

Service-Oriented Computing Module

February 2026

Agenda

- | | |
|--|-------|
| ① Context and Motivation – Why four paradigms in one system? | 2 min |
| ② System Architecture – Master diagram and data flows | 3 min |
| ③ Paradigm Deep Dive – Contract, sequence, and trade-offs | 6 min |
| ④ Cross-Paradigm Analysis – Decision tree, payload, aggregation | 5 min |
| ⑤ Comparison Matrix and Takeaways | 2 min |

SOAP

REST

GraphQL

gRPC

The Retail Integration Problem

A global retail brand must communicate with **four fundamentally different stakeholders**, each imposing distinct technical constraints:

SOAP Manufacturers

Legacy ERP systems (SAP, Oracle) that expose SOAP/WS-* interfaces natively. Imposing REST would require an additional translation layer.

GraphQL Store Managers

Internal dashboards aggregating inventory, orders, and robot telemetry. A REST-based approach requires $O(N)$ calls per store.

REST Partner Boutiques

Third-party developers need the simplest integration possible. HTTP + JSON, no SDK, no code generation step.

gRPC Warehouse Robots

100+ robots streaming telemetry at 10 msg/sec. Bandwidth and latency constraints prohibit text-based formats.

Methodology: Contract-First Design

Every module follows a strict contract-first workflow:

- ① Define the contract** – WSDL, OpenAPI, SDL, or Protobuf – written *before* any implementation code.
- ② Implement the mock server** – returns realistic data conforming strictly to the contract.
- ③ Validate with automated tests** – Postman collection with status code and payload assertions.
- ④ Analyse trade-offs** – document what was gained and what was sacrificed for each choice.

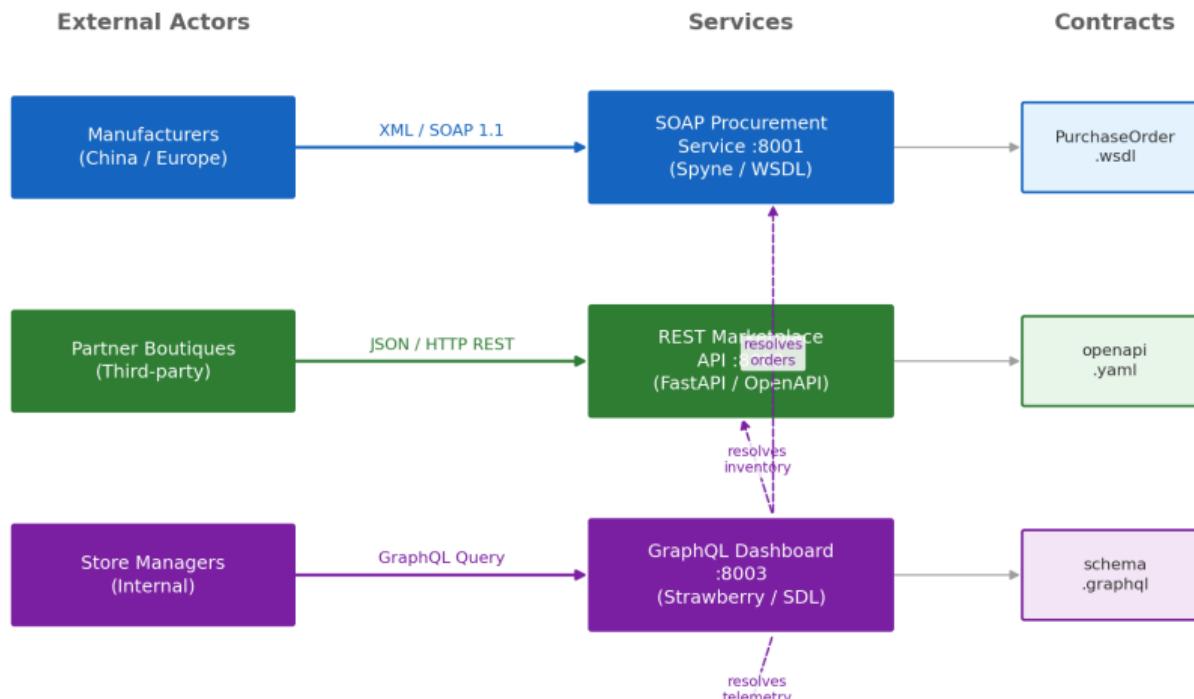
Module	Contract File
SOAP	PurchaseOrder.wsdl
REST	openapi.yaml
GraphQL	schema.graphql
gRPC	warehouse.proto

Principle

The contract is the shared source of truth between producer and consumer. Code is derived from the contract, never the reverse.

RetailSync – System Architecture

RetailSync – System Architecture



Contract: PurchaseOrder.wsdl

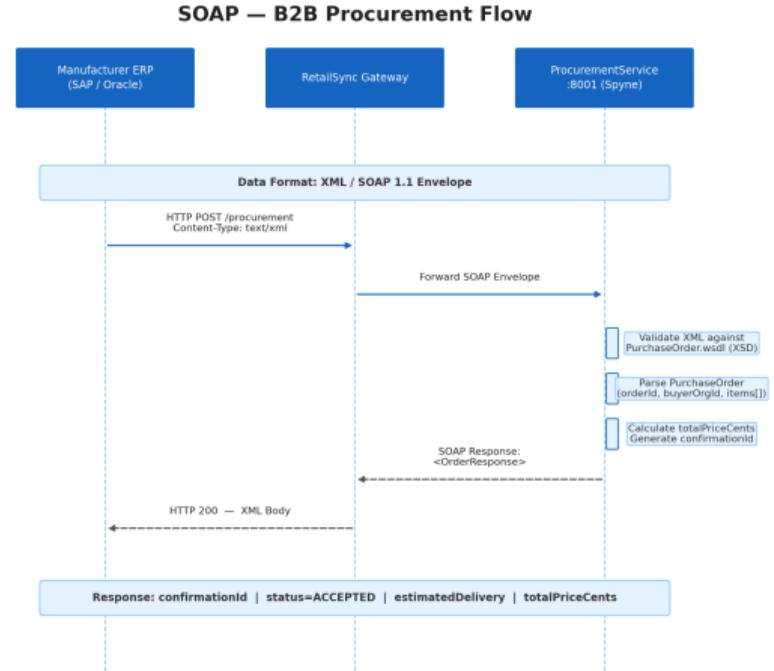
Framework: Spyne (Python SOAP)

Justification:

- **XSD validation** enforces field types, ordering, and cardinality at the wire level.
- **WSDL code generation** – both parties generate stubs from one file.
- **WS-Security** provides message-level encryption, not just transport-level TLS.

+ Strict type safety
+ Formal contract
+ WS-Security

- Human readability
- Heavy toolkits
- Parsing performance



Contract: openapi.yaml (OpenAPI 3.0)

Framework: FastAPI + Pydantic

Justification:

- **Zero integration barrier** – any HTTP client, any language, no SDK required.
- **HTTP caching** – GET responses are cacheable by CDNs and reverse proxies.
- **Semantic verbs** – GET, PATCH, DELETE are self-documenting.

+ Universal adoption

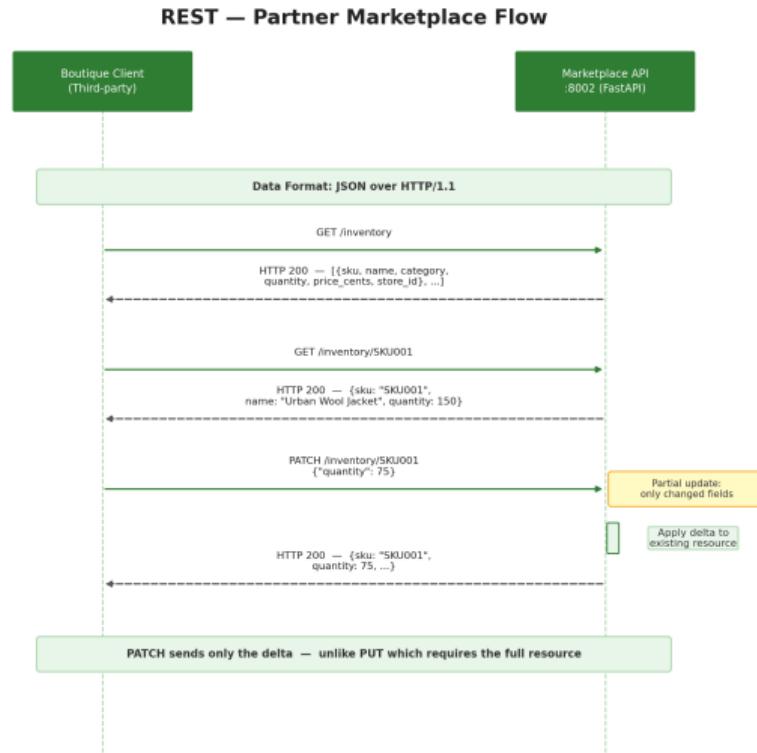
– No schema enforcement

+ HTTP caching

– Under-fetching
 $(N+1)$

+ Stateless scaling

– No native streaming



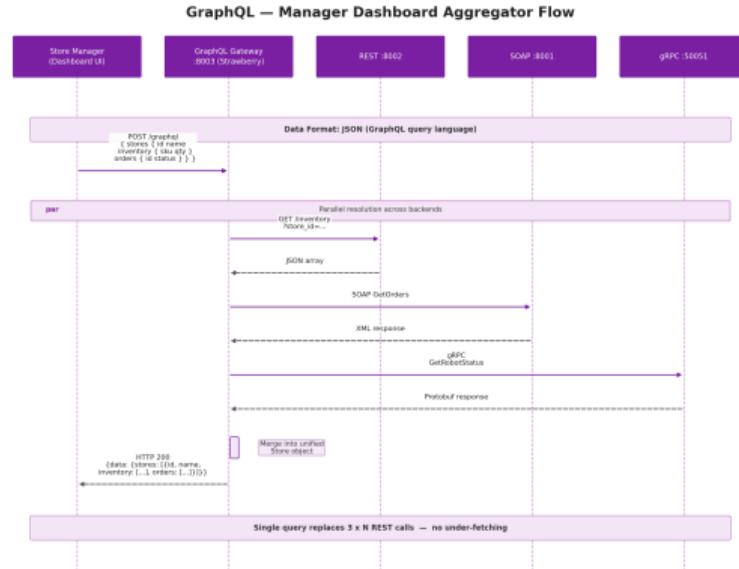
Contract: schema.graphql (SDL)

Framework: Strawberry + FastAPI

Justification:

- **Eliminates under-fetching** – nested queries collapse $3 \times N$ REST calls into one POST.
- **Client-driven shape** – each widget requests only the fields it needs.
- **Aggregator pattern** – frontend is decoupled from backend topology.

- | | |
|-----------------------|---------------------|
| + Single request | - No HTTP caching |
| + Client-driven shape | - Query complexity |
| + Backend-agnostic | - Resolver overhead |



Contract: warehouse.proto (Protocol Buffers)

Framework: grpcio + protobuf

Justification:

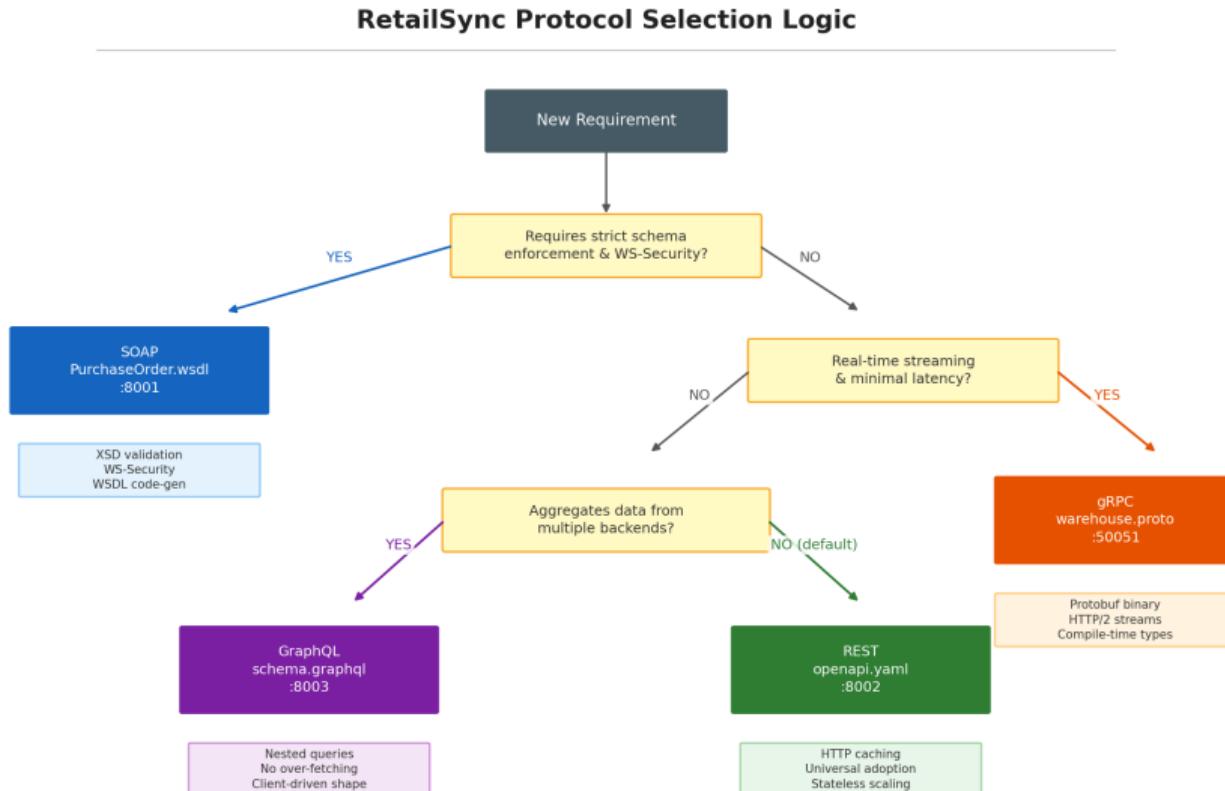
- **Binary serialisation** – Protobuf encodes a telemetry message in ~50 B vs ~1 KB for XML.
- **Bi-directional streaming** – commands and telemetry flow simultaneously on one HTTP/2 connection.
- **Compile-time types** – field errors caught before deployment via code generation.

+ 20x smaller payloads - Not human-readable
 RetailSync – Comparative API Study
 + Bi-directional stream - No browser support

gRPC – Warehouse Robot Bi-directional Streaming



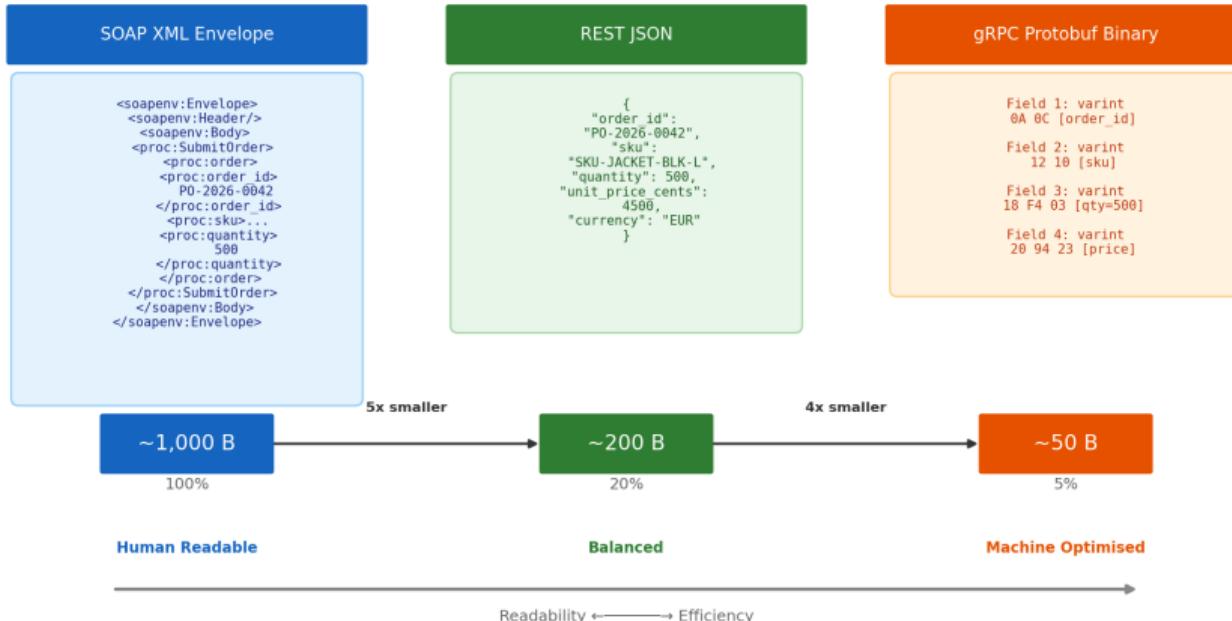
Protocol Selection Decision Tree



Payload Overhead Comparison

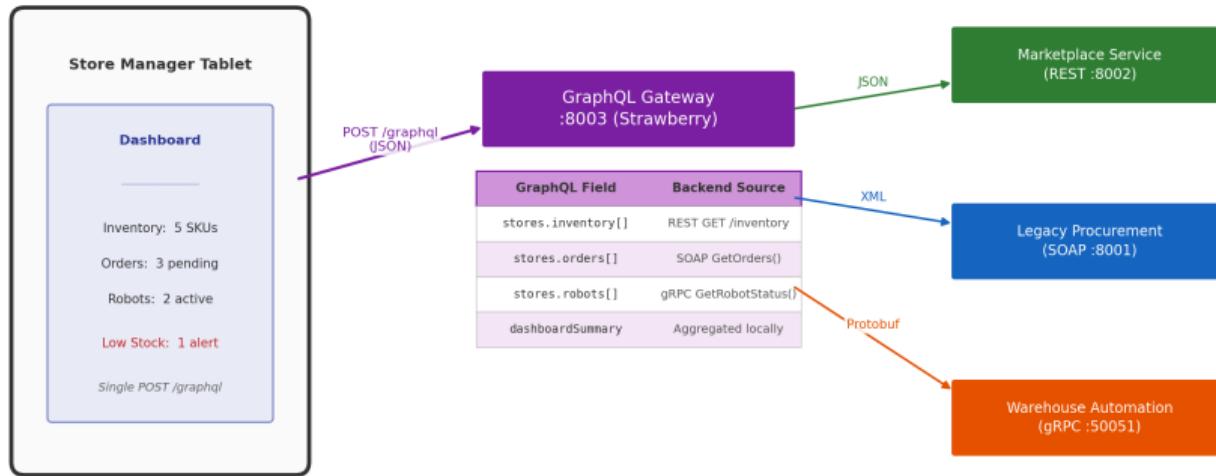
Payload Overhead Comparison — Order Object

Same business data (SKU, quantity, price) encoded in three wire formats



GraphQL Aggregator Pattern

GraphQL: The Omnichannel Unified Interface



Final Comparison Matrix

Criterion	SOAP	REST	GraphQL	gRPC
Data format	XML	JSON	JSON	Protobuf
Transport	HTTP/1.1	HTTP/1.1	HTTP/1.1	HTTP/2
Contract	WSDL (mandatory)	OpenAPI (optional)	SDL Schema	.proto (mandatory)
Type safety	Strict (XSD)	Runtime (Pydantic)	Runtime (resolvers)	Compile-time
Streaming	No	No	Subscriptions	Bi-directional
Caching	No	Yes (HTTP GET)	No (POST)	No
Browser support	Via AJAX	Native	Native	gRPC-Web proxy
Best suited for	Formal B2B	Public APIs	Aggregation	Machine-to-machine

There is no universally superior paradigm. Each cell represents a design decision with measurable consequences.

Key Takeaways

- ① **No universal winner.** Each paradigm excels within its specific operational constraints.
The architecture must be polyglot by design.
- ② **Contract-first is non-negotiable.** Whether WSDL, OpenAPI, SDL, or Protobuf, the contract is the shared source of truth between distributed teams.
- ③ **REST as default, deviate with justification.** Every non-REST technology choice in RetailSync is driven by a concrete limitation that REST cannot address.
- ④ **GraphQL complements, it does not replace.** It acts as an aggregation gateway over heterogeneous backends. Both REST and GraphQL coexist by design.
- ⑤ **The cost of the wrong paradigm is measurable.** At 100 robots \times 10 msg/sec, the difference between 1 KB (SOAP) and 50 B (gRPC) is the difference between network saturation and idle capacity.

Final Reflection

Architectural maturity is not knowing the latest framework. It is knowing when *not* to use it.

Thank You

Questions?

SOAP :8001 **REST :8002** **GraphQL :8003** **gRPC :50051**

github.com/AyaMor/omnichain-retail-mesh