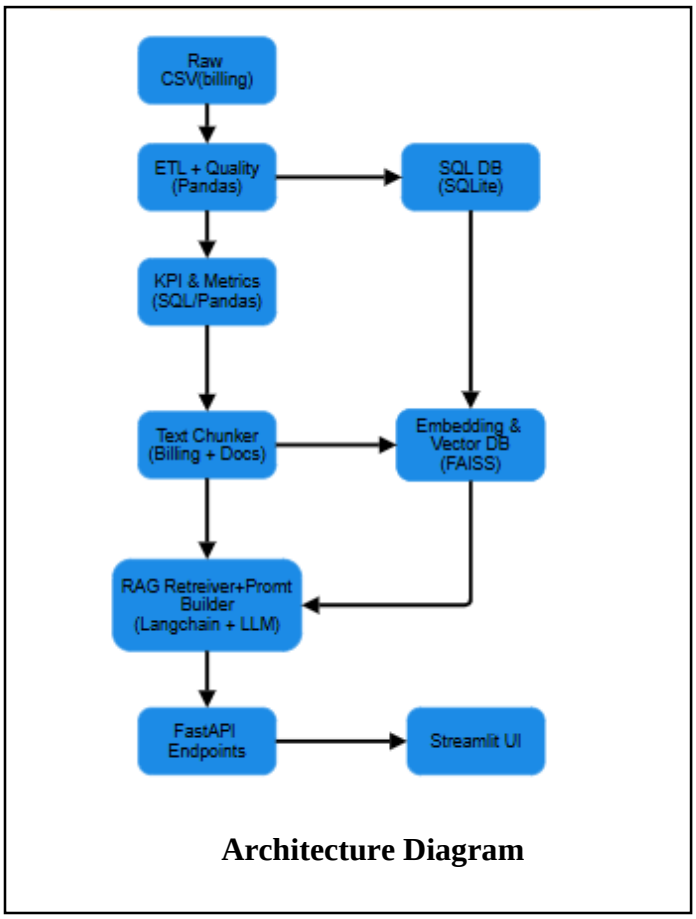
Technical Design Document (TDD)

Architecture Overview:
Modular system with 5 layers:

1. **Data Ingestion & ETL**
   - Pull raw billing and resource metadata CSVs
   - Clean, validate, and load into a SQL database
2. **Data Warehouse / Storage**
   - SQLite for local development
   - PostgreSQL for deployment
3. **Analytics & KPI Engine**
   - Pandas / SQL queries for cost trends, top drivers, anomalies
4. **Vector Store + RAG Pipeline**
   - FAISS or Chroma to index FinOps knowledge and cost text chunks
   - LangChain orchestrates retrieval → prompt → response
   - OpenAI embeddings (or sentence-transformers for local mode)
5. **API + UI Layer**
   - FastAPI to serve REST APIs `/kpi` and `/ask`
   - Streamlit dashboard for interactive KPI + chat interface



Architecture Diagram

Data Model:

- **Billing:**
  invoice_month (text), account_id (text), service (text), resource_id (text), usage_qty (float), unit_cost (float), cost (float)
- **Resources:**
  resource_id (text), owner (text), env (text), tags_json (text)

Trade-offs:

- **SQLite** chosen for simplicity - easy to migrate to Postgres later
- **FAISS** is in-process (fast) but not multi-user - fine for demo
- **LangChain** simplifies RAG but adds abstraction - documented to avoid black-box issues
- **Streamlit** quick to build but not production-grade UI - acceptable for demo

Risks:

- Limited data variety may reduce RAG answer quality → mitigate with synthetic data
- LLM hallucination risk → mitigate with retrieval confidence check + "I don't know" fallback
- Time-boxed build → prioritize working E2E slice over full feature depth

Alternatives Considered:

- Could use DuckDB instead of SQLite for faster local analytics
- Could use LlamaIndex instead of LangChain
- Could use React frontend instead of Streamlit (not needed for MVP)