

ENGINEERING PLAYBOOK

The Agentic SDLC Playbook

How modern engineering teams ship faster, smarter, and more reliably using AI agents — with a real-world ecommerce reference architecture.

Headless

API-First

Cloud-Native

Kubernetes

Reactive μ s

CQRS

DDD

Events

For Engineers, Architects & Technology Leaders



What This Playbook Covers

Six sections — from the mental model to the machine commands engineers type every day.

01

The Mental Model

Roles, plans, and why structure beats prompting

02

The Architecture

Headless ecommerce — domains, services, tech stack

03

The Plan Document

The contract between humans and AI agents

04

The Agent Team

Architect, developer, QA, reviewer — defined roles

05

The /ship Workflow

End-to-end orchestration: ticket → merged PR

06

Patterns & Practice

CQRS, DDD, events, reactive — how agents handle them

SECTION 01

The Mental Model

Why AI agents need roles, not just prompts — and how structured planning multiplies every downstream output.



Give AI a Role, Not Just a Prompt

Each agent.md file encodes a professional persona — its responsibilities, constraints, and perspective.



Architect Agent

`architect.md`

- Resolves architectural Q&A
- Approves plan before coding
- Defines tech constraints
- Defers out-of-scope items



Developer Agent

`developer.md`

- Implements against the plan
- Follows codebase conventions
- Writes unit tests
- Does not freelance on design



QA Agent

`qa-tester.md`

- Runs tests, exercises edges
- Produces evidence report
- Validates against acceptance criteria
- Raises defects before review



Reviewer Agent

`senior-reviewer.md`

- Reviews diff vs plan
- Checks for scope drift
- Enforces code standards
- Approves or requests changes

Each agent reads the Plan Document first — the shared source of truth for every decision in the story.

SECTION 02

The Ecommerce Reference Architecture

A headless, API-first, cloud-native ecommerce platform — the running example throughout this playbook.



Bounded Contexts — The DDD Domain Map

Each bounded context maps to one or more microservices. Agents respect these boundaries during implementation.



Catalogue

Product, Category
Search & Discovery



Cart

Session, Line Items
Promotion Rules



Orders

Order Lifecycle
CQRS + Event Sourcing



Inventory

Stock Levels
Warehouse Allocation



Fulfilment

Picking, Packing
Shipment Tracking



Payments

PSP Gateway
Refunds & Chargebacks



Identity

Auth, Profiles
OAuth2 / OIDC



Notifications

Email, SMS, Push
Event-Driven Alerts



Apache Kafka Event Bus — Domain events flow between all bounded contexts asynchronously

The Technology Stack

Frontend / Channels

Next.js (Headless Storefront)

GraphQL BFF

REST Mobile API

Partner B2B API

API Gateway / Edge

Kong / AWS API Gateway

OAuth2 + JWT

Rate limiting

Edge caching (CDN)

Reactive Microservices

Quarkus (Java 21) + Mutiny

Non-blocking I/O (Vert.x)

SmallRye Reactive
Messaging

OpenAPI + AsyncAPI
contracts

Data Layer

PostgreSQL (Commands /
Write)

MongoDB (Query / Read
projections)

Redis (Cart sessions, rate
limits)

Event Store (Kafka + Schema
Registry)

Platform / Infra

Docker + Kubernetes (GKE)

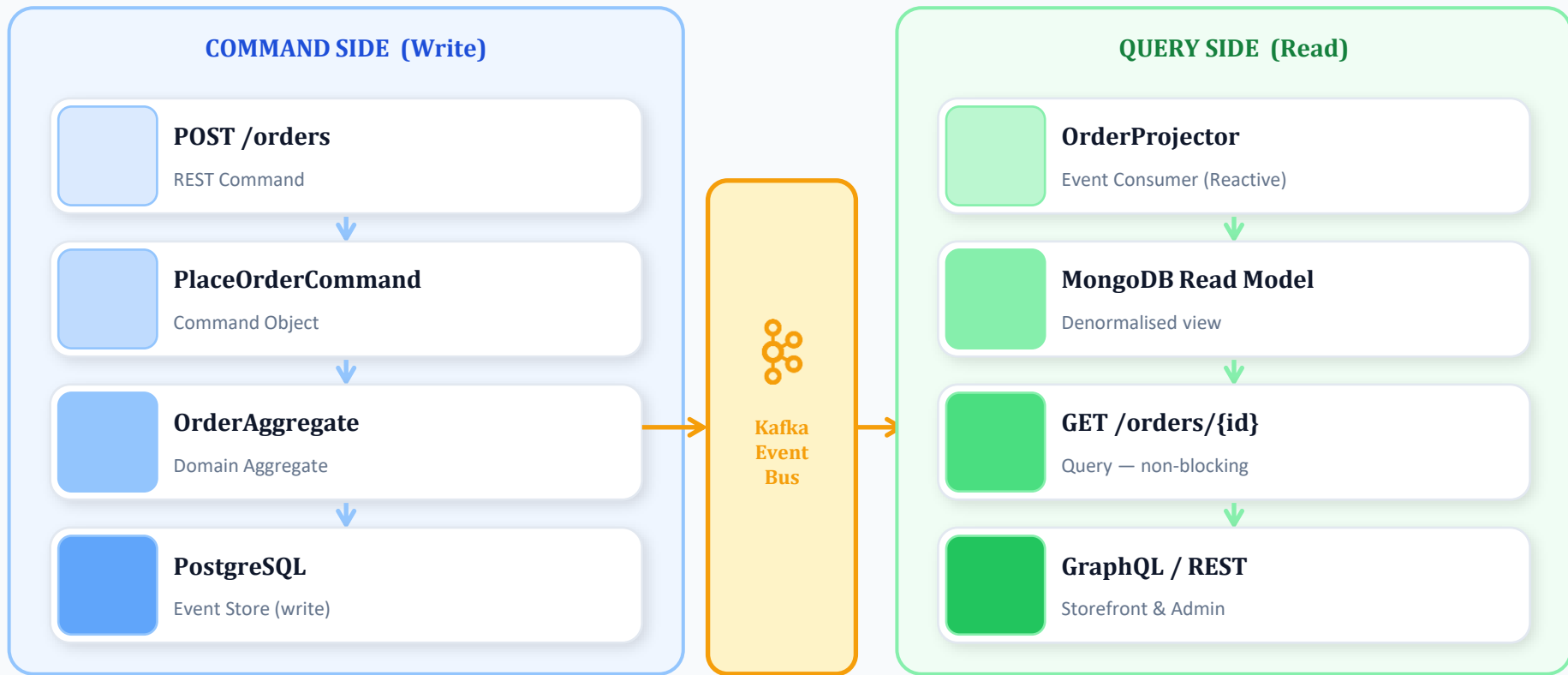
Helm charts, Kustomize

Istio service mesh

GCP Cloud Run / Artifact
Registry

CQRS + Event Sourcing — Orders Domain

Commands mutate state → events are published → read models are projected asynchronously. Non-blocking throughout.



The Developer Agent is pre-instructed on this pattern — it knows commands go to PostgreSQL and read projections go to MongoDB.

SECTION 03

The Plan Document

The single most important artefact in the agentic SDLC. The contract between human judgment and AI execution.



Anatomy of a Plan Document

1 Header — Status & Ownership

Status: APPROVED

Approved by: Lead Architect | Date: 2026-06-10

Service: order-service | Jira: ORDERS-412

Epic: Order Management Platform

2 Context — Why This Story Exists

When a customer places an order, the write path (PostgreSQL) succeeds but the read projection (MongoDB) is stale for up to 30s because the Kafka consumer is synchronous and blocks under load. This story makes the OrderProjector fully reactive using Mutiny's Uni/Multi pipeline.

3 Architectural Q&A — Decisions Made Up-Front

Q1: Use Mutiny or CompletionStage? → Mutiny (Uni<T>). Already in BOM.

Q2: Retry strategy for projection failures? → Exponential backoff, max 5 retries, DLQ after.

Q3: Idempotency key on projector? → YES — use orderId + eventVersion composite.

Q4: Add Micrometer counter for DLQ events? → YES, tagged by event type.

4 Explicit Scope — In vs Deferred

IN SCOPE: Reactive OrderProjector, retry + DLQ, Micrometer counter.

DEFERRED: OpenTelemetry trace propagation (ORDERS-419).

DEFERRED: Batch back-fill of existing events (ORDERS-421).

OUT OF SCOPE: Any changes to the command side.

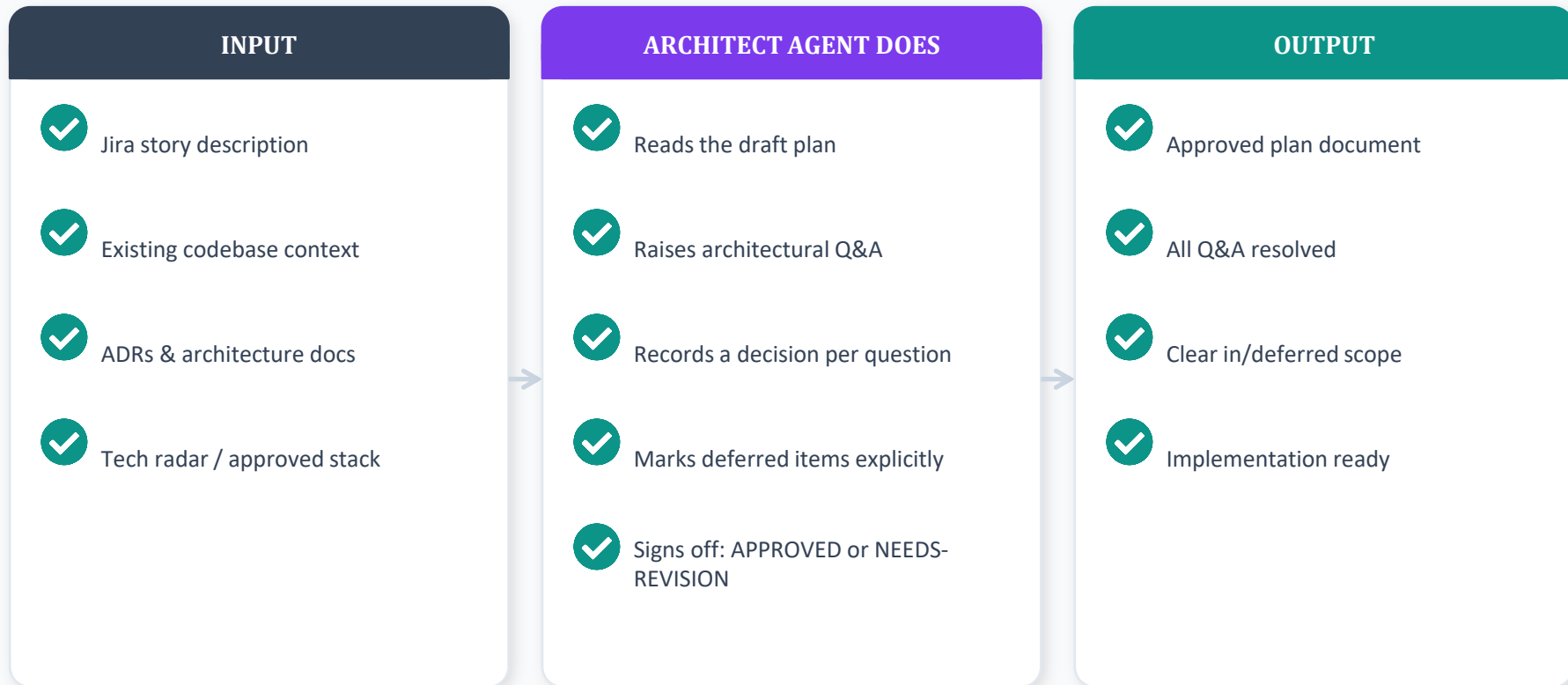
SECTION 04

The Agent Team in Action

Worked example: making the OrderProjector reactive. Follow one story through the complete agent pipeline.



Step 1 — Architect Agent: Resolve Before You Code



Example: Q — 'Should we use Mutiny or CompletionStage?' → A — 'Mutiny (Uni<T>), already in BOM. Rationale: consistent non-blocking API across service.'

Step 2 — Developer Agent: Implement Against the Plan

The developer agent reads the approved plan and produces reactive, non-blocking Quarkus code — no ad-hoc decisions.

OrderProjector.java

```
@ApplicationScoped
public class OrderProjector {

    @Inject OrderReadRepository repo; // MongoDB (panache reactive)
    @Inject MeterRegistry metrics;

    @Incoming("order-events")
    public Uni<Void> project(OrderEvent event) {
        return repo.findById(event.orderId())
            .onItem().ifNull().switchTo(
                () -> Uni.createFrom().item(new OrderReadModel()))
            .chain(model -> {
                model.apply(event); // pure projection
                return repo.persistOrUpdate(model);
            })
            .onFailure().retry()
                .withBackOff(Duration.ofSeconds(1))
                .atMost(5)
            .onFailure().invoke(ex ->
                metrics.counter("orders.dlq.events",
                    "type", event.type()).increment())
            .replaceWithVoid();
    }
}
```

Non-blocking

Uni<Void> pipeline. Every call is async — no thread blocking.

Plan-compliant

Retry + DLQ exactly as Q&A decided: 5 retries, exponential backoff.

Idempotent

findOrCreate pattern ensures re-processing the same event is safe.

Observable

Micrometer counter on DLQ — matches the plan's observability requirement.

Steps 3 & 4 — QA Agent + Reviewer Agent



QA AGENT — qa-tester.md

Unit tests

Verifies reactive pipeline: projection applied, retry triggered, DLQ counter incremented

Integration tests

Kafka → projector → MongoDB round-trip. Tests idempotency: sends duplicate events.

Load test

Confirms non-blocking: 1,000 concurrent events processed without thread exhaustion.

Evidence report

All tests PASS. DLQ counter fires at attempt 6. Backoff intervals verified in logs.



REVIEWER AGENT — senior-reviewer.md

Plan vs diff

Checks every decision in the plan is reflected in the code. Flags any undocumented change.

Scope control

No command-side changes present. Deferred items (OpenTelemetry) not touched. ✓

Code standards

Reactive chain structured correctly. No `.await().indefinitely()` calls present. ✓

PR description

Auto-generated from plan: context, decisions, test evidence, deployment notes all included.

SECTION 05

The `/ship` Workflow

One command. Ticket to merged PR. Full audit trail by default.

```
$ /ship ORDERS-412
```



The /ship Pipeline — End to End

Each stage gates the next. No stage begins until the previous produces verified output.



Total elapsed: ~19 minutes from /ship command to reviewed, evidence-backed, merged PR — with full decision audit trail.

The sequence, gates, and handoffs are encoded in skills/ship/SKILL.md — versioned alongside the codebase.

SECTION 06

Patterns & Practice

How agents handle the hardest parts: DDD, events, reactive, CQRS, SQL vs NoSQL, and containerisation decisions.



Six Patterns Every Agent Must Know

Encoded in developer.md — agents implement these patterns automatically when the plan calls for them.



Domain-Driven Design

Agents never cross bounded context boundaries. Each service owns its aggregate. No direct DB calls across domains — only events.

eg: Payments service emits PaymentAuthorised event. Orders service reacts — it never reads the payments DB.



CQRS

Commands → PostgreSQL (write). Queries → MongoDB read projections. Agent enforces this split in every implementation.

eg: PlaceOrder command writes to pg.orders. GET /orders/:id reads from mongo.order_views. Zero overlap.



Event-Driven (Kafka)

Domain events are first-class. Agent names events in past tense (OrderPlaced, StockReserved). Consumers are always idempotent.

eg: OrderPlaced → InventoryService reserves stock → FulfilmentService schedules pick → NotifService emails customer.



Reactive / Non-blocking

No blocking I/O. All DB and HTTP calls return Uni<T> or Multi<T>. No .await().indefinitely() outside test code.

eg: OrderProjector uses Mutiny pipeline: chain → retry → DLQ counter. Thread pool never exhausted under load.



SQL vs NoSQL

PostgreSQL for transactional writes where ACID matters. MongoDB for read projections — schema-flexible, fast for nested docs.

eg: Order write aggregate in PG (normalised, FK-constrained). Order detail view in Mongo (denormalised, single-doc fetch).



Container & K8s

One service per Docker image. Helm chart per service. Agents produce Dockerfile + helm/values.yaml in each story.

eg: order-service: JVM image (GraalVM native optional). HPA: minReplicas:2, CPU 70%. Liveness: /q/health/live.

Getting Started — 90-Day Roadmap

Days 1–14

One Agent, One Story

- Pick the role where AI already helps most (developer or reviewer)
- Write a proper role definition in agent.md
- Run it on one real story. Review its output critically.
- Measure: did the output match your standards?



Days 15–45

Plan Document Discipline

- Before every story, write the architectural Q&A
- Force all decisions to be resolved before coding begins
- Review 3 plans with your team. Improve the template.
- Introduce the four-agent team on two stories.



Days 46–90

Wire the /ship Workflow

- Build the skills/ship/SKILL.md orchestration file
- Start with a 3-step sequence (architect → developer → reviewer)
- Add QA agent. Add evidence gates between stages.
- Measure cycle time. Iterate on agent definitions like code.

Iteration is the mechanism. Agents are engineering artefacts — version them, review them, improve them continuously.

The system is the advantage.

AI raises the floor. Your engineers raise the ceiling.
The team that masters this pattern doesn't just move faster — it ships with more consistency, more auditability, and more confidence.

- Give AI a role, not just a prompt
- Resolve architecture before code, not during
- The plan document is the shared brain
- Quality is structural, not dependent on who's available

