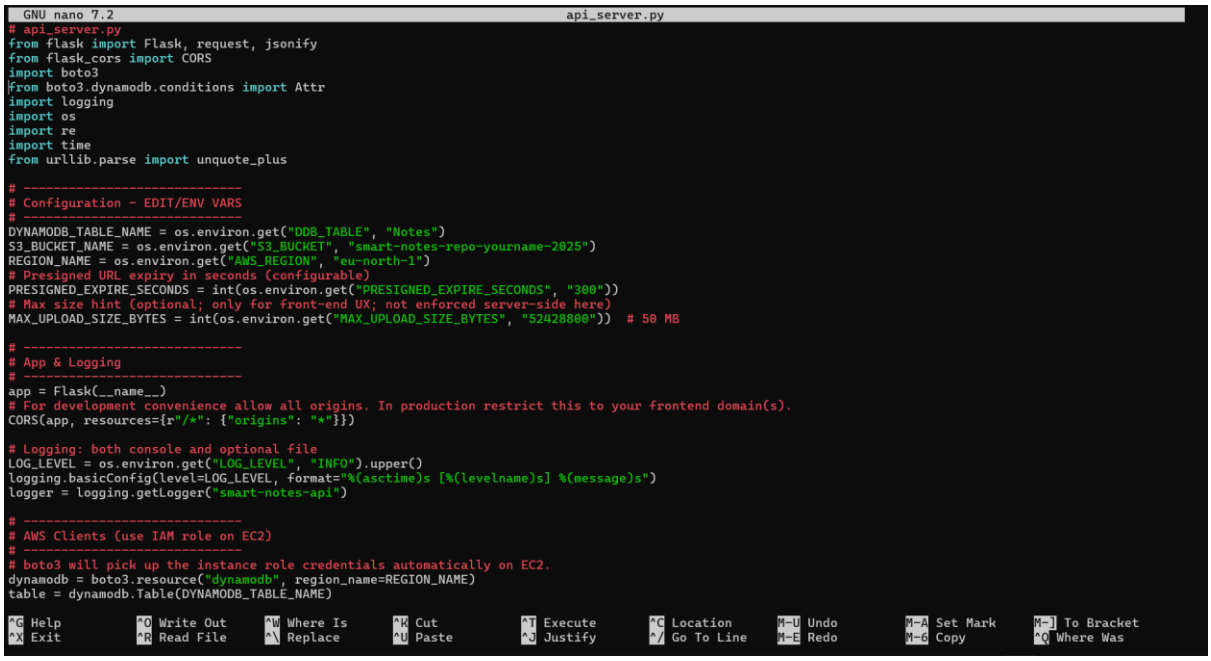
Fig 10: The 70% CPU utilization Alarm

3.0 Application Development

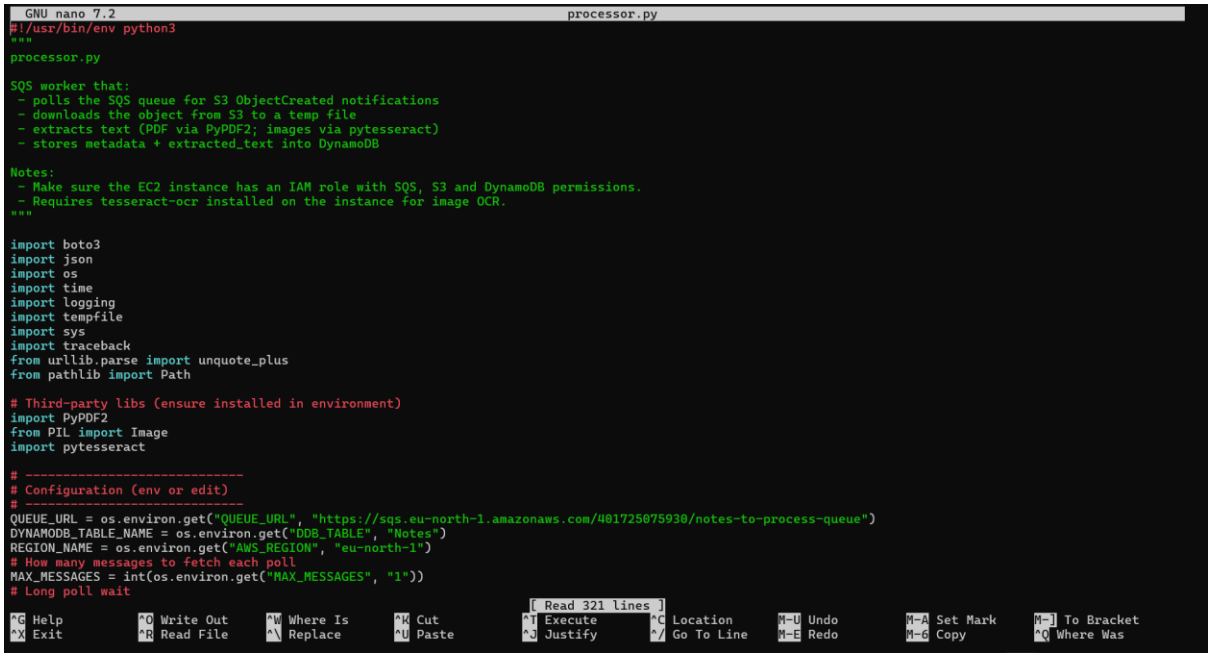
3.1 Backend (Python API & Processor)

The backend consists of two Python scripts running on the EC2 server:

1. **processor.py:** This is our asynchronous worker. It runs in an infinite loop, polling the SQS queue. When a message is received, it downloads the file from S3, uses the **PyPDF2** library for PDFs and **Pytesseract** for images to extract text. It then sanitizes the file key and stores the result in DynamoDB.
2. **api_server.py:** This is our web server, built with the **Flask** framework. It uses **Flask-CORS** to allow our S3 website to make requests. It exposes key endpoints like /search (which performs a case-insensitive scan of DynamoDB) and /get-upload-url / /get-download-url (which generate secure S3 presigned URLs).



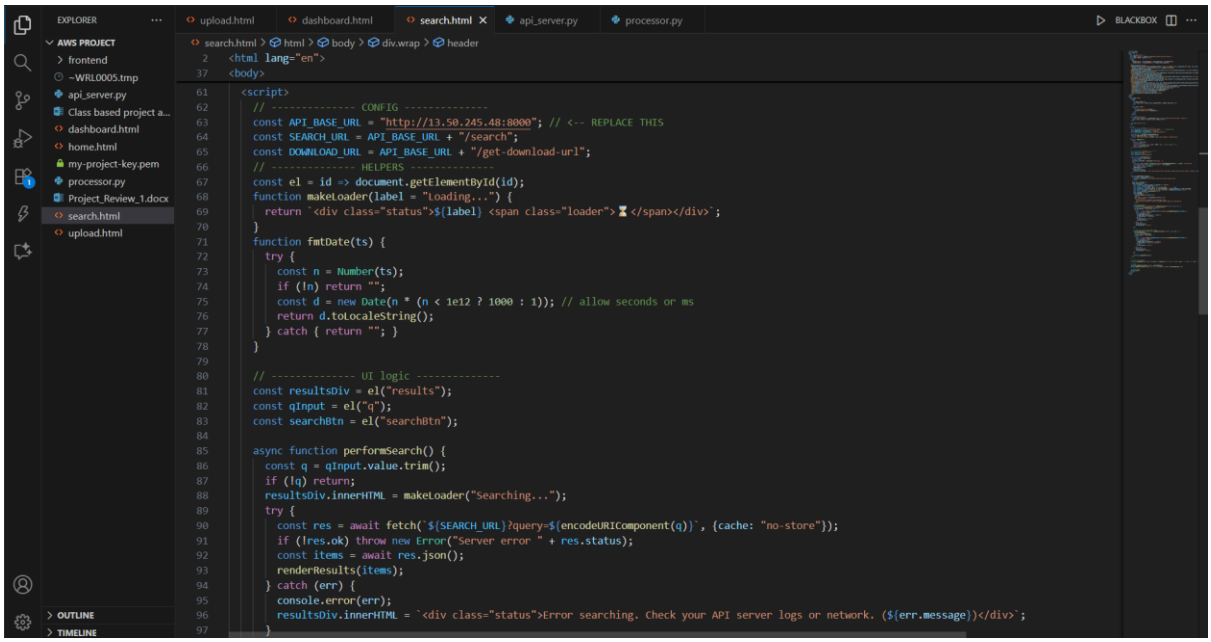
Caption: *Fig 11: A snippet of the api_server.py Flask code, showing the secure presigned URL generation.*



Caption: *Fig 12: The main worker loop of processor.py, showing the SQS receive_message call.*

3.2 Frontend (HTML/JS/CSS)

The frontend is a modern, dark-mode application built with HTML, CSS, and JavaScript. It consists of three pages (search.html, upload.html, dashboard.html). It uses the browser's fetch API to communicate asynchronously with our backend, sending search queries and requesting upload/download links.



Caption: *Fig 13: Frontend JavaScript in search.html, showing the API_BASE_URL and the performSearch function that calls our API.*

4.0 Sample Screenshots (Results)

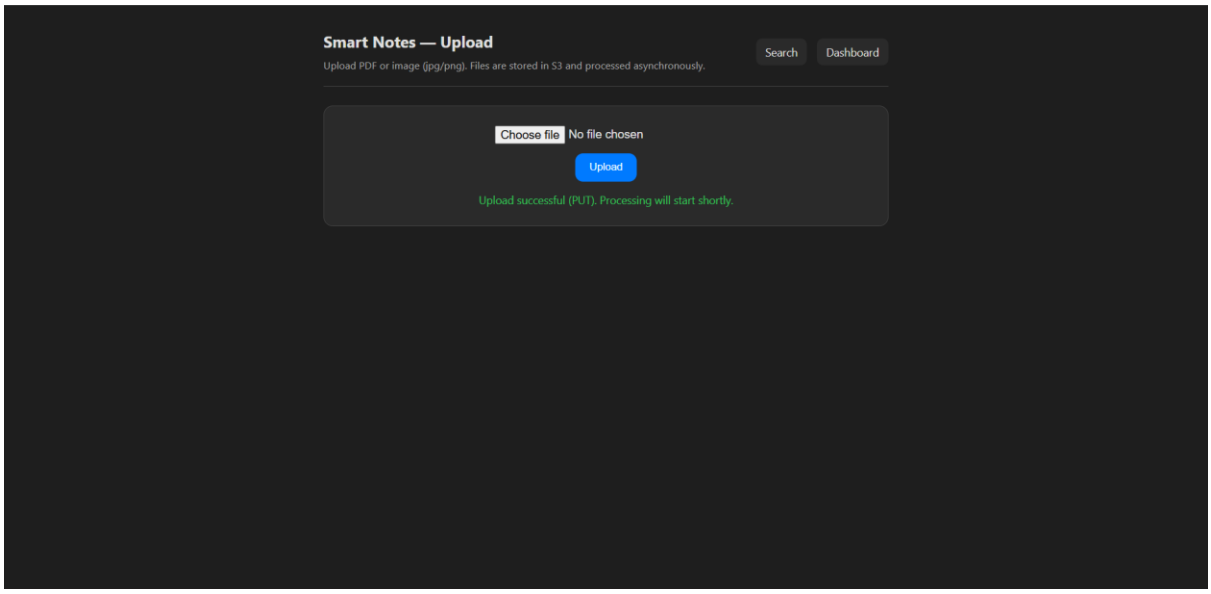


Fig 14: The public-facing 'Upload' page on our S3 static website, with a file selected.

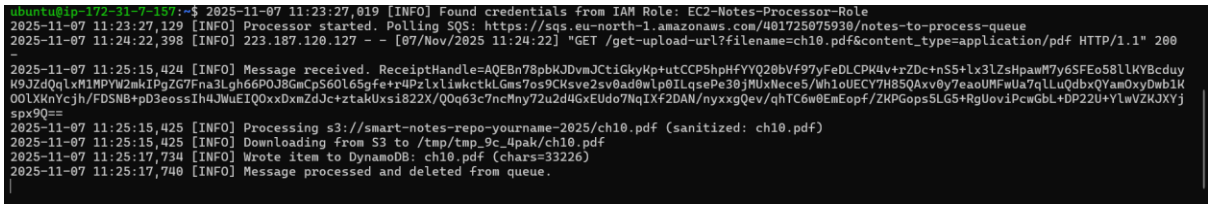


Fig 15: The EC2 processor.py log showing a file being successfully received from SQS, processed, and saved to DynamoDB.

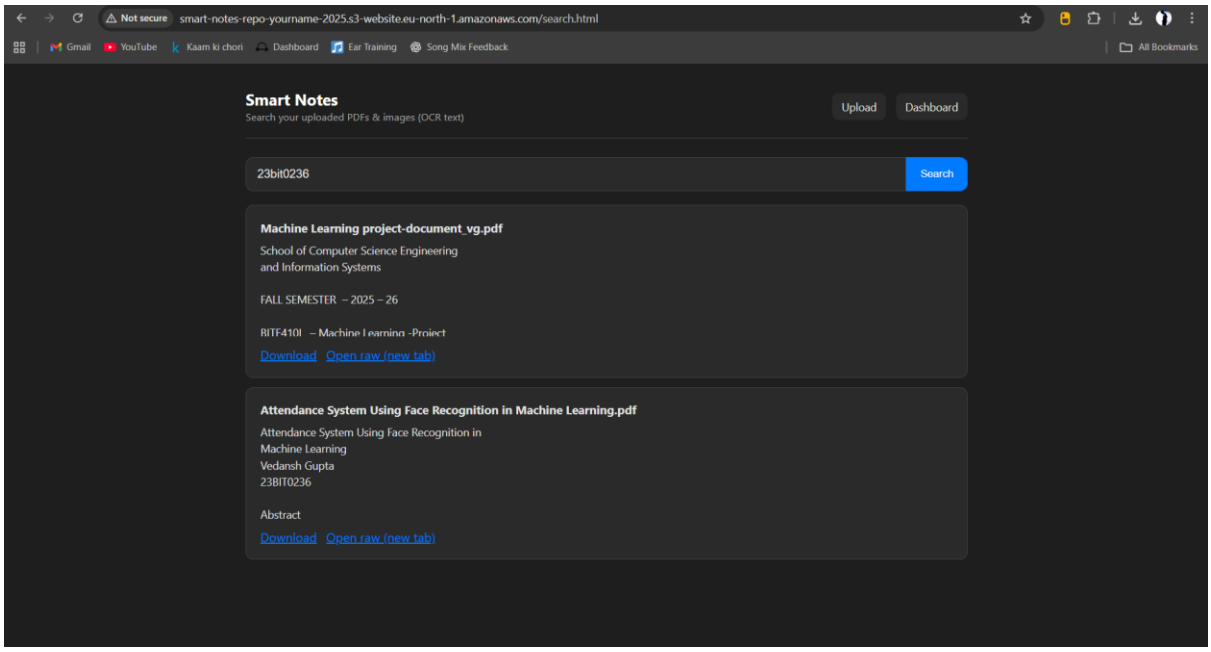


Fig 16: A successful search for '23BIT0236' on the frontend, which found the text inside AWS Assignment2.pdf.

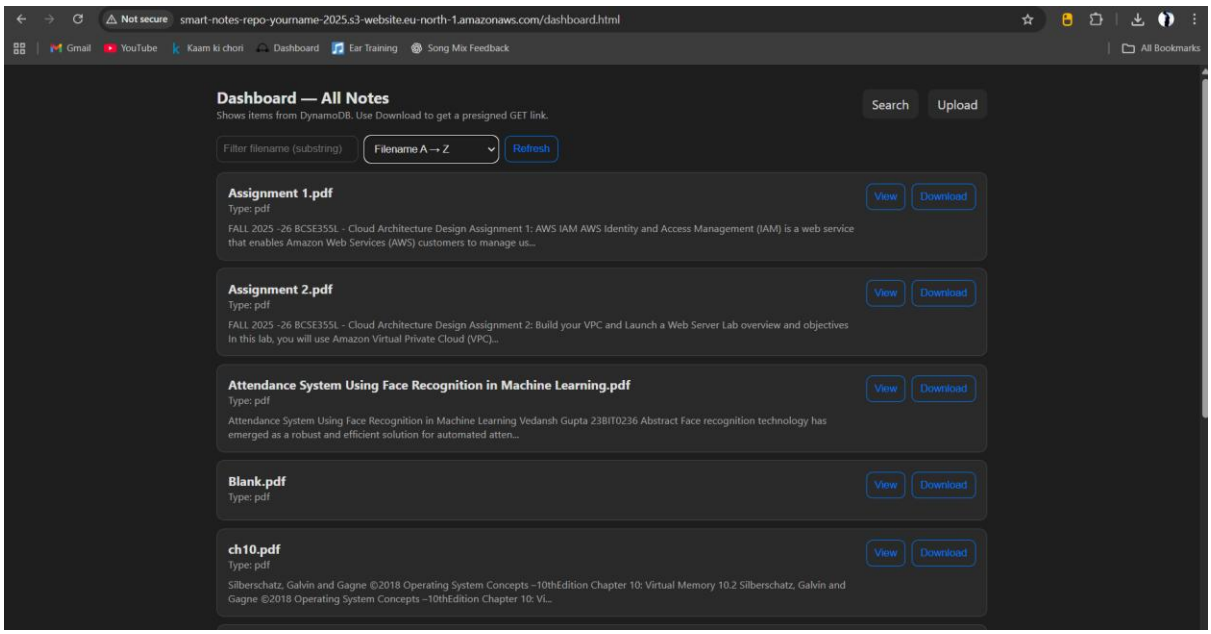


Fig 17: The 'Dashboard' view, which calls the /all-notes API to list all items currently in DynamoDB.

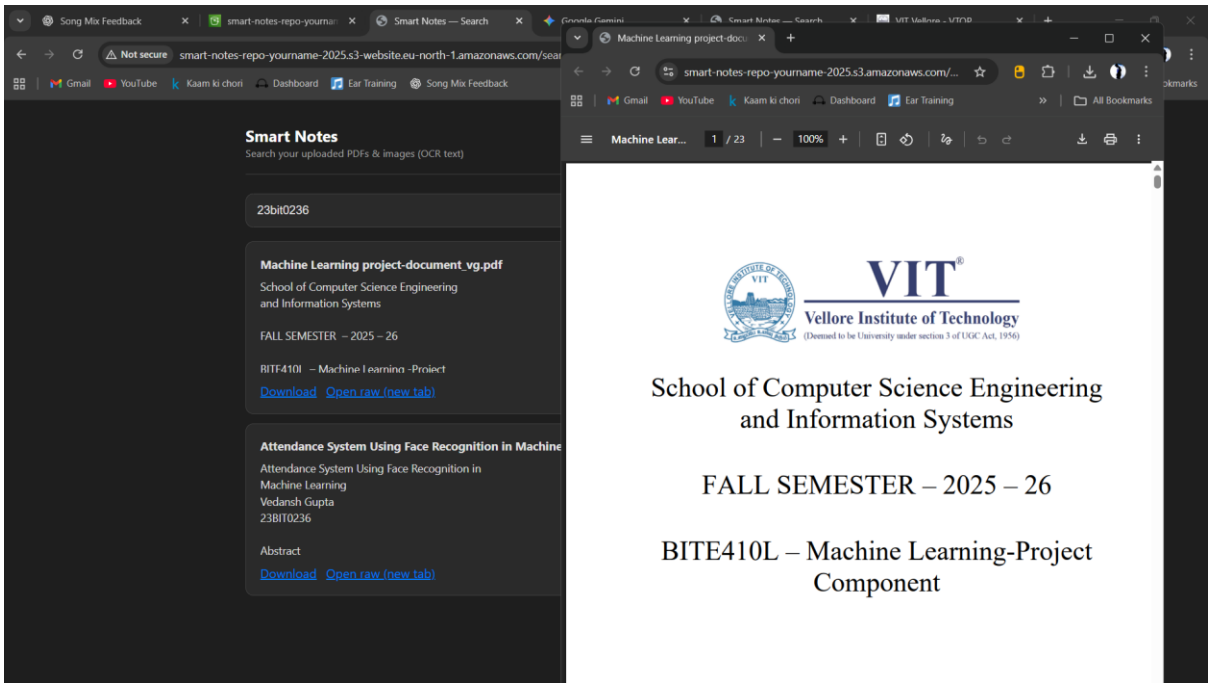


Fig 18: Clicking the 'Download' link generates a secure S3 presigned URL, allowing the user to download the original file.

5.0 Conclusion

This project successfully demonstrates a complete, end-to-end cloud application. By integrating core AWS services, we built a tool that is not only functional but also secure, highly scalable, and cost-managed.

The key achievements were:

- **Decoupling:** Using SQS to decouple the system, ensuring stability and fault tolerance.
- **Security:** Implementing a multi-layer security model with IAM Roles, Security Groups, WAF, and presigned URLs to protect all data and infrastructure.
- **Cost Management:** Proactively managing our project's finances from day one with AWS Budgets.

This architecture proves that by combining the right AWS services, we can build powerful, intelligent, and robust applications that solve real-world problems.

6.0 GitHub Repository Link

https://github.com/KushagraMukhija/Smart-Notes-Cloud-