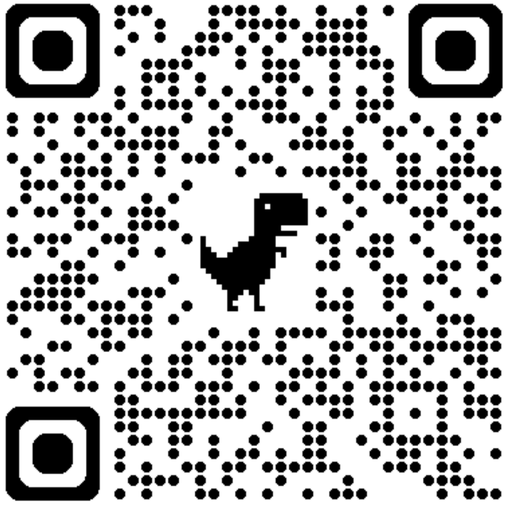
Architectural Patterns Focus: Transactions

Daniel Hinojosa

In this Presentation

- Two-Phase Commit
- The Problem with 2-PC
- Using Event-Driven-Architecture to manage transactions
- Transactional Outbox
- Compensating Transaction
- Optimistic vs Pessimistic Locking
- TCC (Try-Confirm/Cancel)
- Saga - Orchestrator
- Saga - Choreography

Slides and Material:
https://github.com/dhinojosa/nfjs-architectural-patterns-transactions



ACID

What is ACID?

- ACID is a set of properties that guarantee reliable processing of database transactions.
- Ensures consistency, reliability, and correctness in database systems even during failures.

What is ACID? (Continued)

- **Atomic:** Ensures that a transaction is treated as a single, indivisible unit.
  - **Example:** In a bank transfer, either both debit and credit operations occur, or neither does.
- **Consistent:** Ensures the database transitions from one valid state to another, maintaining all predefined rules (e.g., constraints, triggers).
  - **Example:** Transferring money ensures the total balance across accounts remains unchanged.

What is ACID? (Continued)

- **Isolation:** Ensures that concurrent transactions do not interfere with each other.
  - **Example:** Two users booking movie tickets simultaneously won’t end up booking the same seat.
- **Durable:** Ensures that once a transaction is committed, the changes are permanent, even in the event of a crash or power failure.
  - **Example:** After confirming a flight booking, the reservation persists even if the system crashes.

BASE

What is BASE?

- **Basically Available:** The system ensures availability even in the face of failures. Not every operation may succeed immediately, but the system as a whole remains operational.
    - **Example:** A shopping website might continue to serve product details even if some backend services are temporarily down.
- **Soft State:** The state of the system is allowed to be inconsistent for a period of time. Changes to data propagate asynchronously across nodes.
    - **Example:** A social media app might show a slightly outdated profile for a user until all nodes sync.

What is BASE? (Continued)

- **Eventual Consistency:** The system does not guarantee immediate consistency after a write. Over time, all updates will propagate, and the system will converge to a consistent state.
    - **Example:** Distributed caching systems like Redis, Kafka, and Cassandra may have delayed propagation of updates but ensure consistency eventually.

BASE vs. ACID

Aspect	BASE	ACID
Focus	Availability and scalability	Consistency and reliability
Consistency	Eventual consistency	Strong, immediate consistency
State	Soft (can change during transactions)	Hard (remains consistent)
Availability	High (even during partial failures)	May sacrifice availability
Latency	Low latency (asynchronous updates)	Higher latency (due to strict consistency)
Use Case	Distributed, high-scale systems	Transactional, relational systems

Two Phase Commit

Two Phase Commit

- A distributed algorithm to ensure all participants in a transaction commit or roll back together, maintaining consistency.
- Commonly used in distributed systems to handle transactions across multiple databases or services.

What make up the two phases?

Phase One

- **Coordinator** sends a `Prepare` message to all participants.
- **Participants:**
  - Validate if they can commit the transaction.
  - Lock the necessary resources.
  - Respond with Yes (prepared to commit) or No (cannot commit).

Phase Two

- If all participants respond **Yes**:
  - Coordinator sends a Commit message.
  - Participants complete the transaction and release locks.
- If any participant responds **No** or times out:
  - Coordinator sends a `Rollback` message.
  - Participants abort the transaction and release locks.

What is XA?

- A standard protocol for distributed transaction processing defined by the Open Group.
- Facilitates coordination between a Transaction Manager and multiple Resource Managers (e.g., databases, message queues).
- Ensures distributed transactions across multiple resources maintain ACID properties. (Atomic, Consistent, Isolated, Durable)

What is a Transaction Manager?

- Software component or service that coordinates transactions across multiple resources (e.g., databases, message brokers) to ensure data consistency and integrity.

- It provides support for distributed transactions, often using protocols like XA or 2PC (Two-Phase Commit) to manage operations involving multiple participants.

What are some Transaction Managers?

- **JBoss Narayana (formerly JBoss Transaction Service):** A widely used open-source transaction manager with XA support. Integrates with Java EE/Jakarta EE application servers like WildFly and JBoss EAP.
- **Bitronix Transaction Manager (BTM):** Lightweight transaction manager supporting XA for Java applications. Often used in Spring Boot and standalone Java environments.
- **Atomikos:** A commercial and open-source XA transaction manager designed for Java. Known for ease of integration with microservices.
- **Java EE/Jakarta EE Servers:** WildFly, WebLogic, GlassFish, and Payara have built-in XA transaction management.

What Databases support XA/2PC?

- **Oracle databases:** Oracle RAC (Real Application Clusters) integrate tightly with XA
- **Microsoft SQL Server:** Implements Distributed Transaction Coordinator (DTC) for managing 2PC.
- **PostgreSQL:** Supports XA through external transaction managers (e.g., Bitronix, Narayana).
- **MySQL:** XA support in the InnoDB storage engine for distributed transactions.
- **IBM Db2:** Integrates XA and 2PC with enterprise transaction processing systems.

What Message Queues support XA/2PC?

- **Apache ActiveMQ:** Full XA support for distributed transactions with JMS (Java Message Service). Integrates with XA-compliant transaction managers like Bitronix or JBoss Narayana.
- **RabbitMQ:** Supports 2PC in distributed transactions using plugins or third-party libraries.
- **IBM MQ:** Enterprise-grade message broker with robust XA support.

2PC - Setup

```
UserTransactionManager transactionManager = new UserTransactionManager();
transactionManager.init();
UserTransaction userTransaction = transactionManager.getUserTransaction();

// Configure Database DataSource
AtomikosDataSourceBean dataSource = new AtomikosDataSourceBean();
dataSource.setUniqueResourceName("DB");
dataSource.setXaDataSourceClassName("com.mysql.cj.jdbc.MysqlXADataSource");
dataSource.setXaProperties(getDatabaseProperties());
dataSource.setPoolSize(5);

// Configure ActiveMQ Connection Factory
ActiveMQXAConnectionFactory xaConnectionFactory = new ActiveMQXAConnectionFactory("tcp://localhost:61616");

// Begin Distributed Transaction
userTransaction.begin();
```

2PC - Execution

```
 try (Connection dbConnection = dataSource.getConnection();
      XAConnection xaConnection = xaConnectionFactory.createXAConnection()) {
      // Step 1: Execute Database Transaction
      executeDatabaseOperation(dbConnection);
      // Step 2: Publish Message to Queue
      executeMessagingOperation(xaConnection);
      // Commit 2PC
      userTransaction.commit();
      System.out.println("Transaction Committed Successfully!");
} catch (Exception e) {
       // Rollback on failure
       userTransaction.rollback();
       System.err.println("Transaction Rolled Back: " + e.getMessage());
} finally {
     transactionManager.close();
}
```

The Problem with 2PC

What's the problem with Two Phase Commit?

- **Multiple Network Round-Trips:** Requires two phases (prepare and commit/rollback), which increase the number of network calls and add latency.
- **Resource Locking:** Each resource locks data during the prepare phase, further slowing down performance. In a system with high contention, locking resources across multiple nodes increases the chance of deadlocks.
- **Coordinator as a Single Point of Failure:** If it crashes or becomes unavailable during a transaction, all participants remain in an uncertain state.
- **Tight Coupling:** 2PC creates tight coupling between services and resources, reducing flexibility and independence. Many modern systems rely on asynchronous communication (e.g., message queues, event-driven architectures) that do not align with 2PC's synchronous nature.

Idempotent Operations

Idempotency

- Operations that can be performed multiple times without changing the result beyond the initial application and is deterministic.
- Reasons for its importance are:
  - **Retries** - Failures in network communication, timeouts, or server errors may require retrying requests.
  - **Duplicates** - Distributed systems may inadvertently send duplicate requests (e.g., during a failover or deduplication delays).
  - **Asynchronous Processing** - Event-driven architectures often process events out of order or multiple times.
- Idempotency is a cornerstone of patterns like TCC, Event Sourcing, and the Transactional Outbox Pattern.

Idempotency in Payment Processing

- Deducting a specific amount is not idempotent (e.g., retrying could result in multiple deductions).
- Idempotency is achieved by ensuring the same payment ID is not processed twice.

Idempotency in Inventory Management

- Reserving inventory might be idempotent if the reservation for the same item and order ID is created only once.

Idempotency in Database Operations

- Updating a record to a specific value `UPDATE inventory SET stock = 10 WHERE product_id = 123` is idempotent.
- Incrementing a value `UPDATE inventory SET stock = stock + 1` is not idempotent.

Event Driven Architecuture To Manage Transactions

What is Event Driven Architecture?

- Architectural design where the flow of the application is determined by events—changes in state or occurrences triggered by users, systems, or external stimuli.
- Captures Events within your domain or business:
  - A significant change in the system's state, such as "order placed," "payment processed," or "user logged in." Typically past tense.
  - Types of Events:
    - State Transfer Events: Include a snapshot of data (e.g., an order with details).
    - Notification Events: Signal that something happened (e.g., "payment successful").

What does Event Driven Architecture have to do with Transactions?

- **Event-Driven Architecture (EDA)** offers a design that avoids traditional distributed transactions (like Two-Phase Commit) while still addressing the need for consistency across distributed systems.
- Distributed transactions in EDA are often handled asynchronously through events rather than tightly coupled, synchronous protocols like XA/2PC.
- Services are loosely coupled and can scale independently without blocking or resource locking.
- Failures in one service do not affect others; events can be replayed for recovery.

Technologies for Event Driven Architecture

Apache Flink

Transactional Outbox

Transactional Outbox

- Design pattern used in distributed systems to ensure reliable, consistent event-driven communication between services without relying on traditional distributed transactions like Two-Phase Commit (2PC).

- **It solves the dual-write problem**, where changes to a database and an event broker (or message queue) need to be performed atomically. By using an outbox, this pattern ensures that:

    1. Database state changes and event publishing are part of a single atomic transaction.

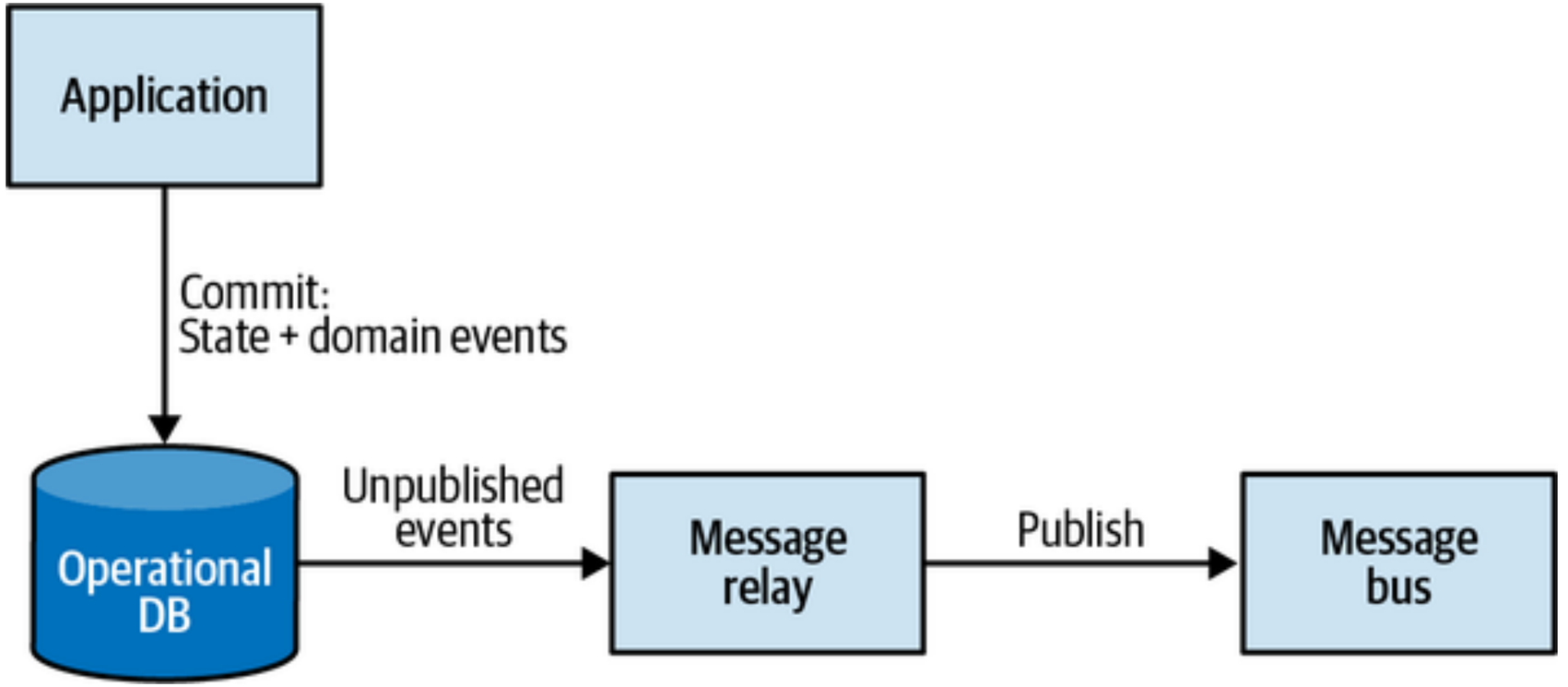
    2. Events are published reliably *after the database transaction is committed*.

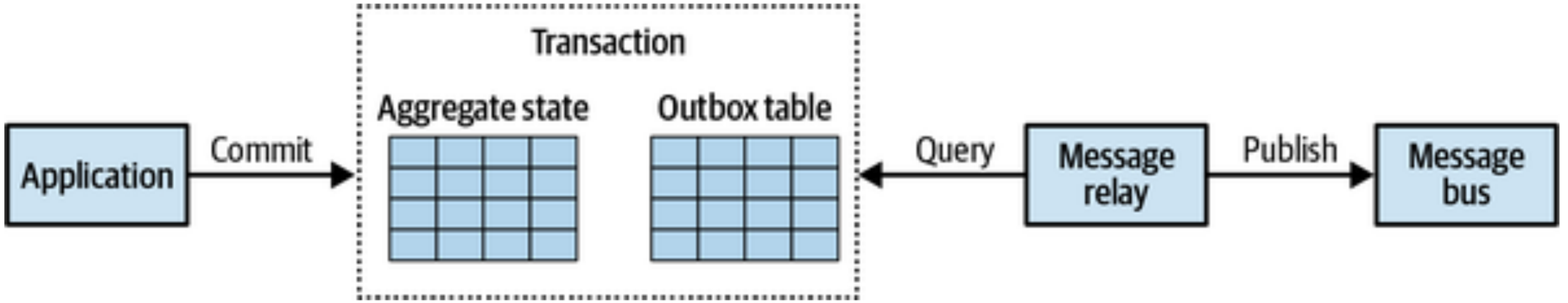
Transactional Outbox Ingredients

- **Primary Business Table**:
  - This table contains the core data relevant to the business logic of the microservice (e.g., orders, payments, inventory).
  - This is the table that the microservice directly interacts with for its operations.
- **Outbox Table:**
  - Dedicated table in the same database where events related to changes in the primary table are stored.
  - This table acts as a staging area for events that need to be published to a message broker or event system.

Transactional Outbox Ingredients (Continued)

- **Event Publisher**
  - A separate process or component that reads the outbox table and publishes the events to the message broker.
  - Can updates the outbox table to mark events as “published” after successful delivery or use an offset table.
- **Message Broker:**
  - Receives events from the Event Publisher and distributes them to the appropriate consumers.


Application
Commit:
State + domain events
Operational
DB
Unpublished
events
Message
relay
Publish
Message
bus



Learning Domain Driven Design - Vlad Khononov Published by O'Reilly Media, Inc.

Demo: Transactional Outbox

- Let’s show a transactional outbox using Kafka Connect

Technologies for Transactional Outbox

Compensating Transactions

Compensating Transactions

- Compensating Transactions are operations used to undo or counteract the effects of a previously completed transaction when a distributed workflow or process fails.
- “For every action, there is an opposing transaction”
- They are a key component of eventual consistency strategies in distributed systems, especially in patterns like the *Saga Pattern*

Compensating Transactions

1. Perform an Action:
    - A transaction is executed by a service or component.
    - **Example:** A payment service charges a customer's credit card.
2. Detect a Failure:
    - If a downstream operation fails (e.g., inventory cannot be reserved), the system must undo the previously completed transactions.
3. Trigger a Compensating Transaction:
    - The system executes an operation to reverse or negate the effects of the previous transaction.
    - **Example:** The payment service issues a refund to the customer.

Compensating Transactions vs 2PC Rollback

Aspect	Compensating Transactions	Rollback (2PC)
Scope	Application-level (logical undo)	Database-level (physical undo)
Coupling	Loosely coupled services	Tightly coupled resources
Latency	Eventual consistency (delayed undo)	Immediate rollback
Scalability	High (no locks, asynchronous)	Limited by locking and coordination
Best For	Distributed systems, microservices	Single database, tightly coupled systems

Optimistic and Pessimistic Locking

Pessimistic Locking

- Concurrency control mechanism which a database locks a resource (e.g., a row or record) to prevent other transactions from modifying it until the lock is released.

- Assumes that conflicts are likely and prevents them by locking resources upfront, ensuring serializability and avoiding data inconsistencies.

How Pessimistic Locking Works

- Acquire Lock:
  - When a transaction reads or modifies a record, it locks the record to ensure no other transactions can modify it until the lock is released.
- Lock types:
  - Shared Lock (Read Lock): Allows multiple transactions to read the resource but prevents any updates.
  - Exclusive Lock (Write Lock): Blocks all other transactions from reading or writing the resource.
- Perform Operations:
  - The transaction performs its operations on the locked resource.
- Release Lock:
  - The lock is released when the transaction completes (either commits or rolls back).

Optimistic Locking

- Concurrency control mechanism used in database systems to handle concurrent updates without locking rows or resources.
- It assumes that conflicts are rare and allows multiple transactions to proceed without locking data upfront.
- Conflicts are detected at the time of the update, and the transaction is retried if a conflict occurs.

How Optimistic Locking Works?

- A transaction reads a record from the database, including a version number or timestamp that tracks changes to the data.
- The transaction modifies the data locally, without locking the record in the database.
- When the transaction attempts to save changes, it checks the current version or timestamp in the database:
  - If the version matches: The transaction is successful, and the record is updated with a new version or timestamp.
  - If the version does not match: The record has been modified by another transaction, and the update fails.
- If a conflict occurs, the application can reload the record, reapply changes, and retry the update.

Optimistic Locking vs Pessimistic Locking

Aspect	Optimistic Locking	Pessimistic Locking
Locking	No locks; relies on conflict detection	Locks rows/resources during transaction
Performance	High concurrency, better performance	Reduced concurrency, higher overhead
Conflict Handling	Detected at write time	Avoided during transaction
Best for	Low-contention scenarios	High-contention scenarios
Deadlock Risk	None	Possible

TCC (Try/Confirm/Cancel)

What is Try-Confirm-Cancel (TCC)?

- **TCC (Try-Confirm-Cancel)** is a distributed transaction management pattern designed to ensure consistency and reliability without relying on centralized protocols like **Two-Phase Commit (2PC)**.

- It divides a transaction into three distinct phases:

  1. **Try:** Reserve resources or validate preconditions without committing the actual operation.

  2. **Confirm:** Commit the reserved resources if all operations succeed.

  3. **Cancel:** Rollback or release the reserved resources if any operation fails.

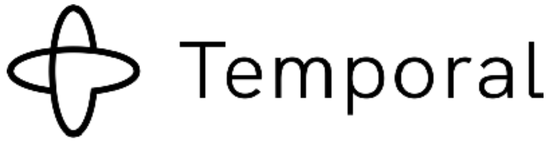
It doesn’t look too different from 2PC

- **TCC (Try-Confirm-Cancel):**
  - Coordination is decentralized and application-driven.
  - Each participant (service/resource) implements its own Try, Confirm, and Cancel logic.
  - The application or transaction manager drives the workflow by invoking these operations.
- **2PC:**
  - Coordination is centralized, driven by a Transaction Coordinator.
  - The coordinator enforces the transaction lifecycle across all participants (prepare, commit/rollback phases).
  - Participants do not implement custom logic but rely on the coordinator to maintain consistency.

TCC vs 2PC

Aspect	TCC Transaction Manager	2PC Transaction Coordinator
Scope	Application-level orchestration.	Database or system-level orchestration.
Customization	Allows participants to define their own Try, Confirm, and Cancel logic.	Participants rely on the coordinator for commit/rollback handling.
Communication	Explicit method calls (e.g., Try, Confirm, Cancel) handled by the application.	Protocol-based communication (e.g., prepare, commit, rollback).
Resource Locking	No implicit locks; participants explicitly reserve resources during Try.	Implicit locks are placed on resources during the prepare phase.
Fault Tolerance	Relies on retries and compensating transactions to handle failures.	Relies on transaction logs and recovery mechanisms.
Flexibility	Highly flexible; participants implement custom actions for Confirm and Cancel phases.	Rigid; all participants must adhere to the 2PC protocol.
Granularity	Fine-grained and designed for business logic and workflows.	Coarse-grained, designed for atomic database operations.
Suitability	Ideal for distributed systems and microservices.	Best for tightly coupled, homogeneous environments.

Technologies for TCC



Saga

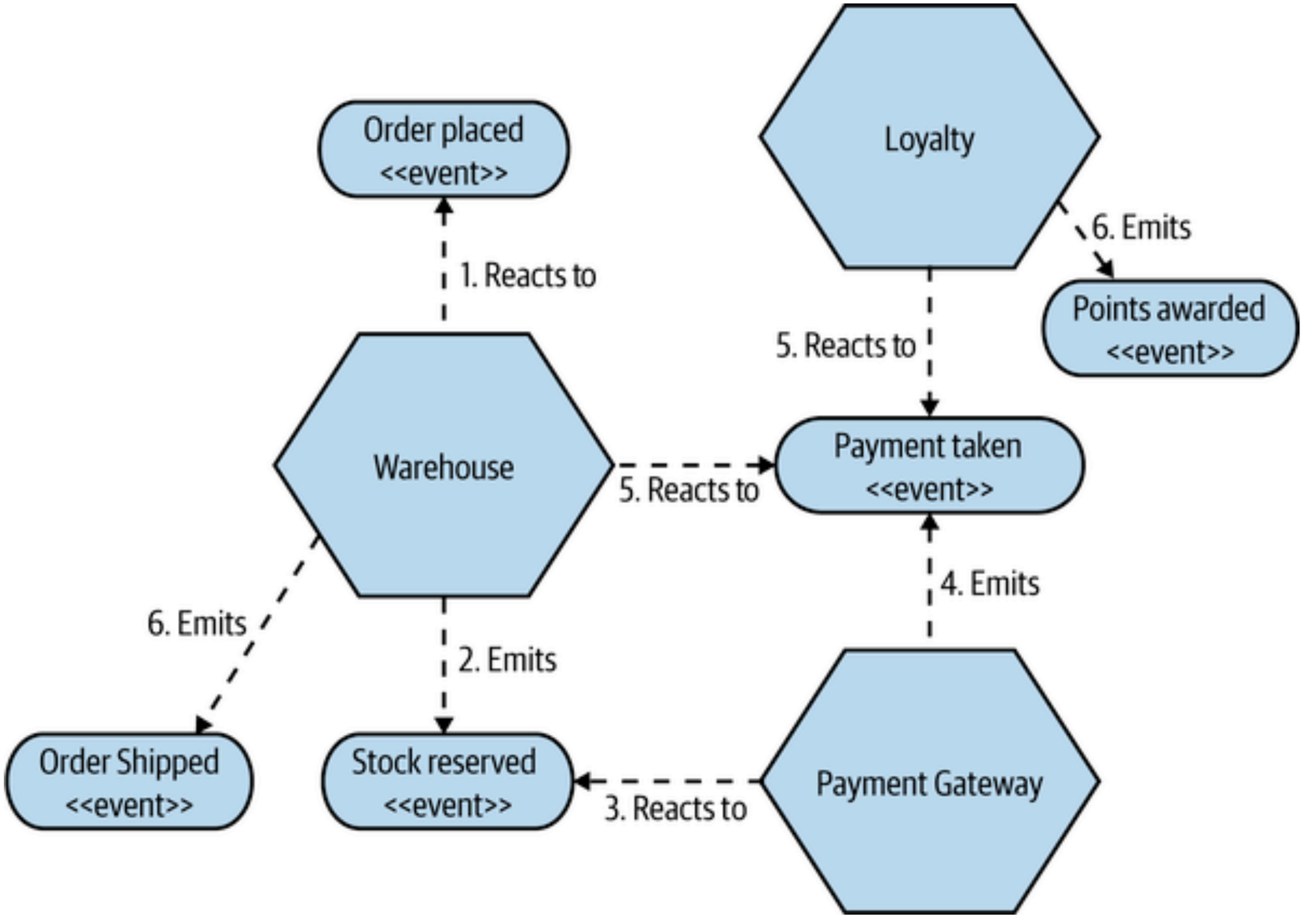
Saga

- **A saga** is a long-running business process.
- It’s long-running not necessarily in terms of time, as sagas can run from seconds to years, but rather in terms of transactions: a business process that spans multiple transactions
- Transactions can be handled not only by aggregates but by any component emitting domain events and responding to commands
- Saga listens to the events emitted by the relevant components and issues subsequent commands to the other components. If one of the execution steps fails, the saga is in charge of issuing relevant compensating actions to ensure the system state remains consistent

Saga: Choreography

Choreography Saga Pattern

- In the **Choreography Saga Pattern**, event-driven communication is used to coordinate distributed transactions.
- Each service involved in the Saga listens to and reacts to events without a central coordinator.
- Services publish events when they complete their steps, triggering subsequent actions in other services.



Building Microservices- Sam Newman Published by O'Reilly Media, Inc.

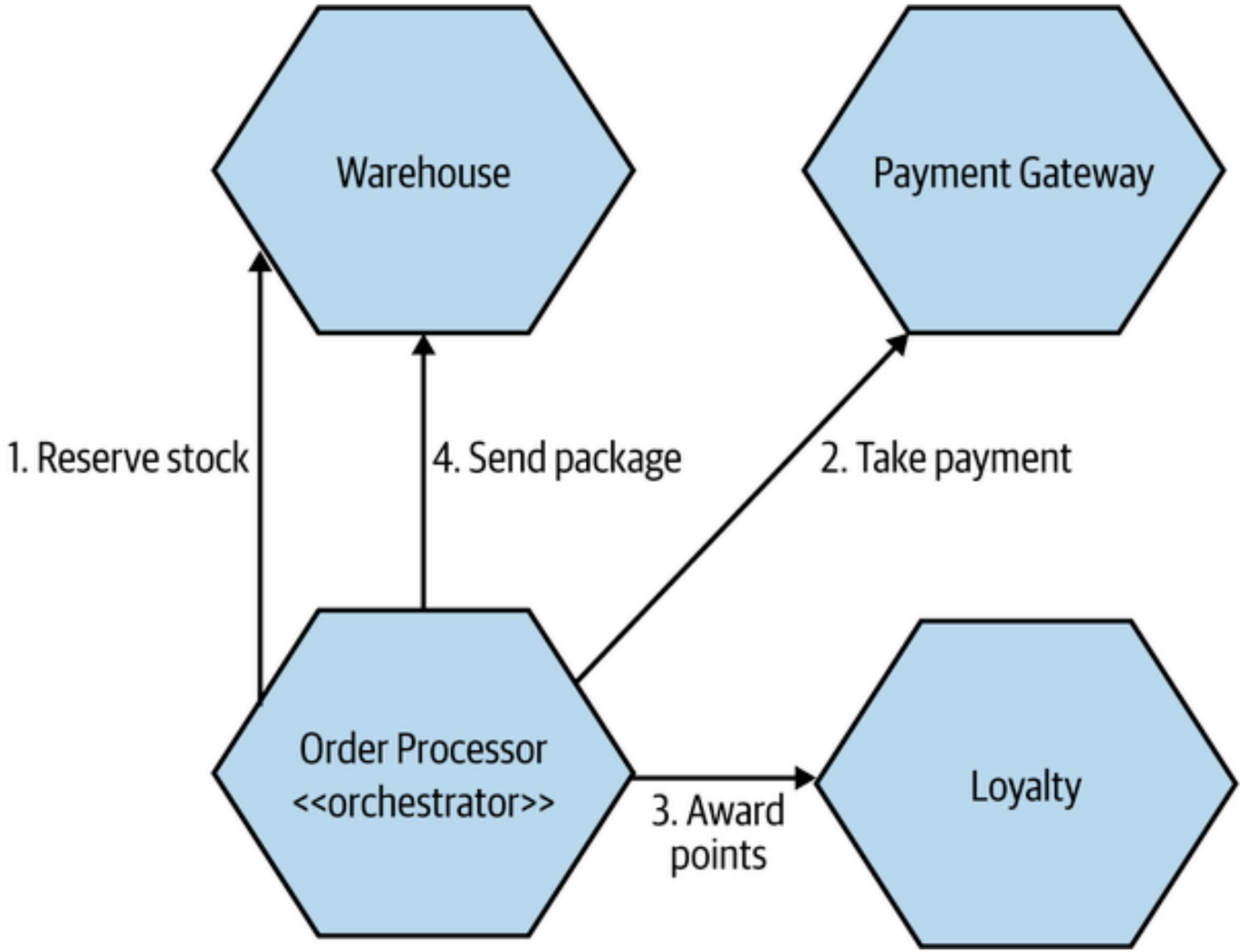
Saga: Orchestrator

Orchestrator Saga Pattern

- In the Orchestrator Saga Pattern, a central coordinator (orchestrator) directs the execution of distributed transactions.
  - The orchestrator determines the sequence of actions for each participating service.
  - Services perform tasks only when instructed by the orchestrator and report back upon completion.
  - If a step fails, the orchestrator triggers compensating actions to undo previous changes, ensuring consistency.

How it works

1. The orchestrator initiates the saga by sending a command to the first service.
2. Each service executes its transaction and reports success or failure back to the orchestrator.
3. Upon success, the orchestrator advances to the next step, instructing the next service.
4. If a failure occurs, the orchestrator initiates compensating actions in reverse order of the saga’s execution.

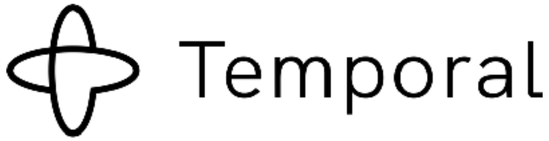


Building Microservices- Sam Newman Published by O'Reilly Media, Inc.

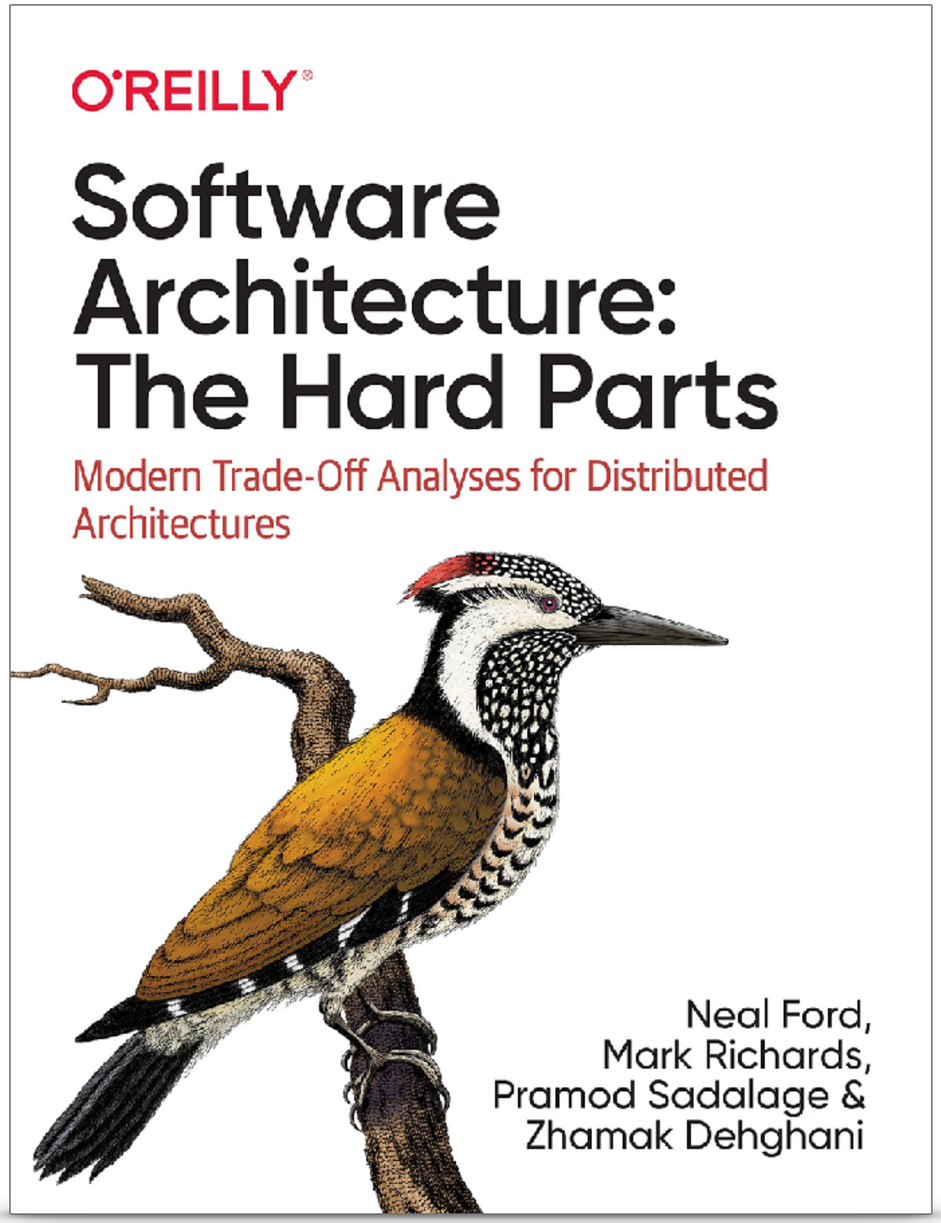
Choreography vs. Orchestration

Aspect	Choreography	Orchestration
Coordination	Decentralized via events	Centralized by an orchestrator
Control	Services independently decide their actions	Orchestrator controls the workflow
Scalability	High (services operate independently)	Moderate (orchestrator can become a bottleneck)
Fault Tolerance	Resilient (no central point of failure)	Orchestrator must handle failures
Complexity	Event management and dependencies can grow	Centralized logic simplifies dependencies
Use Case	Systems with loosely coupled services	Systems requiring strict workflow control

Technologies for Saga Patterns



Ranking the Saga Patterns



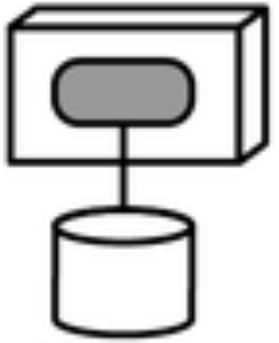
Software Architecture: The Hard Parts

Neal Ford, Mark Richards, Pramod Sadalage, Zhamak Dehghani

Terminology


Transaction
boundary


Domain
Service


Mediator
Service


Error condition
causing rollback


Async one-way
communication
Sync one-way
communication
Sync two-way
communication
Sync compensation
request
Async
compensation

Communication

- **Synchronous Communication**
  - Immediate, blocking call where the sender waits for a response.
  - Used for critical steps requiring immediate feedback (e.g., Payment Confirmation).
  - Simplifies coordination but introduces latency and tightly couples services.
  - Failure cascades quickly across the workflow.
- **Asynchronous Communication**
  - Non-blocking, fire-and-forget message exchange (e.g., via message queues).
  - Ideal for loosely coupled, distributed systems.
  - Improves resilience and scalability, as services operate independently.
  - Requires event-driven design and careful handling of retries and idempotency.
- **Key Consideration:** Use synchronous for time-critical interactions; asynchronous for long-running or distributed tasks.

Consistency

- **Atomic Consistency**
  - All-or-nothing guarantee: Either all steps succeed, or none are applied.
  - Achieved via compensating transactions in Sagas.
  - Sacrificed in distributed systems due to coordination challenges (e.g., network latency, failures).
- **Eventual Consistency**
  - System converges to a consistent state over time.
  - Steps may succeed or fail independently, with corrections via compensations.
  - Prioritizes availability and performance over strict transactional guarantees.
- **Tradeoff:**
  - **Atomic:** Strong consistency, but slower and complex in distributed systems.
  - **Eventual:** Resilient and scalable, aligning with the nature of Sagas.

Coupling

- **Tight Coupling**
  - Direct dependencies between services or an orchestrator.
  - Common in Orchestrator Sagas, where services rely on centralized coordination.
  - Easier to manage workflows but limits scalability and service independence.
- **Loose Coupling**
  - Services interact indirectly through events or messages.
  - Core of Choreography Sagas, enabling independent service execution.
  - Promotes scalability and flexibility but increases complexity in coordination.
- **Key Consideration:**
  - Choose tight coupling for simpler workflows and explicit control.
  - Use loose coupling for resilience and distributed scalability.

Responsiveness/Availability

- **Responsiveness**
  - How quickly the system completes a user-requested operation.
  - **Synchronous Steps:** Impact responsiveness due to blocking calls.
  - **Asynchronous Steps:** Improve responsiveness by decoupling services but may delay full completion.
  - **Tradeoff:** Optimizing responsiveness may increase complexity (e.g., managing partial failures).
- **Availability**
  - System’s ability to remain operational and serve requests.
  - **Choreography Sagas:** Higher availability as services operate independently.
  - **Orchestrator Sagas:** Lower availability risk due to reliance on a central controller.
- **Key Balance:**
  - Prioritize responsiveness for user-facing actions.
  - Focus on availability for backend workflows and fault tolerance.

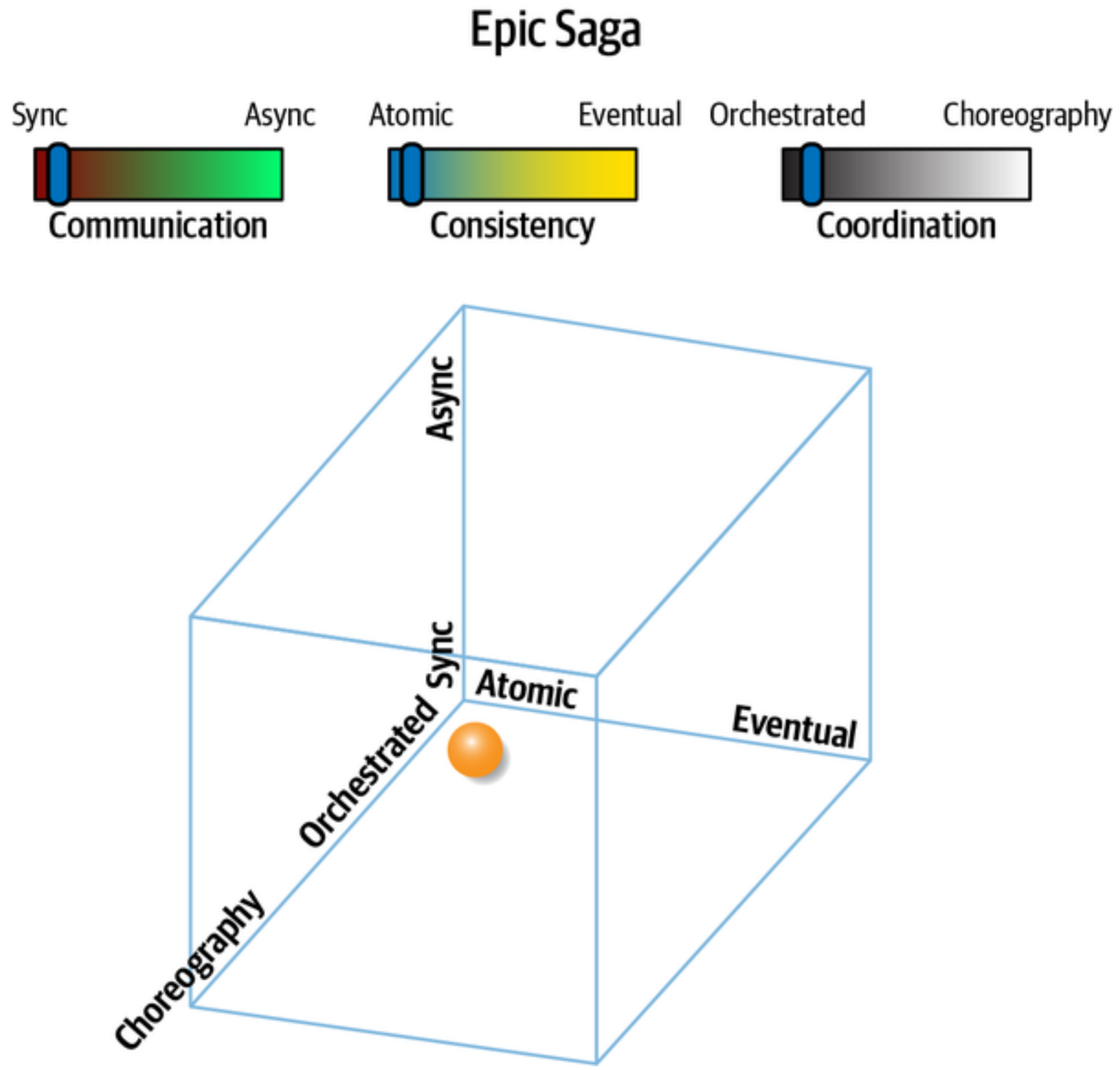
Scale/Elasticity

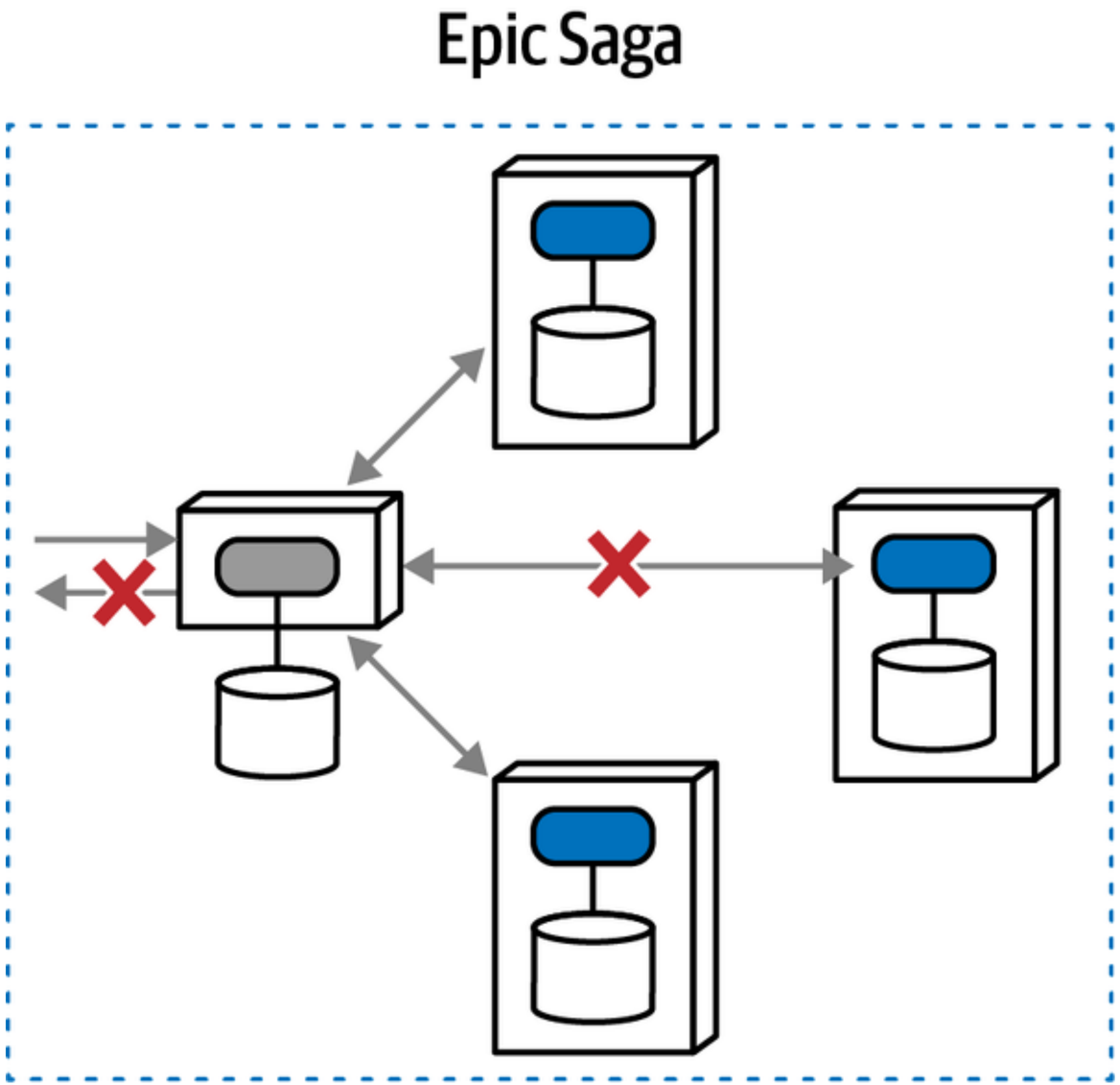
- **Scale**
  - The system’s capacity to handle increasing workloads.
  - **Orchestrator Sagas:** Limited by the orchestrator as a potential bottleneck.
  - Requires horizontal scaling of the orchestrator to handle high transaction volumes.
  - **Choreography Sagas:** Naturally scalable due to independent service execution.
  - More services can be added without central coordination constraints.
- **Elasticity**
  - The system’s ability to dynamically adjust resources to match demand.
  - **Choreography Sagas:** Better suited for elastic environments, as services scale independently.
  - **Orchestrator Sagas:** Elasticity is constrained by the central orchestrator and service coordination.

Comparing the Sagas

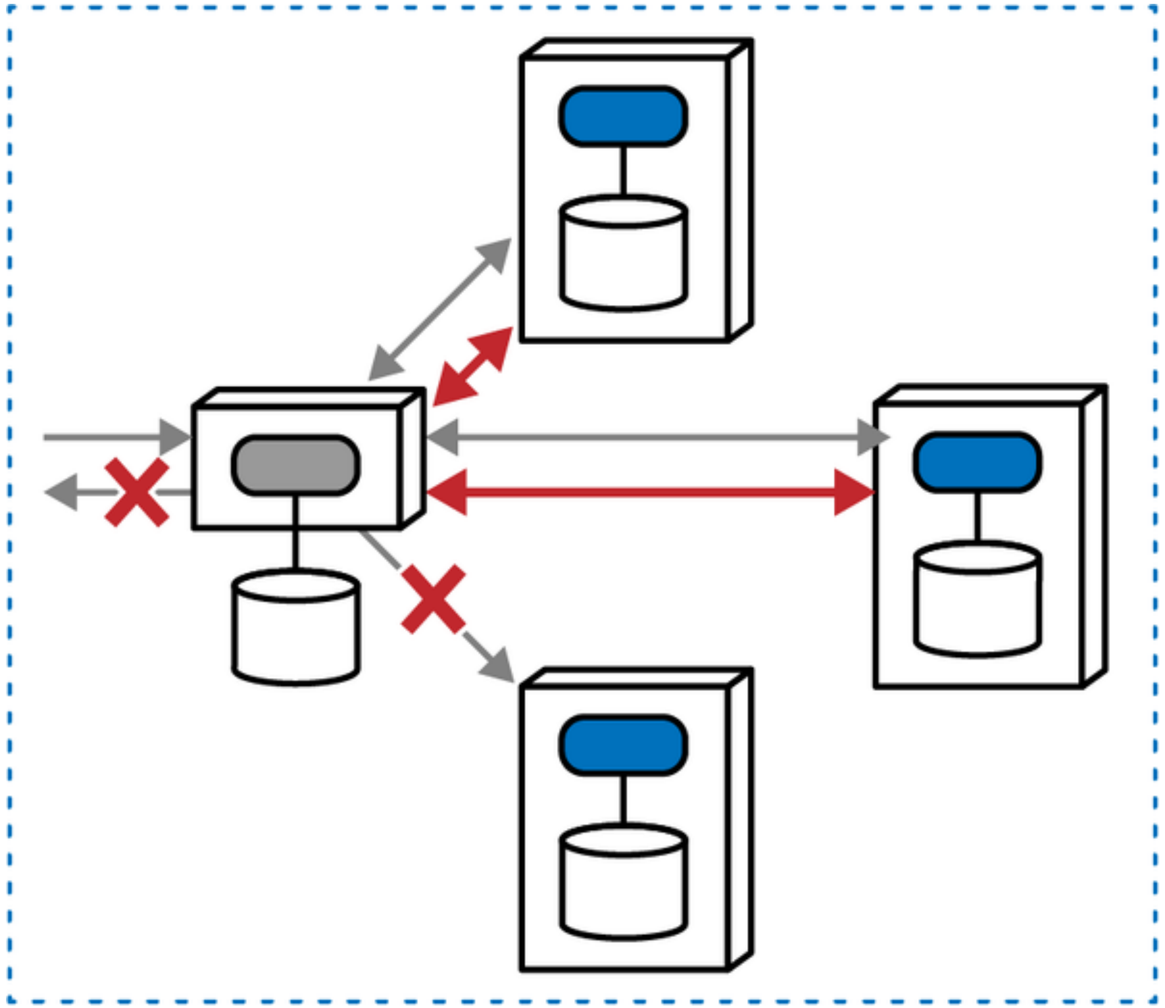
Pattern name	Communication	Consistency	Coordination
Epic Saga**(sao)**	Synchronous	Atomic	Orchestrated
Phone Tag Saga**(sac)**	Synchronous	Atomic	Choreographed
Fairy Tale Saga**(seo)**	Synchronous	Eventual	Orchestrated
Time Travel Saga**(sec)**	Synchronous	Eventual	Choreographed
Fantasy Fiction Saga**(aao)**	Asynchronous	Atomic	Orchestrated
Horror Story**(aac)**	Asynchronous	Atomic	Choreographed
Parallel Saga**(aeo)**	Asynchronous	Eventual	Orchestrated
Anthology Saga**(aec)**	Asynchronous	Eventual	Choreographed

Epic Saga


Epic Saga
Sync
Async
Communication
Atomic
Eventual
Consistency
Orchestrated
Choreography
Coordination
Async
Sync
Atomic
Eventual
Orchestrated
Choreography



Epic Saga(sao) pattern	Ratings
Communication	Synchronous
Consistency	Atomic
Coordination	Orchestrated
Coupling	Very high
Complexity	Low
Responsiveness/ availability	Low
Scale/elasticity	Very low



Epic Saga(sao) pattern	Ratings
Communication	Synchronous
Consistency	Atomic
Coordination	Orchestrated
Coupling	Very high
Complexity	Low
Responsiveness/ availability	Low
Scale/elasticity	Very low

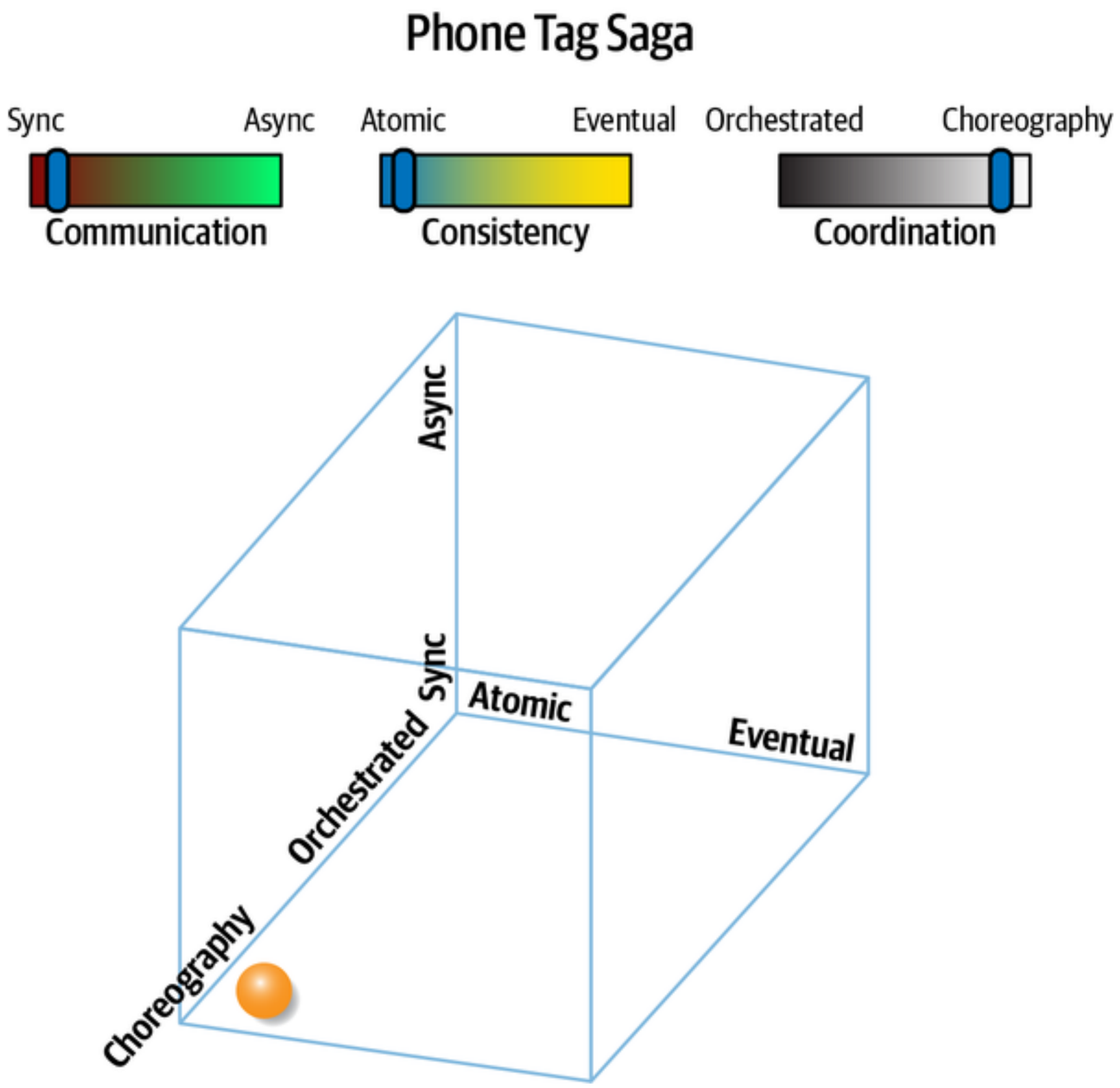
Simple Review

- The clear advantage of the Epic Saga(sao) is the transactional coordination that mimics monolithic systems, coupled with the clear workflow owner represented via an orchestrator

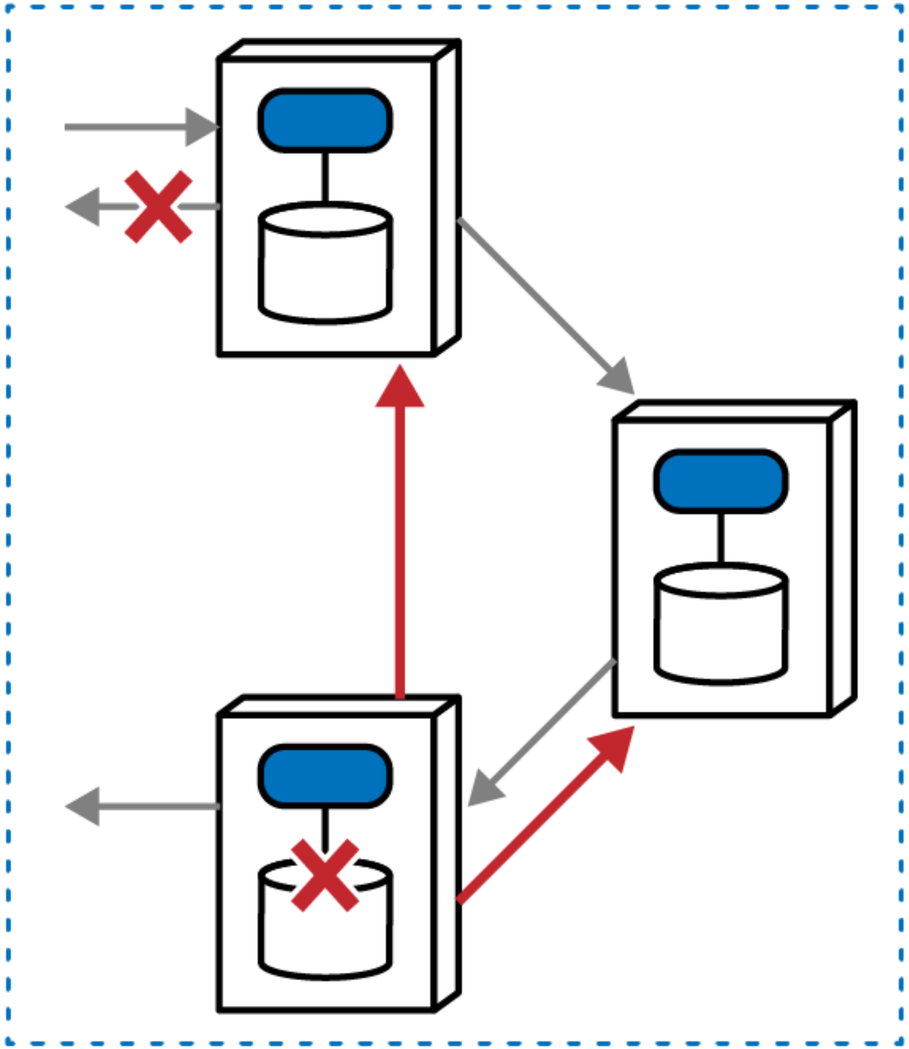
- The various patterns used to implement distributed transactionality (such as compensating transactions) succumb to a wide variety of failure modes and boundary conditions, along with adding inherent complexity via undo operations.

- The orchestrator must make sure that all participants in the transaction have succeeded or failed, creating timing bottlenecks.

Phone Tag Saga


Phone Tag Saga
Sync
Async
Communication
Atomic
Eventual
Consistency
Orchestrated
Choreography
Coordination
Async
Sync
Atomic
Eventual
Orchestrated
Choreography

Phone Tag Saga

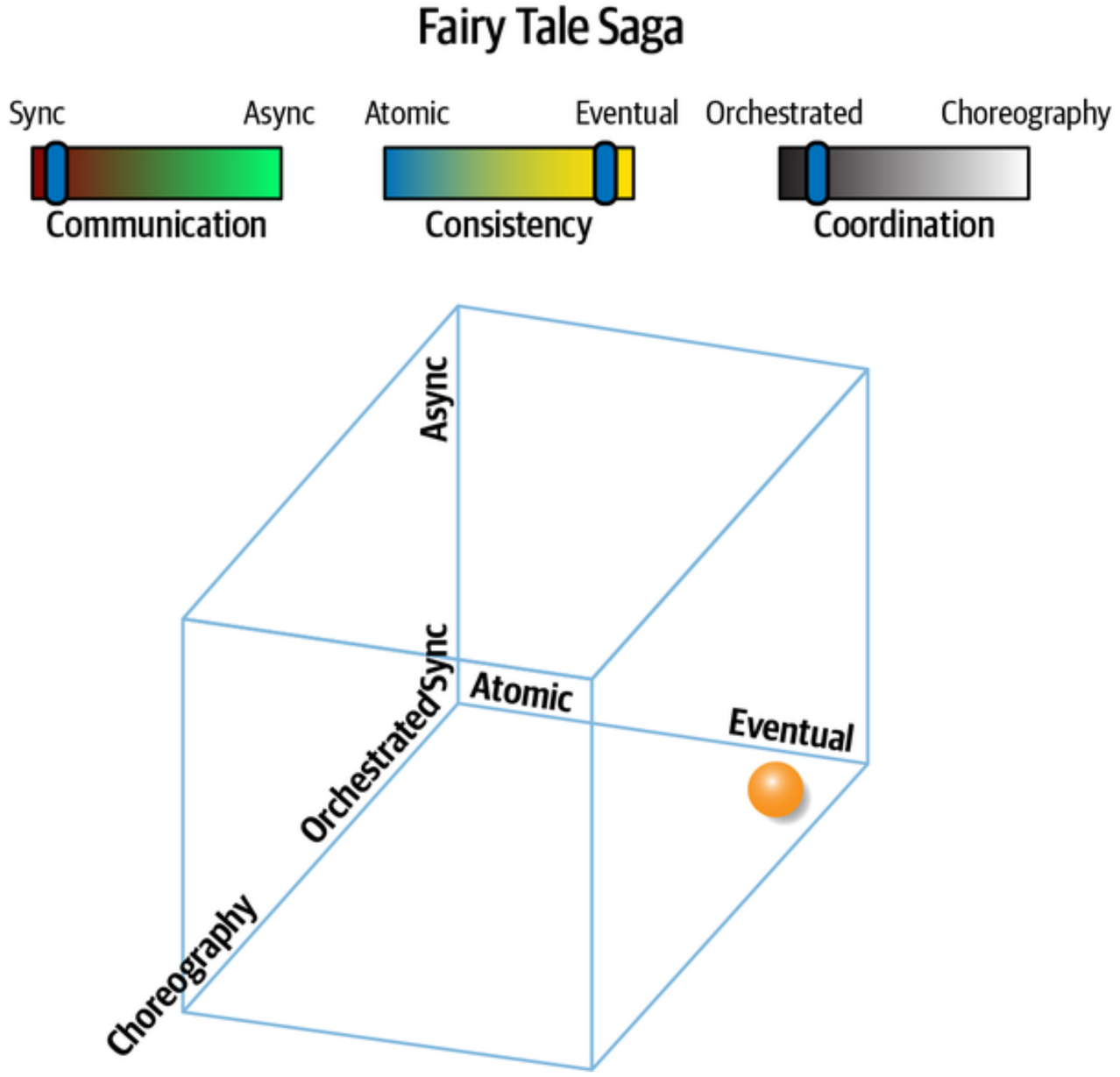


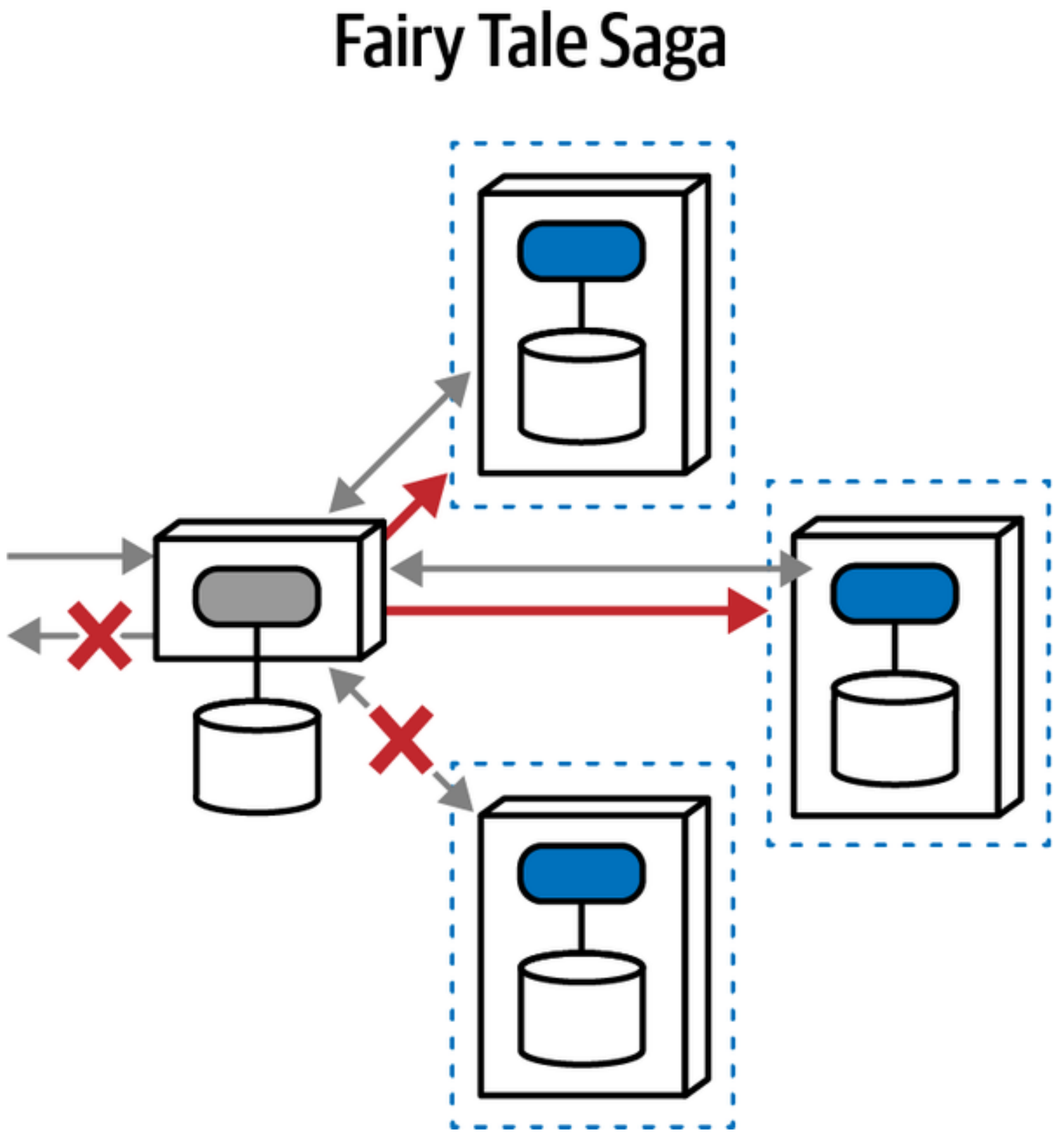
Phone Tag Saga(sac)	Ratings
Communication	Synchronous
Consistency	Atomic
Coordination	Choreographed
Coupling	High
Complexity	High
Responsiveness/ availability	Low
Scale/elasticity	Low

Simple Review

- Domain services must contain more logic about the workflow context they participate within, including error handling and routing.
- For complex workflows, the *front controller* in this pattern will become as complex as most mediators, reducing the appeal and applicability of this pattern.
- The Phone Tag Saga(sac) pattern is better for simple workflows that don't have many common error conditions.
- While it offers a few better characteristics than the **Epic Saga(sao)**, the complexity introduced by lack of an orchestrator offsets many of the advantages.

Fairy Tale Saga


Fairy Tale Saga
Sync
Async
Communication
Atomic
Eventual
Consistency
Orchestrated
Choreography
Coordination
Async
Sync
Atomic
Eventual
Orchestrated
Choreography



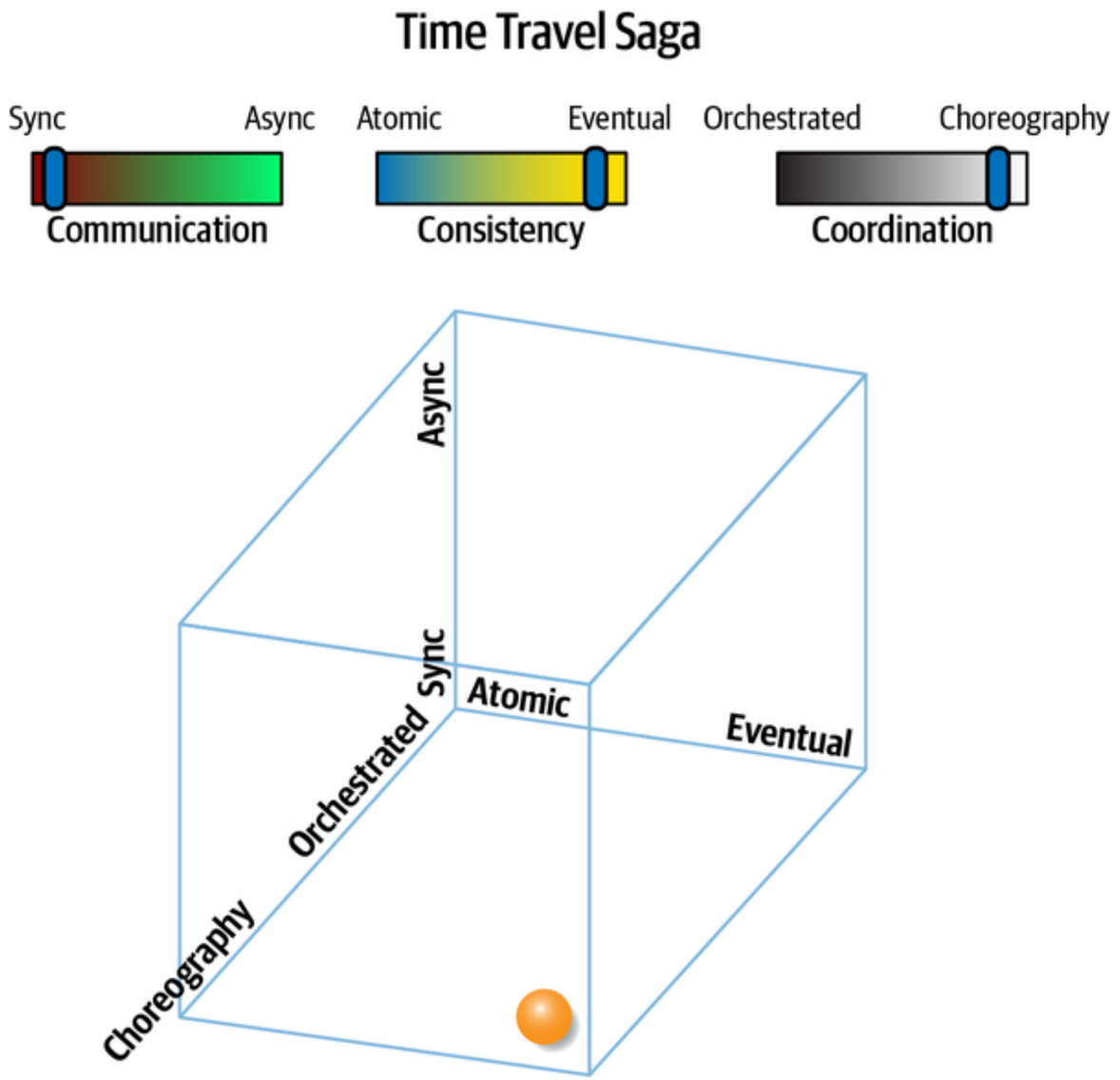
Fairy Tale Saga(seo)	Ratings
Communication	Synchronous
Consistency	Eventual
Coordination	Orchestrated
Coupling	High
Complexity	Very low
Responsiveness/ availability	Medium
Scale/elasticity	High

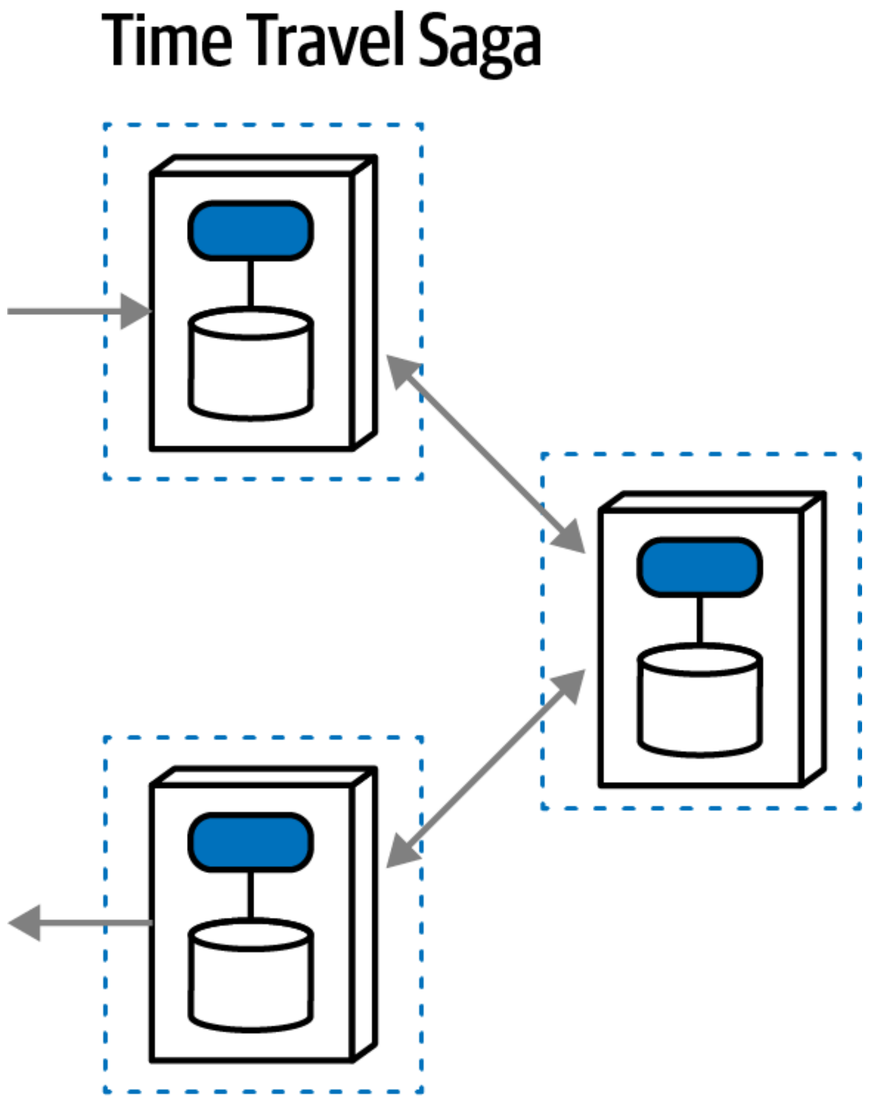
Simple Review

- The biggest appealing advantage of the **Fairy Tale Saga(seo)** is the lack of holistic transactions. Each domain service manages its own transactional behavior, relying on eventual consistency for the overall workflow.

- This is a much more attractive pattern and appears commonly in many microservices architectures. Having a mediator makes managing workflows easier, synchronous communication is the easier of the two choices, and eventual consistency removes the most difficult coordination challenge, especially for error handling.

Time Travel Saga


Time Travel Saga
Sync
Async
Communication
Atomic
Eventual
Consistency
Orchestrated
Choreography
Coordination
Async
Sync
Atomic
Eventual
Orchestrated
Choreography

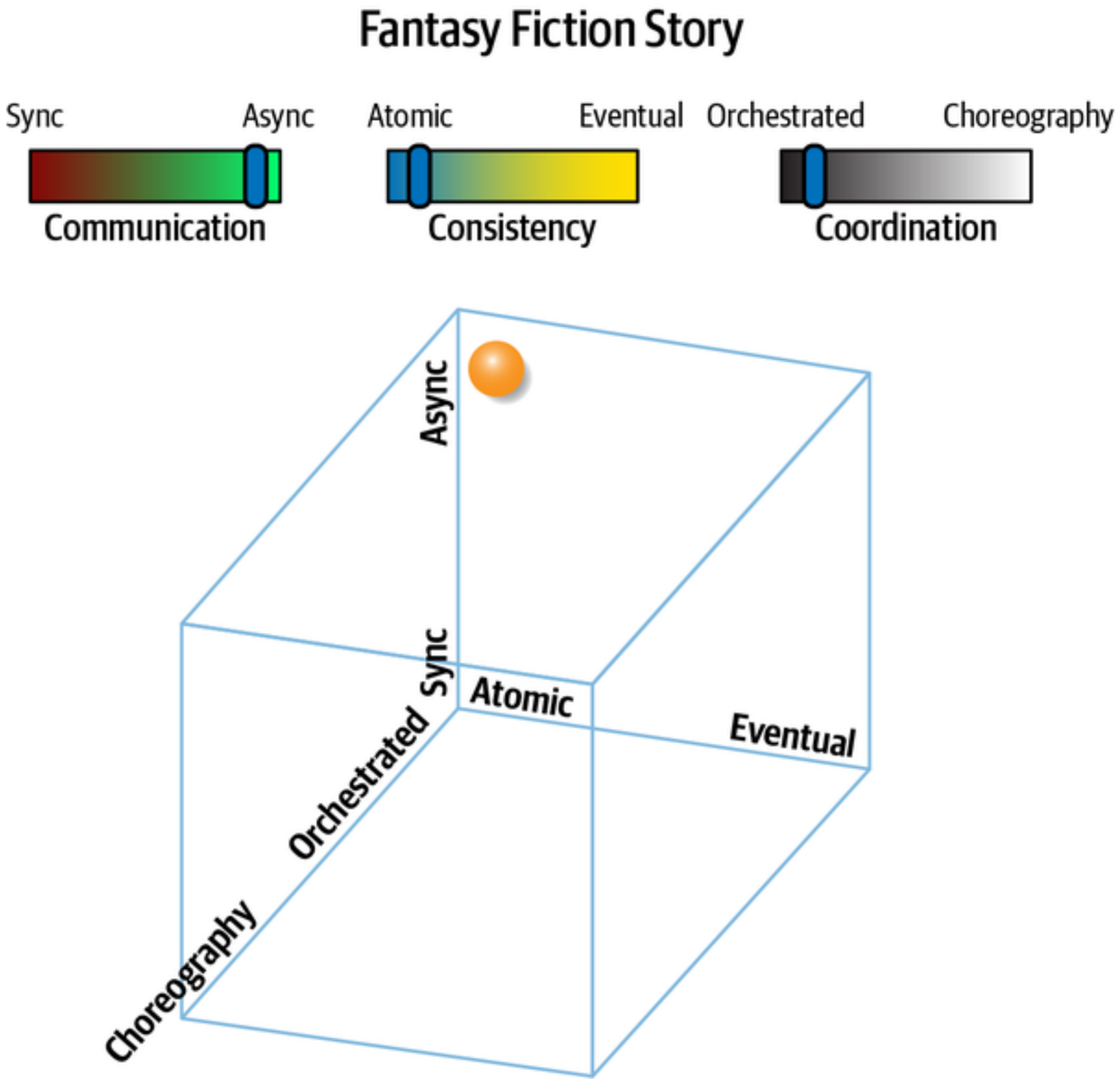


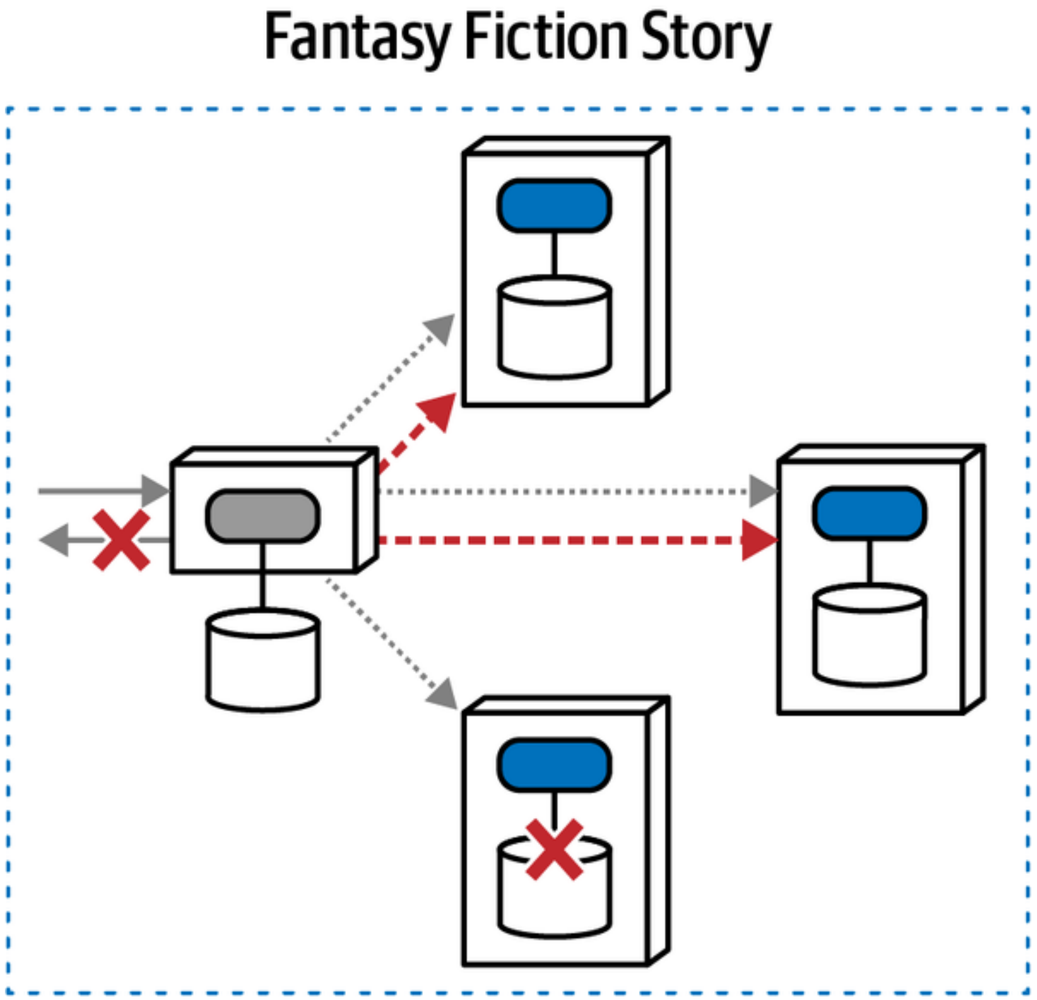
Time Travel Saga(sec)	Ratings
Communication	Synchronous
Consistency	Eventual
Coordination	Choreographed
Coupling	Medium
Complexity	Low
Responsiveness/availability	Medium
Scale/elasticity	High

Simple Review

- The lack of transactions in the **Time Travel Saga(sec)** pattern makes workflows easier to model; however, the lack of an orchestrator means that each domain service *must include most workflow state and information*.

Fantasy Fiction Saga


Fantasy Fiction Story
Sync
Async
Communication
Atomic
Eventual
Consistency
Orchestrated
Choreography
Coordination
Async
Sync
Atomic
Eventual
Orchestrated
Choreography



Fantasy Fiction	Ratings
Communication	Asynchronous
Consistency	Atomic
Coordination	Orchestrated
Coupling	High
Complexity	High
Responsiveness/availability	Low
Scale/elasticity	Low

Simple Review

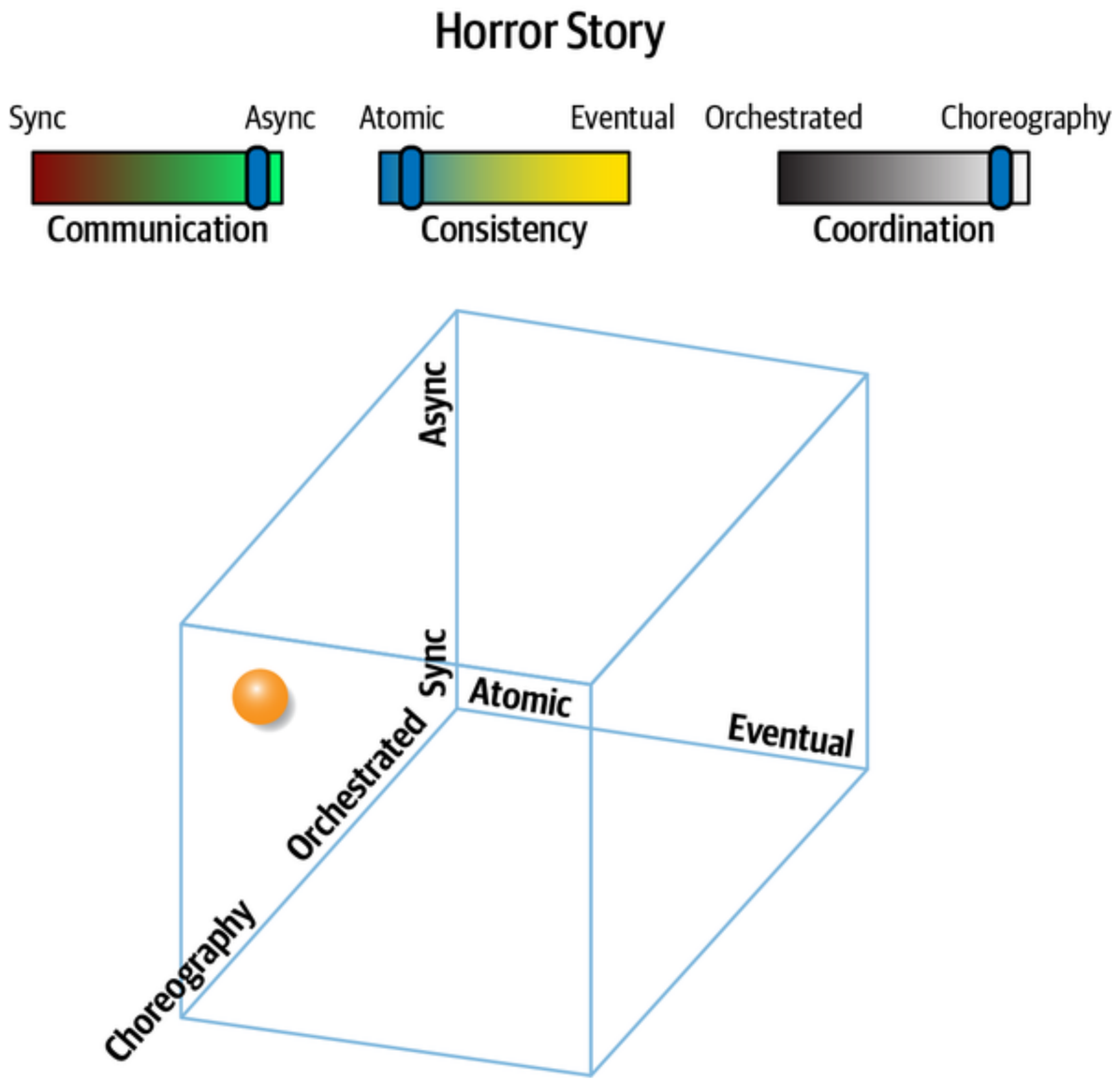
- Adding asynchronicity to orchestrated workflows adds asynchronous transactional state to the equation, removing serial assumptions about ordering and adding the possibilities of deadlocks, race conditions, and a host of other parallel system challenges.

- For example, suppose a transactional workflow Alpha begins. Because everything is asynchronous, while Alpha is pending, transactional workflow Beta begins. Now, the mediator must keep track of the state of all ongoing transactions in pending state.

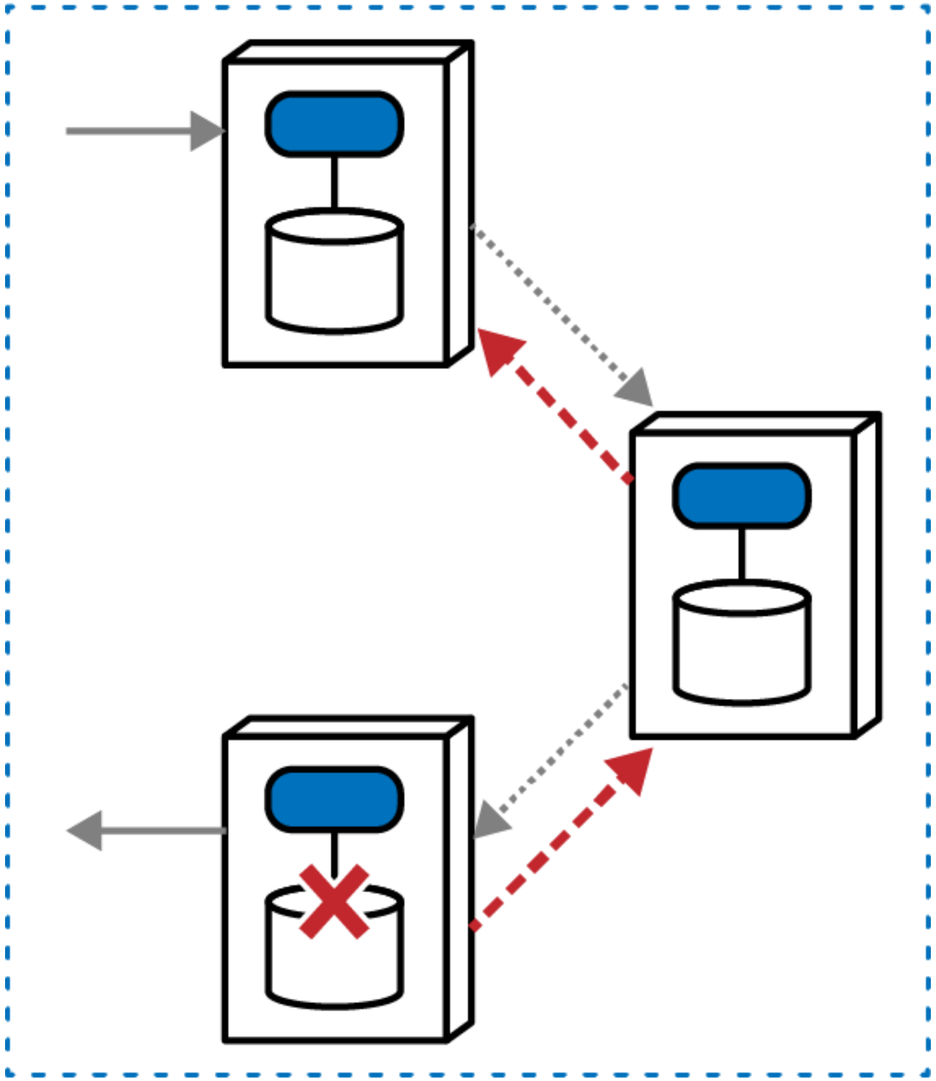
- It gets worse. Suppose that workflow Gamma begins, but the first call to the domain service depends on the still pending outcome of Alpha—how can an architect model this behavior? While possible, the complexity grows and grows.

- This pattern is unfortunately more popular than it should be, mostly from the mis-guided attempt to improve the performance of **Epic Saga(sao)** while maintaining transactionality; a better option is usually **Parallel Saga(aeo)**.

Horror Story Saga


Horror Story
Sync
Async
Communication
Atomic
Eventual
Consistency
Orchestrated
Choreography
Coordination
Async
Sync
Atomic
Eventual
Orchestrated
Choreography

Horror Story



Horror Story(aac)	Ratings
Communication	Asynchronous
Consistency	Atomic
Coordination	Choreographed
Coupling	Medium
Complexity	Very high
Responsiveness/availability	Low
Scale/elasticity	Medium

Simple Review

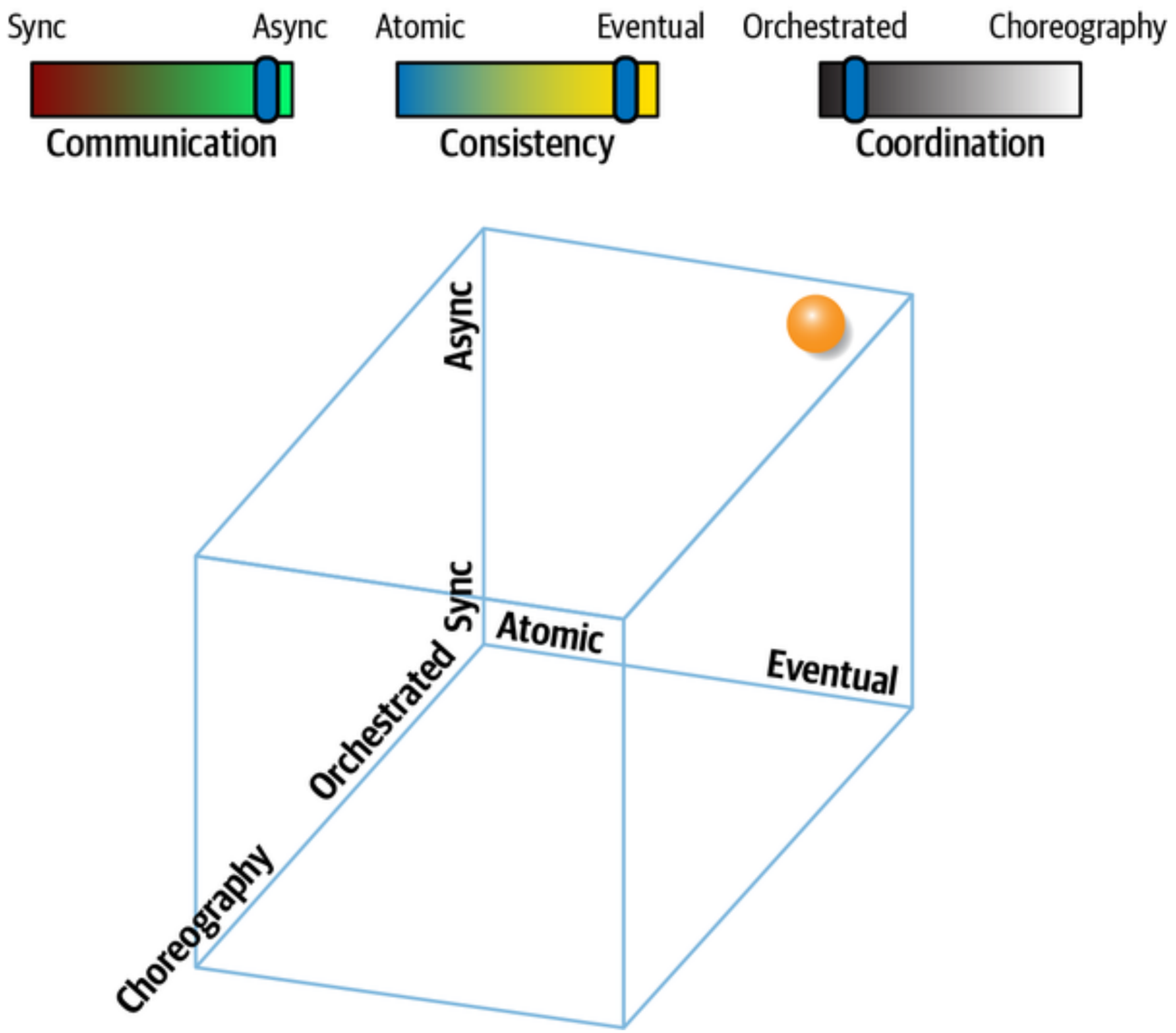
- The aptly named **Horror Story(aac)** pattern is often the result of a well-meaning architect starting with an **Epic Saga(sao)** pattern, noticing slow performance because of complex workflows, and realizing that techniques to improve performance include asynchronous communication and choreography.

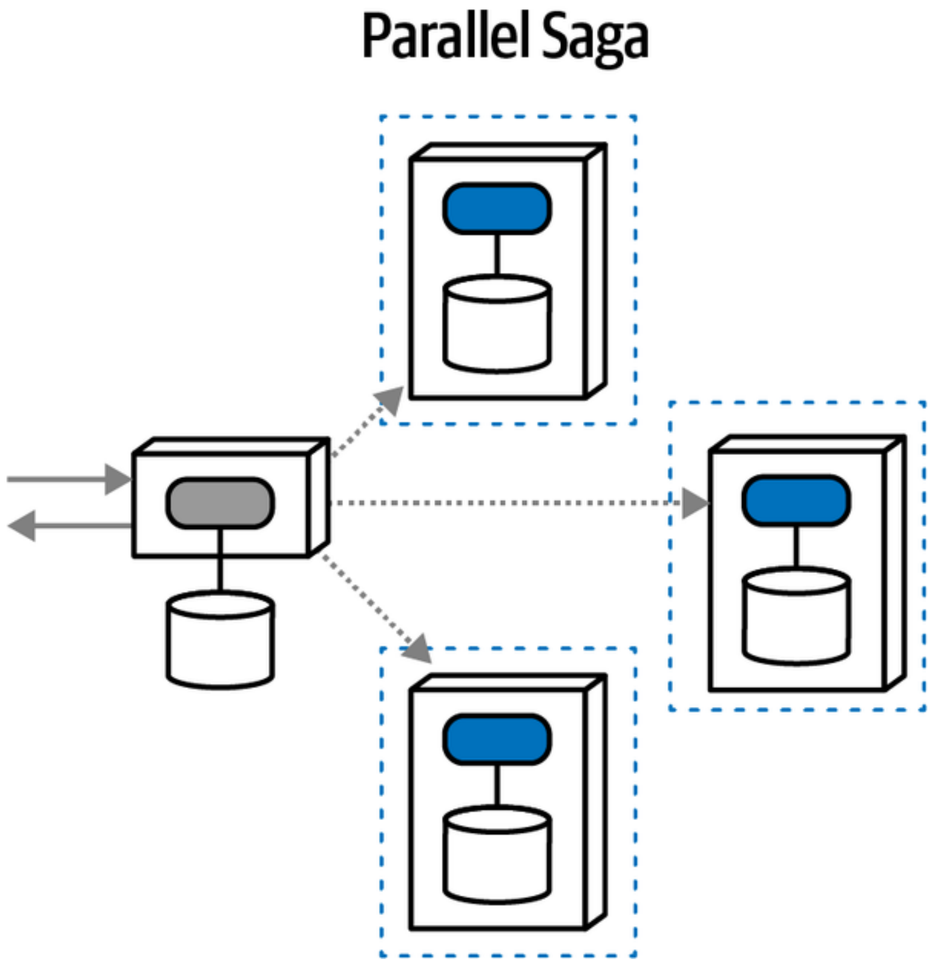
- However, this thinking provides an excellent example of not considering all the entangled dimensions of a problem space.

- In isolation, asynchronous communication improves performance. However, as architects, we cannot consider it in isolation when it is entangled with other architecture dimensions, such as consistency and coordination.

Parallel Saga

Parallel Saga





Parallel Saga(aeo)	Ratings
Communication	Asynchronous
Consistency	Eventual
Coordination	Orchestrated
Coupling	Low
Complexity	Low
Responsiveness/availability	High
Scale/elasticity	High

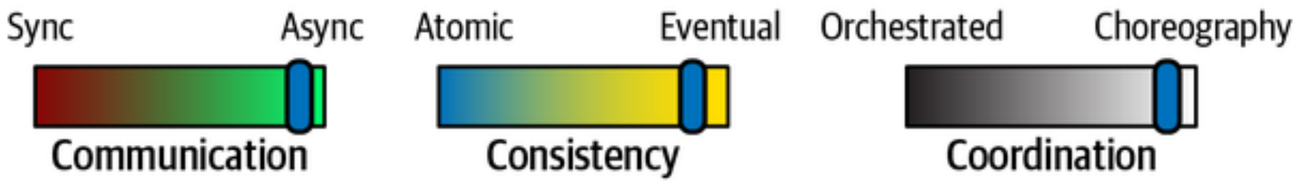
Simple Review

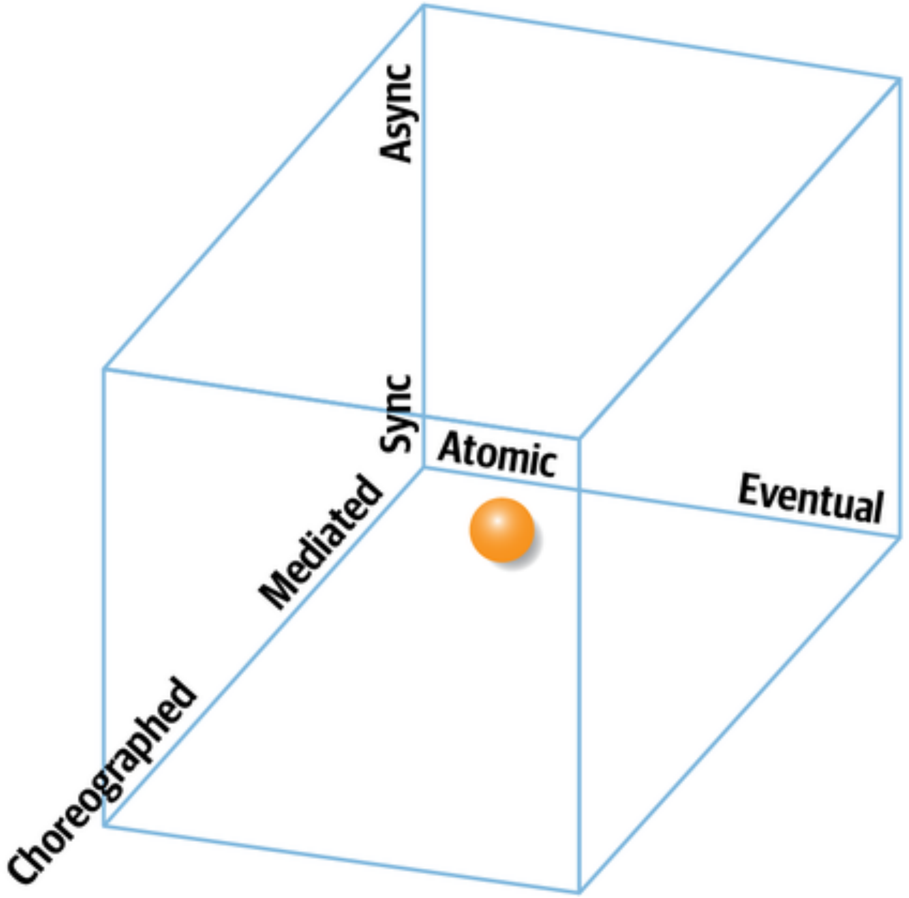
- Overall, the **Parallel Saga(aeo)** pattern offers an attractive set of trade-offs for many scenarios, especially with complex workflows that need high scale.

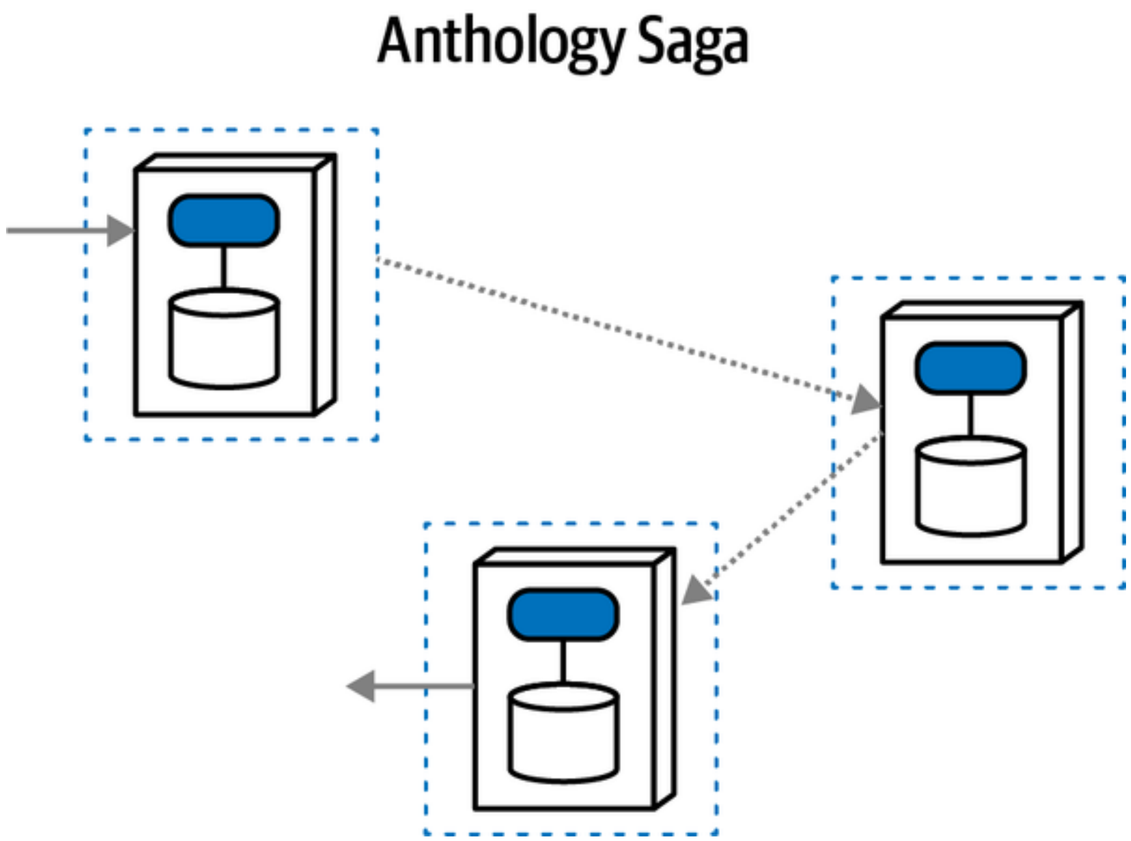
- If an error occurs during the execution of a workflow, the mediator can send asynchronous messages to each involved domain service to compensate for the failed change, which may entail retries, data synchronization, or a host of other remediations.

Anthology Saga

Anthology Saga







Anthology Saga(aec)	Ratings
Communication	Asynchronous
Consistency	Eventual
Coordination	Choreographed
Coupling	Very low
Complexity	High
Responsiveness/availability	High
Scale/elasticity	Very high

Simple Review

- Pattern is well suited to extremely high throughput communication with simple or infrequent error conditions.
- However, this pattern doesn’t work particularly well for complex workflows, especially around resolving data consistency errors.
- This pattern works best for simple, mostly linear workflows, where architects desire high processing throughput.
- This pattern provides the most potential for both high performance and scale, making it an attractive choice when those are key drivers for the system.

Case Study: Exactly Once Semantics in Kafka

Mathias Verraes
@mathiasverraes

Follow

There are only two hard problems in distributed systems: 2. Exactly-once delivery 1. Guaranteed order of messages 2. Exactly-once delivery

RETWEETS 6,775 LIKES 4,727

10:40 AM - 14 Aug 2015

69 6.8K 4.7K

Producer Transactions in Kafka

```
producer.initTransactions();
try {
  producer.beginTransaction();
  producer.send(record1);
  producer.send(record2);
  producer.commitTransaction();
} catch(ProducerFencedException e) {
  producer.close();
} catch(KafkaException e) {
  producer.abortTransaction();
}
```

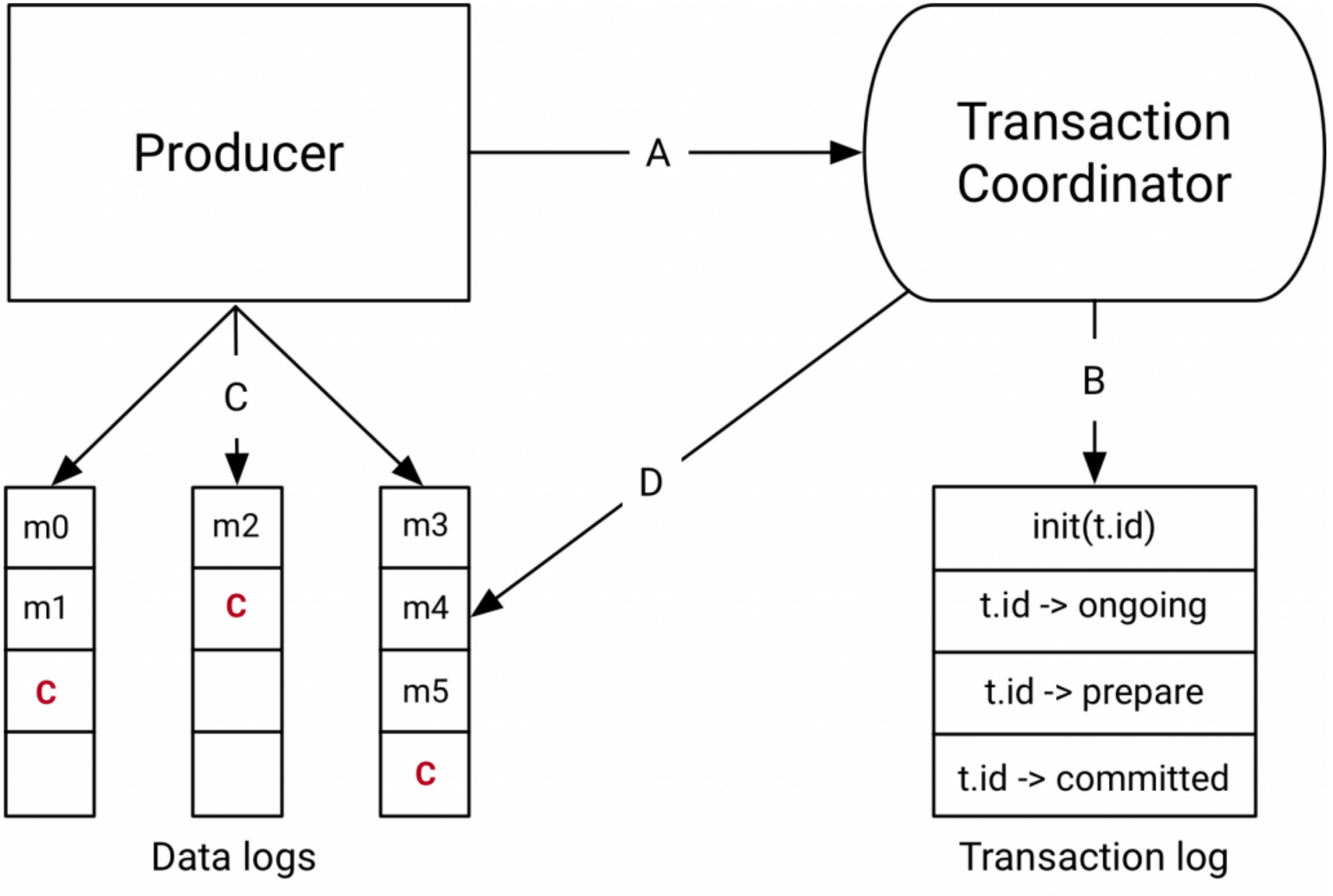
Enabling Idempotence on the Producer

```
// create Producer properties
Properties properties = new Properties();

properties.setProperty(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
"true");
```

Consumer in Kafka

```
properties.put(ConsumerConfig.ISOLATION_LEVEL_CONFIG,
“read_committed");
```


Producer
A
Transaction Coordinator
C
B
D
m0
m1
C
m2
C
m3
m4
m5
C
Data logs
init(t.id)
t.id -> ongoing
t.id -> prepare
t.id -> committed
Transaction log

Stream Processor in Kafka

```
producer.initTransactions();

consumer.assign(inputTopicPartitions(“inputTopic”));

while (true) {
  ConsumerRecords consumed = consumer.poll(Duration.ofMillis(5000));
  Map<TopicPartition, OffsetAndMetadata> consumedOffsets = offsets(consumed);
  List processed = process(consumed);
  try {
    // Write the records and commit offsets under a single transaction
    producer.beginTransaction();
    for (ProducerRecord record : processed)
      producer.send(record);
    producer.sendOffsetsToTransaction(consumedOffsets, groupId);
    producer.commitTransaction();
  } catch(ProducerFencedException e) {
    producer.close();
  } catch(KafkaException e) {
    producer.abortTransaction();
    resetToLastCommittedPositions(consumer);
  }
}
```

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Mastodon: https://mastodon.social/@dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext