AWS Miami Learning Days

Application Modernization: Monolith to Microservices with Containers

Diego Voltz

Sr. Solutions Architect
Amazon Web Services

Agenda

Developer’s perspective

Primer – monolith vs. microservices

Where do we start?

Challenges and learnings

Using containers

Questions for Audience

1. Do you use microservices today?

    - If so, which pattern do you use ?

1. Do you support monolith applications today?

    - If so, how long is your deployment time ?

    -whats your rollback strategy ?

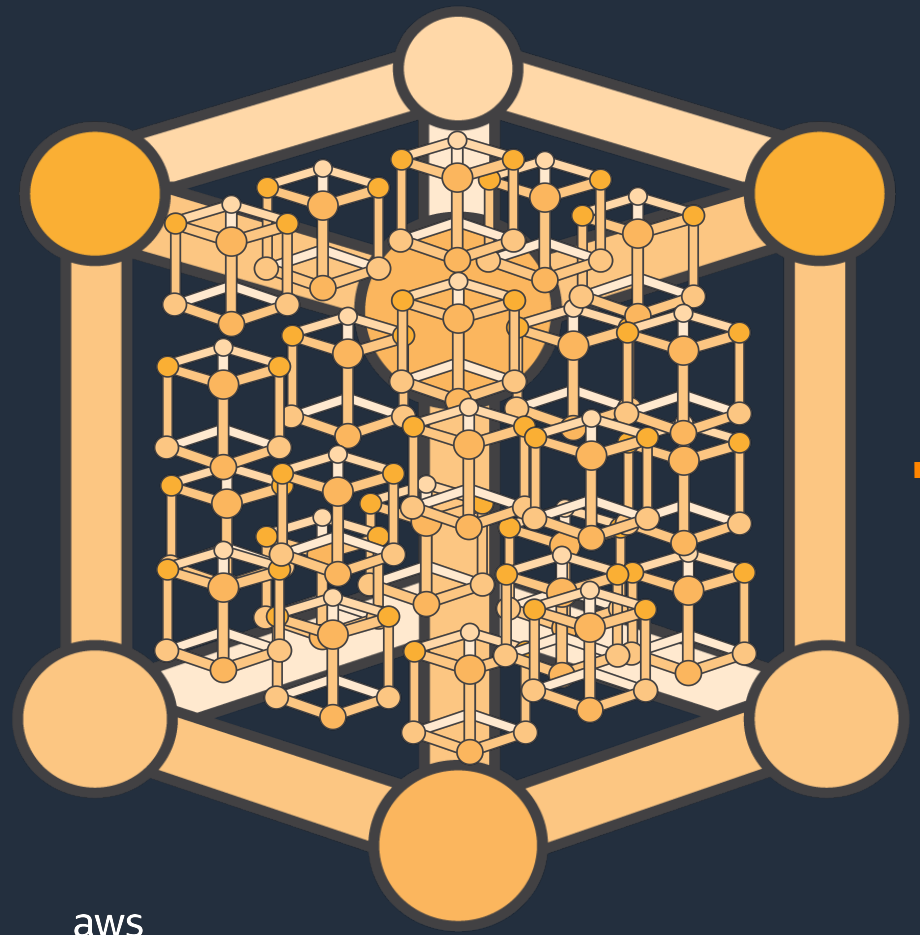
Developer’s Perspective

- Application redesign patterns
- Data layer
- Synchronous to Asynchronous
- Orchestration
- Monitoring
- Containerization

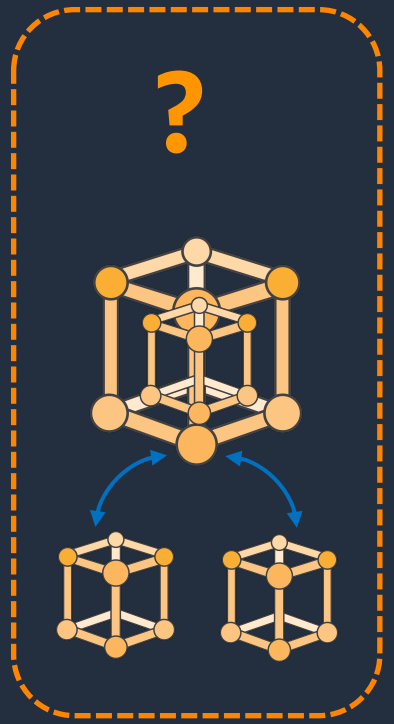
Source: Amazon Freeway

Basic Concepts and Definitions

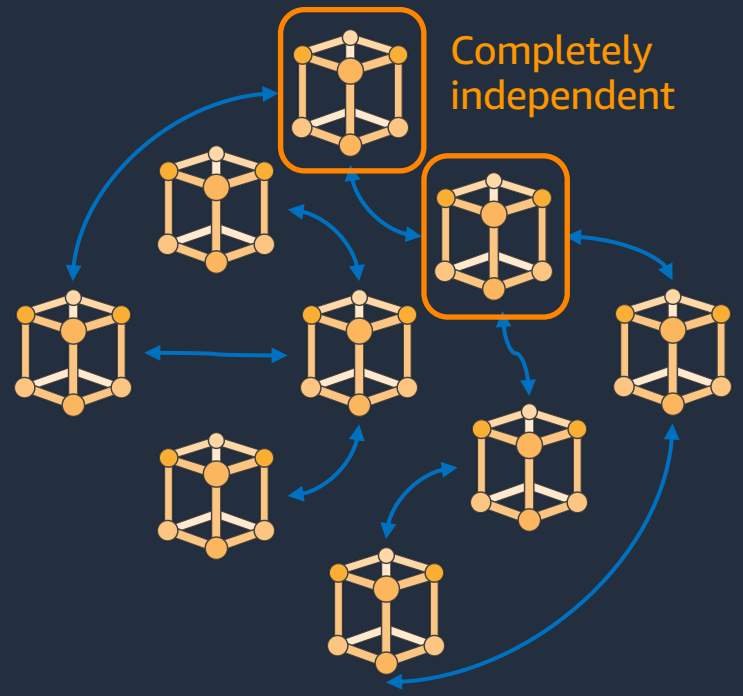
Monolith



Miniservices

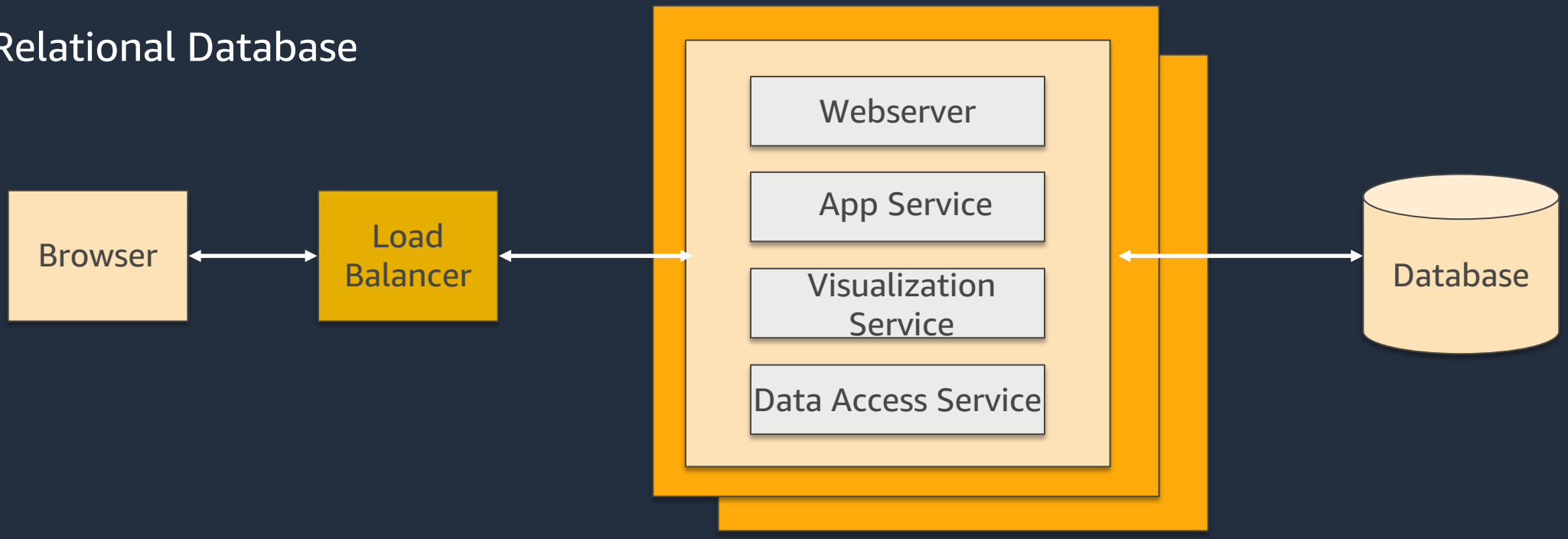


Microservices



Original Monolithic Application - Example

- On-premises
- Tightly coupled application components
- Load balancer
- Relational Database



Monolithic Applications - Limitations

Hard to Scale

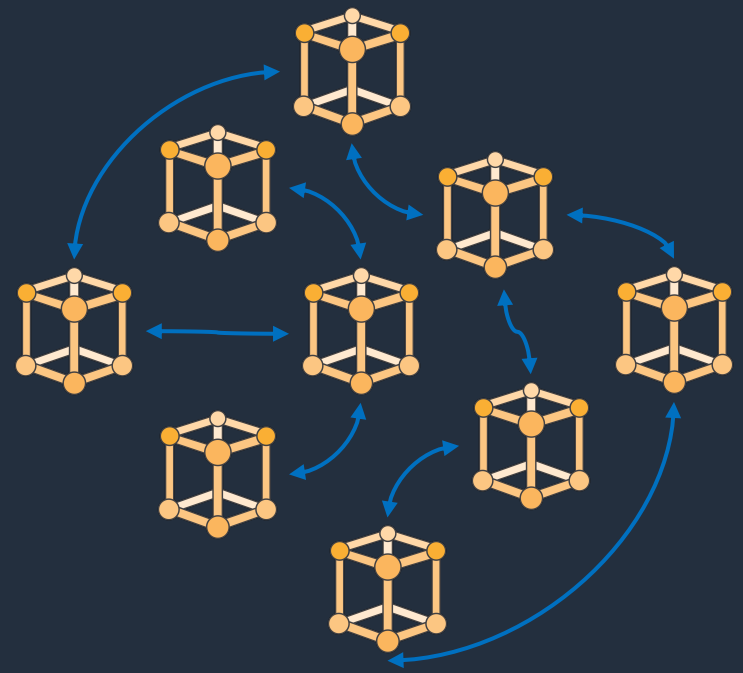
Can’t Handle Component Failures

Slow Deployment Process

Limited options

Drivers to Switch to Microservices

- Time to Market
- Time to Repair
- Enabled Hyperscaling
- Technologically Independent



Where do we start?

Where Do We Start? - Discover

1. Identify Components

- Webserver
- App Service
- Visualization Service
- Data Access Service
- Database

2. Outline Requirements

- State?
- Compute?
- API?
- Storage?
- Security?
- Managed?
- Estimated scale?
- etc.

3. Map to Amazon Web Services(AWS)Resources

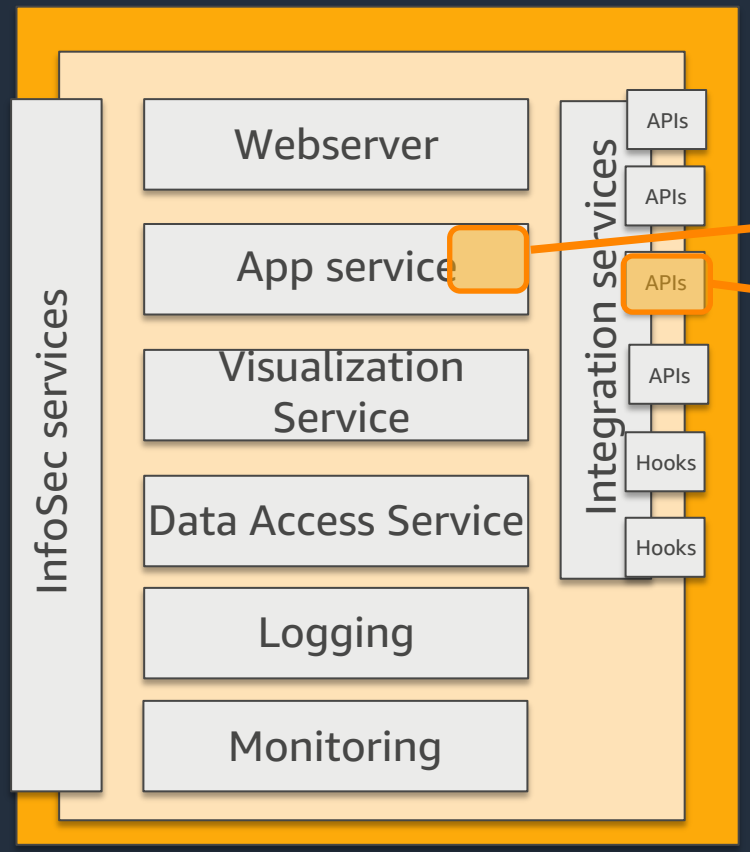
Amazon DynamoDB

AWS Lambda

Amazon API Gateway

Amazon S3

Design, Develop, Deploy - a Pilot



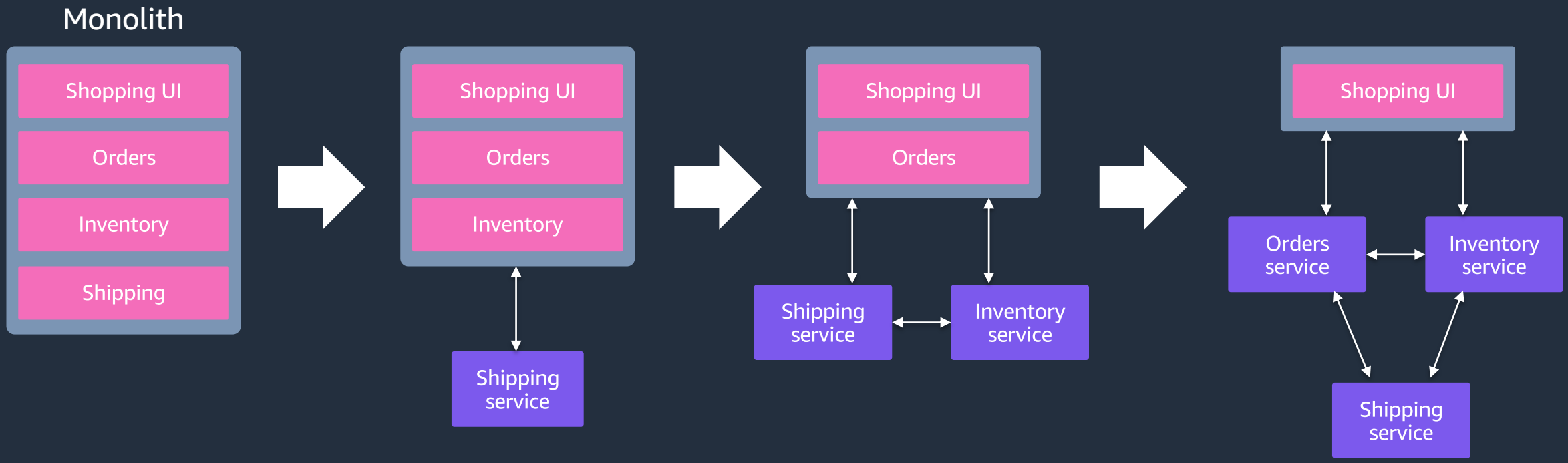
Technical requirements

- API-driven
- Independent DBs
- Containerized or serverless

Organizational requirements

- Dedicated product team
- Small frequent incremental changes

Recommended Approach



Strangler Fig Application Pattern:

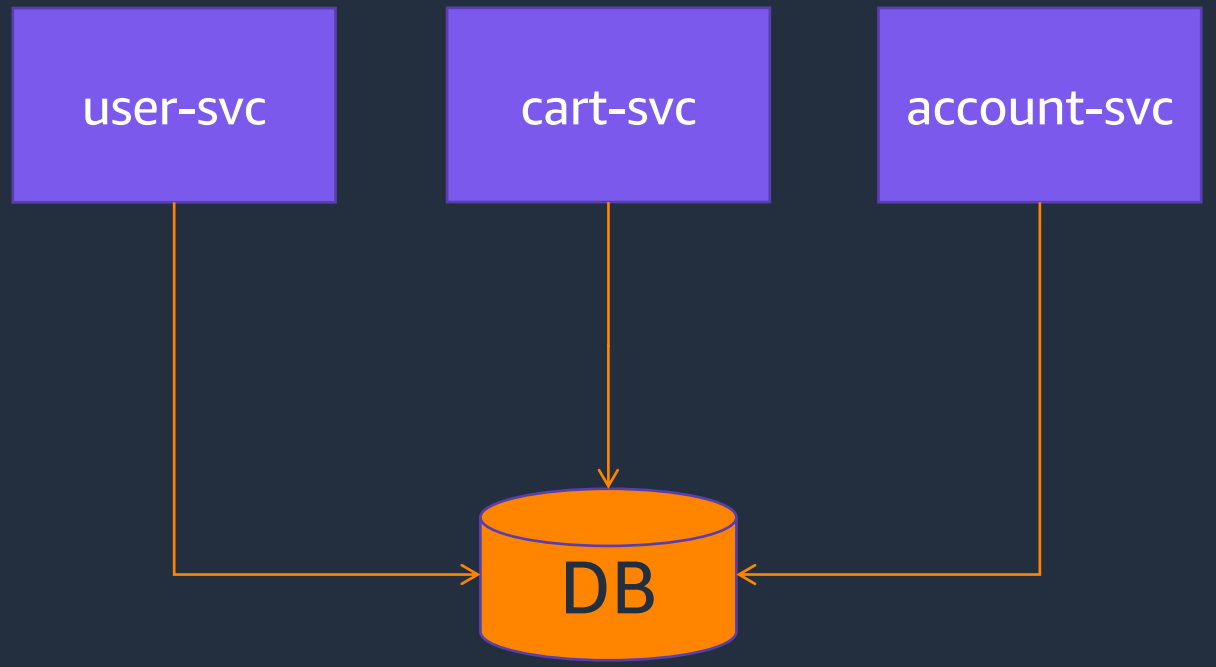
https://www.martinfowler.com/bliki/StranglerApplication.html

Challenges and Learnings

Challenge: Centralized Database

Applications often have a **monolithic** data store

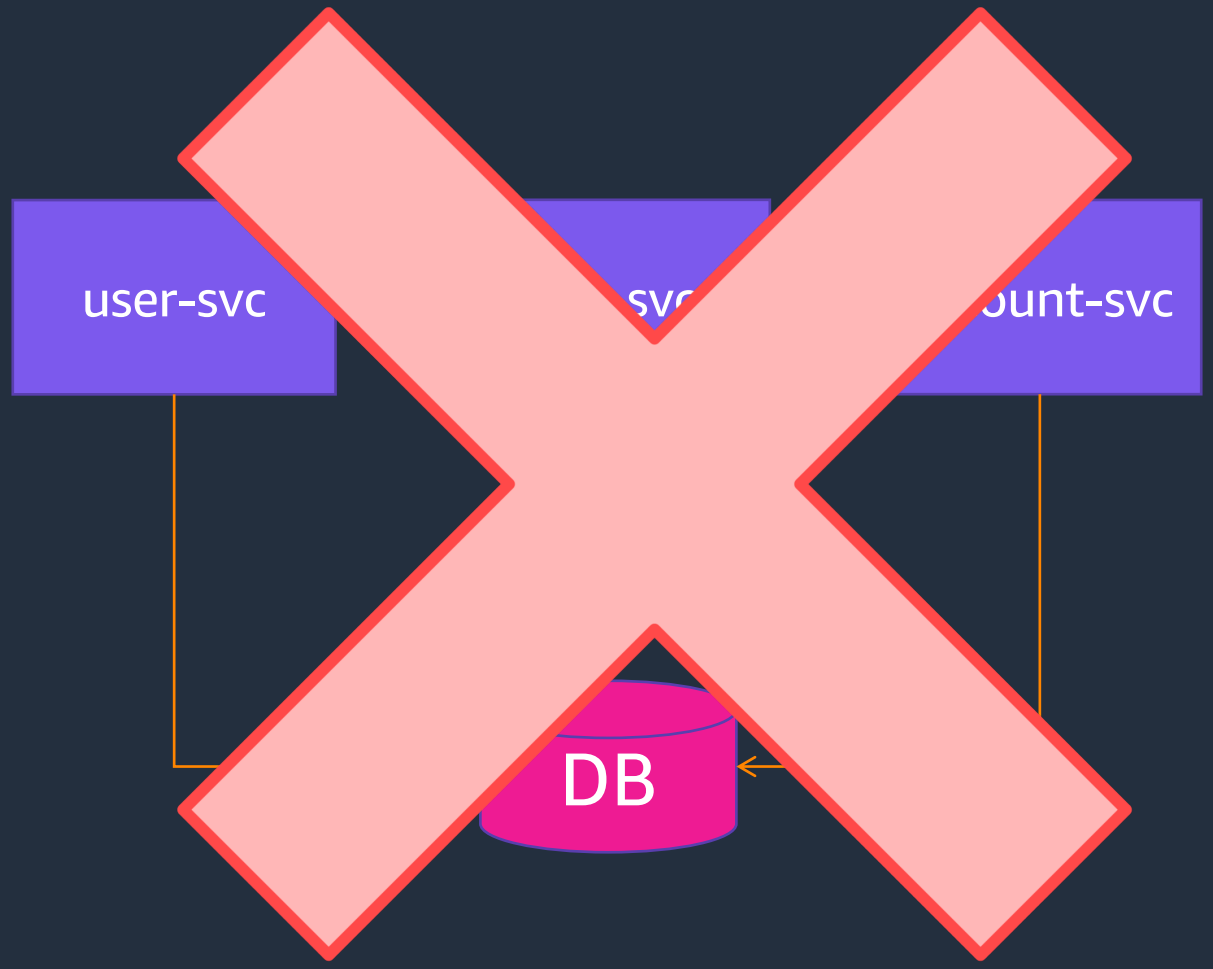
- Difficult to make schema changes
- Technology lock-in
- Vertical scaling
- Single point of failure



Centralized Database – Anti-Pattern

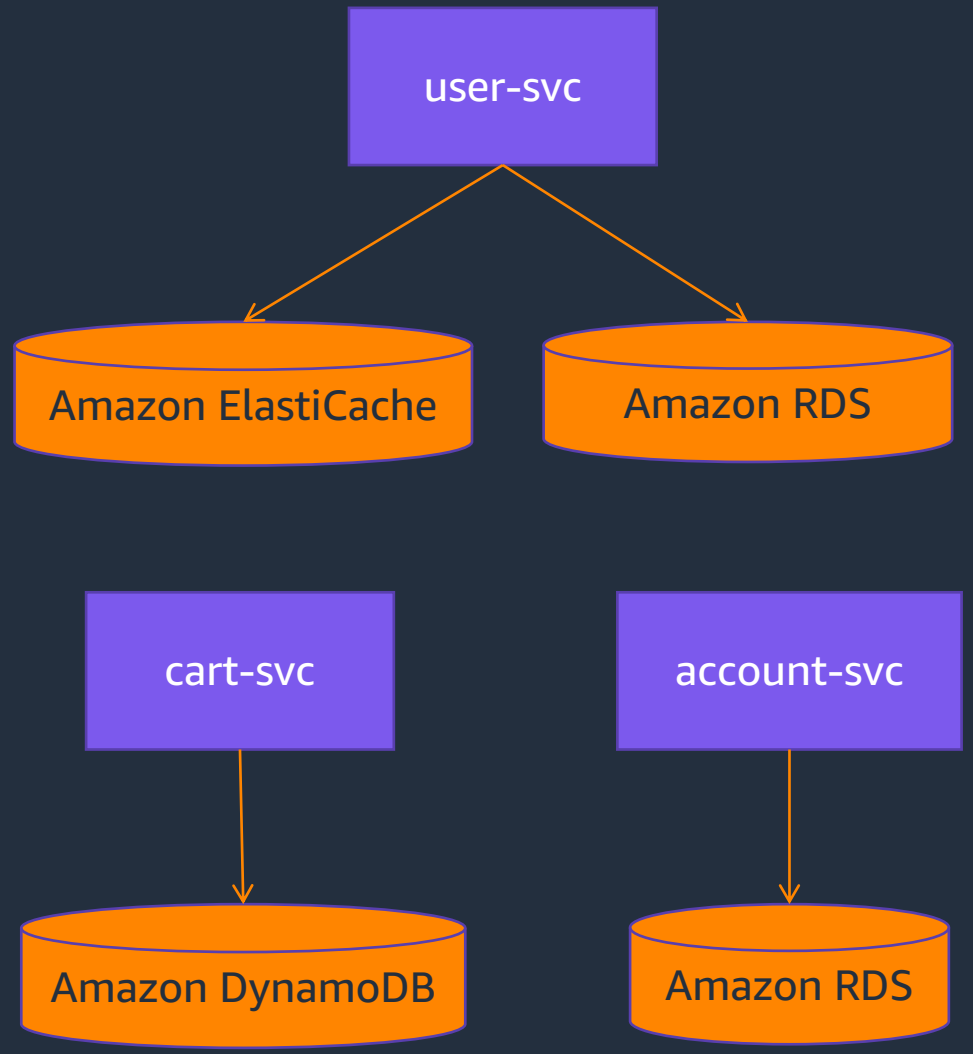
Applications often have a **monolithic** data store

- Difficult to make schema changes
- Technology lock-in
- Vertical scaling
- Single point of failure



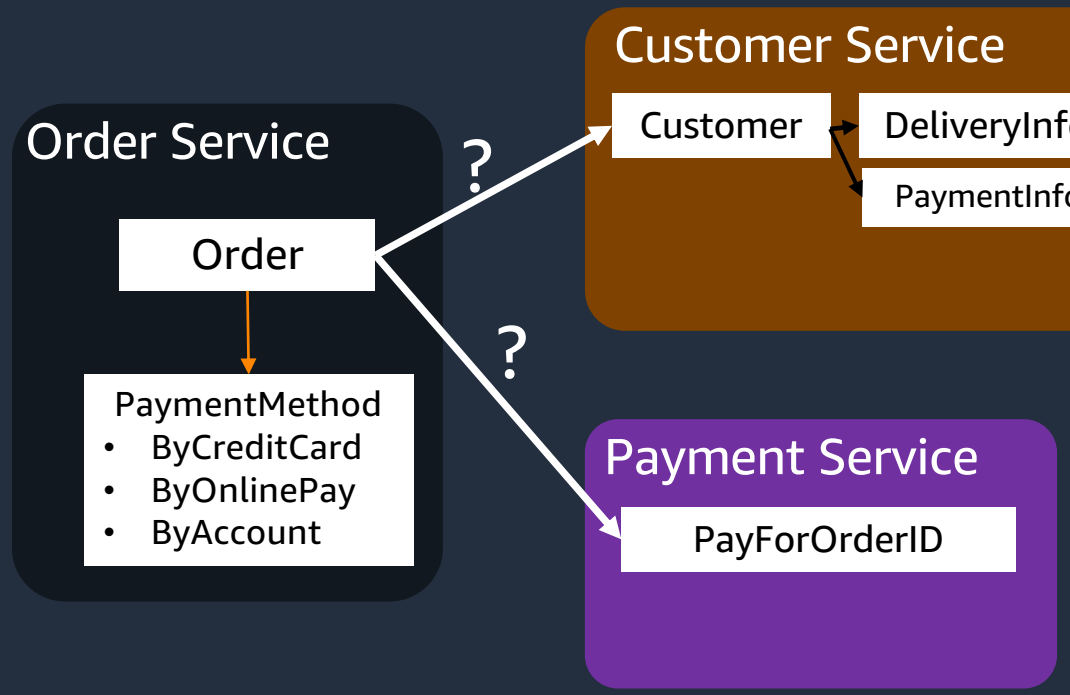
Decentralized Data Stores

- **Polyglot** Persistence
- Each service chooses its data store technology
- Low impact schema changes
- Independent scalability
- Data is gated through the service API

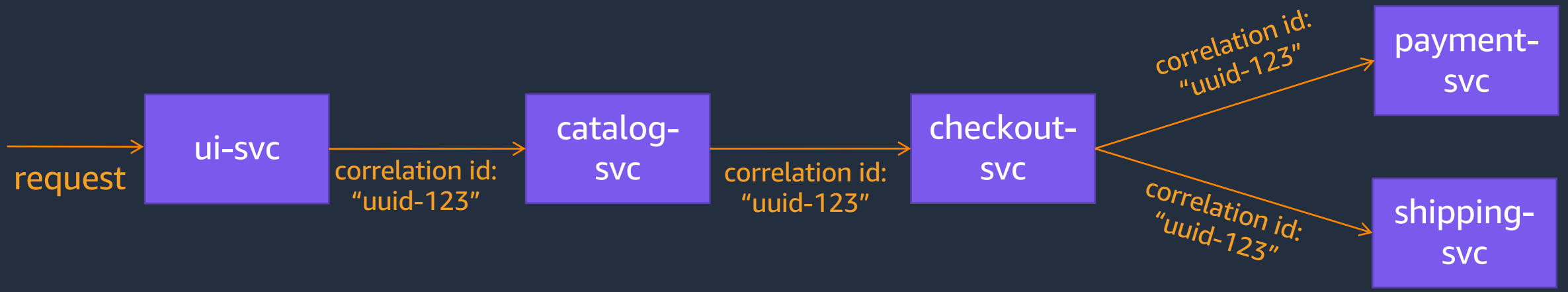


Challenge: Transactional Integrity

- Polyglot persistence generally translates into **eventual consistency**
- **Asynchronous calls** allow non-blocking, but returns need to be handled properly
- How about **transactional integrity?**
  - Event-sourcing – Capture changes as sequence of events
  - Staged commit
  - Rollback on failure



Best Practice: Use Correlation IDs

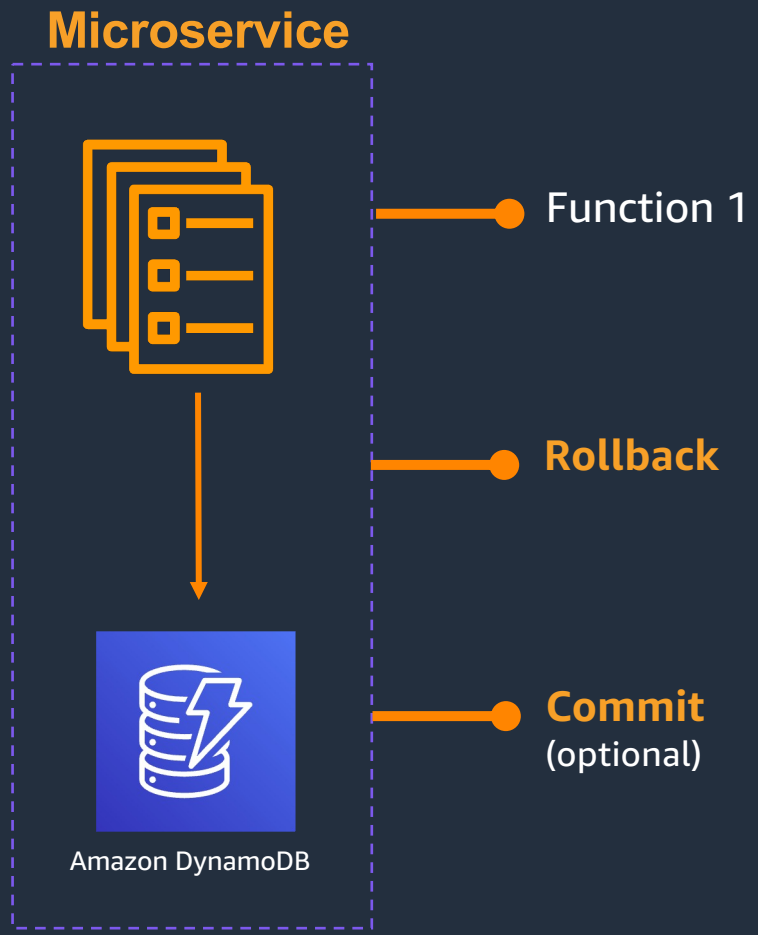



```
09-02-2015 15:03:24 ui-svc INFO [uuid-123] ……
09-02-2015 15:03:25 catalog-svc INFO [uuid-123] ……
09-02-2015 15:03:26 checkout-svc ERROR [uuid-123] ……
09-02-2015 15:03:27 payment-svc INFO [uuid-123] ……
09-02-2015 15:03:27 shipping-svc INFO [uuid-123] ……
```

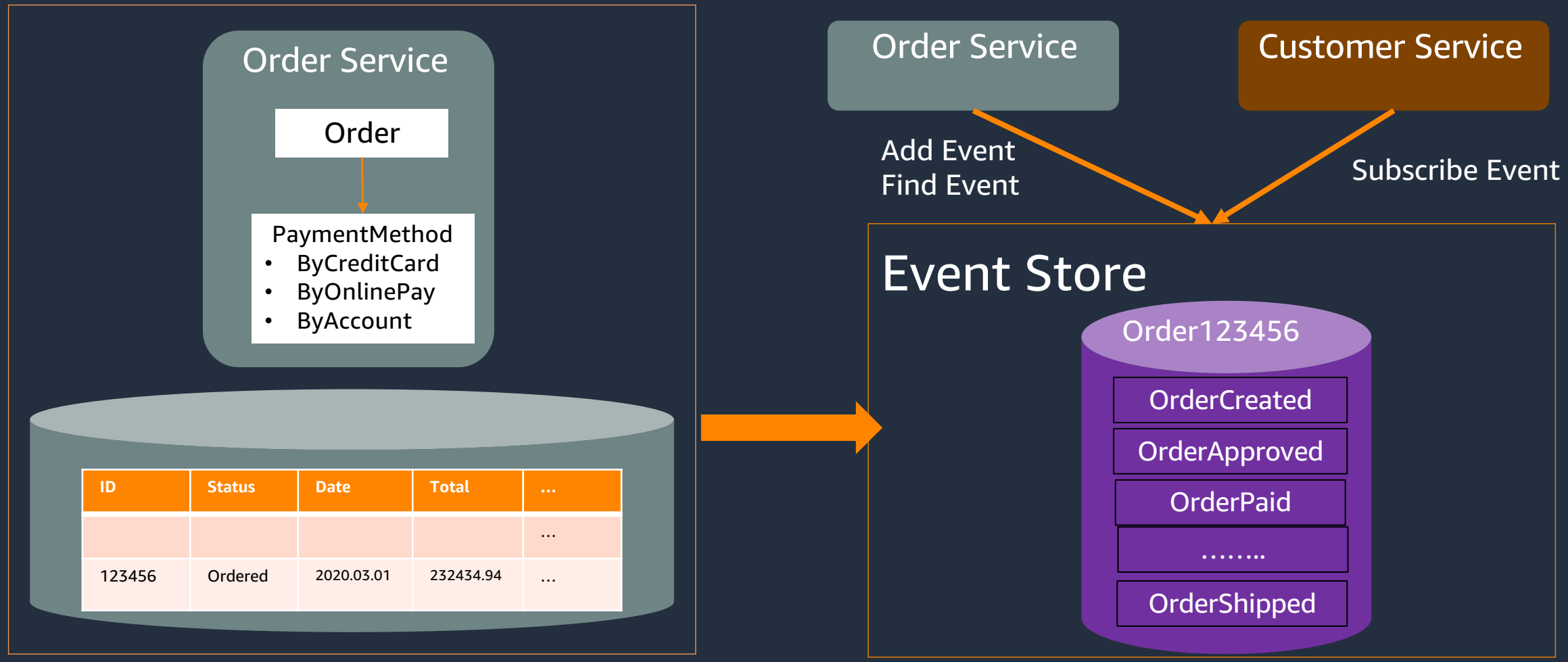
aws

Best Practice: Microservice Owns Rollback

- Every microservice should expose its own “rollback” method
- This method could just **rollback** changes, or trigger **subsequent actions**
  - Could send a notification
- If you implement **staged commit**, also expose a commit function



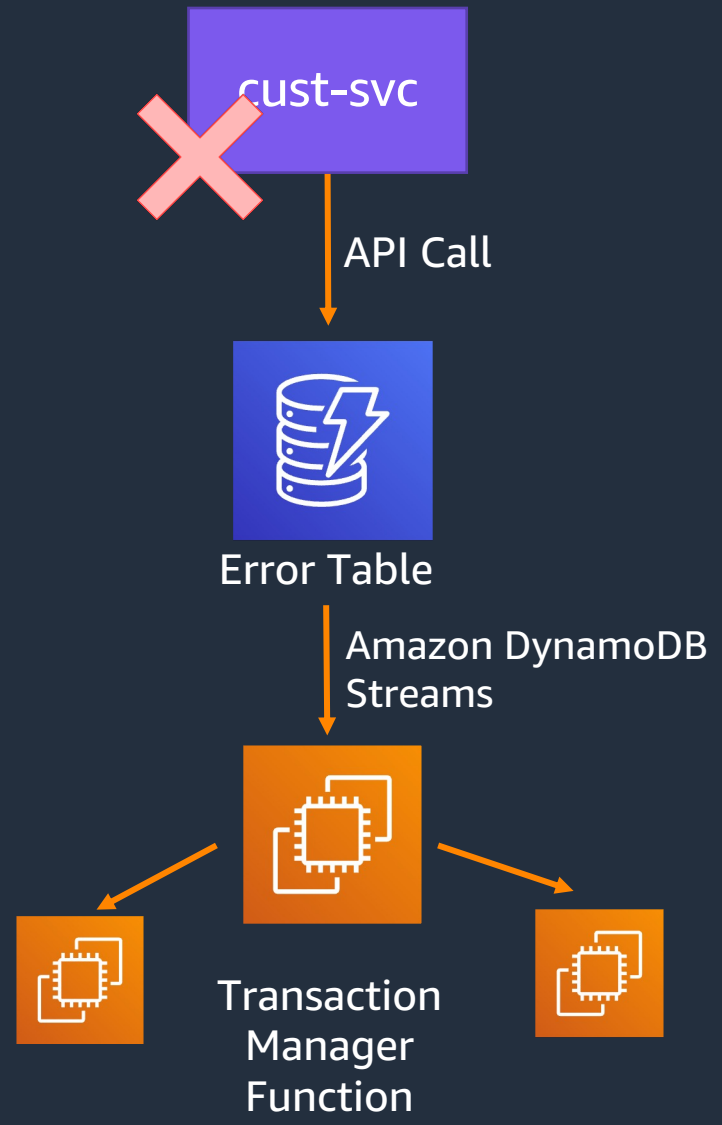
Best Practice: Event Sourcing Pattern



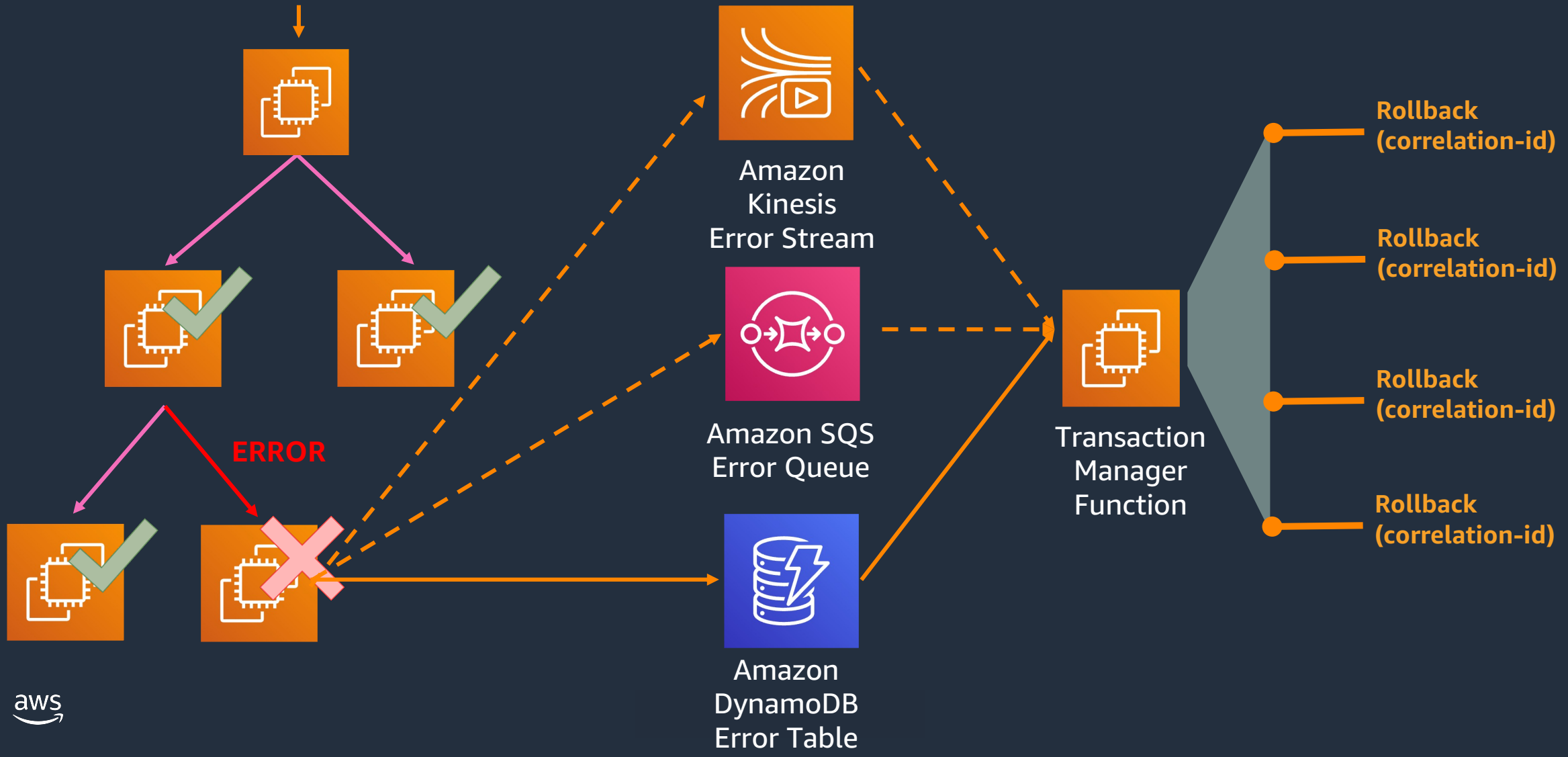
Best Practice: Event Sourcing Pattern on AWS

Challenge: Report Errors / Rollback

- What if functions fail? (business logic failure, not code failure)
- Create a **"Transaction Manager"** microservice that notifies all relevant microservices to rollback or take action
- Amazon DynamoDB is the **trigger** for the clean-up function (could be Amazon SQS, Amazon Kinesis etc.)
- Use **Correlation ID** to identify relations

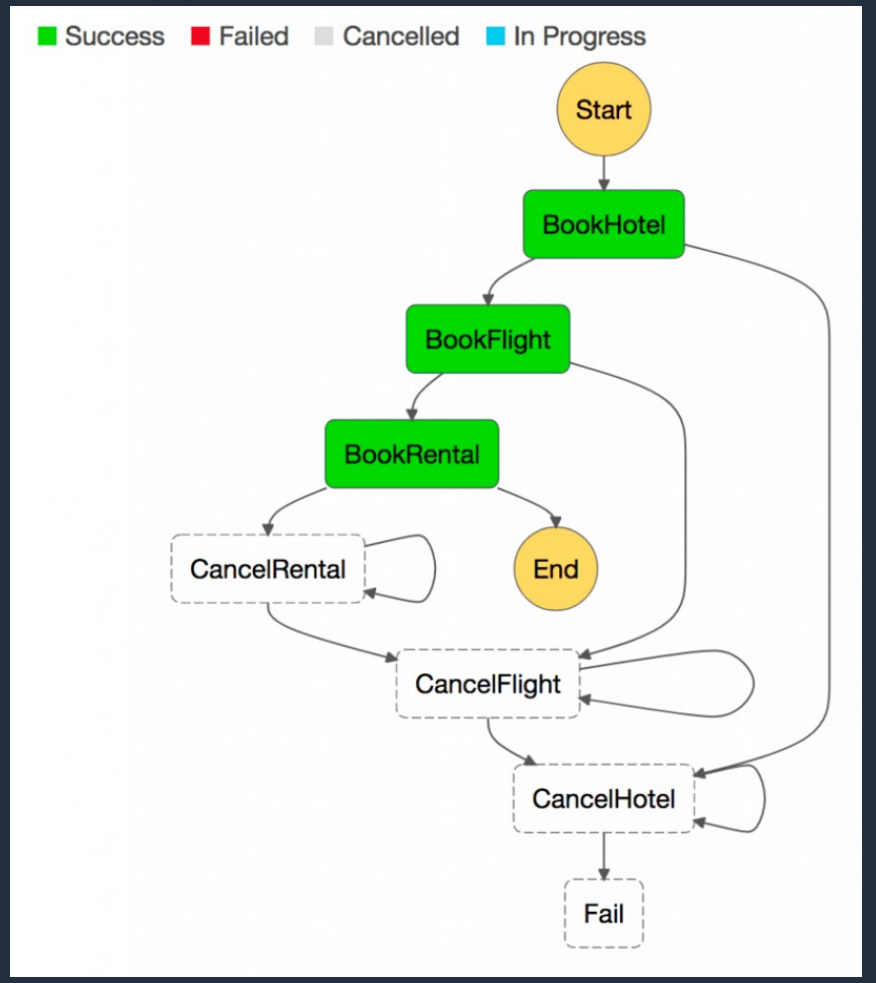


Challenge: Report Errors / Rollback



Challenge: Saga Pattern using AWS Step Functions & AWS Lambda

Using AWS Step Functions as a **“Transaction Manager”** to catch failure situations and perform rollbacks.

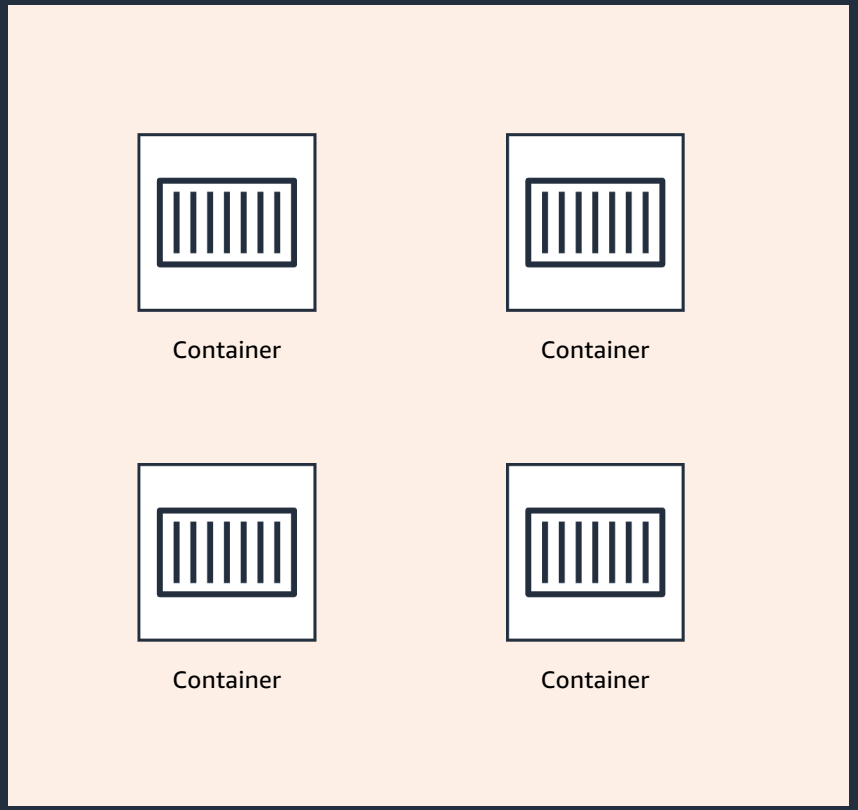


https://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions/

Using containers

Containers and Microservices

- Do one thing, really well
- Any app, any language
- Test and deploy same artifact
- Self-contained services
- Isolated execution environment
- Faster startup
- Scaling and upgrading



Container Orchestration Platform Options

ECS

Powerful simplicity

EKS

Open flexibility

Amazon Elastic Container Service (ECS)

Developed by Amazon

Used within Amazon

- Amazon SageMaker
- AWS Batch
- Recommendation engine

Natively integrates with AWS

Amazon Elastic Kubernetes Service (EKS)

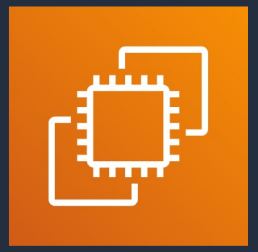
Open-source Kubernetes

Fully-managed environment

Full compatibility with upstream

Integrates with AWS services

Choosing a Compute Layer



EC2

- Consistent utilization
- Pack instances as full as possible
- Specialized resource needs (GPU, Inference)
- Maintenance & updates are customer responsibility
- Windows & Linux

Fargate

- Variable or unpredictable scaling
- Batch workloads
- Low overhead – no server maintenance
- Linux only

Please complete the session survey by scanning the QR code

Thank you!

Diego Voltz
Sr. Solutions Architect
Amazon Web Services
diegovf@amazon.com

Track: Application Modernization and Security
Session: Application Modernization: Monolith to Microservices with Containers