Order Platform – Architecture, Refactor & Defense

1. Assessment A – Build + Defend

1.1 Architecture Overview / Scenario

Design an **Order + Payments + Fulfilment platform** with the following services: - Orders Service -

Payments Service - Fulfilment Service - Catalog Read Service - API Gateway

The system must support **10k orders/min burst**, provide **exactly-once effects (as close as feasible)**, and

ensure **auditable workflows**

This section describes the **target production architecture**, focusing on **service boundaries, data ownership, interaction patterns, and scalability**. The design explicitly avoids shared databases and enforces clear responsibility per service.

1.2 Service Boundaries & Ownership

Orders Service (Core Orchestrator) - Owns the Order aggregate and lifecycle - Responsible for order creation, idempotency, and workflow initiation - Emits `OrderCreated` domain events - Owns `OrdersDb`

Payments Service - Owns payment state and financial consistency - Handles retries and deduplication of payments - Emits `PaymentCompleted` events - Owns `PaymentsDb`

Fulfilment Service - Owns shipment / fulfilment lifecycle - Reacts to payment completion - Owns `FulfilmentDb`

Catalog Read Service - Read-only service optimized for UI queries - No transactional writes in the critical path - Owns `CatalogDb` (MongoDB)

API Gateway - Single external entry point - Enforces authZ, rate limiting, and correlation IDs - Aggregates data across services

---

1.3 Logical Architecture Diagram (Textual Description)

```
Client
  │
  ▼
API Gateway
  │
  ├──► Orders Service ──► OrdersDb (SQL)
  │        │
  │        └──► Outbox → Payments Service
  │                     │
  │                     └──► PaymentsDb (SQL)
  │                             │
  │                             └──► Outbox → Fulfilment Service
  │                                           │
  │                                           └──► FulfilmentDb (SQL)
  │
  └──► Catalog Read Service ──► CatalogDb (MongoDB)
```

Events flow **asynchronously**; no service calls another service's database.

1.4 CQRS Artifacts

Commands - CreateOrder - RecordPayment - CreateFulfilment

Queries - GetOrderById - GetOrderSummary - GetCatalogItems

Writes are handled via EF Core. Reads are served via Dapper or MongoDB.

1.5 Consistency & Exactly-Once Effects

- Idempotency keys on write endpoints
- Outbox pattern for reliable event dispatch
- Consumer-side deduplication
- At-least-once delivery with exactly-once effects

1.6 Data Access Strategy

EF Core – Write Model

- Aggregate root per service
- Explicit transaction boundaries
- Optimistic concurrency via `RowVersion`

Dapper – Read / Reporting Model

- Explicit SQL queries

- Projection into DTOs
- OFFSET/FETCH paging
- No entity tracking

1.7 LINQ Performance Section (Advanced Queries)

Query 1 – Avoiding N+1

```
var orders = context.Orders
    .AsNoTracking()
    .Select(o => new OrderSummaryDto
    {
        OrderId = o.Id,
        Amount = o.Amount
    })
    .ToList();
```

Translated fully to SQL; avoids navigation loading.

Query 2 – Server-side Filtering

```
var recentOrders = context.Orders
    .Where(o => o.CreatedAt >= from && o.CreatedAt <= to)
    .OrderByDescending(o => o.CreatedAt)
    .Take(50);
```

Fully server-evaluated; indexed on CreatedAt.

Query 3 – Anti-pattern (Client Evaluation to Avoid)

```
context.Orders
    .AsEnumerable()
    .Where(o => ExpensiveCheck(o));
```

Causes client-side evaluation and memory pressure — explicitly avoided.

1.8 API Gateway Responsibilities

- Authentication & authorization
- Rate limiting (protect 10k/min burst)
- Correlation ID propagation
- Aggregation endpoints (Order Summary)

---

2. Assessment B – Debug + Refactor

2.1 Given Flawed System

A single ASP.NET Core service handling: - Orders - Payments - Fulfilment - Reporting - Caching

Issues: - Violates SOLID - EF Core used for massive reads - Naive in-memory cache - LINQ N+1 queries - No gateway or policies

---

2.2 SOLID Refactor Plan

Before: Controller → DbContext → Cache → External calls

After: - Controller (HTTP only) - Application service (use cases) - Domain model (business rules) - Repository (EF Core writes) - Query service (Dapper reads)

2.3 Replace EF Reads with Dapper

- Remove `Include()` heavy queries
- Introduce SQL-based projections
- Add paging and filtering

Result: Predictable performance and lower memory usage.

2.4 Memory Leak Diagnosis

Likely leak points: 1. Static in-memory caches 2. EF Core tracking large graphs 3. Event handlers not unsubscribed 4. HttpClient misuse 5. Large object materialization

Fixes: - TTL + size-limited cache - `AsNoTracking()` - `IHttpClientFactory` - Paging everywhere

---

2.5 Microservices + CQRS Redesign

Monolith → Services: - Orders - Payments - Fulfilment - Catalog

Communication via events, not shared DB.

---

3. Assessment C – Architecture Defense

3.1 Service Boundaries

- **Orders**: Business workflow owner
- **Payments**: Financial state & retries
- **Fulfilment**: Shipping lifecycle

Separated to avoid data coupling and allow independent scaling.

3.2 Where CQRS Is NOT Used

CQRS is avoided for: - Simple admin CRUD - Low-volume internal tools

Reason: Complexity outweighs benefits.

3.3 API Gateway vs BFF

- API Gateway: cross-cutting concerns
- BFF: UI-specific shaping (not required here)

3.4 EF Core vs Dapper – Defense

Use Case	Tool
Writes	EF Core
Reads	Dapper
Reports	Dapper

3.5 Non-Relational Partition Strategy

- Partition by `CustomerId`
- Avoid hot keys
- Time-based buckets if required

3.6 Memory Leak Experience

Top causes observed: - Unbounded caches - Static collections - EF tracking - Long-lived async tasks - Large DTOs

Detection: - dotMemory - PerfView - GC logs

3.7 Azure Conceptual Mapping

Requirement	Azure Service
Event processing	Azure Functions
Workflow orchestration	Logic Apps
Cache	Azure Redis
Files	Blob Storage
Databases	Azure SQL / Cosmos DB