● JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#54

THE EVENT-DRIVEN MINDSET: PUB/SUB PATTERNS

MOVING FROM "TELLING" TO "REACTING"

Commands vs. Events

Shifting the Perspective: Intent vs. Fact

The transition from a synchronous monolith to a distributed system begins with how you define communication. A Command is an instruction, an Intent directed at a specific target. When Service A sends *CreateInvoice* to Service B, it expects a specific outcome and is often "blocked" until it receives a confirmation.

An Event, however, is a Fact about the past. It is a statement that something has already occurred: *OrderPlaced*. It is immutable and broadcasted without concern for who is listening.

The Power of Facts:
When you publish an event, you decouple the cause from the effect. The "Order Service" simply announces, "An order happened". It doesn't need to know that the "Shipping Service", "Analytics Service", and "Email Service" are all reacting to that information. This shift removes the "Chain of Command" and replaces it with a "Reactionary Flow", allowing your system to grow horizontally without modifying the original source of the event.

The Pub/Sub Topology

The Broadcast Model: Topics and Subscriptions

In a standard queue, a message is consumed once and then vanishes. In a Pub/Sub (Publish/Subscribe) topology, we move to a "One-to-Many" distribution model. A Publisher sends a single message to a Topic (sometimes called an Exchange). The message broker then clones that message for every Subscriber that has expressed interest in that specific topic.

The Architectural Flexibility: Each subscriber maintains its own independent "pointer" or queue. This means the "Email Service" can process messages at 1,000 per second while the "Heavy Analytics Service" chugs along at 10 per second on the exact same data stream. They don't interfere with each other.

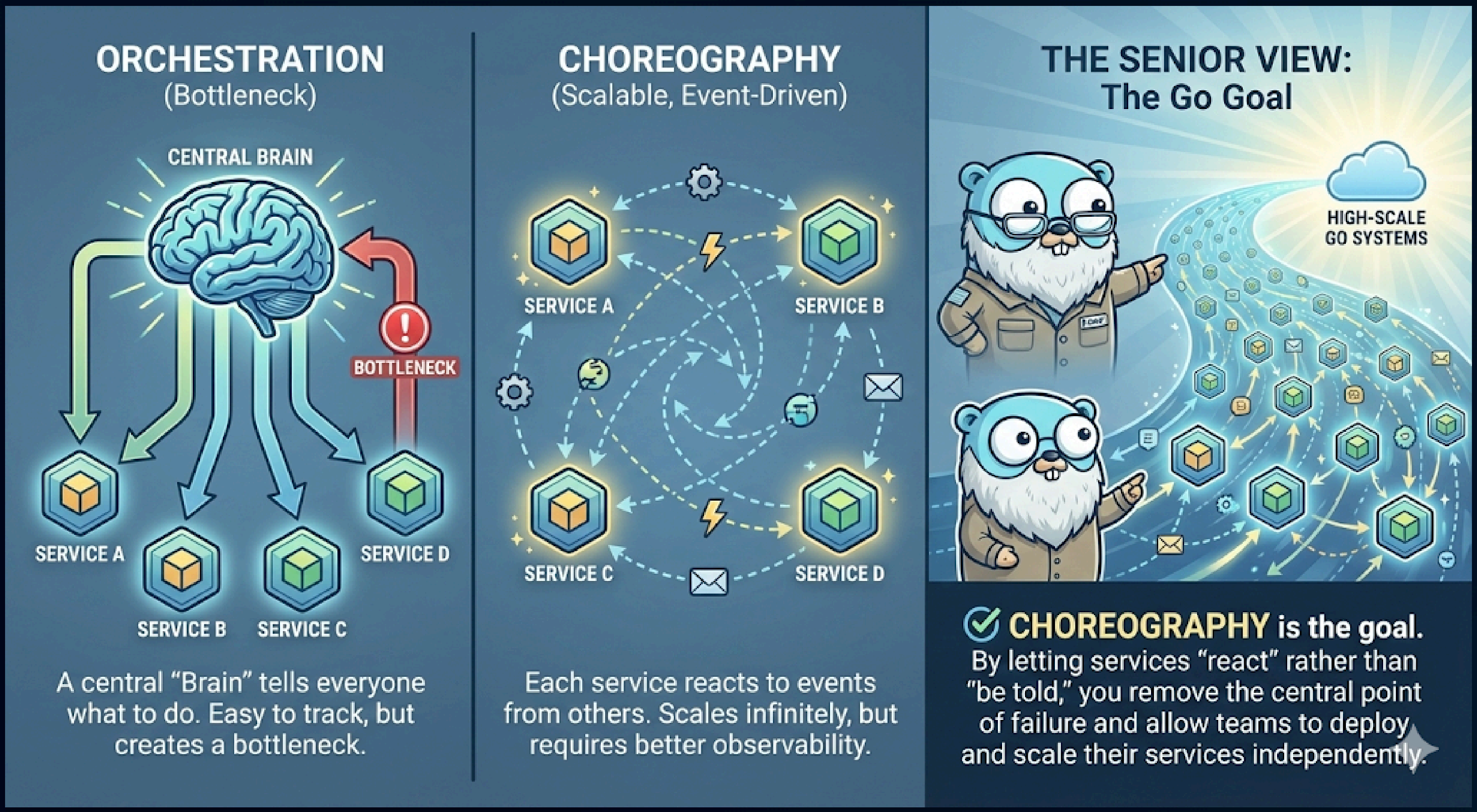
The Real-World Win: Imagine you need to add a new "Fraud Detection" service to your platform. In a traditional system, you'd have to go back into the "Order Service" code and add a new API call. In a Pub/Sub world, you don't touch the "Order Service" at all. You simply deploy the "Fraud Service" and tell it to subscribe to the *OrderPlaced* topic. Your system becomes truly "Open for Extension but Closed for Modification."

Choreography vs. Orchestration

Who Leads the Dance?

- **Orchestration**: A central "Brain" tells everyone what to do (Easy to track, but creates a bottleneck).
- **Choreography**: Each service reacts to events from others (Scales infinitely, but requires better observability).

Choreography is the goal for high-scale Go systems. By letting services "react" rather than "be told", you remove the central point of failure and allow teams to deploy and scale their services independently.

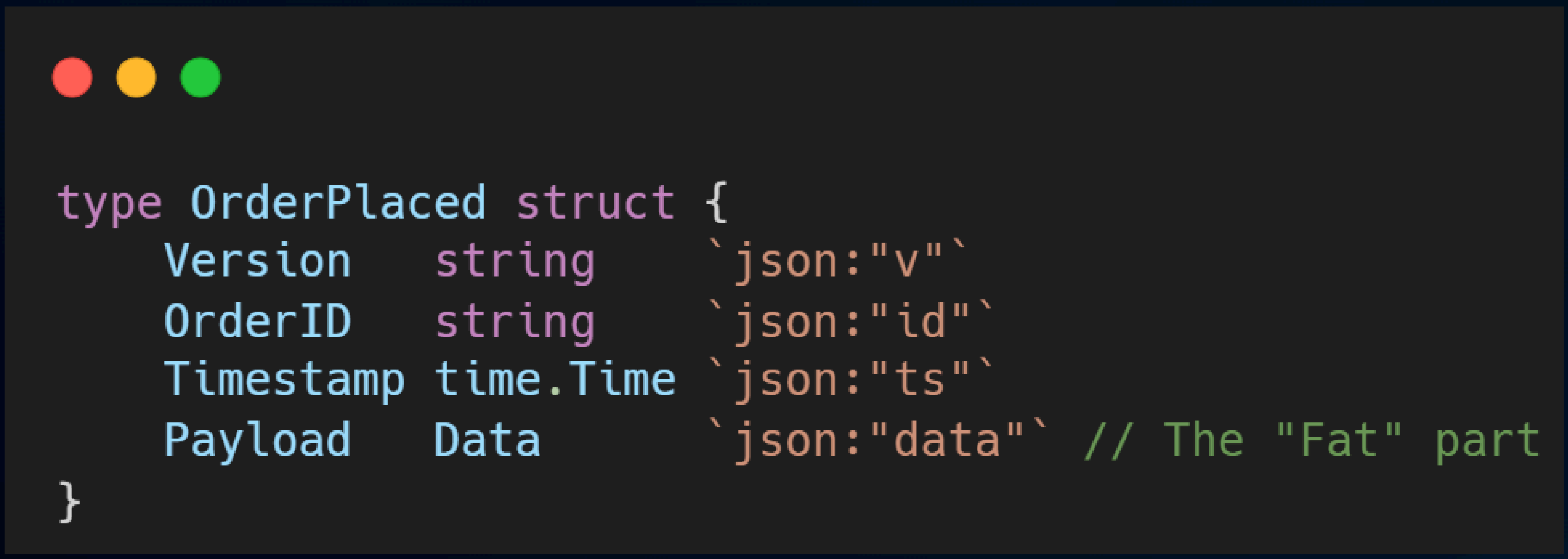

ORCHESTRATION
(Bottleneck)
CENTRAL BRAIN
BOTTLENECK
SERVICE A
SERVICE B
SERVICE C
SERVICE D
A central "Brain" tells everyone what to do. Easy to track, but creates a bottleneck.
CHOREOGRAPHY
(Scalable, Event-Driven)
SERVICE A
SERVICE B
SERVICE C
SERVICE D
Each service reacts to events from others. Scales infinitely, but requires better observability.
THE SENIOR VIEW:
The Go Goal
HIGH-SCALE
GO SYSTEMS
CHOREOGRAPHY is the goal.
By letting services "react" rather than "be told," you remove the central point of failure and allow teams to deploy and scale their services independently.

Designing Event Payloads

Thin vs. Fat Events

- Thin Events: Just an ID (*Order: 123*). The subscriber must call back to the source to get details. It's always up-to-date, but creates network load.
- Fat Events: Includes all data (*Order: 123, Total: $50, Items: […]*) - Faster, but can lead to data stale-ness.

The Strategy:
Start with "Event-Carried State Transfer" (Fat Events) for performance, but ensure your events are versioned. This prevents a change in the "Order" schema from breaking every subscriber in the building.



```go
type OrderPlaced struct {
    Version   string    `json:"v"`
    OrderID   string    `json:"id"`
    Timestamp time.Time `json:"ts"`
    Payload   Data      `json:"data"` // The "Fat" part
}
```

The Distributed Time Problem

Dealing with Out-of-Order Events

In an event-driven world, there is no guarantee that *OrderPlaced* arrives before *PaymentProcessed*. Networks are chaotic.

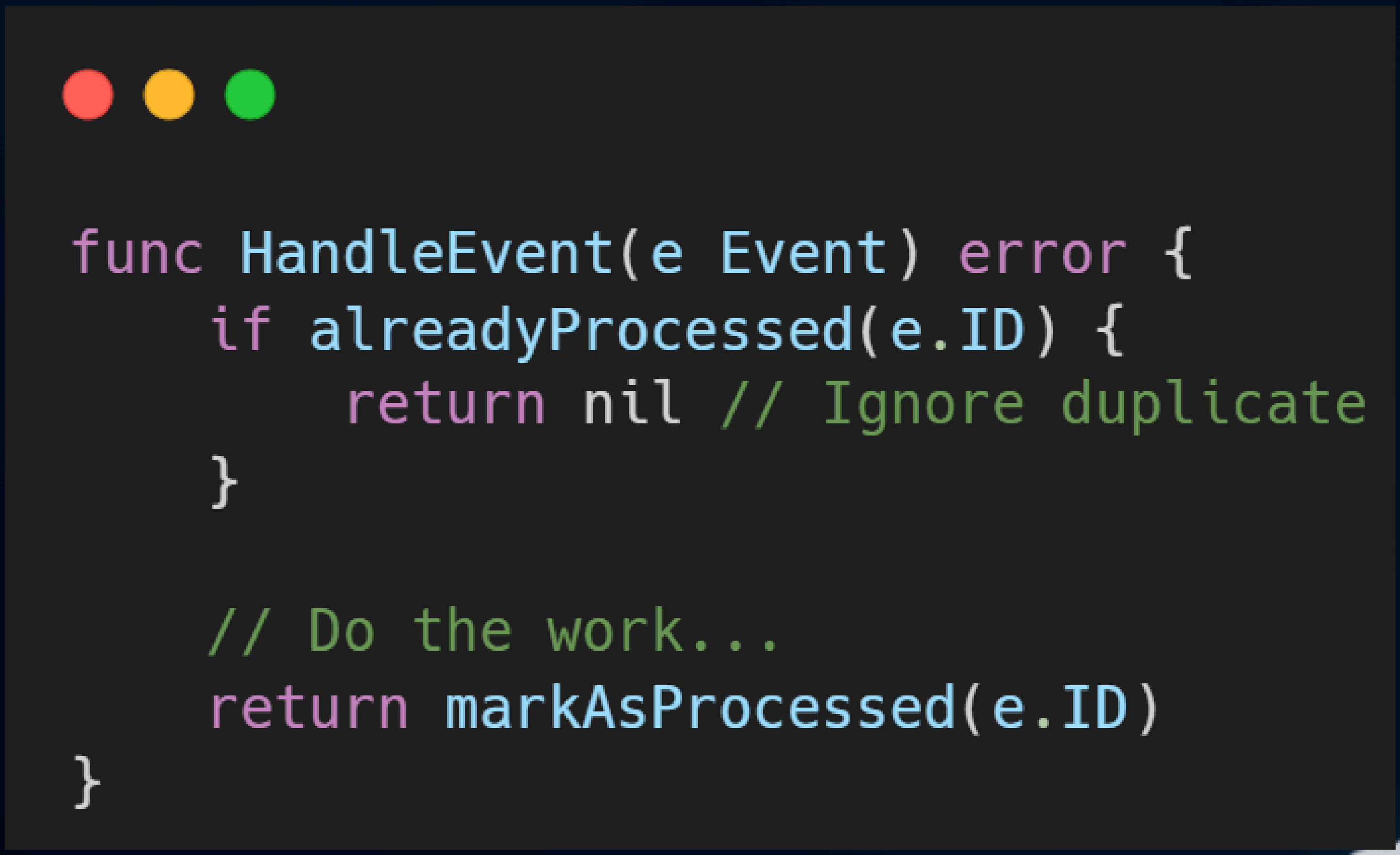
The Fix:
Your services must be smart enough to handle "out-of-order" data. Use Sequence Numbers or Idempotency Keys. If a service receives a *PaymentProcessed* event for an order it hasn't seen yet, it might store that state in a "Pending" table until the main event arrives.

At-Least-Once Processing

Embracing Idempotency

Most event brokers guarantee "at-least-once" delivery. This means your subscriber will eventually get the message, but it might get it twice.

The Strategy:
Every event handler should be Idempotent. Before processing an event, check: "Have I already processed *EventID: ABC?*" If yes, skip it. This simple check is what makes your distributed system bulletproof against network retries.



```go
func HandleEvent(e Event) error {
    if alreadyProcessed(e.ID) {
        return nil // Ignore duplicate
    }

    // Do the work...
    return markAsProcessed(e.ID)
}
```

Eventual Consistency

Living with the Lag

In Pub/Sub, your system is "Eventually Consistent." There is a small window of time where the Order is "Placed" but the "Stock" hasn't been updated yet.

The Reality:
As an experienced engineer, you must design for this. If a user refreshes the page immediately, they might see old data. You solve this with UX (spinners, "Processing..." states) or by using "Read Models" that are updated as part of the event flow.

Summary:

- **Facts over Commands**: Publish what happened, not what should happen.
- **Pub/Sub**: Use topics to broadcast to multiple listeners simultaneously.
- **Choreography**: Let services lead themselves through event reactions.
- **Idempotency**: Prepare for duplicate events; it's a feature, not a bug.

Tomorrow we ground our data back in the physical world: SQL & Postgres with pgx/sqlx.