



# Event-Driven Architecture Workshop (v2)

## Moving from Command-Driven to Event Choreography

### Workshop Overview

This workshop explores different patterns for event-driven architecture, focusing on how services should communicate through events while maintaining loose coupling, single responsibility principles, and safe concurrency at scale.

---

### Executive Summary

### Why This Workshop Exists

Many teams move from command-driven, step-by-step processing ("Lambda A calls Lambda B") to event-driven designs ("Lambda A emits an event; something else reacts"). The tricky parts are:

- What data belongs in each event
- How to coordinate work that spans multiple services
- How to handle concurrency, duplicates, and out-of-order events
- How to ensure we *reliably* publish events without losing them

### Core Ideas

- **Emit facts, not commands:** Each service emits events only about its own work and its own data.
- **Correlate by ID:** Downstream services correlate multiple events using a shared business key (e.g. `orderId`, `fileId`).

- **Keep handlers idempotent:** Event handlers should safely run multiple times without breaking invariants.
- **Make state explicit:** Use a state store (e.g. DynamoDB) or an event store to track which events have arrived.
- **Design for concurrency:** Assume events for the same key may arrive in parallel, out of order, or more than once.
- **Use outbox/event sourcing when needed:** For high integrity and auditability, persist state and events together, then publish from a durable store.

## Recommended Default for Most Workflows

- Use **event choreography** (services react to events independently), with a **state table** (DynamoDB) to correlate multiple events per business key.
  - Use **optimistic concurrency** or **versioning** in the state table to avoid lost updates under concurrent event processing.
  - Use an **outbox pattern** or transactional write when you must not lose events between your database and EventBridge.
- 

## Table of Contents

1. [Core Concepts](#)
2. [Architecture Overview \(C4-Style\)](#)
3. [Event-Driven Patterns Comparison](#)
4. [Concurrency & High-Throughput Considerations](#)
5. [Workshop Scenario: The Pizza Order System](#)
6. [Event Payload Examples \(CloudEvents\)](#)
7. [State Management Strategies](#)
8. [Reliable Event Delivery \(Outbox Pattern\)](#)
9. [Handling Edge Cases](#)
10. [Hands-On Activities](#)
11. [Best Practices](#)
12. [Summary: When to Use Each Pattern](#)
13. [Additional Resources](#)
14. [Workshop Wrap-Up Discussion Questions](#)

## 15. Appendix: Quick Reference

### Facilitation Tip

For a 60–90 minute session, focus on sections 1–5, 7, 9, and Activities 1–3. Treat sections 4, 8, and the Event Sourcing parts of 7 as advanced/optional for follow-up.

---

## Core Concepts

### What Are We Solving?

When moving from command-driven to event-driven architecture, we need to decide:

- **What data should events contain?**
- **Who is responsible for orchestrating multi-step workflows?**
- **How do we handle services that need data from multiple events?**
- **How do we stay correct when many events are processed concurrently?**

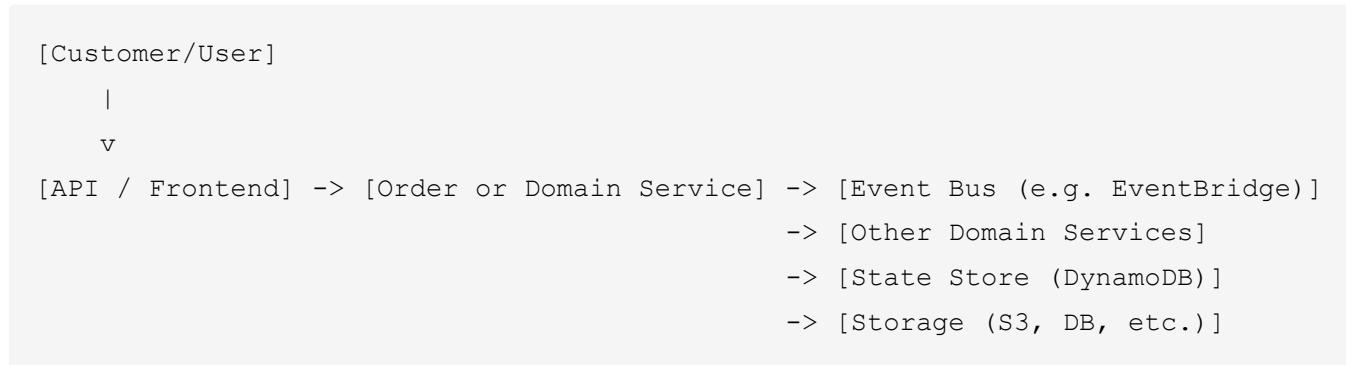
### The Key Principle

**Services should only emit events about their own domain and only include data they are responsible for.**

---

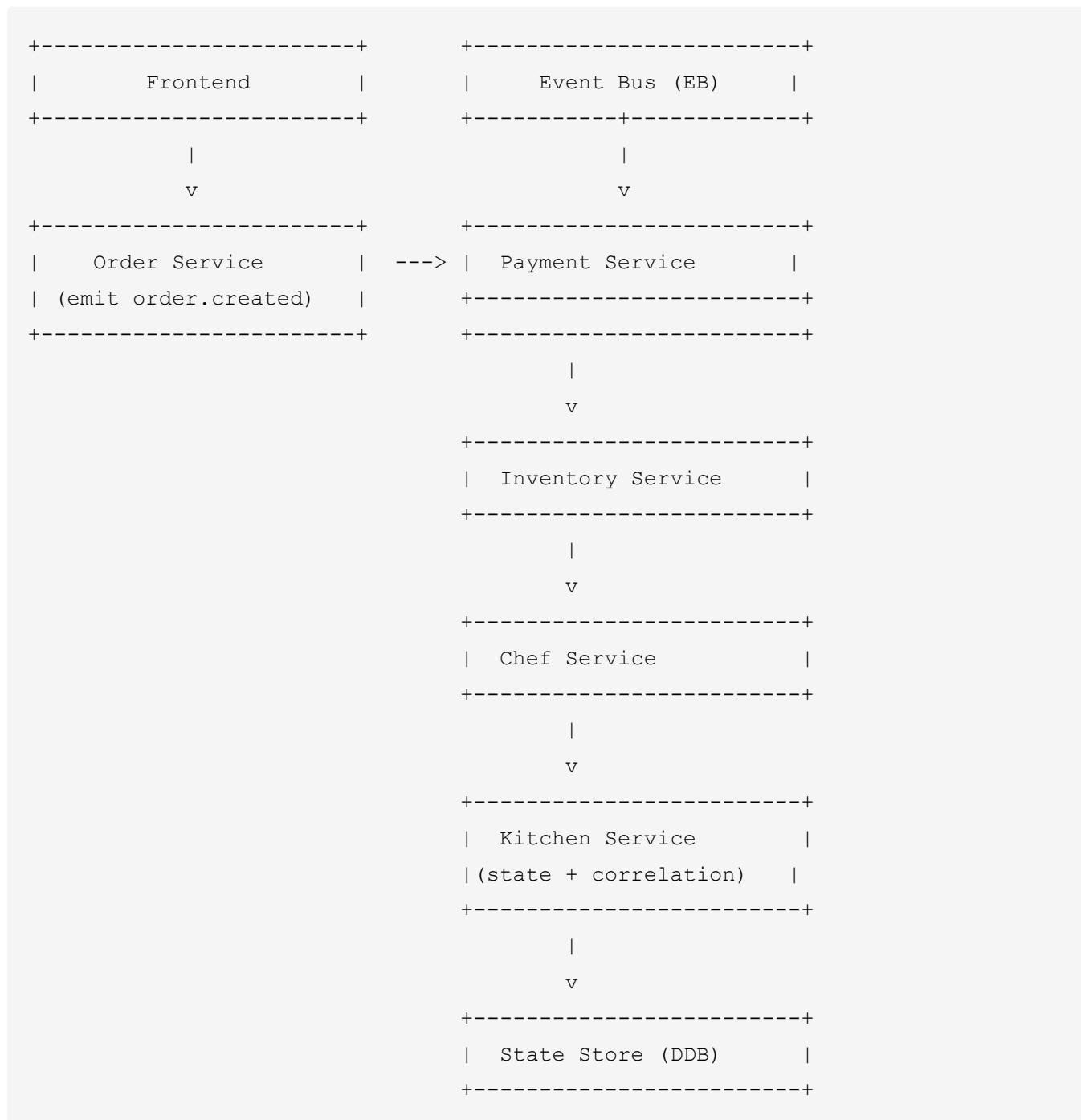
# Architecture Overview (C4-Style)

## System Context (High-Level)



- The **user** or upstream system triggers an initial action (e.g. place order, upload file).
- A **front door** (API Gateway / frontend / HTTP endpoint) hands off to a **domain service Lambda**.
- That service emits events to **EventBridge** (or a similar bus) using CloudEvents.
- Other services (payment, inventory, kitchen, etc.) **subscribe** to events, do work, and emit their own domain events.
- A **state store** keeps track of which events have been seen per business key.

# Container View (Pizza Example)

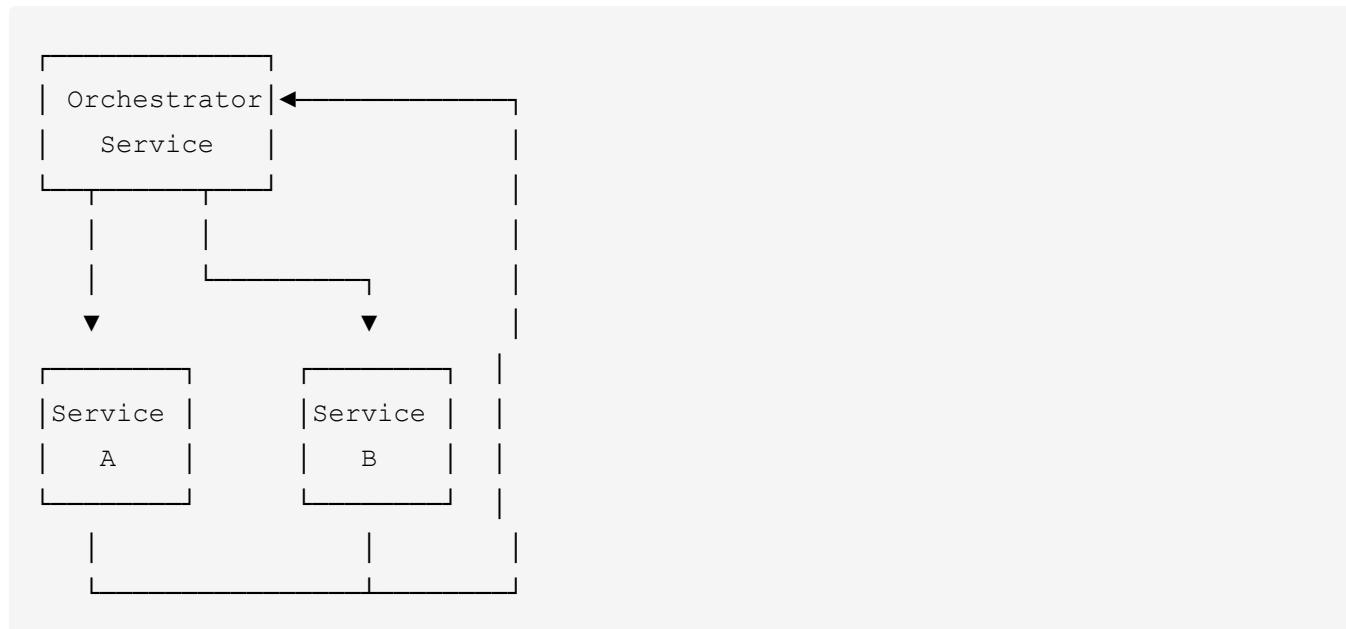


- All services publish and subscribe via **EventBridge**.
- **Kitchen Service** correlates `payment`, `inventory`, and `chef` events using `orderId` and persists state in DynamoDB.
- This same shape applies to many real-world flows.

# Event-Driven Patterns Comparison

## Pattern 1: Event Orchestration (Central Orchestrator)

A central orchestrator service coordinates the workflow by listening to events and explicitly commanding the next steps.



### Pros:

- ✓ Clear workflow visibility - easy to understand the entire process
- ✓ Simple debugging - one place to look for workflow logic
- ✓ Easy to implement timeouts and compensating transactions
- ✓ Good for complex business processes with many decision points

### Cons:

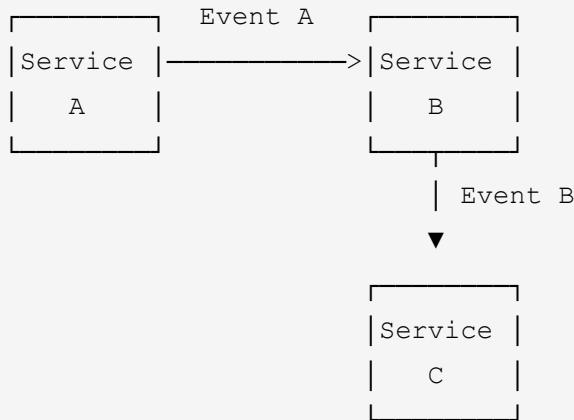
- ✗ Single point of failure
- ✗ Orchestrator becomes a "god service" with too much knowledge
- ✗ Tight coupling - orchestrator must know about all services
- ✗ Harder to scale - orchestrator can become a bottleneck
- ✗ Changes to workflow require orchestrator updates

### When to Use:

- Complex workflows with conditional logic
  - Business processes requiring audit trails
  - When you need saga pattern with compensations
- 

## Pattern 2: Event Choreography (Distributed Events)

Services react to events independently. No central coordinator. Each service listens for events it cares about and emits events about its own work.



### Pros:

- ✓ Loose coupling - services don't know about each other
- ✓ Better scalability - no central bottleneck
- ✓ Easy to add new services without changing existing ones
- ✓ Services focus on their domain only (single responsibility)
- ✓ More resilient - no single point of failure

### Cons:

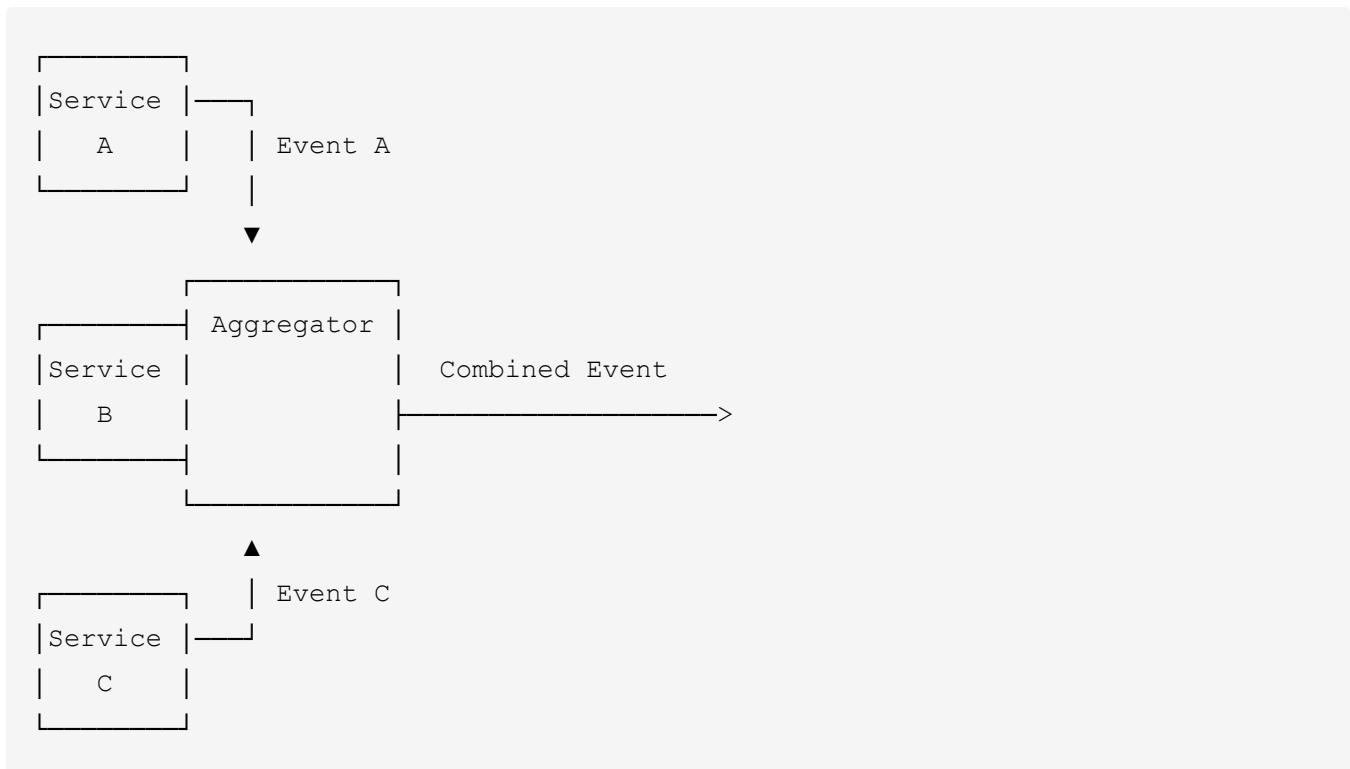
- ✗ Harder to understand complete workflows
- ✗ Debugging distributed logic is challenging
- ✗ Need to manage event correlation and state
- ✗ Risk of event storms or circular dependencies
- ✗ Requires good monitoring and observability

## When to Use:

- Simple, linear workflows
  - When services should be highly independent
  - When you expect frequent changes to the workflow
  - Domain-driven design with clear bounded contexts
- 

## Pattern 3: Hybrid (Event Aggregator)

An aggregator service collects events from multiple sources and emits a combined event when all pieces are ready. The aggregator doesn't orchestrate - it just correlates.



## Pros:

- ✓ Separates correlation logic into one place
- ✓ Other services remain simple and focused
- ✓ Easy to handle timeouts and missing events
- ✓ Can be reused for different aggregation patterns

## Cons:

- ✗ Additional service to maintain
- ✗ Still introduces some coupling (aggregator knows about multiple events)
- ✗ Can become complex if many aggregation patterns exist

#### **When to Use:**

- Multiple services produce data needed by one consumer
  - Complex correlation logic (timeouts, partial matches)
  - When you want to hide complexity from business services
- 

## **Pattern 4: Event Carrying State Transfer (Data Replication)**

Events contain all necessary data. Services maintain local copies of data from other domains.

#### **Pros:**

- ✓ Services can work independently with local data
- ✓ Fast - no need to wait for or query other services
- ✓ Resilient to other service failures

#### **Cons:**

- ✗ Data duplication and eventual consistency
- ✗ Large event payloads
- ✗ Synchronization challenges
- ✗ Not suitable for sensitive data or frequently changing data

#### **When to Use:**

- Read-heavy operations
  - When availability is more important than consistency
  - Reference data that changes infrequently
-

# Concurrency & High-Throughput Considerations

Event-driven systems are naturally concurrent: many Lambdas can process many events at the same time. That's a feature, but it introduces hazards.

## Typical Concurrency Problems

- **Lost updates:** Two events for the same `orderId` arrive concurrently and both update the same item, with the second write overwriting the first.
- **Duplicate side effects:** Retries or duplicate events cause the same action (e.g. charge card, send email) to run twice.
- **Out-of-order decisions:** A failure event processed after a success event might incorrectly mark an order as failed.
- **Hot keys:** A few IDs receive a very high volume of events, causing contention and throttling in your state store.

## Design Principles

- **Per-key idempotency:** Make handlers idempotent *per business key* (e.g. storing `paymentId` and ignoring repeats).
- **Optimistic concurrency:** Use version numbers or condition expressions in your state table so concurrent updates don't silently overwrite each other.
- **Append-only where possible:** Prefer appending new facts (events) over overwriting state; derive state when needed.
- **Partition wisely:** Use correlation IDs as partition keys so related events land together in storagestreams.

## Example: Optimistic Concurrency with DynamoDB

Use a `version` field and conditional updates to ensure state isn't overwritten by concurrent processors:

```

// state-manager-concurrent.ts

import { DynamoDBDocumentClient, UpdateCommand } from "@aws-sdk/lib-dynamodb";


interface OrderState {
  orderId: string;
  events: {
    payment?: PaymentEvent;
    inventory?: InventoryEvent;
    chef?: ChefEvent;
  };
  status: 'waiting' | 'ready' | 'processing' | 'timeout';
  version: number; // optimistic concurrency counter
}

export async function recordEventConcurrent(
  orderId: string,
  eventType: 'payment' | 'inventory' | 'chef',
  eventData: any
): Promise<void> {
  const now = new Date().toISOString();

  await docClient.send(new UpdateCommand({
    TableName: TABLE_NAME,
    Key: { orderId },
    UpdateExpression: [
      'SET',
      '#events.#type = :data,',
      'updatedAt = :now,',
      'ttl = :ttl,',
      'version = if_not_exists(version, :zero) + :one'
    ].join(' '),
    ExpressionAttributeNames: {
      '#events': 'events',
      '#type': eventType,
    },
    ExpressionAttributeValues: {
      ':data': eventData,
      ':now': now,
      ':ttl': Math.floor(Date.now() / 1000) + TTL_HOURS * 3600,
    }
  }));
}

```

```
        ':zero': 0,
        ':one': 1,
    },
    ReturnValues: 'ALL_NEW',
})) ;
}
```

- Each update increments `version` atomically.
- You can compare `version` in subsequent reads to detect races or implement higher-level invariants.

## Reducing Duplicate Side Effects

- Store a **deduplication key** (e.g. `eventId` or `paymentId`) in your state; if you see it again, skip side effects.
- Prefer **at-least-once events + idempotent handlers** over at-most-once delivery (which risks data loss).
- For external side effects (emails, payments), keep a log of what's been sent/charged keyed by a stable ID.

## Handling Hot Keys

- Consider per-key **serialized processing** (e.g. FIFO queues or per-key locks) when correctness matters more than throughput.
- Or design aggregates so that particularly hot entities are split into sub-keys (sharding by day, region, etc.) when safe.

---

## Workshop Scenario: The Pizza Order System

### The Story

A customer orders a custom pizza. Three independent services must complete their work before the pizza can be assembled:

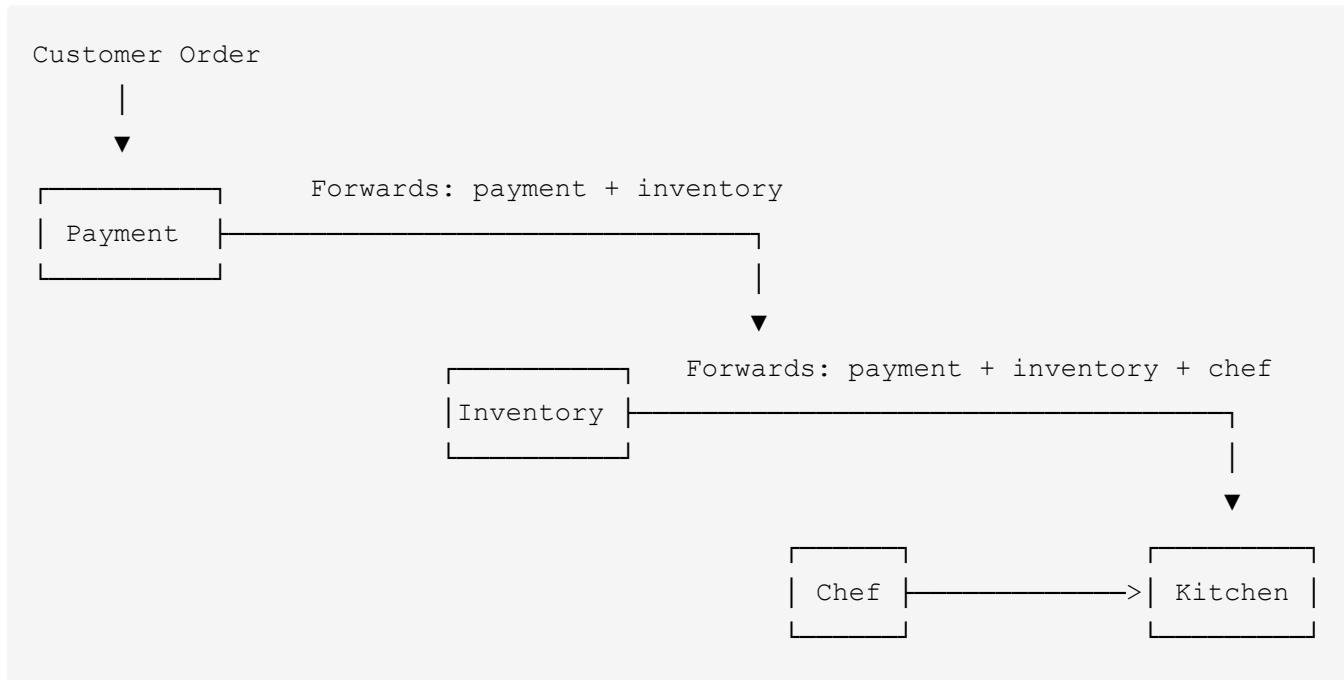
1. **Payment Service** - Validates payment and processes transaction

2. **Inventory Service** - Checks and reserves ingredients
3. **Chef Service** - Validates the recipe and assigns a chef

Once all three have completed successfully, the **Kitchen Service** assembles and bakes the pizza.

## The Problem: Two Approaches

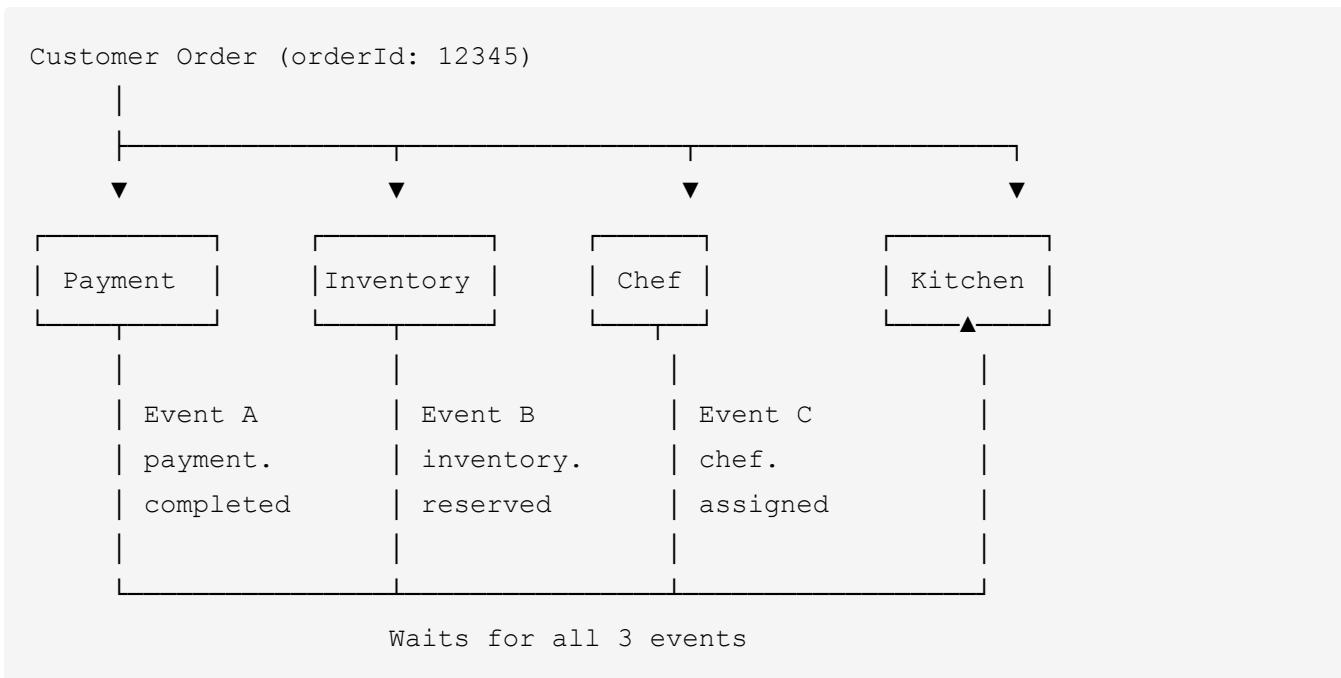
### ✗ Anti-Pattern: Event Chain with Data Forwarding



### Problems:

- Payment service knows about inventory data
- Inventory service knows about payment and chef data
- Changes ripple through the chain
- Services are tightly coupled

## Better Pattern: Event Choreography with Correlation



### Benefits:

- Each service emits only its own domain data
  - Kitchen service correlates events by orderId
  - Easy to add a 4th requirement (e.g., delivery service)
  - Services are loosely coupled
-

# Event Payload Examples (CloudEvents)

## Initial Order Event

```
{  
    "specversion": "1.0",  
    "type": "com.pizzaco.order.created",  
    "source": "order-service",  
    "id": "order-12345-evt-001",  
    "time": "2026-01-13T10:00:00Z",  
    "datacontenttype": "application/json",  
    "data": {  
        "orderId": "12345",  
        "customerId": "cust-789",  
        "pizzaType": "margherita",  
        "toppings": ["extra-cheese", "olives"],  
        "size": "large",  
        "totalAmount": 15.99,  
        "currency": "GBP"  
    }  
}
```

# Payment Completed Event

```
{  
  "specversion": "1.0",  
  "type": "com.pizzaco.payment.completed",  
  "source": "payment-service",  
  "id": "payment-12345-evt-001",  
  "time": "2026-01-13T10:00:15Z",  
  "datacontenttype": "application/json",  
  "data": {  
    "orderId": "12345",  
    "paymentId": "pay-xyz-789",  
    "status": "approved",  
    "amountCharged": 15.99,  
    "currency": "GBP",  
    "transactionId": "txn-001122"  
  }  
}
```

## Inventory Reserved Event

```
{  
  "specversion": "1.0",  
  "type": "com.pizzaco.inventory.reserved",  
  "source": "inventory-service",  
  "id": "inventory-12345-evt-001",  
  "time": "2026-01-13T10:00:18Z",  
  "datacontenttype": "application/json",  
  "data": {  
    "orderId": "12345",  
    "reservationId": "res-456",  
    "ingredientsReserved": [  
      { "ingredient": "dough-large", "quantity": 1 },  
      { "ingredient": "cheese", "quantity": 200, "unit": "grams" },  
      { "ingredient": "olives", "quantity": 50, "unit": "grams" }  
    ],  
    "expiresAt": "2026-01-13T10:30:00Z"  
  }  
}
```

## Chef Assigned Event

```
{  
  "specversion": "1.0",  
  "type": "com.pizzaco.chef.assigned",  
  "source": "chef-service",  
  "id": "chef-12345-evt-001",  
  "time": "2026-01-13T10:00:20Z",  
  "datacontenttype": "application/json",  
  "data": {  
    "orderId": "12345",  
    "chefId": "chef-mario",  
    "chefName": "Mario Rossi",  
    "estimatedPrepTime": 12,  
    "specialtyMatch": true  
  }  
}
```

## Kitchen Ready Event (After All 3 Received)

```
{  
  "specversion": "1.0",  
  "type": "com.pizzaco.kitchen.ready",  
  "source": "kitchen-service",  
  "id": "kitchen-12345-evt-001",  
  "time": "2026-01-13T10:00:25Z",  
  "datacontenttype": "application/json",  
  "data": {  
    "orderId": "12345",  
    "status": "ready-for-preparation",  
    "correlationData": {  
      "paymentId": "pay-xyz-789",  
      "reservationId": "res-456",  
      "chefId": "chef-mario"  
    }  
  }  
}
```

## State Management Strategies

When a service needs to wait for multiple events, it must track which events have been received. Here are common approaches.

### Strategy 1: Database State Table

Store received events in a database table, keyed by correlation ID.

**Example Schema:**

```
interface OrderEventState {  
    orderId: string;  
    paymentCompleted: boolean;  
    paymentData?: PaymentCompletedData;  
    paymentReceivedAt?: Date;  
  
    inventoryReserved: boolean;  
    inventoryData?: InventoryReservedData;  
    inventoryReceivedAt?: Date;  
  
    chefAssigned: boolean;  
    chefData?: ChefAssignedData;  
    chefReceivedAt?: Date;  
  
    status: 'waiting' | 'ready' | 'timeout' | 'failed';  
    createdAt: Date;  
    updatedAt: Date;  
}
```

## Pros:

- ✓ Durable - survives service restarts
- ✓ Queryable - can see current state
- ✓ Easy to implement timeouts

## Cons:

- ✗ Database dependency
- ✗ Transaction complexity
- ✗ Potential bottleneck under very high concurrency

### Note

For high concurrency, combine this pattern with optimistic locking (version numbers) or use an append-only event store (see Event Sourcing below).

## Strategy 2: In-Memory Cache (Redis/ElastiCache)

Store state in Redis with TTL for automatic cleanup.

```
// Pseudo-code

const key = `order:${orderId}:events`;

// Store event
await redis.hset(key, 'payment', JSON.stringify(paymentEvent));
await redis.expire(key, 3600); // 1 hour TTL

// Check if all events received
const events = await redis.hgetall(key);
if (events.payment && events.inventory && events.chef) {
    // All events received!
    await processOrder(orderId, events);
    await redis.del(key); // Clean up
}
```

### Pros:

- ✓ Fast
- ✓ Automatic expiry (TTL)
- ✓ Reduced database load

### Cons:

- ✗ Not durable (can lose state on cache failure)
- ✗ Additional infrastructure

---

## Strategy 3: DynamoDB Streams / EventBridge Pipes

Use AWS-native event correlation capabilities.

### Pros:

- ✓ Serverless and managed

- ✓ Scales automatically
- ✓ Integrated with AWS ecosystem

#### Cons:

- ✗ AWS-specific
  - ✗ Learning curve
  - ✗ Limited query capabilities
- 

## Strategy 4: Event Sourcing (Append-Only Event Store)

Store all events as an append-only log and rebuild state by replaying them.

#### Key Ideas:

- Each change is a new event appended to an event store (e.g., DynamoDB table, Kinesis, or specialized store).
- Current state is derived by replaying events (optionally using **snapshots** to speed things up).
- Concurrency is managed via **expected version** checks on the event stream.

#### Pros:

- ✓ Complete audit trail (what happened, in what order)
- ✓ Can replay history or rebuild projections
- ✓ Natural fit for event-driven systems

#### Cons:

- ✗ More complex to implement and operate
- ✗ Requires careful design of projections/read models
- ✗ Query performance challenges without projections
- ✗ Overkill for simple use cases

#### Example: Optimistic Concurrency with Versioned Event Stream

```

// events-table: PK = orderId, SK = version
// To append a new event, require the last known version

async function appendEvent(orderId: string, expectedVersion: number, event: DomainEvent) {
  await docClient.send(new PutCommand({
    TableName: EVENTS_TABLE,
    Item: {
      orderId,
      version: expectedVersion + 1,
      type: event.type,
      data: event.data,
      createdAt: new Date().toISOString(),
    },
    ConditionExpression: 'attribute_not_exists(#v) OR #v = :expected',
    ExpressionAttributeNames: {
      '#v': 'version',
    },
    ExpressionAttributeValues: {
      ':expected': expectedVersion,
    },
  }));
}

```

- If another writer has already appended version `expectedVersion + 1`, this write fails and you can retry with the new version.
  - Downstream projections (like the Kitchen state table) subscribe to these domain events.
- 

## Recommended Approach for AWS Lambda + TypeScript

For most teams and use cases:

- Use **DynamoDB with TTL** for state management of multi-event workflows.
- Use **idempotent handlers** and **optimistic concurrency** for correctness under retries and parallel processing.
- Introduce **event sourcing** only where you need full history and strong auditability.

```
// state-manager.ts

import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import { DynamoDBDocumentClient, PutCommand, GetCommand, UpdateCommand } from "@aws-sdk/...

const client = new DynamoDBClient({});
const docClient = DynamoDBDocumentClient.from(client);

const TABLE_NAME = process.env.STATE_TABLE_NAME!;
const TTL_HOURS = 24;

export interface OrderState {
    orderId: string;
    events: {
        payment?: PaymentEvent;
        inventory?: InventoryEvent;
        chef?: ChefEvent;
    };
    status: 'waiting' | 'ready' | 'processing' | 'timeout';
    ttl: number;
    createdAt: string;
    updatedAt: string;
    version: number; // optimistic concurrency
}

export async function recordEvent(
    orderId: string,
    eventType: 'payment' | 'inventory' | 'chef',
    eventData: any
): Promise<OrderState> {
    const now = new Date().toISOString();
    const ttl = Math.floor(Date.now() / 1000) + TTL_HOURS * 3600;

    // Optimistic update: increment version, set event atomically
    const result = await docClient.send(new UpdateCommand({
        TableName: TABLE_NAME,
        Key: { orderId },
        UpdateExpression: [
            'SET',
            'events.#type = :data,',
            ':data = { "type": "' + eventType + '", "data": ' + JSON.stringify(eventData) + ' }'
        ],
        ExpressionAttributeNames: { '#type': 'events.type' },
        ExpressionAttributeValues: { ':data': { version: 1 } }
    }));
    return result;
}
```

```

'status = if_not_exists(status, :waiting),',
'ttl = :ttl,' ,
'createdAt = if_not_exists(createdAt, :now),',
'updatedAt = :now,' ,
'version = if_not_exists(version, :zero) + :one'
].join(' '),
ExpressionAttributeNames: {
  '#type': eventType,
},
ExpressionAttributeValues: {
  ':data': eventData,
  ':waiting': 'waiting',
  ':ttl': ttl,
  ':now': now,
  ':zero': 0,
  ':one': 1,
},
ReturnValues: 'ALL_NEW',
}));
}

const state = result.Attributes as OrderState;

// Check if all events received
if (state.events.payment && state.events.inventory && state.events.chef) {
  state.status = 'ready';
  await docClient.send(new UpdateCommand({
    TableName: TABLE_NAME,
    Key: { orderId },
    UpdateExpression: 'SET #status = :ready, updatedAt = :now',
    ExpressionAttributeNames: { '#status': 'status' },
    ExpressionAttributeValues: { ':ready': 'ready', ':now': now },
  }));
}

return state;
}

export async function getOrderState(orderId: string): Promise<OrderState | null> {
  const result = await docClient.send(new GetCommand({
    TableName: TABLE_NAME,

```

```

    Key: { orderId },
} ) );

return (result.Item as OrderState) || null;
}

export function isReady(state: OrderState): boolean {
  return state.status === 'ready';
}

```

## Reliable Event Delivery (Outbox Pattern)

### Problem: Lost Events Between DB and Event Bus

A common failure mode:

- Service writes data to a database (e.g. order status).
- Then tries to publish an event to EventBridge.
- The publish fails (network issue, permissions, Lambda crash).
- The database now shows the new state, but no event was published. Other services will never hear about the change.

### Outbox Pattern Overview

The **outbox pattern** solves this by:

- Writing both **business state** and a **pending event** into the same database transaction.
- A separate **outbox publisher** process polls the outbox table, publishes events to the bus, and marks them as sent.

This gives you **at-least-once** publication from a durable store.

# Simple DynamoDB Outbox Example

```
// When handling a command or incoming event
async function handleOrderCreated(command: OrderCreatedCommand) {
  const orderItem = {
    orderId: command.orderId,
    status: 'created',
    createdAt: new Date().toISOString(),
  };

  const outboxItem = {
    pk: `outbox#${command.orderId}`,
    sk: `event#${Date.now()}`,
    type: 'com.pizzaco.order.created',
    payload: command,
    published: false,
  };

  // Write both in a single transaction
  await docClient.send(new TransactWriteCommand({
    TransactItems: [
      { Put: { TableName: ORDERS_TABLE, Item: orderItem } },
      { Put: { TableName: OUTBOX_TABLE, Item: outboxItem } },
    ],
  }));
}
```

Then, a separate Lambda (triggered on a schedule or via Streams) reads from `OUTBOX_TABLE`:

```

async function publishOutboxEvents() {
  const items = await readUnpublishedOutboxItems();

  for (const item of items) {
    await publishToEventBridge(item.type, item.payload);

    await docClient.send(new UpdateCommand({
      TableName: OUTBOX_TABLE,
      Key: { pk: item.pk, sk: item.sk },
      UpdateExpression: 'SET published = :true, publishedAt = :now',
      ExpressionAttributeValues: {
        ':true': true,
        ':now': new Date().toISOString(),
      },
    }));
  }
}

```

## When to Use the Outbox Pattern

- You must **not** lose events when publishing to EventBridge.
  - You want **stronger guarantees** than "best effort" publishing from Lambdas.
  - You already have a database in the flow and can leverage transactions/Streams.
- 

## Handling Edge Cases

### 1. Timeout - Not All Events Received

**Problem:** What if the chef service is down and never emits an event?

**Solutions:**

#### Option A: Time-based Timeout (TTL + Streams)

Use DynamoDB Streams to trigger a Lambda when TTL expires (or when items are removed):

```

// timeout-handler.ts
export async function handleExpiredState(event: DynamoDBStreamEvent) {
  for (const record of event.Records) {
    if (record.eventName === 'REMOVE') {
      const state = record.dynamodb?.OldImage;
      const orderId = state?.orderId?.S;

      if (state && orderId) {
        const events = state.events?.M || {};
        const hasAllEvents =
          !!events.payment &&
          !!events.inventory &&
          !!events.chef;

        if (!hasAllEvents) {
          // Emit timeout event
          await emitEvent({
            type: 'com.pizzaco.order.timeout',
            data: {
              orderId,
              receivedEvents: Object.keys(events),
              reason: 'not-all-events-received',
            },
          });
        }
      }
    }
  }
}

```

## Option B: EventBridge Scheduled Rule

Create a scheduled rule to check for stale orders:

```
// scheduled-timeout-checker.ts
export async function checkTimeouts() {
  const now = Date.now();
  const timeoutThreshold = now - 30 * 60 * 1000; // 30 minutes

  // Scan for waiting orders older than threshold
  const staleOrders = await scanWaitingOrders(timeoutThreshold);

  for (const order of staleOrders) {
    await emitTimeoutEvent(order.orderId);
    await compensateOrder(order.orderId, order);
  }
}
```

---

## 2. Duplicate Events

**Problem:** EventBridge may deliver events more than once.

**Solution:** Make event handlers idempotent.

```
// Idempotent event handler
export async function handlePaymentEvent(event: CloudEvent) {
  const { orderId, paymentId } = event.data;

  // Check if already processed
  const existing = await getOrderState(orderId);
  if (existing?.events.payment?.paymentId === paymentId) {
    console.log(`Payment event ${paymentId} already processed, skipping`);
    return; // Idempotent - safe to ignore
  }

  // Process event
  await recordEvent(orderId, 'payment', event.data);
}
```

## 3. Events Arrive Out of Order

**Problem:** Inventory event arrives before payment event.

**Solution:** This is fine! Event choreography naturally handles this:

```
// Each event handler just records its event
// Order doesn't matter - we check for completeness each time

export async function handleAnyEvent(eventType: string, eventData: any) {
  const state = await recordEvent(eventData.orderId, eventType, eventData);

  // Check if ready (works regardless of order)
  if (isReady(state)) {
    await processOrder(state);
  }
}
```

---

## 4. Failure Events

**Problem:** Payment failed or inventory not available.

**Solution:** Emit failure events and handle compensation.

```
{
  "specversion": "1.0",
  "type": "com.pizzaco.payment.failed",
  "source": "payment-service",
  "id": "payment-12345-fail-001",
  "time": "2026-01-13T10:00:15Z",
  "data": {
    "orderId": "12345",
    "reason": "insufficient-funds",
    "errorCode": "E001"
  }
}
```

```
// Kitchen service listens for both success and failure events
export async function handlePaymentFailed(event: CloudEvent) {
  const { orderId, reason } = event.data;

  // Mark order as failed
  await updateOrderStatus(orderId, 'failed', reason);

  // Emit cancellation event
  await emitEvent({
    type: 'com.pizzaco.order.cancelled',
    data: { orderId, reason: `payment-failed: ${reason}` },
  });
}
```

---

## 5. Partial Success

**Problem:** Payment and inventory succeeded, but chef service failed.

**Solution:** Implement compensation/rollback.

```
export async function compensateOrder(orderId: string, state: OrderState) {
  const compensations = [];

  // Refund payment if it was completed
  if (state.events.payment) {
    compensations.push(
      emitEvent({
        type: 'com.pizzaco.payment.refund',
        data: {
          orderId,
          paymentId: state.events.payment.paymentId,
          reason: 'order-timeout',
        },
      })
    );
  }

  // Release inventory reservation
  if (state.events.inventory) {
    compensations.push(
      emitEvent({
        type: 'com.pizzaco.inventory.release',
        data: {
          orderId,
          reservationId: state.events.inventory.reservationId,
          reason: 'order-timeout',
        },
      })
    );
  }

  await Promise.all(compensations);
}
```

# Hands-On Activities

## Activity 1: Identify the Anti-Pattern (15 minutes)

**Scenario:** Review this event payload from a virus-scanning service:

```
{  
  "type": "com.fileservice.virus.scanned",  
  "data": {  
    "fileId": "file-123",  
    "scanResult": "safe",  
    "fileDescription": "User profile photo",  
    "uploadedBy": "user-456",  
    "s3Bucket": "my-bucket",  
    "s3Key": "uploads/user-456/photo.jpg",  
    "metadata": {  
      "originalFilename": "vacation.jpg",  
      "contentType": "image/jpeg"  
    }  
  }  
}
```

### Questions:

1. What data doesn't belong in this event?
2. Which service should own each piece of data?
3. How would you refactor this event?
4. What events should the "saver" service listen to?

### Discussion Points:

- Single Responsibility: What is the virus scanner's job?
- Coupling: How does including metadata couple services?
- Change Impact: What happens if we add a new metadata field?



## Activity 2: Design Event Payloads (20 minutes)

**Scenario:** Design a system where a photo must pass through:

1. **Upload Service** - Receives photo from user
2. **Virus Scanner** - Checks for malware
3. **Image Processor** - Resizes and optimizes
4. **Metadata Extractor** - Reads EXIF data
5. **Storage Service** - Saves to S3 only when all 3 checks pass

**Task:** Design the CloudEvents for each service.

**Questions to Consider:**

- What correlation ID will you use?
- What data does each event contain?
- How does the storage service know when to proceed?

**Starter Template:**

```
{  
  "specversion": "1.0",  
  "type": "com.photoapp.???",  
  "source": "???-service",  
  "data": {  
    // What goes here?  
  }  
}
```

---

## Activity 3: Implement State Management (30 minutes)

**Task:** Implement a simple event correlator for the pizza scenario.

**Starting Code:**

```

// kitchen-service/handler.ts

import { EventBridgeEvent } from 'aws-lambda';

// TODO: Implement state management
export async function handleEvent(event: EventBridgeEvent<string, any>) {
    const eventType = event['detail-type'];
    const orderId = event.detail.orderId;

    // TODO: Record this event

    // TODO: Check if all events received

    // TODO: If ready, process order
}

```

## Requirements:

1. Track payment, inventory, and chef events
2. Process order when all 3 received
3. Handle duplicate events (idempotency)
4. Add timeout after 30 minutes

## Test Cases:

- Events arrive in order
- Events arrive out of order
- Duplicate event received
- Only 2 of 3 events received (timeout)

## Activity 4: Handle Failures (20 minutes)

**Scenario:** The inventory service determines ingredients are not available.

**Task:** Design the failure handling:

1. What event does inventory service emit?
2. What should payment service do?

3. What should customer service do?
4. How do you prevent chef service from being assigned?

**Bonus:** What if inventory was available initially but reservation expires before chef is assigned?

---

## Activity 5: Pattern Selection (15 minutes)

For each scenario, choose the best pattern and justify:

Scenario	Best Pattern?	Why?
Simple 3-step process: upload → scan → save		
Complex loan application with 15 steps and conditional logic		
User profile updates that need to be reflected in 5 different services		
Medical prescription requiring doctor approval, insurance check, and pharmacy availability		
E-commerce checkout with payment, inventory, shipping		

### Patterns:

- A) Event Orchestration
  - B) Event Choreography
  - C) Event Aggregator
  - D) Event-Carrying State Transfer
-

# Best Practices

## 1. Event Design Principles

### ✓ DO:

- Use correlation IDs (orderId, requestId, etc.)
- Include timestamps
- Use semantic event types (`order.created`, not `order.event`)
- Follow CloudEvents specification
- Version your event schemas
- Make events immutable
- Include only data the service owns

### ✗ DON'T:

- Include data from other domains
- Use events as RPC calls
- Create circular event dependencies
- Put sensitive data in events without encryption
- Make events too large (>256KB)

---

## 2. Service Boundaries

### Good Example:

```
Payment Service emits:
```

- `payment.initiated`
- `payment.completed`
- `payment.failed`
- `payment.refunded`

```
Payment Service does NOT emit:
```

- `inventory.reserved` (that's inventory service's job)
- `order.completed` (that's order service's job)

---

### 3. Correlation ID Strategy

Use hierarchical correlation IDs:

```
orderId: 12345          (Business ID)
└ payment: pay-12345-001 (Service transaction ID)
└ inventory: inv-12345-001 (Service transaction ID)
└ chef: chef-12345-001   (Service transaction ID)
```

This allows:

- Distributed tracing
  - Debugging across services
  - Audit trails
- 

### 4. Monitoring and Observability

#### Essential Metrics:

- Event processing latency per service
- Number of incomplete order states (waiting)
- Timeout rate
- Event processing failures
- Average time from first to all-events-received

#### Dashboards:

- Funnel view: How many orders at each stage?
- Bottleneck detection: Which service is slowest?
- Correlation completion rate

#### Example CloudWatch Insights Query:

```
fields @timestamp, orderId, status, eventType
| filter status = "waiting"
| stats count() by orderId
| filter count > 10 -- orders with many waiting log records may be stuck
```

## 5. Testing Strategies

### Unit Tests

Test individual event handlers:

```
describe('handlePaymentEvent', () => {
  it('should record payment event', async () => {
    const event = createPaymentEvent({ orderId: '123' });
    await handlePaymentEvent(event);

    const state = await getOrderState('123');
    expect(state.events.payment).toBeDefined();
  });

  it('should be idempotent', async () => {
    const event = createPaymentEvent({ orderId: '123', paymentId: 'pay-1' });
    await handlePaymentEvent(event);
    await handlePaymentEvent(event); // Second time

    // Should only have one payment event recorded
    const state = await getOrderState('123');
    expect(state.events.payment.paymentId).toBe('pay-1');
  });
});
```

### Integration Tests

Test event correlation:

```

describe('order correlation', () => {
  it('should process order when all events received', async () => {
    const orderId = '123';

    await handlePaymentEvent({ orderId, paymentId: 'pay-1' });
    await handleInventoryEvent({ orderId, reservationId: 'res-1' });
    await handleChefEvent({ orderId, chefId: 'chef-1' });

    // Should trigger processing
    const state = await getOrderState(orderId);
    expect(state.status).toBe('ready');
  });

  it('should handle out-of-order events', async () => {
    const orderId = '456';

    // Events arrive in reverse order
    await handleChefEvent({ orderId, chefId: 'chef-1' });
    await handleInventoryEvent({ orderId, reservationId: 'res-1' });
    await handlePaymentEvent({ orderId, paymentId: 'pay-1' });

    const state = await getOrderState(orderId);
    expect(state.status).toBe('ready');
  });
});

```

## Chaos Testing

- What if a service is down?
  - What if events are delayed by 10 minutes?
  - What if EventBridge duplicates events?
-

# **Summary: When to Use Each Pattern**

## **Use Event Choreography when:**

- Services have clear domain boundaries
- Workflow is relatively simple or linear
- You value loose coupling and scalability
- Teams are autonomous and own their services
- **This is your default choice for most scenarios**

## **Use Event Orchestration when:**

- Complex business logic with many conditionals
- Need strong consistency guarantees
- Require detailed audit trails and workflow visibility
- Compensation and rollback are critical
- Few services involved (< 5)

## **Use Event Aggregator when:**

- One service needs data from multiple events
- Complex correlation logic
- Want to hide complexity from business services
- Multiple consumers need the aggregated result

## **Use Event-Carrying State Transfer when:**

- Read-heavy operations
- Reference data
- Services need to work independently
- Availability > Consistency

## **Use Event Sourcing when:**

- You need a complete, queryable history of changes
- Auditability and traceability are first-class requirements

- You can invest in projections and operational maturity

## Use Outbox Pattern when:

- Business state is stored in a database
  - Events must not be silently dropped on publish
  - You can leverage DB transactions or Streams
- 

## Additional Resources

### Tools for AWS + TypeScript

- **EventBridge**: Message bus for events
- **DynamoDB**: State storage with TTL
- **DynamoDB Streams**: Trigger actions on state changes
- **Lambda**: Event handlers
- **X-Ray**: Distributed tracing
- **CloudWatch**: Monitoring and dashboards

### Recommended Reading

- "Building Event-Driven Microservices" by Adam Bellemare
- "Enterprise Integration Patterns" by Gregor Hohpe
- AWS Well-Architected Framework: Event-Driven Architecture

# Example Repository Structure

```
event-driven-pizza/
├── services/
│   ├── payment-service/
│   │   ├── handler.ts
│   │   └── events.ts
│   ├── inventory-service/
│   ├── chef-service/
│   └── kitchen-service/
│       ├── handler.ts
│       ├── state-manager.ts
│       └── correlation.ts
├── shared/
│   ├── types/
│   │   └── events.ts
│   └── utils/
│       └── event-emitter.ts
└── infrastructure/
    └── eventbridge.yml
```

## Workshop Wrap-Up Discussion Questions

### 1. How would you refactor your current event-driven services?

- What data are you passing that you shouldn't?
- Which services have too much knowledge of other domains?

### 2. What challenges do you anticipate?

- Team boundaries and ownership?
- Monitoring and debugging?
- Testing strategies?

### 3. What's your action plan?

- Start with new services or refactor existing?
- What pattern makes sense for each use case?
- How will you measure success?

---

# Appendix: Quick Reference

## Event Checklist

- Uses CloudEvents specification
- Has unique event ID
- Has correlation ID (orderId, requestId, etc.)
- Event type is semantic (`domain.entity.action`)
- Contains only data the service owns
- Includes timestamp
- Is immutable
- Size < 256KB

## State Management Checklist

- Correlation ID strategy defined
- State storage chosen (DynamoDB, Redis, etc.)
- TTL/cleanup strategy implemented
- Timeout handling defined
- Idempotency handled
- Monitoring in place
- Concurrency strategy defined (optimistic locking, append-only store, etc.)

## Failure Handling Checklist

- Timeout duration defined
- Compensation logic implemented
- Failure events defined
- Dead letter queue configured
- Alerting configured
- Runbook created for operational support