AWS re:Invent

DAT306-R

Implement microservice architectures with Amazon DynamoDB and AWS Lambda

Edin Zulich

NoSQL Solutions Architect

Amazon Web Services

AWS re:Invent

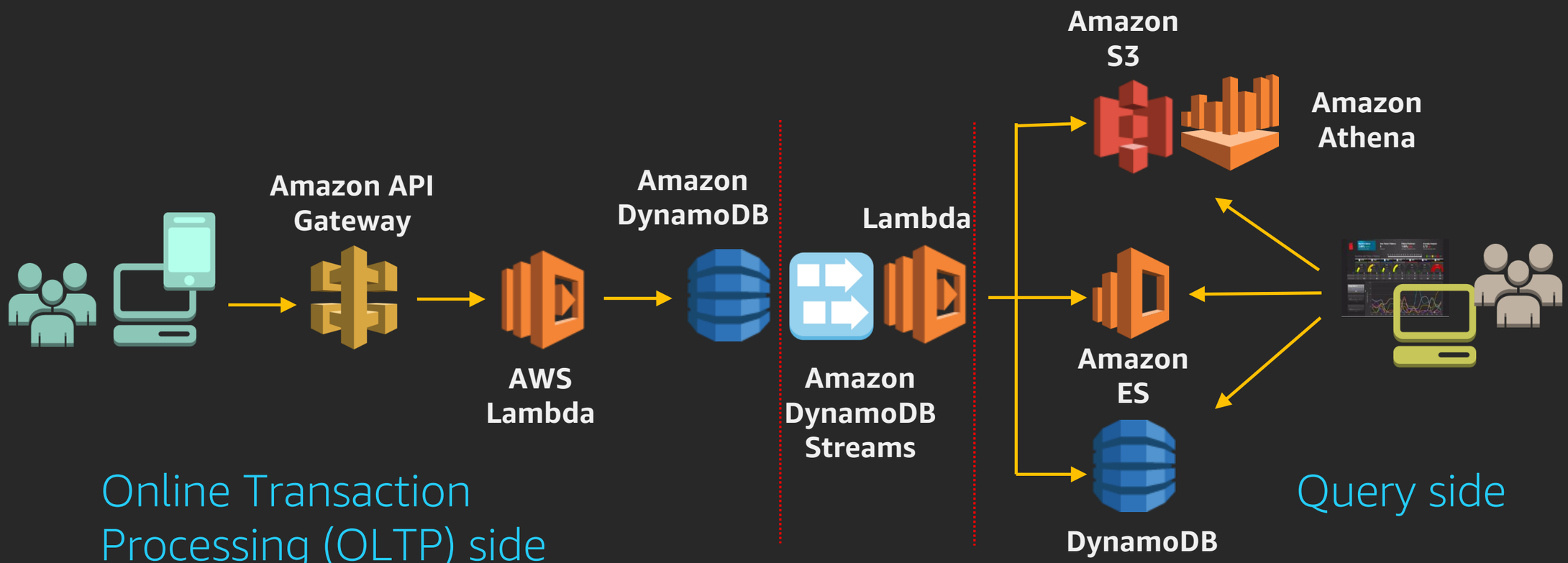
aws

Agenda

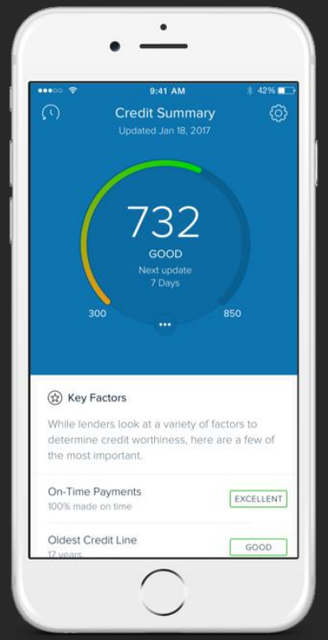
- Building blocks for serverless microservices
- Challenges with microservices
- Transactions in a distributed microservices architecture
- Enabling querying in a distributed microservices architecture

Example

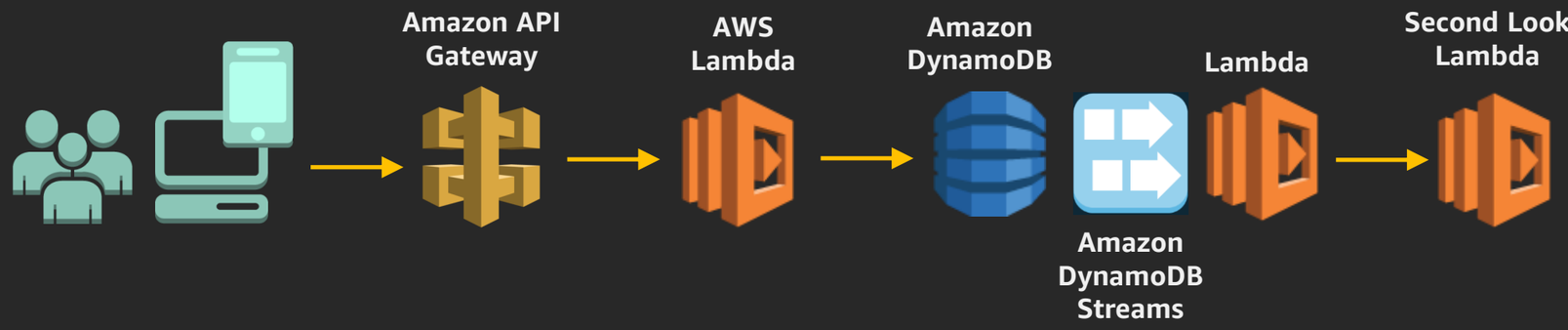
A serverless app



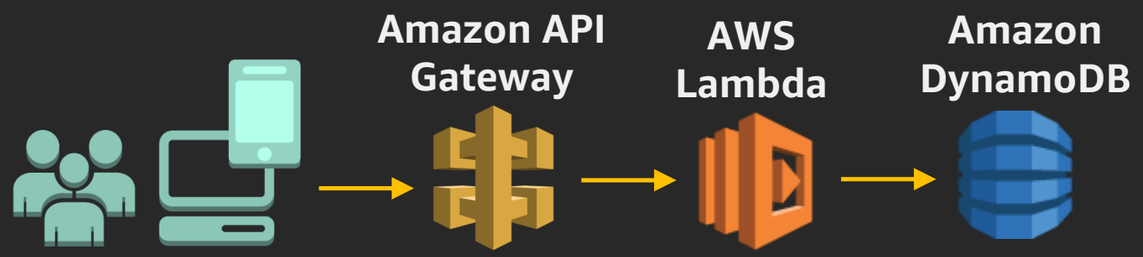
Second Look feature

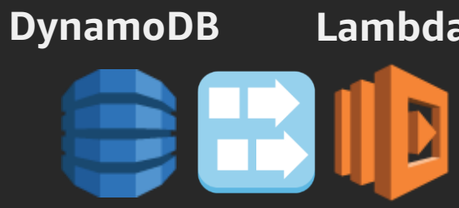


- Enabled streams on the DynamoDB transactions table
- Lambda polls the streams and delivers new events to the Lambda function
- This Lambda function invokes the Lambda function for a feature called Second Look
- Second Look Lambda function alerts the customer about certain types of transactions
- It evaluates approximately 10 million transactions per day



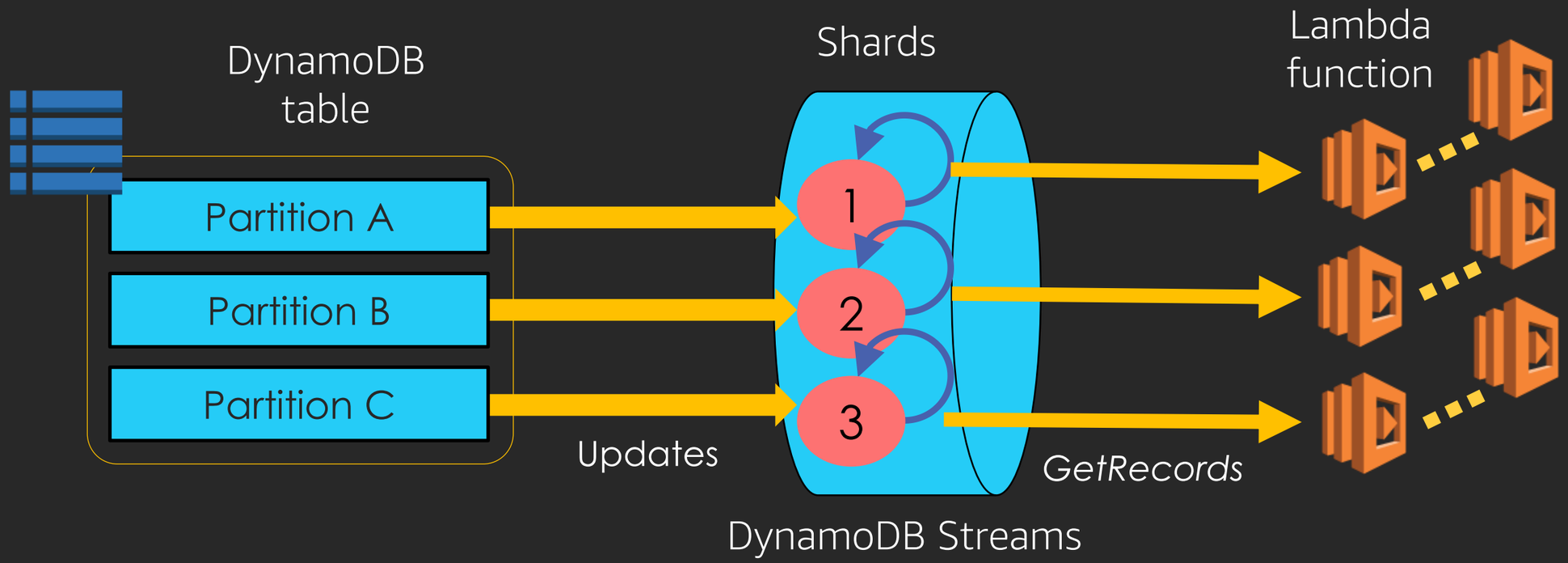
Building blocks





1. API Gateway + Lambda + DynamoDB: core building block
2. DynamoDB Streams + AWS Lambda: building block for reliable event delivery

DynamoDB Streams and Lambda



DynamoDB Streams + AWS Lambda = reliable “at least once” event delivery

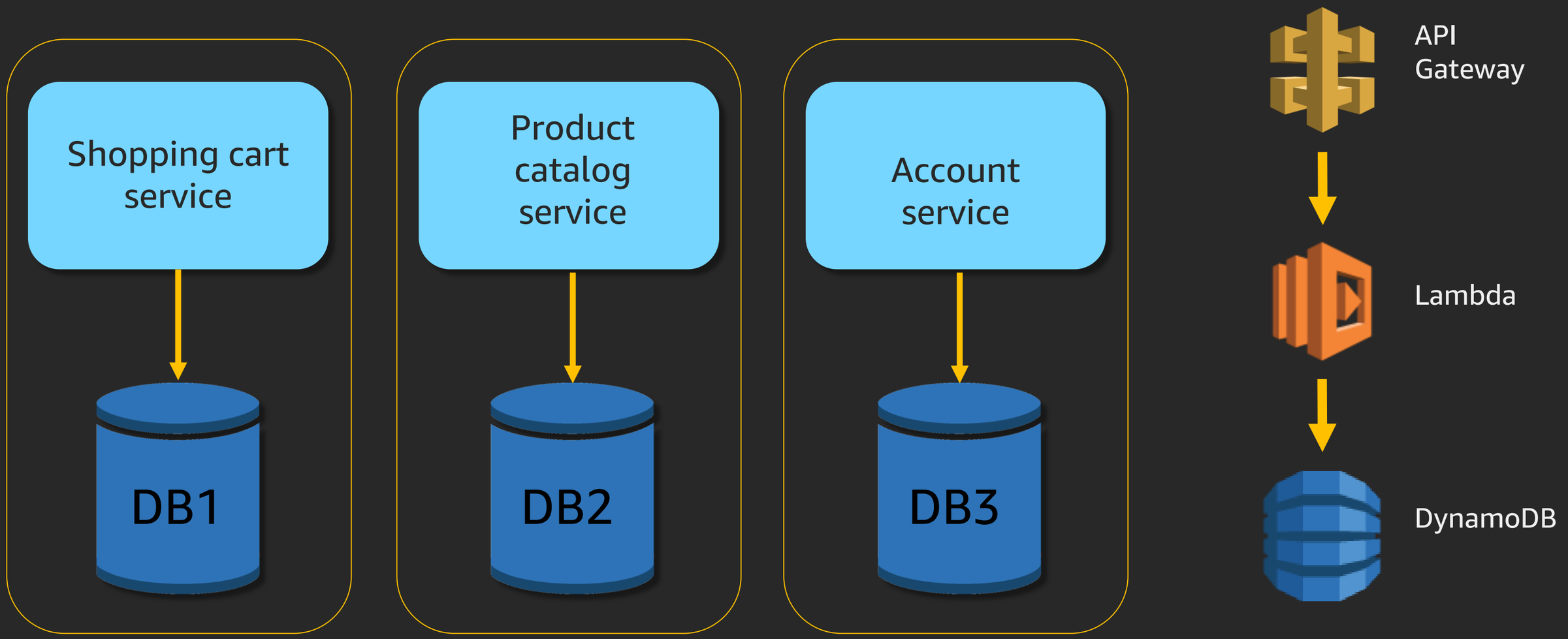
Stream sources and Lambda functions

- By default, Lambda creates a function instance per stream shard
- There is no guarantee that compute bandwidth will match the stream throughput
    - There is a possibility of impedance mismatch
- Tuning options
    - Tune your function to execute faster
        - Optimize for Lambda execution (container reuse)
        - Monitor Lambda memory (and indirectly CPU)
    - Adjust Lambda batch size
    - Increase Lambda Parallelization Factor
    - Increase the number of shards

Example: transactions and querying

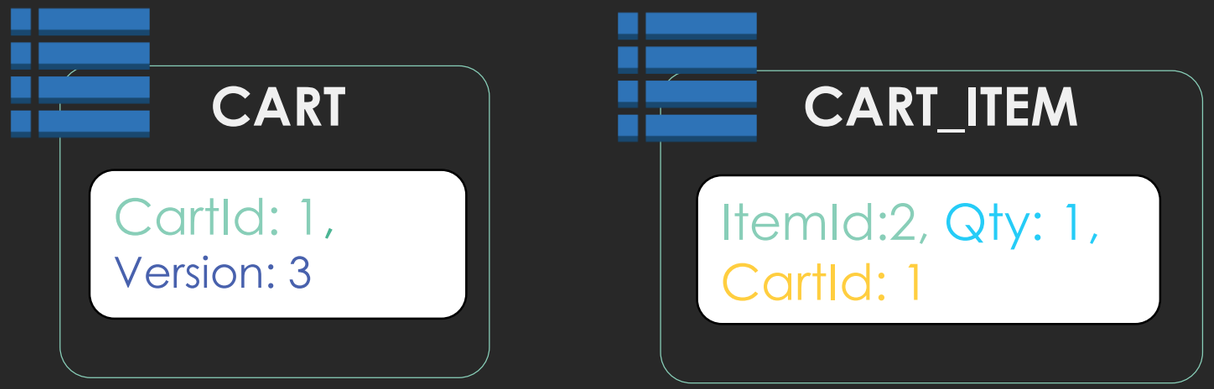
- An ecommerce app
- Relational vs. nonrelational data models
- Transactions in a distributed microservices architecture
- Enabling querying in a distributed microservices architecture

Example: ecommerce app using microservices



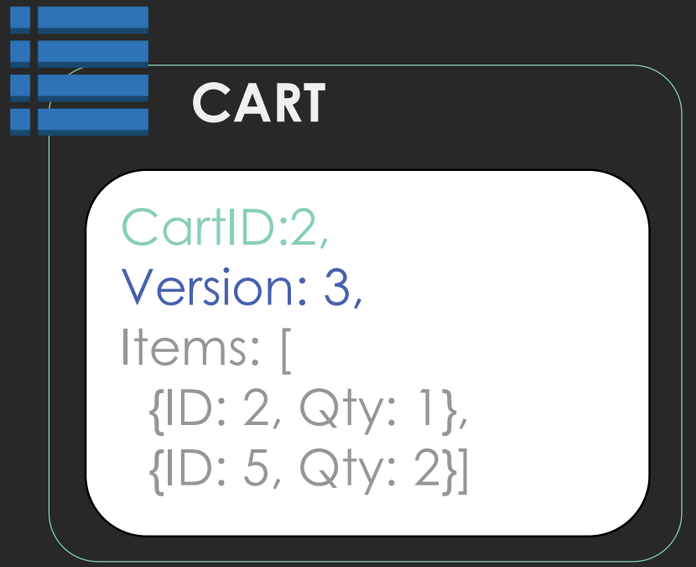
Shopping cart data model

Relational database



Normalized schema: multiple tables

Nonrelational database



Denormalized schema: aggregates in a single table

Data consistency on cart updates

Example: add/remove shopping cart items

1. get cart => v_{read} = Version
2. update cart:
   IF Version = v_{read}
      add/remove cart items
      ++Version
   ELSE go back to Step 1.

Use **ConditionExpression** in DynamoDB

Optimistic concurrency control (OCC)

Updating cart: data consistency using OCC

1. Get the cart: **GetItem**

```
{   "TableName": “Cart”,
    "Key": {“CartId”: {“N”: “2”}}
}
```

CART

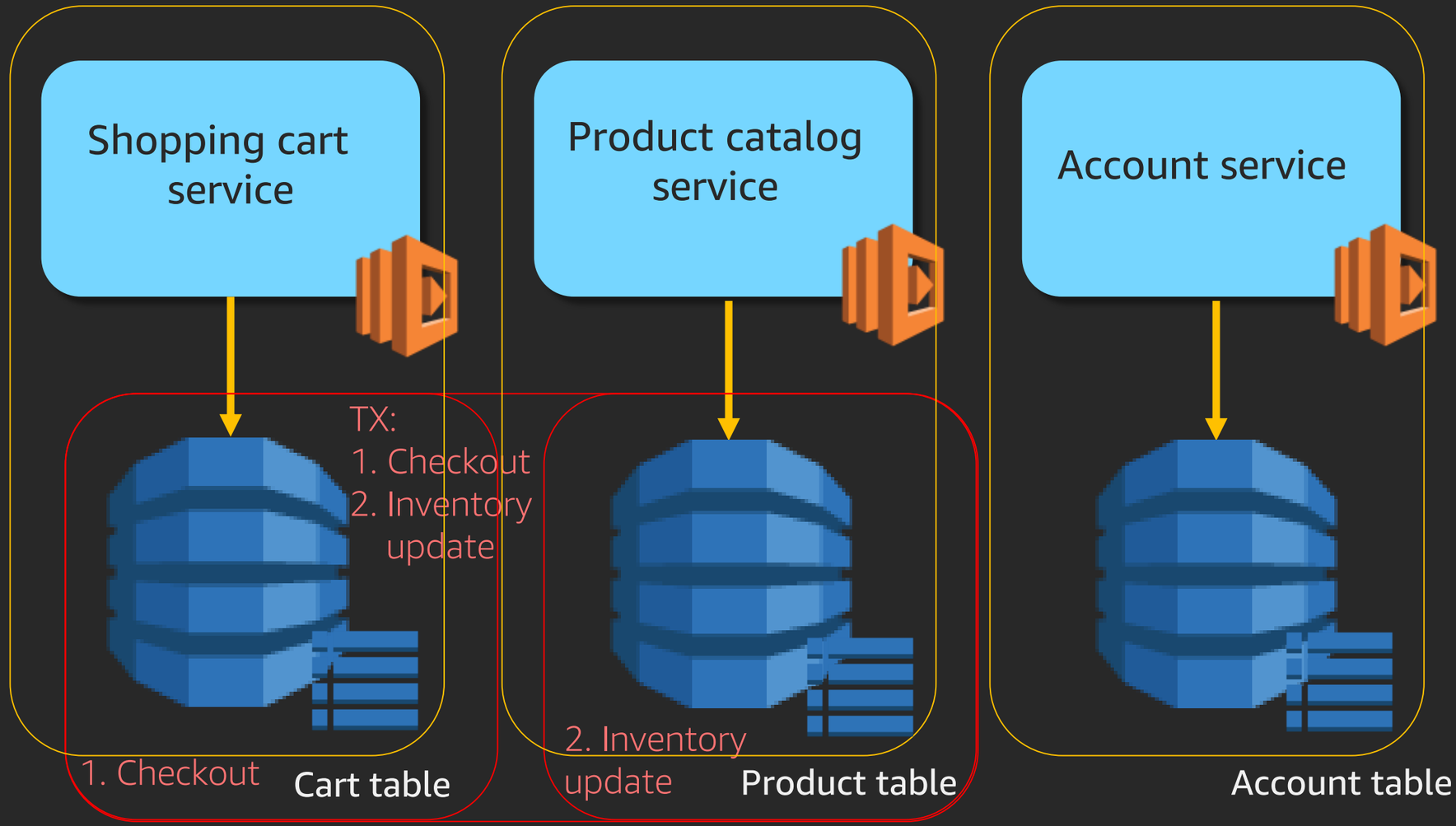
CartID:2,
Version: 3,
CartItems: [
{ID: 2, Qty: 1},
{ID: 5, Qty: 2}]

2. Update the cart: **conditional PutItem (or UpdateItem)**

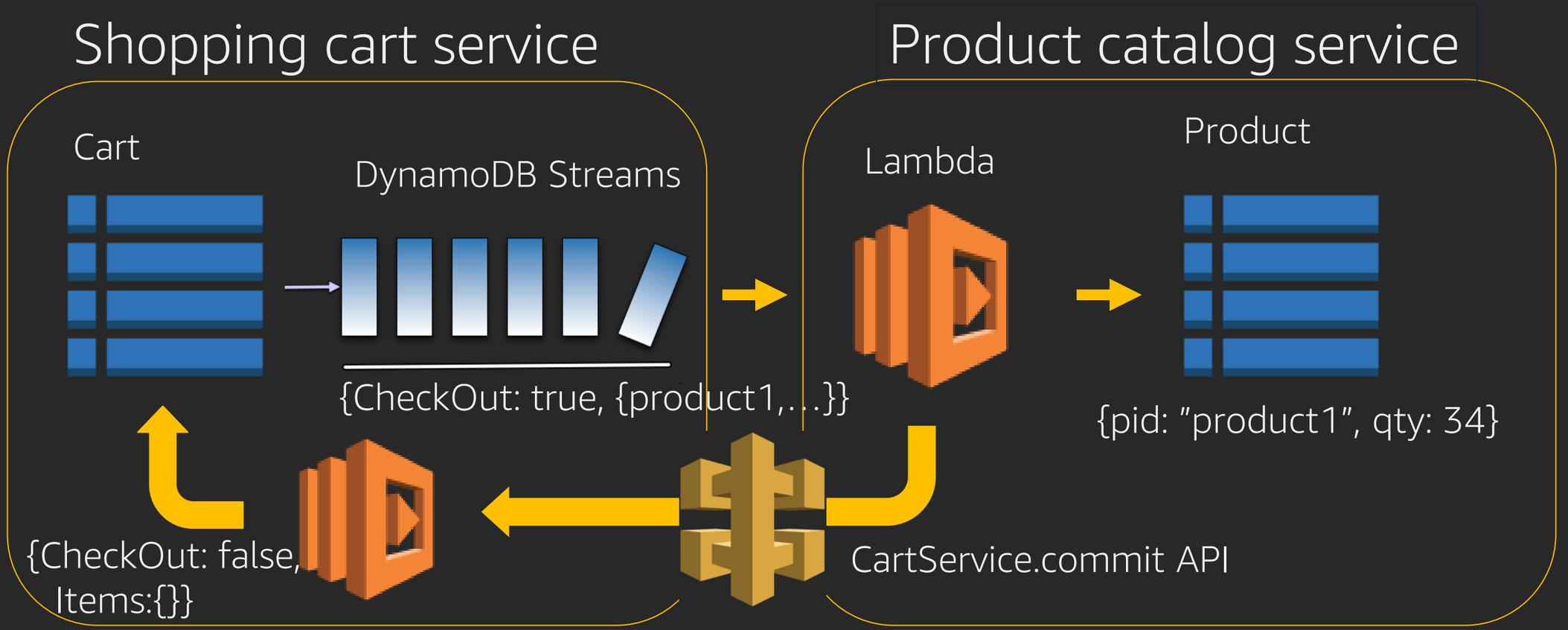
```
{   "TableName": “Cart”,
    "Item": {
        “CartID”: {“N”: “2”},
         “Version”: {“N”:”4”},
          “CartItems”: {…}
    },
    "ConditionExpression": “Version = :v1”,
    "ExpressionAttributeValues": {":v1": {“N”: “3”}}
}
```

- ✓ Use conditions to implement OCC, ensuring data consistency
- ✓ GetItem call can eventually be consistent

Transactions that span microservices

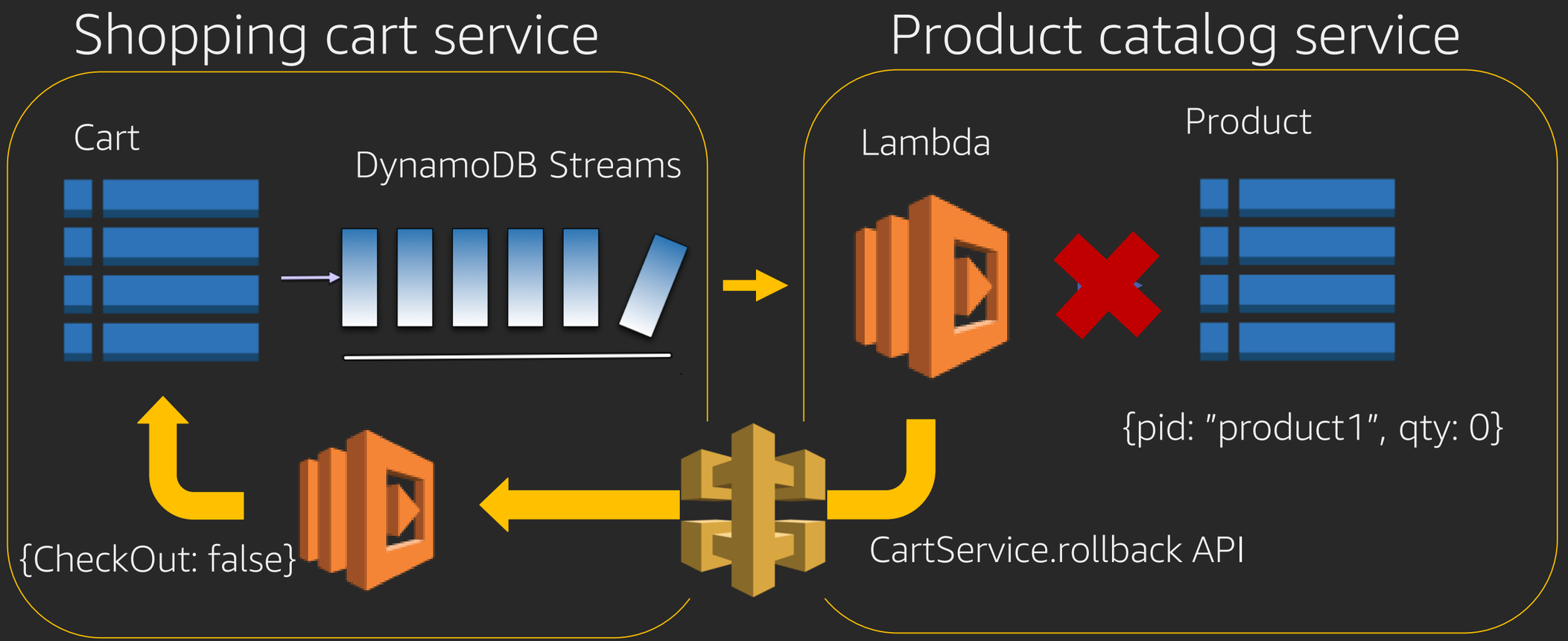


Transaction: cart checkout



Pattern: choreography-based saga using DynamoDB Streams and Lambda

Transaction: cart checkout failure



Cart table update scenarios using OCC

1. Add/remove item
    1. get cart => v_{read} = version
    2. update cart
       add/remove IF (version = v_{read} AND CheckOut = false)
2. Checkout begin
    1. get cart => v_{read} = version
    2. update cart
       SET CheckOut = true IF (version = v_{read} AND CheckOut = false)
3. Checkout commit
    1. get cart => v_{read} = version
    2. update cart
       SET items={}, CheckOut = false IF (version = v_{read} AND CheckOut = true)
4. Checkout rollback
    1. get cart => v_{read} = version
    2. update cart
       SET CheckOut = false IF (version = v_{read} AND CheckOut = true)

ProductTable update scenario

With DynamoDB transactions API

{

1. update the total quantity AND
2. insert event (ADD or SUB)
   if the event does not exist.

}

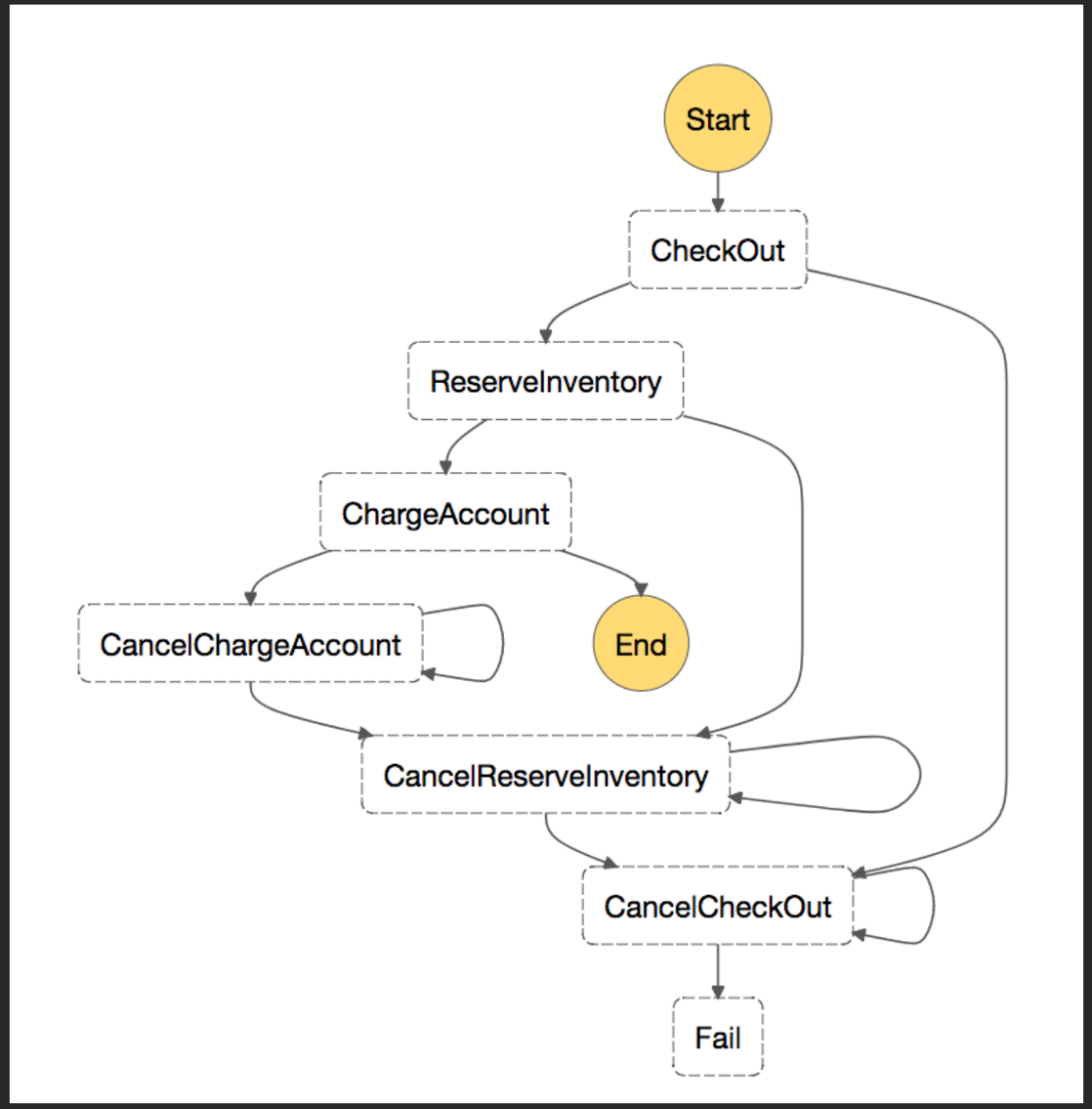
Idempotency

ProductCatalog				
Primary key		**Attributes**		
ProductID	**TranID**			
SKU1	0	Qty	…	LastUpdate
		1234		1543426328
	uuid1	Action	Qty	DateTime
		SUB	2	1543426328
	uuid2	Action	Qty	DateTime
		SUB	1	1543426241
	uuid3	Action	Qty	DateTime
			200	1543425732
SKU2	0	Qty	…	LastUpdate
		583		1543426712
	uuid4	Action	Qty	DateTime
		SUB	3	1543426712
	uuid5	Action	Qty	DateTime
			1	1543425896

Another building block for transactions: AWS Step Functions

- Define steps as a state machine
- Each step is implemented as a Lambda function
- Lambda functions have to be idempotent
- This approach can be combined with the DynamoDB Streams–based approach

Pattern: orchestration-based saga using Step Functions



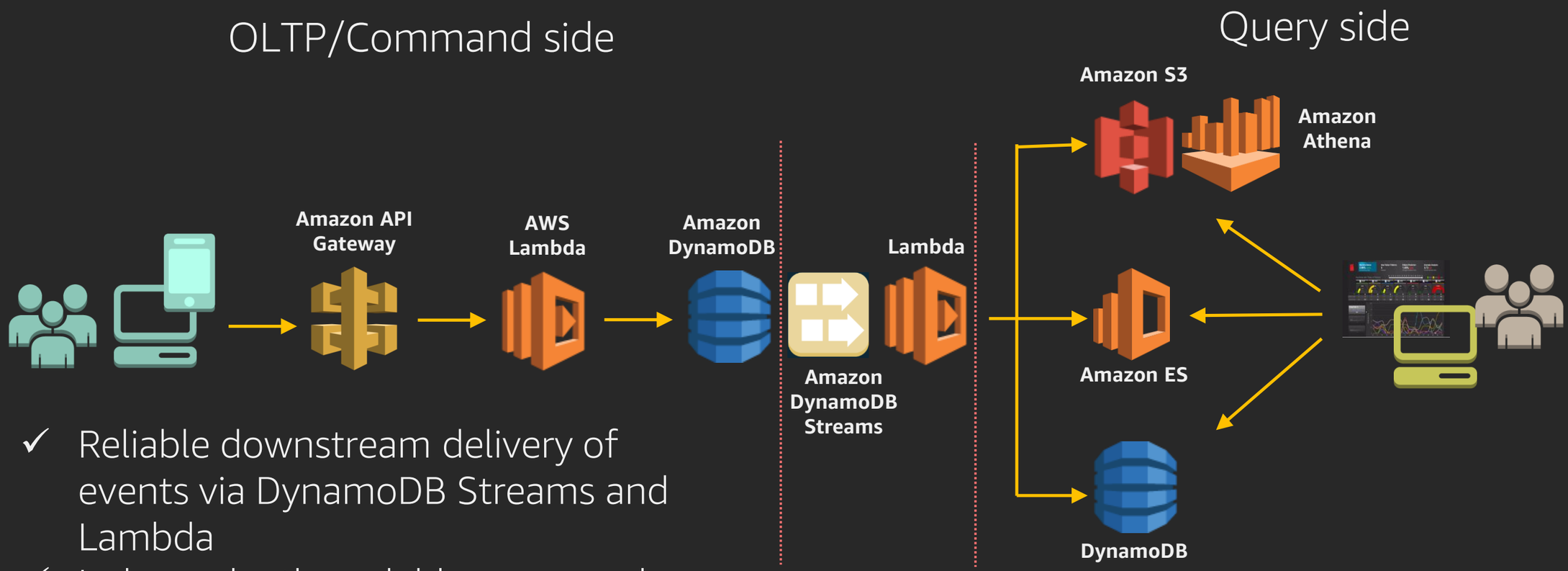
Transactions with microservices: takeaways

- Requirements
  - Service automatically updates database and publishes events
  - Events delivered at least once
- DynamoDB Streams + Lambda approach satisfies both
- Denormalized schema (aggregates)
- Building blocks: DynamoDB Streams + Lambda and Step Functions
- Asynchronous and eventually consistent
- Each service publishes its rollback method
- Lambda functions have to be idempotent
- Increased total transaction latency

Querying in microservices architectures

- Challenges
    - Data is segregated in different microservices' databases
    - Operational view of data does not satisfy querying needs
- Solution
    - Separate the operational and querying views
    - Polyglot persistence: use the right database for the job
    - Command Query Responsibility Segregation (CQRS)

Enabling querying with microservices

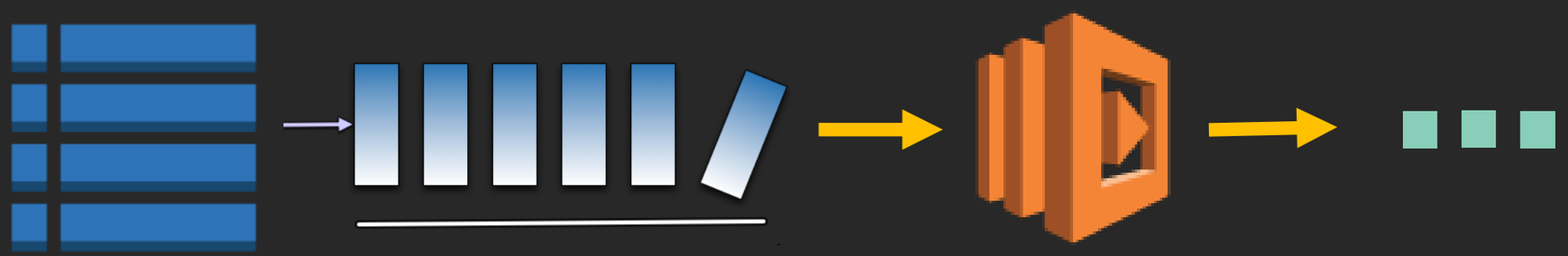


- ✓ Reliable downstream delivery of events via DynamoDB Streams and Lambda
- ✓ Independently scalable command and query sides

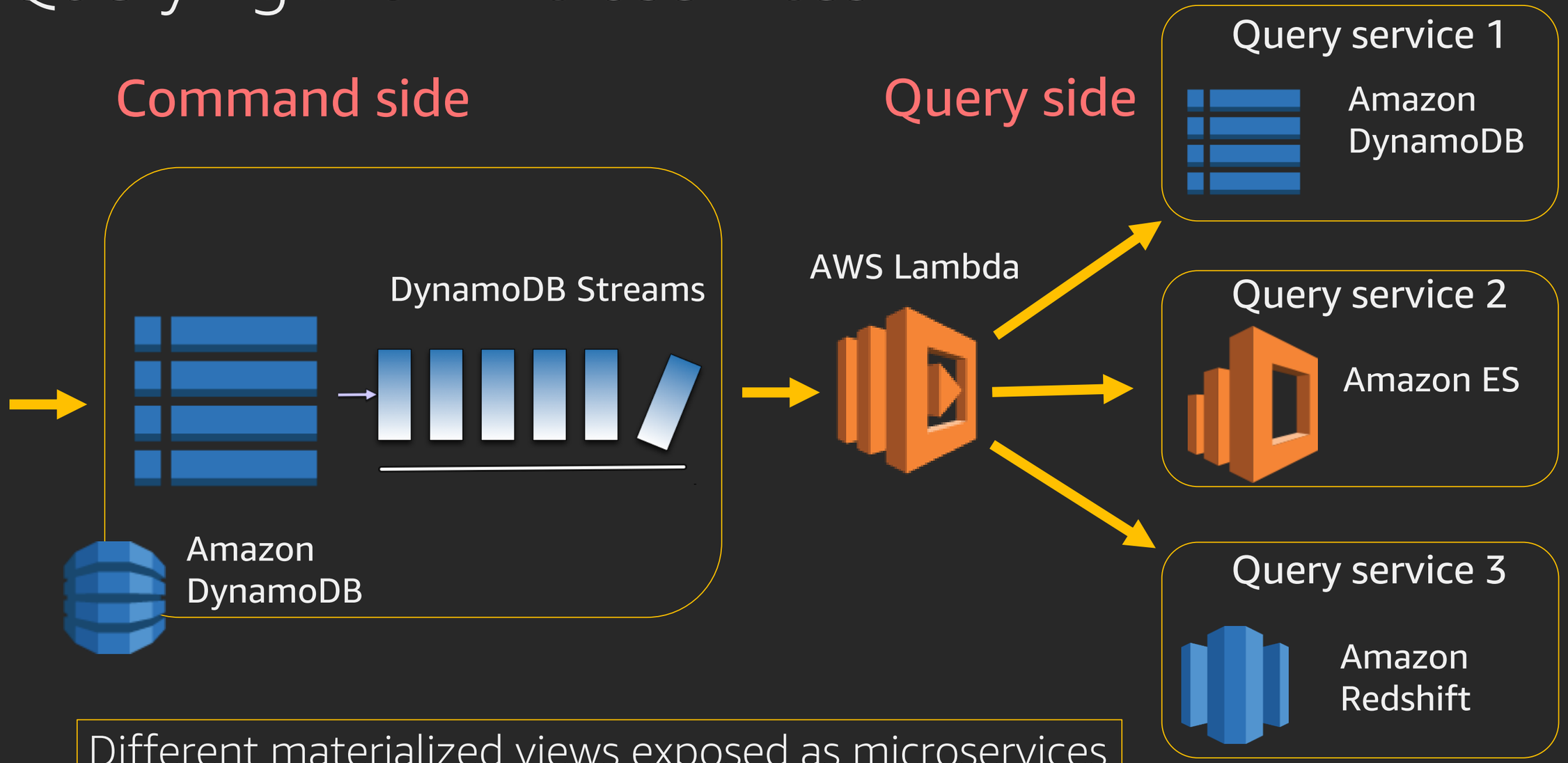
Command Query Responsibility Segregation

Complex queries and analytics

- Use the service that best meets the requirements
    - Amazon Athena, Amazon Redshift, Amazon Elasticsearch Service
- Deliver data updates reliably with DynamoDB Streams
    - Integrates with Lambda
    - End-to-end serverless

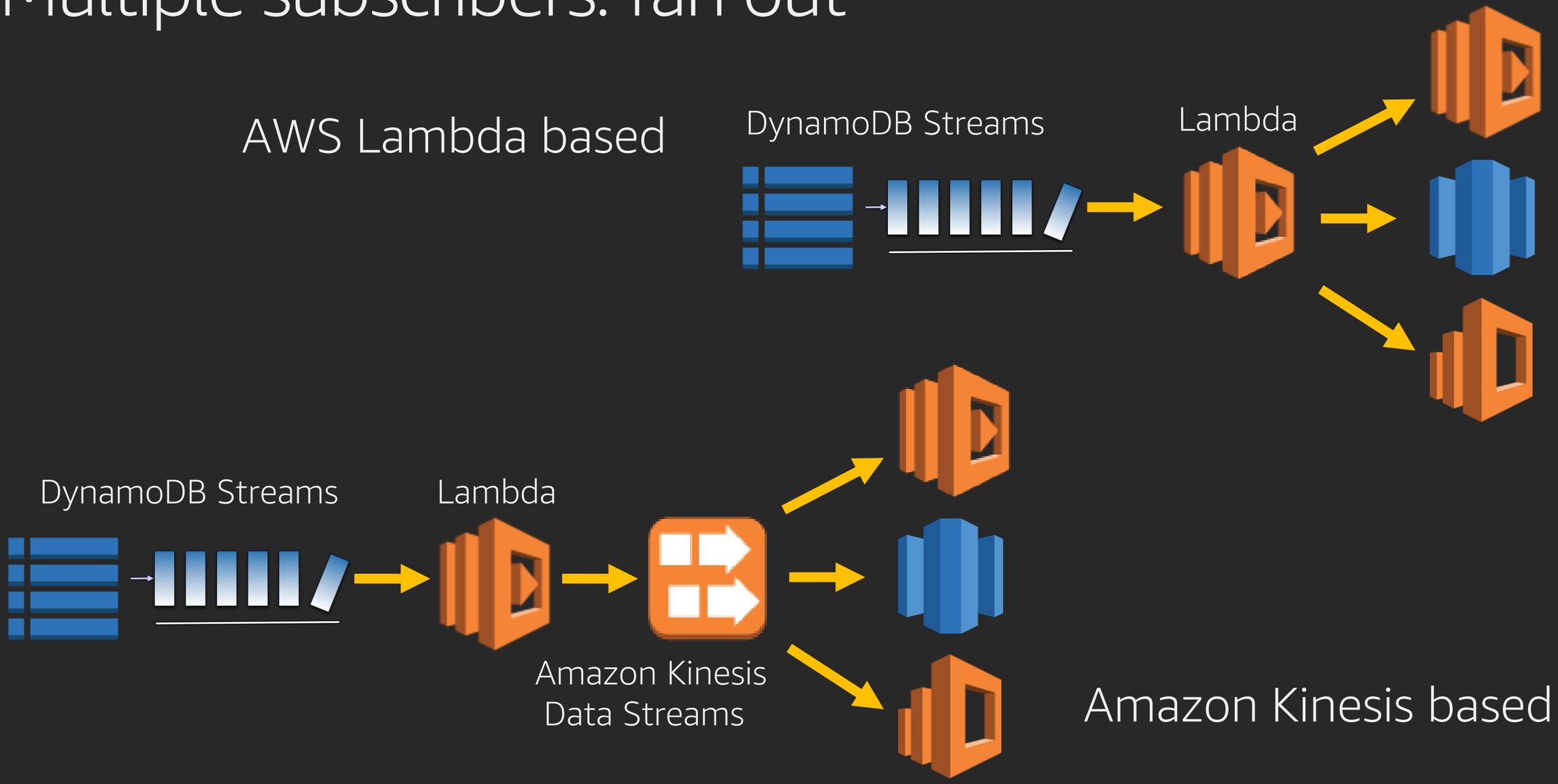


Querying with microservices



Different materialized views exposed as microservices

Multiple subscribers: fan out

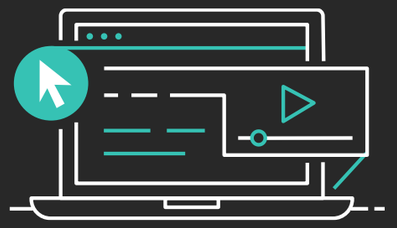


Querying with microservices: takeaways

- CQRS provides separation of concerns
  - CRUD operations and queries separated from each other
  - Each side optimized for its responsibility
- Each side is independently scalable
- Each view can be exposed as a microservice
- Async event delivery decouples
  - But also introduces eventual consistency and longer latency
- Added complexity
- Limit: only 2 stream readers supported

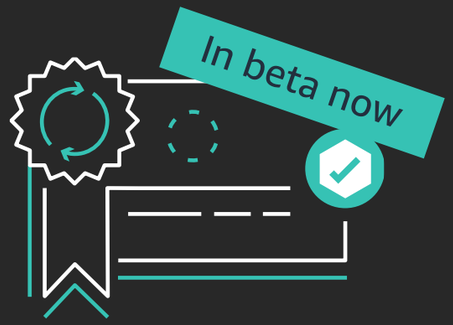
Learn databases with AWS Training and Certification

Resources created by the experts at AWS to help you build and validate database skills



25+ free digital training courses cover topics and services related to databases, including:

- Amazon Aurora
- Amazon DocumentDB
- Amazon DynamoDB
- Amazon ElastiCache
- Amazon Neptune
- Amazon RDS
- Amazon Redshift



Validate expertise with the new **AWS Certified Database - Specialty** beta exam

Visit aws.training

aws training and certification

Thank you!

Edin Zulich

ezzulich@amazon.com

@EdinZulich

AWS re:Invent

aws

Please complete the session survey in the mobile app.