



Distributed Sagas with AWS Step Functions



Melvin Jones
Solutions Architect

<https://github.com/mjones3>



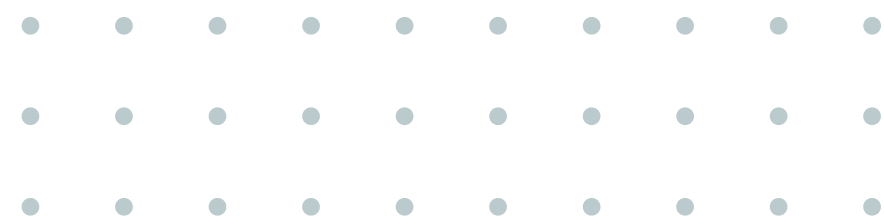
What you'll learn



01. Why sagas solve consistency in microservices

02. How to orchestrate with Step Functions & Spring Boot

03. Implementing compensations and error flows



Motivation & Concepts

Understand the limits of two-phase commits

01. Single Point of Failure:

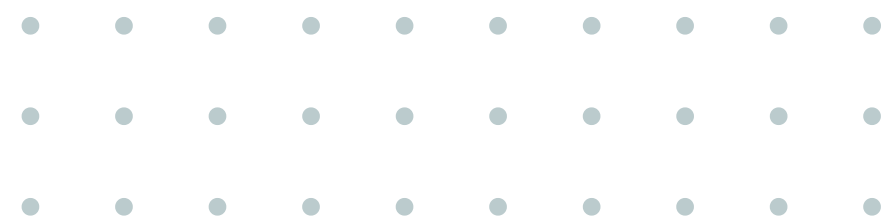
The coordinator in a two-phase commit (2PC) can become a bottleneck, making the system vulnerable if it fails.

02. Blocking:

If a participant in the commit process crashes, it can block other participants, leading to delays.

03. Scalability Issues:

Two-phase commit is not well-suited for large-scale distributed systems, as it requires synchronous communication between all participants



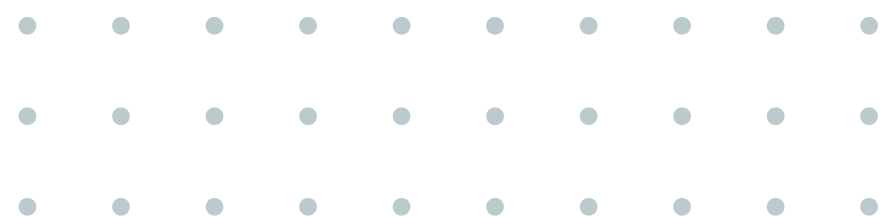
What is the Saga pattern?

(Choreography vs Orchestration)



The Saga Pattern manages long-running, distributed transactions by splitting them into smaller, isolated steps with compensations for failures.

- **Choreography:** Each service independently handles its part and triggers the next service without a central coordinator.
- **Orchestration:** A central service controls the flow of the saga and coordinates the actions of each service.

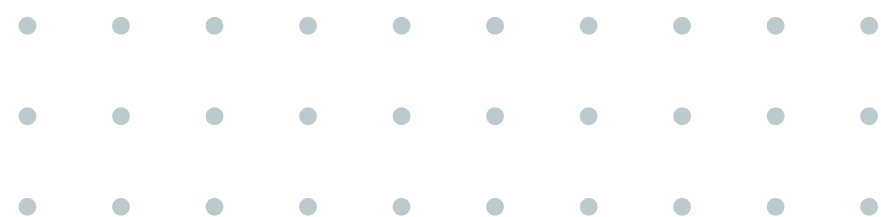


Real-world scenario:

Order → Inventory → Payment



01. **Order:** The customer places an order, triggering the Order Service to create a new order in the system.
02. **Inventory:** The Inventory Service checks stock levels, reserves the items, and updates inventory to reflect the order.
03. **Payment:** The Payment Service processes the payment; if successful, the order is confirmed and finalized.

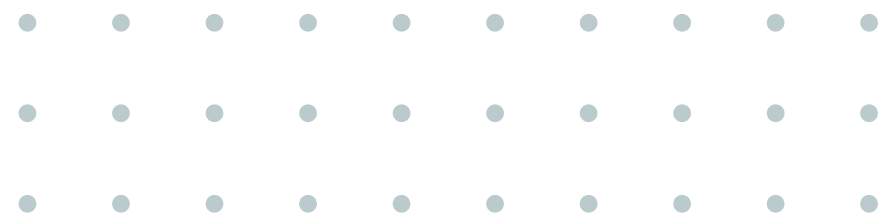


Failure Scenario (Compensations)



If the **Inventory Service** fails, the Order Service will cancel the order

If the **Payment Service** fails, the Inventory Service will roll back the stock reservation and the Order Service will cancel the order



Microservice isolation & independent data stores

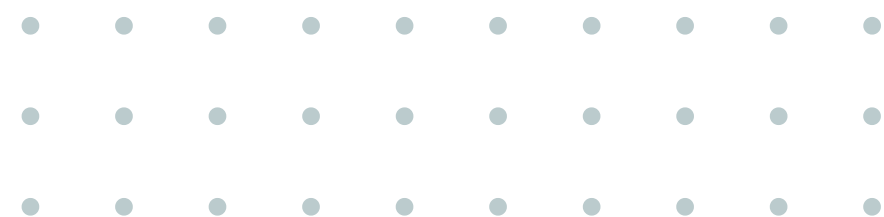


Microservice Isolation:

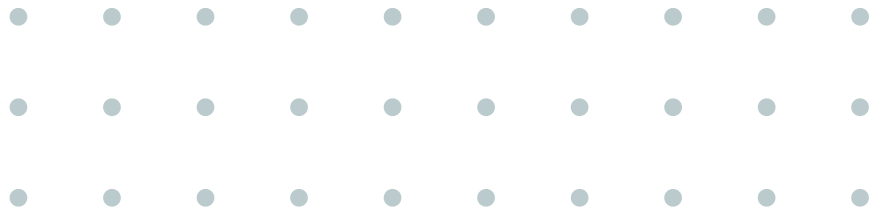
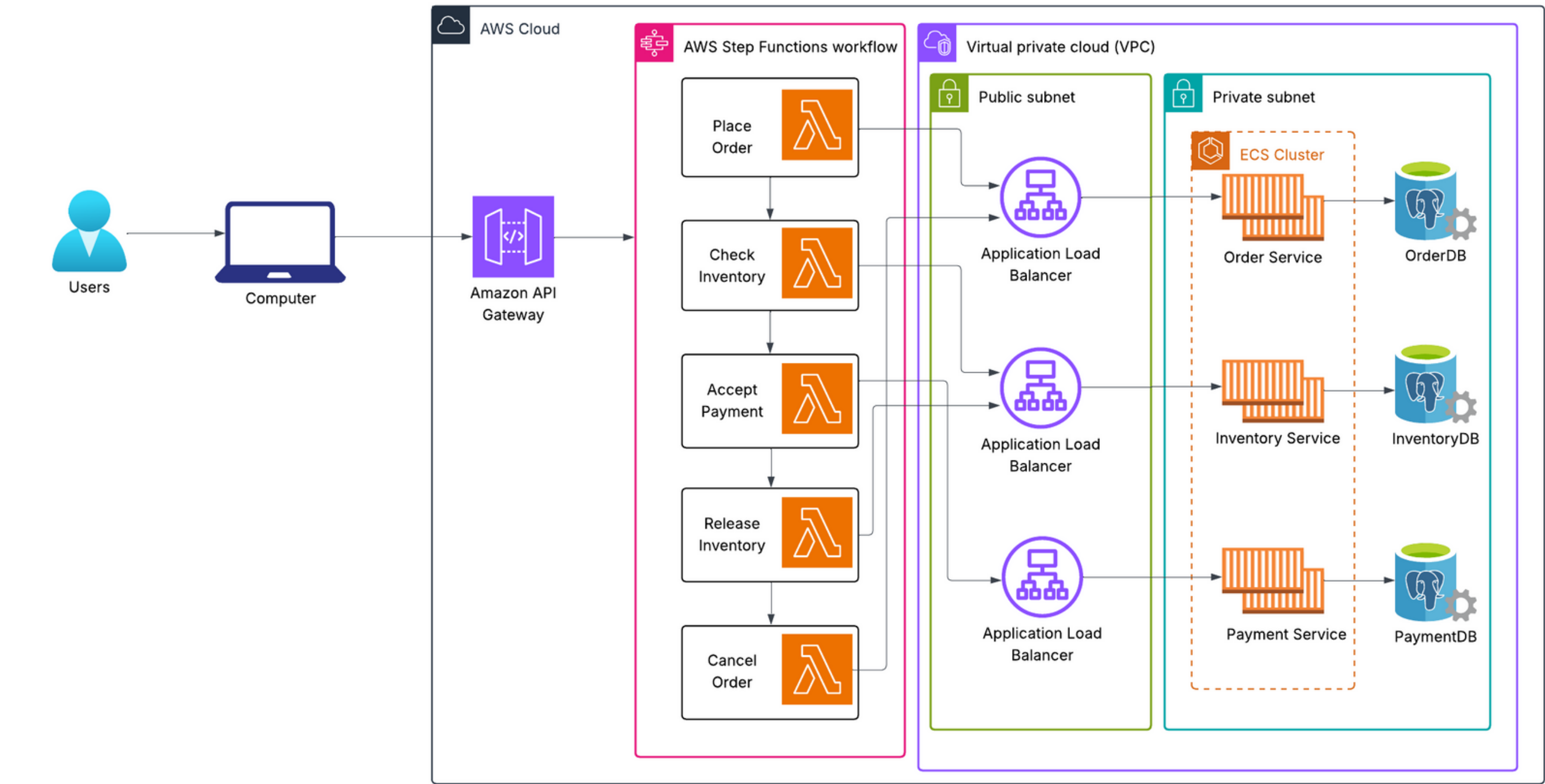
Each microservice handles a single domain, ensuring clear separation and scalability

Independent Data Stores:

Each service owns its own data store, promoting decoupling and independent evolution.



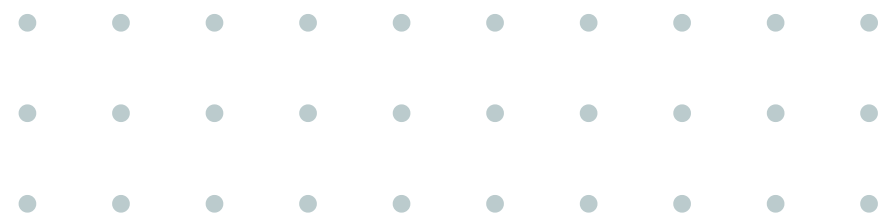
System Architecture



Building the Spring Boot Microservices



01. Create Order, Inventory, Payment services with JPA & REST
02. Expose compensation endpoints (/cancel, /release)



Code snippets for @RestController mapping



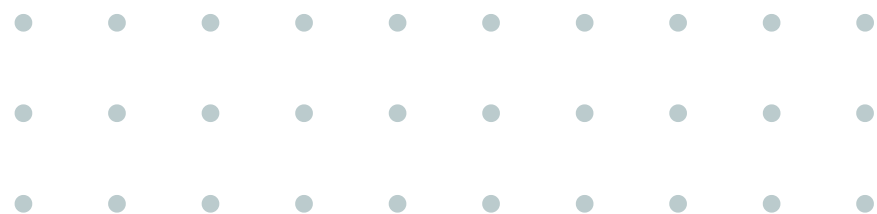
```
@PostMapping("/orders/create")
public ResponseEntity<OrderResponse> createOrder(@RequestBody OrderRequest req) {

    logger.info(req);

    Order order = orderService.createOrder(req);
    OrderResponse response = new OrderResponse(order);

    logger.info(response);

    return new ResponseEntity<>(response, HttpStatus.CREATED);
}
```

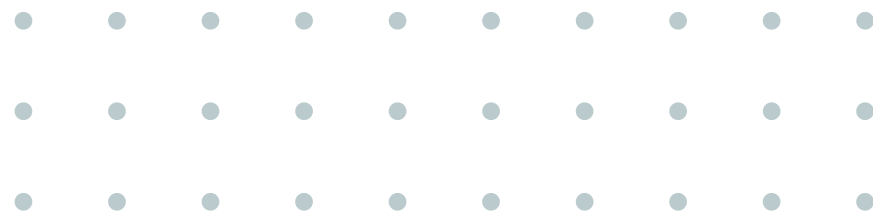


Code snippets for @RestController mapping



```
@PostMapping("/orders/{id}/cancel")
public ResponseEntity<Order> cancelOrder(@PathVariable("id") Long orderId) {

    Optional<Order> order = orderService.cancelOrder(orderId);
    if (order.isPresent()) {
        return new ResponseEntity<>(order.get(), HttpStatus.OK);
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```



Defining the State Machine

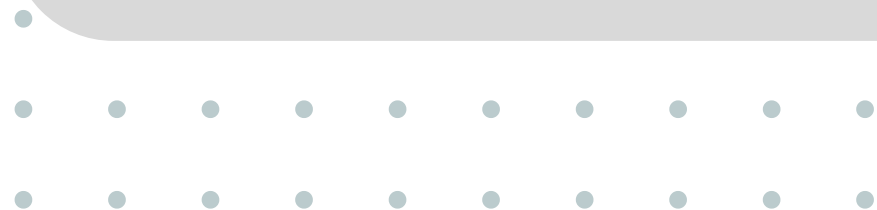


01. Author Amazon States Language (ASL) JSON
02. Use Choice states for business results

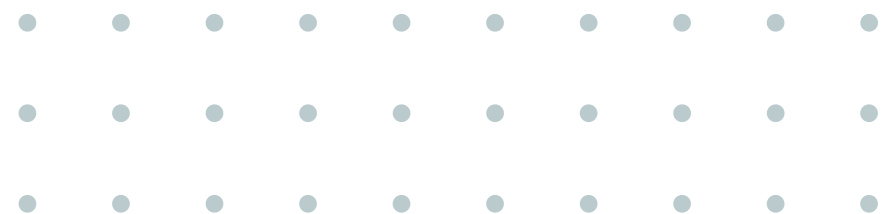
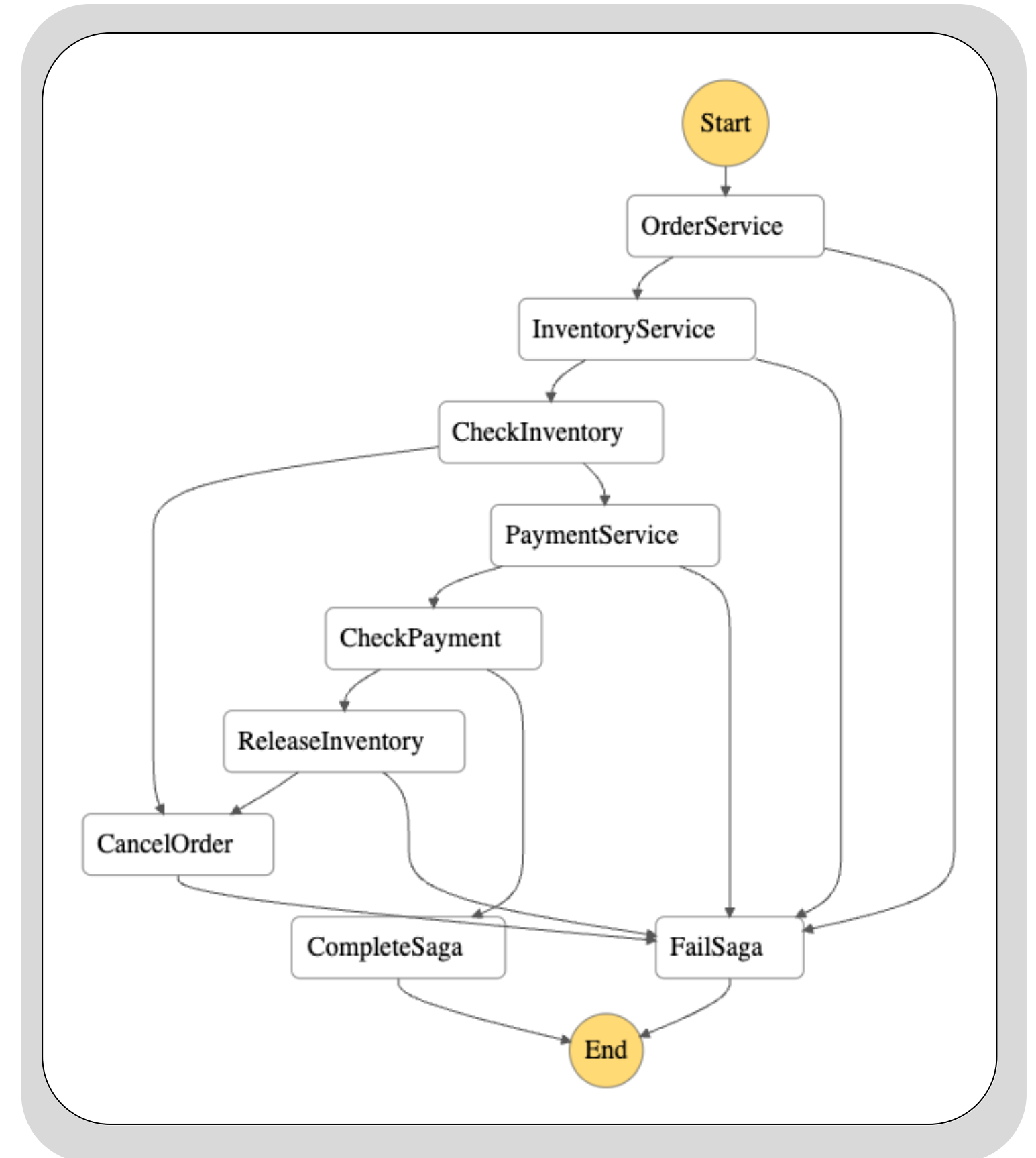
Amazon States Language (ASL) JSON



```
{
  "Comment": "State machine for processing the order saga with Choice states",
  "StartAt": "OrderService",
  "States": {
    "OrderService": {
      "Type": "Task",
      "Resource": "arn:aws:states:::lambda:invoke",
      "Parameters": {
        "FunctionName": "${aws_lambda_function.order_service.arn}",
        "Payload": { "input.$": "$" }
      },
      "Catch": [
        {
          "ErrorEquals": ["States.ALL"],
          "Next": "FailSaga"
        }
      ],
      "Next": "InventoryService"
    },
  },
}
```



State Machine Diagram



Compensation & Error Handling

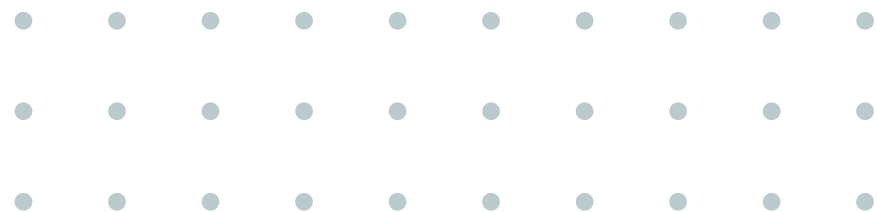


01. Implement compensating actions in microservices
02. Catch unexpected Lambda errors

Order POST /cancel example

CheckInventory choice transitions to **CancelOrder** if **InventoryService** returns a 404

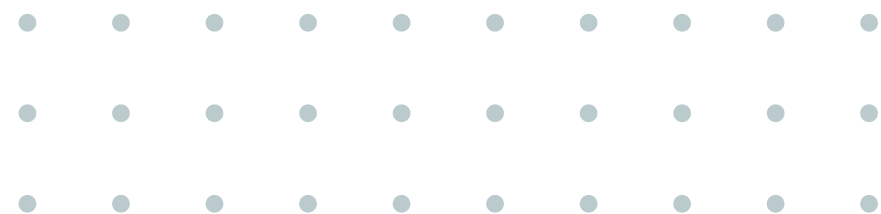
```
"CheckInventory": {  
  "Type": "Choice",  
  "Choices": [  
    {  
      "Variable": "$.Payload.statusCode",  
      "NumericEquals": 404,  
      "Next": "CancelOrder"  
    }  
  ],  
  "Default": "PaymentService"  
},
```



Inventory POST /release example

CheckPayment Choice state transitions to
ReleaseInventory if 402 is received from PaymentService

```
"CheckPayment": {  
  "Type": "Choice",  
  "Choices": [  
    {  
      "Variable": "$.Payload.statusCode",  
      "NumericEquals": 402,  
      "Next": "ReleaseInventory"  
    }  
  ],  
  "Default": "CompleteSaga"  
},
```



Inventory Service: POST /release example

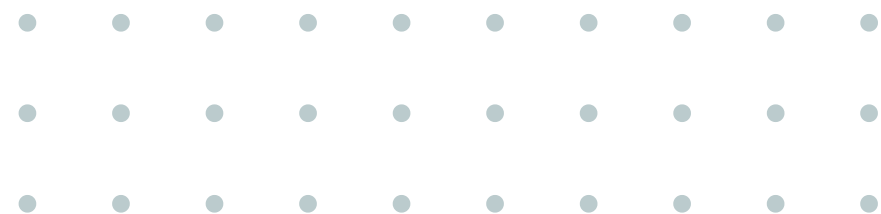
Release inventory compensation

```
{
  "orderId": 5,
  "items": [
    {
      "orderId": 9,
      "productId": "A00003",
      "releaseQuantity": 2
    },
    {
      "orderId": 10,
      "productId": "A00004",
      "releaseQuantity": 2
    }
  ]
}
```

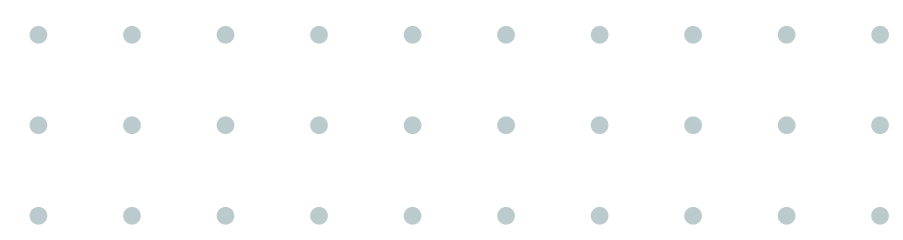
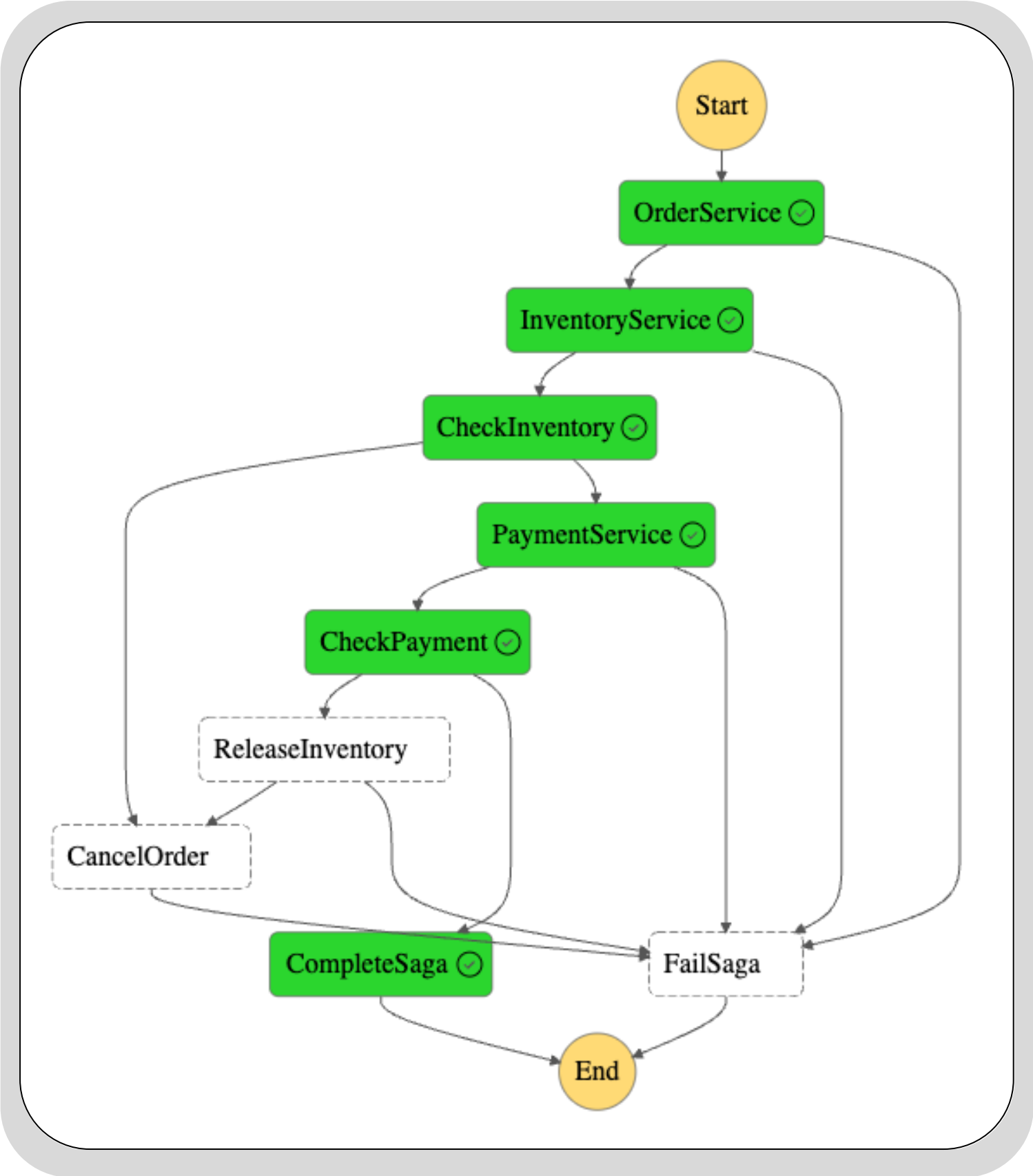
Order Service: POST /cancel example

Cancel order compensation

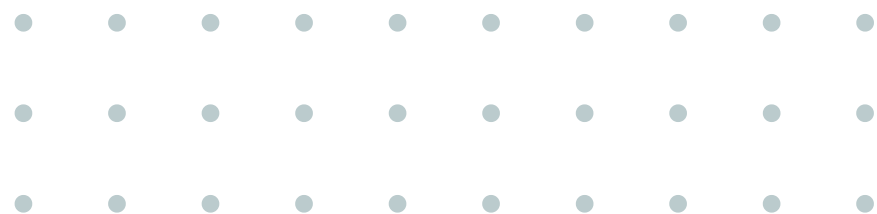
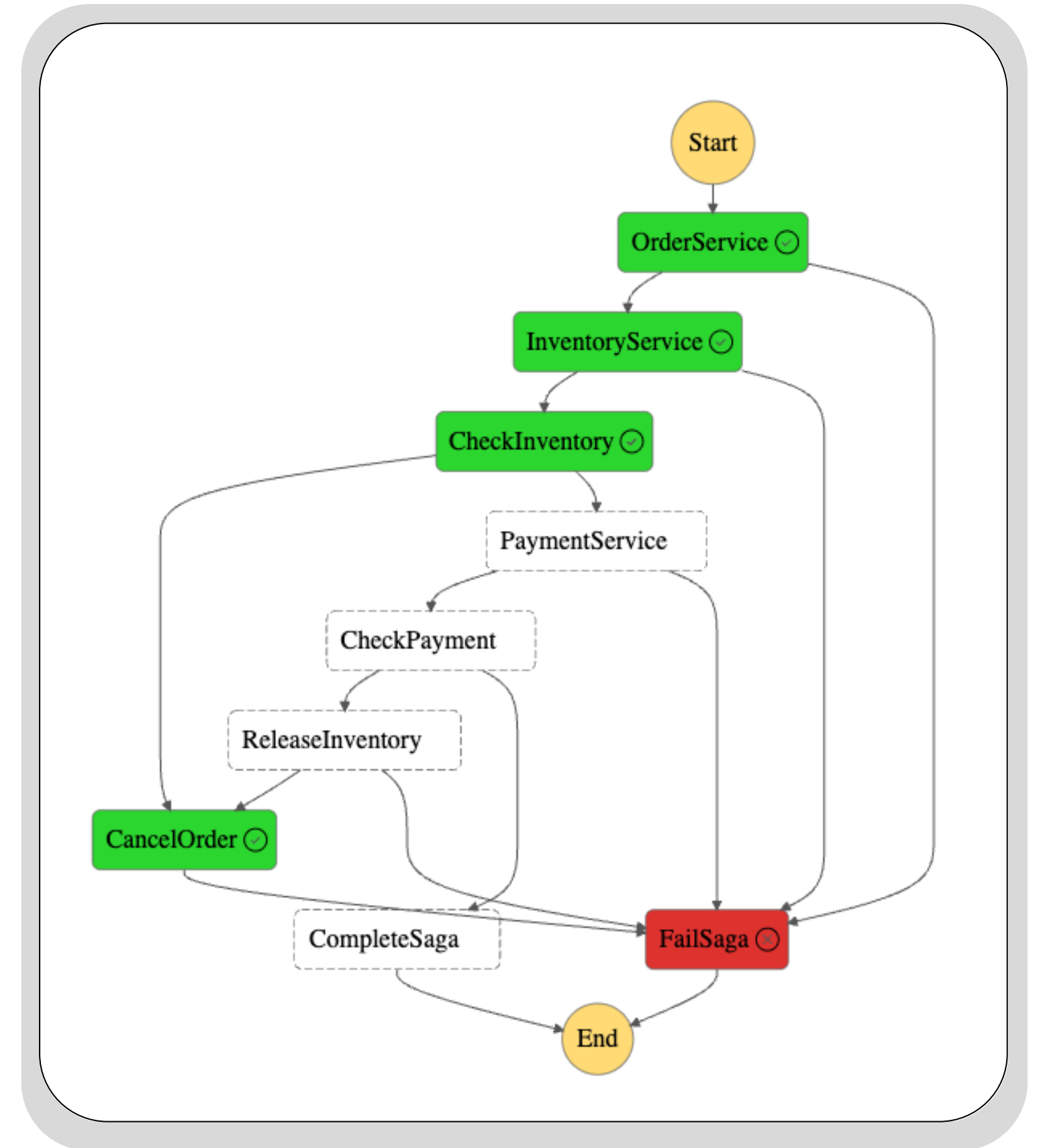
```
{  
  "orderId": 5  
}
```



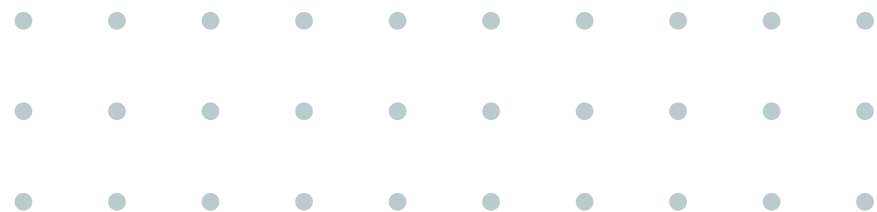
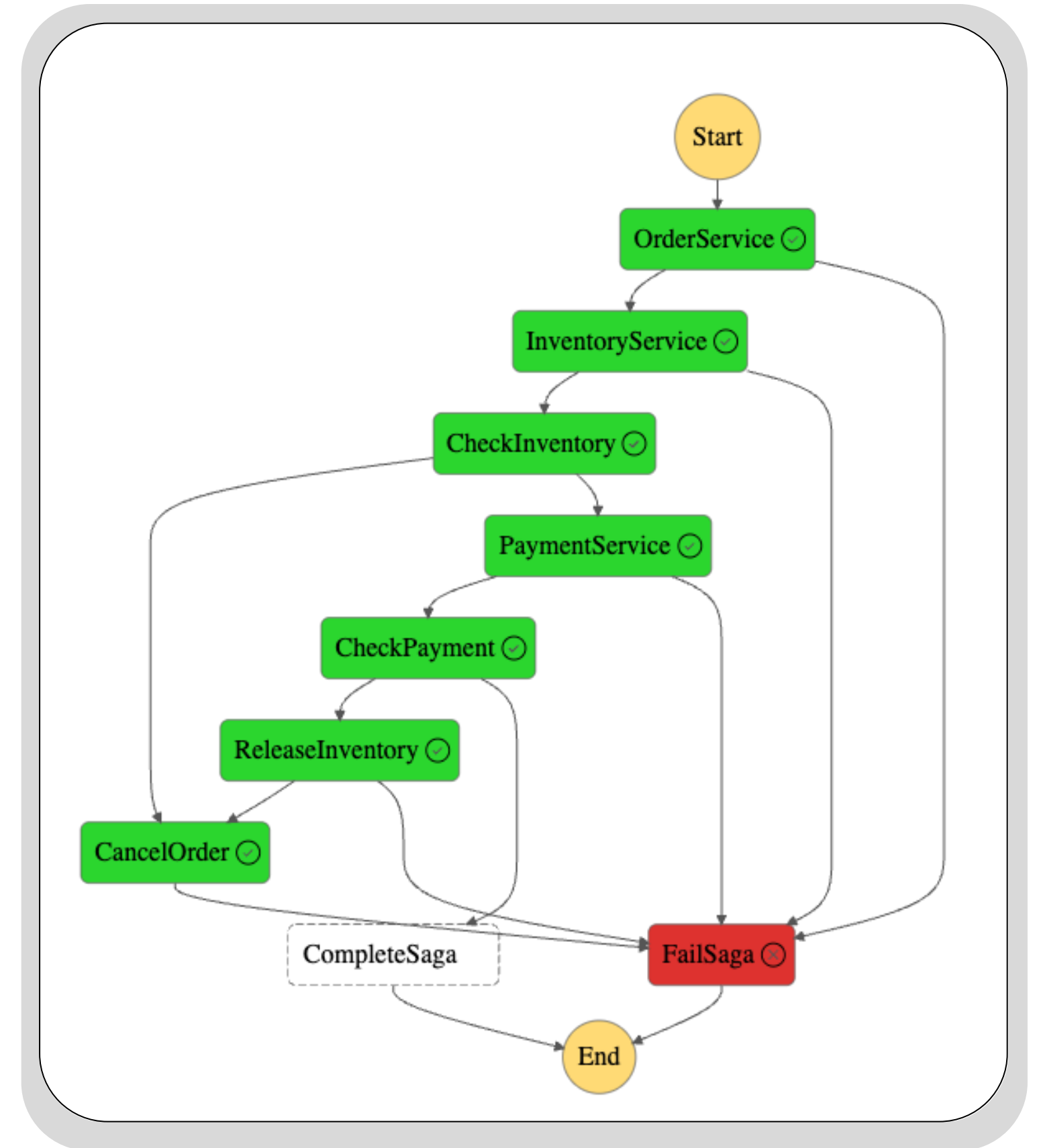
Step Function Flow: Approved Payment



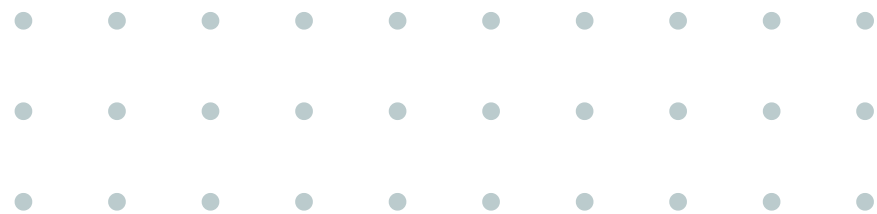
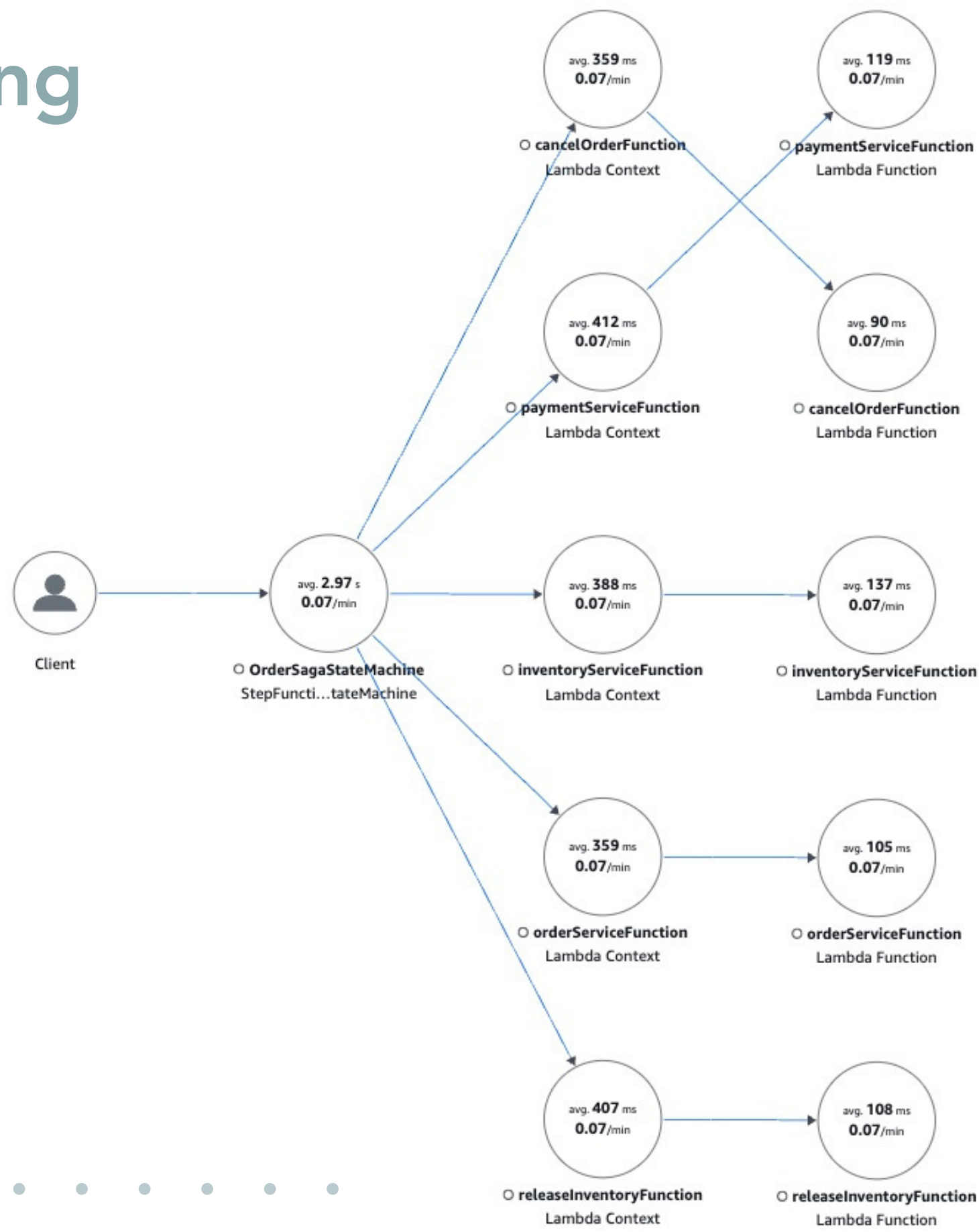
Step Function Flow: Insufficient Inventory



Step Function Flow: Failed Payment



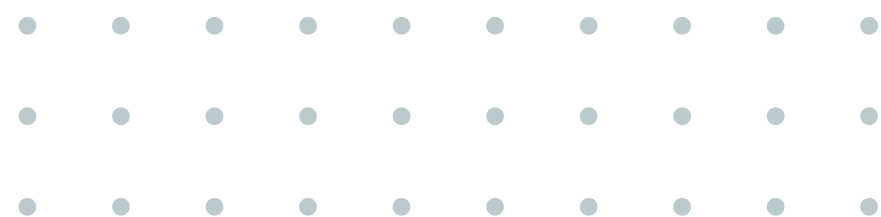
X-Ray Tracing



Key Takeaways



- 01. Sagas for data consistency without locks
- 02. Clear separation: Choice for business, Catch for errors
- 03. Two terminal states: Succeed vs Fail

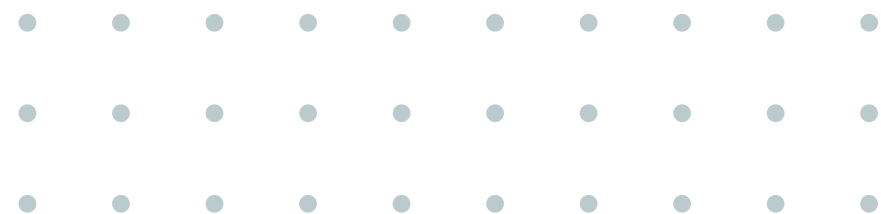


Dos & Don'ts:

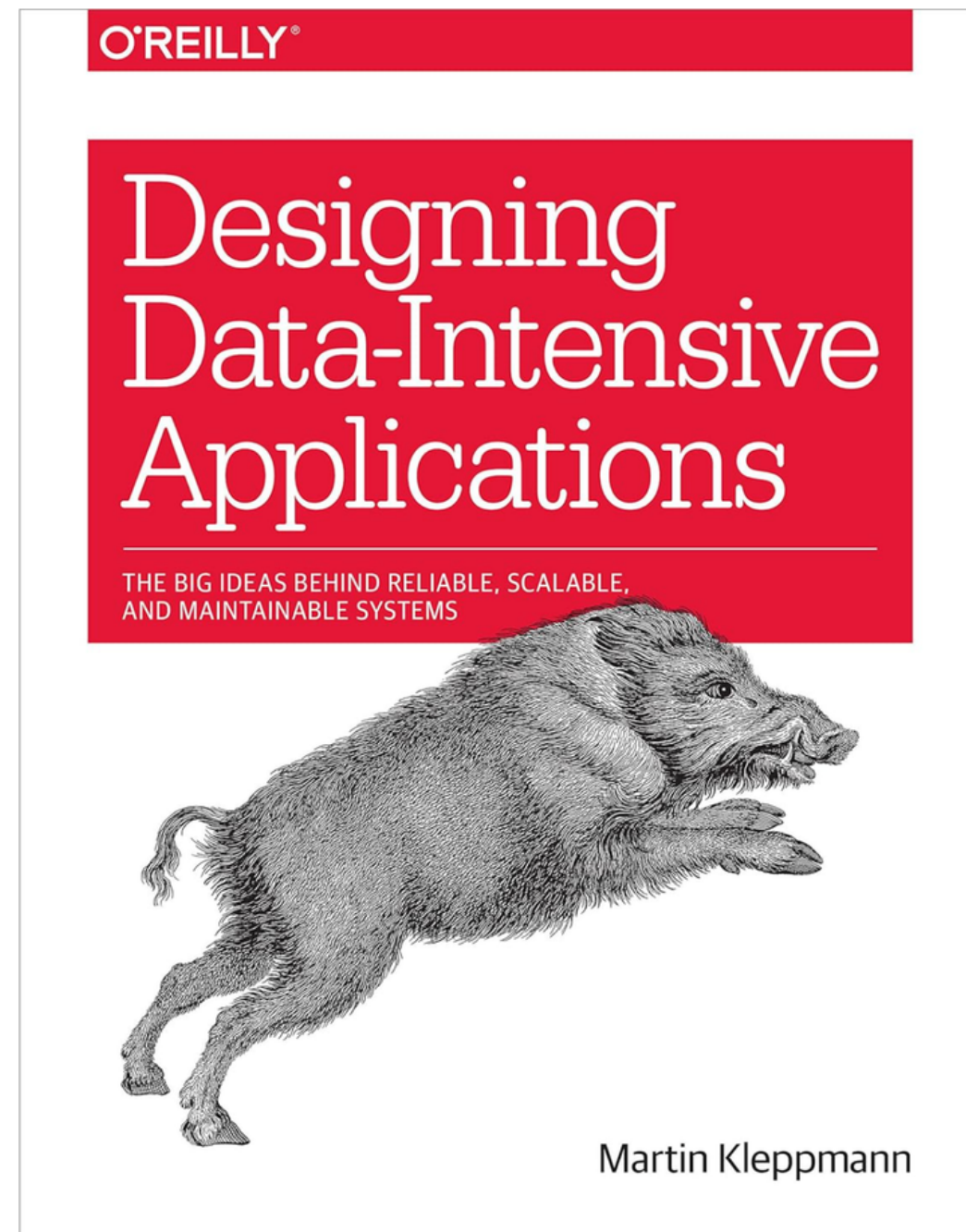


01. Do use idempotent compensations

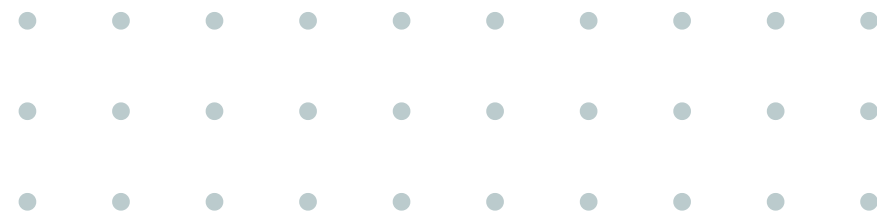
02. Don't use exceptions for expected flows



Further reading



Designing Data-Intensive Applications (Martin Kleppmann)



Download Source



<https://github.com/mjones3/order-system>

