



VIT[®]

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)

Section	Title	Page
1.0	Problem Identification	2
2.0	AWS Architecture and Services	2
	2.1 Workflow Diagram	2
	2.2 Table of AWS Services Used	3
	2.3 Detailed Service Explanation	4
3.0	Application Development	7
	3.1 Backend (Python API & Processor)	7
	3.2 Frontend (HTML/JS/CSS)	8
4.0	Sample Screenshots (Results)	9
5.0	Conclusion	11
6.0	GitHub Link	11

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

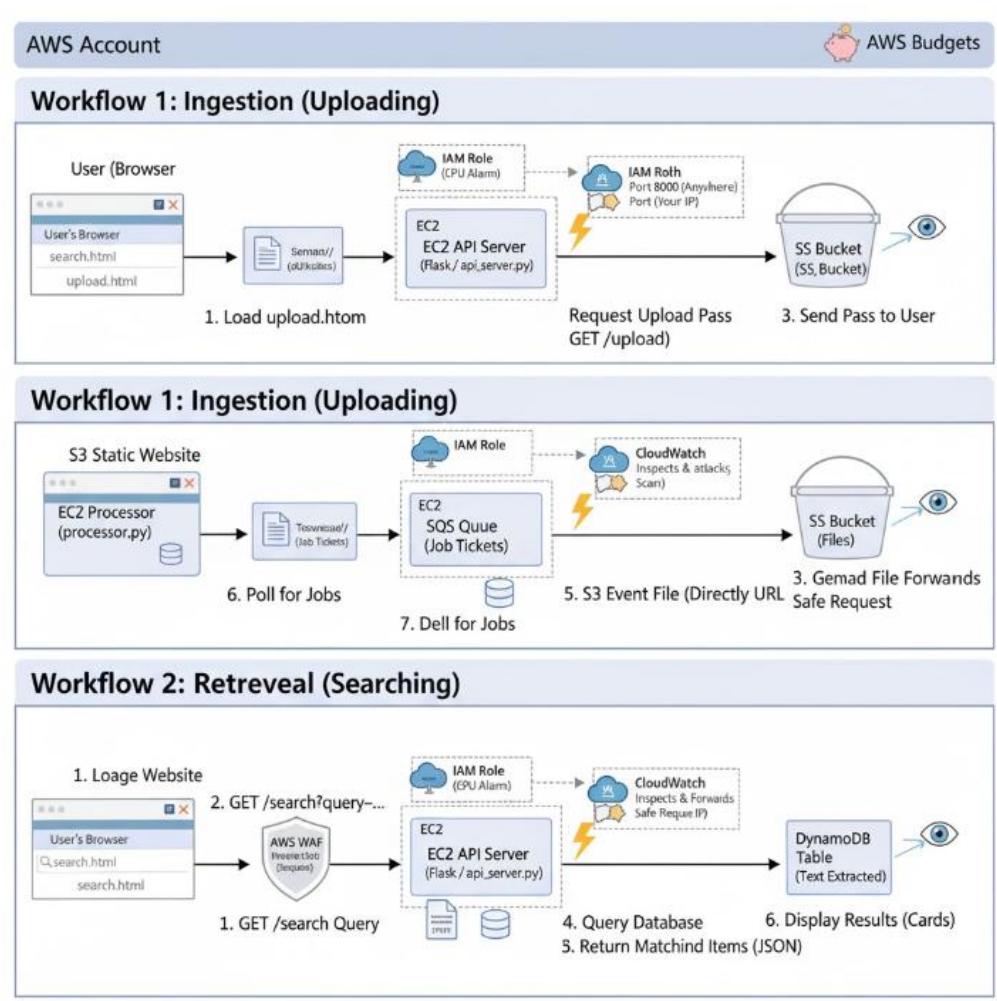


Fig 1: The decoupled, event-driven architecture of the Smart Notes Repository.

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:

- 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.
- 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 <i>forecasted</i> 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."

Objects (9) Copy S3 URI Copy URL Download Open Delete Actions Create folder Upload

Name	Type	Last modified	Size	Storage class
Assignment 2.pdf	pdf	November 6, 2025, 00:12:12 (UTC+05:30)	772.1 KB	Standard
Attendance System Using Face Recognition in Machine Learning.pdf	pdf	November 6, 2025, 23:30:12 (UTC+05:30)	355.5 KB	Standard
Blank.pdf	pdf	November 7, 2025, 14:47:12 (UTC+05:30)	12.8 KB	Standard
ch10.pdf	pdf	November 7, 2025, 15:25:49 (UTC+05:30)	1.6 MB	Standard
dashboard.html	html	November 7, 2025, 14:46:30 (UTC+05:30)	8.1 KB	Standard
Machine Learning project-document_vg.pdf	pdf	November 6, 2025, 23:32:06 (UTC+05:30)	988.7 KB	Standard
search.html	html	November 7, 2025, 14:46:31 (UTC+05:30)	7.2 KB	Standard
upload.html	html	November 7, 2025, 14:46:32 (UTC+05:30)	5.6 KB	Standard

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

Requester pays Edit

When enabled, the requester pays for requests and data transfer costs, and anonymous access to this bucket is disabled. [Learn more](#)

Requester pays
Disabled

Static website hosting Edit

Use this bucket to host a website or redirect requests. [Learn more](#)

We recommend using AWS Amplify Hosting for static website hosting
Deploy a fast, secure, and reliable website quickly with AWS Amplify Hosting. Learn more about [Amplify Hosting](#) or [View your existing Amplify apps](#) Create Amplify app

S3 static website hosting
Enabled

Hosting type
Bucket hosting

Bucket website endpoint
When you configure your bucket as a static website, the website is available at the AWS Region-specific website endpoint of the bucket. [Learn more](#)
<http://smart-notes-repo-yourname-2025.s3-website.eu-north-1.amazonaws.com>

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."

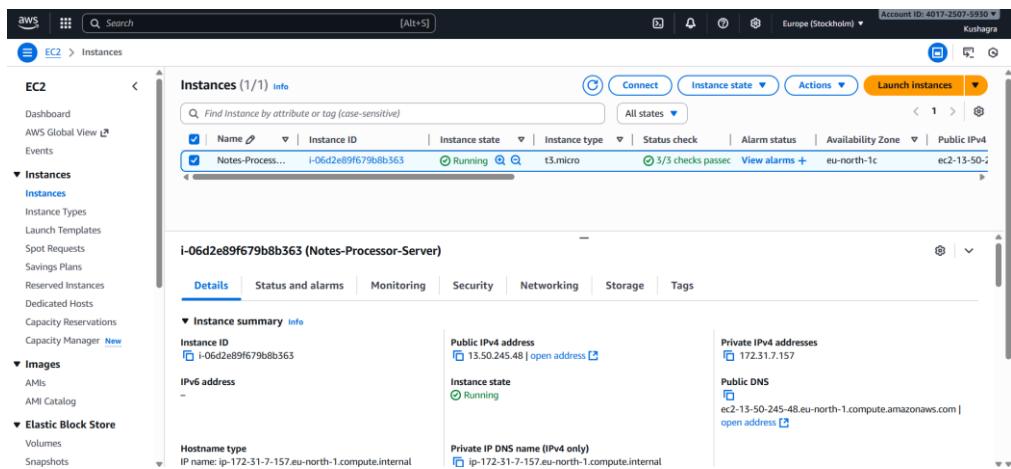


Fig 4: The Notes-Processor-Server (t3.micro) in the "Running" state in the eu-north-1 region.

- **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."

The screenshot shows the AWS SQS Queue configuration for 'notes-to-process-queue'. It includes fields for Name (notes-to-process-queue), Type (Standard), ARN (arn:aws:sqs:eu-north-1:401725075930:notes-to-process-queue), Encryption (Amazon SQS key (SSE-SQS)), URL (https://sns.eu-north-1.amazonaws.com/401725075930/notes-to-process-queue), and Dead-letter queue (-). Below the configuration, there are tabs for Queue policies, Monitoring, SNS subscriptions, Lambda triggers, EventBridge Pipes, Dead-letter queue, Tagging, Encryption, and Dead-letter queue redir.

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

The screenshot shows the AWS DynamoDB Explore items page for the 'Notes' table. The left sidebar has sections for Dashboard, Tables, Explore items (PartiQL editor, Backups, Exports to S3, Imports from S3, Integrations, Reserved capacity, Settings), and DAX (Clusters, Subnet groups, Parameter groups, Events). The main area shows a table titled 'Table: Notes - Items returned (10)' with a scan started on November 07, 2025, at 16:39:48. The table lists 10 items with columns for file_name, extracted_text, file_type, processed_at, reason, and sk.

file_name (String)	extracted_text	file_type	processed_at	reason	sk
Machine Learning proj...	School of Compu...	pdf	1762452127		
dashboard.html	<empty>	html	1762506990	unsupporte...	true
home.html	<empty>	html	1762488952	unsupporte...	true
ch10.pdf	Silberschatz, Galv...	pdf	1762509352		
Attendance System Us...	Attendance Syste...	pdf	1762452013		
Assignment 2.pdf	FALL 2025 -26 BC...				
search.html	<empty>	html	1762506991	unsupporte...	true
Assignment 1.pdf	FALL 2025 -26 BC...	pdf	1762489106		
Blank.pdf	<empty>	pdf	1762507032		
upload.html	<empty>	html	1762506992	unsupporte...	true

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."

The screenshot shows the 'Permissions policies' section of the AWS IAM console. It lists four AWS managed policies attached to a role:

Policy name	Type	Attached entities
AmazonDynamoDBFullAccess	AWS managed	1
AmazonS3FullAccess	AWS managed	1
AmazonSQSFullAccess	AWS managed	1
CloudWatchAgentServerPolicy	AWS managed	1

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

The screenshot shows the 'Inbound rules' section of the AWS Security Groups console. It displays two rules:

Name	Security group rule ID	Port range	Protocol	Source	Security groups
-	sgr-0e3d23bfa347bbef7	22	TCP	0.0.0.0/0	project-server-sg
-	sgr-066804ec3bb93d3cb	8000	TCP	0.0.0.0/0	project-server-sg

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."

The screenshot shows the 'Budget details' page for 'My_Budget' in the AWS Billing and Cost Management console. It includes sections for budget health, alerts, and detailed budget parameters:

- Budget health:** Current vs. budgeted (0.00%) and Forecasted vs. budgeted (MTD) (0.00%).
- Alerts:** Shows a green 'OK' threshold status and a link to 'View all alerts'.
- Details:** Health status (Healthy), Budget amount (\$10.00), Start date (2025-11-01), Budget type (Cost budget info), Period (Monthly), and End date (-).
- Additional budget parameters:** A section for setting up alerts.

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.

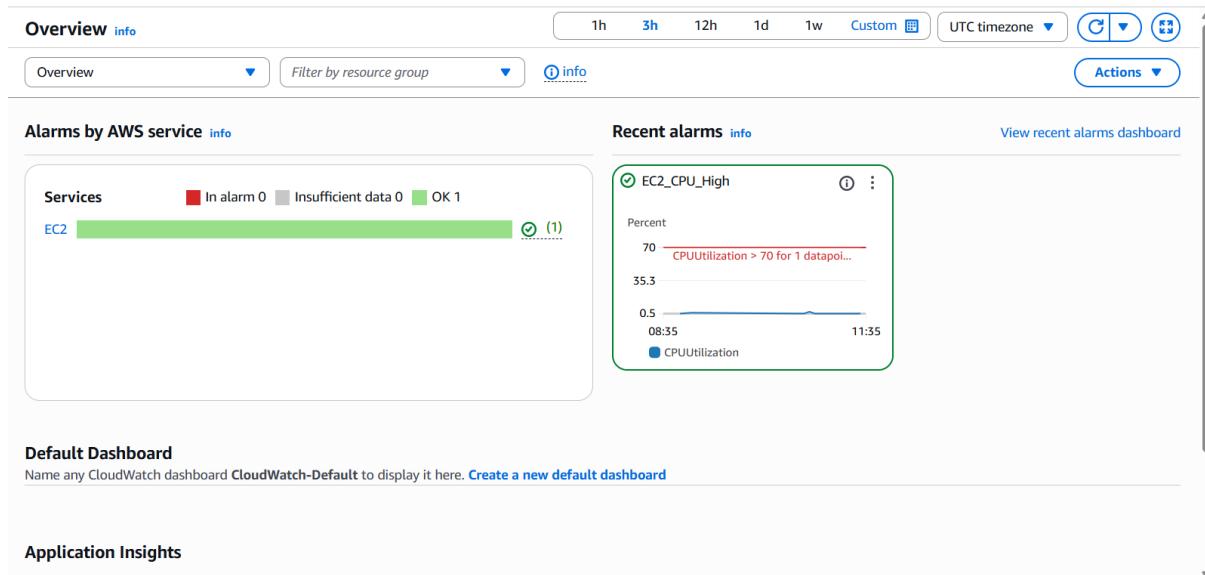


Fig 10: The 70% CPU utilization Alarm

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).

```

GNU nano 7.2                               api_server.py
# api_server.py
from flask import Flask, request, jsonify
from flask_cors import CORS
import boto3
from boto3.dynamodb.conditions import Attr
import logging
import os
import re
import time
from urllib.parse import unquote_plus

# -----
# Configuration - EDIT/ENV VARS
#
# DYNAMODB_TABLE_NAME = os.environ.get("ddb_table", "Notes")
S3_BUCKET_NAME = os.environ.get("s3_bucket", "smart-notes-repo-yourname-2025")
REGION_NAME = os.environ.get("aws_region", "eu-north-1")
# Presigned URL expiry in seconds (configurable)
PRESIGNED_EXPIRE_SECONDS = int(os.environ.get("presigned_expire_seconds", "300"))
# Max size hint (optional; only for front-end UX; not enforced server-side here)
MAX_UPLOAD_SIZE_BYTES = int(os.environ.get("max_upload_size_bytes", "52428800")) # 50 MB

# -----
# App & Logging
#
app = Flask(__name__)
# For development convenience allow all origins. In production restrict this to your frontend domain(s).
CORS(app, resources={r"/": {"origins": "*"}})

# Logging: both console and optional file
LOG_LEVEL = os.environ.get("log_level", "INFO").upper()
logging.basicConfig(level=LOG_LEVEL, format"%(asctime)s [%({levelname}s)] %(message)s")
logger = logging.getLogger("smart-notes-api")

# -----
# AWS Clients (use IAM role on EC2)
#
# boto3 will pick up the instance role credentials automatically on EC2.
dynamodb = boto3.resource("dynamodb", region_name=REGION_NAME)
table = dynamodb.Table(DYNAMODB_TABLE_NAME)

[G Help      ^O Write Out    ^W Where Is     ^K Cut          ^T Execute      ^C Location     M-U Undo      M-A Set Mark   M-J To Bracket
^X Exit      ^R Read File    ^R Replace     ^U Paste        ^J Justify      ^G Go To Line   M-E Redo      M-G Copy       ^Q Where Was

```

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

```

GNU nano 7.2                               processor.py
#!/usr/bin/env python3
processor.py

SQS worker that:
- polls the SQS queue for S3 ObjectCreated notifications
- downloads the object from S3 to a temp file
- extracts text (PDF via PyPDF2; Images via pytesseract)
- stores metadata + extracted_text into DynamoDB

Notes:
- Make sure the EC2 instance has an IAM role with SQS, S3 and DynamoDB permissions.
- Requires tesseract-ocr installed on the instance for image OCR.
"""

import boto3
import json
import os
import time
import logging
import tempfile
import sys
import traceback
from urllib.parse import unquote_plus
from pathlib import Path

# Third-party libs (ensure installed in environment)
import PyPDF2
from PIL import Image
import pytesseract

# -----
# Configuration (env or edit)
#
QUEUE_URL = os.environ.get("QUEUE_URL", "https://sqs.eu-north-1.amazonaws.com/401725075938/notes-to-process-queue")
DYNAMODB_TABLE_NAME = os.environ.get("ddb_table", "Notes")
REGION_NAME = os.environ.get("aws_region", "eu-north-1")
# How many messages to fetch each poll
MAX_MESSAGES = int(os.environ.get("max_messages", "1"))

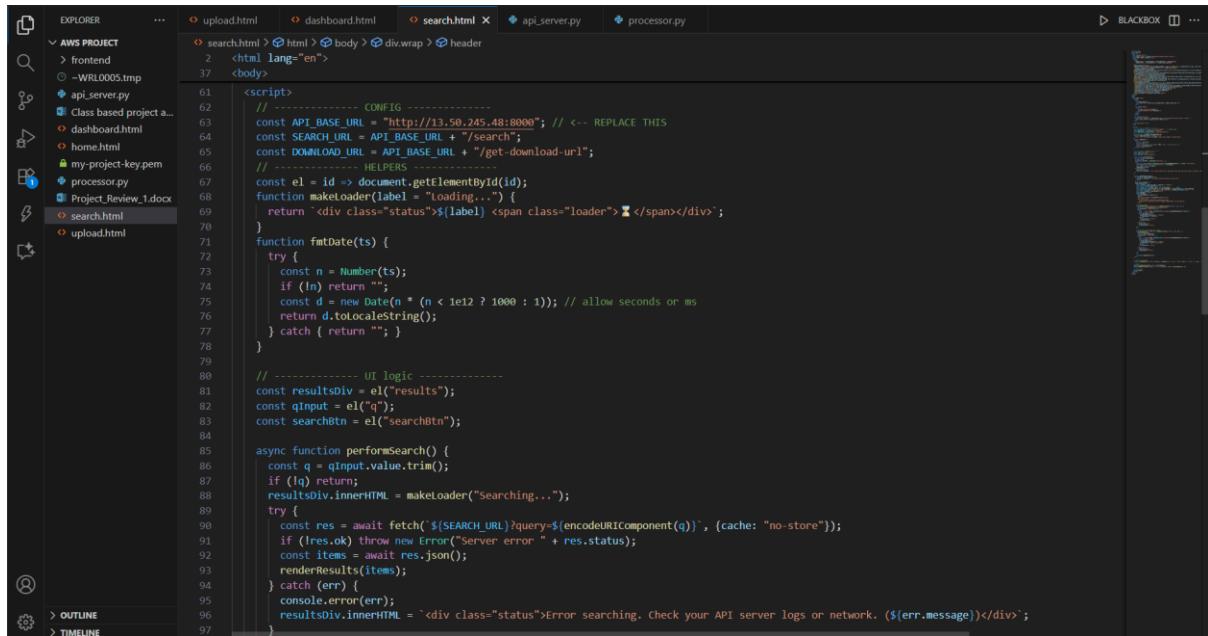
# Long poll wait
[G Help      ^O Write Out    ^W Where Is     ^K Cut          ^T Execute      ^C Location     M-U Undo      M-A Set Mark   M-J To Bracket
^X Exit      ^R Read File    ^R Replace     ^U Paste        ^J Justify      ^G Go To Line   M-E Redo      M-G Copy       ^Q Where Was

```

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.



```

EXPLORER          upload.html    dashboard.html    search.html ✘    api_server.py    processor.py
AWS PROJECT
> frontend
○ -WRL0005.tmp
api.server.py
Class based project a...
○ dashboard.html
○ home.html
my-project-key.pem
processor.py
Project_Review_1.docx
search.html
upload.html

search.html
2   <html lang="en">
37   </body>
61   <script>
62     // ----- CONFIG -----
63     const API_BASE_URL = "http://13.50.245.48:8000"; // <-- REPLACE THIS
64     const SEARCH_URL = API_BASE_URL + "/search";
65     const DOWNLOAD_URL = API_BASE_URL + "/get-download-url";
66     // ----- HELPERS -----
67     const el = id => document.getElementById(id);
68     function makeLoader(label = "Loading...") {
69       return <div class="status">${label}</div> <span class="loader">⠼</span></div>;
70     }
71     function fmtDate(ts) {
72       try {
73         const n = Number(ts);
74         if (!n) return "";
75         const d = new Date(n * (n < 1e12 ? 1000 : 1)); // allow seconds or ms
76         return d.toLocaleString();
77       } catch { return ""; }
78     }
79     // ----- UI logic -----
80     const resultsDiv = el("results");
81     const qInput = el("q");
82     const searchBtn = el("searchBtn");
83
84     async function performSearch() {
85       const q = qInput.value.trim();
86       if (!q) return;
87       resultsDiv.innerHTML = makeLoader("Searching...");
88       try {
89         const res = await fetch(`${SEARCH_URL}?query=${encodeURIComponent(q)}`, { cache: "no-store"});
90         if (!res.ok) throw new Error(`Server error ${res.status}`);
91         const items = await res.json();
92         renderResults(items);
93       } catch (err) {
94         console.error(err);
95         resultsDiv.innerHTML = `<div class="status">Error searching. Check your API server logs or network. ${err.message}</div>`;
96       }
97     }

```

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)

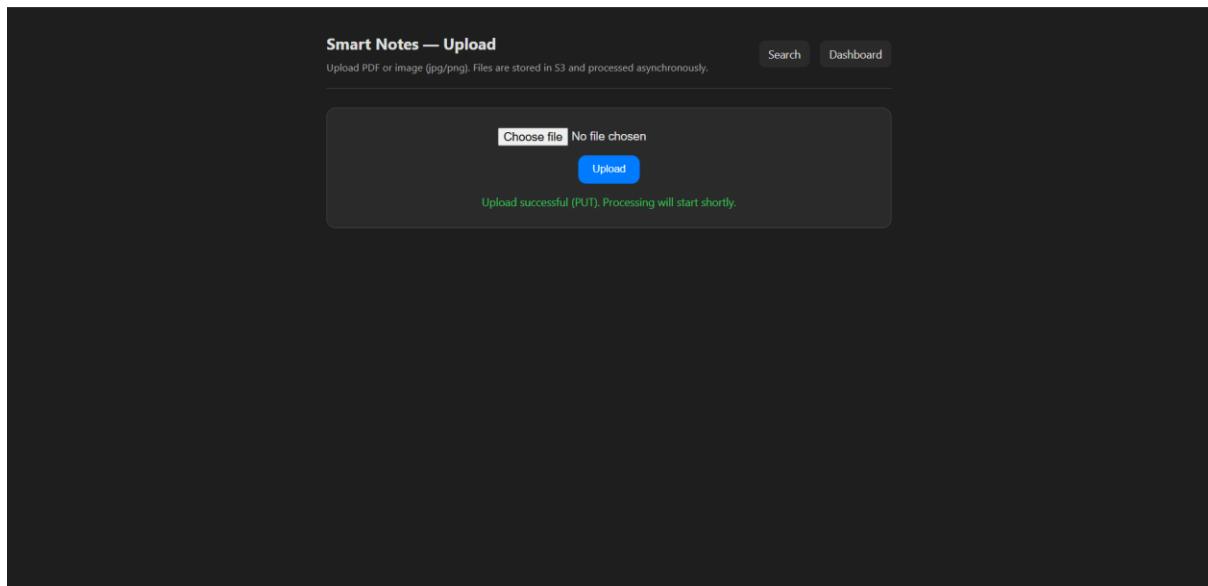


Fig 14: The public-facing 'Upload' page on our S3 static website, with a file selected.

```

ubuntu@ip-172-31-7-157:~$ 2025-11-07 11:23:27,019 [INFO] Found credentials from IAM Role: EC2-Notes-Processor-Role
2025-11-07 11:23:27,129 [INFO] Processor started. Polling SQS: https://sqs.eu-north-1.amazonaws.com/401725075936/notes-to-process-queue
2025-11-07 11:24:22,398 [INFO] 223.187.120.127 - - [07/Nov/2025 11:24:22] "GET /get-upload-url?filename=ch10.pdf&content_type=application/pdf HTTP/1.1" 200
-
2025-11-07 11:25:15,424 [INFO] Message received. ReceiptHandle=AQEBr78pbKJDVmJCTiGkyKp+utCCP5hpHFYYQ20bVf97yFeDLCPK4v+rZDc+nS5+lx3lZsHpwM7y6SEo58llKYBcduy
H9JzJdQqlx1MPW2m1PgZG7fna3Lgh66P0J8GmP560L65gf+rI4PzLxiwcktkLGms7os9Ckseve2sv0ad6wlpoILqsePe30jHUxNce5/Whi1oECY7H85QAxv0y7ea0UfMfwu7qLluQdbxQYam0xyDwB1K
001XKnYcjh/FDSNB+pD3eoss1h4JWuEIQ0xxDxm2dCjtztak0uks1822X/Q0g63c7ncMy72u2dGxEUdo7NqIXf2DAN/nyxgQev/qhTC6wEmEopf/ZkPGops5LG5+RgUoviPcwGbL+DP22U+yLwVZKJXYj
spx9Q==
2025-11-07 11:25:15,425 [INFO] Processing s3://smart-notes-repo-yourname-2025.s3-website.eu-north-1.amazonaws.com/search.html
2025-11-07 11:25:15,425 [INFO] Downloading from S3 to /tmp/tmp_9c_4pak/ch10.pdf
2025-11-07 11:25:17,734 [INFO] Wrote item to DynamoDB: ch10.pdf (chars=33226)
2025-11-07 11:25:17,740 [INFO] Message processed and deleted from queue.
|

```

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

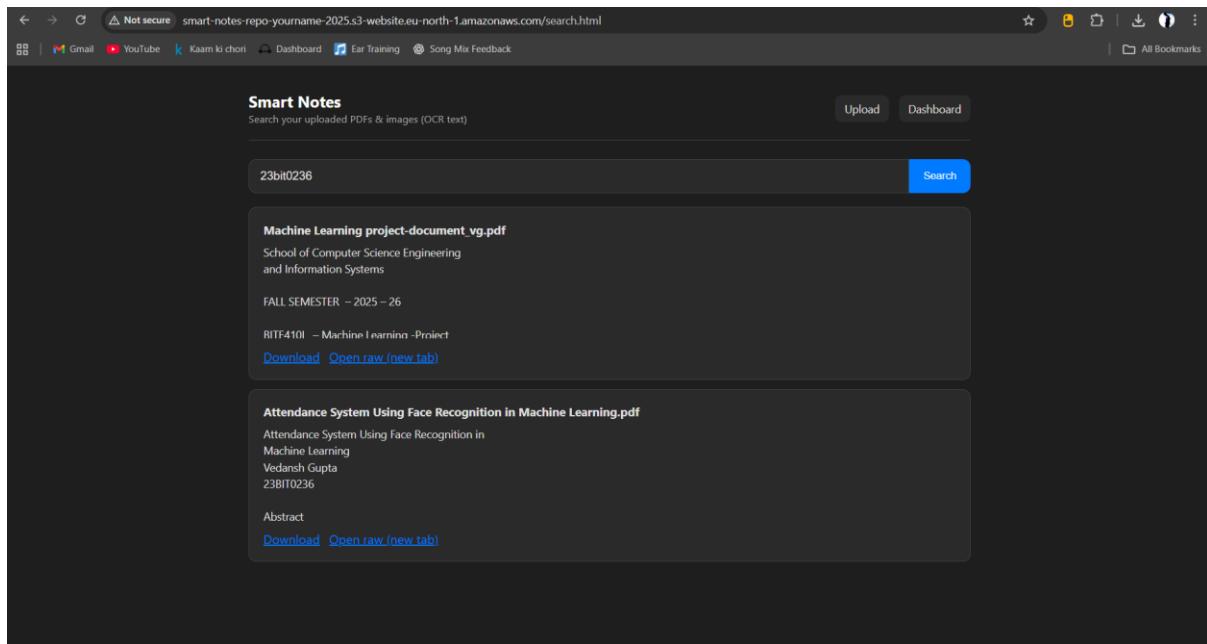


Fig 16: A successful search for '23BIT0236' on the frontend, which found the text inside AWS Assignment2.pdf.

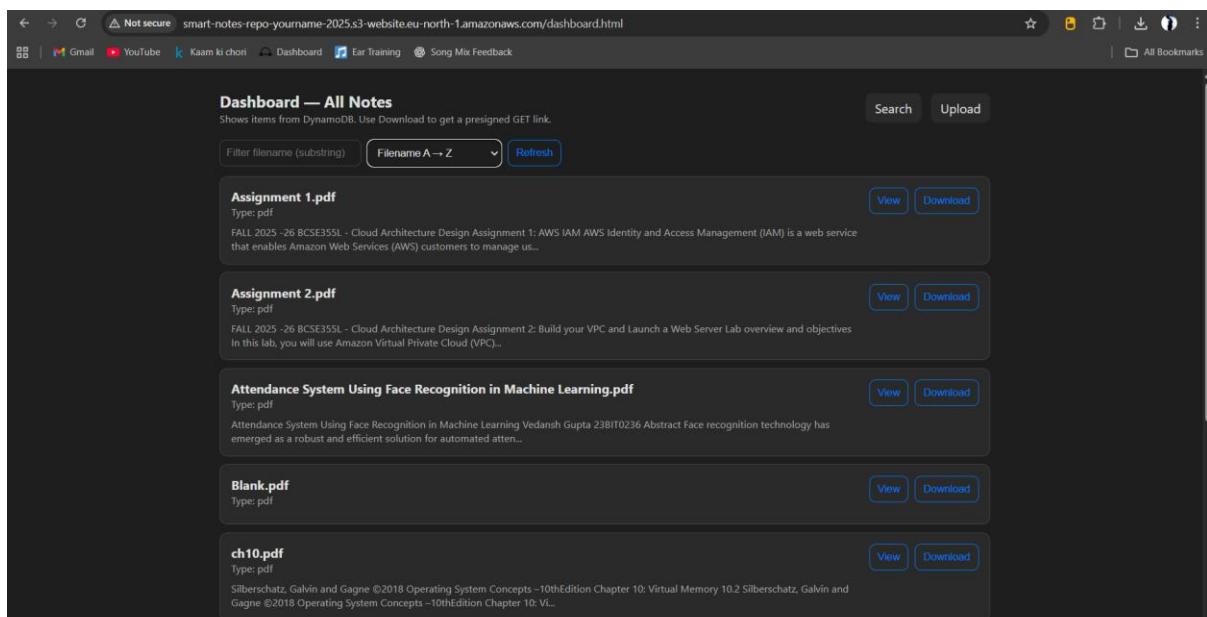


Fig 17: The 'Dashboard' view, which calls the /all-notes API to list all items currently in DynamoDB.

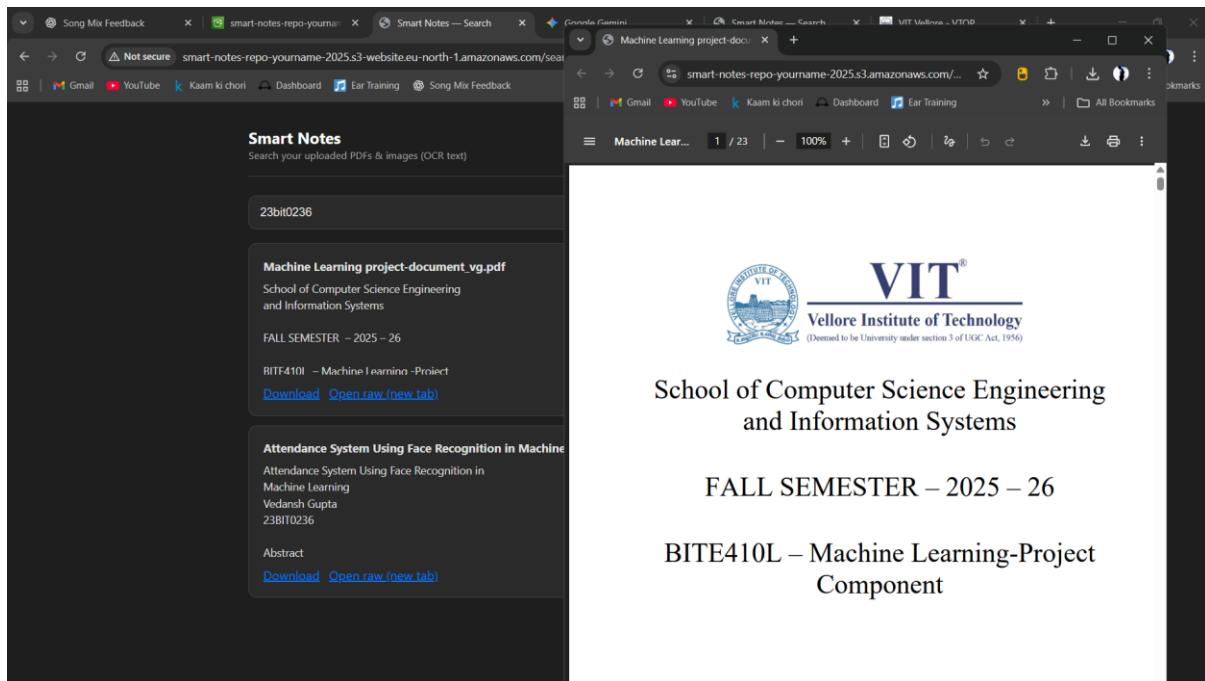


Fig 18: Clicking the 'Download' link generates a secure S3 presigned URL, allowing the user to download the original file.

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->