ADR-001: Database Choice SQL (PostgreSQL) over NoSQL
Status: Accepted

Context (problem & background)

The application is a transactional retail system. Each purchase creates a Sale with one or more SaleItems, records a Payment outcome, and decrements Product stock. Amounts are stored in minor units for accuracy and audit. The system must support reliable reads (SKU/name lookup), reporting (revenue by period, top SKUs, margins), and ongoing schema evolution (discounts, promotions, tax rules). Data integrity and auditability are non-negotiable.

Decision (what we chose)
Agreed to adopt a relational SQL database as the system of record, specifically, PostgreSQL. Enforce referential integrity (FKs), uniqueness (e.g., SKU), and basic checks (non-negative stock/amounts). Execute the entire purchase flow within a single ACID transaction. Manage schema through versioned Prisma migrations.

Consequences (pros/cons, trade-offs)
Pros
• ACID guarantees prevent partial writes and double-selling under concurrency.
• Foreign keys and constraints keep the ledger coherent and auditable.
• SQL enables expressive joins and aggregations for operational analytics.
• Migrations are reproducible and reviewable across environments.

Cons
• Requires migration discipline and database operations hygiene.
• Horizontal write scaling is more complex than some NoSQL options.

Trade-offs
Prioritizes correctness, integrity, and analytics over early horizontal scale. Read replicas and caching can be introduced later as needed.

Alternatives considered
• Document store (e.g., MongoDB): flexible schema, but integrity shifts to application code; eventual consistency risks stale stock and inconsistent ledgers; analytics require extra tooling.
• Graph database: optimized for traversal, not for tabular ledgers and financial constraints.
• Event sourcing + CQRS: strong audit trail and replay, but significantly higher operational complexity; unnecessary for current scope.
• OLAP warehouse as primary store: built for analytics, not OLTP; transactional guarantees and latency are unsuitable.
• Files or in-memory storage: fails durability, concurrency, and transaction requirements

ADR-002: Persistence Style

Prisma ORM behind a Repository Boundary (vs DAO)
Status: Accepted

Context (problem & background)
The system must deliver a full “Register a Sale / Purchase” workflow with atomic writes across Sale, SaleItems, Payment, and stock updates, while remaining type-safe and easy to run locally. Controllers should not leak database concerns; business rules (tax/fees, validation, error mapping) should live in services. The design should allow swapping data access strategies in future checkpoints without rewriting business logic.

Decision (what you chose)
Use Prisma ORM for database access and migrations. Keep a clear persistence boundary: services depend on repository interfaces; Prisma lives in adapter implementations. Use prisma.$transaction to execute the purchase flow atomically. Keep pricing/totals as pure functions for straightforward unit testing.

Consequences (pros/cons, trade-offs)
Pros
• End-to-end type safety from schema to code via the generated client.
• Concise data access with straightforward nested writes and transactions.
• First-class, versioned migrations integrated into the workflow.
• Faster delivery and simpler onboarding than handwritten DAOs.
• Clean layering: controllers → services → repositories → adapters.

Cons
• Some SQL control is abstracted; hotspots may need targeted raw queries.
• Contributors must learn Prisma’s schema/migrate workflow.

Trade-offs
Accept a thin ORM layer to gain speed and safety now. Preserve portability by keeping repositories as the seam for future DAO or ORM changes.

Alternatives considered
• DAO with raw SQL (pg/sqlite3): maximum control and transparency; more boilerplate for mapping and transactions; slower iteration; migrations must be hand-authored.
• Query builder (Knex): reduces SQL verbosity but lacks Prisma’s generated types and schema-first workflow.
• Other ORMs (TypeORM, Sequelize, Mikro-ORM): viable choices; Prisma preferred for schema-first design, generated types, nested writes, and tooling quality.
• Stored procedures/DB logic: good for hot paths, increases coupling to a specific dialect, complicates tests/CI.