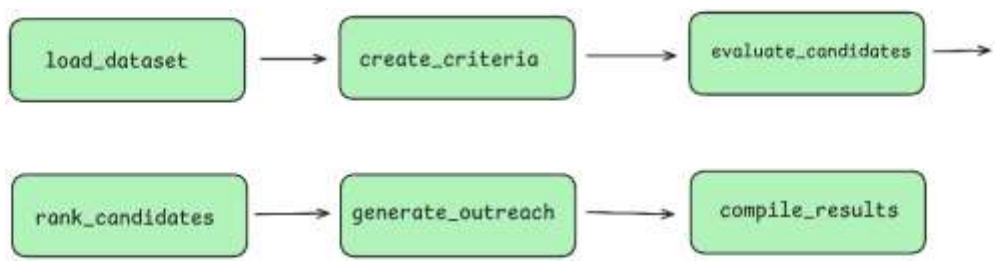
LangGraph Recruiting Agent - Setup & Documentation

Overview

This project implements an AI-powered recruiting agent using LangGraph that automatically scores and suggests top candidates for given job descriptions. The agent processes resume data from Kaggle and uses Google's Gemini AI to evaluate candidates against customized assessment criteria.

Architecture & Workflow

1. LangGraph Workflow Structure



Components

1. Load Dataset Node

- Downloads resume dataset from Kaggle using kagglehub
- Cleans and preprocesses the data
- Selects relevant columns for evaluation

2. create_criteria_node Logic

This node transforms a job description into structured evaluation criteria using LLM analysis.

Process Flow:

- Analyzes job description using Gemini AI
- Extracts key requirements:
    - Technical skills (required/preferred)
    - Experience requirements
    - Education qualifications
    - Soft skills
- Creates weighted assessment criteria

Criteria Structure:

```
json
{
"technical_skills": {
"required": ["Java", "Python", "SQL"],
"preferred": ["Spring", "React"],
"weight": 0.4  // 40% of total score
},
"experience": {
"min_years": 5,
"relevant_domains": ["software engineering"],
"weight": 0.3  // 30% of total score
},
"education": {
"required_degree": "Bachelor's",
"preferred_fields": ["Computer Science"],
"weight": 0.2  // 20% of total score
},
"soft_skills": {
"required": ["teamwork", "communication"],
"weight": 0.1  // 10% of total score
}
}
```

Key Logic:

- **Automated Extraction**: Uses LLM to identify skills, experience requirements, and qualifications from free-text job descriptions
- **Weighted Scoring**: Assigns importance weights to different criteria categories
- **Fallback Mechanism**: If LLM parsing fails, uses default criteria for software engineering roles

---

3. evaluate_candidates_node Logic

This is the core evaluation engine that scores each candidate against the established criteria.

Process Flow:

1. **Data Preparation**: Extracts relevant information from each candidate's resume
2. **Skill Matching**: Identifies technical skills from resume text
3. **Experience Calculation**: Computes years of experience from date ranges
4. **Multi-dimensional Scoring**: Evaluates across 4 dimensions
5. **Score Aggregation**: Combines weighted scores into final score

Detailed Scoring Logic:

A. Technical Skills Scoring (40% weight)

```
# Logic Implementation
required_matches = sum(1 for skill in required_skills
               if skill.lower() in candidate_skills)
preferred_matches = sum(1 for skill in preferred_skills
               if skill.lower() in candidate_skills)

skill_score = (required_matches / len(required_skills)) * 0.7 +
         (preferred_matches / len(preferred_skills)) * 0.3
```

Breakdown:

- **Required Skills**: 70% weight within technical skills
- **Preferred Skills**: 30% weight within technical skills

- **Matching Strategy**: Case-insensitive substring matching
- **Normalization**: Score capped at 1.0 (100%)

B. Experience Scoring (30% weight)

```
experience_score = min(experience_years / min_years, 1.0)
```

Logic:

- **Linear Scaling**: Score increases linearly with experience
- **Capping**: Maximum score of 1.0 when experience meets/exceeds requirement
- **Date Parsing**: Extracts years from start/end date strings

C. Education Scoring (20% weight)

```
education_score = 0.5  # Default

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

   education_score = 0.8

if any(field in education.lower() for field in preferred_fields):

   education_score = min(education_score + 0.2, 1.0)
```

Scoring Rules:

- **Base Score**: 0.5 (50%)
- **Degree Bonus**: +0.3 for bachelor's/master's/PhD
- **Field Relevance**: +0.2 for relevant field of study
- **Maximum**: 1.0 (100%)

D. Alignment Scoring (10% weight)

```
def calculate_alignment_score(self, career_objective: str, criteria: Dict) -> float:
   # Keyword matching approach
   software_keywords = ['software', 'development', 'programming', 'coding']
   keyword_matches = sum(1 for keyword in software_keywords
                  if keyword in objective_lower)

   alignment_score = (keyword_matches / len(software_keywords)) * 0.6 +
                (tech_matches / len(tech_terms)) * 0.4
```

Logic:

- **Career Objective Analysis**: Matches keywords in candidate's career objective
- **Domain Relevance**: Checks for software engineering terms
- **Technical Alignment**: Looks for specific technology mentions

---

4. Candidate Ranking Logic

Final Score Calculation:

```
total_score = (
    skill_score * criteria['technical_skills']['weight'] +      # 40%
    experience_score * criteria['experience']['weight'] +       # 30%
    education_score * criteria['education']['weight'] +         # 20%
    alignment_score * criteria['soft_skills']['weight']         # 10%
)
```

Ranking Process:

1. **Score Calculation**: Each candidate gets a total score (0-100)
2. **Sorting**: Candidates sorted by total score (descending)
3. **Top-K Selection**: Top 3 candidates selected
4. **Tie-Breaking**: Natural order (first occurrence wins)

Output Format

For each top candidate:

- Overall score (0-100%)
- Detailed breakdown by category
- Matched and missing skills
- Personalized outreach email
- Decision explanation

Quick Start

Prerequisites

- Python 3.9 or higher
- Google Cloud API key for Gemini AI

2. Environment Setup

```
GOOGLE_API_KEY=your_google_api_key_here
```

Getting Google API Key:

1. Go to [Google AI Studio]
2. Create a new API key
3. Copy the key to your .env file

3. Run the Application

```
python recruiting_agent.py
```

Scaling Recommendations

1. Parallel Processing with LangGraph

- LangGraph's parallel execution nodes for simultaneous candidate evaluation
- Benefits: Process multiple candidates concurrently, reducing evaluation time from hours to minutes
- Key Features: Batch processing nodes, conditional routing based on candidate volume

2. Distributed Workflow

- Deploy multiple LangGraph instances across different servers/containers
- Infrastructure: Use message queues (Redis/RabbitMQ) for coordination between instances
  Implement checkpointing for fault tolerance and resume capabilities

3. Caching & Optimization

- LLM Response Caching: Cache frequently used evaluation responses to reduce API calls
- Vector Database Integration: Use Pinecone or Weaviate for efficient skill matching and similarity search
- Result Persistence: Implement database caching for candidate evaluations and rankings

4. Real-time Processing

- Process candidates as they're uploaded rather than batch processing
- Live Updates: Use WebSocket connections for real-time status updates to users
- Event-driven Architecture: Trigger evaluations based on system events (new job posting, candidate upload)

5. Database & Storage

- Production Database: Replace in-memory storage with PostgreSQL or MongoDB for scalability
- Indexing Strategy: Implement proper indexing on candidate skills, experience, and evaluation scores
- Historical Tracking: Maintain evaluation history for model improvement and audit trails

6. Performance Monitoring & Observability

- Metrics Collection: Track processing time, accuracy rates, and system performance
- Bottleneck Identification: Monitor each pipeline stage to identify and resolve performance issues

- Health Checks: Implement automated monitoring for system reliability and uptime