ChatGPT

Distributed Orchestration Approaches for Per-Source FIFO with Retry

This document outlines the problem, requirements, and architecture patterns to ensure strict per-source FIFO processing, retry-until-success, and multi-region/multi-instance scalability.

Problem Statement

Design a system that:

- Processes operations in FIFO order **per sourceId**.
- **Retries until success** for each message before moving to the next.
- Supports **multi-region deployment**.
- Works across **multiple ECS/Fargate instances**.
- Ensures **only one message per source is processed at a time**.
- Offers **resiliency, observability, and scalability**.

Functional Requirements

- Strict ordering of messages per source.
- Message is processed **only after previous one for the same source is complete**.
- Message should **retry until success**, not go to DLQ.
- Scalable horizontally and regionally.

Non-Functional Requirements

- High availability.
- Fault tolerance.
- Low operational overhead.
- Cloud-native compatibility (AWS services preferred).

Approach 1: DynamoDB Only (Polling and Locking Logic)

Architecture Diagram

```
Client/Event Source
     ↓
DynamoDB Global Table
     ↓
Polling Workers (ECS/Lambda)
```

```
     ↓
Main App with Locking Mechanism
```

How it Works

- Messages are written to a DynamoDB table partitioned by `sourceId`.
- Polling workers continuously scan for unprocessed messages in FIFO order.
- Use conditional writes (e.g., `UpdateItem` with `ConditionExpression`) to acquire a lock on the oldest unprocessed message.
- Retry logic is implemented in code.
- Message is only marked processed if successful.

Advantages

- No need for additional AWS services (e.g., SQS).
- Complete control over retry, ordering, and locking.
- Multi-region support via global tables.

Disadvantages

- Requires precise design of locking, polling, and failure recovery.
- Higher complexity in custom orchestration code.
- Potential read amplification and cost if not optimized.

Best For

- Teams comfortable writing custom orchestration.
- Scenarios with strong FIFO and retry consistency requirements.

Approach 2: DynamoDB + SQS FIFO (Direct Worker Model)

Architecture Diagram

```
Client/Event Source
     ↓
DynamoDB Global Table
     ↓
SQS FIFO Queue (MessageGroupId = sourceId)
     ↓
ECS/Fargate Workers
     ↓
Main App
```

How it Works

- Operations are written to DynamoDB.
- Message pushed to SQS FIFO queue with `MessageGroupId = sourceId`.
- Multiple workers consume messages.
- FIFO maintained per source.
- Retry handled by SQS (visibility timeout).

Advantages

- Strict per-source FIFO via `MessageGroupId`.
- Auto-scaling across ECS instances.
- Retry handled by infrastructure.
- Easy to operate and scale.

Disadvantages

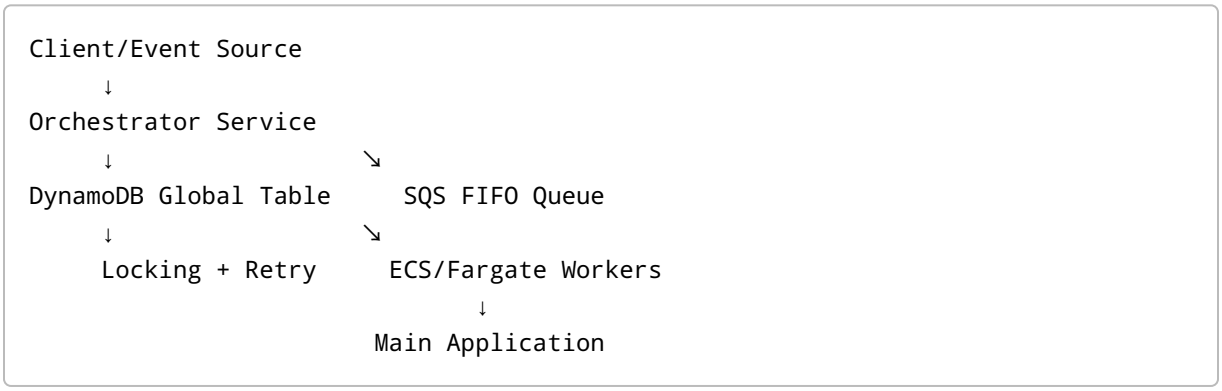
- Message retry behavior is timer-based (not conditional).
- No global routing unless you design it.
- Need to manage stuck messages with watchdog/DLQ.

Best For

- High concurrency, multi-source event pipelines.
- Simpler orchestration, fully infra-driven retry.

Approach 3: Orchestrator Service + Main Application (Separation of Concerns)

Architecture Diagram



How it Works

- Orchestrator accepts writes, pushes them to DynamoDB & SQS.
- Orchestrator polls and controls FIFO + retry logic.

- Main application is stateless, only processes calls.
- Orchestrator enforces: "next message processed only if previous is successful."

Advantages

- Full control of ordering and retry behavior.
- Clean decoupling between business logic and orchestration.
- Easy to extend (e.g., throttling, alerting, metrics).
- Supports regional queues + routing.

Disadvantages

- Requires additional orchestration code and deployments.
- Slightly more operational complexity.

Best For

- Enterprise-grade pipelines.
- Multi-region, multi-tenant architectures.
- Fine-grained control over ordering and retries.

Other Notable Approaches (Short Summary)

Step Functions + DynamoDB Streams

- Use DynamoDB Streams to trigger Step Functions state machines per source.
- State machine enforces FIFO by maintaining execution state.
- Complex for large-scale concurrency and limited flexibility with custom retry.

Kafka with Partitions by SourceId

- Use Kafka with topic partitions keyed by `sourceId` to ensure per-source FIFO.
- Retry can be implemented with a separate retry topic.
- Needs self-managed infra or MSK.

SNS + Lambda FIFO Emulation

- SNS to fan out events; use sourceId hash to route to designated Lambda or SQS FIFO.
- Requires careful mapping and routing logic.
- Works better with moderate scale.

⏳ Aurora + Advisory Locks

- Use a relational DB (like Aurora) with advisory locks for source-level locking.
- Suitable where SQL querying and ordering is a priority.
- Higher latency and complexity at scale.

🏋️ Feature Comparison Table

Feature	DynamoDB Only	SQS FIFO Direct	Orchestrator Service
Strict per-source FIFO	Custom	Native via SQS	Full control
Retry until success	Custom	Via SQS	Custom retry
Multi-region routing	Global Table	with design	Flexible
Horizontal scalability	With effort	Yes	Yes
Orchestration observability	Minimal	Minimal	Full logs/metrics
Dev/Operational simplicity	Complex	Moderate	Complex
Fine-grained ordering control	Custom logic	By group	Full

Recommendations

- **DynamoDB Only**: When full control is needed and you're comfortable building orchestration from scratch.
- **SQS FIFO**: For high-scale FIFO pipelines with infra-managed simplicity.
- **Orchestrator Service**: For enterprise-grade, region-aware, retry-until-success flows with rich observability and control.

Let me know if you'd like this exported to PDF, with diagrams or editable format.