Architecture Analysis: Stack Choices & Design Justifications

Stack Justification

Frontend (HTML, CSS, JS) This choice prioritizes accessibility and broad browser compatibility. A lightweight frontend minimizes dependencies and keeps the user experience responsive. For a data-sorting application, JavaScript provides sufficient interactivity without the overhead of a heavy framework, allowing direct DOM manipulation for real-time UI updates as sorting results change.

Backend (Flask/Python) Flask is an excellent choice for this use case because it's minimal yet powerful. Python's rich ecosystem (Pandas, data manipulation libraries) makes it ideal for handling data pipeline logic. Flask's simplicity allows to focus on core business logic rather than framework complexity. The lightweight nature means faster startup times and easier debugging during development.

Database (SQLite/SQLite.db) SQLite is pragmatic for this application stage. It requires zero server configuration, making development and deployment straightforward. For a sorting and data processing application, SQLite handles moderate data volumes well. The file-based approach ("taxi-trips.db") also simplifies backups and version control, though this choice trades scalability, a future migration to PostgreSQL would be straightforward if data volume or concurrent users grows significantly.

Offline Processing (Pandas + Python Pipeline) Pandas is the standard for tabular data manipulation. Using it for cleaning, validation, and transformation keeps data processing logic centralized and testable. The pipeline approach (separate modules for clean, database operations, processing, and transformation) enables modularity, each step can be developed and tested independently.

Schema & Architectural Trade-offs

Separation of Concerns The architecture splits offline processing from online serving. This trades some architectural simplicity for resilience and scalability, you can update/reprocess data without impacting user availability. The downside is additional deployment complexity (two separate pipelines to manage).

JSON API Response Using JSON responses from Flask provides language-agnostic communication and simplifies frontend integration. This is standard practice, though it adds serialization overhead compared to binary protocols. For a data sorting application, this overhead is negligible.

Custom Algorithm Implementation Including "Custom Algorithm (Quick and Manual Sort)" in the server layer suggests business logic lives on the backend. This is more secure than client-side sorting (sensitive data stays protected) but increases server load. The trade-off: slower responses for very large datasets versus data security and business logic protection.

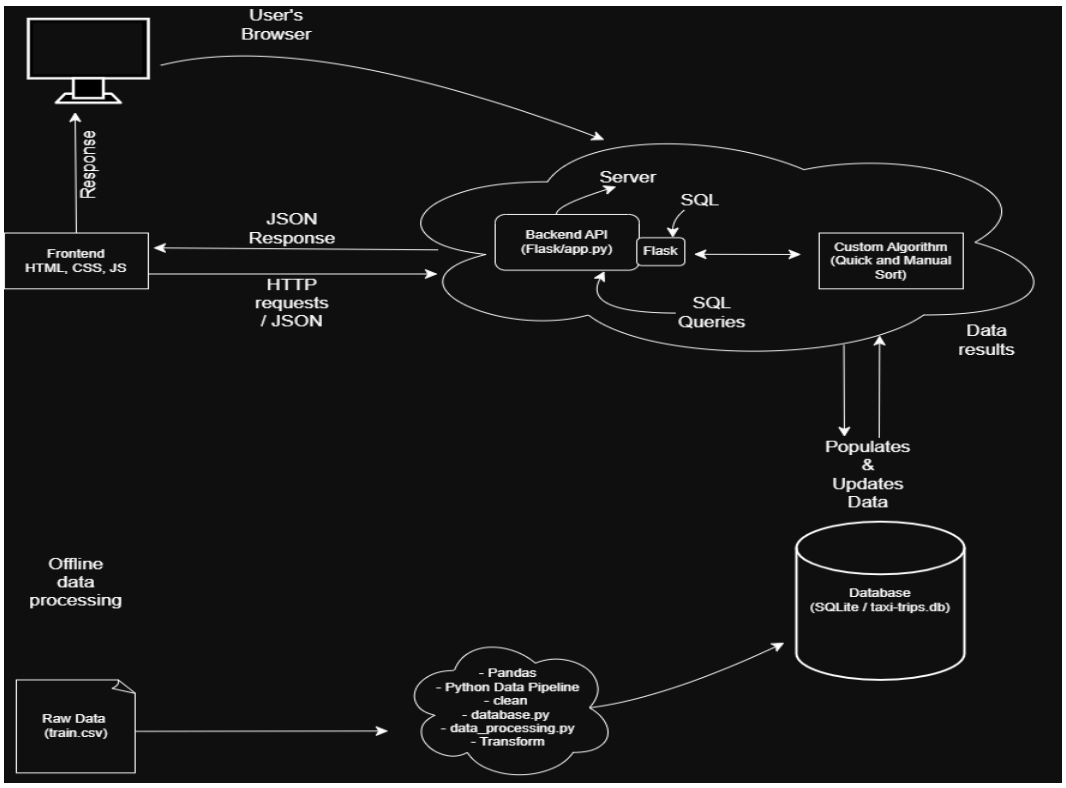
Database-Level SQL Queries Executing SQL queries directly rather than loading all data into memory keeps resource usage predictable. This is crucial for scalability but means you can't leverage complex in-memory operations without careful design. The Custom Algorithm box suggests some processing happens post-query, which makes sense for refinements.

Potential Design Tensions

Offline vs. Online: Raw data flows through offline processing before database population. This ensures data quality but means users always work with processed data. If real-time raw data is ever needed, you'd need to reconsider this split.

Complexity vs. Simplicity: The pipeline approach (Pandas, separate modules, custom algorithms) adds complexity compared to a single monolithic script. However, this complexity is justified by maintainability and testability for a growing application.

Scalability Ceiling: SQLite and single-server Flask can handle moderate load (~1000s of concurrent users), but you'd hit limits with massive scale.


User's Browser
Response
Server
SQL
JSON Response
Backend API (Flask/app.py)
Flask
Custom Algorithm (Quick and Manual Sort)
Frontend HTML, CSS, JS
HTTP requests / JSON
SQL Queries
Data results
Populates & Updates Data
Offline data processing
Database (SQLite / taxi-trips.db)
- Pandas
- Python Data Pipeline
- clean
- database.py
- data_processing.py
- Transform
Raw Data (train.csv)