

Nearform_

Enterprise Integration Patterns in Salsa Cloud

Da Gregor Hohpe a Google Cloud

11 October, DevFest Milano 2025

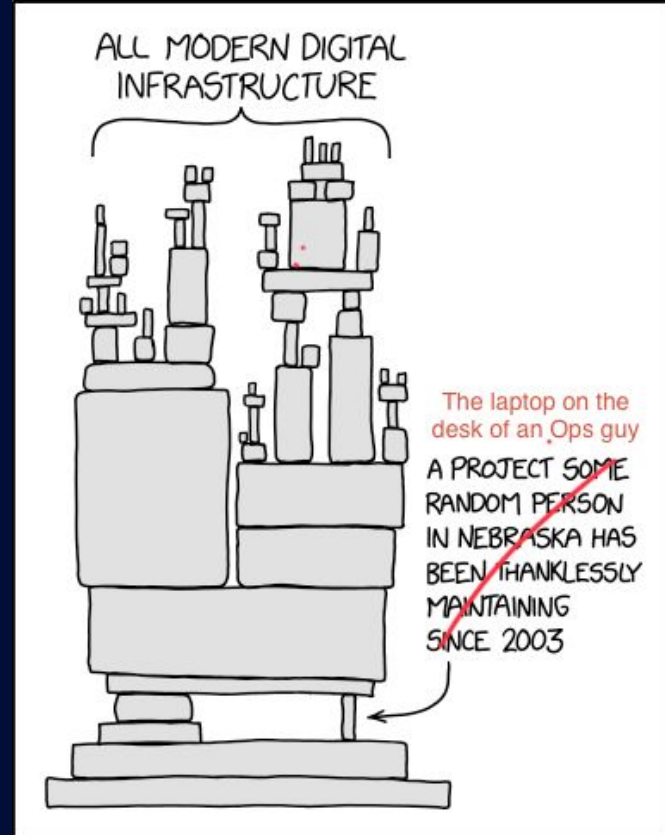
Antonio Perrone

The Problem: The Integration Jungle

The Domino Effect

Inconsistent Data

Spread Business Logic



Adapted from <https://xkcd.com/2347/>

Antonio Perrone

Technical Leader

@Nearform_



@anperrone



The Solution: Let's Use a Map!

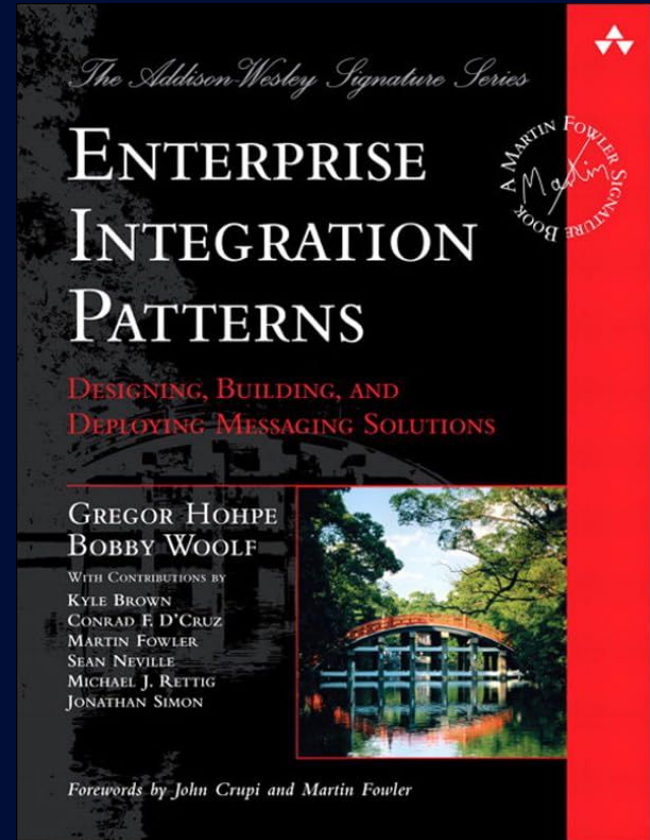
What is a pattern?

It's not a technology, but a proven recipe.

It provides a common vocabulary for architects and developers.

It allows you to design decoupled, resilient, and maintainable solutions.

We don't need to reinvent the wheel, just use the right recipes!



EIP: the Fab4 Pattern

 **Message Channel:** The pipe that connects services

 **Publish-Subscribe:** One message → N recipients

 **Message Router:** if (type=VIP) → express-queue

 **Message Translator:** XML → JSON → Protobuf

Pattern → GCP Service

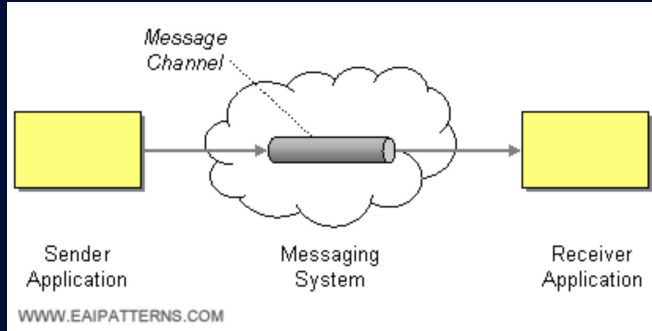
Cloud Pub/Sub: Our messaging system. The "pipe" for our *Message Channels*.

Cloud Functions: Our "on-demand" business logic. Perfect for *Routers* and *Translators*.

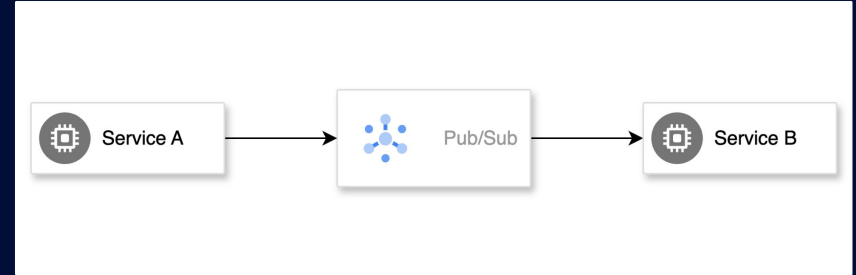
Cloud Run: Our containerized microservices. Useful as publishers or subscribers.

Cloud Workflows: Our orchestrator. To tie together multiple patterns in a complex process.

Pattern 1: Message Channel



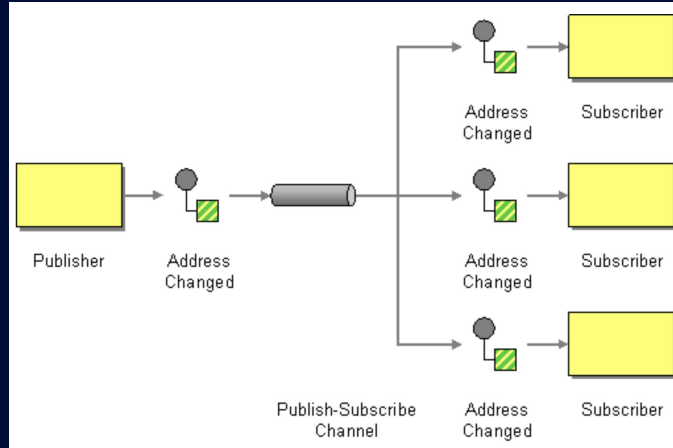
Ref: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageChannel.html>



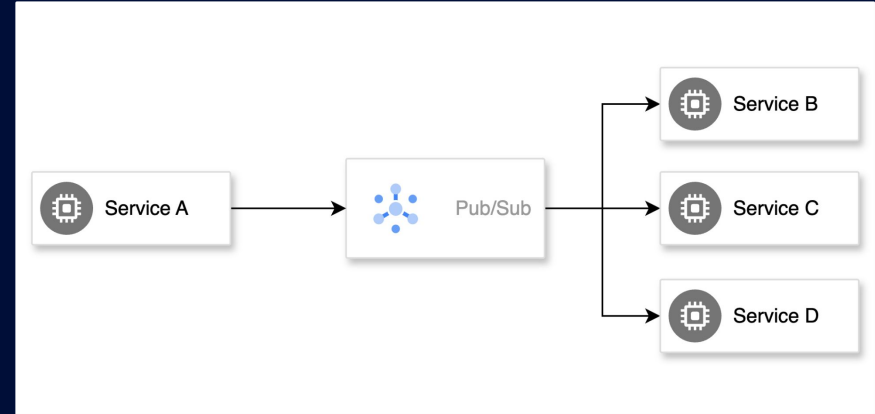
Common Derived Patterns:

- **Point-to-Point Channel:** Only one receiver consumes the message (like a task queue).
- **Dead Letter Channel:** A destination for messages that cannot be processed successfully.

Pattern 2: Publish-Subscribe Channel



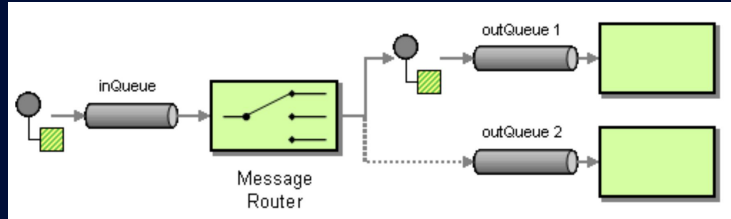
Ref: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>



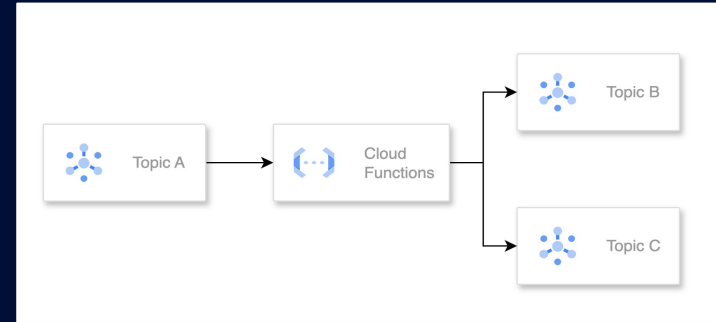
Common Derived Patterns:

- **Durable Subscriber:** Ensures a subscriber receives messages even if it was disconnected when they were published. Crucial for avoiding data loss.
- **Event Message:** The message is an immutable, timestamped notification that something has happened.

Pattern 3: Message Router



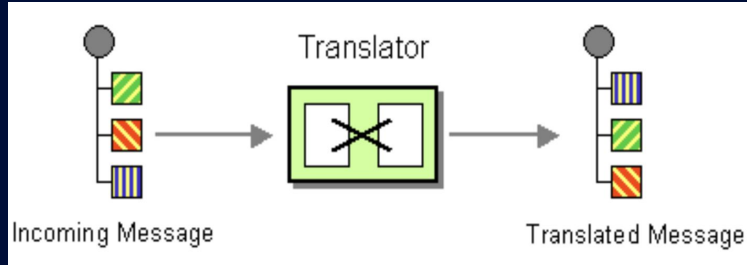
Ref: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageRouter.html>



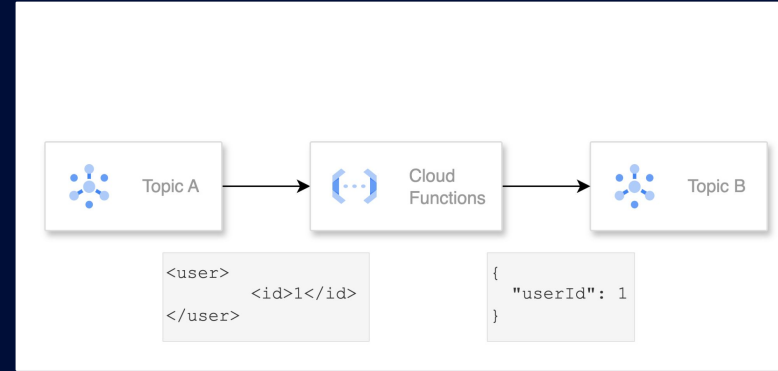
Common Derived Patterns:

- **Content-Based Router:** Routes a message based on its content (e.g., if order.amount > 1000).
- **Message Filter:** A special router that either passes a message or discards it based on a condition.
- **Recipient List:** Routes a single message to a list of dynamically specified recipients.

Pattern 4: Message Translator



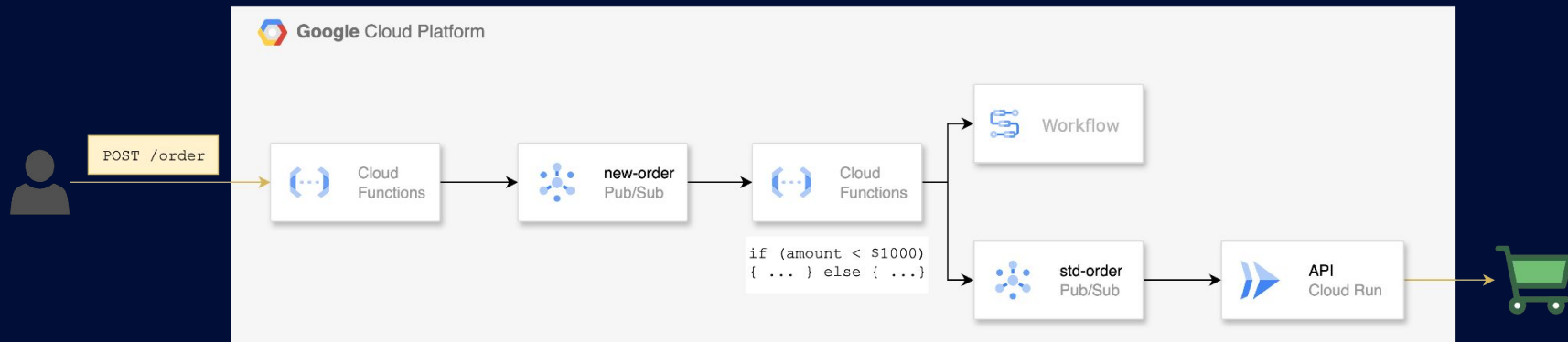
Ref: <https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageTranslator.html>



Common Derived Patterns:

- **Content Enricher:** Adds missing data to a message by retrieving it from an external source.
- **Content Filter:** Removes unnecessary or sensitive data from a message.
- **Claim Check:** Moves a large message payload to external storage and passes along only a reference (a "claim check") to it.

Use Case: Processing an Order



- ① User → API: 200ms ✓
- ② API → Pub/Sub: "order.created"
- ③ Router Function:
 - amount > \$1000 → Workflow
 - else → std-order topic
- ④ Services process async

User response time: 200ms

Resilience: If a service is down, automatic retry

The Quantified Benefits

Zero Lost Orders

If your order manager is down for 2 hours, messages queue up in Pub/Sub. No revenue is lost.



Handles Black Friday Peaks

The system scales from 0 to 1000s of requests/sec automatically, with zero manual intervention.



Fast User Response

The user never waits for slow internal systems. The experience is always fast and fluid.

5 Mistakes We Made (So You Don't Have To)



❌ **Error #1:** "Messages always arrive in order"

Reality Check: Pub/Sub does not guarantee global ordering



❌ **Error #3:** "All errors are temporary"

Reality Check: Some errors won't be fixed by retries

Solution: Classify errors

Transient: Network timeout → Retry

Permanent: Invalid data → DLQ



❌ **Error #2:** "Idempotency isn't needed, Pub/Sub is at-least-once"

Reality Check: At-least-once = possible duplicates!

Consequence: Customer charged 3 times for the same order 😱

Solution: Use an idempotency key on EVERY critical operation



❌ **Error #4:** "Pub/Sub can handle any message"

Reality Check: 10 MB message size limit

Solution: Use the Claim Check Pattern



"In distributed systems, anything that can go wrong will go wrong... simultaneously... on Friday at 5:00 PM"

GCP Pub/Sub vs Apache Kafka vs RabbitMQ

Cloud Pub/Sub (GCP)

When to use:

- 📌 Already on GCP ecosystem
- 📊 Very high volume (billion of msg/day)
- 🌐 Need to scale globally
- 💰 Predictable cost model

Use case: Real-time analytics, IoT telemetry

Apache Kafka

When to use:

- 📝 Event sourcing / log streaming
- 🔄 Needs to event reply
- 📈 Stream processing
- 🏢 On-premise or hybrid cloud

Use case: Event store for CQRS, CDC from database

RabbitMQ

When to use:

- 🔧 Complex Routing (topic exchanges)
- 📧 Messages priority
- 🔒 Multiples protocol (AMQP, MQTT)
- 💻 Deployment on-premise

Use case: Task queue for job processing, enterprise microservice

Key Takeaways

- ✓ Classic EIPs are your map for modern distributed systems.
- ✓ GCP provides the perfect serverless ingredients.
- ✓ **Decoupling isn't a luxury** ; it's the foundation for resilience and speed.
- ✓ Start simple (`Pub/Sub` + `Functions`), then compose for complexity.
- ✓ **Measure the benefits** : less downtime, better UX, more agility.

Thank You! Question?

Repository code



References

- eaipatterns.com
- cloud.google.com/run
- cloud.google.com/pubsub

Contacts

 @anperrone

 @antonioperrone

 <https://antonioperrone.dev>