1. Architecture & Data Model

My design is simple yet highly scalable, ensuring efficient handling of line-of-credit applications and transactions. The architecture consists of the following components:

- **Relational Database**: I used SQLite for simplicity in my solution but any relational database should work well
- **Backend Server**: A Fastify-based backend handles read requests from the frontend and queues state transition requests.
- **Task Queue**: BullMQ with a Redis instance is used to process write operations in a serial manner, preventing race conditions and ensuring data integrity.
- **Worker Service**: Independent worker instances pick up tasks from the queue and execute them asynchronously. These workers can scale vertically to handle increased load.
- **Frontend Application**: A Next.js React-based UI provides user interaction with the system

2. Trade-Offs

- The main tradeoff between this design and a simpler one—where all processing happens in the backend with no queue—is scalability. The simpler approach has benefits like synchronous processing and a single-service architecture, making it easier to manage. However, it struggles under high load since processing requests directly can tie up resources and slow down frontend interactions.
- With third-party payment networks involved, network operations increase, making scalability even more critical. This design offloads heavy processing to independent workers, allowing the backend to focus on database reads, which can be optimized with read replicas. This ensures high database writes don't impact backend performance. Plus, since workers run independently, they can scale vertically to handle a large number of tasks efficiently.

3. Scaling & Performance

- **Concurrency Handling**: BullMQ ensures that state transitions are processed serially, avoiding conflicts.
- **Database Indexing**: Indexing critical fields (e.g., application ID, transaction timestamps) optimizes query performance.
- **Vertical Scaling**: Worker instances can be increased to process more requests in parallel, effectively managing higher loads.
- **AWS Considerations**: In a cloud environment, moving to AWS RDS (PostgreSQL/MySQL) and utilizing ElastiCache for Redis would improve scalability and reliability.

4. Success Criteria

- **Technical Success**: The system must handle concurrent state transitions without race conditions, maintain data integrity, and support scalability.
- **Product Success**: Users must be able to apply for a line of credit, view transaction history, and experience smooth UI interactions with minimal latency.

Frontend pages

- /login - login and signup for users (admin is just a flag in signup)
- /user/application - view applications for a user
- /user/application/:id - application details and actions like cance, request disbursement and request cancellation
- /user/application/create - create an application
-
- /admin - for admin only can view all open applications and reject them