Runbook Document

EduBuk

Designed By

Rapyder Cloud Solutions Pvt Ltd

Bangalore | Delhi | Mumbai

2025

1 Contents

2 Introduction

In the evolving landscape of AI-driven recruitment, streamlining and enhancing processes is crucial for operational efficiency and effective candidate evaluation. Our solution leverages advanced AI capabilities within a robust AWS serverless architecture to deliver a comprehensive system that automates job description and resume ingestion, semantic matching, and intelligent filtering for a seamless recruitment experience.

The GenAI JD-Resume Matching System utilizes Amazon Bedrock models to extract metadata and generate vector embeddings from both job descriptions (uploaded as or JSON) and candidate resumes (uploaded as JSON). These embeddings and metadata are indexed in Amazon OpenSearch, enabling precise, context-aware matching and ranking of candidates against job requirements. Recruiters interact with the system through an intuitive application UI, where they can upload documents, search, and apply advanced filters—such as skills, experience, and location—to efficiently shortlist the most suitable candidates.

All documents are securely archived in Amazon S3, ensuring data integrity and traceability. The entire workflow is orchestrated using AWS Lambda and API Gateway, providing a scalable, serverless, and cost-effective solution. This integrated approach optimizes recruitment workflows, reduces manual screening effort, and empowers recruiters with actionable insights for better hiring decisions.

3 Project Overview

The customer seeks a solution to automate and enhance the job description and resume matching process using Generative AI (GenAI) on AWS. The objective is to streamline candidate evaluation, enable efficient recruiter workflows, and reduce manual effort through intelligent automation and semantic search. Key elements of the solution include:

1. **GenAI JD-Resume Matching System**
   An AI-powered backend ingests job descriptions (JSON) and candidate resumes (JSON), extracts relevant metadata and skills using Amazon Bedrock, and generates vector embeddings for both. These are indexed in Amazon OpenSearch, enabling recruiters to efficiently match resumes to job descriptions based on contextual relevance, skills, experience, and location. The system provides ranked candidate lists and supports advanced filtering for more precise shortlisting
2. **Intelligent Tagging and Filtering**
   The solution automatically tags both job descriptions and resumes with extracted metadata (role, skills, experience, location, etc.), allowing recruiters to apply fine-grained filters and efficiently surface the most suitable candidates from large talent pools.
3. **Scalable, Serverless AWS Architecture**
   The entire workflow is orchestrated using AWS Lambda, API Gateway, Bedrock, OpenSearch, and S3, ensuring high scalability, secure document storage, and cost-effective operations. All data is securely archived in S3 and indexed for rapid semantic search.

4 Technical Implementation

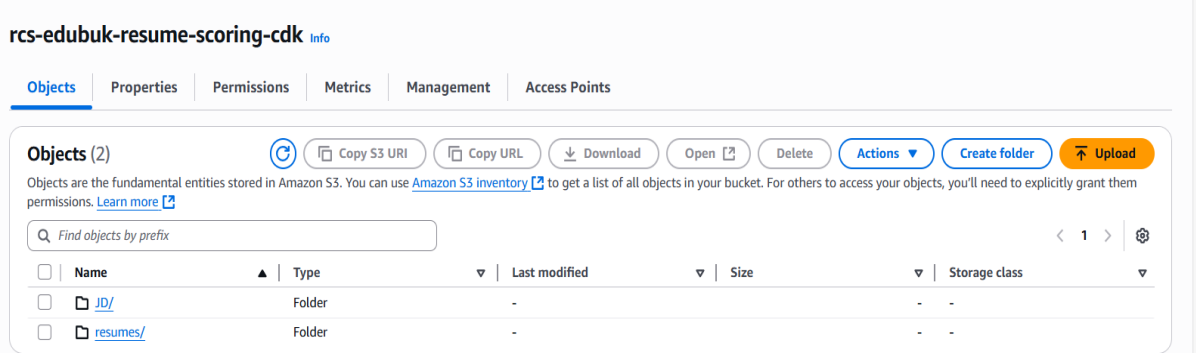
4.1 Resource Creation

This section covers broadly all the necessary resources created during the project:

4.1.1 Creation of S3 bucket

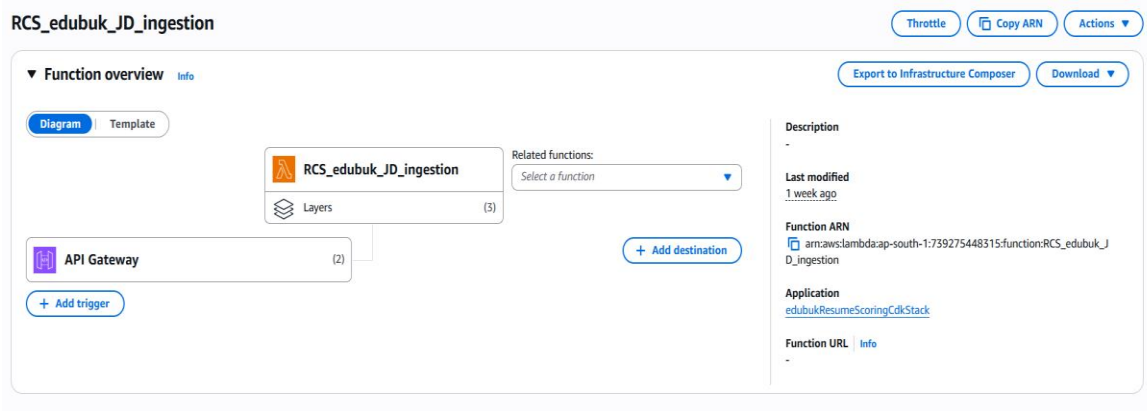
1. Signed in to the AWS Management Console and opened the Amazon S3 console

2. In the left navigation pane, choose Buckets.
3. Choose Create bucket; the Create bucket page opened.
4. For Bucket name, enter a name for the bucket.
5. For Region, chose the AWS Region (Mumbai(ap-south-1)) where the bucket resided.
6. Chose Create bucket.
7. Clicked on the created S3 bucket. Clicked on Properties in the Event notification section to create an event notification. Provided a notification name and specified the destination.



4.1.2 Creation of AWS Lambda.

1. Went to the AWS Management Console.
2. Navigated to the AWS Lambda Function.
3. Clicked on “Create function.”
4. Choose Author from scratch. In Basic Information, provided the Function name, Runtime Language, and Architecture.
5. Granted the required permissions to the Lambda Function.
6. “Create function.”
7. In Configuration, changed the General Configuration. Clicked on edit and changed the Timeout and Memory settings.



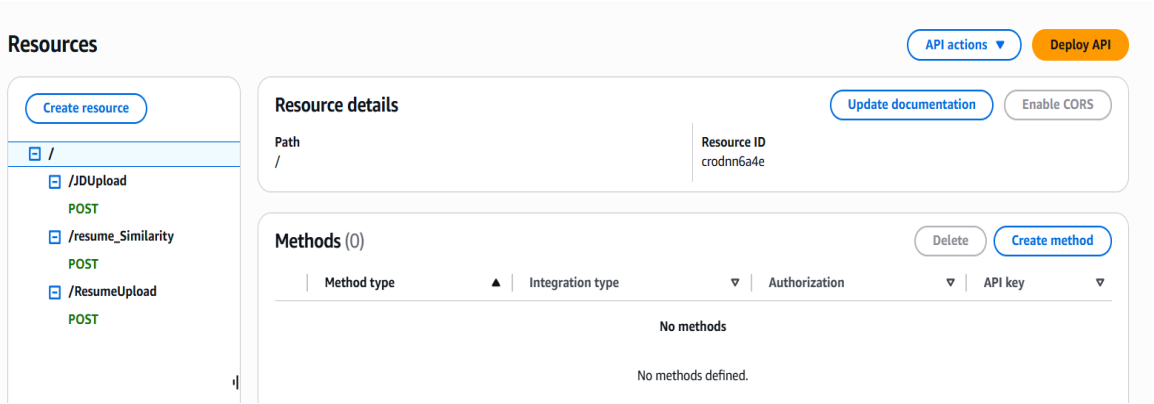
4.1.3 Requesting and accessing Amazon Bedrock.

1. Went to the AWS Management Console.
2. Navigated to Bedrock.
3. Requested access to the Anthropic Claude 3 Haiku Model.
4. Clicked on Model Access in the Bedrock Console.

5. In the Model Access Console, clicked on manage model access.
6. Clicked on the Anthropic Claude 3 Haiku model, then submitted the use case details.
7. Once the requested model was accessed, select the model from the AWS Bedrock Console.

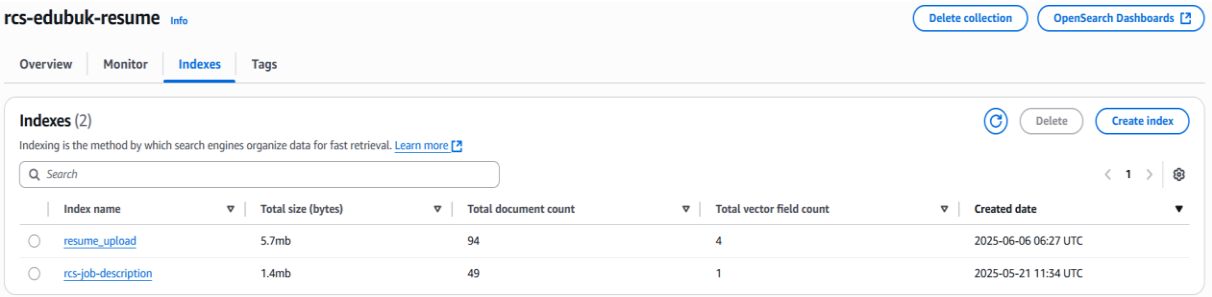
4.1.4 Creation of AWS API Gateway

1. Went to the AWS Management Console.
2. Navigated to “API Gateway.”
3. In the Amazon API Gateway Console, clicked on “Create API.”
4. Chose “REST API” and selected “Build.”
5. Entered a name for the API and configured the settings (e.g., endpoint type).
6. Clicked on “Create API.”
7. Defined resources and methods (e.g., POST) under the created API.
8. Configured the integration settings (e.g., Lambda function).
9. Deployed the API by creating a new deployment stage (e.g., “dev”) and clicked on “Deploy” to finalize.



4.1.5 Creation of Amazon OpenSearch

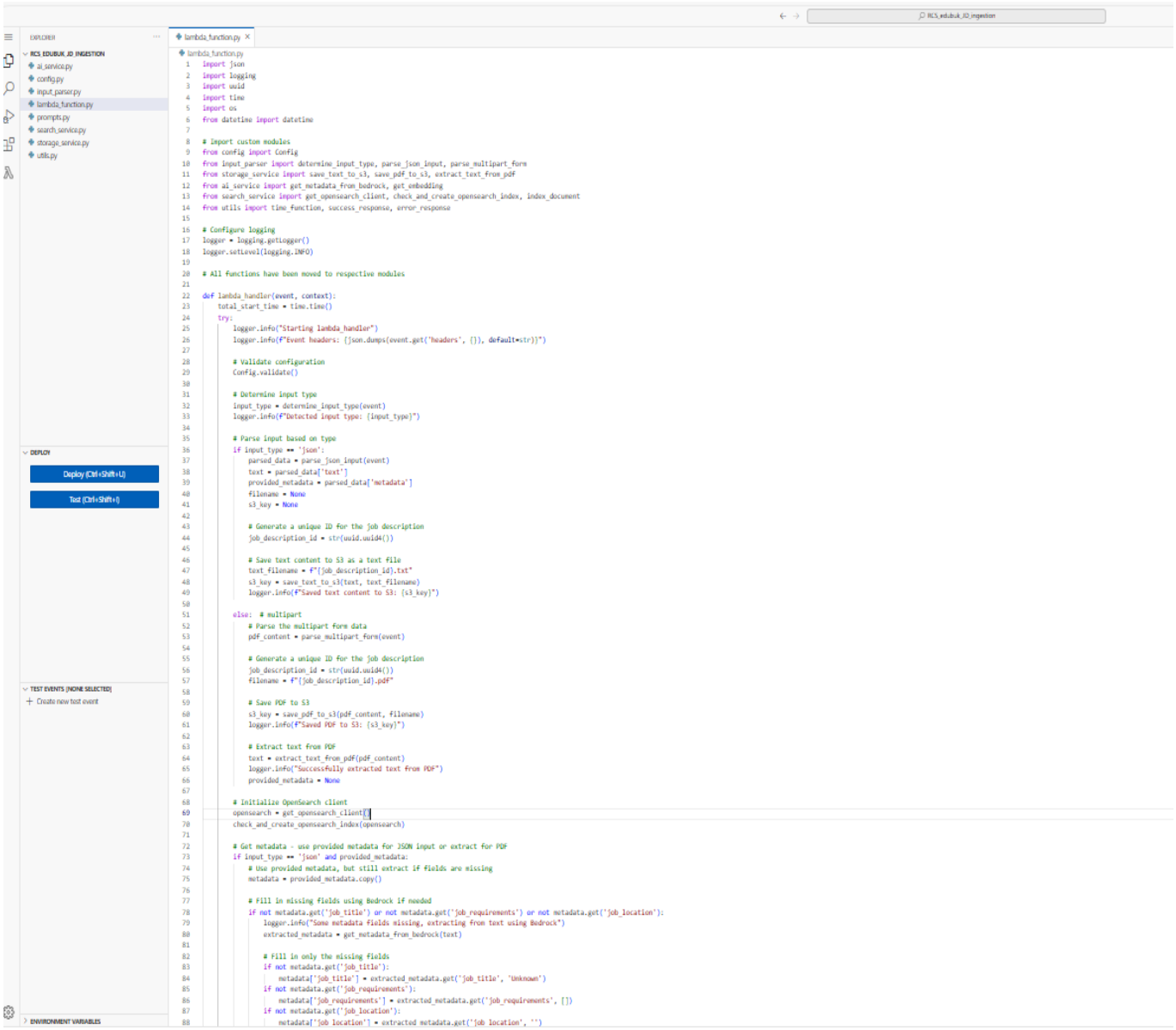
1. In AWS Management Console, we navigated to the Amazon OpenSearch Service console and expanded the Serverless section to select Collections.
2. Clicked on "Create collection," provided a unique name (3-32 characters, lowercase letters, numbers, and hyphens), and added a description.
3. Chose the collection type (Search, Time series, or Vector search) based on the data needs.
4. Decided on redundancy settings and selected an AWS KMS key for data encryption.
5. Specified the access type (public or private), set VPC endpoints, and defined resource types for access through OpenSearch and/or Dashboards endpoints.
6. Added IAM roles or SAML users for data access, granted permissions, and either created a new data access policy or added to an existing one, then reviewed and submitted the configuration.
7. Once submitted, the collection was created.



5 Code Explanation:

In This will provide code explanation of all lambda functions.

5.1 Explanation of Lambda Code RCS_edubuk_JD_ingestion



```python
import json
import logging
import uuid
import time
import os
from datetime import datetime

# Import custom modules
from config import Config
from input_parser import determine_input_type, parse_json_input, parse_multipart_form
from storage_service import save_text_to_s3, save_pdf_to_s3, extract_text_from_pdf
from ai_service import get_metadata_from_bedrock, get_embedding
from search_service import get_opensearch_client, check_and_create_opensearch_index, index_document
from utils import time_function, success_response, error_response

# Configure logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)

# All functions have been moved to respective modules

def lambda_handler(event, context):
    total_start_time = time.time()
    try:
        logger.info("Starting lambda_handler")
        logger.info(f"Event headers: {json.dumps(event.get('headers', {}), default=str)}")

        # Validate configuration
        Config.validate()

        # Determine input type
        input_type = determine_input_type(event)
        logger.info(f"Detected input type: {input_type}")

        # Parse input based on type
        if input_type == 'json':
            parsed_data = parse_json_input(event)
            text = parsed_data['text']
            provided_metadata = parsed_data['metadata']
            filename = None
            s3_key = None

            # Generate a unique ID for the job description
            job_description_id = str(uuid.uuid4())

            # Save text content to S3 as a text file
            text_filename = f"{job_description_id}.txt"
            s3_key = save_text_to_s3(text, text_filename)
            logger.info(f"Saved text content to S3: {s3_key}")

        else:  # multipart
            # Parse the multipart form data
            pdf_content = parse_multipart_form(event)

            # Generate a unique ID for the job description
            job_description_id = str(uuid.uuid4())
            filename = f"{job_description_id}.pdf"

            # Save PDF to S3
            s3_key = save_pdf_to_s3(pdf_content, filename)
            logger.info(f"Saved PDF to S3: {s3_key}")

            # Extract text from PDF
            text = extract_text_from_pdf(pdf_content)
            logger.info("Successfully extracted text from PDF")
            provided_metadata = None

        # Initialize OpenSearch client
        opensearch = get_opensearch_client()
        check_and_create_opensearch_index(opensearch)

        # Get metadata - use provided metadata for JSON input or extract for PDF
        if input_type == 'json' and provided_metadata:
            # Use provided metadata, but still extract if fields are missing
            metadata = provided_metadata.copy()

            # Fill in missing fields using Bedrock if needed
            if not metadata.get('job_title') or not metadata.get('job_requirements') or not metadata.get('job_location'):
                logger.info("Some metadata fields missing, extracting from text using Bedrock")
                extracted_metadata = get_metadata_from_bedrock(text)

                # Fill in only the missing fields
                if not metadata.get('job_title'):
                    metadata['job_title'] = extracted_metadata.get('job_title', 'Unknown')
                if not metadata.get('job_requirements'):
                    metadata['job_requirements'] = extracted_metadata.get('job_requirements', [])
                if not metadata.get('job_location'):
                    metadata['job_location'] = extracted_metadata.get('job_location', '')
```

Code Explanation:

Purpose: This pipeline automates the ingestion and processing of job descriptions submitted as JSON payloads or PDF files via application UI. The system parses the input, extracts relevant fields, generates embeddings using Amazon Bedrock, and indexes the data in OpenSearch for semantic search and matching, while also archiving the input in Amazon S3.

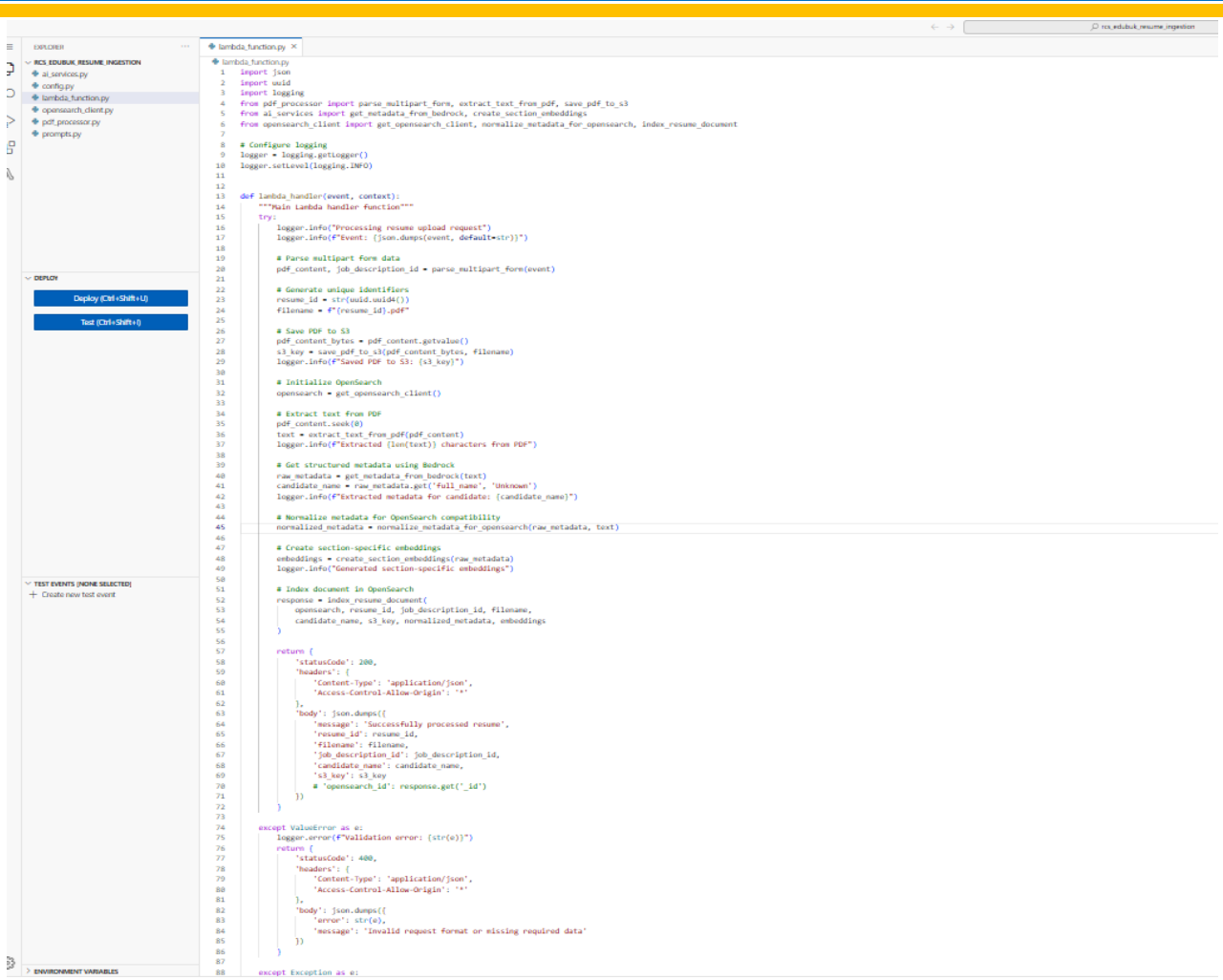
1. Imports and Setup

The Lambda function imports required libraries, including boto3 for AWS interactions, opensearchpy for OpenSearch integration, and standard Python utilities for logging, encoding, and timing. AWS clients for

S3 and Bedrock are initialized, and key configuration values such as the S3 bucket name and JD prefixes are set through the Config class.

2. **Input Type Determination**
   The function checks the incoming request's Content-Type header and body to determine if the input is JSON or multipart form data (PDF). For JSON ingestion, the system expects a payload containing fields such as job_description, job_title, job_requirements, and job_location.
3. **JSON Parsing**
   The Lambda function parses the JSON body, validates required fields (e.g., job_description), and extracts optional metadata (job_title, job_requirements, location). If the payload is base64-encoded, it is decoded before processing.
4. **S3 Storage**
   The job description text or PDF file is saved to Amazon S3 under a unique filename (using UUID), providing secure archival and traceability for each job posting. For PDFs, the text is extracted using PyPDF2.
5. **Metadata Extraction**
   If additional metadata extraction is required, the job description text is sent to Amazon Bedrock's Claude model for further enrichment (e.g., extracting structured job requirements or location from the text). Otherwise, provided metadata from the JSON is used directly.
6. **Embedding Generation**
   The job description text is sent to Bedrock's Titan embedding model to generate a high-dimensional vector embedding, enabling semantic search and matching against candidate resumes in OpenSearch.
7. **OpenSearch Indexing**
   The Lambda function initializes an OpenSearch client, ensures the job description index exists with proper KNN vector mappings, and then indexes a document containing the job description text, extracted metadata, embedding, and a unique Job Description ID.
8. **Logging and Monitoring**
   All major steps are logged for observability and debugging, with performance timing for key operations through the time_function utility.
9. **Lambda Handler Orchestration**
   The Lambda handler orchestrates the full workflow: parses and stores the input (JSON/PDF), extracts or uses metadata, generates embeddings, and indexes the data in OpenSearch. On success, it returns the Job Description ID and S3 path; on error, it returns a descriptive message using error_response utility.

5.2 Explanation of Lambda Code rcs_edubuk_resume_ingestion



```python
import json
import uuid
import logging
from pdf_processor import parse_multipart_form, extract_text_from_pdf, save_pdf_to_s3
from ai_services import get_metadata_from_bedrock, create_section_embeddings
from opensearch_client import get_opensearch_client, normalize_metadata_for_opensearch, index_resume_document

# Configure logging
logger = logging.getLogger()
logger.setLevel(logging.INFO)


def lambda_handler(event, context):
    """Main Lambda handler function"""
    try:
        logger.info("Processing resume upload request")
        logger.info(f"Event: {json.dumps(event, default=str)}")

        # Parse multipart form data
        pdf_content, job_description_id = parse_multipart_form(event)

        # Generate unique identifiers
        resume_id = str(uuid.uuid4())
        filename = f"{resume_id}.pdf"

        # Save PDF to S3
        pdf_content_bytes = pdf_content.getvalue()
        s3_key = save_pdf_to_s3(pdf_content_bytes, filename)
        logger.info(f"Saved PDF to S3: {s3_key}")

        # Initialize OpenSearch
        opensearch = get_opensearch_client()

        # Extract text from PDF
        pdf_content.seek(0)
        text = extract_text_from_pdf(pdf_content)
        logger.info(f"Extracted {len(text)} characters from PDF")

        # Get structured metadata using Bedrock
        raw_metadata = get_metadata_from_bedrock(text)
        candidate_name = raw_metadata.get('full_name', 'Unknown')
        logger.info(f"Extracted metadata for candidate: {candidate_name}")

        # Normalize metadata for OpenSearch compatibility
        normalized_metadata = normalize_metadata_for_opensearch(raw_metadata, text)

        # Create section-specific embeddings
        embeddings = create_section_embeddings(raw_metadata)
        logger.info("Generated section-specific embeddings")

        # Index document in OpenSearch
        response = index_resume_document(
            opensearch, resume_id, job_description_id, filename,
            candidate_name, s3_key, normalized_metadata, embeddings
        )

        return {
            'statusCode': 200,
            'headers': {
                'Content-Type': 'application/json',
                'Access-Control-Allow-Origin': '*'
            },
            'body': json.dumps({
                'message': 'Successfully processed resume',
                'resume_id': resume_id,
                'filename': filename,
                'job_description_id': job_description_id,
                'candidate_name': candidate_name,
                's3_key': s3_key
                # 'opensearch_id': response.get('_id')
            })
        }

    except ValueError as e:
        logger.error(f"Validation error: {str(e)}")
        return {
            'statusCode': 400,
            'headers': {
                'Content-Type': 'application/json',
                'Access-Control-Allow-Origin': '*'
            },
            'body': json.dumps({
                'error': str(e),
                'message': 'Invalid request format or missing required data'
            })
        }

    except Exception as e:
```

Code Explanation:

Purpose: This pipeline processes PDF resumes uploaded through a multipart form submission. The system extracts text from PDFs, uses AI to structure the information, generates embeddings using Amazon Bedrock, and indexes the data in OpenSearch for semantic search capabilities, while storing the original PDFs in Amazon S3.

1. **Imports and Setup**

   The application utilizes key Python libraries including PyPDF2 for PDF processing, boto3 for AWS service interactions, and opensearchpy for OpenSearch operations. The configuration is centralized in config.py , managing AWS credentials, endpoints, and processing limits.

2. **Input Type Determination**

   The system processes multipart form data from API Gateway, specifically handling PDF file uploads and job description IDs. The parse_multipart_form function in pdf_processor.py validates the Content-Type header and extracts the form boundary for proper parsing.

3. **PDF Processing**

   The application processes the uploaded PDF file, handling both raw and base64-encoded content. It validates the presence of required fields (PDF file and job description ID) and properly segments multipart form data sections.

4. **S3 Storage**

   Each PDF is stored in S3 with a unique UUID-based filename under the configured resume prefix. This ensures secure storage and easy retrieval of original documents for each candidate.

5. **Metadata Extraction**

The system uses Amazon Bedrock's Claude model to extract structured information from resume text, following strict rules for information extraction. It processes various sections including personal details, skills, experience, and education, ensuring only explicitly stated information is captured.

6. Embedding Generation

The application generates section-specific embeddings using Amazon Titan model, enabling semantic search capabilities for different resume components (skills, experience, projects, certifications).

7. OpenSearch Indexing

The Lambda function initializes an OpenSearch client, The system maintains a sophisticated OpenSearch index with Multiple vector fields for different resume sections embedding, and a unique Resume ID. This makes the resume efficiently searchable and matchable against job descriptions.

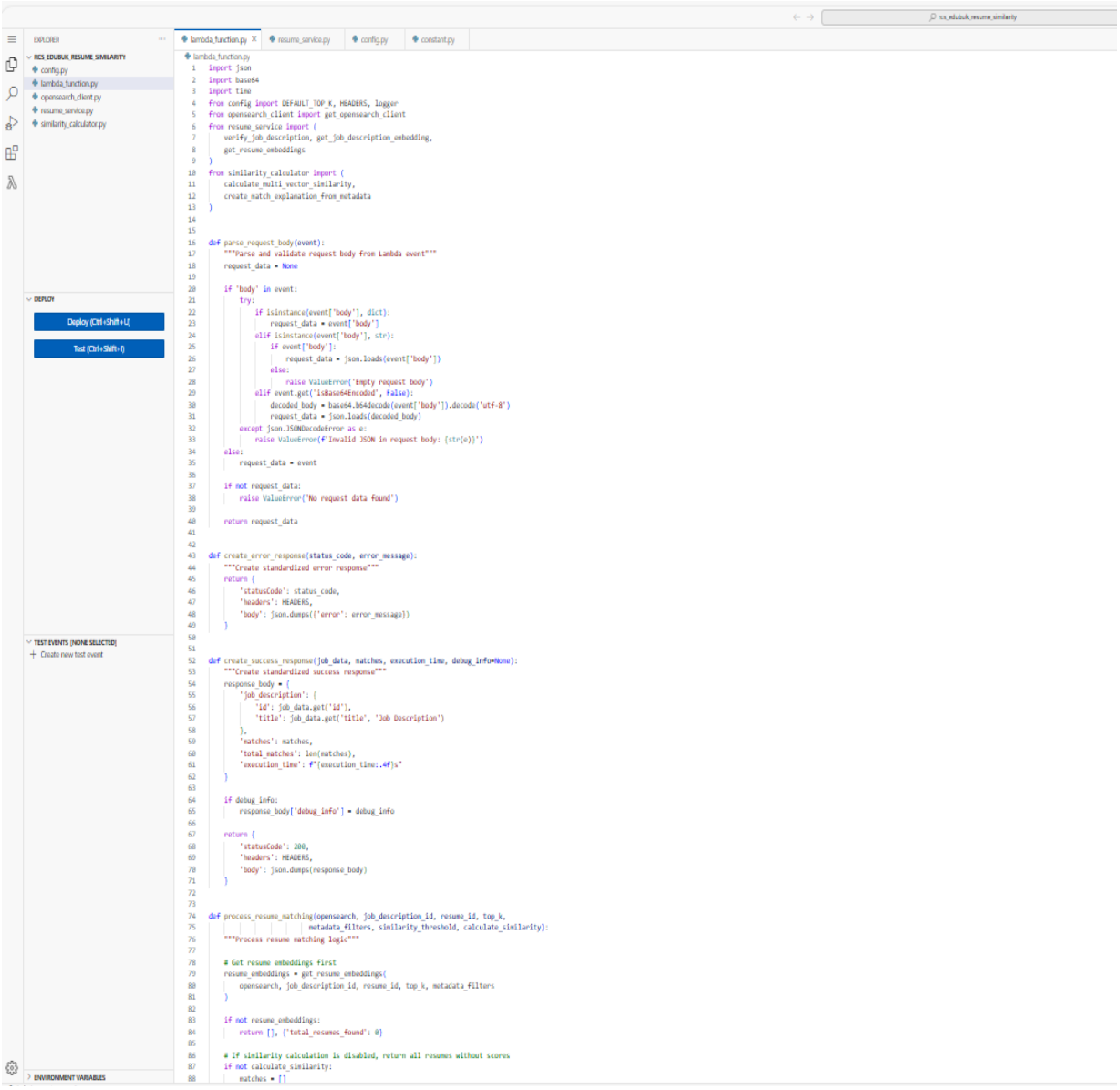
8. Logging and Monitoring

All major operations are logged for observability and debugging, with timing information for performance monitoring.

9. Lambda Handler Orchestration

The Lambda handler orchestrates the workflow: it determines input type, parses and stores the JSON input, extracts or uses metadata, generates embeddings, and indexes the data in OpenSearch. On success, it returns the Resume ID and S3 path; on error, it returns a descriptive error message.

5.3 Explanation of Lambda Code rcs_edubuk_resume_similarity



```
import json
import base64
import time
from config import DEFAULT_TOP_K, HEADERS, logger
from opensearch_client import get_opensearch_client
from resume_service import (
    verify_job_description, get_job_description_embedding,
    get_resume_embeddings
)
from similarity_calculator import (
    calculate_multi_vector_similarity,
    create_match_explanation_from_metadata
)


def parse_request_body(event):
    """Parse and validate request body from Lambda event"""
    request_data = None

    if 'body' in event:
        try:
            if isinstance(event['body'], dict):
                request_data = event['body']
            elif isinstance(event['body'], str):
                if event['body']:
                    request_data = json.loads(event['body'])
                else:
                    raise ValueError('Empty request body')
            elif event.get('isBase64Encoded', False):
                decoded_body = base64.b64decode(event['body']).decode('utf-8')
                request_data = json.loads(decoded_body)
        except json.JSONDecodeError as e:
            raise ValueError(f'Invalid JSON in request body: {str(e)}')
    else:
        request_data = event

    if not request_data:
        raise ValueError('No request data found')

    return request_data


def create_error_response(status_code, error_message):
    """Create standardized error response"""
    return {
        'statusCode': status_code,
        'headers': HEADERS,
        'body': json.dumps({'error': error_message})
    }


def create_success_response(job_data, matches, execution_time, debug_info=None):
    """Create standardized success response"""
    response_body = {
        'job_description': {
            'id': job_data.get('id'),
            'title': job_data.get('title', 'Job Description')
        },
        'matches': matches,
        'total_matches': len(matches),
        'execution_time': f"{execution_time:.4f}s"
    }

    if debug_info:
        response_body['debug_info'] = debug_info

    return {
        'statusCode': 200,
        'headers': HEADERS,
        'body': json.dumps(response_body)
    }


def process_resume_matching(opensearch, job_description_id, resume_id, top_k,
                            metadata_filters, similarity_threshold, calculate_similarity):
    """Process resume matching logic"""

    # Get resume embeddings first
    resume_embeddings = get_resume_embeddings(
        opensearch, job_description_id, resume_id, top_k, metadata_filters
    )

    if not resume_embeddings:
        return [], {'total_resumes_found': 0}

    # If similarity calculation is disabled, return all resumes without scores
    if not calculate_similarity:
        matches = []
```

Code Explanation:

Purpose: This system implements a sophisticated similarity calculation engine that processes multiple vectors (skills, experience, certifications, and projects) using NumPy-based computations. It leverages OpenSearch for efficient indexing and retrieval, while employing advanced normalization techniques for accurate matching of skills and locations.

1. **Imports and Setup**
   The system relies on several critical Python libraries to function efficiently. It uses NumPy for high-performance vector computations, the OpenSearch Python client for document indexing and retrieval, and standard libraries such as json and logging for data handling and diagnostics. Configuration parameters are managed through a centralized config.py file, simplifying both deployment and ongoing maintenance.
2. **Input Type Determination**
   Inputs to the similarity engine are handled through a standardized interface that expects well-structured vectors for different attributes such as skills, experience, certifications, and projects.

Each of these vectors is required to maintain consistent dimensions, ensuring the reliability and precision of the similarity calculations.

3. **JSON Processing**
   The system processes incoming data in JSON format, validating the structure and dimensional integrity of each vector. It also performs necessary data transformations to align inputs with the required formats, which is vital to maintaining the accuracy of similarity calculations, especially when data is heterogeneous.
4. **Vector Normalization**
   To achieve consistency across varying types of input data, all vectors undergo normalization prior to similarity analysis. This process ensures that each vector operates on the same scale, which is particularly important for calculating unbiased and meaningful similarity scores across different attributes.
5. **Metadata Processing**
   Intelligent metadata handling is implemented to enhance matching precision. This includes normalizing skill labels, flexible matching for locations to account for geographic variations, verifying levels of experience, and validating certification information to ensure they meet expected standards.
6. **Similarity Calculation**
   At its core, the system uses NumPy's highly optimized array operations to calculate cosine similarity between vectors. It supports the simultaneous processing of multiple vector types, allowing for a more comprehensive and nuanced assessment of similarity between profiles or documents**.**
7. **Score Aggregation**
   The results from the individual similarity computations are aggregated using a weighted averaging strategy. This approach assigns importance to each vector type based on its relevance and reliability, allowing the system to produce a well-balanced overall similarity score.
8. **Performance Optimization**
   To ensure high throughput and responsiveness, the system is optimized with several performance-enhancing techniques. These include batch processing for bulk comparisons, memory-efficient NumPy computations, and the caching of frequently reused calculations, which together enable the system to scale efficiently under load.
9. **Result Generation**
   The output includes a comprehensive similarity report consisting of an overall similarity score, individual scores for each vector type, explanatory insights into the matching process, and key performance metrics. This detailed result format aids in transparency and facilitates better decision-making based on the similarity analysis.

5.4 Adding a Lambda Layer Using AWS Management Console

Step 1: Create an EC2 Instance

1. **Log in to AWS Console:**
   - Go to the AWS Management Console.
   - Log in to your account.
2. **Navigate to EC2:**
   - Go to the EC2 Dashboard.
3. **Launch an Instance:**
   - Click on "Launch Instance."
   - Choose an Amazon Machine Image (AMI) (e.g., Amazon Linux 2).
   - Select an instance type.

- Configure instance details, add storage, add tags, configure security group, and review.

4. **Create a Key Pair:**
   - Create a new key pair or use an existing one. This key pair will be used to connect to the EC2 instance.
5. **Launch Instance:**
   - Click "Launch" and select your key pair.
   - Click "Launch Instances."
6. **Connect to the Instance:**
   - Use an SSH client to connect to your EC2 instance using the private key.

Step 2: Install Python 3.9 on EC2 Instance

1. **Update Packages:**
   - sudo yum update -y
2. **Install Python 3.9:**
   - sudo amazon-linux-extras install python3.9

Step 3: Install Python Library

1. **Install the Python library (replace <library_name> with the actual library name):**
   - sudo yum install python3.9-devel sudo /usr/bin/pip-3.9 install <library_name> -t /path/to/folder

Step 4: Create a ZIP Archive

1. **Navigate to the folder:**
   - cd /path/to/folder
2. **Create a ZIP archive:**
   - zip -r library_layer.zip .

Step 5: Upload to S3 Bucket

1. **Install AWS CLI (if not installed):**
   - sudo yum install aws-cli
2. **Configure AWS CLI with your credentials:**
   - aws configure
3. **Upload the ZIP file to S3:**
   - aws s3 cp library_layer.zip s3://your-s3-bucket/

5.5 Create layer in Lambda

- Create or Select a Function: Choose the Lambda function to which you want to add the layer. You can either create a new function or select an existing one from the list.
- Function Configuration: Scroll to the "Function code" section of your Lambda function's configuration page.
- Layers Section:
  - Click on "Layers", which is located below the Lambda function code editor.
    In the Layers panel, you will see a list of layers that are currently applied to the function, if any.
- Add a Layer:
  - Click on "Add a layer" to open the "Add layer" configuration pane.
    You have two options: choose a layer from a list of AWS-provided layers or specify an ARN for a custom layer.

- To use a custom layer, select "Provide a layer version ARN" and input the ARN of the layer you wish to add.
  Choose Layer Version: If you are adding a custom layer, you will also need to specify the version of the layer you want to use.
- Add Layer to Function: Once you have selected the layer and its version, click on "Add" to apply the layer to your Lambda function.
- Save Function Configuration: After adding the layer, you will be redirected back to the main configuration page of your Lambda function.
  Make sure to click on "Save" at the top of the page to apply the changes.
- Review and Test: After saving, it is good practice to test your Lambda function to ensure that it works as expected with the new layer.

6 Operational Runbook for critical resources:

Service	Error	Resolution	Documentation Link
Lambda	Concurrently Limit Reached	Increase the concurrent execution limit in Lambda console or via AWS CLI. Optimize function code to reduce concurrency spikes.	Reference
	Function Fails to Execute	Inspect CloudWatch logs for any error messages. Review function code and dependencies for issues. Check for expired or incorrect permissions. Redeploy function with fixes if necessary.	Reference
	Execution Environment Limit Reached	Configure the Execution Environments in Lambda Console or via AWS CLI. Change the parameters for any error messages and application requirement.	Reference
OpenSearch	High CPU Utilization	Review the output for hot threads, currently running tasks, thread pool statistics, and shard allocation information. Identify the cause of high CPU utilization from this data and take appropriate action to optimize your cluster's performance. Ensure all required IAM permissions are in place before using the runbook	Reference
	Indexing Failures	Check for errors in the indexing process through logs or monitoring tools. Ensure that your data mapping is correct and that there are no issues with data types. If bulk indexing fails, implement exponential backoff strategies to retry failed requests. Validate that all required fields are present in documents being indexed	Reference
	Throttling Issues	Implement rate limiting in your application code to prevent overwhelming the OpenSearch cluster with requests. Monitor request rates and adjust accordingly to avoid throttling situations.	Reference

	Slow Query Performance	Optimize query patterns by reviewing slow logs and identifying inefficient queries. Consider using filters instead of queries where applicable, and ensure that appropriate indexes are in place. Utilize OpenSearch's profiling tools to analyze query performance and make necessary adjustments	Reference
API Gateway	Integration Timeout Error	Increase the integration timeout setting in API Gateway for the affected endpoint. Review backend service performance to identify potential bottlenecks causing timeouts. Adjust the timeout value accordingly	Reference
	Rate Limit Exceeded	Review CORS settings in API Gateway to ensure they match the requirements of client applications. Update CORS headers as needed to allow cross-origin requests from permitted domains. Test CORS configuration using tools like curl or browser developer tools	Reference
	Invalid CORS Configuration	Review CORS settings in API Gateway to ensure they match the requirements of client applications. Update CORS headers as needed to allow cross-origin requests from permitted domains. Test CORS configuration using tools like curl or browser developer tools	Reference
	Authorization Failure	Verify IAM roles and policies associated with API Gateway resources and Lambda authorizers. Ensure correct permissions are granted for invoking backend services. Test authorization logic with different user roles to identify and resolve issues	Reference
Bedrock	Permission issues	If permission error occurred check IAM roles, and it should have permission of accessing the Bedrock model.	Reference
	Authorize to perform iam:passrole	If you receive an error that you're not authorized to perform the iam:PassRole action, your policies must be updated to allow you to pass a role to Amazon Bedrock.	Reference
	Token count exceeded	If you encounter an error similar to the preceeding example, make sure that the number of tokens conforms to the token quota.	Reference
	Character quota exceeded	If you encounter an error beginning with the text above, ensure that the number of characters conforms to the character quota.	Reference

7 Backup and Recovery:

1. **Lambda:**
   - **1.1 Code Retrieval Steps:**
     - **1.1.1** To retrieve your lambda function code, navigate to the AWS Lambda console, select your function, and review the version history to restore the deleted code.
     - **1.1.2** Alternatively, if you had previously stored your code in a version control system like Git, you could clone the repository and retrieve the code from there.
   - **1.2 Preventive Measures:**
     - **1.2.1** To prevent accidental deletions, enable versioning for your Lambda functions, which maintains a history of all versions and configurations.
     - **1.2.2** Implement strict IAM policies to control access permissions, limiting the number of users who can modify or delete functions. Additionally, establish a regular backup routine for your Lambda function code, either through automated scripts or manual backups to an external storage solution.
2. **S3 Bucket Deletion:**
   - **2.1 Data Retrieval Steps:**
     - **2.1.1** To retrieve your files stored in Amazon S3, access the AWS S3 console, navigate to the bucket containing the deleted files, and review the object versioning or delete marker history to restore the deleted files.
     - **2.1.2** Alternatively, if you had backups stored elsewhere or replicated your data to another S3 bucket or storage service, you can retrieve the files from there.
   - **2.2 Preventive Measures:**
     - **2.2.1** For preventive measures, enable versioning for your S3 buckets to maintain a history of object versions and prevent accidental deletions. Implement S3 bucket policies and IAM policies to control access permissions, limiting the number of users who can modify or delete objects within the buckets.
     - **2.2.2** Additionally, consider enabling MFA (Multi-Factor Authentication) delete on critical buckets to require additional authentication for deletion actions, and regularly back up your S3 data to another storage location or region for added redundancy and disaster recovery capabilities.
3. **API Gateway:**
   - **3.1 Data Retrieval Steps:**
     - **3.1.1** Re-Create API Gateway configurations, having routes, methods, and other integrations.
     - **3.1.2** Reconnect integrations with backend services, like Lambda functions.
     - **3.1.3** Update client applications so that it shows any API changes.
   - **3.2 Preventive Measures:**
     - **3.2.1** Regularly back up API Gateway configurations using AWS CloudFormation. Enable versioning for APIs to track changes and allow for rollbacks if necessary.
     - **3.2.2** Use AWS CloudTrail logging to monitor and audit API Gateway usage and configuration changes.

8 Reference Links and documentation:

Amazon S3: https://docs.aws.amazon.com/AmazonS3/latest/userguide/creating-bucket.html

Amazon Bedrock: https://aws.amazon.com/bedrock/
Anthropic Claude model: https://aws.amazon.com/bedrock/claude/
AWS Lambda: https://docs.aws.amazon.com/lambda/
Amazon API Gateway: https://docs.aws.amazon.com/apigateway/