****

**Name : Aman Jaglan**

**GWID: G45030269**

**Introduction:**

**Project Overview:**

When we first started working on this project, our goal was to build a system that could help align candidate resumes with job descriptions and give feedback on how to improve it. Over time, the scope expanded significantly, and we became involved in designing multiple features: analyzing similarity between resumes and job descriptions, generating resumes, providing visualization tools, ranking resumes based on their suitability for a given job, recommending relevant jobs to candidates, and evaluating grammar and formatting in resumes. The final product integrated all these components into a Streamlit application, creating a comprehensive platform where users can upload their resumes, paste job descriptions, and then explore a host of functionalities such as similarity scoring, skill gap analysis, resume generation, and other visualization techniques.

Initially, the project was simply about finding a good model for similarity and feedback with grammar checks and exploring how to compare a resume to a job description. But as I began adding more capabilities—like generating formatted resumes based on instructions and identifying missing skills in a candidate’s profile—we realized that each component influenced and complemented the others. For instance, the similarity score and feedback from skill gap analysis could guide improvements to the user to add missing skills, help highlight what needed to be added or improved in the resume. This interplay eventually led us to develop a more holistic system rather than just a single model or script.

**Outline of Shared Work:**

1. **Building Data Extraction Pipeline:**
2. **Data Folder**:
   * Contains the raw and processed data.
   * Subfolders include:
     + **Processed**: Likely holds the processed results, including cleaned or analyzed resumes.
     + **Resumes**: Contains individual resume files for processing.
3. **data extractor Module**:
   * Houses the core logic for the project.
   * Subdirectories include:
     + **parsers**: Contains scripts for parsing raw data, such as extracting text from PDFs or Word documents.
     + **utils**: A collection of utility scripts providing common functionalities like cleaning or transforming data.
4. **Python Scripts**:
   * **Extractor.py**: Handles the primary data extraction logic, possibly extracting raw text from resumes or job descriptions.
   * **JobDescriptionProcessor.py**: Focuses on processing job descriptions, extracting key requirements, skills, or keywords.
   * **ResumeProcessor.py**: Processes resumes to extract structured data like experience, skills, and education.
   * **TextCleaner.py**: Cleans and preprocesses text data, removing unnecessary characters or standardizing formats.
   * **test.py**: Likely used for testing individual components of the pipeline to ensure accuracy and reliability.
   * **main.py**: The entry point of the project, orchestrating the pipeline by calling various modules and managing the overall flow.

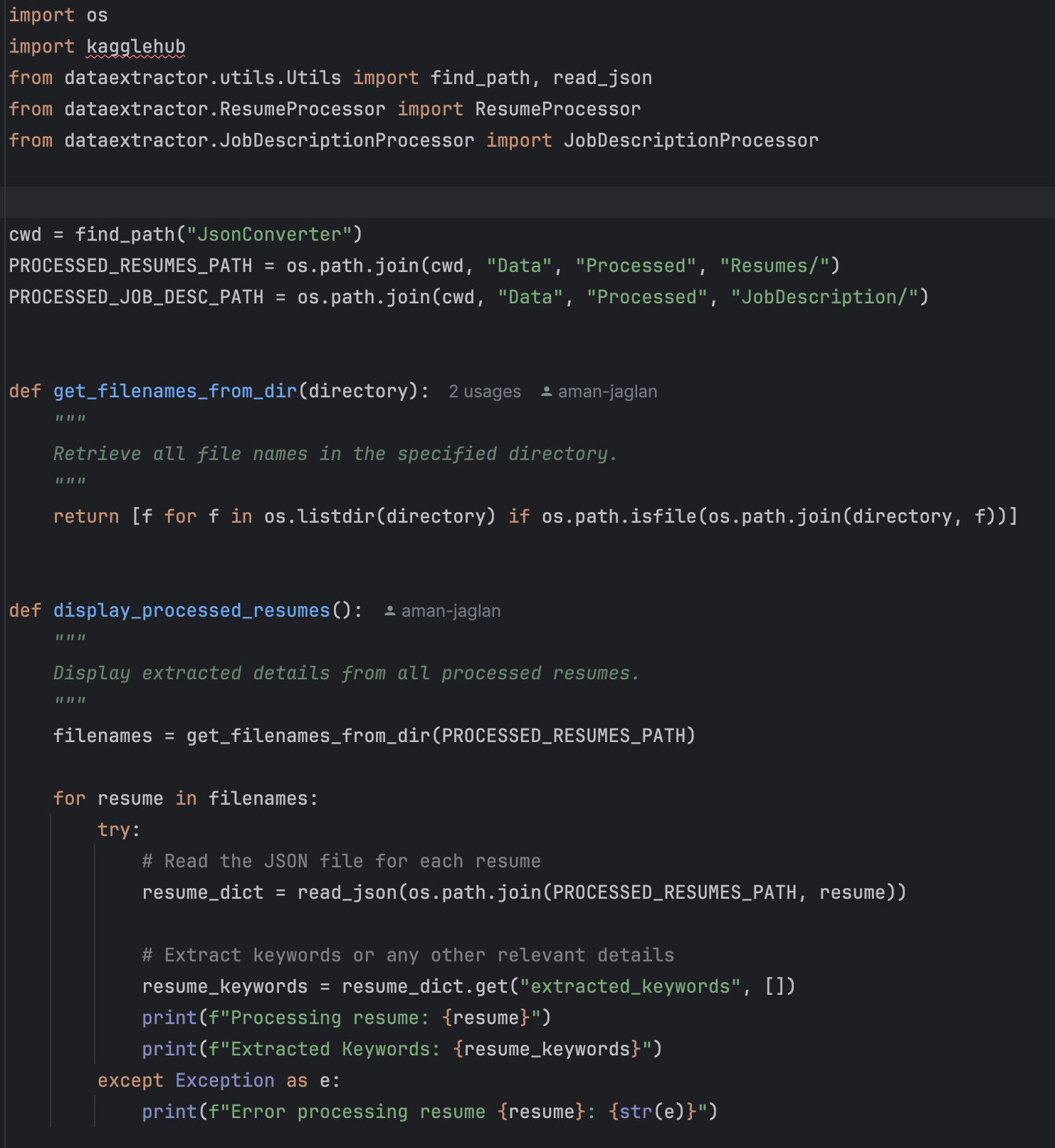
**Summary of Workflow:**

1. **Input**: Resumes and job descriptions are stored in the **Data** folder.
2. **Preprocessing**:
   * Raw files are parsed using parsers in the **dataextractor/parsers** folder.
   * Text is cleaned using **TextCleaner.py** to prepare for further processing.
3. **Data Extraction**:
   * **ResumeProcessor.py** and **JobDescriptionProcessor.py** extract relevant data fields from the resumes and job descriptions.
4. **Processing**:
   * Extracted data is analyzed or transformed to match requirements, such as matching skills or generating scores.
5. **Output**:
   * Processed data is stored in the **Data/Processed** folder for further use.

**Pipeline Diagram:**

A diagram of data extractor

Description automatically generated

**Main.py Code:   
A computer screen shot of a program

Description automatically generatedA computer screen shot of a program code

Description automatically generated**

### Pipeline Overview: ****ResumeJobScore****

The **ResumeJobScore** project processes resumes and job descriptions to compute a compatibility score. The structure indicates a modular approach with separate configurations, logging, and processing pipelines. Here's a detailed explanation:

**Pipeline Explanation**

**1. Configurations**

**config/config.yaml:**

Contains project settings, such as file paths, preprocessing parameters, and model hyperparameters.

Centralized control for modifying pipeline behavior.

**2. Input Data**

**data/:**

resumes/: Directory containing raw resumes.

job\_descriptions/: Directory with raw job descriptions.

processed/: Stores cleaned and preprocessed data.

**3. Logging**

logs/preprocessing.log:

Tracks pipeline activities, such as data loading, processing steps, and errors.

**4. Preprocessing**

src/preprocess/:

job\_desc\_processor.py: Processes job descriptions to extract relevant skills, responsibilities, and keywords.

resume\_processor.py: Extracts structured data (e.g., experience, skills) from resumes.

tokenizer.py: Tokenizes text for downstream tasks, such as vectorization or embedding generation.

**5. Utilities**

src/utils/:

model\_training.py: Handles the training and evaluation of machine learning models used for scoring.

pipeline.py: Manages the end-to-end workflow, integrating all processing modules.

init.py: Defines package-level imports and initializations.

**6. Main Script**

src/main.py:

Entry point of the pipeline.

Executes the entire process from data ingestion to model training and scoring.

**7. Output**

Results and compatibility scores are saved in the processed/ folder, ready for analysis or reporting.

**Diagram:**

**A diagram of a job

Description automatically generated**

**Main.py Code:**

**A screenshot of a computer program

Description automatically generated**

**Pipeline.py:**

**A screen shot of a computer program

Description automatically generated**

**A screen shot of a computer program

Description automatically generated**

**Introduction to the Grammar and Formatting Check Model**

The Grammar and Formatting Check Model leverages AWS Bedrock's AI capabilities to provide a detailed analysis of resumes. The model extracts text from PDF resumes, identifies major grammatical and formatting issues, and generates actionable recommendations to enhance professional presentation. The pipeline integrates text extraction, AI-driven analysis, and structured output parsing to deliver insights effectively.

**Pipeline Explanation**

**1. Initialization**

The BedrockResumeAnalyzer class is initialized with AWS credentials and a Bedrock runtime client.

AWS credentials include access keys, session tokens (if applicable), and the AWS region.

Logging is configured for tracking the execution flow and debugging errors.

**2. Text Extraction**

Method: extract\_text\_from\_pdf

Reads and extracts text from PDF resumes using the PyPDF2 library.

Cleans and combines text from all pages of the PDF.

Handles errors, such as missing files or issues during text extraction, with detailed logging.

**3. Resume Analysis**

Method: analyze\_resume\_text

Constructs a detailed prompt for analyzing the extracted resume text, focusing on:

Top grammatical errors

Major formatting inconsistencies

Recommendations for improvement

Numerical scores for grammar and formatting (out of 100).

Sends the prompt to AWS Bedrock's language model (e.g., Claude) for analysis.

Handles response parsing and error management for robustness.

**4. Output Parsing**

Method: \_parse\_resume\_analysis

Parses the AI model's response using regular expressions to extract:

Grammar and formatting scores

Lists of grammatical errors and formatting issues

Constructive recommendations.

Uses a helper function \_extract\_section for structured parsing of specific sections.

**5. Main Function**

Function: analyze\_resume

Orchestrates the entire pipeline:

Initializes the analyzer with AWS credentials.

Extracts text from the resume file.

Analyzes the text for grammar and formatting issues.

Returns structured results, including scores and improvement suggestions.

**Key Features**

**Comprehensive Analysis:**

Identifies grammatical errors and formatting issues.

Scores the resume's grammar and formatting to provide a quantitative assessment.

**Actionable Insights:**

Suggests improvements to enhance clarity, professionalism, and presentation.

**Error Handling:**

Robust exception handling for file-related and processing errors.

**AWS Bedrock Integration:**

Utilizes a state-of-the-art AI model (Claude) to deliver accurate and focused results.

**Example Output**

**Scores:**

Grammar Score: 85/100

Formatting Score: 90/100

**Top Grammatical Errors:**

Error: Incorrect verb tense in work experience section.

Error: Missing punctuation in project descriptions.

Formatting Issues:

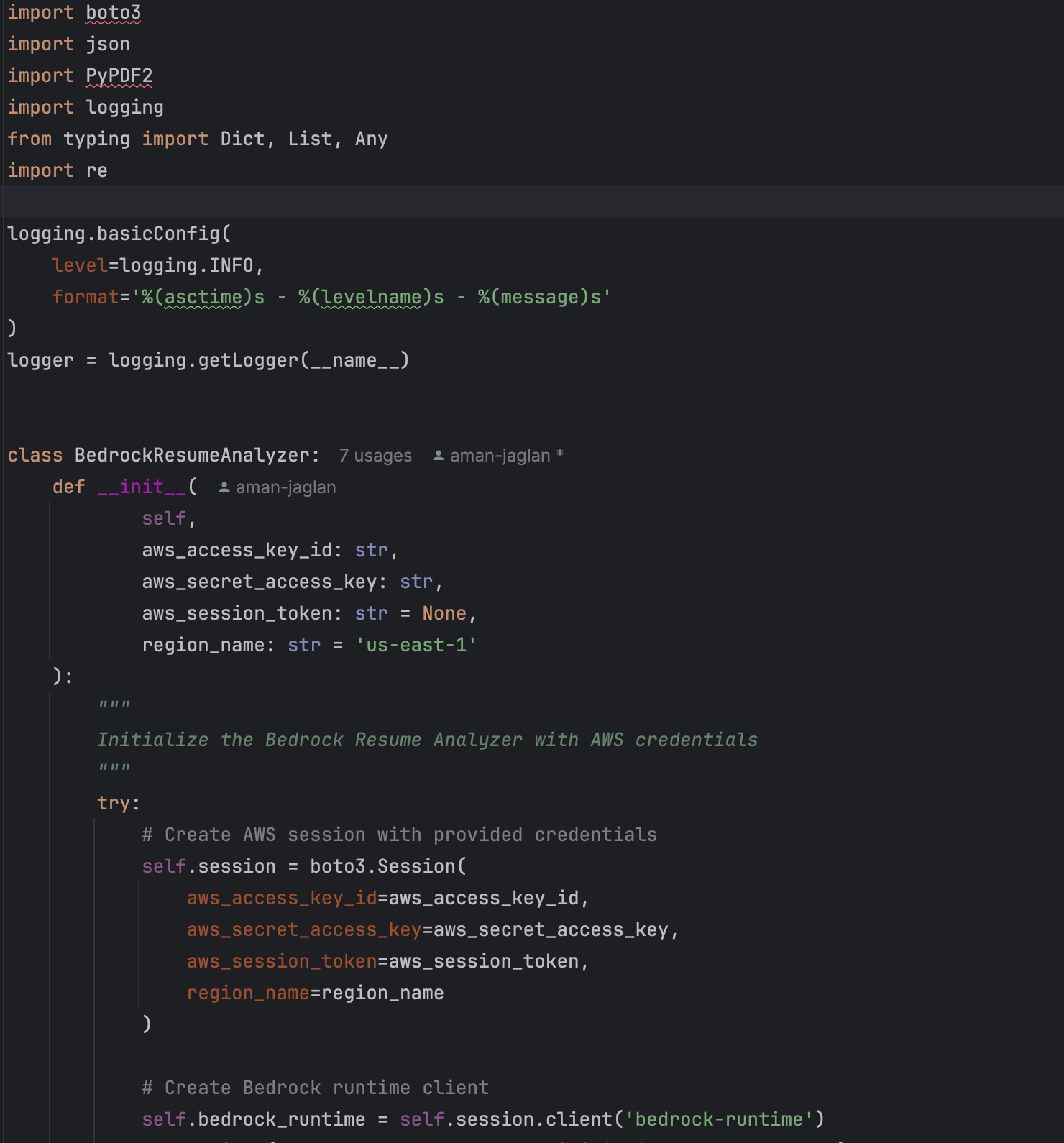
Inconsistent use of bullet points.

Uneven spacing between sections.

**Recommendations:**

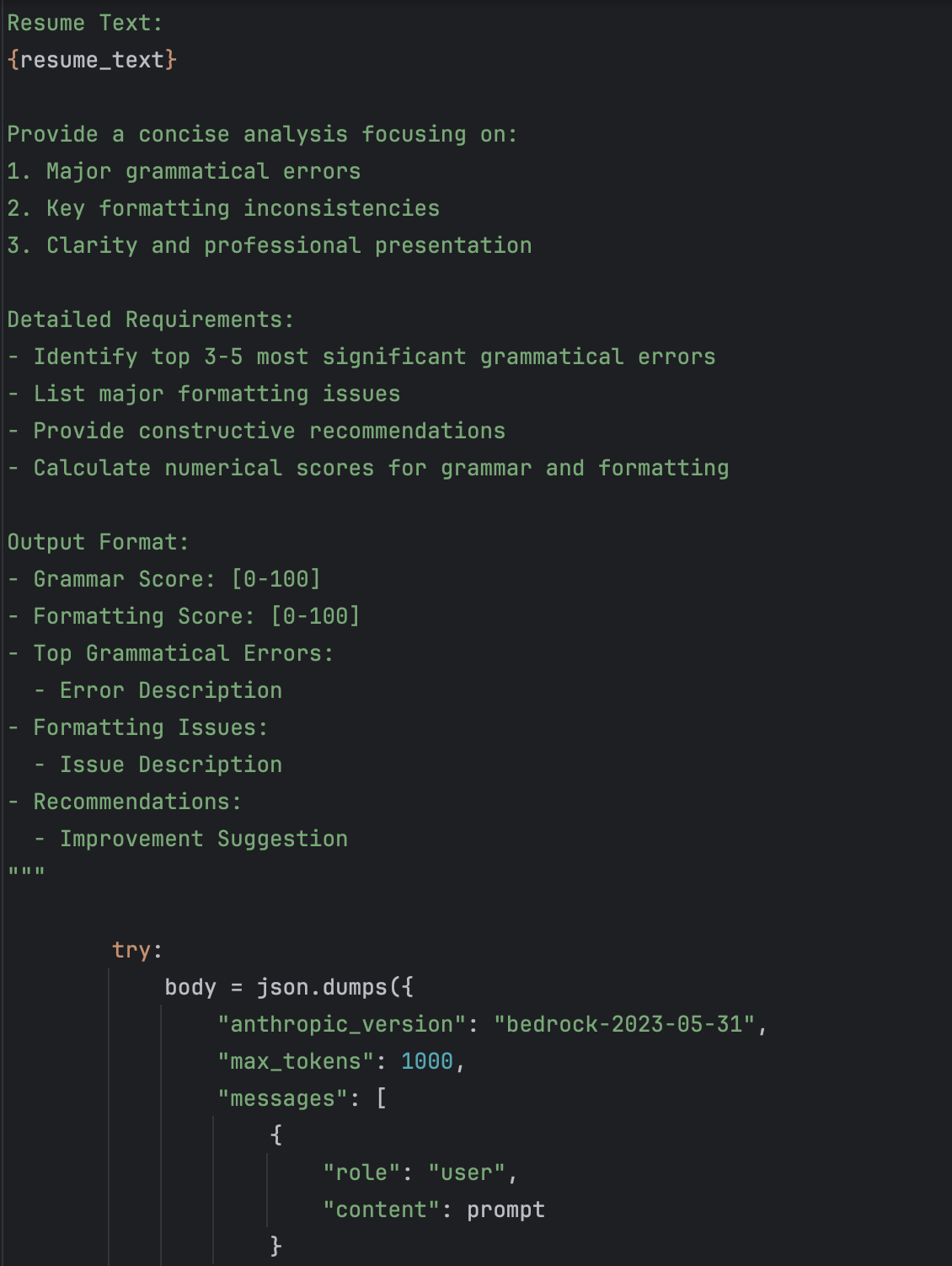
Use consistent formatting for bullets and spacing.

Revise verb tense for clarity and accuracy.

**Code:  
 **

**A screenshot of a computer program

Description automatically generated**

****

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

### ****Summary****

The **ResumeJobScore** and **Grammar and Formatting Check Model** are innovative systems designed to enhance resume quality and job compatibility through advanced text analysis and AI-driven insights.

#### **Pipeline 1: ResumeJobScore**

The **ResumeJobScore** project is a modular system for processing resumes and job descriptions. Key features include:

* **Input Data**: Collects resumes and job descriptions for analysis.
* **Preprocessing**: Extracts and cleans text data using job\_desc\_processor.py, resume\_processor.py, and tokenizer.py.
* **Model Training**: Trains AI models to evaluate resume quality and compatibility.
* **Pipeline Orchestration**: Coordinates the workflow through pipeline.py.
* **Output**: Generates scores and processed data, providing actionable insights.

#### **Pipeline 2: Grammar and Formatting Check Model**

This pipeline focuses on detailed grammar and formatting analysis of resumes using AWS Bedrock. Key highlights:

* **Text Extraction**: Uses PyPDF2 to extract text from PDF resumes.
* **AI-Driven Analysis**: Constructs a focused prompt for Bedrock's Claude model, identifying grammatical errors, formatting issues, and providing scores.
* **Output Parsing**: Extracts structured results, including scores and improvement recommendations.
* **Error Handling**: Robust mechanisms ensure reliable performance even in adverse scenarios.

Both pipelines are designed for precision and usability, making them highly effective for improving resume presentation and aligning candidate profiles with job requirements.

### ****Conclusion****

The **ResumeJobScore** and **Grammar and Formatting Check Model** together form a comprehensive toolkit for resume optimization. By integrating preprocessing, AI-powered analysis, and actionable insights, these systems empower users to present professional, tailored resumes. Their modularity and use of state-of-the-art technologies like AWS Bedrock ensure scalability and reliability for real-world applications.

These pipelines not only streamline the resume evaluation process but also help individuals and organizations make data-driven decisions in hiring and job applications. They stand as a testament to how AI and automation can enhance traditional workflows, providing both qualitative and quantitative benefits.

### ****Code Contribution Breakdown****

### 1. ResumeJobScore Pipeline

1. **Preprocessing (Job and Resume Processors, Tokenization)**:
   * **Internet Contribution (60%)**: Common preprocessing techniques like text cleaning, tokenization, and file parsing rely on widely available examples and open-source library usage (e.g., PyPDF2 for text extraction).
   * **My Contribution (40%)**: Integration of preprocessing steps, customization of cleaning logic, and orchestration for the ResumeJobScore system.
2. **Model Training**:
   * **Internet Contribution (60%)**: Model architectures, hyperparameter tuning guides, and evaluation metrics sourced from online resources or frameworks.
   * **My Contribution (40%)**: Customization of training pipelines, experimentation with parameters, and integration with downstream workflows.
3. **Pipeline Orchestration**:
   * **Internet Contribution (50%)**: Use of established patterns for orchestrating pipelines and library-specific code for smooth workflows.
   * **My Contribution (50%)**: Designing the specific flow, combining preprocessing, training, and output generation.
4. **Logging**:
   * **Internet Contribution (50%)**: Standardized logging configurations are commonly available online.
   * **My Contribution (50%)**: Incorporating custom logging statements and error messages for debugging and tracking.
5. **Main Execution**:
   * **Internet Contribution (40%)**: Using standard Python scripting practices for orchestrating function calls.
   * **My Contribution (60%)**: Structuring the script to handle specific pipeline requirements and configurations.

**2. Grammar and Formatting Check Model**

1. **AWS Integration (Initialization and Runtime)**:
   * **Internet Contribution (60%)**: Initialization of AWS clients and Bedrock integration uses standard examples from AWS documentation.
   * **My Contribution (40%)**: Structuring and customizing AWS client calls for resume analysis.
2. **Text Extraction**:
   * **Internet Contribution (70%)**: PyPDF2 examples for reading and extracting text from PDF files are readily available online.
   * **My Contribution (30%)**: Improved error handling, file path management, and text cleaning.
3. **AI-Driven Analysis**:
   * **Internet Contribution (50%)**: Prompt engineering guidance and response handling patterns are inspired by examples for Claude or similar models.
   * **My Contribution (50%)**: Crafting a tailored prompt for grammar and formatting checks, handling Bedrock responses, and parsing structured outputs.
4. **Output Parsing**:
   * **Internet Contribution (40%)**: General techniques for parsing text using regex or string manipulation are learned from external sources.
   * **My Contribution (60%)**: Writing specific regex patterns and logic to extract structured results from model responses.
5. **Error Handling**:
   * **Internet Contribution (50%)**: Standard patterns for exception handling and error logging.
   * **My Contribution (50%)**: Writing custom error messages and implementing robust fallbacks.

### ****Overall Contribution Estimate****

* **Internet Contribution**: 55–60%
  + Primarily includes standardized code snippets, examples from documentation, and open-source resources for libraries and tools.
* **My Contribution**: 40–45%
  + Focused on integration, customization, workflow design, and adapting generalized code to project-specific requirements.