Future Considerations & Architectural Evolution

Author: Karan Mehta

Date: 2026-02-25

Table of Contents

Future Considerations & Architectural Evolution

This document outlines the anticipated architectural bottlenecks and proposed system evolutions required to transition this banking service from a local, monolithic MVP to a highly available, globally distributed financial platform.

Roadmap Overview

This roadmap translates the future-state themes below into a phased delivery plan. It is intentionally pragmatic: early phases focus on correctness, auditability, and operability before scaling throughput and geographic footprint.

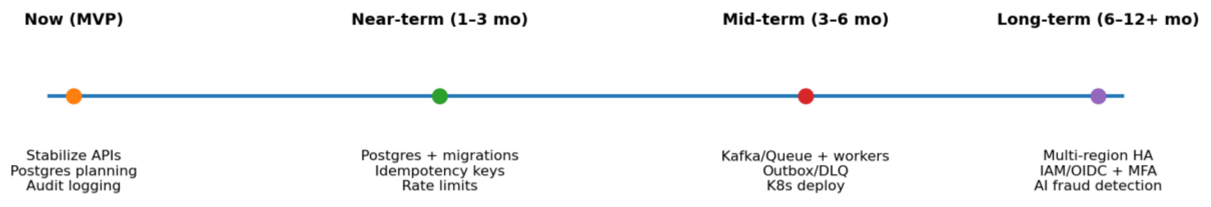

Figure 1 — High-level roadmap timeline (phased evolution).

Target Evolution Diagrams

The following diagrams illustrate what the system looks like as we move from a synchronous MVP to an event-driven, production-grade platform.

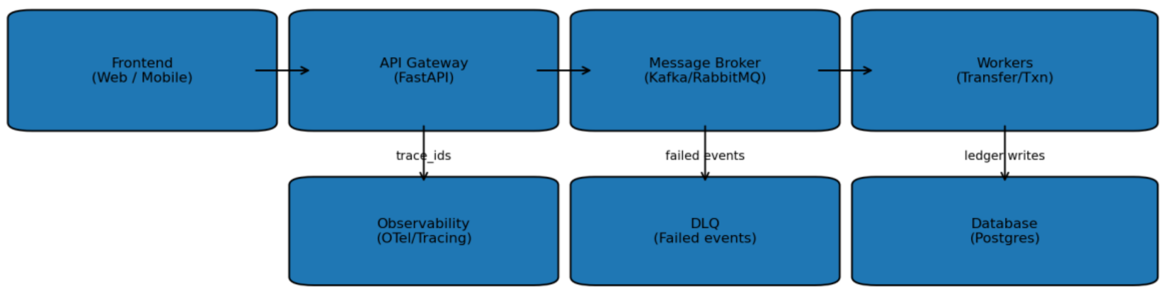

Figure 2 — Event-driven transaction processing (API → broker → workers → DB).

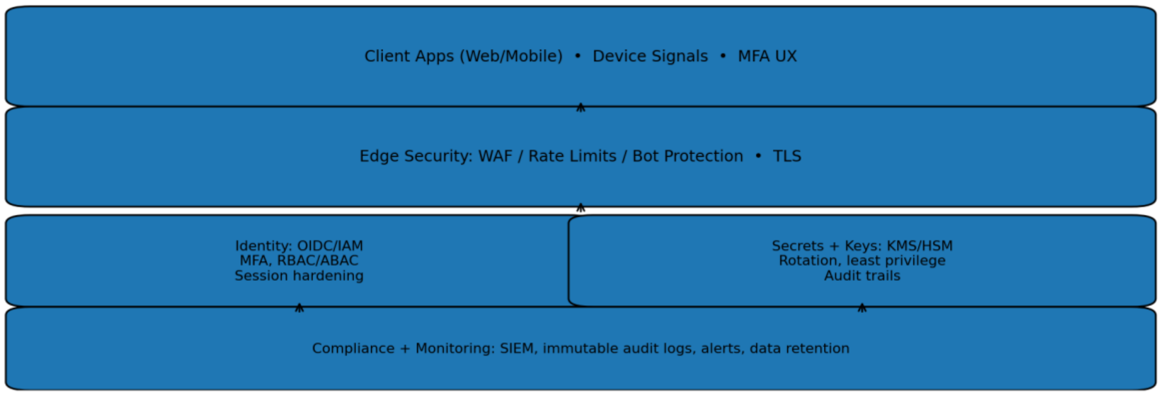
Client Apps (Web/Mobile) • Device Signals • MFA UX
Edge Security: WAF / Rate Limits / Bot Protection • TLS
Identity: OIDC/IAM
MFA, RBAC/ABAC
Session hardening
Secrets + Keys: KMS/HSM
Rotation, least privilege
Audit trails
Compliance + Monitoring: SIEM, immutable audit logs, alerts, data retention

Figure 3 — Security & compliance stack (defense-in-depth).

---

1. Multi-Currency & Cryptocurrency Scaling

Current State: To prevent floating-point precision errors inherent to SQLite and Python `float` types, the current architecture stores all fiat currency (e.g., USD) as `Integer` types representing the smallest fractional unit (cents).

Future Considerations: As the platform scales to support cryptocurrencies (which require significantly higher precision, such as Bitcoin's 8 decimal places or Ethereum's 18 decimal places), the data layer will need to evolve.

- **Storage Pattern:** We will maintain the "smallest indivisible unit" pattern (e.g., Bitcoin stored as Satoshis, Ethereum as Wei).
- **Database Migration:** The database will be migrated from SQLite to a robust relational database like PostgreSQL.
- **Type Casting:** We will transition currency columns to PostgreSQL's arbitrary precision `NUMERIC(precision, scale)` types (e.g., `NUMERIC(36, 18)` for EVM-compatible chains) to guarantee mathematical exactness without overflowing integer limits.

---

2. Event-Driven Architecture & High-Throughput Processing

Current State: Money transfers and account updates are handled synchronously within the FastAPI request lifecycle. While functional for low traffic, this couples the API response time directly to database write speeds and external system latency.

Future Considerations: To handle thousands of transactions per second (TPS), the system will pivot to a cloud-native, event-driven architecture.

- **Message Brokers:** Implementing Apache Kafka or RabbitMQ to decouple the API layer from the transaction processing engine.

- **Asynchronous Workers:** The API will instantly return a "Transfer Initiated" status (HTTP 202), while background worker nodes process the complex ledger updates, balance checks, and anti-fraud verifications from the queue.
- **Dead Letter Queues (DLQ):** Implementing DLQs to safely capture and manually review failed transaction events without dropping financial data.
- **Distributed Tracing:** Implementing OpenTelemetry and Jaeger to inject trace IDs into every request. This ensures that as a transaction moves from the API, into Kafka, and through the worker nodes, we have a complete, visual timeline of the event for rapid debugging.

---

3. AI-Enhanced Security & AI companion in the app

Current State: Transaction validation relies on deterministic, rule-based logic (e.g., sufficient balance checks) handled entirely by standard Python services.

Future Considerations: Integrating advanced AI system design to elevate both platform security and user experience.

- **Real-Time Fraud Detection:** Deploying machine learning models to analyze transaction velocity, location anomalies, and behavioral patterns in real-time, temporarily freezing suspicious transfers.
- **Intelligent Transaction Categorization:** Utilizing LLM APIs to automatically enrich and categorize raw vendor strings into clean user-facing categories (e.g., turning "POS DEBIT 1234 SQ *COFFEE" into "Food & Dining").
- **Conversational AI Chatbot:** Integrating a secure, LLM-powered chatbot directly into the frontend interface. By utilizing a Retrieval-Augmented Generation (RAG) architecture, this assistant will be securely isolated to query only the active user's transaction history, allowing users to ask complex natural language questions (e.g., "How much did I spend on groceries last month compared to this month?") without exposing PII to public models.

---

4. Cloud-Native Deployment & Orchestration

Current State: The application is containerized using Docker and orchestrated locally via `docker-compose.yml` with a local SQLite file.

Future Considerations: Moving to a highly available, fault-tolerant infrastructure.

- **Kubernetes (K8s):** Transitioning deployment to a managed Kubernetes cluster (e.g., EKS or GKE) for automated scaling, self-healing, and zero-downtime rolling updates.
- **Distributed Databases:** Replacing the local database with a globally distributed, highly available database cluster (like Amazon Aurora or CockroachDB) to ensure multi-region redundancy and low-latency read replicas for user statements.

---

5. Advanced Security & Compliance

Current State: The system utilizes standard JWT-based authentication and secure password hashing via bcrypt.

Future Considerations: Banking platforms require military-grade security and strict regulatory compliance.

- **Identity Provider Integration:** Move authentication to an enterprise IdP (OIDC/OAuth2) with MFA, session hardening, and step-up auth for sensitive actions.
- **Key Management (KMS/HSM):** Store and rotate secrets/keys using a managed KMS (or HSM-backed keys) and enforce least-privilege access from services.
- **Encryption Everywhere:** TLS in transit; encryption at rest for databases and backups; per-environment key separation and rotation policies.
- **Audit Trails & Tamper Resistance:** Immutable, append-only audit logs for all security-relevant actions (login, password change, transfers, admin actions) with retention policies.
- **Compliance Foundations:** Prepare controls and evidence for SOC 2; align data handling with privacy requirements (PII minimization, retention, deletion workflows).
- **Security Testing in CI:** SAST, dependency scanning, secret scanning, container image scanning, and periodic DAST for the API surface.
- **Operational Security:** Rate limiting, anomaly detection, alerting, and incident response playbooks tied to on-call and post-incident review.

---

6. Ledger, Accounting, and Reconciliation

Why it matters: As financial complexity grows (fees, reversals, chargebacks, holds), a simple balance update becomes fragile. A double-entry ledger becomes the source of truth.

- Introduce **double-entry ledger tables** (journal entries + postings) and compute balances from ledger aggregates.
- Add **idempotency keys** across all money-moving operations (deposit/withdraw/transfer) to prevent duplicate processing.
- Support **reversals** and **compensating transactions** rather than destructive updates.
- Add **daily reconciliation** jobs and discrepancy reporting (ledger vs. derived balances).

---

7. Reliability, Limits, and User Experience

- User-configurable **transaction limits** (daily/weekly), and soft/hard controls with clear UX messaging.
- **Notifications** (email/push/webhooks) for transfers, low balance, suspicious activity, and account changes.
- **Statements & exports** (monthly statements, CSV export) for transparency and troubleshooting.
- **Scheduled transfers** and recurring payments with clear cancellation and audit history.

---

8. Data, Analytics, and Observability

- Adopt **structured event schemas** (versioned) and an **outbox pattern** to keep DB writes and emitted events consistent.
- Build **dashboards** for operational metrics (TPS, failure rates, latency, queue depth, balances drift) with alert thresholds.
- Support **data warehousing** (CDC into analytics store) for product analytics without impacting OLTP performance.

---

9. Developer Experience and Release Safety

- Feature flags and gradual rollouts for high-risk changes (ledger, async processing).
- Contract tests between frontend and backend; schema versioning for API changes.
- Blue/green or canary deployments; automated rollback signals tied to error rate and latency.