**Resume AI Scanning & Job Search Feature Implementation Plan**

**Technical Implementation Steps**

**Phase 1: Resume Upload & Processing**

1. **File Upload System**
   * Accept PDF, DOC, DOCX formats
   * File size validation (max 5MB)
   * Virus scanning integration
   * Secure cloud storage (AWS S3/Google Cloud Storage)
2. **Document Conversion**
   * Convert all formats to text using libraries like PyPDF2, python-docx
   * OCR integration for image-based PDFs using Tesseract or Google Vision API
   * Text preprocessing and cleaning

**Phase 2: AI Resume Parsing**

1. **AI-Powered Data Extraction**
   * Use OpenAI GPT-4 or specialized resume parsing APIs (like Affinda, Resume Library API)
   * Extract key fields:
     + Personal information (name, contact)
     + Work experience (companies, roles, dates, responsibilities)
     + Education (degrees, institutions, dates)
     + Skills (technical, soft skills)
     + Certifications and achievements
     + Salary history (if available)
2. **Data Structuring**
   * Create standardized data models
   * Validate and clean extracted information
   * Store in database with user profile linkage

**Phase 3: Job Search Integration**

1. **Job Search API Integration**
   * Primary: Indeed API, LinkedIn Job Search API
   * Secondary: Glassdoor API, ZipRecruiter API, Google Jobs API
   * Filter jobs by:
     + Salary range (targeting 15-30% increase)
     + Location (within user's metro area)
     + Experience level match
     + Skills alignment
2. **Intelligent Matching Algorithm**
   * Score job compatibility based on:
     + Skills match percentage
     + Experience level alignment
     + Industry fit
     + Salary improvement potential
     + Location preference

**Phase 4: Recommendations & Integration**

1. **Personalized Job Recommendations**
   * Generate ranked job suggestions
   * Calculate potential income impact on financial forecast
   * Show timeline for application and potential start dates
   * Integration with existing Mingus cash flow projections
2. **User Dashboard Integration**
   * Career progression tracking
   * Application status management
   * Income goal vs. opportunity analysis
   * Interview preparation resources

**Development Costs**

**Initial Development (6-8 months)**

* **Senior Full-Stack Developer**: $120k/year × 0.6 FTE = $72,000
* **AI/ML Specialist**: $140k/year × 0.4 FTE = $56,000
* **UI/UX Designer**: $90k/year × 0.3 FTE = $27,000
* **Project Management**: $100k/year × 0.2 FTE = $20,000
* **Testing & QA**: $70k/year × 0.2 FTE = $14,000

**Total Development Cost: $189,000**

**Alternative Lower-Cost Approach**

* **Contract Developers** (offshore): $40-60/hour × 1,200 hours = $48,000-72,000
* **AI Integration Specialist**: $80/hour × 200 hours = $16,000
* **UI/UX Contract Work**: $60/hour × 150 hours = $9,000

**Budget Alternative Total: $73,000-97,000**

**Monthly Deployment Costs**

**AI & API Services**

* **OpenAI GPT-4 API**: ~$0.03 per resume + ~$0.002 per job recommendation
* **Resume Parsing API** (alternative): $0.50-2.00 per resume
* **Job Search APIs**:
  + Indeed: $0.10-0.25 per search
  + LinkedIn: $0.15-0.30 per search
  + Glassdoor: $0.20-0.40 per search

**Infrastructure (AWS/Google Cloud)**

* **Storage**: $50-150/month (for resume files)
* **Database**: $100-300/month (managed database service)
* **Compute**: $200-500/month (API processing, background jobs)
* **CDN & Bandwidth**: $50-100/month

**Estimated Monthly Costs by Usage**

* **500 users/month**: $800-1,200
* **1,500 users/month**: $1,800-2,800
* **3,000 users/month**: $3,200-4,800

**Potential for Use & Value Proposition**

**Target Market Alignment**

**Highly Relevant** - Addresses Problem #3: "Low Access to High-Growth Career Paths"

* 67% of target customers (ages 25-35, $40k-100k) actively seeking career advancement
* Particularly valuable for recent graduates and early-career professionals

**Feature Adoption Projections**

* **Budget Tier ($10)**: 40% adoption rate (133 users)
* **Mid-Tier ($20)**: 65% adoption rate (368 users)
* **Professional ($50)**: 85% adoption rate (85 users)

**Total Expected Users: 586 out of 1,000 signups**

**Revenue Enhancement Opportunities**

1. **Premium Job Alerts**: $5/month add-on
2. **Resume Optimization Service**: $25 one-time fee
3. **Interview Coaching Integration**: $15/month
4. **Salary Negotiation Tools**: $10/month

**Competitive Advantages**

* **Holistic Integration**: Links career growth directly to financial forecasting
* **Cultural Relevance**: Addresses specific barriers faced by target demographic
* **Actionable Timeline**: Shows real impact on monthly cash flow projections
* **Local Focus**: Prioritizes opportunities in target metro areas

**Success Metrics**

* **Primary**: 25% of users find and apply to higher-paying positions within 6 months
* **Secondary**: Average 18% salary increase for successful job changes
* **Financial Impact**: $200-400/month additional income reflected in user forecasts

**Risk Mitigation**

* **Start with MVP**: Basic resume parsing + job search integration
* **Gradual Enhancement**: Add AI sophistication over time
* **API Backup Plans**: Multiple job search sources to ensure reliability
* **Freemium Model**: Basic feature free, advanced matching paid

This feature directly addresses your target market's core challenge while providing a clear path to increased revenue and user engagement within the Mingus ecosystem.

**Budget-Friendly AI Services for MVP Launch**

**Resume Parsing Services (Ranked by Cost-Effectiveness)**

**1. Affinda Resume Parser ⭐ RECOMMENDED FOR MVP**

* **Cost**: $0.50 per resume (first 100 free)
* **Accuracy**: 95%+ for standard formats
* **Integration**: Simple REST API, Python SDK available
* **Pros**:
  + Excellent for beginners
  + Handles 20+ resume formats
  + Pre-built field extraction (experience, education, skills)
  + Good documentation
* **Cons**: Per-use pricing can scale up
* **Monthly Cost**: ~$250 for 500 resumes

**2. HrFlow.ai Resume Parsing**

* **Cost**: $0.30 per resume (1,000 free credits)
* **Features**: Skills matching, job recommendations built-in
* **Integration**: Python library, REST API
* **Best For**: Comprehensive solution with job matching
* **Monthly Cost**: ~$150 for 500 resumes

**3. Lever Resume Parser**

* **Cost**: $0.25 per resume
* **Accuracy**: 90%+
* **Pros**: Lower cost, good for high volume
* **Cons**: Less detailed extraction than Affinda
* **Monthly Cost**: ~$125 for 500 resumes

**4. DIY with Open Source + Basic AI**

* **Tools**: spaCy + Custom regex patterns + Hugging Face models
* **Cost**: Nearly free (just compute costs)
* **Accuracy**: 70-80% (requires more development)
* **Best For**: Long-term cost savings, full control

**Job Search APIs (Free & Low-Cost Options)**

**1. Adzuna Job Search API ⭐ BEST FREE OPTION**

* **Cost**: FREE up to 1,000 calls/month
* **Coverage**: 1M+ jobs globally, good US coverage
* **Features**: Salary data, location filtering, skills matching
* **Perfect for MVP testing**

**2. JSearch by RapidAPI**

* **Cost**: FREE tier: 150 searches/month
* **Paid**: $10/month for 1,500 searches
* **Features**: Indeed, LinkedIn, ZipRecruiter aggregation
* **Good middle-ground option**

**3. Reed Jobs API**

* **Cost**: FREE (rate-limited)
* **Coverage**: UK-focused but has US jobs
* **Best for**: Testing and small-scale launch

**4. Custom Web Scraping (Use with caution)**

* **Tools**: BeautifulSoup + Selenium
* **Cost**: Nearly free
* **Risks**: Legal concerns, breaking changes
* **Only if other APIs don't work**

**Text Processing & AI Services**

**1. Hugging Face Transformers ⭐ BEST FOR CUSTOM AI**

* **Cost**: FREE for most models
* **Models**: Pre-trained resume/job matching models available
* **Hosting**: $0.60/hour on Hugging Face Spaces
* **Perfect for**: Skills extraction, job matching scoring
* **Python Integration**: Excellent

**2. Google Cloud Natural Language API**

* **Cost**: $1 per 1,000 text units (first 5,000 free monthly)
* **Features**: Entity extraction, sentiment analysis
* **Use Case**: Extract skills and keywords from resumes
* **Monthly Cost**: ~$50 for 500 resumes

**3. AWS Comprehend**

* **Cost**: $0.0001 per unit (first 5M units free for 12 months)
* **Features**: Key phrase extraction, entity recognition
* **Integration**: boto3 Python library
* **Very cost-effective for basic NLP**

**4. OpenAI GPT-3.5-Turbo (Budget Option)**

* **Cost**: $0.002 per 1K tokens (much cheaper than GPT-4)
* **Use**: Resume summarization, job description matching
* **Monthly Cost**: ~$30-50 for 500 resumes
* **Good balance of cost and capability**

**Document Processing & OCR**

**1. PyPDF2 + python-docx ⭐ FREE BASELINE**

* **Cost**: FREE
* **Handles**: Most text-based PDFs and Word docs
* **Limitations**: No OCR for image-based PDFs
* **Perfect starting point**

**2. Google Cloud Vision API**

* **Cost**: $1.50 per 1,000 images (first 1,000 free monthly)
* **Features**: OCR for scanned resumes
* **Only needed for image-based PDFs**

**3. Tesseract OCR**

* **Cost**: FREE (open source)
* **Quality**: Good but requires tuning
* **Setup**: More complex but no ongoing costs

**Recommended MVP Tech Stack (Under $500/month)**

**Core Setup**

1. **Resume Parsing**: Affinda ($250/month for 500 resumes)
2. **Job Search**: Adzuna Free tier + JSearch ($10/month)
3. **AI Processing**: Hugging Face + GPT-3.5-Turbo ($50/month)
4. **Document Processing**: PyPDF2/python-docx (FREE)
5. **Infrastructure**: Basic cloud hosting ($100/month)

**Total Monthly Cost: ~$410 for 500 resumes processed**

**Alternative Ultra-Budget Setup (Under $150/month)**

1. **Resume Parsing**: HrFlow.ai ($150/month)
2. **Job Search**: Adzuna (FREE)
3. **AI Processing**: Hugging Face only (FREE)
4. **Document Processing**: PyPDF2 (FREE)
5. **Infrastructure**: Basic hosting ($50/month)

**Total Monthly Cost: ~$200**

**Implementation Priority for MVP**

**Phase 1 (Weeks 1-4): Basic Functionality**

* PyPDF2 for text extraction
* Hugging Face for basic skills extraction
* Adzuna API for job search
* Simple matching algorithm

**Phase 2 (Weeks 5-8): Enhanced Parsing**

* Add Affinda for better resume parsing
* Integrate GPT-3.5-turbo for improved matching
* Add salary comparison features

**Phase 3 (Weeks 9-12): Scale & Polish**

* Add paid job search APIs for more results
* Implement machine learning for better matching
* Add user feedback loops

**Quick Start Code Examples**

**Resume Text Extraction (Free)**

import PyPDF2

from docx import Document

def extract\_text\_from\_resume(file\_path):

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

with open(file\_path, 'rb') as file:

pdf\_reader = PyPDF2.PdfReader(file)

text = ""

for page in pdf\_reader.pages:

text += page.extract\_text()

return text

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

doc = Document(file\_path)

return ' '.join([paragraph.text for paragraph in doc.paragraphs])

**Affinda Integration**

import requests

def parse\_resume\_with\_affinda(file\_path, api\_key):

url = "https://api.affinda.com/v3/resumes"

headers = {"Authorization": f"Bearer {api\_key}"}

with open(file\_path, 'rb') as file:

files = {"file": file}

response = requests.post(url, headers=headers, files=files)

return response.json()

This approach gets you to market quickly with proven technologies while keeping costs manageable for an MVP launch.

**Ultra-Budget Resume AI Implementation Guide**

**Prerequisites & Setup (Day 1)**

**1. Development Environment Setup**

# Create project directory

mkdir mingus-resume-ai

cd mingus-resume-ai

# Create virtual environment

python -m venv venv

source venv/bin/activate # On Windows: venv\Scripts\activate

# Install required packages

pip install requests PyPDF2 python-docx transformers torch pandas flask python-dotenv

**2. Project Structure**

mingus-resume-ai/

├── app.py # Main Flask application

├── resume\_parser.py # Resume processing logic

├── job\_searcher.py # Job search functionality

├── ai\_matcher.py # AI matching logic

├── config.py # Configuration settings

├── templates/ # HTML templates

│ ├── index.html

│ ├── upload.html

│ └── results.html

├── static/ # CSS, JS files

├── uploads/ # Temporary file storage

├── requirements.txt

└── .env # Environment variables

**Step 1: Basic Resume Parser (Days 2-3)**

**Create resume\_parser.py**

import PyPDF2

import docx

import re

import os

from typing import Dict, List

class ResumeParser:

def \_\_init\_\_(self):

self.skills\_keywords = [

'python', 'javascript', 'java', 'react', 'node.js', 'sql', 'excel',

'project management', 'marketing', 'sales', 'customer service',

'data analysis', 'communication', 'leadership', 'teamwork'

]

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

"""Extract text from PDF or DOCX files"""

try:

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

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

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

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

else:

raise ValueError("Unsupported file format")

except Exception as e:

print(f"Error extracting text: {e}")

return ""

def \_extract\_from\_pdf(self, file\_path: str) -> str:

"""Extract text from PDF"""

text = ""

with open(file\_path, 'rb') as file:

pdf\_reader = PyPDF2.PdfReader(file)

for page in pdf\_reader.pages:

text += page.extract\_text() + "\n"

return text

def \_extract\_from\_docx(self, file\_path: str) -> str:

"""Extract text from DOCX"""

doc = docx.Document(file\_path)

text = ""

for paragraph in doc.paragraphs:

text += paragraph.text + "\n"

return text

def parse\_resume(self, file\_path: str) -> Dict:

"""Main parsing function - returns structured data"""

text = self.extract\_text\_from\_file(file\_path)

return {

'name': self.\_extract\_name(text),

'email': self.\_extract\_email(text),

'phone': self.\_extract\_phone(text),

'experience': self.\_extract\_experience(text),

'education': self.\_extract\_education(text),

'skills': self.\_extract\_skills(text),

'raw\_text': text

}

def \_extract\_name(self, text: str) -> str:

"""Extract name (first non-empty line usually)"""

lines = [line.strip() for line in text.split('\n') if line.strip()]

if lines:

# Usually the name is in the first few lines

for line in lines[:3]:

if not any(keyword in line.lower() for keyword in ['email', '@', 'phone', 'address']):

return line

return "Name not found"

def \_extract\_email(self, text: str) -> str:

"""Extract email address"""

email\_pattern = r'\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'

emails = re.findall(email\_pattern, text)

return emails[0] if emails else "Email not found"

def \_extract\_phone(self, text: str) -> str:

"""Extract phone number"""

phone\_pattern = r'(\+?1[-.\s]?)?\(?([0-9]{3})\)?[-.\s]?([0-9]{3})[-.\s]?([0-9]{4})'

phones = re.findall(phone\_pattern, text)

if phones:

return ''.join(phones[0])

return "Phone not found"

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

"""Extract work experience (basic pattern matching)"""

experience\_keywords = ['experience', 'employment', 'work history', 'professional']

lines = text.split('\n')

experience\_section = []

in\_experience = False

for line in lines:

line = line.strip()

if any(keyword in line.lower() for keyword in experience\_keywords):

in\_experience = True

continue

elif in\_experience and any(keyword in line.lower() for keyword in ['education', 'skills', 'certifications']):

break

elif in\_experience and line:

# Look for job titles (lines with years or company indicators)

if re.search(r'\d{4}', line) or any(word in line.lower() for word in ['inc', 'llc', 'corp', 'company']):

experience\_section.append(line)

return experience\_section[:5] # Return up to 5 most recent

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

"""Extract education information"""

education\_keywords = ['education', 'degree', 'university', 'college', 'bachelor', 'master']

lines = text.split('\n')

education\_section = []

in\_education = False

for line in lines:

line = line.strip()

if any(keyword in line.lower() for keyword in education\_keywords):

in\_education = True

if any(degree in line.lower() for degree in ['bachelor', 'master', 'phd', 'degree']):

education\_section.append(line)

continue

elif in\_education and line:

if any(word in line.lower() for word in ['university', 'college', 'institute']):

education\_section.append(line)

elif any(keyword in line.lower() for keyword in ['experience', 'skills', 'certifications']):

break

return education\_section

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

"""Extract skills from resume"""

found\_skills = []

text\_lower = text.lower()

for skill in self.skills\_keywords:

if skill in text\_lower:

found\_skills.append(skill.title())

return found\_skills

# Test the parser

if \_\_name\_\_ == "\_\_main\_\_":

parser = ResumeParser()

# Test with a sample file

# result = parser.parse\_resume("sample\_resume.pdf")

# print(result)

**Step 2: Job Search Integration (Days 4-5)**

**Create job\_searcher.py**

import requests

import time

from typing import List, Dict

from dataclasses import dataclass

@dataclass

class Job:

title: str

company: str

location: str

salary: str

description: str

url: str

posted\_date: str

class JobSearcher:

def \_\_init\_\_(self):

self.adzuna\_app\_id = "YOUR\_APP\_ID" # Get free from adzuna.com

self.adzuna\_api\_key = "YOUR\_API\_KEY"

self.base\_url = "https://api.adzuna.com/v1/api/jobs/us/search"

def search\_jobs(self, keywords: str, location: str = "",

salary\_min: int = None, max\_results: int = 20) -> List[Job]:

"""Search for jobs using Adzuna API"""

params = {

'app\_id': self.adzuna\_app\_id,

'app\_key': self.adzuna\_api\_key,

'what': keywords,

'where': location,

'results\_per\_page': min(max\_results, 50),

'content-type': 'application/json'

}

if salary\_min:

params['salary\_min'] = salary\_min

try:

response = requests.get(self.base\_url + "/1", params=params)

response.raise\_for\_status()

data = response.json()

jobs = []

for job\_data in data.get('results', []):

job = Job(

title=job\_data.get('title', 'N/A'),

company=job\_data.get('company', {}).get('display\_name', 'N/A'),

location=job\_data.get('location', {}).get('display\_name', 'N/A'),

salary=self.\_format\_salary(job\_data.get('salary\_min'), job\_data.get('salary\_max')),

description=job\_data.get('description', 'N/A')[:500] + '...',

url=job\_data.get('redirect\_url', ''),

posted\_date=job\_data.get('created', 'N/A')

)

jobs.append(job)

return jobs

except requests.RequestException as e:

print(f"Error searching jobs: {e}")

return []

def \_format\_salary(self, min\_salary, max\_salary):

"""Format salary information"""

if min\_salary and max\_salary:

return f"${min\_salary:,.0f} - ${max\_salary:,.0f}"

elif min\_salary:

return f"${min\_salary:,.0f}+"

elif max\_salary:

return f"Up to ${max\_salary:,.0f}"

else:

return "Salary not specified"

def search\_by\_skills(self, skills: List[str], location: str = "",

current\_salary: int = None) -> List[Job]:

"""Search jobs based on extracted skills"""

all\_jobs = []

# Search for each skill combination

for skill in skills[:3]: # Limit to top 3 skills to avoid rate limits

jobs = self.search\_jobs(

keywords=skill,

location=location,

salary\_min=current\_salary \* 1.1 if current\_salary else None,

max\_results=10

)

all\_jobs.extend(jobs)

time.sleep(1) # Rate limiting

# Remove duplicates based on title and company

unique\_jobs = []

seen = set()

for job in all\_jobs:

job\_key = (job.title.lower(), job.company.lower())

if job\_key not in seen:

seen.add(job\_key)

unique\_jobs.append(job)

return unique\_jobs[:20] # Return top 20

# Test the job searcher

if \_\_name\_\_ == "\_\_main\_\_":

searcher = JobSearcher()

# jobs = searcher.search\_jobs("python developer", "Atlanta")

# for job in jobs[:3]:

# print(f"{job.title} at {job.company} - {job.salary}")

**Step 3: Simple AI Matching (Days 6-7)**

**Create ai\_matcher.py**

from typing import List, Dict, Tuple

from dataclasses import dataclass

import re

@dataclass

class JobMatch:

job: 'Job' # From job\_searcher.py

match\_score: float

salary\_increase: float

matching\_skills: List[str]

reasons: List[str]

class SimpleAIMatcher:

def \_\_init\_\_(self):

# Skill categories for better matching

self.skill\_categories = {

'technical': ['python', 'javascript', 'java', 'sql', 'react', 'node.js', 'html', 'css'],

'analytical': ['data analysis', 'excel', 'statistics', 'research'],

'management': ['project management', 'leadership', 'team lead', 'supervisor'],

'communication': ['communication', 'presentation', 'writing', 'customer service']

}

# Experience level indicators

self.experience\_levels = {

'entry': ['entry', 'junior', 'associate', '0-2 years'],

'mid': ['mid', 'senior', '3-5 years', '2-5 years'],

'senior': ['senior', 'lead', 'principal', '5+ years', 'manager']

}

def calculate\_job\_match(self, resume\_data: Dict, job, current\_salary: int = None) -> JobMatch:

"""Calculate how well a job matches the resume"""

resume\_skills = [skill.lower() for skill in resume\_data.get('skills', [])]

job\_description = (job.title + " " + job.description).lower()

# Calculate skill match

matching\_skills = []

for skill in resume\_skills:

if skill in job\_description:

matching\_skills.append(skill.title())

skill\_match\_score = len(matching\_skills) / max(len(resume\_skills), 1)

# Calculate experience level match

experience\_match = self.\_calculate\_experience\_match(resume\_data, job\_description)

# Calculate salary improvement

salary\_increase = self.\_estimate\_salary\_increase(job, current\_salary)

# Overall match score (weighted)

match\_score = (

skill\_match\_score \* 0.4 +

experience\_match \* 0.3 +

min(salary\_increase / 20, 1.0) \* 0.3 # Normalize salary increase impact

)

# Generate reasons

reasons = self.\_generate\_match\_reasons(matching\_skills, salary\_increase, experience\_match)

return JobMatch(

job=job,

match\_score=match\_score,

salary\_increase=salary\_increase,

matching\_skills=matching\_skills,

reasons=reasons

)

def \_calculate\_experience\_match(self, resume\_data: Dict, job\_description: str) -> float:

"""Estimate experience level match"""

resume\_experience = resume\_data.get('experience', [])

years\_experience = len(resume\_experience) # Simple approximation

# Determine required experience level from job description

if any(term in job\_description for term in self.experience\_levels['entry']):

required\_level = 'entry'

elif any(term in job\_description for term in self.experience\_levels['senior']):

required\_level = 'senior'

else:

required\_level = 'mid'

# Match experience level

if required\_level == 'entry' and years\_experience <= 2:

return 1.0

elif required\_level == 'mid' and 2 <= years\_experience <= 5:

return 1.0

elif required\_level == 'senior' and years\_experience >= 5:

return 1.0

else:

return 0.6 # Partial match

def \_estimate\_salary\_increase(self, job, current\_salary: int) -> float:

"""Estimate percentage salary increase"""

if not current\_salary:

return 0.0

# Extract salary from job (basic parsing)

salary\_text = job.salary.lower()

# Find numbers in salary text

salary\_numbers = re.findall(r'\$?(\d{1,3}(?:,\d{3})\*)', salary\_text)

if salary\_numbers:

# Take the higher number if range, or the only number

max\_salary = max([int(num.replace(',', '')) for num in salary\_numbers])

# If number is too small, it might be hourly - convert to annual

if max\_salary < 1000:

max\_salary \*= 2000 # Approximate annual from hourly

increase\_percentage = ((max\_salary - current\_salary) / current\_salary) \* 100

return max(0, increase\_percentage)

return 0.0

def \_generate\_match\_reasons(self, matching\_skills: List[str],

salary\_increase: float, experience\_match: float) -> List[str]:

"""Generate human-readable reasons for the match"""

reasons = []

if matching\_skills:

reasons.append(f"Skills match: {', '.join(matching\_skills[:3])}")

if salary\_increase > 15:

reasons.append(f"Potential {salary\_increase:.1f}% salary increase")

elif salary\_increase > 5:

reasons.append(f"Modest {salary\_increase:.1f}% salary increase")

if experience\_match >= 0.8:

reasons.append("Experience level is a good fit")

if not reasons:

reasons.append("Basic qualifications match")

return reasons

def rank\_jobs(self, resume\_data: Dict, jobs: List, current\_salary: int = None) -> List[JobMatch]:

"""Rank all jobs by match score"""

job\_matches = []

for job in jobs:

match = self.calculate\_job\_match(resume\_data, job, current\_salary)

job\_matches.append(match)

# Sort by match score (highest first)

job\_matches.sort(key=lambda x: x.match\_score, reverse=True)

return job\_matches

# Test the matcher

if \_\_name\_\_ == "\_\_main\_\_":

matcher = SimpleAIMatcher()

# sample\_resume = {'skills': ['Python', 'SQL', 'Project Management']}

# sample\_job = Job(title="Python Developer", description="Looking for Python and SQL skills", ...)

# match = matcher.calculate\_job\_match(sample\_resume, sample\_job, 50000)

# print(f"Match score: {match.match\_score}")

**Step 4: Web Interface (Days 8-9)**

**Create app.py (Flask web application)**

from flask import Flask, render\_template, request, redirect, url\_for, flash, jsonify

import os

from werkzeug.utils import secure\_filename

from resume\_parser import ResumeParser

from job\_searcher import JobSearcher

from ai\_matcher import SimpleAIMatcher

app = Flask(\_\_name\_\_)

app.secret\_key = 'your-secret-key-here' # Change this in production

app.config['UPLOAD\_FOLDER'] = 'uploads'

app.config['MAX\_CONTENT\_LENGTH'] = 5 \* 1024 \* 1024 # 5MB max file size

# Ensure upload directory exists

os.makedirs(app.config['UPLOAD\_FOLDER'], exist\_ok=True)

# Initialize components

parser = ResumeParser()

searcher = JobSearcher()

matcher = SimpleAIMatcher()

ALLOWED\_EXTENSIONS = {'pdf', 'docx'}

def allowed\_file(filename):

return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED\_EXTENSIONS

@app.route('/')

def index():

return render\_template('index.html')

@app.route('/upload', methods=['GET', 'POST'])

def upload\_resume():

if request.method == 'POST':

# Check if file was uploaded

if 'resume' not in request.files:

flash('No file selected')

return redirect(request.url)

file = request.files['resume']

if file.filename == '':

flash('No file selected')

return redirect(request.url)

if file and allowed\_file(file.filename):

filename = secure\_filename(file.filename)

filepath = os.path.join(app.config['UPLOAD\_FOLDER'], filename)

file.save(filepath)

# Get additional information from form

current\_salary = request.form.get('current\_salary', type=int)

location = request.form.get('location', '')

try:

# Process resume

resume\_data = parser.parse\_resume(filepath)

# Search for jobs

jobs = searcher.search\_by\_skills(

resume\_data['skills'],

location,

current\_salary

)

# Rank jobs

job\_matches = matcher.rank\_jobs(resume\_data, jobs, current\_salary)

# Clean up uploaded file

os.remove(filepath)

return render\_template('results.html',

resume\_data=resume\_data,

job\_matches=job\_matches[:10], # Top 10 matches

current\_salary=current\_salary)

except Exception as e:

flash(f'Error processing resume: {str(e)}')

if os.path.exists(filepath):

os.remove(filepath)

return redirect(url\_for('upload\_resume'))

else:

flash('Invalid file type. Please upload PDF or DOCX files only.')

return redirect(request.url)

return render\_template('upload.html')

@app.route('/api/quick-search')

def quick\_search():

"""API endpoint for quick job searches"""

keywords = request.args.get('keywords', '')

location = request.args.get('location', '')

if not keywords:

return jsonify({'error': 'Keywords required'}), 400

jobs = searcher.search\_jobs(keywords, location, max\_results=5)

# Convert to JSON-serializable format

job\_data = [

{

'title': job.title,

'company': job.company,

'location': job.location,

'salary': job.salary,

'url': job.url

}

for job in jobs

]

return jsonify({'jobs': job\_data})

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**Create Templates**

**templates/index.html**

<!DOCTYPE html>

<html>

<head>

<title>Mingus Resume AI</title>

<style>

body { font-family: Arial, sans-serif; max-width: 800px; margin: 0 auto; padding: 20px; }

.header { text-align: center; margin-bottom: 40px; }

.feature-box { background: #f5f5f5; padding: 20px; margin: 20px 0; border-radius: 8px; }

.cta-button { background: #007bff; color: white; padding: 15px 30px; text-decoration: none; border-radius: 5px; display: inline-block; margin: 20px 0; }

.cta-button:hover { background: #0056b3; }

</style>

</head>

<body>

<div class="header">

<h1>Mingus Career Advancement</h1>

<p>Connect your skills to higher-paying opportunities</p>

</div>

<div class="feature-box">

<h3>🤖 AI-Powered Resume Analysis</h3>

<p>Upload your resume and let our AI extract your skills, experience, and qualifications</p>

</div>

<div class="feature-box">

<h3>💼 Smart Job Matching</h3>

<p>Find jobs that match your skills and offer salary increases of 15-30%</p>

</div>

<div class="feature-box">

<h3>📈 Financial Impact Analysis</h3>

<p>See exactly how a new job would improve your monthly cash flow</p>

</div>

<div style="text-align: center;">

<a href="{{ url\_for('upload\_resume') }}" class="cta-button">Get Started - Upload Resume</a>

</div>

</body>

</html>

**templates/upload.html**

<!DOCTYPE html>

<html>

<head>

<title>Upload Resume - Mingus</title>

<style>

body { font-family: Arial, sans-serif; max-width: 600px; margin: 0 auto; padding: 20px; }

.form-group { margin: 20px 0; }

label { display: block; margin-bottom: 5px; font-weight: bold; }

input, select { width: 100%; padding: 10px; border: 1px solid #ddd; border-radius: 4px; }

.upload-area { border: 2px dashed #ddd; padding: 40px; text-align: center; margin: 20px 0; border-radius: 8px; }

.upload-area:hover { border-color: #007bff; }

.submit-btn { background: #007bff; color: white; padding: 15px 30px; border: none; border-radius: 5px; cursor: pointer; width: 100%; }

.submit-btn:hover { background: #0056b3; }

.flash-messages { margin: 20px 0; }

.flash-message { padding: 10px; background: #f8d7da; color: #721c24; border-radius: 4px; }

</style>

</head>

<body>

<h1>Upload Your Resume</h1>

<div class="flash-messages">

{% with messages = get\_flashed\_messages() %}

{% if messages %}

{% for message in messages %}

<div class="flash-message">{{ message }}</div>

{% endfor %}

{% endif %}

{% endwith %}

</div>

<form method="POST" enctype="multipart/form-data">

<div class="form-group">

<label for="resume">Resume File (PDF or DOCX):</label>

<div class="upload-area">

<input type="file" name="resume" id="resume" accept=".pdf,.docx" required>

<p>Drag and drop your resume here or click to browse</p>

</div>

</div>

<div class="form-group">

<label for="current\_salary">Current Annual Salary (optional):</label>

<input type="number" name="current\_salary" id="current\_salary" placeholder="e.g., 50000">

</div>

<div class="form-group">

<label for="location">Preferred Job Location:</label>

<select name="location" id="location">

<option value="">Any Location</option>

<option value="Atlanta">Atlanta, GA</option>

<option value="Houston">Houston, TX</option>

<option value="Washington DC">Washington, DC</option>

<option value="Dallas">Dallas, TX</option>

<option value="New York">New York, NY</option>

<option value="Philadelphia">Philadelphia, PA</option>

<option value="Chicago">Chicago, IL</option>

<option value="Charlotte">Charlotte, NC</option>

<option value="Miami">Miami, FL</option>

<option value="Baltimore">Baltimore, MD</option>

</select>

</div>

<button type="submit" class="submit-btn">Analyze Resume & Find Jobs</button>

</form>

</body>

</html>

**templates/results.html**

<!DOCTYPE html>

<html>

<head>

<title>Job Matches - Mingus</title>

<style>

body { font-family: Arial, sans-serif; max-width: 1000px; margin: 0 auto; padding: 20px; }

.summary-box { background: #e8f5e8; padding: 20px; margin: 20px 0; border-radius: 8px; }

.job-card { background: white; border: 1px solid #ddd; margin: 15px 0; padding: 20px; border-radius: 8px; box-shadow: 0 2px 4px rgba(0,0,0,0.1); }

.match-score { float: right; background: #007bff; color: white; padding: 5px 10px; border-radius: 20px; font-size: 0.9em; }

.salary-increase { color: #28a745; font-weight: bold; }

.skills-match { background: #f8f9fa; padding: 10px; margin: 10px 0; border-radius: 4px; }

.apply-btn { background: #28a745; color: white; padding: 10px 20px; text-decoration: none; border-radius: 4px; display: inline-block; margin-top: 10px; }

.apply-btn:hover { background: #218838; }

.back-btn { background: #6c757d; color: white; padding: 10px 20px; text-decoration: none; border-radius: 4px; margin-bottom: 20px; display: inline-block; }

</style>

</head>

<body>

<a href="{{ url\_for('upload\_resume') }}" class="back-btn">← Upload Another Resume</a>

<h1>Your Career Opportunities</h1>

<div class="summary-box">

<h3>Resume Analysis Summary</h3>

<p><strong>Name:</strong> {{ resume\_data.name }}</p>

<p><strong>Skills Found:</strong> {{ resume\_data.skills|join(', ') }}</p>

<p><strong>Experience Entries:</strong> {{ resume\_data.experience|length }}</p>

{% if current\_salary %}

<p><strong>Current Salary:</strong> ${{ "{:,}".format(current\_salary) }}</p>

{% endif %}

</div>

<h2>Job Matches ({{ job\_matches|length }} found)</h2>

{% for match in job\_matches %}

<div class="job-card">

<div class="match-score">{{ "%.0f"|format(match.match\_score \* 100) }}% Match</div>

<h3>{{ match.job.title }}</h3>

<p><strong>Company:</strong> {{ match.job.company }}</p>

<p><strong>Location:</strong> {{ match.job.location }}</p>

<p><strong>Salary:</strong> {{ match.job.salary }}</p>

{% if match.salary\_increase > 5 %}

<p class="salary-increase">

💰 Potential {{ "%.1f"|format(match.salary\_increase) }}% salary increase!

</p>

{% endif %}

{% if match.matching\_skills %}

<div class="skills-match">

<strong>Matching Skills:</strong> {{ match.matching\_skills|join(', ') }}

</div>

{% endif %}

<p><strong>Why this matches:</strong> {{ match.reasons|join(', ') }}</p>

<p>{{ match.job.description[:300] }}...</p>

{% if match.job.url %}

<a href="{{ match.job.url }}" target="\_blank" class="apply-btn">View Job Details</a>

{% endif %}

</div>

{% endfor %}

{% if not job\_matches %}

<div class="job-card">

<h3>No matches found</h3>

<p>Try uploading a more detailed resume or check back later for new opportunities.</p>

</div>

{% endif %}

</body>

</html>

**Step 5: Configuration & Deployment (Day 10)**

**Create config.py**

import os

from dotenv import load\_dotenv

load\_dotenv()

class Config:

SECRET\_KEY = os.environ.get('SECRET\_KEY') or 'dev-secret-key'

ADZUNA\_APP\_ID = os.environ.get('ADZUNA\_APP\_ID')

ADZUNA\_API\_KEY = os.environ.get('ADZUNA\_API\_KEY')

UPLOAD\_FOLDER = 'uploads'

MAX\_CONTENT\_LENGTH = 5 \* 1024 \* 1024 # 5MB

**Create .env file**

SECRET\_KEY=your-super-secret-key-here

ADZUNA\_APP\_ID=your-adzuna-app-id

ADZUNA\_API\_KEY=your-adzuna-api-key

**Create requirements.txt**

Flask==2.3.3

PyPDF2==3.0.1

python-docx==0.8.11

requests==2.31.0

python-dotenv==1.0.0

transformers==4.35.0

torch==2.1.0

pandas==2.1.3

gunicorn==21.2.0

**Quick Start Instructions**

**1. Get Adzuna API Keys (Free)**

1. Go to https://developer.adzuna.com/
2. Sign up for free account
3. Create an app to get APP\_ID and API\_KEY
4. Add to .env file

**2. Test Locally**

# Install dependencies

pip install -r requirements.txt

# Run the application

python app.py

# Visit http://localhost:5000

**3. Deploy to Heroku (Free Tier)**

# Install Heroku CLI

# Create Heroku app

heroku create mingus-resume-ai

# Set environment variables

heroku config:set SECRET\_KEY=your-secret-key

heroku config:set ADZUNA\_APP\_ID=your-app-id

heroku config:set ADZUNA\_API\_KEY=your-api-key

# Deploy

git add .

git commit -m "Initial deployment"

git push heroku main

**4. Alternative: Deploy to PythonAnywhere (Free)**

1. Upload files to PythonAnywhere
2. Create web app with Flask
3. Set environment variables in web app settings
4. Update WSGI file to point to your app.py

**Total Cost Breakdown**

* **Development Time**: 10 days
* **Monthly Costs**: $0-50 (Heroku/PythonAnywhere hosting)
* **API Costs**: FREE (Adzuna free tier: 1,000 searches/month)
* **Total Monthly Operating Cost**: $0-50

This implementation gives you a fully functional MVP that can process resumes, search for jobs, and provide intelligent matching - all while staying within your ultra-budget requirements!

**Start Building Today - Immediate Setup Guide**

**Step 1: Environment Setup (Next 30 minutes)**

**A. Create Project Directory**

Open your terminal/command prompt and run:

# Create and navigate to project directory

mkdir mingus-resume-ai

cd mingus-resume-ai

# Create Python virtual environment

python -m venv venv

# Activate virtual environment

# On Windows:

venv\Scripts\activate

# On Mac/Linux:

source venv/bin/activate

# Verify Python is working

python --version

**B. Install Required Packages**

# Install core packages one by one to avoid conflicts

pip install PyPDF2==3.0.1

pip install python-docx==0.8.11

pip install requests==2.31.0

pip install flask==2.3.3

pip install python-dotenv==1.0.0

# Verify installations

pip list

**C. Create Project Structure**

# Create directories

mkdir templates

mkdir static

mkdir uploads

mkdir test\_files

# Create main Python files

touch resume\_parser.py

touch app.py

touch config.py

touch .env

touch requirements.txt

# Verify structure

ls -la

**Step 2: Build the Resume Parser (Next 60 minutes)**

**A. Create resume\_parser.py**

Copy this exact code into your resume\_parser.py file:

import PyPDF2

import docx

import re

import os

from typing import Dict, List

class ResumeParser:

def \_\_init\_\_(self):

# Expanded skills list focused on your target market

self.skills\_keywords = [

# Technical skills

'python', 'javascript', 'java', 'sql', 'html', 'css', 'react',

'node.js', 'excel', 'powerpoint', 'word', 'outlook', 'salesforce',

# Business skills

'project management', 'data analysis', 'marketing', 'sales',

'customer service', 'accounting', 'finance', 'budgeting',

# Soft skills

'communication', 'leadership', 'teamwork', 'problem solving',

'time management', 'organization', 'presentation', 'training'

]

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

"""Extract text from PDF or DOCX files"""

print(f"Processing file: {file\_path}")

try:

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

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

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

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

else:

raise ValueError("Unsupported file format. Use PDF or DOCX.")

except Exception as e:

print(f"Error extracting text: {e}")

return ""

def \_extract\_from\_pdf(self, file\_path: str) -> str:

"""Extract text from PDF"""

text = ""

try:

with open(file\_path, 'rb') as file:

pdf\_reader = PyPDF2.PdfReader(file)

print(f"PDF has {len(pdf\_reader.pages)} pages")

for page\_num, page in enumerate(pdf\_reader.pages):

page\_text = page.extract\_text()

text += page\_text + "\n"

print(f"Extracted {len(page\_text)} characters from page {page\_num + 1}")

except Exception as e:

print(f"Error reading PDF: {e}")

return text

def \_extract\_from\_docx(self, file\_path: str) -> str:

"""Extract text from DOCX"""

text = ""

try:

doc = docx.Document(file\_path)

print(f"DOCX has {len(doc.paragraphs)} paragraphs")

for paragraph in doc.paragraphs:

text += paragraph.text + "\n"

except Exception as e:

print(f"Error reading DOCX: {e}")

return text

def parse\_resume(self, file\_path: str) -> Dict:

"""Main parsing function - returns structured data"""

print(f"Starting to parse resume: {file\_path}")

text = self.extract\_text\_from\_file(file\_path)

if not text.strip():

print("Warning: No text extracted from file")

return {

'error': 'Could not extract text from file',

'name': 'Unknown',

'email': 'Not found',

'phone': 'Not found',

'experience': [],

'education': [],

'skills': [],

'raw\_text': ''

}

print(f"Extracted {len(text)} characters total")

parsed\_data = {

'name': self.\_extract\_name(text),

'email': self.\_extract\_email(text),

'phone': self.\_extract\_phone(text),

'experience': self.\_extract\_experience(text),

'education': self.\_extract\_education(text),

'skills': self.\_extract\_skills(text),

'raw\_text': text[:1000] + "..." if len(text) > 1000 else text # Truncate for display

}

print(f"Parsing complete. Found {len(parsed\_data['skills'])} skills")

return parsed\_data

def \_extract\_name(self, text: str) -> str:

"""Extract name (first non-empty line usually)"""

lines = [line.strip() for line in text.split('\n') if line.strip()]

for line in lines[:5]: # Check first 5 lines

# Skip lines that look like contact info

if not any(keyword in line.lower() for keyword in

['email', '@', 'phone', 'address', 'linkedin', 'github', 'http']):

# Skip lines that are too long (probably not a name)

if len(line) < 50 and len(line.split()) <= 4:

return line

return "Name not found"

def \_extract\_email(self, text: str) -> str:

"""Extract email address"""

email\_pattern = r'\b[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'

emails = re.findall(email\_pattern, text)

return emails[0] if emails else "Email not found"

def \_extract\_phone(self, text: str) -> str:

"""Extract phone number"""

# Multiple phone patterns

patterns = [

r'(\+?1[-.\s]?)?\(?([0-9]{3})\)?[-.\s]?([0-9]{3})[-.\s]?([0-9]{4})',

r'(\d{3})[-.](\d{3})[-.](\d{4})',

r'\((\d{3})\)\s\*(\d{3})[-.](\d{4})'

]

for pattern in patterns:

phones = re.findall(pattern, text)

if phones:

# Format the phone number

if isinstance(phones[0], tuple):

return ''.join(phones[0])

return phones[0]

return "Phone not found"

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

"""Extract work experience"""

experience\_keywords = [

'experience', 'employment', 'work history', 'professional experience',

'career history', 'work experience'

]

lines = text.split('\n')

experience\_section = []

in\_experience = False

for i, line in enumerate(lines):

line = line.strip()

# Check if we're entering experience section

if any(keyword in line.lower() for keyword in experience\_keywords):

in\_experience = True

continue

# Check if we're leaving experience section

elif in\_experience and any(keyword in line.lower() for keyword in

['education', 'skills', 'certifications', 'projects']):

break

# If we're in experience section and line looks like a job

elif in\_experience and line:

# Look for job titles (lines with years, company names, or job-like words)

if (re.search(r'\d{4}', line) or # Contains year

any(word in line.lower() for word in

['inc', 'llc', 'corp', 'company', 'university', 'hospital']) or

any(title in line.lower() for title in

['manager', 'analyst', 'specialist', 'coordinator', 'assistant',

'developer', 'engineer', 'consultant', 'representative'])):

experience\_section.append(line)

return experience\_section[:10] # Return up to 10 most relevant

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

"""Extract education information"""

education\_keywords = [

'education', 'degree', 'university', 'college', 'bachelor',

'master', 'phd', 'doctorate', 'associates'

]

lines = text.split('\n')

education\_section = []

in\_education = False

for line in lines:

line = line.strip()

# Check if we're entering education section

if any(keyword in line.lower() for keyword in education\_keywords):

in\_education = True

# If this line itself contains degree info, add it

if any(degree in line.lower() for degree in

['bachelor', 'master', 'phd', 'degree', 'associates']):

education\_section.append(line)

continue

# If we're in education section

elif in\_education and line:

# Add lines that look like education

if (any(word in line.lower() for word in

['university', 'college', 'institute', 'school']) or

any(degree in line.lower() for degree in

['bachelor', 'master', 'phd', 'bs', 'ba', 'ms', 'ma']) or

re.search(r'\d{4}', line)): # Contains graduation year

education\_section.append(line)

# Stop if we hit another section

elif any(keyword in line.lower() for keyword in

['experience', 'skills', 'certifications']):

break

return education\_section[:5] # Return up to 5 entries

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

"""Extract skills from resume"""

found\_skills = []

text\_lower = text.lower()

print("Searching for skills...")

for skill in self.skills\_keywords:

if skill in text\_lower:

found\_skills.append(skill.title())

print(f"Found skill: {skill}")

# Also look for programming languages and tools

additional\_skills = [

'c++', 'c#', 'php', 'ruby', 'swift', 'kotlin', 'typescript',

'mysql', 'postgresql', 'mongodb', 'aws', 'azure', 'docker',

'git', 'jira', 'slack', 'zoom', 'teams'

]

for skill in additional\_skills:

if skill in text\_lower:

found\_skills.append(skill.upper() if skill in ['aws', 'sql'] else skill.title())

# Remove duplicates while preserving order

unique\_skills = []

for skill in found\_skills:

if skill not in unique\_skills:

unique\_skills.append(skill)

return unique\_skills

# Test function

def test\_parser():

"""Test the parser with a sample file"""

parser = ResumeParser()

# Test with your own resume file

test\_file = input("Enter path to your test resume file (PDF or DOCX): ")

if os.path.exists(test\_file):

result = parser.parse\_resume(test\_file)

print("\n" + "="\*50)

print("RESUME PARSING RESULTS")

print("="\*50)

for key, value in result.items():

if key != 'raw\_text': # Skip raw text for cleaner output

print(f"\n{key.upper()}:")

if isinstance(value, list):

for item in value:

print(f" - {item}")

else:

print(f" {value}")

print("\n" + "="\*50)

else:

print(f"File not found: {test\_file}")

if \_\_name\_\_ == "\_\_main\_\_":

test\_parser()

**B. Test the Resume Parser Right Now**

1. **Save a test resume** to your test\_files folder (use your own resume or download a sample PDF/DOCX)
2. **Run the parser**:

python resume\_parser.py

1. **When prompted, enter the path to your test file**:

# Example paths:

test\_files/sample\_resume.pdf

test\_files/my\_resume.docx

**C. Expected Output**

You should see something like:

Processing file: test\_files/sample\_resume.pdf

PDF has 1 pages

Extracted 1247 characters from page 1

Starting to parse resume: test\_files/sample\_resume.pdf

Extracted 1247 characters total

Searching for skills...

Found skill: python

Found skill: excel

Found skill: project management

Parsing complete. Found 3 skills

==================================================

RESUME PARSING RESULTS

==================================================

NAME:

John Smith

EMAIL:

john.smith@email.com

PHONE:

(555) 123-4567

EXPERIENCE:

- Software Developer at ABC Company 2020-2023

- Junior Analyst at XYZ Corp 2018-2020

EDUCATION:

- Bachelor of Science in Computer Science

- University of Technology 2018

SKILLS:

- Python

- Excel

- Project Management

==================================================

**Step 3: Quick Fixes if Something Goes Wrong**

**A. Common Issues and Solutions**

**Issue: "ModuleNotFoundError: No module named 'PyPDF2'"**

# Solution: Reinstall packages

pip uninstall PyPDF2

pip install PyPDF2==3.0.1

**Issue: "Permission denied" or file access errors**

# Solution: Check file permissions

ls -la test\_files/

# Make sure your resume file is readable

chmod 644 test\_files/your\_resume.pdf

**Issue: "No text extracted from file"**

* Try a different PDF (some PDFs are image-based)
* Test with a Word document instead
* Use a simple text-based resume

**Issue: Python virtual environment not working**

# Alternative setup without virtual environment

pip install --user PyPDF2 python-docx requests flask python-dotenv

**B. Debugging Steps**

If the parser isn't finding skills/info correctly:

1. **Check raw text extraction**:

# Add this to test\_parser() function:

print(f"\nRAW TEXT (first 500 chars):")

print(result['raw\_text'][:500])

1. **Test with a simple resume**: Create a simple Word document with:

* Your name at the top
* Email and phone
* "Work Experience" section
* "Skills: Python, Excel, Project Management"

**Step 4: Build Basic Web Interface (Next 30 minutes)**

**A. Create Simple Flask App (app.py)**

from flask import Flask, render\_template, request, redirect, url\_for, flash

import os

from werkzeug.utils import secure\_filename

from resume\_parser import ResumeParser

app = Flask(\_\_name\_\_)

app.secret\_key = 'mingus-dev-key' # Change in production

app.config['UPLOAD\_FOLDER'] = 'uploads'

app.config['MAX\_CONTENT\_LENGTH'] = 5 \* 1024 \* 1024 # 5MB

ALLOWED\_EXTENSIONS = {'pdf', 'docx'}

parser = ResumeParser()

def allowed\_file(filename):

return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED\_EXTENSIONS

@app.route('/')

def index():

return '''

<h1>Mingus Resume Parser - MVP</h1>

<form action="/upload" method="post" enctype="multipart/form-data">

<p>

<label>Upload Resume (PDF or DOCX):</label><br>

<input type="file" name="resume" accept=".pdf,.docx" required>

</p>

<p>

<input type="submit" value="Analyze Resume">

</p>

</form>

'''

@app.route('/upload', methods=['POST'])

def upload\_resume():

if 'resume' not in request.files:

return 'No file uploaded', 400

file = request.files['resume']

if file.filename == '':

return 'No file selected', 400

if file and allowed\_file(file.filename):

filename = secure\_filename(file.filename)

filepath = os.path.join(app.config['UPLOAD\_FOLDER'], filename)

file.save(filepath)

try:

# Parse the resume

result = parser.parse\_resume(filepath)

# Clean up file

os.remove(filepath)

# Display results

html = f'''

<h1>Resume Analysis Results</h1>

<h2>Basic Info</h2>

<p><strong>Name:</strong> {result['name']}</p>

<p><strong>Email:</strong> {result['email']}</p>

<p><strong>Phone:</strong> {result['phone']}</p>

<h2>Skills Found ({len(result['skills'])})</h2>

<ul>

'''

for skill in result['skills']:

html += f'<li>{skill}</li>'

html += f'''

</ul>

<h2>Experience ({len(result['experience'])} entries)</h2>

<ul>

'''

for exp in result['experience']:

html += f'<li>{exp}</li>'

html += '''

</ul>

<p><a href="/">← Analyze Another Resume</a></p>

'''

return html

except Exception as e:

if os.path.exists(filepath):

os.remove(filepath)

return f'Error processing resume: {str(e)}', 500

return 'Invalid file type', 400

if \_\_name\_\_ == '\_\_main\_\_':

# Create uploads directory

os.makedirs('uploads', exist\_ok=True)

app.run(debug=True, port=5000)

**B. Test Web Interface**

# Run the web app

python app.py

# Open browser to http://localhost:5000

# Upload a resume and see results

**Step 5: Validate It Works (Next 15 minutes)**

**A. Success Checklist**

✅ Resume parser extracts text from PDF/DOCX  
✅ Parser finds name, email, phone  
✅ Parser identifies skills from resume  
✅ Parser finds work experience entries  
✅ Web interface accepts file uploads  
✅ Web interface displays parsed results

**B. Quick Validation Test**

1. Upload 2-3 different resume formats
2. Check that skills are being found correctly
3. Verify experience and education sections work
4. Make sure web interface doesn't crash

**Next Steps (Tomorrow)**

Once this is working, you'll add:

1. **Job search integration** (Adzuna API - free)
2. **Better web interface** (HTML templates)
3. **Job matching algorithm** (skills comparison)

**Total time invested so far: ~2.5 hours**  
**Total cost so far: $0**  
**Working features: Complete resume analysis**

This gives you a solid foundation to build on and proves the concept works before investing more time or money!