CareerCompass: Technical Implementation Plan

Objective: Build a scalable, containerized career guidance system for Computer Engineering [
SOURCE]   [ETL LAYER]   [PERSISTENCE]   [API LAYER]   [PRESENTATION]

job.csv -------> Python/Pandas -------> SQLite (.db) <-------> FastAPI <-------> React (Vite)

(Raw Data)   (Sanitization)   (Structured)   (Logic)   (User)students, leveraging an ETL pipeline, a RESTful API, and a stateful React frontend.

1. System Architecture Overview

The application follows a standard **ETL -> Persistence -> API -> UI** architecture:

- **Data Tier:** job.csv (Raw) → Python/Pandas (ETL) → SQLite (Structured Storage).
- **Logic Tier:** FastAPI (RESTful API) with Pydantic for schema enforcement.
- **Presentation Tier:** React (Vite-based) using Tailwind CSS for responsive design.
- **Infrastructure:** Docker Compose for multi-container orchestration.

2. Phase 1: Ingestion & Data Sanitization (ETL)

Focus: Data integrity and schema standardization.

1. **Schema Enforcement**:
   - Read job.csv into a Pandas DataFrame.
   - Normalize headers: Lowercase, strip whitespace, replace spaces with underscores.
2. **Sanitization Logic**:
   - **Type Casting**: Ensure salary ranges are strings and handle null values (NaN to 'N/A').
   - **Normalization**: Cast required_skills and specialization to lowercase to facilitate case-insensitive SQL LIKE queries.
3. **Validation**: Implement basic assertions to ensure no critical fields (Job Title, Specialization) are missing post-clean.

3. Phase 2: Persistence Layer (SQLite)

Focus: Transitioning from flat-file storage to relational persistence.

1. **DB Initialization**: Establish a connection to cpe_careers.db using sqlite3.
2. **Storage Logic**: Use df.to_sql() with index=False and if_exists='replace' to persist the cleaned DataFrame.
3. **Optimization**: (Optional) Add a basic index on the job_title or specialization columns if the dataset grows beyond 1,000 records.

4. Phase 3: RESTful API Layer (FastAPI)

Focus: Secure and performant data delivery.

1. **Service Setup**: Initialize FastAPI with uvicorn.
2. **Endpoint Design**:
   - GET /api/jobs: Fetch all records.
   - GET /api/jobs/search?skill={str}: Filter jobs based on skills.
3. **Middleware**: Configure **CORS** (Cross-Origin Resource Sharing) to allow requests from the React development server (localhost:5173).
4. **Error Handling**: Implement 404 and 500 status codes for database connection failures or empty results.

5. Phase 4: Frontend Development (React + Tailwind)

Focus: Declarative UI and efficient state management.

1. **Component Architecture**:
   - <SearchBar />: Controlled input for real-time filtering.
   - <JobGrid />: Logic for mapping data to the UI.
   - <JobCard />: Atomic component for individual job data visualization.
2. **Data Fetching**:
   - Implement axios or native fetch.
   - **Optimization**: Implement "Debouncing" on the search input to limit API calls during rapid typing.
3. **State Management**: Use useState for local data caching and useEffect for the lifecycle management of API calls.

6. Phase 5: Containerization (DevOps)

Focus: Environment parity and reproducibility.

1. **Dockerfile (Backend)**:
   - Base: python:3.9-slim.
   - Requirements: Install fastapi, uvicorn, pandas, and sqlite3.
2. **Dockerfile (Frontend)**:
   - Base: node:18-alpine.
   - Process: Build optimized production assets.
3. **Docker Compose**:
   - services.backend: Links to the database volume.
   - services.frontend: Maps port 3000 to the container.
   - networks: Establish a shared bridge network for internal communication.

7. Definition of Done (DoD)

- [ ] Data is cleaned and persisted to a .db file via a single Python script.

- [ ] API successfully returns JSON data for filtered searches.
- [ ] Frontend renders responsive cards with no console errors.
- [ ] docker-compose up launches the entire stack without manual configuration.