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	**Live Demonstration** – Running all four services	*3 min*
5	**Cross-Paradigm Analysis** – Decision tree, payload, aggregation	*2 min*
6	**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:

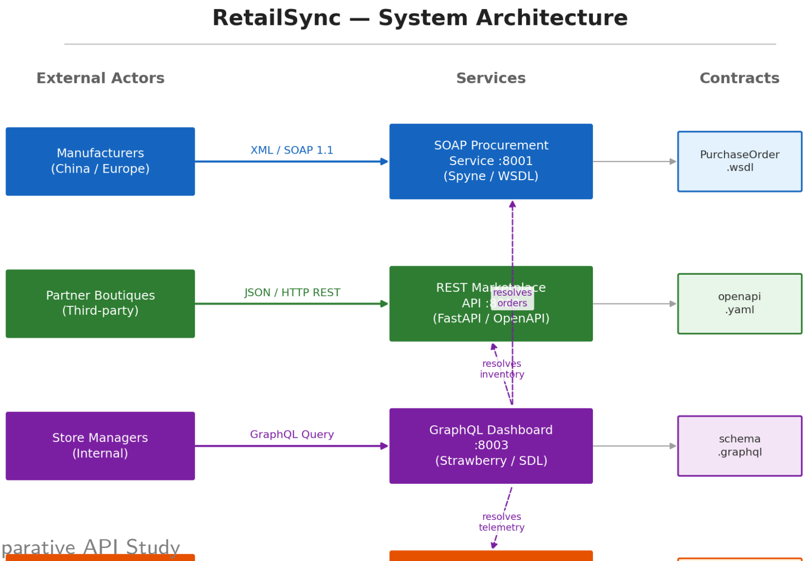
1. **Define the contract** – WSDL, OpenAPI, SDL, or Protobuf – written *before* any implementation code.
2. **Implement the mock server** – returns realistic data conforming strictly to the contract.
3. **Validate with automated tests** – Postman collection with status code and payload assertions.
4. **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

External Actors

Services

Contracts

Manufacturers
(China / Europe)

XML / SOAP 1.1

SOAP Procurement
Service :8001
(Spyne / WSDL)

PurchaseOrder
.wsdl

Partner Boutiques
(Third-party)

JSON / HTTP REST

REST Marketplace
API :8
(FastAPI / OpenAPI)

resolves
orders

openapi
.yaml

resolves
inventory

Store Managers
(Internal)

GraphQL Query

GraphQL Dashboard
:8003
(Strawberry / SDL)

schema
.graphql

resolves
telemetry

SOAP B2B Procurement Service – Port 8001

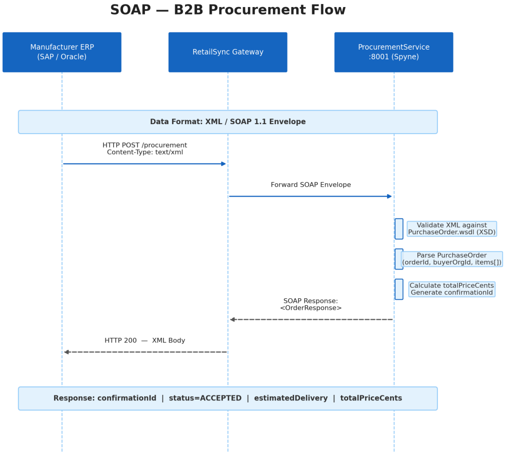
Contract: `PurchaseOrder.wsdl`
Framework: Spyne (Python SOAP)

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


SOAP — B2B Procurement Flow
Manufacturer ERP (SAP / Oracle)
RetailSync Gateway
ProcurementService :8001 (Spyne)
Data Format: XML / SOAP 1.1 Envelope
HTTP POST /procurement Content-Type: text/xml
Forward SOAP Envelope
Validate XML against PurchaseOrder.wsdl (XSD)
Parse PurchaseOrder (orderId, buyerOrgId, items[])
Calculate totalPriceCents Generate confirmationId
SOAP Response: <OrderResponse>
HTTP 200 — XML Body
Response: confirmationId | status=ACCEPTED | estimatedDelivery | totalPriceCents

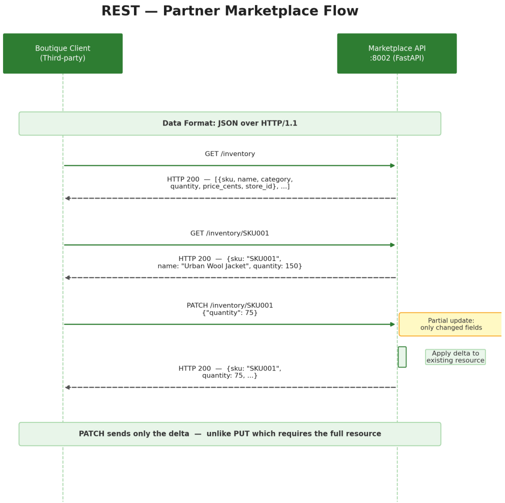
REST Partner Marketplace API – Port 8002

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



REST — Partner Marketplace Flow

Boutique Client (Third-party)

Marketplace API :8002 (FastAPI)

Data Format: JSON over HTTP/1.1

GET /inventory

HTTP 200 — [{sku, name, category, quantity, price_cents, store_id}, ...]

GET /inventory/SKU001

HTTP 200 — {sku: "SKU001", name: "Urban Wool Jacket", quantity: 150}

PATCH /inventory/SKU001 {"quantity": 75}

Partial update: only changed fields

Apply delta to existing resource

HTTP 200 — {sku: "SKU001", quantity: 75, ...}

PATCH sends only the delta — unlike PUT which requires the full resource

GraphQL Manager Dashboard – Port 8003

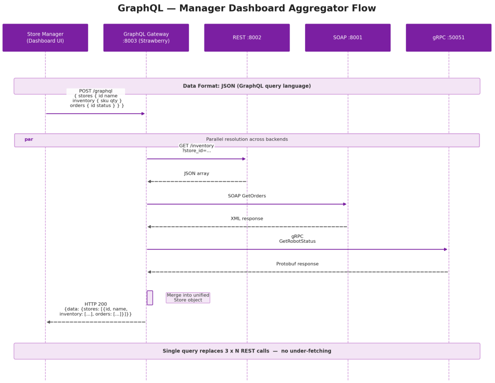
Contract: `schema.graphql` (SDL)
Framework: Strawberry + FastAPI

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
+ Client-driven shape
+ Backend-agnostic

– No HTTP caching
– Query complexity
– Resolver overhead

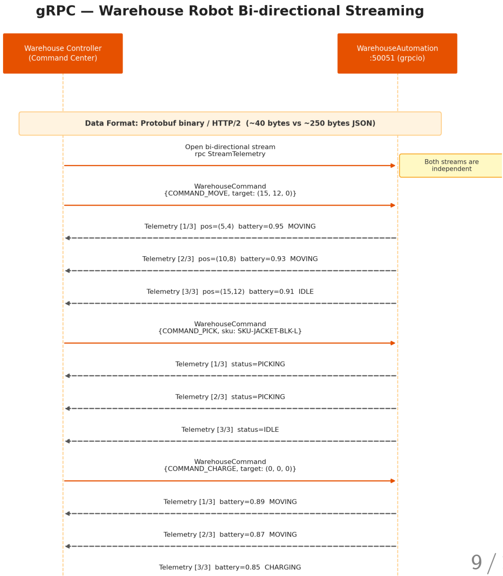

GraphQL — Manager Dashboard Aggregator Flow
Store Manager (Dashboard UI)
GraphQL Gateway :8003 (Strawberry)
REST :8002
SOAP :8001
gRPC :50051
Data Format: JSON (GraphQL query language)
POST /graphql { stores { id name inventory { sku qty } orders { id status } } }
par
Parallel resolution across backends
GET /inventory ?store_id=...
JSON array
SOAP GetOrders
XML response
gRPC GetRobotStatus
Protobuf response
Merge into unified Store object
HTTP 200 {data: {stores: [{id, name, inventory: [...], orders: [...]}]}}
Single query replaces 3 x N REST calls — no under-fetching

gRPC Warehouse Automation – Port 50051

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.


gRPC — Warehouse Robot Bi-directional Streaming
Warehouse Controller (Command Center)
WarehouseAutomation :50051 (grpcio)
Data Format: Protobuf binary / HTTP/2 (~40 bytes vs ~250 bytes JSON)
Open bi-directional stream rpc StreamTelemetry
Both streams are independent
WarehouseCommand {COMMAND_MOVE, target: (15, 12, 0)}
Telemetry [1/3] pos=(5,4) battery=0.95 MOVING
Telemetry [2/3] pos=(10,8) battery=0.93 MOVING
Telemetry [3/3] pos=(15,12) battery=0.91 IDLE
WarehouseCommand {COMMAND_PICK, sku: SKU-JACKET-BLK-L}
Telemetry [1/3] status=PICKING
Telemetry [2/3] status=PICKING
Telemetry [3/3] status=IDLE
WarehouseCommand {COMMAND_CHARGE, target: (0, 0, 0)}
Telemetry [1/3] battery=0.89 MOVING
Telemetry [2/3] battery=0.87 MOVING
Telemetry [3/3] battery=0.85 CHARGING

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

Live Demonstration – Setup

All four services running locally, each on its designated port:

Service	Command	Port	Validation
SOAP	`python procurement/mock-server/server.py`	8001	WSDL served at `/?wsdl`
REST	`python marketplace/mock-server/server.py`	8002	Swagger UI at `/docs`
GraphQL	`python dashboard/mock-server/server.py`	8003	GraphiQL at `/graphql`
gRPC	`python logistics/mock-server/server.py`	50051	Client script

Demonstration Plan

We will demonstrate the following scenario in sequence:

1. Submit a purchase order via SOAP (XML envelope).
2. Query and update marketplace inventory via REST (GET, PATCH).
3. Aggregate all data through a single GraphQL query.
4. Stream robot commands and receive telemetry via gRPC.

Live Demonstration – Execution

Step 1: SOAP – Submit Purchase Order

```
curl -X POST http://localhost:8001/ \
  -H "Content-Type:␣text/xml" \
  -d @procurement/mock-server/test_request.xml
```

Expected: XML response with `order_id` and `CONFIRMED` status.

Step 2: REST – Inventory Operations

```
# List all inventory
curl http://localhost:8002/inventory

# Update stock (PATCH = delta only)
curl -X PATCH http://localhost:8002/inventory/
     SKU001 \
  -H "Content-Type:␣application/json" \
  -d ’{"quantity":␣75}’
```

Step 3: GraphQL – Aggregated Query

```
curl -X POST http://localhost:8003/graphql \
  -H "Content-Type:␣application/json" \
  -d ’{"query":␣"{␣stores␣{␣id␣name
␣␣␣␣inventory␣{␣sku␣quantity␣}
␣␣␣␣orders␣{␣id␣status␣}␣}␣}"}’
```

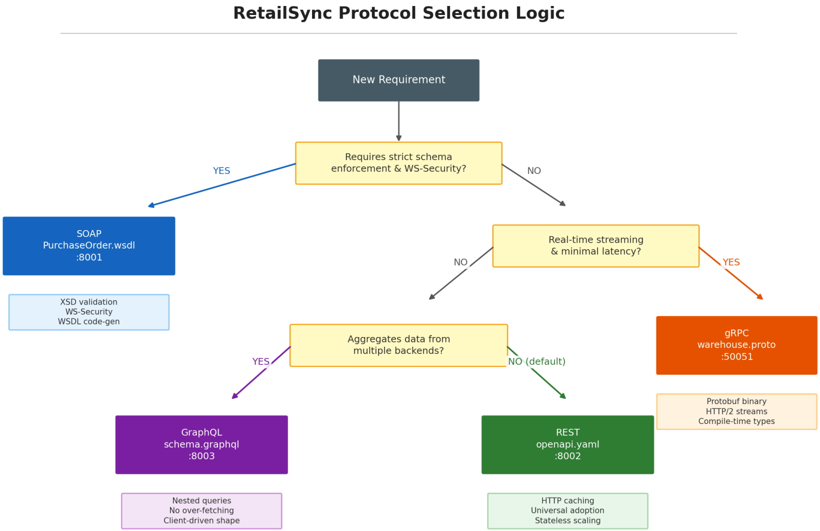
Expected: single JSON response aggregating data from all backends.

Step 4: gRPC – Streaming Telemetry

```
python logistics/mock-server/client.py
```

Expected: unary status response, then bi-directional stream with MOVE, PICK, CHARGE commands and telemetry events.

Protocol Selection Decision Tree



RetailSync Protocol Selection Logic

New Requirement

Requires strict schema enforcement & WS-Security?

SOAP
PurchaseOrder.wsdl
:8001

XSD validation
WS-Security
WSDL code-gen

Real-time streaming & minimal latency?

gRPC
warehouse.proto
:50051

Protobuf binary
HTTP/2 streams
Compile-time types

Aggregates data from multiple backends?

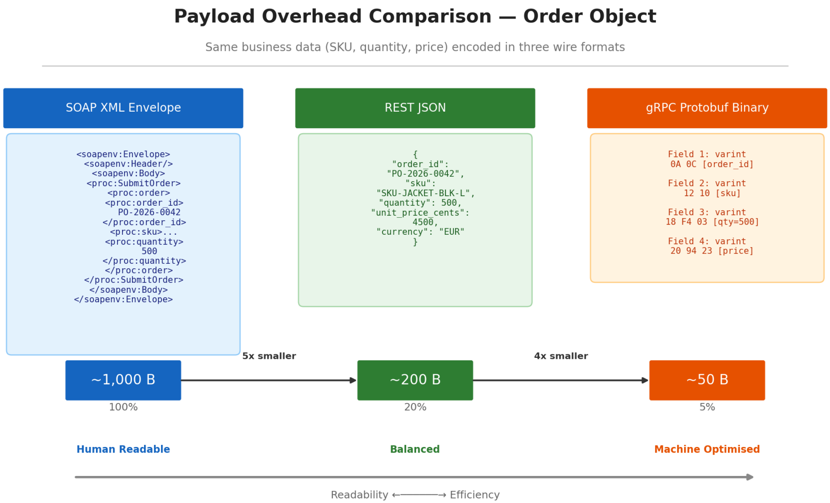
GraphQL
schema.graphql
:8003

Nested queries
No over-fetching
Client-driven shape

REST
openapi.yaml
:8002

HTTP caching
Universal adoption
Stateless scaling

Payload Overhead Comparison



Payload Overhead Comparison — Order Object

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

SOAP XML Envelope

```
<soapenv:Envelope>
  <soapenv:Header/>
    <soapenv:Body>
   <proc:SubmitOrder>
       <proc:order>
      <proc:order_id>
         PO-2026-0042
     </proc:order_id>
       <proc:sku>...
      <proc:quantity>
             500
     </proc:quantity>
      </proc:order>
  </proc:SubmitOrder>
   </soapenv:Body>
</soapenv:Envelope>
```

REST JSON

```
        {
   "order_id":
  "PO-2026-0042",
      "sku":
"SKU-JACKET-BLK-L",
 "quantity": 500,
"unit_price_cents":
       4500,
"currency": "EUR"
        }
```

gRPC Protobuf Binary

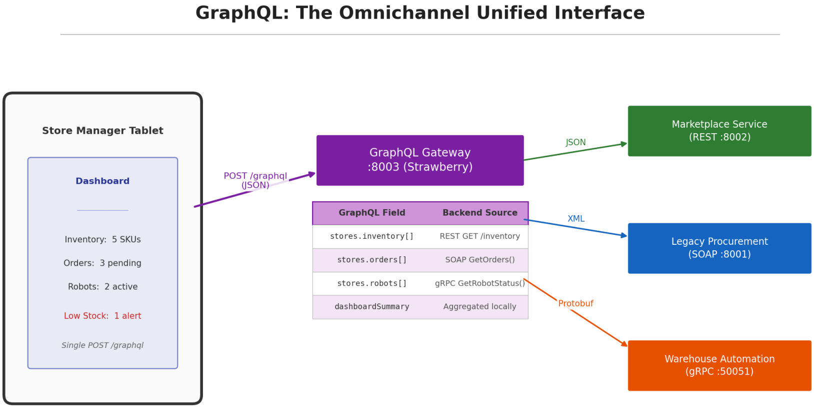
```
 Field 1: varint
 0A 0C [order_id]

 Field 2: varint
   12 10 [sku]

 Field 3: varint
18 F4 03 [qty=500]

 Field 4: varint
 20 94 23 [price]
```

GraphQL Aggregator Pattern



GraphQL: The Omnichannel Unified Interface

GraphQL Field	Backend Source
`stores.inventory[]`	REST GET /inventory
`stores.orders[]`	SOAP GetOrders()
`stores.robots[]`	gRPC GetRobotStatus()
`dashboardSummary`	Aggregated locally

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 × 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`