Solution Blueprint

Ultra-Fast Loan Ingestion & Live Ops Dashboard

---

1. Why We Chose What We Chose – Stack & Strategy

Frontend – Next.js + React

- **Why**: Super fast, SEO friendly, and has built-in routing. React 19 gives us awesome performance boosts and less boilerplate.

- **How**: We use the App Router (not old `pages/`) for file-based routing. Server Components for data fetching. Client Components for dynamic charts and live updates.

Backend – Node.js + Express + TypeScript

- **Why**: Simple, scalable, and the same language as frontend (JavaScript). TypeScript keeps our code clean and less buggy.

- **How**: REST APIs handle loan ingestion. We also run a WebSocket server here for pushing live updates to the frontend.

Database – MongoDB (with Mongoose)

- **Why**: Super flexible for our JSON loan data. No need to define rigid schemas like in SQL. Handles thousands of writes easily.

- **How**: Create Mongoose models, optimize with indexes on important fields (like status, date), and use aggregate queries to get dashboard metrics.

Real-time updates – WebSockets

- **Why**: Instant two-way communication. We can push updates to dashboards the moment something happens.

- **How**: Use `ws` package to create a WebSocket server. Add ping-pong heartbeats, retry logic, and fallback to polling if it breaks.

UI Components – Tailwind CSS + shadcn/ui

- **Why**: Modern, beautiful, and super fast to style. Shadcn gives us pre-built, accessible components.

- **How**: Design responsive layout with cards, charts, alerts. Use Tailwind utility classes for clean design.

---

2. Flow Diagrams & Key Processes

Data Flow (End to End)

User submits loan → POST /api/loan/ingest → Validate JSON → Store in MongoDB → Broadcast to all dashboards via WebSocket.

Real-Time Metrics Update

New Loan Added → MongoDB Write → Trigger WebSocket broadcast →
Dashboards auto-update charts & metrics.

Error Handling Flow

Submission →
If validation fails → Show error on frontend (400 Bad Request)
If success → Store in DB →
If DB fails → Retry + Show Error Alert
If WebSocket fails → Fallback to polling

---

3. Challenges We See & How We Plan to Fix Them

Scaling

- **Problem**: 10,000+ requests per hour & 1,000 users online.

- **Fix**: Add rate limiting with Redis, WebSocket message queuing, Mongo connection pooling, scale backend with Load Balancer.

Data Consistency

- **Problem**: Real-time dashboard might lag behind MongoDB.
- **Fix**: Use MongoDB transactions. Also run background cron jobs to reconcile metrics once per day.

Failure Handling

- **WebSocket crashes**: Auto-reconnect with delay. Fallback to polling.
- **Database downtime**: Add retry + error logging. Alert ops team instantly.

---

4. What We Learned, Tried, and Tweaked

Trade-offs We Took

- Chose **WebSockets over Server-Sent Events** → More control, two-way updates, better UX.
- Used **real-time updates + eventual consistency** → UI feels fast, backend syncs up later if needed.

Things That Were Cool

- Real-time charts with 100ms latency.
- WebSocket fallback to polling – dashboard still works even if connection drops.
- Simple REST API + WebSocket combo worked like magic.

---

5. Success Metrics – What Will Make This a Win

Metric	Target
API response time (95% of cases)	under 200 milliseconds
WebSocket push latency	under 50 milliseconds

Dashboard update speed	under 100 milliseconds per update
MongoDB read/write time	under 100 milliseconds

---

6. Business Impact – Why This Is Worth It

Benefit	Before	After	Result
Loan submission time	~2-3 minutes	~30 seconds	75% faster
Error spotting	Manual, slow	Real-time	Instant detection
Dashboard refresh	Manual click	Auto-live	No refresh needed
Ops issue detection	Hours later	Within minutes	90% faster recovery

---

What Success Looks Like

1. Users can submit loans with zero delay.
2. Dashboards reflect changes in under a second.
3. System can handle 10x load without crashing.
4. Clean, bug-free UI that doesn’t feel laggy.
5. Ops can see everything happening in real-time and act fast.
