How this wallet system can scale in production

1. **Business logic is isolated in the service**
    - All wallet rules (funding, transfers) live in `WalletsService`.
    - Controllers are thin, so scaling the API layer doesn't affect business logic.
    - This makes it easy to add more endpoints or versions without breaking core behavior.

2. **Database transactions guarantee consistency**
    - Transfers run inside a database transaction.
    - Either both wallets are updated, or nothing is.
    - This prevents partial updates and keeps balances consistent even under load.

3. **Idempotency protects against retries**
    - Each fund or transfer request uses an idempotency key.
    - If a client retries the same request, the system returns the previous result instead of processing again.
    - This is critical for production systems where network retries are common.

4. **Balances are stored as decimals, not floats**
   Wallet balances are stored as strings backed by database decimals.
    - This avoids rounding errors and makes the system safe for financial calculations at scale.

5. **Writes are controlled, reads can scale independently**
    - Wallet updates are strongly consistent and transactional.
    - Read operations (like wallet history) can later be moved to read replicas without changing the code.
    - This allows horizontal scaling as traffic grows.

6. **Wallet history enables auditing and reconciliation**
    - Every balance change creates a history record.
    - This makes it easy to debug issues, reconcile accounts, or run analytics.
    - In production, this is essential for trust and compliance.

1. **Stateless API design**
    - The service does not keep in-memory state.
    - Any instance can handle any request.
    - This makes horizontal scaling straightforward using load balancers.

2. **Easy to extend without redesign**
    - New features like limits, fees, reversals, or multi-currency wallets can be added without changing the core flow.

    - The existing transaction and history structure already supports this.

3. **Clear failure handling**
    - Errors like insufficient balance or invalid wallets are caught early.
    - Failed operations do not leave dirty data behind.
    - This keeps the system stable under high traffic.

4. **Path to production hardening**
    - Row locking can be added to prevent race conditions.
    - Caching can be introduced for performance.
    - The current design does not block any of these improvements.