Sat 18 Oct

Think first:
Who will use this platform?
What information do users need?
What is their core purpose — what do they want to do, and what kind of information do they want to see?
Where does this information come from?
Spend at least three hours studying how Langfuse and MLflow are implemented, focusing on the visualization part.

Currently, the data source is CSV, but in the future it will come from the production server/database, so the interface should be reserved for that.
Assume it's from a database — the fetching logic should be consistent.
Questions like how the training data is generated or the actual production data volume — not necessary to know right now.

Thoughts about backend:
main.py boots the app and wires routes.
routes.py exposes HTTP endpoints and only handles request/response, delegating real logic to services.py.
The services talk to data_loader.py, which reads and caches the CSVs under data/(experiments, trials, runs).
Data shapes and enums live in models.py (Pydantic), so everything shares one source of truth.
config.py holds settings (paths, toggles).
utils.py is for small helpers.
The dependency flow is one-way: routes → services → data_loader → CSVs, with models referenced across layers.
main.py only does assembly: imports routes and loads config; it must not be imported backward by business modules.
routes.py handles the HTTP layer; use models.py for type validation.
Actual processing calls services.py.
services.py implements the logic — read/write, compute, etc. — using data_loader.py.
data_loader.py manages data access (CSV reading, caching, simple filtering).
models.py stores Pydantic + Enum (like TrialStatus) so everyone depends on one definition and avoids circular imports.

http://127.0.0.1:8000/ → returns JSON
http://127.0.0.1:8000/api/v1/docs → Swagger UI

Frontend — decided to use D3.js for visualizations
CostPieChart.tsx — interactive pie/donut chart implemented with D3.js
DailyCostChart.tsx — bar chart + trend line
AccuracyCurve.tsx — line chart showing accuracy over time
Why D3:
Complete DOM control, pixel-level precision
Smooth animations (hover enlarges slice, bar color changes)
Custom interactions (tooltip follows cursor, drill-down on click)
Direct SVG manipulation, no React re-render overhead

Dashboard component (frontend/src/components/Dashboard/Dashboard.tsx)
It should:

Top: 4 statistic cards (Experiments / Trials / Runs / Total Cost), data from /dashboard/stats
Two charts:
Left: CostPieChart (from /dashboard/cost-breakdown)
Right: DailyCostChart (from /dashboard/daily-costs?days=30)
Layout: two rows (first row = cards; second = two chart cards)
Validation:
Visit http://localhost:5173/ → should show cards + placeholder charts (replace with D3 later).

Things done today:
Backend:
FastAPI is up, CSV → DataFrame metrics computed.
/api/v1 endpoints ok (health, experiments, trials, dashboard).
DataLoader cleans column names (duration(s) → duration_seconds, experiment name → name).
CI (backend): GitHub Actions + Ruff linting.
Frontend skeleton:
Vite + React + TS + React Router + React Query + Tailwind
Pages: Experiments list (search/sort/pagination), Experiment Detail (stats + trials + AccuracyCurve placeholder)
Sidebar built
Tailwind: customized primary color
Thoughts:
Charts will use D3.js (not Recharts) for precise control and animations.
React handles structure; D3 manipulates SVG via ref + useEffect with data joins.
Backend stays simple for now.
Type-first approach — align backend and frontend fields (e.g. avg_accuracy can be null, avoid mistaking it for 0).
Problems & fixes:
id conflict as both index and column → renamed to avoid merge ambiguity.
avg_accuracy null handling fixed.
Tailwind text-primary-600 missing → extended color palette.
React Query pagination flicker → added keepPreviousData + staleTime.
Why layered like this:
services/api.ts centralizes Axios logic (baseURL, timeout, error).
hooks/ manage data flow (React Query).
components/Visualizations/ keeps chart logic self-contained.
types/ ensures a single source of truth for all fields.

Evening work:
Backend:
Ran FastAPI with CSV data.
Added DataManager to clean column names and compute summaries (total cost, run count, avg accuracy).
Added health check, pagination, dashboard endpoints.
Enabled CORS for frontend at localhost:5173.
Added CI (Ruff + Pytest).
Frontend:
Vite + React + TS + Tailwind setup
Sidebar with Dashboard/Experiments
Experiments list (search/sort/pagination)
Dashboard connected to /dashboard/stats, /cost-breakdown, /daily-costs
Four stat cards + two D3 charts working
Why D3:
I want to learn SVG manipulation directly for full control — e.g. hover enlargement, tooltip following mouse, click-to-drill.
Type consistency between frontend and backend avoids null/0 confusion.

Sun 19 Oct 2025
Plan for today:
Implement AccuracyCurve: D3 line chart in Experiment Detail; clicking a point highlights corresponding trial row.
Chart interactions:
Pie chart click → go to corresponding experiment
Bar chart → add tooltip
Format numbers (currency, time, duration)
Handle empty/error states gracefully
Write frontend README explaining setup, D3 rationale, and folder structure
Checked Hugging Face Dataset Explorer for reference (especially their search logic).
Feature idea: Chatbot — where/how to integrate? Which endpoint?

Issues found:
Run decomposition wrong — fix needed
Trials table needs search/filter toolbar
Once a Trial is clicked → show two tables:
Average token, average cost, average latency of Runs
Run list belonging to the Trial
Each page should have search/sort/filter toolbar for interaction
Backend:
Add Trial Detail Endpoint (/api/v1/trials/{trial_id}) → returns trial + experiment name, handle 404
Frontend:
Add getTrialDetails(id) in api.ts
Create TrialDetail.tsx: top stats + run table (search/sort/filter)
Add route /trials/:id and link Trial IDs in Experiment Detail
Result:
Clicking a trial → navigates to detail page
Top shows avg tokens/cost/latency
Below is run table with search/sort/filter

Mon 20 Oct 2025
To do:
Fix display issue with pending/failed
CSS styling
Chatbot
Better visualization
Check filtering/sorting logic
Fix Runs issue
Currency unit
Auto formatting
Improve index.css: light gray background (Notion/Linear style), faint colored glows (blue, violet, indigo) with grid effect — like a sketch pad look

Current problems:
Pending/failed trials not showing
→ Backend only returned finished trials
Exp-3 and Exp-4 blank page
→ They had only pending/failed, so no runs → merged NaN → JSON serialization failed
Root causes:
NaN in pandas merge → ValueError: Out of range float values are not JSON compliant: nan
Field mismatch: backend used run_count, frontend expected total_runs
Didn't filter out is_deleted = 1
Fixes:

Handle NaN before serialization
Align field names
Filter deleted experiments

Results:
Exp-3/4 no longer crash, pending/failed visible
RUNS column correct
Exp-4 hidden (soft delete)
Empty values show as null
New problem:
Dashboard still shows Exp-4 (soft delete not filtered in stats)
Pie chart shows Exp-4
Clicking Exp-3 keeps loading

Root:
Soft delete not filtered in all endpoints
Fix: added filter in get_experiments(), get_dashboard_stats(), and get_cost_by_experiment()

Handle pandas NaN at multiple levels (before merge and before returning).
JSON doesn't support NaN — must convert.
Keep field names consistent front and back.

Tue 21 Oct 2025

To do:
Chatbot
UI styling improvements
Documentation
Docker
README
Endpoint and feature testing
Filtering logic testing
Chatbot idea:
Frontend: floating button opens chat window (simple input + send).
Displays Q&A (no history).
Context: auto-collect current page data —
Dashboard → summary stats
Experiment Detail → experiment + trials
Currently stateless; each question independent.

Backend:
Single-turn Q&A
Accepts question + context
Keyword filtering to avoid irrelevant queries
Calls API key (DeepInfra), returns response
Implementation:
config.py: added DeepInfra API settings, optional key, enable_chatbot property based on key presence

.env: stores sensitive API key (never commit)
models.py: updated ChatRequest (flexible context dict), ChatResponse (structured return with context tracking)
routes.py: added logic for keyword filtering, context formatting, and LLM call
Frontend:
Floating ChatButton opens ChatModal
useChat.ts handles state + API (React Query)
Added endpoint in api.ts
Styled with Tailwind; button accessible globally from App.tsx
Status:
Chatbot functional — messages sent and replies received correctly.
Currently context is empty {}, so it replies: “I don’t have any experiment data.”
Need to fix by passing actual page context.
Finally got Docker building successfully.