Project Build Guide: Real-Time CSV Chatbot Analytics App

This markdown file outlines the **step-by-step build guide** for your application, based on your frontend-backend split. It is carefully designed to ensure that both you (Frontend Lead) and Person B (Backend Lead) can work **in parallel** without blocking each other, with clear interlocks and interface definitions.

✅Phase 0: Project Bootstrap

Shared Setup

1. Initialize monorepo if not already (Turborepo recommended).
2. Set up directory structure:

```
/frontend  (Next.js 14 - App Router)
/backend   (FastAPI + Celery)
/infra     (Docker Compose, MinIO, Redis, Postgres)
/docs      (Project-wide documentation)
```

3. Set up `.env` files in both frontend and backend directories.
4. Establish shared constants folder (e.g., `shared/types.ts`) if needed for typing across services.
5. Setup CI using GitHub Actions:
6. Lint + test both frontend and backend
7. Auto-deploy main branches

✅You (Frontend)

- Scaffold Next.js project (`app/` based routing)
- Install Tailwind CSS + daisyUI + Zustand + Recharts

✅Person B (Backend)

- Scaffold FastAPI project
- Setup PostgreSQL + Redis + MinIO using Docker Compose
- Setup Celery worker connected to Redis

🔞Phase 1: Auth System

Frontend (You):

1. Implement Google OAuth with `next-auth`

2. Create basic UI components:
3. Hero section
4. Login button (top right)
5. Auth wrapper HOC (to protect routes)

Backend (Person B):

1. Create `POST /auth/register` to sync Google account to Postgres
2. Create `GET /auth/me` endpoint to fetch user info
3. Ensure JWT/session support is consistent

Shared:

- Define user schema and table
- Agree on auth token/session management
- You test login/logout on frontend via `next-auth`

🖤Phase 2: Dashboard + Projects View

Frontend (You):

1. Build dashboard page (`/dashboard`):
2. Grid-based bento box layout
3. Show past projects (cards with project name + timestamp)
4. Add a floating `+` button for "Create New Project"
5. Implement modal to:
6. Input project name
7. Drag-and-drop or upload `.csv`
8. On submission, call backend to:
9. Register project
10. Get presigned upload URL
11. Upload file
12. Trigger processing
13. Build skeleton loading states + error handling

Backend (Person B):

1. Create `POST /projects` to create project entry in Postgres
2. Generate presigned S3/MinIO URL for upload (`/projects/upload-url`)
3. Create Celery task to:
4. Load file with Polars
5. Store in DuckDB
6. Save schema/meta to Postgres
7. Compute embeddings for semantic search
8. Return project status endpoint: `GET /projects/:id/status`

Shared:

- Agree on project schema (id, name, owner_id, created_at, status, etc.)
- Share mock endpoints early so frontend can test

💬Phase 3: Chatbot Page (Main UX)

Frontend (You):

1. After upload completion, route to `/chat/:projectId`
2. Build 3-pane layout:
3. **Left:** Chatbot interface (messages, text input)
4. **Top Right:** CSV preview (headers + 50 rows)
5. **Bottom Right:** Query result pane (charts/tables/text)
6. Hook up message input to call backend for query resolution
7. Display query suggestions

Backend (Person B):

1. Create `POST /chat/:projectId/message` endpoint:
2. Receives user message
3. Uses LangChain agent to decide:
    - SQL on DuckDB **or** semantic search from Pinecone
4. Executes query and returns result (with type flag: chart/text/table)
5. Add `GET /chat/:projectId/preview` to return CSV preview
6. Add query suggestion engine:
7. Use OpenAI embeddings to suggest relevant prompts
8. Endpoint: `GET /chat/:projectId/suggestions`

Shared:

- Define chat message schema: { message, sender, timestamp, result_type }
- Determine result rendering types: `table`, `chart`, `summary`, `error`

🈂️Phase 4: Analytics, Embeddings, and Suggestions

Frontend (You):

1. Add suggestion bar above chat input box (use `/suggestions` endpoint)
2. Add Recharts/Observable Plot to render results (handle multiple types)
3. Improve UI/UX responsiveness, dark mode, transitions

Backend (Person B):

1. Refine vector search with Pinecone/Weaviate
2. Store embeddings per column/project using OpenAI API
3. Monitor prompt latency + DuckDB query duration

Shared:

- Design consistent API response for any query: result type, payload, title
- Add support for retries and timeout handling

♀️ Phase 5: DevOps & Deployment

Infra (Shared):

1. Write Dockerfiles for frontend and backend
2. Build `docker-compose.dev.yml` for local testing
3. Set up Postgres, Redis, MinIO, Pinecone (or Weaviate local)
4. Add production deploys (Render, Railway, Fly.io, GKE)

Monitoring (Shared):

1. Add Sentry to frontend and backend
2. Add PostHog to track:
3. File uploads
4. Chat messages
5. Chart interactions
6. Add alerts for failed Celery jobs or high latency

🧳 Phase 6: Testing & Docs

Frontend (You):

1. Unit test components with Vitest + React Testing Library
2. E2E test login → project → chat using Playwright

Backend (Person B):

1. Pytest tests for:
2. Upload flow
3. File processing
4. Chat LLM handler
5. Integration test for auth + database

Shared:

- `README.md` for dev setup
- `/docs/ARCHITECTURE.md` with diagram
- Usage walkthrough for end-users
- API reference for backend

⬜ Final Notes

- Always stub/mock endpoints with test data before backend is done
- Use feature branches and PRs — tag each other for interlocks
- Prioritize **end-to-end flow testing** over individual microfeatures
- Maintain a `TODO.md` and a `/tests/coverage-report/` folder

Let me know if you want this turned into GitHub Issues or a Kanban board.