****

RESUME PARSER AND SKILL EXTRACTOR SYSTEM

CASE STUDY

**Submitted To:**

Cognizant Technology Solutions

**Submitted By:**

Kinjal Kanjilal

Mirza Imad Baig

Sayed Mohammad Abdur Rahman

Syun Mandal

Smruti Ranjan Rout

**Project Duration:**

1st July, 2025 – 31st July, 2025

**Batch:**

AWS and Data Engineering

# Table of Contents

**1. Project Overview**

* 1.1 Problem Statement
* 1.2 Solution
* 1.3 Key Features
* 1.4 Business Objectives

**2. System Architecture & Technology**

* 2.1 Architecture Diagram
* 2.2 Technology Stack

**3. Implementation Details**

* 3.1 Core AWS Services Utilized
* 3.2 Component Deep Dive
  + 3.2.1 Web Frontend & Flask Application (app.py)
  + 3.2.2 AWS S3 Buckets: The Data Backbone
  + 3.2.3 AWS Lambda Function 1: Text Extraction
  + 3.2.4 AWS Lambda Function 2: AI-Powered Structuring (Amazon Bedrock)
  + 3.2.5 AWS Lambda Function 3: JSON Aggregation
  + 3.2.6 AWS Lambda Function 4: Data Anonymization
  + 3.2.7 AWS Lambda Function 5: Email Notification (AWS SES)
  + 3.2.8 AWS DynamoDB: Metadata Logging
* 3.3 End-to-End Data Workflow
  + 3.3.1 Stage 1: Ingestion and Raw Text Extraction
  + 3.3.2 Stage 2: AI-Driven Structuring and Aggregation
  + 3.3.3 Stage 3: Reporting, Anonymization, and Notification
* 3.4 Deployment and Initialization
  + 3.4.1 Automated Setup via Flask Application
  + 3.4.2 Configuration Management (config.py)
* 3.5 Security and Compliance
  + 3.5.1 Secure Uploads with Pre-signed URLs
  + 3.5.2 IAM Roles and Permissions
  + 3.5.3 Data Privacy and Anonymization

**4. Challenges & Solutions**

* 4.1 Accurate Name Extraction from Diverse Resumes
* 4.2 PDF Library Compatibility in the AWS Lambda Environment
* 4.3 Managing AWS Permissions Securely

**5. Future Scope & Improvements**  
**6. Output Screenshots**

**1. Project Overview**

**1.1 Problem Statement**

Organizations and educational institutions often receive hundreds of resumes for a single opening. The manual process of parsing these documents to identify candidate skills is extremely time-consuming and susceptible to human error.

**1.2 Solution**

This project is an automated, serverless solution that ingests resume files, intelligently parses them to extract key information, and stores the structured data for easy access and analysis. By leveraging AWS cloud services, the system is scalable, cost-effective, and efficient.

**1.3 Key Features**

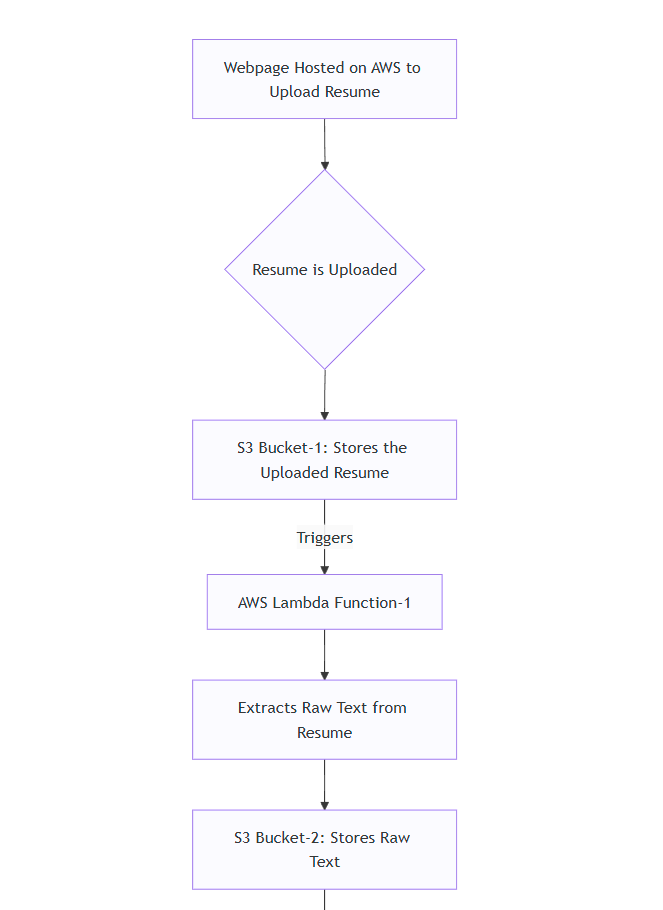
* **File Ingestion:** Accepts resumes in .pdf format only
* **Automated Parsing:** Automatically extracts entities like:
  + Candidate ID
  + Name
  + Contact Information (Email, Phone)
  + Skills (e.g., Python, AWS, SQL)
  + Work Experience
  + languages
* **Structured Output:** Saves the parsed information as structured JSON data in a S3 bucket. And metadata is stored in Dynamo DB.
* **Dashboard:** The final JSON is then presented in a Flask-based dashboard.

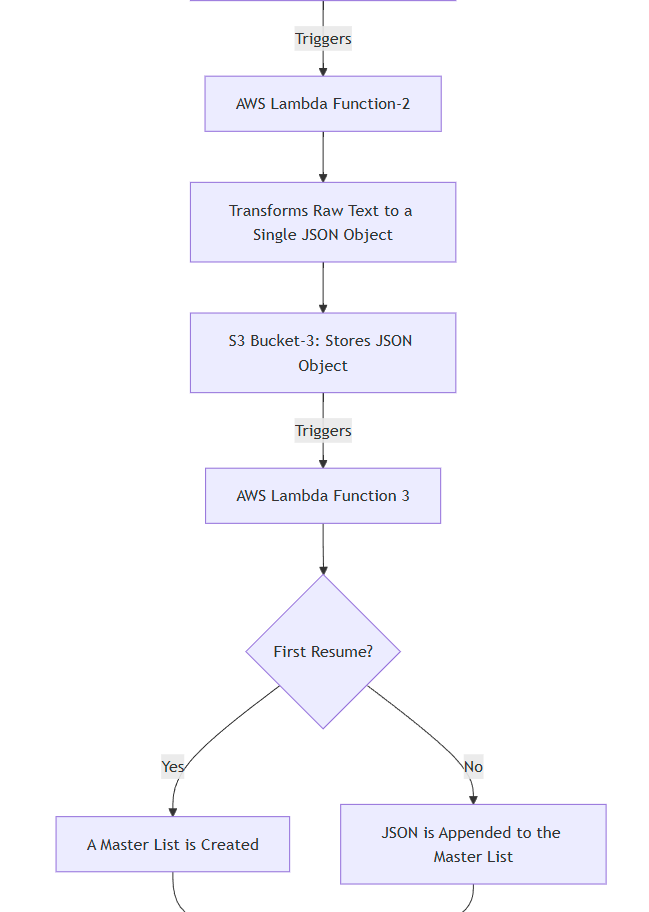
**1.4 Business Objectives**

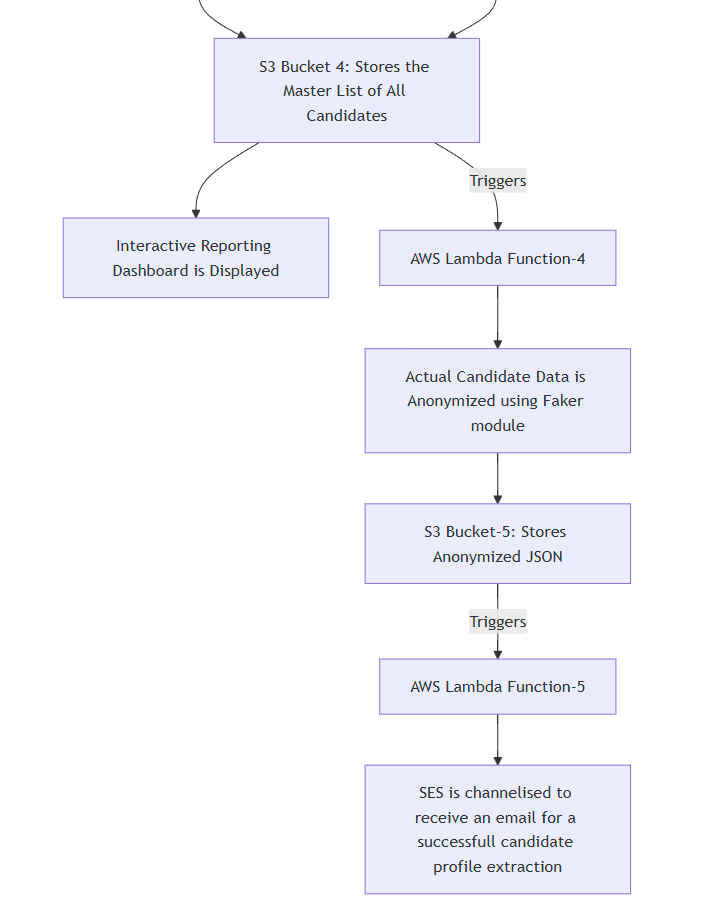
* **Reduce Manual Effort:** Eliminate the time-consuming task of manually reading and categorizing resumes.
* **Accelerate Time-to-Hire:** Quickly identify qualified candidates by structuring and standardizing resume data.
* **Improve Candidate Matching:** Use AI to extract key skills, experience, and qualifications for better search and filtering.
* **Enable Data-Driven Decisions:** Create a centralized, searchable database of candidates to inform recruitment strategies.
* **Enhance Security & Compliance:** Ensure candidate data is handled securely and support unbiased reviews through data anonymization.

**2. System Architecture & Technology**

**2.1 Architecture Diagram**

****

****

****

Here’s a description of the flow:

1. **Web Upload:** A user uploads a PDF resume through a Flask web application, which securely sends the file directly to an S3 bucket (resume-parser-input-bucket-testv10).
2. **Text Extraction:** The upload triggers the first Lambda function (ResumeExtractorFunctionv10), which opens the PDF and extracts all the raw text from it.
3. **AI-Powered Analysis:** This raw text is saved to another S3 bucket, triggering a second Lambda function (jsonlambdaextractorv10). This function uses Amazon Bedrock (an AI service) to read the text and understand it, extracting structured information like the candidate's name, skills, and experience summary.
4. **Structured JSON Creation:** The AI-extracted information is formatted into a clean, individual JSON file and saved to a third S3 bucket (json-extraction-bucket-testv10).
5. **Master List Aggregation:** The creation of this individual JSON file triggers a third Lambda function (jsonlambdaappenderv10). This function adds the new candidate's JSON data to a single master candidates.json file located in a fourth S3 bucket (json-dashboard-bucket-testv10).
6. **Dashboard Data Source:** This master candidates.json file acts as the live database for the administrative dashboard, allowing recruiters to see and search for all processed candidates.
7. **Data Anonymization:** When the master list is updated, a fourth Lambda function is triggered. It creates a copy of the list but removes all personally identifiable information (like name and email) to ensure privacy and support unbiased screening.
8. **Anonymized Data Storage:** This new, anonymized list is saved to a fifth and final S3 bucket.
9. **Email Notification:** The arrival of the anonymized file triggers the fifth Lambda function, which uses AWS Simple Email Service (SES) to automatically send an email to the recruitment team, alerting them that a new candidate has been processed.
10. **Fully Automated & Serverless:** The entire pipeline is event-driven and serverless, meaning each step automatically triggers the next without any manual intervention or need to manage servers.

**2.2 Technology Stack**

**Cloud & Serverless Infrastructure**

* **AWS S3 (Simple Storage Service)**

Used for durable, scalable object storage of the uploaded resume files, intermediate text files, and final JSON outputs across a series of dedicated buckets.

* + **Why S3?** It's cost-effective, highly available, and integrates seamlessly as the event-driven backbone for other AWS services like Lambda.
* **AWS Lambda**

The core of the serverless compute logic. It runs the Python code to orchestrate the text extraction, AI analysis, and data aggregation processes without needing a dedicated server.

* + **Why Lambda?** It's event-driven (triggers on S3 uploads), scales automatically, and you only pay for the compute time you use, making it highly efficient.
* **Amazon Bedrock**

A fully managed service used to access powerful foundation models (FMs) for intelligent data extraction. It is used to analyze the raw resume text to accurately identify the candidate's name and generate a professional experience summary.

* + **Why Bedrock?** It provides easy, serverless access to state-of-the-art AI models without the complexity of managing infrastructure. Its seamless integration within AWS makes it superior for this pipeline compared to external APIs or self-hosted models.
* **Amazon DynamoDB**

A NoSQL key-value database used to optionally store metadata (filename, timestamp) about each resume upload.

* + **Why DynamoDB?** It offers single-digit millisecond performance for reads/writes, is fully managed, and scales seamlessly. This makes it perfect for a high-speed, queryable audit trail of processed files, separate from the primary data storage in S3.
* **Amazon SES (Simple Email Service)**

A cloud email service used to send automated notifications to the recruitment team after a candidate's profile has been successfully processed and anonymized.

* + **Why SES?** It is a highly scalable, cost-effective, and reliable way to integrate email functionality directly into the serverless workflow without managing an email server.

**Application & Language**

* **Python**

The exclusive programming language used for all backend logic, including the Lambda functions and the Flask web application. The Boto3 library is used as the AWS SDK for Python.

* + **Why Python?** It has excellent, mature support for AWS via Boto3, a vast ecosystem of libraries for tasks like PDF parsing (pdfminer) and data manipulation, and is a premier language for data scripting.
* **Flask**

A lightweight Python web framework used to build the Admin Dashboard. This dashboard allows users to upload resumes, view parsed profiles, search for candidates by skill, and download reports.

* + **Why Flask?** It's simple, flexible, and requires minimal setup, making it ideal for quickly developing a functional user interface for the project without the overhead of a larger framework.
* **Flask**

The AWS SDK (Software Development Kit) for Python. It allows Python developers to write software that makes use of services like Amazon S3, EC2, and DynamoDB.

* + **Why *Boto3?*** It is the official and essential library for programmatically interacting with any AWS service from within Python code, providing the necessary bridge between the application logic and the cloud infrastructure.

**Deployment Target**

* **AWS EC2 (Elastic Compute Cloud)**

A virtual server in the cloud that serves as the deployment target for hosting the Flask web application.

* + **Why EC2?** It provides complete control over the hosting environment (OS, software, security configurations) and is a standard, reliable way to deploy web applications, demonstrating knowledge of hosting on traditional cloud infrastructure.

**3. Implementation Details**

**3 System Architecture**

**3.1 Core AWS Services Utilized**

* **AWS S3 (Simple Storage Service):** Used as the data backbone, providing durable and scalable object storage for resumes, extracted text, and structured JSON data across five distinct buckets.
* **AWS Lambda:** The core of the serverless compute engine. Five separate Lambda functions are used to execute specific tasks in the workflow without provisioning servers.
* **Amazon Bedrock:** A fully managed service that provides access to foundation models. This project uses it as the "brain" to understand resume text and extract structured information.
* **AWS IAM (Identity and Access Management):** Manages permissions for all AWS services, ensuring that Lambda functions have the precise roles needed to access S3 buckets and other services securely.
* **AWS SES (Simple Email Service):** A cloud-based email sending service used to dispatch notifications to the recruitment team**.**
* **AWS DynamoDB:** A NoSQL database service, used optionally in this project for logging metadata about uploaded files.
* **Flask:** A Python web framework used to build the user-facing upload portal and administrative dashboard.

**3.2 Component Deep Dive**

**3.2.1 Web Frontend & Flask Application (app.py)**

The user interaction point is a web application powered by the Flask framework.

* **Role:**
  + Serves the HTML upload page (upload.html) to the user.
  + Provides an administrative dashboard for viewing processed data.
  + Handles the backend logic for secure file uploads and initializes the entire AWS infrastructure on startup.
* **Secure Uploads:** To avoid handling large file uploads on the server, the application uses S3 pre-signed URLs. The process is as follows:
  + The user selects a file in the browser.
  + The frontend sends a request to the /generate-url endpoint on the Flask server.
  + The server generates a temporary, secure URL that grants write-only permission to a specific key in the raw resumes S3 bucket.
  + The frontend receives this URL and uploads the file directly to S3.

1. # From app.py: Generating a pre-signed URL

2. @app.route('/generate-url', methods=['POST'])

3. def generate\_presigned\_url():

4. data = request.json

5. filename = data.get('filename')

6. # ... validation ...

7. presigned\_url = s3.generate\_presigned\_url(

8. ClientMethod='put\_object',

9. Params={'Bucket': 'resume-parser-input-bucket-testv10', 'Key': filename, 'ContentType': 'application/pdf'},

10. ExpiresIn=300

11. )

12. return jsonify({'url': presigned\_url})

13.

**3.2.2 AWS S3 Buckets: The Data Backbone**

The pipeline uses five S3 buckets, each with a specific purpose defined in config.py, to ensure a clean separation of data states.

* **Bucket 1 (resume-parser-input-bucket-testv10):** Raw Resumes - The landing zone for all original PDF resumes uploaded by users.
* **Bucket 2 (resume-parser-output-bucket-testv10):** Extracted Text - Stores the unstructured text (.txt files) extracted from the PDFs.
* **Bucket 3 (json-extraction-bucket-testv10):** Individual JSON Objects - Stores the structured JSON file for each individual candidate after AI processing.
* **Bucket 4 (json-dashboard-bucket-testv10):** Master Candidate List - Contains the single, aggregated candidates.json file used by the dashboard.
* **Bucket 5 (json\_anonymous\_bucket):** Anonymized Data - Stores the anonymized version of the master candidate list. (Note: Name defined in config.py)

**3.2.3 AWS Lambda Function 1: Text Extraction**

* **Function Name:** ResumeExtractorFunctionv10
* **Trigger:** s3:ObjectCreated event in resume-parser-input-bucket-testv10.
* **Code:** text\_extraction.py
* **Purpose:** To extract raw text from PDF files. It uses the pdfminer library to parse the document and saves the output as a .txt file in resume-parser-output-bucket-testv10.

**3.2.4 AWS Lambda Function 2: AI-Powered Structuring (Amazon Bedrock)**

* **Function Name:** jsonlambdaextractorv10
* **Trigger:** s3:ObjectCreated event in resume-parser-output-bucket-testv10.
* **Code:** bedrock\_lambda.py
* **Purpose:** This is the intelligence hub of the pipeline. It takes the raw text and:
  1. **Calls Amazon Bedrock**: Uses targeted prompts to extract complex data like the candidate's name and a professional summary.
  2. **Uses Regex and Libraries:** Extracts structured data like email, phone number, skills, and languages.
  3. **Outputs JSON:** Saves the structured data as a single JSON object to json-extraction-bucket-testv10.

1. # From bedrock\_lambda.py: Calling the Bedrock model

2. def call\_bedrock\_nova\_model(prompt: str, max\_tokens: int) -> str:

3. model\_id = "amazon.nova-pro-v1:0"

4. response = bedrock\_runtime.converse(

5. modelId=model\_id,

6. messages=[{"role": "user", "content": [{"text": prompt}]}],

7. inferenceConfig={"maxTokens": max\_tokens, "temperature": 0.1}

8. )

9. return response['output']['message']['content'][0]['text'].strip()

10.

**3.2.5 AWS Lambda Function 3: JSON Aggregation**

* **Function Name:** jsonlambdaappenderv10
* **Trigger:** s3:ObjectCreated event in json-extraction-bucket-testv10.
* **Code:** json\_appender\_lambda.py
* **Purpose:** To maintain the master list of candidates. It reads the existing candidates.json from json-dashboard-bucket-testv10, appends the new candidate's data, and overwrites the master file. It handles the initial creation of the file if it doesn't exist.

**3.2.6 AWS Lambda Function 4: Data Anonymization**

* **Function Name:** json\_data\_anonymus\_lambda (from config.py)
* **Trigger:** s3:ObjectCreated event in json-dashboard-bucket-testv10.
* **Code:** anonymous\_lambda.py
* **Purpose:** To create a privacy-compliant version of the candidate data. It uses the Faker library to replace PII with realistic-looking fake data and saves the result to the json\_anonymous\_bucket.

1. # From anonymous\_lambda.py: Anonymizing a name

2. from faker import Faker

3. fake = Faker()

4. anonymized['name'] = fake.name()

5.

**3.7 AWS Lambda Function 5: Email Notification (AWS SES)**

* **Function Name:** email\_notifier\_lambda (from config.py)
* **Trigger:** s3:ObjectCreated event in the json\_anonymous\_bucket.
* **Code:** ses\_lambda\_function.py
* **Purpose:** To alert the recruitment team. It uses AWS SES to send a formatted HTML email to a predefined recipient, signaling that a new candidate has been successfully processed.

**3.2.8 AWS DynamoDB: Metadata Logging**

* **Enabled by:** USE\_DYNAMO = True in config.py.
* **Table Name:** ResumeMetadata-testv10
* **Purpose:** To log metadata about each file upload, such as the filename and timestamp, into the specified DynamoDB table. This provides a quick, queryable audit trail of all received resumes.

**3.3 End-to-End Data Workflow**

The entire pipeline is a seamless, event-driven process.

**3.3.1 Stage 1: Ingestion and Raw Text Extraction**

A user uploads a PDF -> The Flask app generates a pre-signed URL -> The file is uploaded to Bucket 1 (resume-parser-input-bucket-testv10) -> Lambda 1 (ResumeExtractorFunctionv10) is triggered -> Text is extracted and saved to Bucket 2 (resume-parser-output-bucket-testv10).

**3.3.2 Stage 2: AI-Driven Structuring and Aggregation**

A .txt file appears in Bucket 2 (resume-parser-output-bucket-testv10) -> Lambda 2 (jsonlambdaextractorv10) is triggered -> It calls Bedrock, extracts data, and saves a JSON object to Bucket 3 (json-extraction-bucket-testv10) -> Lambda 3 (jsonlambdaappenderv10) is triggered -> It appends the new data to the master list in Bucket 4 (json-dashboard-bucket-testv10).

**3.3.3 Stage 3: Reporting, Anonymization, and Notification**

The master list in Bucket 4 (json-dashboard-bucket-testv10) is updated -> The Dashboard can now display the new candidate -> Lambda 4 (json\_data\_anonymus\_lambda) is triggered -> It anonymizes the data and saves it to Bucket 5 (json\_anonymous\_bucket) -> Lambda 5 (email\_notifier\_lambda) is triggered -> An email notification is sent via SES.

**3.4 Deployment and Initialization**

**3.4.1 Automated Setup via Flask Application**

A key feature of this project is its streamlined setup process. The app.py script is designed to initialize all the necessary AWS resources upon startup.

* **On app.run():** The application sequentially calls the setup functions for each required S3 bucket and each of the five Lambda functions.
* **Idempotency:** The setup scripts (s3\_utils.py, lambda\_utils\*.py) are idempotent, meaning they can be run multiple times without creating duplicate resources. They check if a resource already exists before attempting to create it.

**3.4.2 Configuration Management (config.py)**

All environment-specific variables are centralized in config.py. This separation of configuration from logic is a best practice that makes the application easy to deploy in different environments. Key variables include:

* **S3 Buckets:** resume-parser-input-bucket-testv10, resume-parser-output-bucket-testv10, json-extraction-bucket-testv10, json-dashboard-bucket-testv10, json\_anonymous\_bucket.
* **Lambda Functions:** ResumeExtractorFunctionv10, jsonlambdaextractorv10, jsonlambdaappenderv10, json\_data\_anonymus\_lambda, email\_notifier\_lambda.
* **DynamoDB:** ResumeMetadata-testv10, USE\_DYNAMO.
* **AWS Settings:** us-east-1.

**3.5 Security and Compliance**

**3.5.1 Secure Uploads with Pre-signed URLs**

Directly exposing an S3 bucket for public uploads is a security risk. This architecture mitigates that risk by using temporary, pre-signed URLs. This ensures that uploads are authenticated and authorized on a per-request basis without exposing long-lived credentials to the client.

**3.5.2 IAM Roles and Permissions**

The principle of least privilege is applied. Each Lambda function is assigned a unique IAM Role with a policy that grants it only the permissions it needs to perform its specific task (e.g., read from one bucket, write to another).

**3.5.3 Data Privacy and Anonymization**

The dedicated anonymization workflow (Lambda 4) is critical for compliance and unbiased hiring practices. By creating a separate, PII-free dataset, the system allows for initial candidate screening based purely on skills and experience, reducing potential bias.

**4 Challenges & Solutions**

1. **Accurate Name Extraction from Diverse Resumes**

* ***Challenge:*** A primary difficulty was reliably extracting candidate names from resumes with inconsistent formatting. Initial approaches using standard Python scripts, the RoBERTa model, and even AWS Comprehend struggled to achieve high accuracy, particularly with diverse name formats like those common in India.
* ***Solution***: After a comparative evaluation of multiple advanced tools, the Gemini API was chosen for name entity recognition. It consistently provided the highest accuracy across all name types, overcoming the limitations of other methods and ensuring the reliability of the core data extraction feature.

1. **PDF Library Compatibility in the AWS Lambda Environment**

* ***Challenge:*** The initial choice for the PDF parsing engine, PyMuPDF, could not be deployed successfully as an AWS Lambda Layer. This was due to an operating system (OS) compatibility issue, where the library's binary dependencies were not compatible with the Amazon Linux environment that AWS Lambda uses.
* ***Solution:*** To resolve this deployment blocker, the project was migrated to the pdfminer.six library. As a pure-Python library, pdfminer.six has no external binary dependencies, which completely resolved the compatibility issues and enabled a smooth, stable deployment.

1. **Managing AWS Permissions Securely**

* ***Challenge:*** A critical security requirement was to ensure that the application's components could not be exploited by having overly broad access permissions.
* ***Solution:*** A specific IAM (Identity and Access Management) Role was created for the Lambda function. This role was configured with the minimum necessary permissions, following the security best practice of "least privilege." It was granted access only to read from the specific S3 bucket, invoke the necessary AI services, and write to the designated DynamoDB table, thereby securing the application's cloud infrastructure.

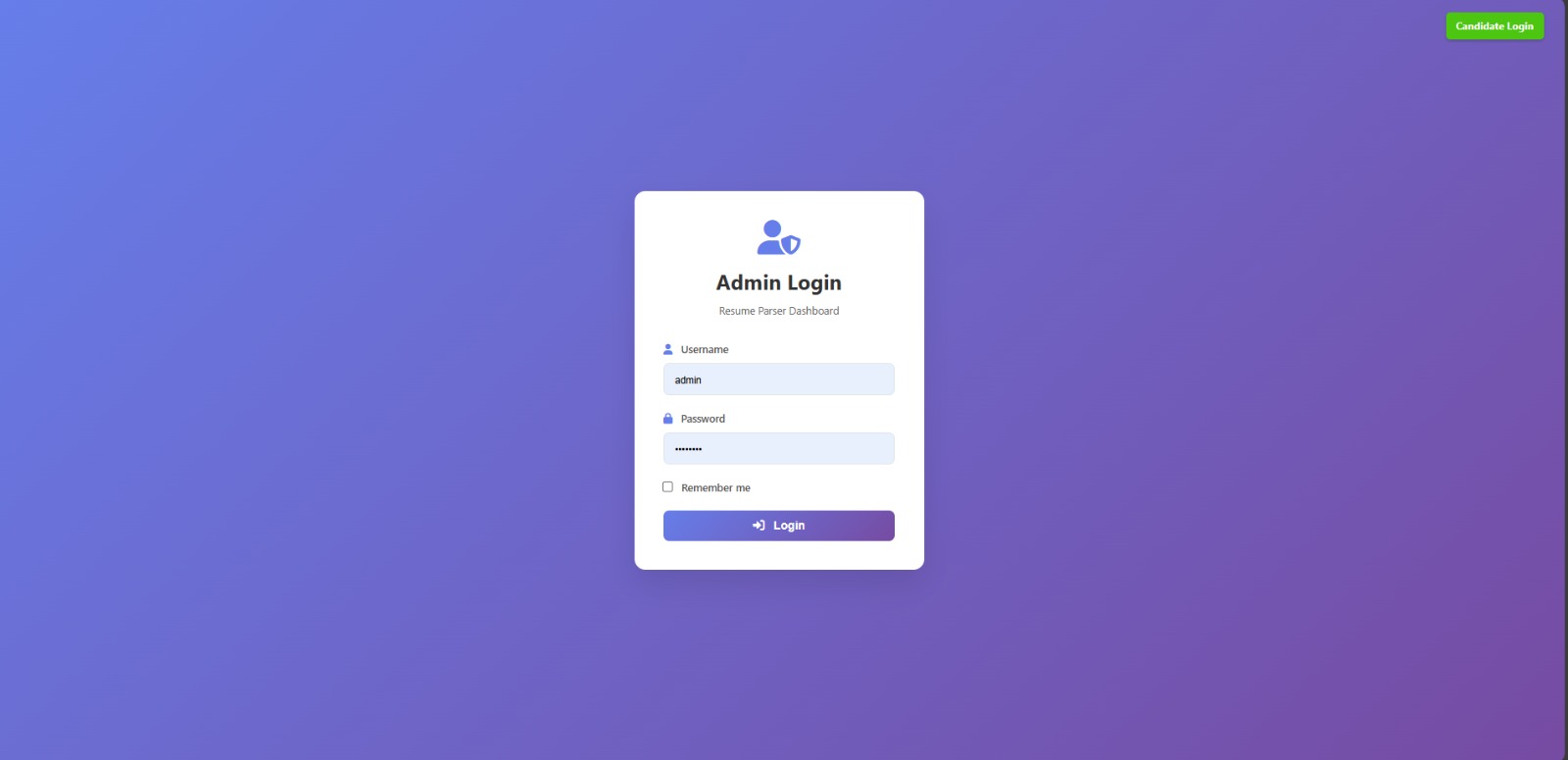
**5. Future Scope & Improvements**

* **Enhance NLP Capabilities**Integrate more advanced NLP services to perform deeper analysis, such as sentiment analysis on cover letters, categorizing job titles into standardized roles, or identifying the tone of professional summaries.
* **Implement Advanced Search Functionality**Deploy a dedicated search service like **Amazon OpenSearch**. This would allow recruiters to perform complex, full-text searches for candidates by skills, years of experience, education, and other criteria across the entire database of parsed resumes, significantly improving the search experience.
* **Support for Additional Resume Formats**Expand the text extraction Lambda function to support more file formats beyond PDF, such as .docx, .doc, and .txt, making the upload process more flexible for candidates.
* **OCR Implementation for Image-Based Resumes**Incorporate an Optical Character Recognition (OCR) service, such as **Amazon Textract**, to process resumes that are submitted as images (e.g., .jpg, .png). This would capture candidates who may scan their resumes or submit them from mobile devices.
* **Automated Candidate Ranking**  
  Develop a scoring algorithm that automatically ranks candidates based on how well their extracted skills and experience match a given job description. This would allow recruiters to instantly prioritize the most promising applicants.
* **CI/CD Pipeline Integration**  
  Implement a full CI/CD pipeline using a service like **AWS CodePipeline** or **GitHub Actions**. This would automate testing and deployment for both the Lambda functions and the Flask web application, leading to faster and more reliable updates.
* **Containerize the Web Application**  
  Package the Flask web application into a **Docker container** and deploy it on a scalable service like **AWS Fargate** or **Amazon ECS**. This would provide better portability, scalability, and resilience compared to a single EC2 instance.

**6. Output Screenshots**



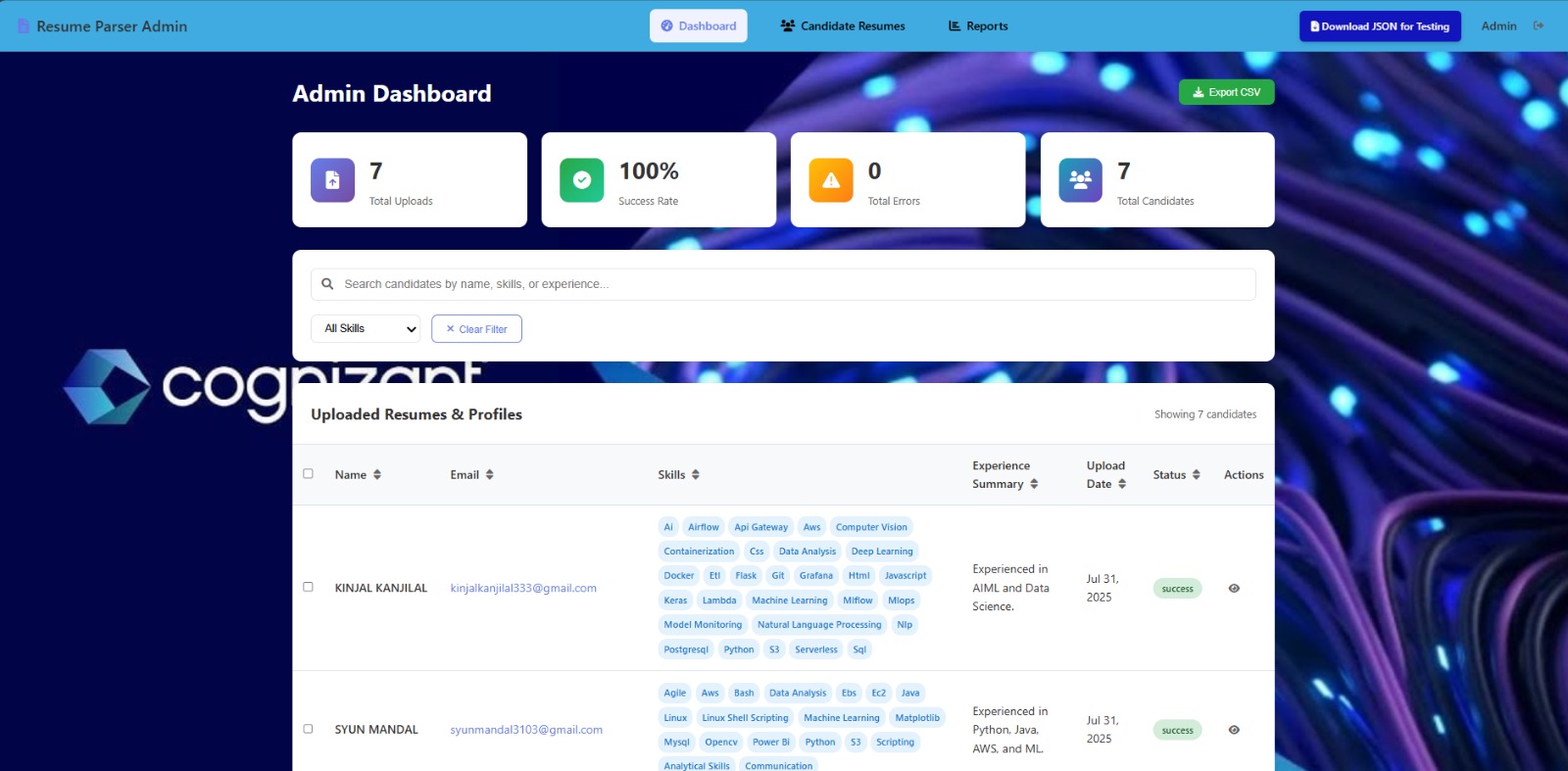
**Fig.1: Candidate Resume Upload page**



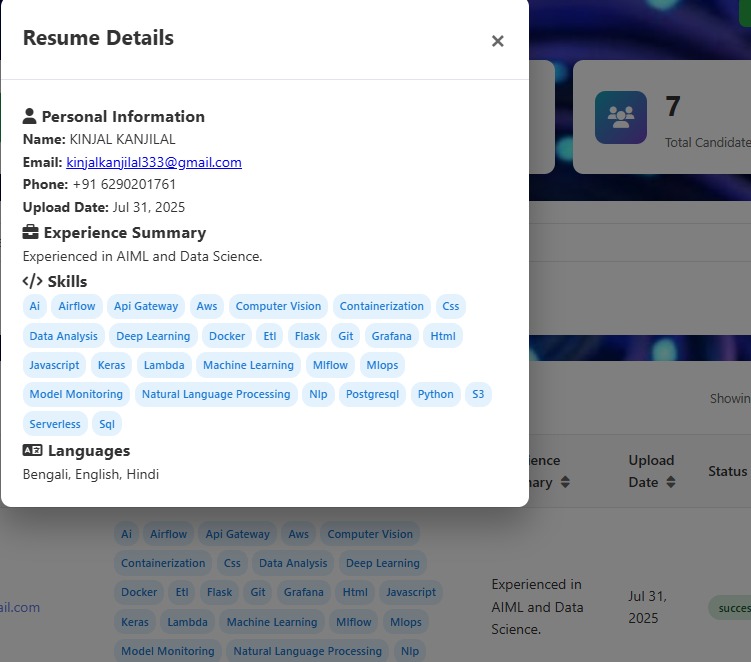
**Fig.2: Admin Login page**



**Fig.3: Can download data in .csv or .json [for testing] formats**



**Fig.4: Home page for admin**



**Fig.5: Sample data card of a candidate**



**Fig.6: Can view original resumes uploaded**

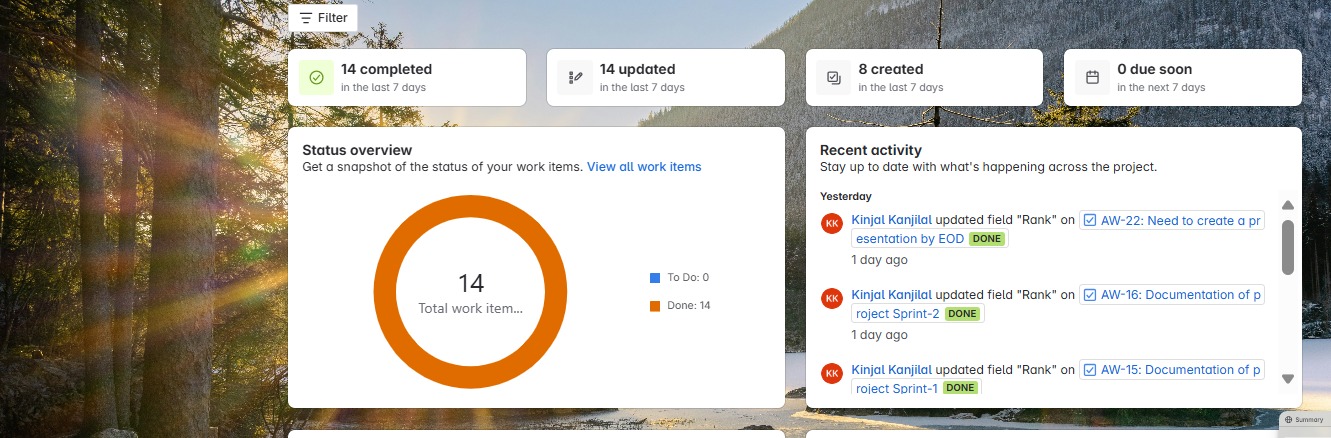


**Fig.7: Analytics page for all resumes [part-1]**

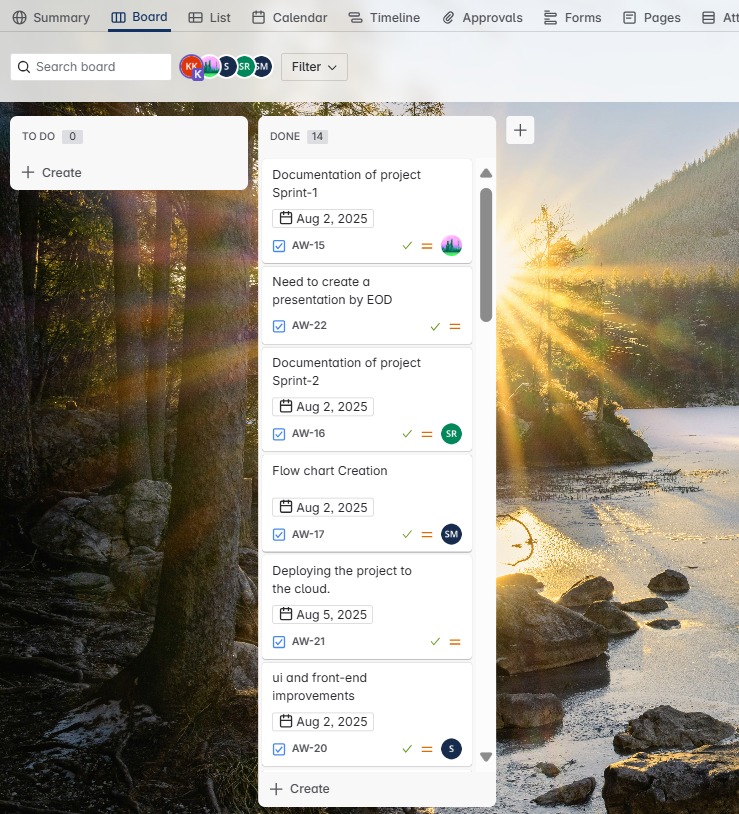


**Fig.8: Analytics page for all resumes [part-2]**

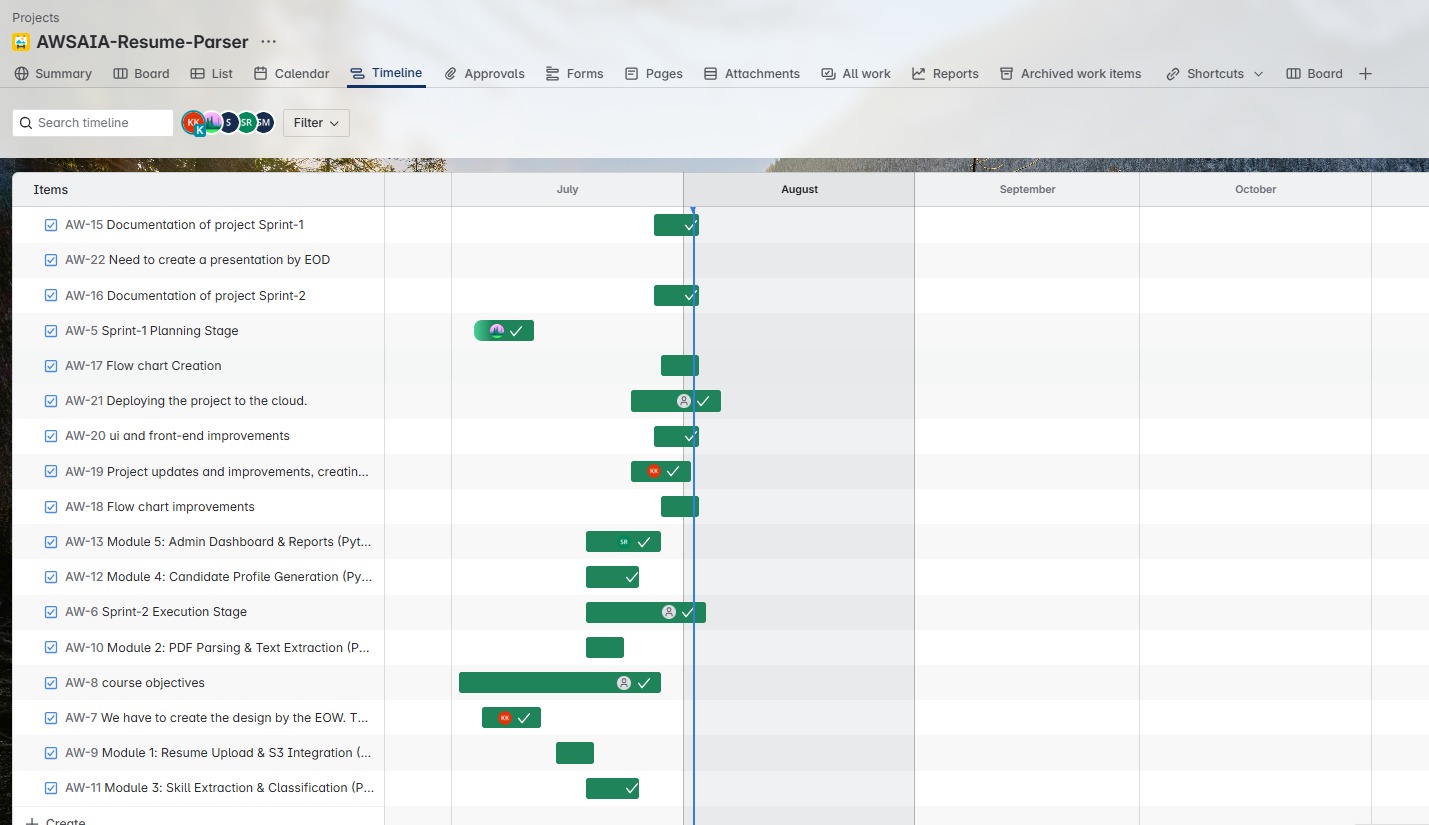
**7. JIRA Timeline**



**Fig.9: Jira Dashboard**



**Fig.10: Jira Board**



**Fig.11: Jira Timeline**