# **Gen‑AI Natural Language → SQL Banking Query System**

**Author: Core Challengers - EFT** **Date:** Aug 31, 2025  
 **Audience:** Hackathon judges, product owners, engineers

## **1) Executive Summary**

Non‑technical staff (analysts, CSRs, branch ops) frequently need ad‑hoc access to transactional banking data without writing SQL. This solution translates natural‑language questions into safe, executable SQL over a normalized banking schema (SQLite for the hackathon) and returns results with explanations. It includes a human‑in‑the‑loop review step, strong guardrails (schema‑aware generation, validation, read‑only enforcement), and auditable logs.

**Outcomes**

* Democratize data access while keeping data secure.
* Reduce time to insight (seconds vs. hours).
* Provide consistent, explainable SQL with governance.

## **2) Problem Statement & Objectives**

**Problem:** Staff lack SQL expertise, slowing decisions and increasing dependency on data teams.

**Objectives**

1. Understand natural‑language requests about accounts, customers, employees, branches, and transactions.
2. Generate correct, parameterized SQL constrained to the published schema.
3. Validate and safely execute queries (read‑only) against SQLite.
4. Show results with query rationale and allow user edits/approvals.
5. Maintain audit trails (who asked what, generated SQL, execution time, result sample sizes).

**Out of Scope (Hackathon)**: Write operations (INSERT/UPDATE/DELETE), cross‑DB federation, PII masking rules for exports beyond basic redaction toggles.

## **3) High‑Level Architecture**

flowchart   
  


**Key Components**

* **Web UI (React/TS)**: Chat‑like prompt input, SQL preview, results grid, export (CSV), quick filters, saved prompts.
* **API Gateway (Node/Express)**: Single entry; auth, rate‑limit, request normalization.
* **NL→SQL Pipeline**:  
  + *Pre‑processor*: intent classification, timespan parsing (e.g., “Q1 2025”), entity normalization (account types, positions), and synonym expansion.
  + *Schema Catalog*: introspected tables/columns, relationships, constraints, example rows, and allowed vocab.
  + *LLM Engine*: schema‑grounded prompting + few‑shot NL↔SQL examples.
  + *Validator*: static checks (read‑only; no DDL/DML; only whitelisted tables), SQL lint, semantic checks (columns exist, joins consistent), cost guard (row/scan limits).
  + *Executor*: parameterized query execution in SQLite; pagination.
* **Observability**: Structured logs (prompt, SQL, duration, row counts), basic metrics, error traces.
* **Security/RBAC**: Role‑based field visibility; PII redaction toggles for email/phone when role < Analyst.

## **4) Reference Data Model (SQLite)**

Already implemented in your hackathon schema; summarized here for context.

**Tables**

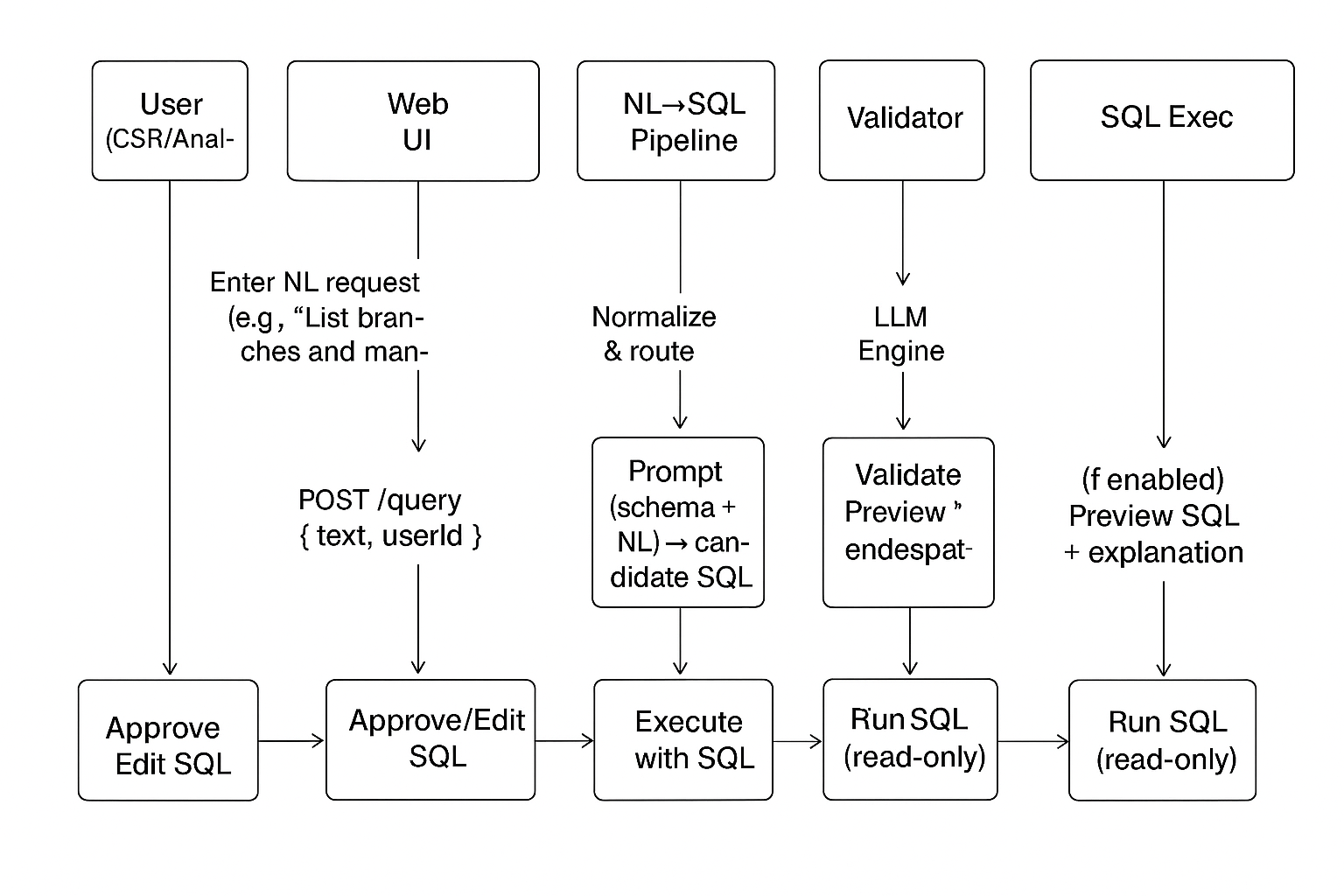
* Branch(branch\_id, branch\_name, manager\_id, address)
* Employee(employee\_id, first\_name, last\_name, branch\_id, position, hire\_date, salary)
* Customer(customer\_id, first\_name, last\_name, dob, gender, address, email, phone)
* Account(account\_id, customer\_id, branch\_id, account\_type, balance)
* Transaction(transaction\_id, account\_id, employee\_id, transaction\_type, amount, status, transaction\_date)

**Relations**

* Branch 1‑\* Employee; Branch 1‑\* Account.
* Customer 1‑\* Account; Account 1‑\* Transaction; Employee 1‑\* Transaction.
* Branch.manager\_id → Employee.employee\_id (nullable).

## **5) Detailed NL→SQL Pipeline**

sequenceDiagram



**Prompting Strategy (LLM)**

* System prompt binds the schema (table DDL snippets, foreign keys), naming conventions, and business semantics (e.g., account types: checking/savings/credit/loan).
* Few‑shot NL→SQL examples tailored to banking tasks (the 35 test cases serve as exemplars).
* Constrained decoding hints: *SELECT only*, limit columns, always qualify tables, use strftime for dates, and prefer JOIN over subqueries for clarity.

**Validation Rules**

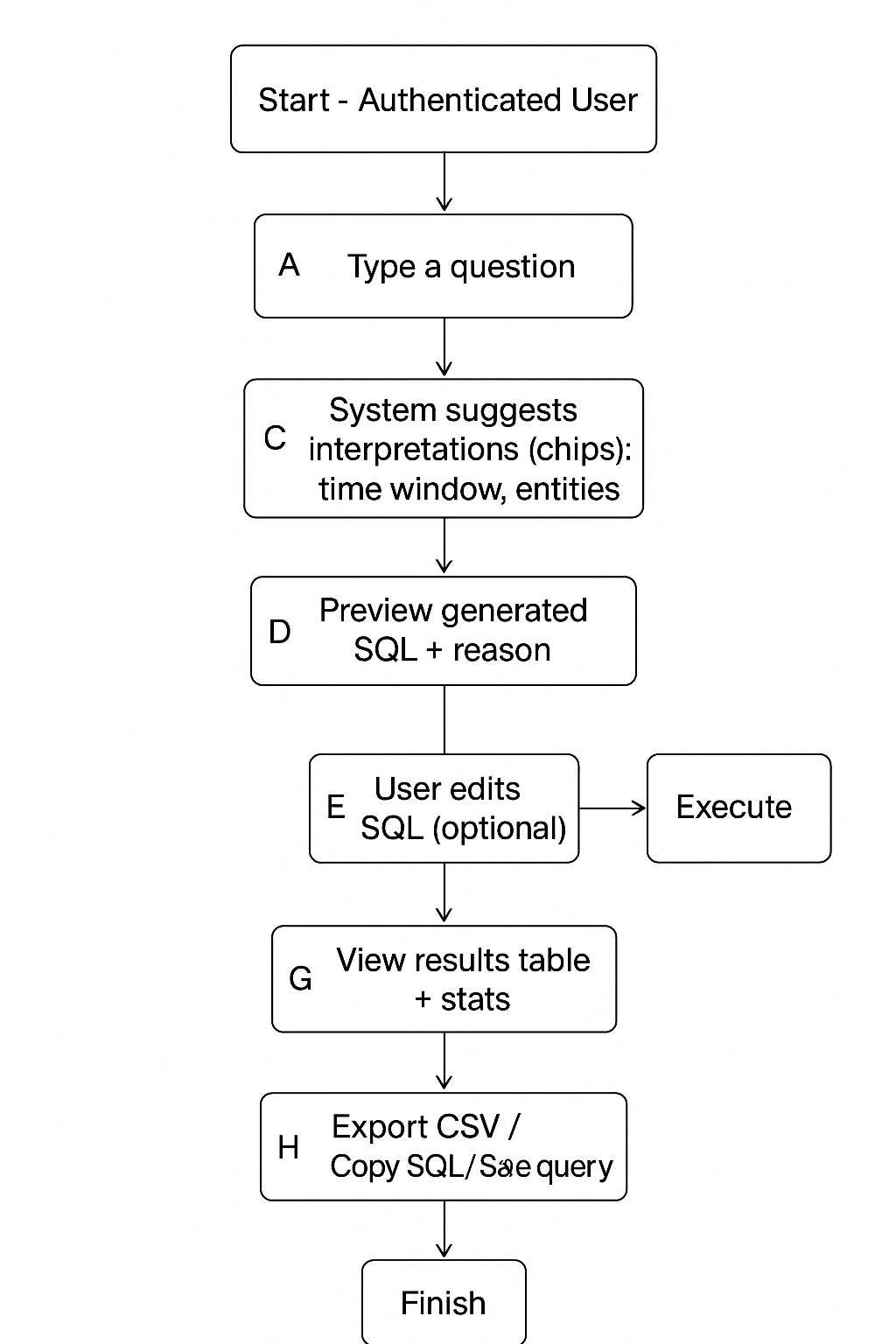
* Deny: INSERT/UPDATE/DELETE/ALTER/DROP; multi‑statements; PRAGMAs.
* Allow only known tables/columns; forbid SELECT \* unless row‑limited.
* Enforce row limit (e.g., default LIMIT 200 with pagination), and execution timeout.
* Cost guard: basic EXPLAIN QUERY PLAN to refuse accidental full scans of huge tables.

**Execution & Result Handling**

* Prepared statements; bind parameters to avoid injection even if editing is allowed.
* Typed serialization (dates as ISO 8601); null safety; totals/aggregates with clear labels.
* Optional *result explanation* summarizing the WHERE/JOIN logic in plain English.

## **6) User Journeys & Flow Diagrams**

### **6.1 Primary Flow — Ask, Review, Run, Share**

flowchart   
  


**UI Highlights**

* Prompt box with quick examples (e.g., “Q1 2025 average transaction amount”).
* Schema side panel: tables, columns, and relationships.
* SQL preview card with *safety badges* (read‑only, limited rows).
* Results grid with sorting, filters, and export.

### **6.2 Admin/Analyst Flow — Audit & Governance**

flowchart TD

A[Admin Login] --> B[Audit Dashboard]

B --> C[Search prompts/SQL by user/time]

C --> D[Review failed validations or slow queries]

D --> E[Adjust guardrails (limits, role access)]

E --> F[Manage saved prompts/examples]

### **6.3 Error/Recovery Flow — Safe Failure**

flowchart TD

A[User submits prompt] --> B[Generation failure/Validation block]

B --> C[Show actionable hints: missing entity, choose timeframe]

C --> D[Offer safe templates]

D --> E[Regenerate]

## **7) Security, Privacy & Compliance**

* **Auth & RBAC**: Role‑based access (CSR, Analyst, Admin). Field‑level protection for PII (mask email, phone) unless Analyst/Admin. Backend enforces; UI only reflects.
* **Read‑only enforcement**: Validator + DB user permissions. SQLite: run in read‑only mode for query execution (open DB with immutable or file permissions in hackathon).
* **Input hardening**: Strict allowlist of tables/columns; auto‑parameterization; remove comments and semicolons from LLM output before validation.
* **Auditability**: Log prompt, SQL, user id, timestamp, row counts, and whether the user edited SQL.
* **Data retention**: Configurable log retention; redact PII in logs.

## **8) Non‑Functional Requirements (NFRs)**

* **Performance**: P50 < 2s for typical queries; generation budget < 1s; result pagination.
* **Reliability**: Graceful degradation—if LLM unavailable, offer canned templates and saved queries.
* **Scalability (future)**: Swap SQLite with Postgres/MySQL; pipeline remains unchanged.
* **Usability**: Clear explanations; deterministic few‑shots; predictable date handling.
* **Observability**: Metrics: gen time, validate failures, execution time, rows returned.

## **9) Tech Stack & Deployment**

* **Frontend**: React + TypeScript + Vite; component library (MUI or Tailwind‑free CSS per your preference); CSV export.
* **Backend**: Python; SQLite client (better‑sqlite3 or knex).
* **LLM Integration**: OpenAI API via a gateway service; retries with backoff; temperature ~0 for determinism.
* **Configuration**: .json schema catalog (table/column names, human labels, sample rows, join paths, canonical values like account types/positions).
* **Deployment (hackathon)**: Single container or local processes; .env for keys; minimal CI script to run unit tests and seed DB.

## **10) Example Prompt Spec (Schema‑Grounded)**

**System Prompt (excerpt)**

* You are an NL→SQL assistant for a banking database. Use only these tables/columns: …
* Generate a single *read‑only* SQLite SQL statement. Prefer explicit JOINs and qualified columns.
* Use strftime('%Y-%m-%d', ...) for date handling; use CASE for weekend detection.
* If ambiguity exists (name collisions), propose the safest interpretation and include a short rationale.

**Few‑Shot Examples** (align with your 35 test cases):

* NL: “List all branches and their managers; include branches without a manager.”  
   SQL: SELECT b.branch\_name, e.first\_name||' '||e.last\_name AS manager FROM Branch b LEFT JOIN Employee e ON e.employee\_id=b.manager\_id;
* NL: “Find customers with checking and savings accounts.”  
   SQL: ... HAVING SUM(account\_type='checking')>0 AND SUM(account\_type='savings')>0.

## **11) Testing Strategy (Mapped to Challenge)**

* **Unit**: Preprocessor (date parsers), validator (deny DDL/DML, table/column checks), SQL generator deterministic tests using the 35 prompts.
* **Integration**: End‑to‑end prompt→SQL→results on the seeded DB; golden‑file comparisons for SQL and row counts.
* **User Acceptance**: Run through all 35 challenge queries from UI; verify explanations and exports.

**Sample E2E Acceptance**

* Prompt: “Show the average transaction amount for transactions completed in the first quarter of 2025.”  
   Expect: SQL uses status='completed' and transaction\_date between 2025-01-01 and 2025-03-31.

## **12) Risks & Mitigations**

* **Hallucinated columns/tables** → Strict schema prompt + validator block + user preview.
* **Sensitive data leakage** → RBAC + masking + read‑only execution + audit.
* **Ambiguous prompts** → UI chips for timeframe/entities; require user confirmation.
* **Slow queries** → Row limits, pagination, EXPLAIN pre‑check.

## **13) Roadmap (Post‑Hackathon)**

* Support synonyms/ontologies (e.g., “Q1”, “first quarter”).
* Add semantic caching (prompt+schema → SQL) and saved dashboards.
* Expand to multiple DB engines; add vector retrieval of schema docs.
* Introduce a fine‑tuned NL→SQL model with your domain examples.

## **14) Appendix — Example Guardrail Pseudocode**

function validate(sql):

deny = [/\b(INSERT|UPDATE|DELETE|ALTER|DROP|CREATE|ATTACH|REINDEX)\b/i, /;\s\*$/]

if any(regex.test(sql) for regex in deny): throw Blocked

if references\_unknown\_tables\_or\_columns(sql, schema): throw Blocked

if not starts\_with\_select(sql): throw Blocked

sql = enforce\_limit(sql, 200)

return sql

**Deliverables for Hackathon Demo**

1. Running web app with chat‑style UI, schema panel, SQL preview, results grid, export.
2. Seeded SQLite DB (from earlier SQL script).
3. Logs/Audit view.
4. Test worksheet showing the 35 prompts and screenshots of results.