deti
universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

Software Architectures

VerdeMart

nopCommerce

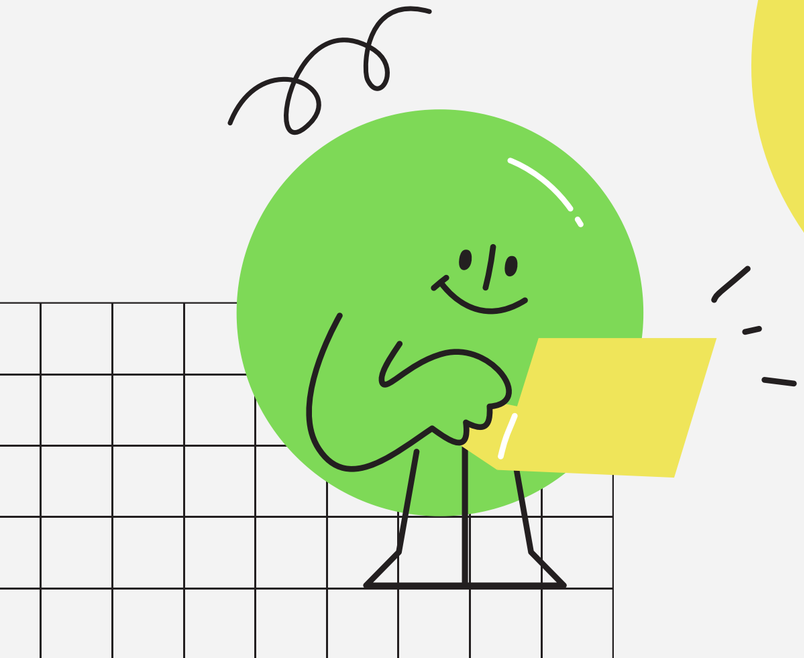
Part 2: Final Delivery

Scenario C
Architectural Evolution of nopCommerce

Presented By: Carolina Reis (131193 · Tech Lead)
Francisco Arada (131117 · Modeler)
Guilherme Silva (131143 · Architect)

03 June 2026

What You'll Discover



01 Architectural Recap
Evolution Path · ADD · Traceability chain

02 nopCommerce Implementation
OrderPublisher plugin · Outbox Pattern

03 Integration Adapter
OrderSyncAdapter · .NET Worker Service · Polly · Circuit Breaker

04 Demo: Normal Flow
Order placed → ERP/WMS via RabbitMQ

05 Demo: Degradation & Recovery
WMS Fails · DLQ · Automatic reconciliation

06 Evidence & Limitations
Measurements · Known trade-offs

07 Architecture Defense
Rejected Alternatives · What we'd do differently

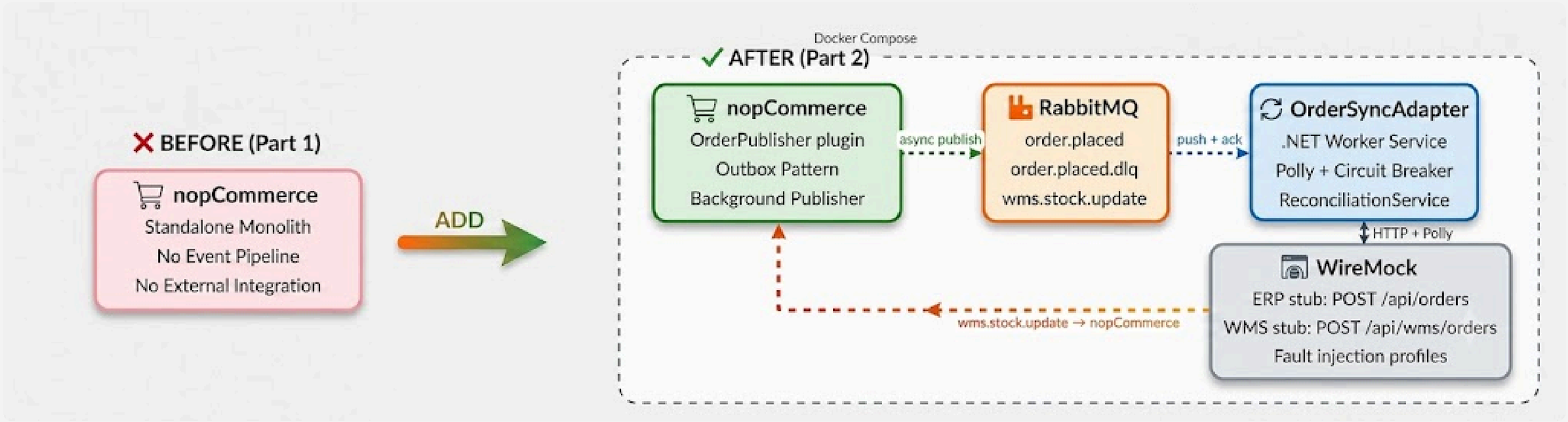
1

Architectural

Recap

Evolution Path · ADD framework · Traceability

From Monolith to Integration



Five pressure points resolved: PP-1 (sync coupling) ✓ PP-2 (no events) ✓ PP-3 (no inbound contract) ✓ PP-4 (no fallback) ✓ PP-5 (no reconciliation) ✓

ADD Traceability: Drivers → QAS → ADR → Code

Driver	QAS	Response Measure	ADR	Implementation
SG-1: Decouple checkout	QAS-1	100% orders accepted; sync ≤2 min after recovery	001	order.placed queue · Outbox-PublisherService
SG-1: No silent loss	QAS-2	Zero message loss; reconnect ≤10 s; idempotent	002	Integration_OutboxMessage table
SG-3: Isolated recovery	QAS-3	Queue drained ≤5 min; no duplicates; no manual restart	003	OrderSyncAdapter · ReconciliationService
SG-2: Cross-channel stock	QAS-4	WMS update → storefront ≤30 s; stale indicator	001	wms.stock.update queue · Wms-StockSyncService
SG-3: ERP lag containment	QAS-5	Storefront ≤500 ms regardless of ERP state	001	Async order path; checkout never calls ERP

ADR-001: RabbitMQ async

Async messaging decouples order acceptance from WMS/ERP availability.

ADR-002: Outbox Pattern

Atomic write: OutboxMessage in same DB operation as Order. At-least-once delivery.

ADR-003: OrderSyncAdapter

Independently deployable Worker Service. Only coupling: RabbitMQ queue schema.

2

nopCommerce

Implementation

OrderPublisher plugin · Outbox Pattern · WmsStockSyncService

Inside nopCommerce:

Outbox Plugin

OrderPlacedConsumer

IConsumer<OrderPlacedEvent>
Writes OutboxMessage row
(EventType + JSON payload + CorrelationId)

↓

Integration_OutboxMessage

SQL Server table
Id · EventType · Payload
CreatedOnUtc · ProcessedOnUtc · CorrelationId

↓

OutboxPublisherService

BackgroundService
Poll every 2 s → WHERE ProcessedOnUtc IS NULL
Publish + WaitForConfirmsOrDie → mark processed

↓

→ RabbitMQ order.placed

Why it works

OutboxMessage INSERT is issued after InsertOrderAsync returns (order row already committed).

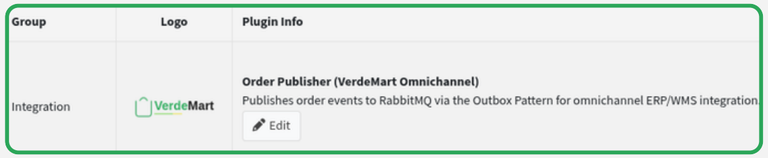
Publisher confirms: broker ack arrives before ProcessedOnUtc is written. Crash between publish and ack → re-publish on next poll: **at-least-once**.

WmsStockSyncService

Second background service in the same plugin.
Consumes wms.stock.update queue

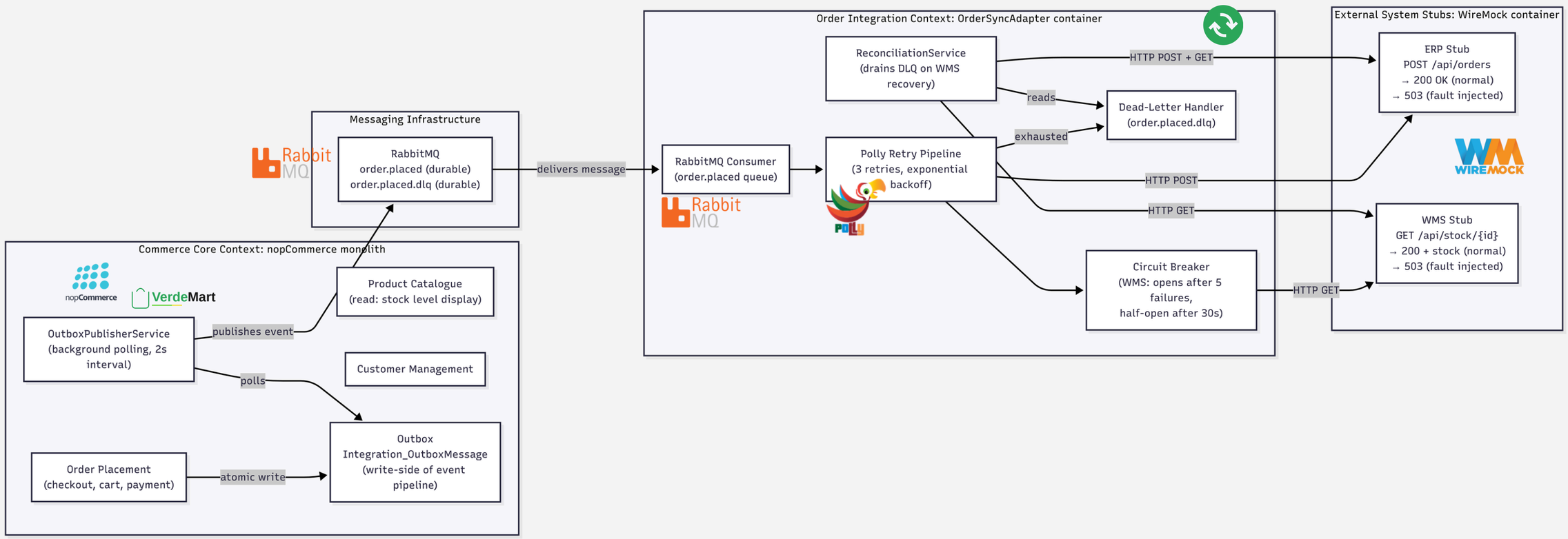
→ calls IProductService.AdjustInventoryAsync
→ stock updated in nopCommerce within seconds.

Reverse flow: WMS pushes stock changes back without manual intervention.

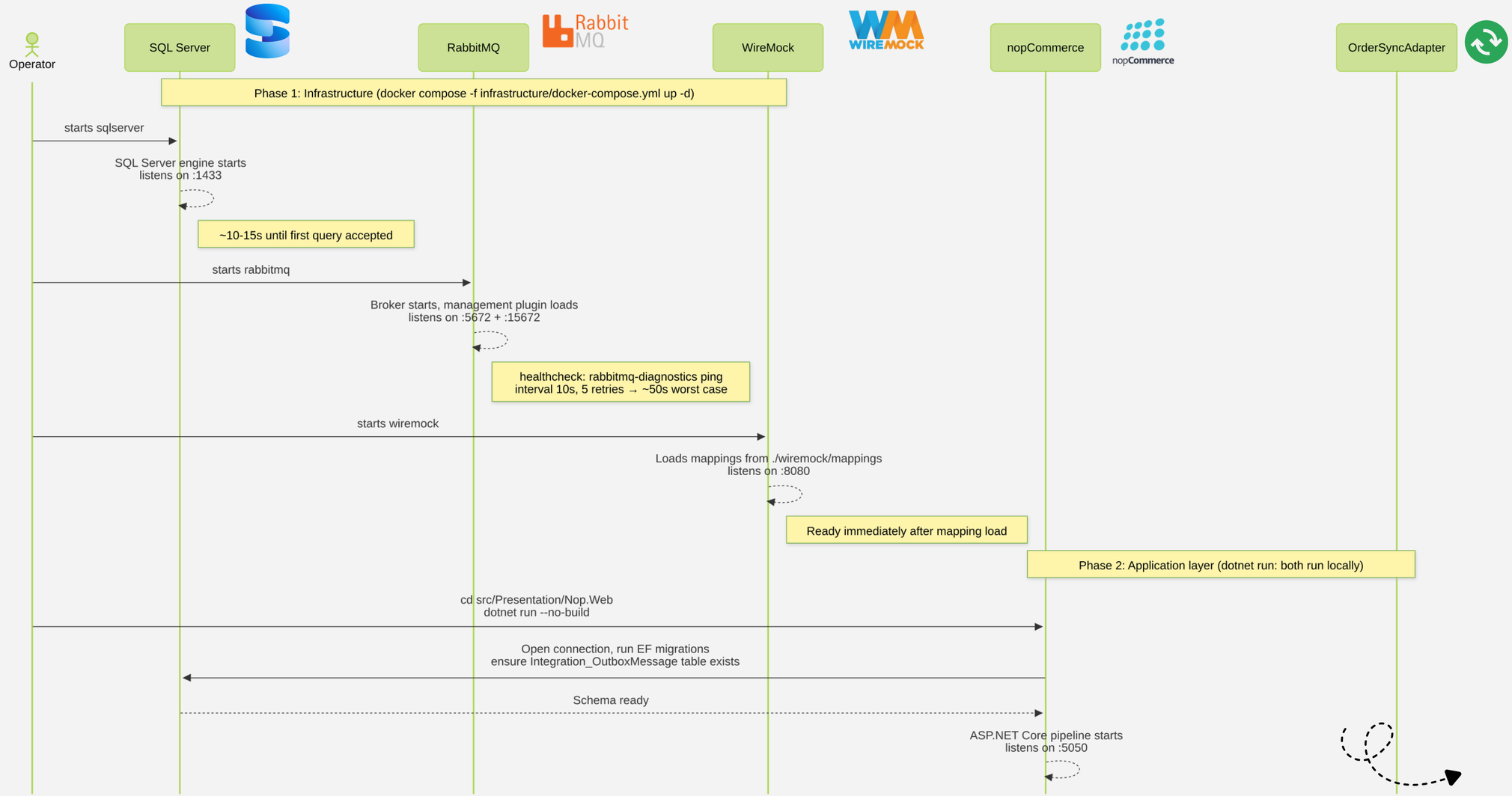


✓ Plugin independently installable:
Nop.Plugin.Integration.OrderPublisher
Zero changes to nopCommerce core.

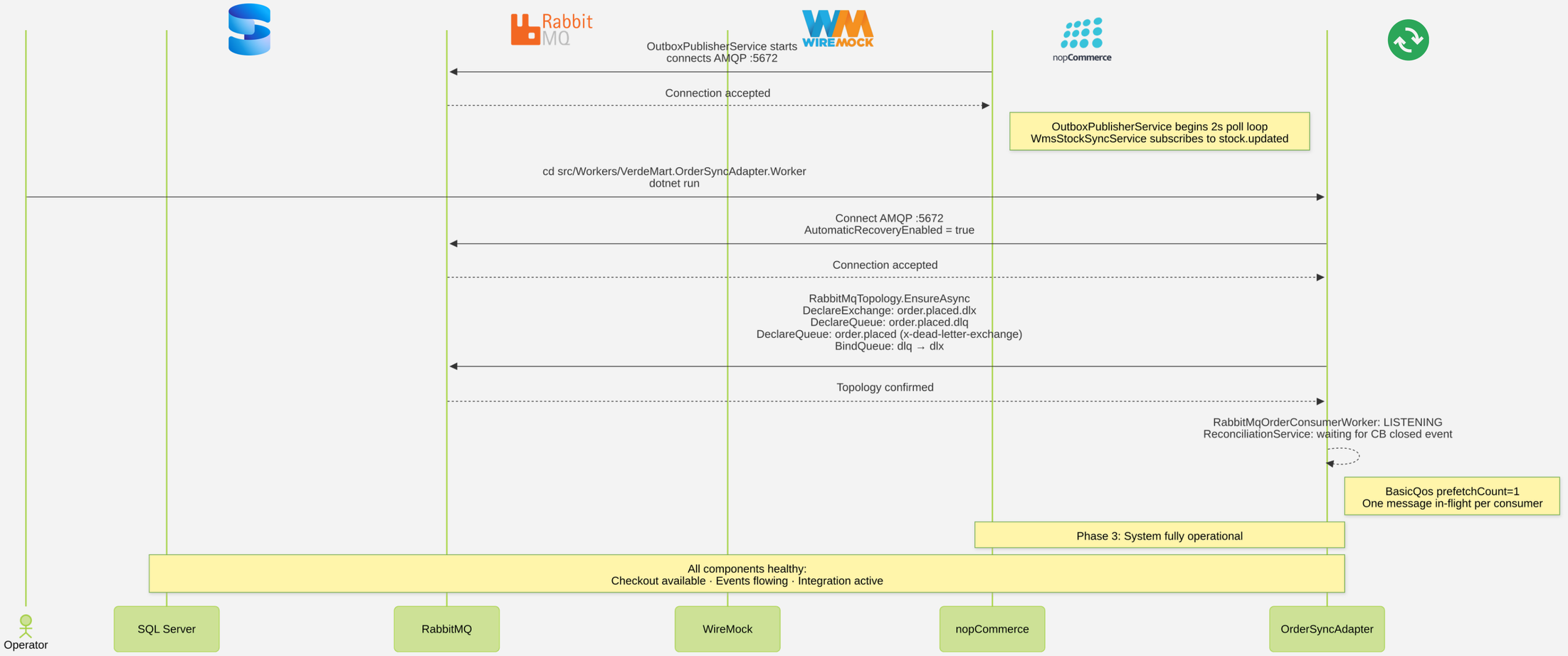
Bounded Context View



Startup Sequence & Dependency Order



Startup Sequence & Dependency Order



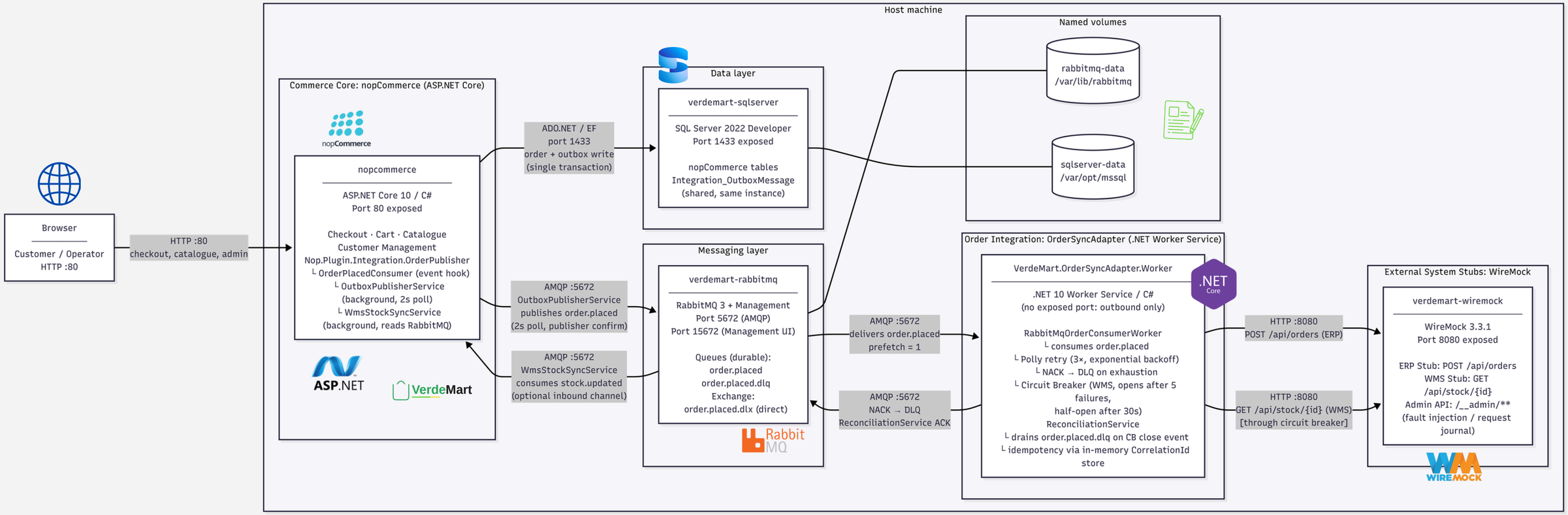
3

Integration

Adapter

OrderSyncAdapter · .NET Worker Service · Polly · Circuit Breaker · Reconciliation

OrderSyncAdapter: Resilience Architecture



Boundary: no Nop.* refs · no nopCommerce DB · only coupling = order.placed queue schema.

4

Demo:

Normal Flow

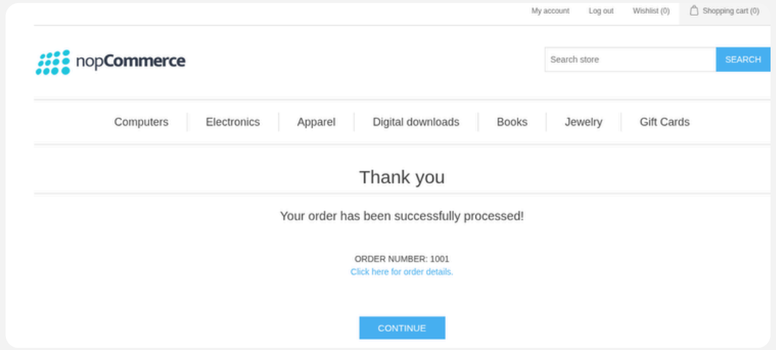
End-to-end order + cross-channel stock visibility

Use Case 1:
Buy-Online / Fulfill-Through-Another-Channel

Normal-flow sequence

1. Customer places order on storefront
2. nopCommerce writes Order + OutboxMessage (atomic)
3. Customer sees 200 OK: order confirmed
4. OutboxPublisherService polls every 2 s
5. Publishes order.placed to RabbitMQ (persistent)
6. Broker ack received → row marked processed
7. OrderSyncAdapter dequeues message
8. POST /api/orders → WireMock ERP → 200 OK
9. GET /api/stock/{id} → WireMock WMS → in_stock
10. ACK message: removed from queue

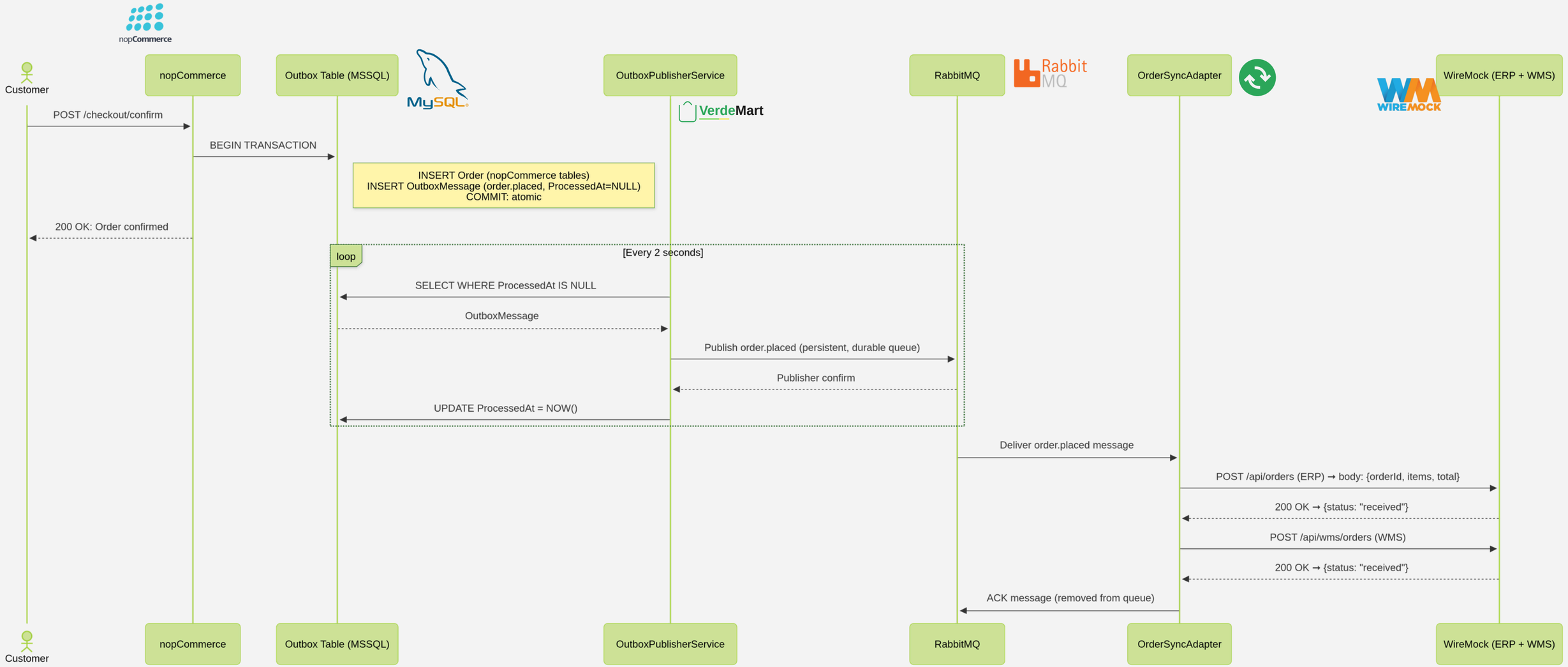
DEMO
Show docker compose up & place order on storefront.



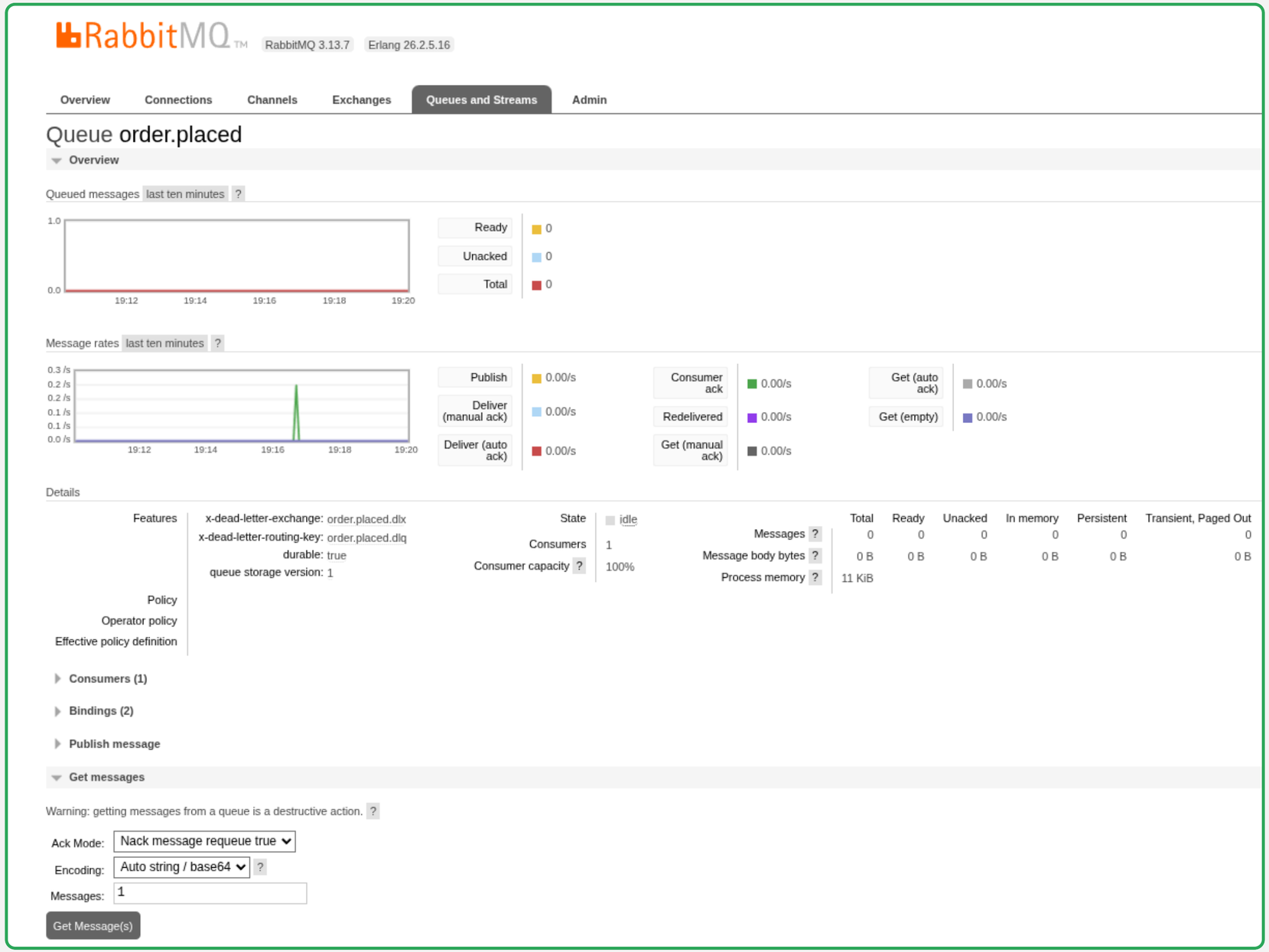
nopCommerce storefront: order confirmation page. Checkout completed independently of ERP/WMS state.

Show in RabbitMQ Management UI (localhost:15672):
queue message delivered + acked.

Show in WireMock journal (localhost:8080/__admin/requests):
ERP + WMS calls.



Normal flow: order → Outbox → RabbitMQ → ERP/WMS.



0 pending · **1 consumer** · **100% capacity**. Empty queue = OrderSyncAdapter consumed message in <2 s.

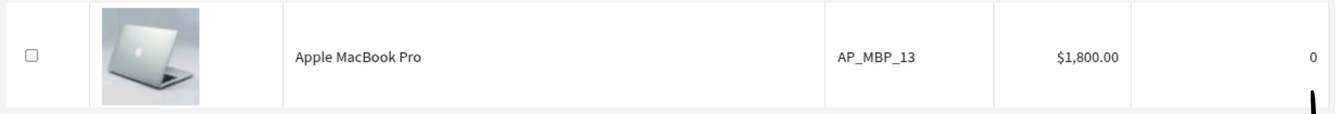
Use Case 2:
Cross-Channel Stock Visibility (WMS → nopCommerce)

Reverse flow

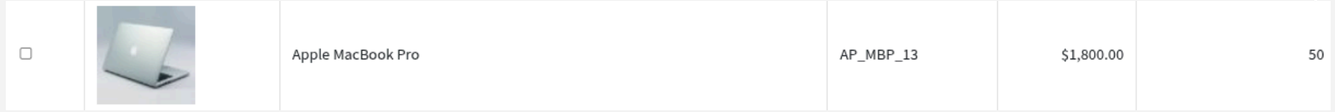
1. AP_MBP_13 stock = 0 → storefront: out of stock
2. WMS operator runs:
simulate-wms-restock.sh AP_MBP_13 50
3. Script publishes {Sku, QuantityDelta:+50} to wms.stock.update via RabbitMQ API
4. WmsStockSyncService dequeues
5. Calls AdjustInventoryAsync(+50)
6. nopCommerce stock updated → storefront: available

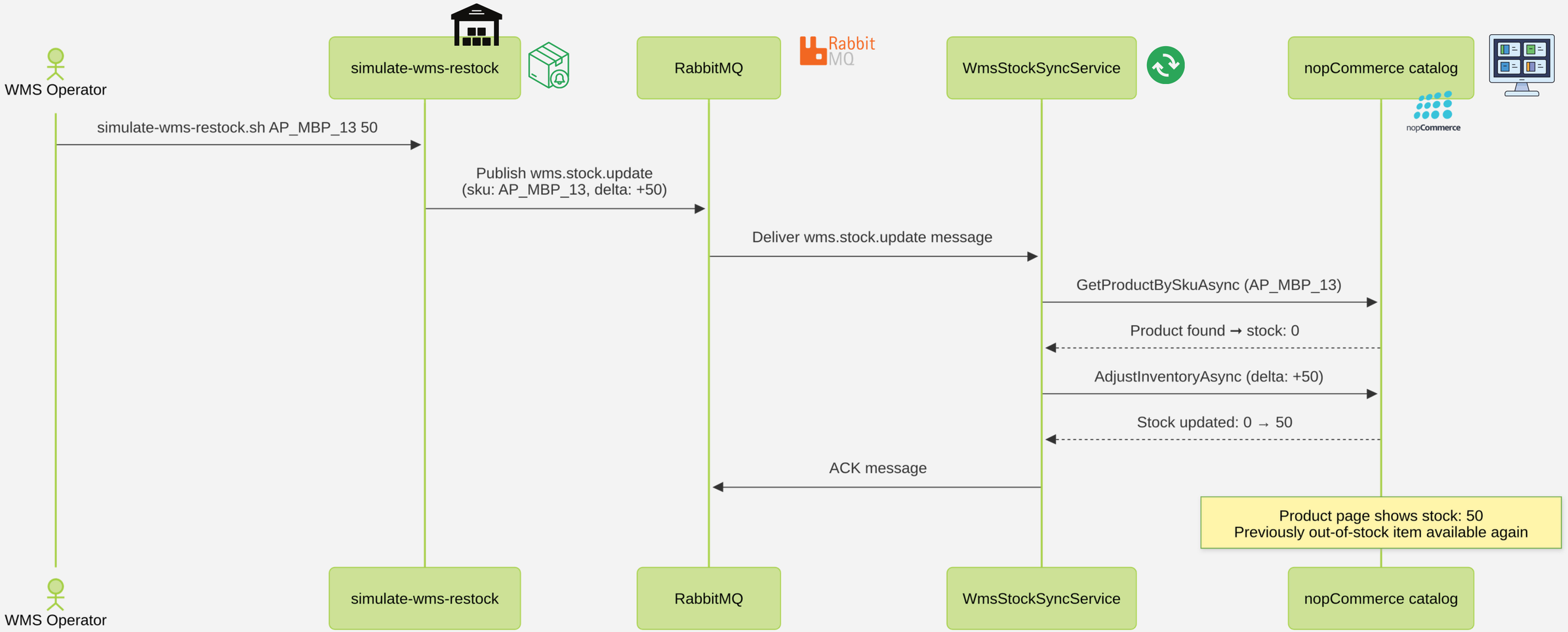
DEMO
Run script · Show admin before/after

Before: stock = 0

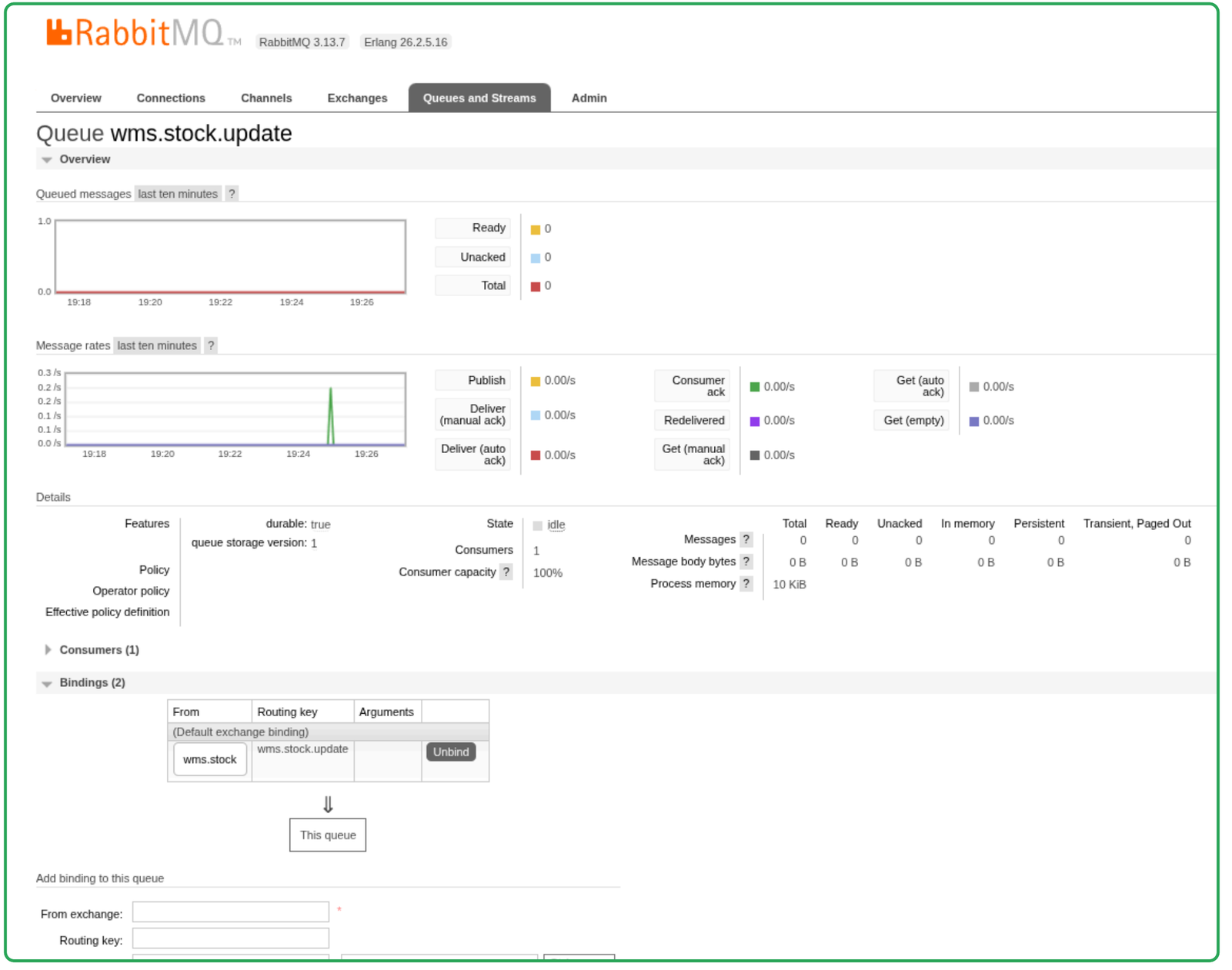


After: stock = 50





Stock sync flow: WMS event → RabbitMQ → nopCommerce catalog.



0 pending · **1 consumer** · spike visible after **simulate-wms-restock.sh**: consumed by WmsStockSyncService.

5

Demo:

Degradation & Recovery

WMS fails · DLQ · Circuit breaker · Automatic reconciliation

Mandatory Pressure Point: WMS Goes Down

Degradation sequence

1. activate-wms-failure.sh
WMS returns 503 on all calls

2. Customer places order → checkout succeeds
200 OK; Order + OutboxMessage written

3. OrderSyncAdapter processes:
ERP → 200 OK WMS → 503
Polly retry 1 (1 s) → 503
Polly retry 2 (2 s) → 503
Polly retry 3 (4 s) → 503 exhausted

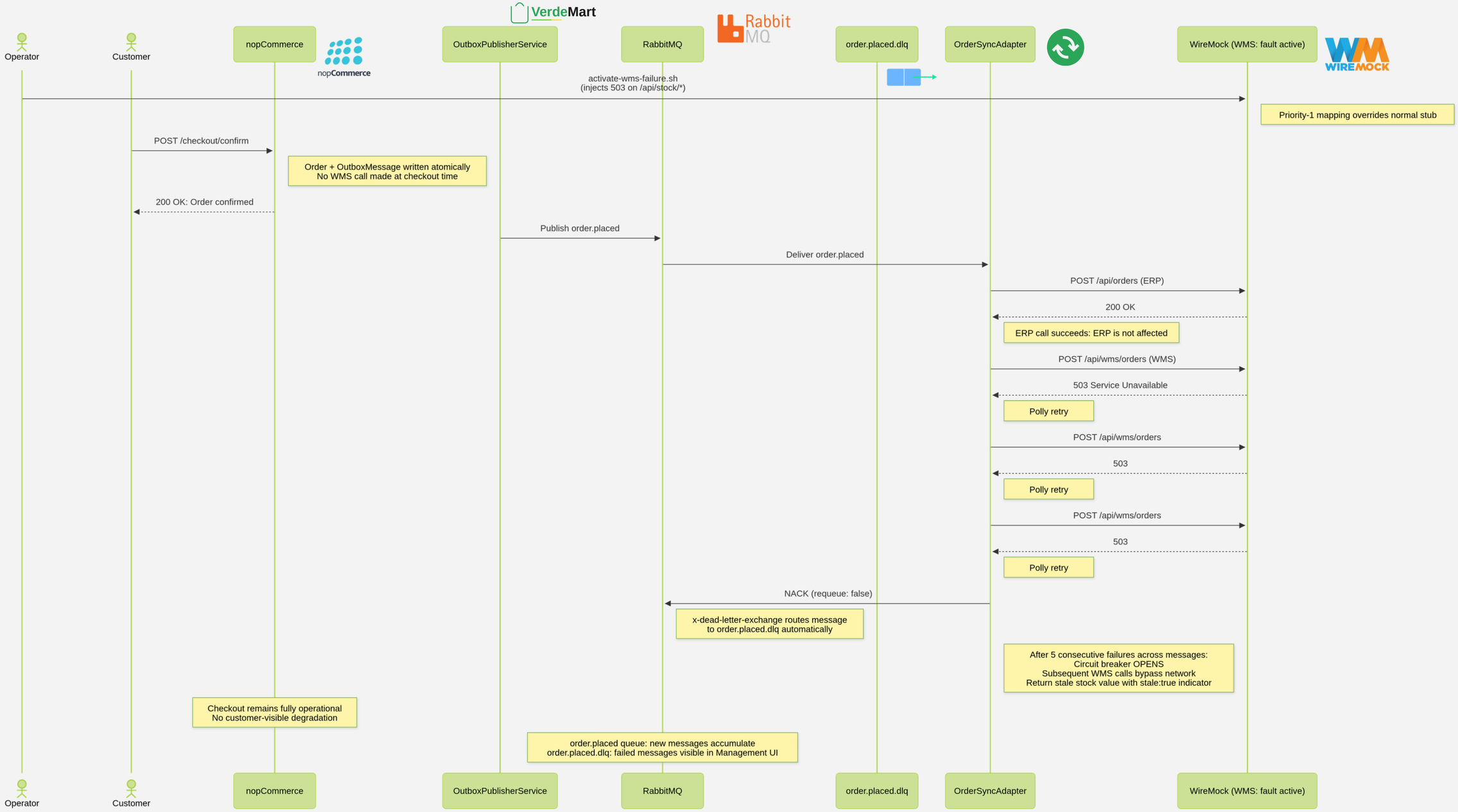
4. Message → order.placed.dlq

5. After 5 failures: Circuit breaker OPENS WMS calls skip network; stale cache served

DEMO
Run failure script · Place order · Show DLQ

```
fail: VerdeMart.OrderSyncAdapter.Implementation.WireMockOrderSyncAdapter[0]
      Sincronizacao incompleta. ERP sucesso: True; WMS sucesso: False.
fail: VerdeMart.OrderSyncAdapter.Worker.RabbitMqOrderConsumerWorker[0]
      Falha ao sincronizar a encomenda 111. Message: WMS: Timeout while calling WMS endpoint.
info: VerdeMart.OrderSyncAdapter.Worker.RabbitMqOrderConsumerWorker[0]
      A processar encomenda 112 da fila order.placed.
info: VerdeMart.OrderSyncAdapter.Implementation.WireMockOrderSyncAdapter[0]
      A iniciar sincronizacao da encomenda com ERP e WMS via WireMock.
info: System.Net.Http.HttpClient.WireMockErp.LogicalHandler[100]
      Start processing HTTP request POST http://localhost:8080/api/orders
info: System.Net.Http.HttpClient.WireMockErp.ClientHandler[100]
      Sending HTTP request POST http://localhost:8080/api/orders
info: System.Net.Http.HttpClient.WireMockWms.LogicalHandler[100]
      Start processing HTTP request POST http://localhost:8080/api/wms/orders
warn: VerdeMart.OrderSyncAdapter.Implementation.WireMockOrderSyncAdapter[0]
      Circuit breaker aberto para WMS. A devolver conteudo stale em cache para OrderId 112.
```

OrderSyncAdapter log: retries exhausted · circuit breaker OPEN.



Degraded flow: retries exhausted→ NACK → DLQ · circuit breaker opens

Recovery:
Automatic Reconciliation Without Manual Intervention

Recovery sequence

1. restore-wms. sh → WMS returns 200 OK

2. CB → HALF-OPEN (30s) → probe OK → CLOSED

3. WmsCircuitBreakerStateTracker fires event

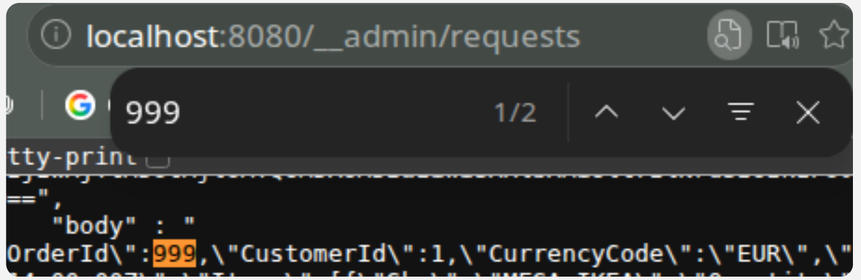
4. ReconciliationService drains DLQ (FIFO)

5. Per message: idempotency check →
POST /api/orders → ERP → 200 OK
GET /api/stock/{id} → WMS → 200 OK
ACK → DLQ emptied

6. WireMock journal: 1 POST per OrderId
No duplicates despite replay

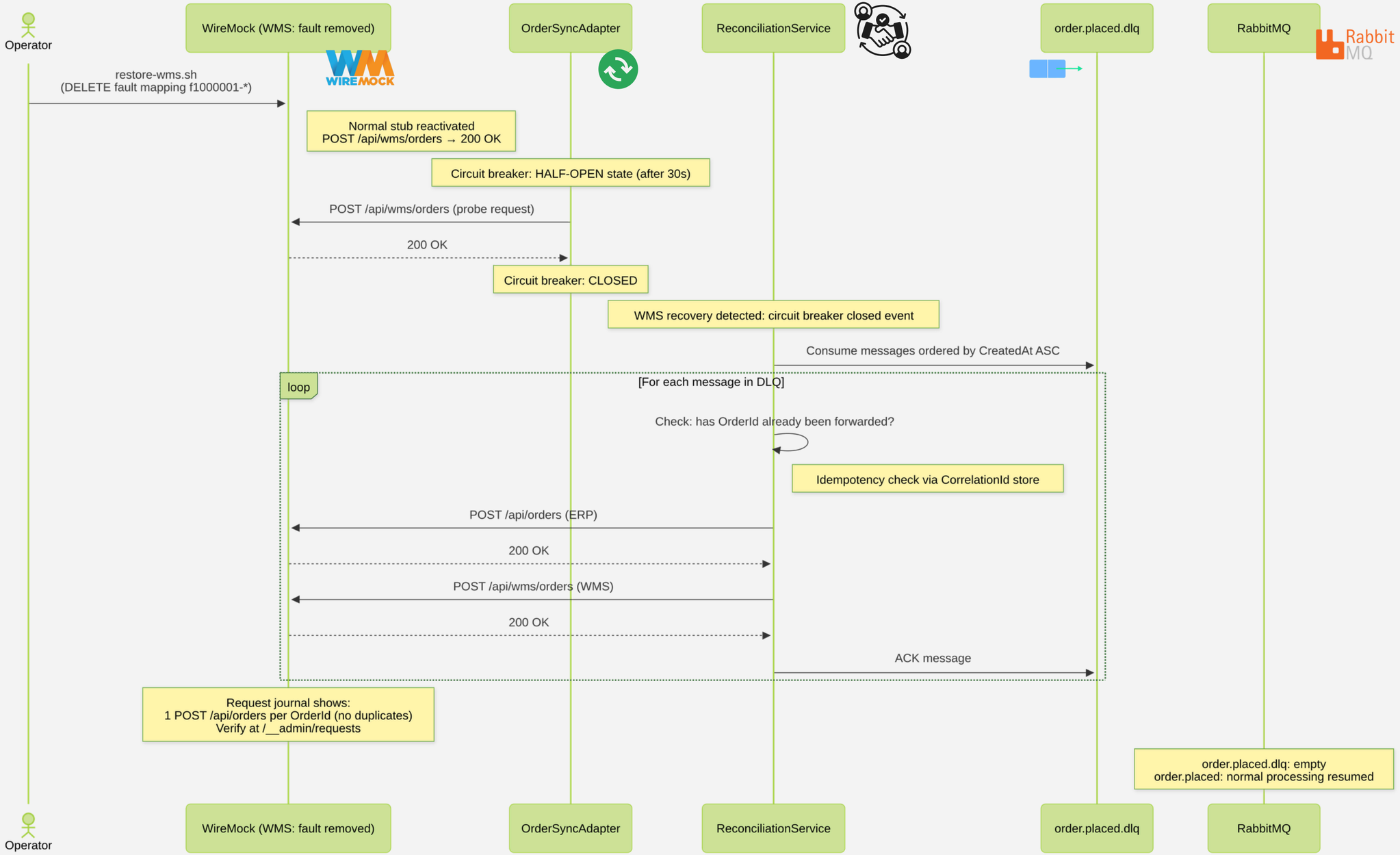
Overview					Messages			Message rates		
Virtual host	**Name**	**Type**	Features	**State**	**Ready**	**Unacked**	**Total**	**Incoming**	**deliver / get**	**ack**
/	**order.placed**	classic	D DLX DLK	running	0	0	0	0.00/s	0.00/s	0.00/s
/	**order.placed.dlq**	classic	D	running	0	0	0		0.00/s	0.00/s

RabbitMQ Management UI: DLQ empty after reconciliation.



WireMock journal: exactly 1 POST per OrderId → no duplicates.

DEMO
restore-wms.sh · DLQ drain · WireMock journal



Recovery flow: CB closes → ReconciliationService drains DLQ in order.

6

Evidence & Limitations

Measurements · QA scenario validation · Known trade-offs

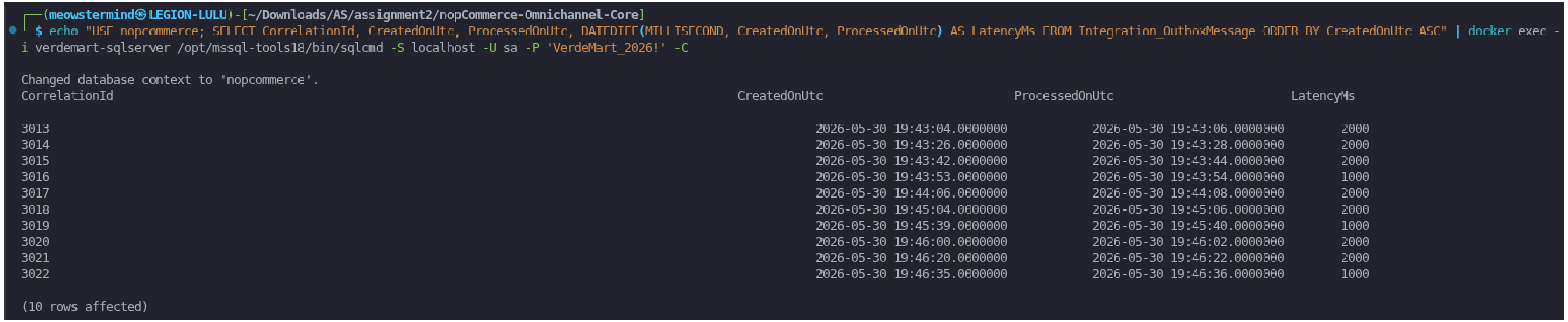
Evidence:

Measurements

1. Outbox Polling Latency (10 orders)		
Metric	Value	SLO
Average	1700 ms	⩽ 2000 ms ✓
P99 / Max	2000 ms	⩽ 2000 ms ✓

2. Broker Reconnection (3 trials, docker restart)		
Metric	Observed	SLO
Reconnection Time	11 s	⩽ 10 ms ⚠️

6 s container restart + 5 s NetworkRecoveryInterval.
Fix: reduce to 2 s.



```
┌──(meowstermind㉿LEGION-LULU)-[~/Downloads/AS/assignment2/nopCommerce-Omnichannel-Core]
└─$ echo "USE nopcommerce; SELECT CorrelationId, CreatedOnUtc, ProcessedOnUtc, DATEDIFF(MILLISECOND, CreatedOnUtc, ProcessedOnUtc) AS LatencyMs FROM Integration_OutboxMessage ORDER BY CreatedOnUtc ASC" | docker exec -
i verdemart-sqlserver /opt/mssql-tools18/bin/sqlcmd -S localhost -U sa -P 'VerdeMart_2026!' -C

Changed database context to 'nopcommerce'.
CorrelationId                                  CreatedOnUtc                      ProcessedOnUtc                    LatencyMs
---------------------------------------------- --------------------------------- --------------------------------- ---------
3013                                           2026-05-30 19:43:04.0000000       2026-05-30 19:43:06.0000000       2000
3014                                           2026-05-30 19:43:26.0000000       2026-05-30 19:43:28.0000000       2000
3015                                           2026-05-30 19:43:42.0000000       2026-05-30 19:43:44.0000000       2000
3016                                           2026-05-30 19:43:53.0000000       2026-05-30 19:43:54.0000000       1000
3017                                           2026-05-30 19:44:06.0000000       2026-05-30 19:44:08.0000000       2000
3018                                           2026-05-30 19:45:04.0000000       2026-05-30 19:45:06.0000000       2000
3019                                           2026-05-30 19:45:39.0000000       2026-05-30 19:45:40.0000000       1000
3020                                           2026-05-30 19:46:00.0000000       2026-05-30 19:46:02.0000000       2000
3021                                           2026-05-30 19:46:20.0000000       2026-05-30 19:46:22.0000000       2000
3022                                           2026-05-30 19:46:35.0000000       2026-05-30 19:46:36.0000000       1000

(10 rows affected)
```

Evidence:
Measurements

3. Publication Success Rate	
Scenario	Result
10 orders, broker restart mid-sequence	10/10 ✓

QA Scenarios Validated:

- ✓ QAS-1: 100% orders accepted under WMS outage
- ✓ QAS-2: 0 messages lost across broker restart
- ✓ QAS-3: DLQ drained; no manual restart
- ✓ QAS-4: WMS restock propagated to storefront
- ✓ QAS-5: checkout ≤500 ms; ERP lag transparent

```
┌──(meowstermind㉿LEGION-LULU)-[~/Downloads/AS/assignment2/nopCommerce-Omnichannel-Core]
└─$ echo "USE nopcommerce; SELECT COUNT(*) AS Total, SUM(CASE WHEN ProcessedOnUtc IS NOT NULL THEN 1 ELSE 0 END) AS Published FROM Integration_OutboxMessage" | docker exec -i verdemart-sqlserv
er /opt/mssql-tools18/bin/sqlcmd -S localhost -U sa -P 'VerdeMart_2026!' -C
Changed database context to 'nopcommerce'.
Total       Published
----------- -----------
         10          10

(1 rows affected)
```

```
┌──(meowstermind㉿LEGION-LULU)-[~/Downloads/AS/assignment2/nopCommerce-Omnichannel-Core]
└─$ echo "USE nopcommerce; SELECT CorrelationId, CreatedOnUtc, ProcessedOnUtc, DATEDIFF(SECOND, CreatedOnUtc, ProcessedOnUtc) AS LatencySec FROM Integration_OutboxMessage ORDER BY CreatedOnUtc
 DESC" | docker exec -i verdemart-sqlserver /opt/mssql-tools18/bin/sqlcmd -S localhost -U sa -P 'VerdeMart_2026!' -C
Changed database context to 'nopcommerce'.
CorrelationId                                                                                        CreatedOnUtc                      ProcessedOnUtc                    LatencySec
---------------------------------------------------------------------------------------------------- --------------------------------- --------------------------------- -----------
3032                                                                                                  2026-05-30 21:24:20.0000000       2026-05-30 21:24:20.0000000           0
3031                                                                                                  2026-05-30 21:24:03.0000000       2026-05-30 21:24:14.0000000          11
3030                                                                                                  2026-05-30 21:19:18.0000000       2026-05-30 21:19:18.0000000           0
3029                                                                                                  2026-05-30 21:19:01.0000000       2026-05-30 21:19:12.0000000          11
3028                                                                                                  2026-05-30 21:16:25.0000000       2026-05-30 21:16:26.0000000           1
3027                                                                                                  2026-05-30 21:16:05.0000000       2026-05-30 21:16:16.0000000          11

(6 rows affected)
```

Honest Limitations

ID	Pattern	What it means	What production needs
L-1	At-least-once delivery	Crash between publish and ack → duplicate message in queue	Idempotency key survives restarts (Redis/DB, not in-memory dict)
L-2	Outbox transaction gap	Order INSERT and OutboxMessage INSERT are not in same DB txn	Compensating scan for orders with no corresponding Outbox row
L-3	Fixed polling interval	Worst-case 2 s Outbox-to-queue latency (OK for async B2C)	Reduce to 200–500 ms or use push based trigger for real-time
L-4	WireMock ≠ real ERP/WMS	No pool exhaustion, no auth expiry, no rate limiting	Full ERP integration test suite; mock only at unit level
L-5	Identity out of scope	customerId from nopCommerce forwarded as-is to ERP	Unified identity layer (Keycloak or CDP) for omnichannel customers
L-6	Single OutboxPublisher instance	Two nopCommerce instances → duplicate Outbox drain	Distributed lock on drain loop (SELECT FOR UPDATE SKIP LOCKED)

✔️ These are deliberate scope cuts, not hidden defects!

L-1 is eliminated by OrderId idempotency (validated in evidence/scenarios/3-idempotency-validation.md).

7

Architecture

Defense

Rejected alternatives · Trade-offs · What we'd do differently

What We Rejected and Why

ADR-001: AsyncMessaging

Sync REST
POST to ERP at checkout time. ERP outage = checkout outage. Violates QAS-1 directly.

Webhooks
nopCommerce POSTs to subscribers on order. Retry/timeout managed inside nopCommerce core. Broker does this better!

ADR-002: Outbox Pattern

Dual-write
BasicPublish() inside SaveOrder(). No atomicity. Crash = committed order, no event.

2PC / MSDTC
Distributed txn over SQL Server + RabbitMQ. RabbitMQ has no XA support. Adds SPOF coordinator.

Debezium CDC
Stream DB changes into Kafka/RabbitMQ.
Requires binary log access; couples event schema to DB schema.

What We Rejected and Why

ADR-003: OrderSyncAdapter Extraction

Inline plugin
RabbitMQ consumer inside nopCommerce process. No enforced boundary. Crash in integration → storefront down. Cannot scale independently.

Azure Function
Cloud FaaS triggered by queue messages. Hard cloud dependency. Obscures boundary. Breaks local demo reproducibility.

The pattern behind all rejections

Every rejected alternative either:

✓ introduces a synchronous dependency on the order path, or
✓ creates hidden coupling across service boundaries, or
✓ adds infrastructure complexity disproportionate to the problem.

The accepted decisions (RabbitMQ + Outbox + extracted adapter) **solve all three pressure points** with the minimum moving parts.

What We'd Do Differently With More Time

Still-open architectural questions

Exactly-once delivery:
Kafka with idempotent producers would fix the in-memory idempotency limitation (L-1). Justified only at production scale.

Outbox transaction boundary:
SELECT FOR UPDATE SKIP LOCKED would support multiple nopCommerce instances (L-6). Single-instance is acceptable for demo.

Stock write path:
WmsStockSyncService mutates stock directly. A read-only projection with staleness indicator would be cleaner.

Production gaps we'd close first

Authentication at integration boundary:
WireMock accepts all requests without auth. Production needs mutual TLS or API key between OrderSyncAdapter and ERP/WMS.

Distributed idempotency store:
_processedOrders is in-memory — lost on restart. Production: Redis set or DB table keyed by OrderId.

Observability layer:
CorrelationId logs are in place. Production: OpenTelemetry tracing across nopCommerce → RabbitMQ → OrderSyncAdapter → ERP/WMS.

Part 2

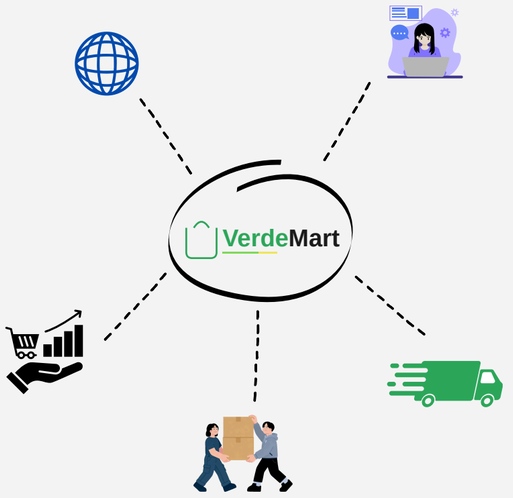
References

[1] RabbitMQ. Dead Letter Exchanges. RabbitMQ Documentation. https://www.rabbitmq.com/docs/dlx

[2] App-vNext. (2026). Polly. GitHub. https://github.com/App-vNext/Polly

[3] Microsoft. (2023). Implement the Circuit Breaker pattern. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/implement-circuit-breaker-pattern

[4] Microsoft. (2023). Implement HTTP call retries with exponential backoff with IHttpClientFactory and Polly policies. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/implement-http-call-retries-exponential-backoff-polly



Part 2

References

[5] WireMock. WireMock Java - API Mocking for Java and JVM. https://wiremock.org/docs/

[6] Microsoft. (2025). Worker services in .NET. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/core/extensions/workers

[7] RabbitMQ. Publishers. RabbitMQ Documentation. https://www.rabbitmq.com/docs/publishers

[8] Microsoft. (2022). Asynchronous message-based communication. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication