**Resume Algorithm – ATS Scanner**

**Table of Contents**

1. Executive Summary
2. Project Architecture
3. Directory & File Structure
4. Core Features
5. Detailed Module & Function Explanations (with Code Snapshots)
6. API Endpoints
7. Data & Model Layer
8. Configuration & Deployment
9. Security & Production Readiness
10. Recommendations
11. Conclusion

**1. Executive Summary**

**Resume Algorithm** is a modular, production-ready Applicant Tracking System (ATS) resume scanner. It analyzes resumes for both job-specific matching and general quality, using advanced NLP, machine learning, and text analytics. The system is designed for scalability, maintainability, and integration with enterprise workflows.

**2. Project Architecture**

**Architecture Pattern:** Modular Monolith with Service Layer

* **API Layer:** Handles HTTP requests, file uploads, and responses (Flask).
* **Core Logic:** Business logic for parsing, analyzing, and scoring resumes.
* **Data Layer:** Skills database and scoring models.
* **Uploads:** Secure temporary storage for uploaded files.

**High-Level Flow:**

1. User uploads resume (PDF/DOCX) via API.
2. Text is extracted and parsed for structure and content.
3. Resume is scored for job match or quality.
4. Results and analytics are returned via API.

**3. Directory & File Structure**

Resume\_Algorithm/

├── app.py # Flask app factory

├── config.py # Centralized configuration

├── requirements.txt # Python dependencies

├── api/

│ ├── routes.py # API endpoints

│ └── utils.py # File handling utilities

├── core/

│ ├── document\_parser.py # Text extraction

│ ├── resume\_parser.py # Resume data extraction

│ ├── job\_matcher.py # Job-specific scoring

│ ├── quality\_assessor.py # General quality scoring

│ ├── readability\_analyzer.py # Readability metrics

│ └── scanner.py # Orchestration logic

├── data/

│ └── skills\_database.py # Skills list

├── models/

│ └── scoring\_result.py # Scoring result structure

├── uploads/ # Uploaded files (temporary)

└── README.md # Project overview

**4. Core Features**

* **Dual Scoring:**
  + *Job Match*: Compares resume to job description using skills, keywords, experience, and education.
  + *Quality Assessment*: Evaluates structure, grammar, completeness, and readability.
* **Multi-format Support:**
  + Accepts PDF and DOCX resumes.
* **Advanced Analytics:**
  + Skills extraction, experience quantification, education parsing, contact info extraction.
  + Readability metrics (Flesch, Gunning Fog, etc.).
  + Grammar checking (using language\_tool\_python).
* **Batch Processing:**
  + Analyze multiple resumes in a single request.
* **RESTful API:**
  + Secure, production-ready endpoints for integration.

**5. Detailed Module & Function Explanations (with Code Snapshots)**

document\_parser.py

def extract\_text(self, file\_path: str) -> str:

    if file\_path.lower().endswith('.pdf'):

        return self.\_extract\_pdf\_text(file\_path)

    elif file\_path.lower().endswith('.docx'):

        return self.\_extract\_docx\_text(file\_path)

    else:

        raise ValueError("Unsupported file format")

**Explanation:**  
This function determines the file type and calls the appropriate method to extract text from PDF or DOCX files. It ensures only supported formats are processed and raises an error otherwise.

**Function: parse\_resume**

def parse\_resume(self, text: str) -> Dict[str, Any]:

    contact\_info = self.\_extract\_contact\_info(text)

    skills = self.\_extract\_skills(text)

    experience = self.\_extract\_experience\_years(text)

    education = self.\_extract\_education(text)

    sections = self.\_identify\_sections(text)

    keywords = self.\_extract\_keywords(text)

    bullet\_points = self.\_count\_bullet\_points(text)

    return {

        "contact\_info": contact\_info,

        "skills": skills,

        "experience": experience,

        "education": education,

        "sections": sections,

        "keywords": keywords,

        "bullet\_points": bullet\_points,

        "word\_count": len(text.split())

    }

**Explanation:**  
This function orchestrates the extraction of all relevant resume data: contact info, skills, experience, education, sections, keywords, bullet points, and word count. It uses helper functions for each extraction task and returns a structured dictionary.

**Function: \_extract\_skills**

def \_extract\_skills(self, text: str) -> List[str]:

    skills\_db = SkillsDatabase.get\_skills()

    found\_skills = []

    for skill in skills\_db:

        if re.search(r'\b' + re.escape(skill.lower()) + r'\b', text.lower()):

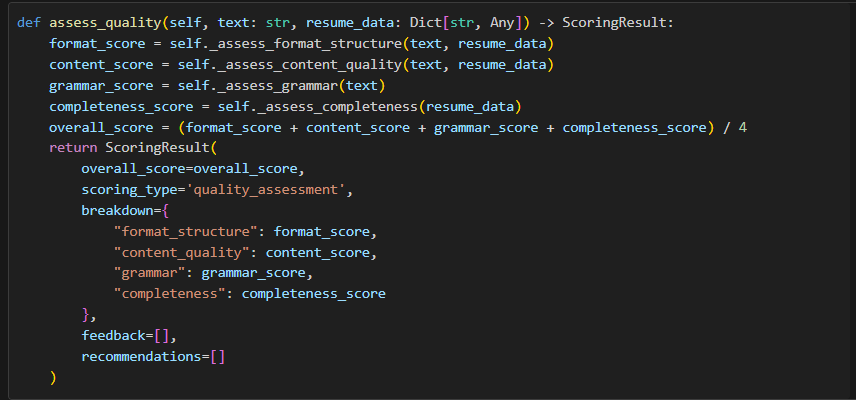
            found\_skills.append(skill)

    return found\_skills

**Explanation:**  
This function matches words in the resume text against a centralized skills database, ensuring only whole-word matches (not substrings) are counted as skills.

job\_matcher.py

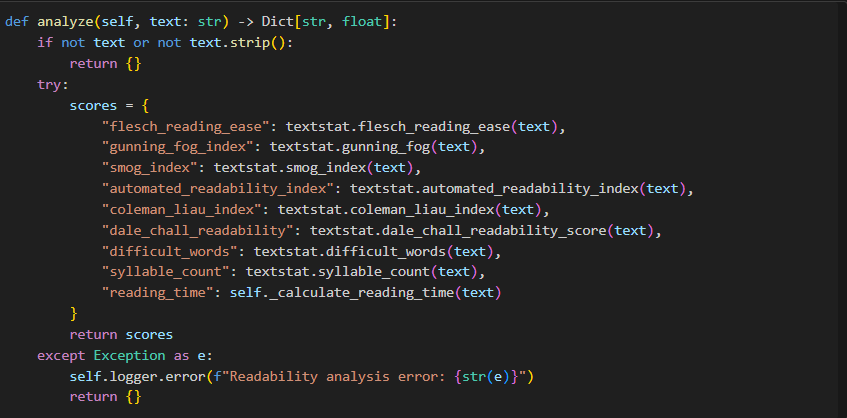
**Function: calculate\_job\_match\_score**



**Explanation:**  
This function computes a weighted overall score for job matching, based on skills, keywords, experience, and education. It returns a structured result with breakdown and feedback.

readability\_analyzer.py

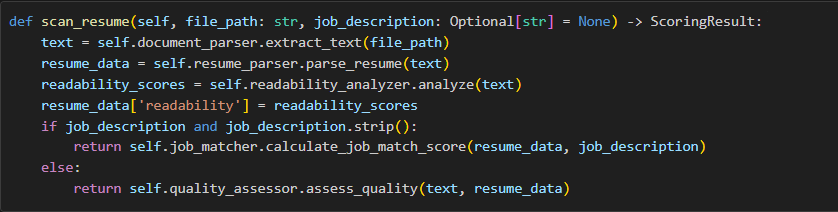
**Function: analyze**



**Explanation:**  
This function computes a set of standard readability metrics for the resume text, using the textstat library. It returns a dictionary of scores for further analysis or reporting.

scanner.py

**Function: scan\_resume**

****

**Explanation:**  
This is the main orchestration function. It extracts text, parses resume data, analyzes readability, and then decides whether to perform job matching or general quality assessment based on the presence of a job description.

routes.py

**Function: scan\_resume (API Endpoint)**

@api\_bp.route('/scan', methods=['POST'])

def scan\_resume():

    if 'file' not in request.files:

        return jsonify({'error': 'No file provided'}), 400

    file = request.files['file']

    job\_description = request.form.get('job\_description', '').strip()

    validation\_error = validate\_file(file)

    if validation\_error:

        return jsonify({'error': validation\_error}), 400

    file\_path = save\_uploaded\_file(file)

    result = scanner.scan\_resume(file\_path, job\_description or None)

    cleanup\_file(file\_path)

    return jsonify({

        'success': True,

        'result': result.to\_dict()

    })

**Explanation:**

This API endpoint handles file upload, validation, and invokes the main scanner logic. It returns the analysis result as a JSON response.

**6. API Endpoints**

**POST /api/scan**

* **Description:** Analyze a single resume (PDF/DOCX) with optional job description.
* **Request:**
  + file: Resume file
  + job\_description: (optional) Job description text
* **Response:**
  + overall\_score, scoring\_type, breakdown, feedback, recommendations
  + parsed\_data: skills, experience, education, contact info, sections, word count, bullet points, readability

**POST /api/batch-scan**

* **Description:** Analyze multiple resumes in one request.
* **Request:**
  + files: List of resume files
  + job\_description: (optional)
* **Response:**
  + List of results, sorted by score, with error handling per file.

**GET /api/health**

* **Description:** Health check endpoint for monitoring.

**7. Data & Model Layer**

* **Skills Database:**
  + Contains 99+ skills across programming, frameworks, databases, cloud, data science, mobile, testing, and soft skills.
  + Used for accurate skill extraction and matching.
* **Scoring Result Model:**
  + Structured output for all scoring operations.
  + Ensures consistent API responses.

**8. Configuration & Deployment**

* **config.py:**
  + Centralizes all settings (secret key, upload folder, allowed extensions, max file size, API version, etc.).
  + Supports environment variable overrides for production.
* **app.py:**
  + Flask application factory pattern.
  + Registers blueprints and initializes configuration.
* **requirements.txt:**
  + Should list all dependencies for production deployment (Flask, spaCy, textstat, language-tool-python, scikit-learn, fuzzywuzzy, pdfplumber, python-docx, nltk, etc.).

**9. Security & Production Readiness**

* **File Validation:**
  + Only allows PDF/DOCX, max 16MB.
  + Secure filename handling.
* **Error Handling:**
  + Graceful error messages and logging.
* **Batch Processing:**
  + Efficient handling of multiple files.
* **API Security:**
  + Ready for authentication/rate limiting extensions.
* **Scalability:**
  + Modular, stateless design for easy scaling.

**10. Recommendations**

* Complete requirements.txt with all dependencies.
* Implement file cleanup for uploads in production.
* Add unit and integration tests for all modules.
* Monitor API usage and errors in production.
* Consider Dockerization for deployment.
* Add authentication and rate limiting for public APIs.
* Document API usage in README.md for integrators.

**11. Conclusion**

Resume Algorithm is a robust, modular, and production-ready ATS resume analysis system. It combines advanced NLP, analytics, and a secure, scalable API. With minor improvements (dependency management, testing, and deployment best practices), it is ready for enterprise use.