Career Intelligence Assistant - Complete Project Guide

Project Overview

Purpose

Build a conversational AI system that analyzes resumes against job descriptions to help candidates understand:

- Skill gaps and missing qualifications
- Experience alignment with specific roles
- Strengths and competitive advantages
- Interview preparation insights
- Career fit assessment

Core User Flows

1. **Document Upload**: User uploads resume (PDF/DOCX) + multiple job descriptions
2. **Document Processing**: System extracts text, chunks intelligently, generates embeddings, stores in vector DB
3. **Conversational Q&A**: User asks questions like "What skills am I missing for the Senior ML Engineer role?" or "How does my Python experience align with Job #2?"
4. **Grounded Answers**: System retrieves relevant chunks, constructs context, generates LLM responses with citations

Key Capabilities

- Multi-document ingestion (resume + multiple JDs)
- Semantic search across career documents
- Comparative analysis (resume vs specific JD)
- Structured entity extraction (skills, experience, requirements)
- Citation-backed responses with source attribution

Recommended Tech Stack

Backend & Core

- **Language**: Python 3.11+
- **Framework**: FastAPI (async, modern, auto-docs)
- **LLM Provider**: Anthropic Claude (Sonnet 4) via API
- **Embedding Model**: OpenAI `text-embedding-3-small` or `all-MiniLM-L6-v2` (sentence-transformers)
- **Orchestration**: LangChain (for modularity and prompt management)

- **Vector Database**: ChromaDB (local-first, easy deployment) or Pinecone (cloud-native)

Supporting Infrastructure

- **Document Parsing**: `pypdf`, `python-docx`, `unstructured`
- **Validation**: Pydantic for schemas and guardrails
- **Testing**: pytest, pytest-asyncio
- **Observability**: LangSmith (optional) + structured logging (loguru)
- **Containerization**: Docker + docker-compose
- **UI**: Streamlit (rapid prototyping) or simple React frontend

Why This Stack?

- **Python**: Industry standard for AI/ML, rich ecosystem
- **FastAPI**: Production-ready, async, type-safe, auto-generated OpenAPI docs
- **Claude Sonnet 4**: Strong reasoning, long context, good at structured analysis
- **ChromaDB**: Simple setup, embeds in app, easy to productionize later
- **LangChain**: Mature RAG patterns, observability hooks, prompt templates

---

Project Structure

```
newpage-careerfit-rag-assistant_ayimer/
├── .gitignore
├── .dockerignore
├── Dockerfile
├── docker-compose.yml
├── pyproject.toml (or requirements.txt)
├── README.md
├── DECISIONS.md (design decisions log)
├── .env.example
│
├── app/
│   ├── __init__.py
│   ├── main.py (FastAPI app entry)
│   ├── config.py (settings, env vars)
│   ├── models.py (Pydantic schemas)
│   │
│   ├── ingestion/
│   │   ├── __init__.py
│   │   ├── parsers.py (PDF/DOCX extraction)
│   │   ├── chunker.py (text splitting strategies)
│   │   ├── pipeline.py (end-to-end ingestion)
│   │
│   ├── retrieval/
│   │   ├── __init__.py
│   │   ├── embeddings.py (embedding generation)
│   │   ├── vector_store.py (ChromaDB wrapper)
│   │   ├── retriever.py (query logic, reranking)
│   │
│   ├── rag/
│   │   ├── __init__.py
│   │   ├── prompts.py (prompt templates)
```

```
│   │   ├── chain.py (LangChain RAG chain)
│   │   ├── guardrails.py (input/output validation)
│   │
│   ├── api/
│   │   ├── __init__.py
│   │   ├── routes.py (FastAPI endpoints)
│   │   ├── dependencies.py (DI for vector store, LLM)
│   │
│   ├── utils/
│   │   ├── __init__.py
│   │   ├── logging.py (structured logging setup)
│   │   ├── metrics.py (observability helpers)
│
├── ui/
│   ├── streamlit_app.py (simple Streamlit UI)
│
├── tests/
│   ├── __init__.py
│   ├── test_ingestion.py
│   ├── test_retrieval.py
│   ├── test_rag.py
│   ├── test_api.py
│   ├── fixtures/ (sample resume + JDs for testing)
│
├── data/ (local storage, gitignored)
│   ├── uploads/
│   ├── vectordb/
│
├── notebooks/ (optional, for experimentation)
│   ├── exploration.ipynb
│
└── docs/
    ├── architecture.md
    ├── api_guide.md
```

Step-by-Step Implementation Plan

Phase 1: Foundation (Steps 1-4)

Step 1: Project Initialization

Goal: Set up basic project scaffolding and dependencies

Tasks:

- Create folder structure as defined above
- Create `.gitignore` (Python, env files, data/, vectordb/)
- Create `pyproject.toml` or `requirements.txt` with core dependencies:
- `fastapi, uvicorn, pydantic, pydantic-settingslangchain, langchain-anthropic, langchain-communitychromadb (or pinecone-client)openai (for embeddings)pypdf, python-docx, unstructuredpython-multipart (file uploads)loguru, pytest, pytest-asyncio, httpxstreamlit (UI)`
- Create `.env.example` with placeholders: `ANTHROPIC_API_KEY`, `OPENAI_API_KEY`, `VECTOR_DB_PATH`
- Initialize git repo

Focus: Clean setup, reproducible environment

Step 2: Configuration & Core Models

Goal: Define app configuration and Pydantic schemas

Tasks:

- Create `app/config.py`:
    - Use `pydantic-settings` for env-based config
    - Define settings: API keys, model names, chunk sizes, retrieval params
    - Add validation for required env vars
- Create `app/models.py`:
    - `DocumentUpload` (file metadata, type: resume/jd)
    - `QueryRequest` (user question, optional JD filter)
    - `QueryResponse` (answer, sources, confidence)
    - `ChunkMetadata` (doc_id, doc_type, page, char_range)

Focus: Type safety, configuration as code, clear contracts

Step 3: Document Ingestion Pipeline

Goal: Parse uploaded documents, extract clean text

Tasks:

- Create `app/ingestion/parsers.py`:
    - `parse_pdf()`: Use pypdf or pdfplumber, extract text + metadata
    - `parse_docx()`: Use python-docx, preserve structure
    - Unified `parse_document()` dispatcher
    - Handle errors gracefully (corrupt files, unsupported formats)
- Create `app/ingestion/chunker.py`:
    - Implement semantic chunking strategy (e.g., RecursiveCharacterTextSplitter)
    - **Decision Point**: Chunk size (512-1024 tokens), overlap (50-100 tokens)
    - Preserve document metadata in each chunk
    - Special handling for structured content (resume sections, JD requirements)
- Create `app/ingestion/pipeline.py`:
    - `ingest_document()`: parse → chunk → return chunks with metadata
    - Add logging at each stage

Focus Areas:

- **Chunking Strategy**: Balance between context preservation and retrieval precision
    - Consider section-aware chunking (e.g., keep "Skills" section together)
    - Preserve job description structure (requirements, responsibilities)

- **Metadata Enrichment**: Tag chunks with doc_type, section, importance

Testing: Create test fixtures (sample resume PDF, sample JD), write unit tests

---

Step 4: Vector Storage & Embeddings

Goal: Generate embeddings and store in vector database

Tasks:

- Create `app/retrieval/embeddings.py`:
  - Wrapper for embedding model (OpenAI or sentence-transformers)
  - Batch embedding generation
  - Caching strategy (optional)
- Create `app/retrieval/vector_store.py`:
  - ChromaDB client initialization
  - `add_documents()`: embed chunks + store with metadata
  - `search()`: vector similarity search + metadata filtering
  - Collection management (separate collections per session/user?)
- Integration: Connect ingestion pipeline to vector store

Focus Areas:

- **Embedding Model Choice**:
  - `text-embedding-3-small`: Cost-effective, OpenAI ecosystem
  - `all-MiniLM-L6-v2`: Free, local, decent quality
  - Document trade-offs in DECISIONS.md
- **Metadata Filtering**: Enable queries like "search only in resume" or "compare to JD #2"

Testing: Test embedding generation, vector search recall

Phase 2: RAG Pipeline (Steps 5-7)

Step 5: Retrieval Logic

Goal: Implement smart retrieval with context management

Tasks:

- Create `app/retrieval/retriever.py`:
  - `retrieve_for_query()`:
    - Embed user query
    - Perform vector search (top-k=5-10)
    - Apply metadata filters (doc_type, specific JD)

        - Implement MMR (Maximal Marginal Relevance) for diversity
    - Hybrid retrieval (optional): Combine vector search + keyword (BM25)
    - Context window management: Fit retrieved chunks within LLM limits

Focus Areas:

- **Retrieval Strategy**:
    - Top-k selection (start with 5-7)
    - Reranking (optional, use LLM or cross-encoder)
    - Handling multi-document queries (resume + specific JD vs all JDs)
- **Context Management**:
    - Prioritize most relevant chunks
    - Include document metadata for grounding

Testing: Test retrieval quality with sample queries

Step 6: Prompt Engineering & LLM Integration

Goal: Build effective prompts and RAG chain

Tasks:

- Create `app/rag/prompts.py`:
    - Define prompt templates using LangChain `PromptTemplate`:
        - System prompt: Role as career advisor, focus on factual analysis
        - Context injection template: Format retrieved chunks with source info
        - Few-shot examples for skill gap analysis, alignment questions
    - Multiple prompt variants for different question types (skill gaps, experience alignment, interview prep)
- Create `app/rag/chain.py`:
    - Build LangChain RAG chain:
        - Retriever → Context formatter → LLM → Response parser
    - Use Claude Sonnet 4 with temperature=0.3 for consistency
    - Implement streaming (optional, for UX)
    - Extract source citations from response

Focus Areas:

- **Prompt Design**:
    - Clear instructions: "Compare resume skills to JD requirements"
    - Output structure: "List missing skills in bullet points, cite sources"
    - Grounding: "Only use information from provided documents"
- **Context Engineering**:
    - Format: "Resume Section: [text]\nJob Description: [text]"
    - Include metadata for transparency
- **Quality Controls**:
    - Instruct model to say "I don't have enough information" if context is insufficient

- Prevent hallucination: "Do not infer skills not mentioned"

Testing: Test prompt variations, measure answer quality manually

---

Step 7: Guardrails & Validation

Goal: Add input/output safety checks

Tasks:

- Create `app/rag/guardrails.py`:
  - Input validation:
    - Query length limits (< 500 chars)
    - Profanity/injection detection (basic regex or simple classifier)
    - Rate limiting logic (optional)
  - Output validation:
    - Check for harmful content (unlikely in this domain, but good practice)
    - Validate citation format
    - Fallback responses for errors
  - Pydantic validators for all user inputs

Focus: Safety, reliability, graceful degradation

Phase 3: API & UI (Steps 8-9)

Step 8: FastAPI Endpoints

Goal: Expose functionality via REST API

Tasks:

- Create `app/api/dependencies.py`:
  - Dependency injection for vector store, LLM client, config
- Create `app/api/routes.py`:
  - `POST /upload` - Upload resume or JD, trigger ingestion, return doc_id
  - `POST /query` - Accept question + optional filters, return answer + sources
  - `GET /documents` - List uploaded documents
  - `DELETE /documents/{doc_id}` - Remove document
  - `GET /health` - Health check
- Create `app/main.py`:
  - FastAPI app initialization
  - CORS middleware (for frontend)
  - Exception handlers
  - Startup event: Initialize vector store
  - Shutdown event: Cleanup

Focus: Clean API design, async handlers, proper error responses (422, 500)

Testing: Write integration tests with TestClient

Step 9: Simple UI

Goal: Build minimal but functional interface

Tasks:

- Create `ui/streamlit_app.py`:
    - File upload widgets for resume + JDs
    - Display uploaded documents
    - Chat interface for questions
    - Show answers with source citations
    - Simple styling
- Alternative: Basic HTML/JS frontend calling FastAPI backend

Focus: Usability over aesthetics, demonstrate functionality

Phase 4: Observability & Production Readiness (Steps 10-12)

Step 10: Logging & Observability

Goal: Add comprehensive logging and metrics

Tasks:

- Create `app/utils/logging.py`:
    - Configure loguru with structured logs (JSON format)
    - Log levels: INFO for requests, DEBUG for retrieval, ERROR for failures
    - Request ID tracking across components
- Create `app/utils/metrics.py`:
    - Track key metrics:
        - Document ingestion time
        - Retrieval latency
        - LLM call latency
        - Query success rate
    - Optional: Export to Prometheus format or simple CSV
- Integrate LangSmith (optional):
    - Trace LangChain calls
    - Log prompts, completions, retrieval results

Focus Areas:

- **Observability Strategy**:

    - Log all LLM calls with input/output
    - Track retrieval quality (chunks returned, relevance scores)
    - Monitor token usage and costs
    - Alert on errors or degraded performance

Testing: Verify logs are generated, metrics are collected

Step 11: Testing Suite

Goal: Comprehensive test coverage

Tasks:

- Create test fixtures in `tests/fixtures/`:
    - Sample resume (PDF)
    - Sample job descriptions (2-3, covering different roles)
- Write unit tests:
    - `test_ingestion.py`: Test parsers, chunkers with fixtures
    - `test_retrieval.py`: Test embedding, vector search, retrieval logic
    - `test_rag.py`: Test prompt formatting, chain execution (mock LLM)
    - `test_api.py`: Integration tests for all endpoints
- Write E2E test:
    - Upload documents → ask question → validate response structure

Focus: Edge cases, error handling, regression prevention

Step 12: Containerization

Goal: Dockerize for consistent deployment

Tasks:

- Create `Dockerfile`:
    - Multi-stage build (builder + runtime)
    - Python 3.11 slim base image
    - Copy dependencies, install
    - Copy app code
    - Expose port 8000
    - CMD to run FastAPI with uvicorn
- Create `docker-compose.yml`:
    - Service for FastAPI backend
    - Volume mounts for data persistence
    - Environment variables from .env
    - Optional: Separate ChromaDB service
- Create `.dockerignore`:
    - Exclude data/, .env, **pycache**, .git

Testing: Build and run container, verify all functionality works

---

Phase 5: Documentation (Step 13)

Step 13: Comprehensive Documentation

Goal: Document everything for reviewers

Tasks:

Create `README.md` with sections:

1. **Project Overview**
   - Brief description of Career Intelligence Assistant
   - Problem solved, key features
   - Tech stack summary
2. **Quick Start**

```
3. # Clone repo
4. # Set up .env (copy from .env.example)
5. # Install dependencies (pip install -r requirements.txt)
6. # Run: uvicorn app.main:app --reload
7. # Or: docker-compose up
8. # Open UI: streamlit run ui/streamlit_app.py
```

9. **Architecture Overview**
   - High-level diagram (draw.io or ASCII art):
   - ```
     User → UI → FastAPI → [Ingestion Pipeline] → Vector DB
     ↓               [RAG Chain] → LLM → Response
     ↑               [Retrieval]
     ```
   - Component descriptions (Ingestion, Retrieval, RAG, API)
10. **RAG/LLM Approach & Decisions**
    - **LLM Selection**: "Chose Claude Sonnet 4 for strong reasoning, long context (200k), cost-efficiency"
    - **Embedding Model**: "Used text-embedding-3-small for balance of quality and cost; considered sentence-transformers for local deployment"
    - **Vector Database**: "ChromaDB for local-first development, easy to migrate to Pinecone/Weaviate for scale"
    - **Orchestration**: "LangChain for RAG chain abstraction, prompt templates, observability hooks"
    - **Chunking Strategy**: "Semantic chunking with 512 token chunks, 50 token overlap; preserves context while enabling precise retrieval"
    - **Retrieval Approach**: "Top-5 vector search with MMR for diversity; metadata filtering for document-specific queries"
    - **Prompt Engineering**: "System prompt emphasizes factual grounding, structured output; few-shot examples for skill gap analysis"
    - **Context Management**: "Limit context to ~4000 tokens to balance relevance and cost; prioritize most similar chunks"
    - **Guardrails**: "Input validation (length, content); output validation (citation format); fallback for low-confidence answers"
    - **Observability**: "Structured logging with loguru; LangSmith integration for trace analysis; metrics on latency, token usage"
     ```

11. **Key Technical Decisions**
    - Why FastAPI over Flask: "Async, type-safe, auto-docs"
    - Why ChromaDB: "Local-first, low ops overhead, good for MVP"
    - Chunking trade-offs: "Smaller chunks = precise retrieval but fragmented context; chose 512 tokens as sweet spot"
    - Retrieval strategy: "Vector-only for MVP; hybrid (vector + BM25) is future improvement"
12. **Engineering Standards**
    - Code structure: "Modular, separation of concerns (ingestion/retrieval/RAG)"
    - Type safety: "Pydantic models for all data contracts"
    - Error handling: "Graceful degradation, structured error responses"
    - Testing: "Unit tests for components, integration tests for API, E2E test for user flow"
    - Standards skipped: "Full CI/CD pipeline (would add GitHub Actions in production)"
13. **AI Tooling Usage**
    - "Used Claude Code for rapid scaffolding, boilerplate generation"
    - "Used GitHub Copilot for test case generation, documentation"
    - "Maintained quality through: code review, type checking (mypy), linting (ruff)"
    - "Do's: Use AI for repetitive code, tests, docs; Don'ts: Blindly accept generated logic, skip manual review"
    - "Ensured maintainability: Clear naming, comments on complex logic, ADR for decisions"
14. **Productionization for AWS/GCP/Azure**
    - **Compute**:
        - AWS: ECS Fargate or Lambda (for API), EC2 for ChromaDB
        - GCP: Cloud Run, GKE
        - Azure: Container Instances, AKS
    - **Vector DB**:
        - Migrate to managed: Pinecone, Weaviate Cloud, or pgvector on RDS/Cloud SQL
    - **Storage**: S3/GCS/Blob Storage for uploaded documents
    - **Secrets**: AWS Secrets Manager, GCP Secret Manager, Azure Key Vault
    - **Scaling**:
        - Horizontal scaling for API (load balancer)
        - Async processing for ingestion (SQS/Pub-Sub + workers)
        - Caching layer (Redis) for embeddings
    - **Monitoring**: CloudWatch/Stackdriver/Azure Monitor + custom metrics
    - **CI/CD**: GitHub Actions → Docker build → ECR/GCR/ACR → Deploy
    - **Cost optimization**: Use smaller LLM (Haiku) for simple queries, cache embeddings
15. **Future Improvements**
    - "Hybrid retrieval (vector + BM25) for better recall"
    - "Reranking with cross-encoder for precision"
    - "Multi-query retrieval for complex questions"
    - "Fine-tune embedding model on career-specific corpus"
    - "Add conversation history for multi-turn dialogue"
    - "Structured output parsing for skills/requirements extraction"
    - "A/B testing framework for prompt variants"

- "User feedback loop to improve retrieval and prompts"
    - "Advanced UI with highlighting, side-by-side comparison"
    - "Integration with LinkedIn API for profile enrichment"

Create `DECISIONS.md` (Architecture Decision Records):

- Document major decisions with format:
- `## ADR-001: LLM Selection**Context**: Need LLM for career analysis**Decision**: Use Claude Sonnet 4**Rationale**: Superior reasoning, long context, cost-effective**Alternatives**: GPT-4 (more expensive), Llama 3 (requires hosting)**Consequences**: Vendor lock-in, API dependency`

Create `docs/architecture.md`:

- Detailed architecture diagram
- Data flow diagrams
- Component interactions

Create `docs/api_guide.md`:

- API endpoint documentation (auto-generated from FastAPI also good)
- Example requests/responses
- Error codes

Key Evaluation Criteria Alignment

Chunking

- **Where**: `app/ingestion/chunker.py`
- **Focus**: Semantic chunking, preserve document structure, optimal chunk size
- **Document**: Trade-offs in README, experiments in notebooks

Embedding Model Selection

- **Where**: `app/retrieval/embeddings.py`, config
- **Focus**: Quality vs cost vs latency, domain relevance
- **Document**: Comparison table in README

Retrieval Approach

- **Where**: `app/retrieval/retriever.py`
- **Focus**: Vector search, metadata filtering, diversity (MMR), potential hybrid
- **Document**: Strategy explanation in README, metrics in observability

Prompt Engineering

- **Where**: `app/rag/prompts.py`

- **Focus**: Clear instructions, grounding, output structure, few-shot examples
- **Document**: Prompt templates in code, design rationale in DECISIONS.md

Context Management

- **Where**: `app/rag/chain.py`, retriever
- **Focus**: Token limits, chunk prioritization, context formatting
- **Document**: Strategy in README, token usage in metrics

Guardrails

- **Where**: `app/rag/guardrails.py`
- **Focus**: Input validation, output safety, error handling
- **Document**: Guardrail policies in README

Quality Controls

- **Where**: Throughout, especially `tests/`
- **Focus**: Testing, validation, monitoring answer quality
- **Document**: Testing approach in README, test results

Observability

- **Where**: `app/utils/logging.py`, `app/utils/metrics.py`
- **Focus**: Structured logging, metrics, tracing, debugging
- **Document**: Observability strategy in README, setup instructions

---

Development Workflow with AI Tools

Recommended Approach

1. **Use Cursor/Claude Code for**:
   - Initial file/folder creation
   - Boilerplate code (FastAPI routes, Pydantic models)
   - Test case scaffolding
   - Documentation templates
2. **Manual Review & Refinement**:
   - Review all AI-generated code for correctness
   - Refactor for clarity and maintainability
   - Add domain-specific logic manually
   - Write complex business logic yourself
3. **Prompting Strategy**:
   - Be specific: "Create a FastAPI endpoint for document upload that validates file type and size"
   - Iterate: "Now add error handling for corrupted PDFs"
   - Request explanations: "Explain why you chose this chunking approach"
4. **Quality Assurance**:

- Run type checker (mypy) on AI-generated code
- Use linter (ruff) for style consistency
- Write tests for all AI-generated components
- Code review mindset: Would I accept this in a PR?

Timeline Estimate

- **Steps 1-2** (Setup): 1-2 hours
- **Steps 3-4** (Ingestion + Vector): 3-4 hours
- **Steps 5-7** (RAG Pipeline): 4-5 hours
- **Steps 8-9** (API + UI): 2-3 hours
- **Steps 10-12** (Observability + Testing + Docker): 3-4 hours
- **Step 13** (Documentation): 2-3 hours

Total: 15-21 hours (comfortable completion before Feb 12 deadline)

Final Checklist Before Submission

- [ ] All code runs without errors
- [ ] Docker container builds and runs successfully
- [ ] Tests pass (run `pytest`)
- [ ] README is complete with all required sections
- [ ] DECISIONS.md documents key choices
- [ ] Code is clean, well-commented, type-hinted
- [ ] `.env.example` provided (no secrets in repo)
- [ ] Sample documents work end-to-end
- [ ] Observability logs visible
- [ ] GitHub repo is public and well-organized

This guide provides a complete roadmap. Feed each step to Cursor incrementally, review the output, iterate, and maintain your engineering standards throughout. Good luck!