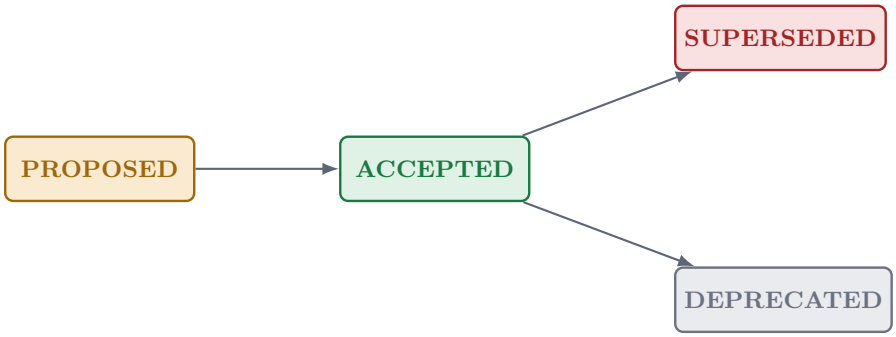
Architecture Decision Records

Nimbus Order Platform

Platform Engineering • Maintained by the Architecture Guild • Last reviewed 2025-06-02

Architecture Decision Records (ADRs) capture the significant, hard-to-reverse technical choices made while building the Nimbus order and fulfillment platform. Each record is immutable once accepted: if circumstances change, we write a new ADR that supersedes the old one rather than editing history. This register is the canonical index — consult it before proposing an approach that a past decision already ruled on.



Decision Register

ID	Title	Status	Date
ADR-0001	Record architecture decisions as lightweight ADRs	ACCEPTED	2024-11-04
ADR-0002	PostgreSQL as system of record for the order ledger	ACCEPTED	2024-11-18
ADR-0003	Kafka for asynchronous order-fulfillment events	ACCEPTED	2025-01-09
ADR-0004	Adopt gRPC for internal service-to-service calls	PROPOSED	2025-05-27
ADR-0005	Retire the legacy SOAP integration layer	SUPERSEDED	2024-09-12

ADR-0001 — Record Architecture Decisions as Lightweight ADRs

Status	ACCEPTED	**Date**	2024-11-04
Deciders	M. Alvarez, R. Okonkwo, S. Bergstrom	**Tags**	process, governance
Supersedes	—	**Superseded by**	—

Context

The order-fulfillment rewrite has grown to nine services and four data stores maintained by three squads. Decisions about data ownership, messaging, and protocol choices were previously discussed in chat threads and design docs that were hard to find six months later. New engineers repeatedly re-opened debates ("why Postgres and not DynamoDB?") that had already been settled, and two squads independently built incompatible retry strategies because neither knew the other's reasoning. We need a durable, low-overhead record of significant technical decisions and the context behind them.

Decision

We will record every architecturally significant decision as a short Markdown file under `/docs/adr/`, numbered sequentially (`NNNN-title.md`), following this five-part shape: *Status*, *Context*, *Decision*, *Consequences*, *Alternatives Considered*. Records are immutable once merged to `main`. A decision is superseded, never edited: a new ADR is written and the old one's status changes to **Superseded by ADR-NNNN**. Any change to a service boundary, data-ownership model, communication protocol, or a dependency that is expensive to reverse requires an ADR before implementation begins.

Consequences

Positive

- New hires read the register instead of re-litigating settled debates.
- Reviewing an ADR PR is a five-minute task, so the process survives contact with deadlines.
- Postmortems can cite the exact reasoning that led to an incident-adjacent choice.

Negative

- Adds a review step before implementation on decisions that are genuinely urgent.
- Requires discipline to keep the register current as squads reorganize.

Alternatives Considered

Option	Pros	Cons	Verdict
Confluence wiki pages	Familiar tooling, rich formatting	Drifts from code, no PR review, hard to search	Rejected
No formal record	Zero process overhead	Already causing repeated debates and drift	Rejected
Full RFC process	Deep stakeholder sign-off	Too heavyweight for day-to-day decisions	Rejected
Markdown ADRs in-repo	Versioned with code, reviewed via PR	Requires discipline to keep terse	**Chosen**

ADR-0002 — PostgreSQL as System of Record for the Order Ledger

Status	ACCEPTED	**Date**	2024-11-18
Deciders	R. Okonkwo, S. Bergstrom, T. Lindqvist	**Tags**	data, ledger, postgres
Supersedes	—	**Superseded by**	—

Context

The order ledger records every state transition of an order — placed, paid, packed, shipped, refunded — and is the source of truth for finance reconciliation and dispute handling. It requires strict transactional guarantees: two writers must never both mark the same order "paid," and a partial write (payment captured, ledger not updated) is unacceptable. Write volume is moderate (peak ~900 writes/sec during flash sales) but correctness matters more than raw throughput. The team evaluated DynamoDB, already used at Nimbus for the session store, against PostgreSQL, already run in production for the catalog service.

Decision

We will use PostgreSQL 15 (Vertex RDS, Multi-AZ) as the system of record for the order ledger. Each order-state transition is written inside a single ACID transaction alongside its audit-log row, using `SELECT ... FOR UPDATE` to serialize concurrent transitions on the same order. Read replicas absorb the reporting and support-tooling read paths so writes are never contended by analytics queries.

Consequences

Positive

- Multi-row ACID transactions make the "paid + ledger" invariant trivial to enforce.
- Existing on-call runbooks and backup tooling already cover Postgres.
- Read replicas isolate analytics load from the write path at negligible cost.

Negative

- Vertical scaling ceiling is lower than DynamoDB; revisit if writes exceed ~5k/sec.
- Multi-AZ failover adds ~30s of write unavailability during an RDS event.

Alternatives Considered

Option	Pros	Cons	Verdict
DynamoDB	Near-infinite write scaling, low ops overhead	Multi-item transactions are limited (25 items, single region); weaker fit for ledger invariants	Rejected
Aurora PostgreSQL	Faster failover (~10s), storage auto-scaling	Higher cost, another platform to certify at short notice	Deferred
Event-sourced store (custom)	Full history is a first-class model	Significant build cost; duplicates what a WAL-backed RDBMS gives for free	Rejected
RDS PostgreSQL 15	Strong consistency, team already runs it in prod	Scaling ceiling lower than DynamoDB	**Chosen**

ADR-0003 — Kafka for Asynchronous Order-Fulfillment Events

Status	ACCEPTED	**Date**	2025-01-09
Deciders	M. Alvarez, T. Lindqvist, P. Nakamura	**Tags**	messaging, kafka, fulfillment
Supersedes	—	**Superseded by**	—

Context

Once an order is marked "paid" in the ledger (ADR-0002), five downstream services must react: inventory reservation, warehouse routing, invoicing, fraud re-scoring, and customer notification. Today the order service calls each synchronously over HTTP, so placement blocks on the slowest of the five, and a single unavailable service (fraud re-scoring, most often) degrades checkout latency for every order, not just the ones it flags. We need to decouple "paid" from "every downstream system has processed it," while keeping strict per-order ordering.

Decision

We will publish order lifecycle events (`order.paid`, `order.cancelled`, `order.refunded`, `order.shipped`) to an Apache Kafka topic (`MSK`, 12 partitions, keyed by `order_id`) immediately after the ledger transaction commits, using the transactional outbox pattern to guarantee the event publishes if and only if the ledger write succeeded. Downstream services consume independently with their own consumer groups and offsets, so a slow or down consumer no longer blocks checkout or any other consumer.

Consequences

Positive

- Checkout latency no longer depends on the slowest downstream consumer.
- A down consumer catches up from its committed offset with zero data loss.
- New consumers (a future loyalty-points service) can subscribe without touching the order service.

Negative

- Introduces eventual consistency; support tooling must show "processing" states honestly.
- Outbox pattern adds a polling publisher process to operate and monitor.
- Squads must learn consumer-group rebalancing and offset-management failure modes.

Alternatives Considered

Option	Pros	Cons	Verdict
Synchronous HTTP (status quo)	Simple mental model, immediate feedback	Couples checkout latency to five downstream services	Rejected
SQS standard queues	Fully managed, minimal ops	No per-order ordering guarantee across consumers	Rejected
SQS FIFO queues	Ordering per message group	Throughput ceiling too low for flash-sale peaks	Rejected
Apache Kafka (MSK)	Ordered per partition, high throughput, replayable	Team must operate consumer-group tooling	**Chosen**

ADR-0004 — Adopt gRPC for Internal Service-to-Service Calls

Status	PROPOSED	**Date**	2025-05-27
Deciders	S. Bergstrom, P. Nakamura	**Tags**	networking, grpc, latency
Supersedes	—	**Superseded by**	—

Context

Inventory reservation calls warehouse routing synchronously on the hot path, and profiling from the April load test shows JSON serialization and HTTP/1.1 connection churn account for ~22ms of the ~140ms call, second only to the database round trip. Nine services now make 40+ internal REST calls per order, and each pair invented its own retry and error-envelope conventions — flagged twice by on-call as a source of confusion during incidents.

Decision

We propose adopting gRPC over HTTP/2 for internal, synchronous service-to-service calls, starting with the inventory-to-warehouse-routing path. Contracts are defined in Protocol Buffers and published from a shared `proto/` repository with generated stubs per language. Timeouts, retries, and error codes follow one shared interceptor library instead of being reinvented per service pair. Public-facing APIs are explicitly out of scope and remain REST/JSON.

Consequences

Positive

- Binary protobuf plus HTTP/2 multiplexing removed ~18ms from the same call in a spike.
- Shared `.proto` contracts catch breaking schema changes at build time, not in production.
- One retry/timeout/error-code convention replaces nine bespoke ones.

Negative

- Debugging with `curl` disappears; engineers need `grpcurl` and protobuf tooling.
- Load balancers and the service mesh need HTTP/2-aware configuration.
- Migration is per-endpoint and will run for several quarters alongside REST.

Alternatives Considered

Option	Pros	Cons	Verdict
Keep REST/JSON	No migration cost, universally debuggable	Serialization and connection overhead stays on the hot path	Rejected
GraphQL federation	Flexible querying for aggregators	Solves a read-shaping problem we do not have	Rejected
gRPC over HTTP/2	Lower latency, typed contracts, shared tooling	Steeper learning curve, mesh reconfiguration	**Proposed**

Open question: pilot on the inventory → warehouse-routing path for one release cycle before deciding whether to migrate the remaining eight call sites. Review scheduled for 2025-08-15.

ADR-0005 — Retire the Legacy SOAP Integration Layer

Status	SUPERSEDED	**Date**	2024-09-12
Deciders	R. Okonkwo, M. Alvarez	**Tags**	integration, legacy, soap
Supersedes	—	**Superseded by**	ADR-0003

Context

The original Vertex-era order pipeline integrated with the warehouse-management system via a SOAP/XML interface built in 2019. It requires a stateful adapter process, XML schema validation adds ~40ms per call, and the WSDL has not been updated by the warehouse vendor since 2021, so every schema drift on their end has been patched with brittle XSLT transforms on ours. Two production incidents in Q2 2024 traced back to the adapter silently dropping malformed messages instead of raising an alert.

Decision

We decided to decommission the SOAP adapter and replace warehouse notifications with an event published to the (then newly proposed) order-events stream, to be consumed directly by the warehouse-routing service. This decision was folded into and formally superseded by ADR-0003, which generalized the same event-driven approach to all five downstream consumers rather than special-casing warehouse routing alone. The SOAP adapter was fully decommissioned on 2025-02-03 once ADR-0003 shipped to production.

Consequences

Positive

- Removed ~40ms of XML validation latency and a full adapter process to operate.
- Malformed-message failures now raise a consumer-lag alert instead of failing silently.

Negative

- Cutover required a two-week dual-write period to validate parity before decommissioning.