COLUMBUS LEARNING DAYS

Application modernization: monolith to microservices with containers

David Stielstra
Solutions Architect
Amazon Web Services

Kiran Palsam
Solutions Architect
Amazon Web Services

Agenda

- Considerations for application modernization
- Primer – monolith vs. microservices
- Where do we start?
- Challenges and learnings
- Using containers

Considerations for application modernization

Considerations for application modernization

- Application redesign patterns
- Data layer
- Synchronous to asynchronous
- Orchestration
- DevSecOps
- Monitoring
- Containerization

Source: Amazon Freeway

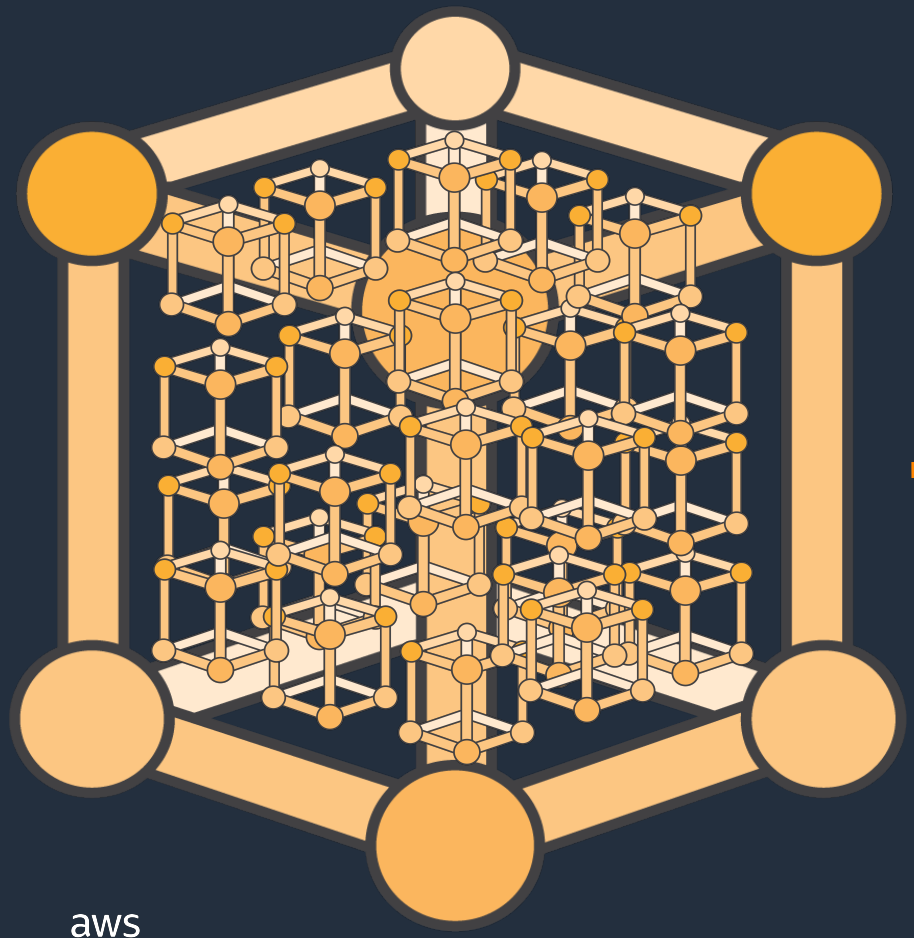
Monolith v. microservices

What is a monolith?

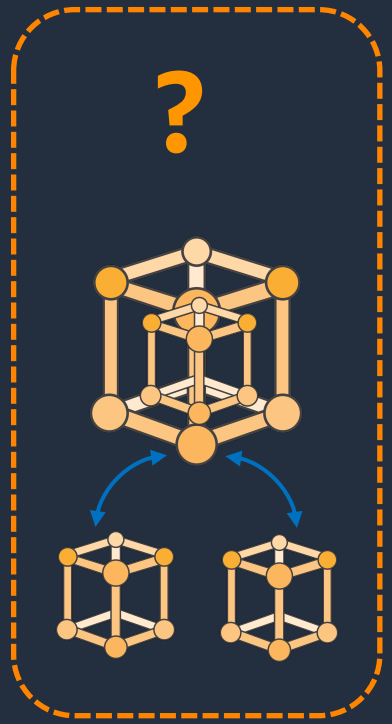
What are microservices?

Basic concepts and definitions

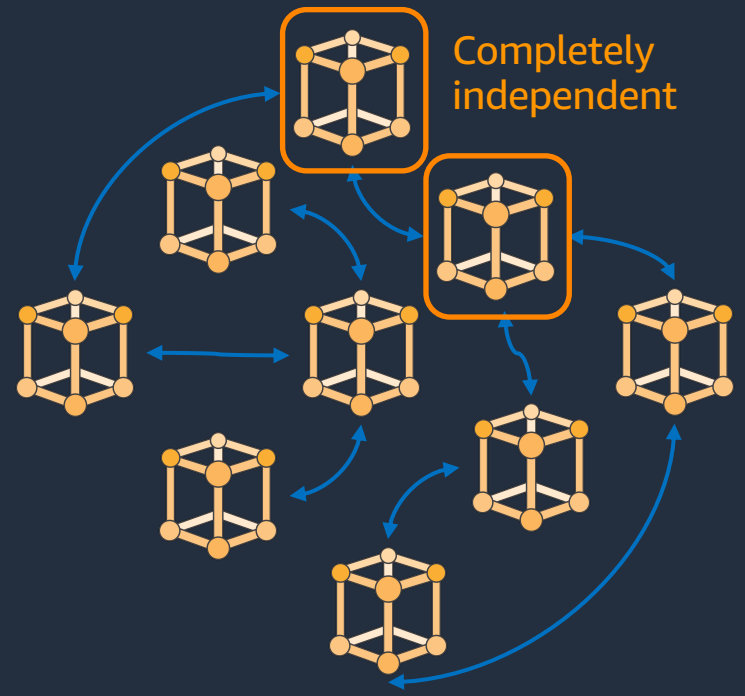
Monolith



Miniservices

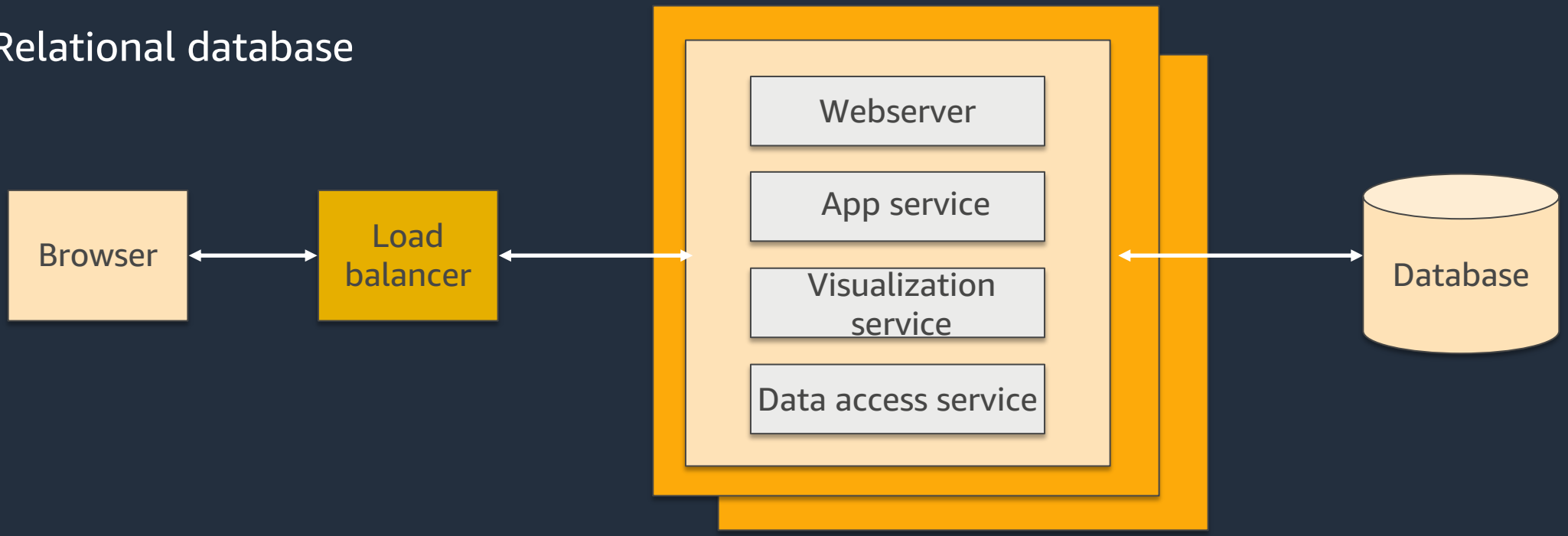


Microservices



Original monolithic application - example

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



Monolithic applications – challenges

Hard to scale

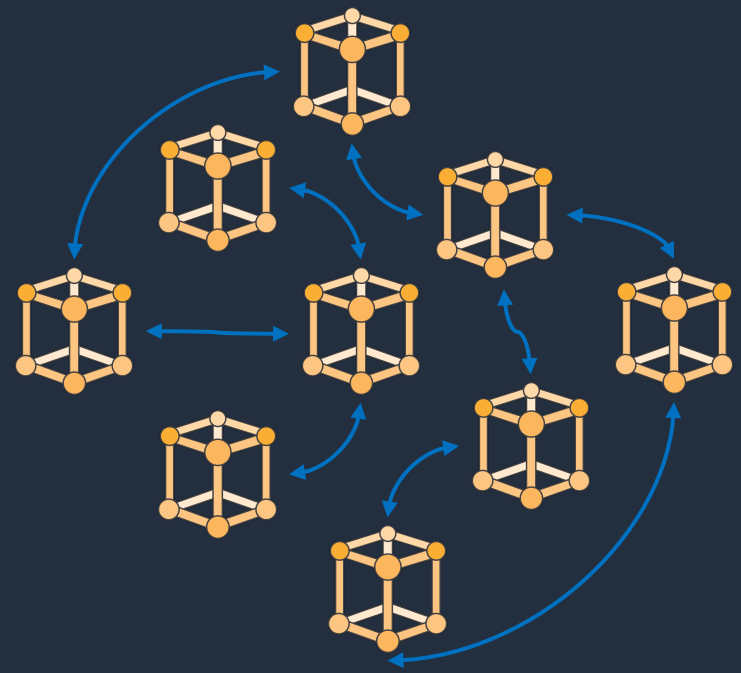
Can’t handle component failures

Slow deployment process

Limited options

Drivers to switch to microservices

- Time to organization value
- Time to repair
- Enabled hyper scaling
- 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 AWS resources

Amazon DynamoDB

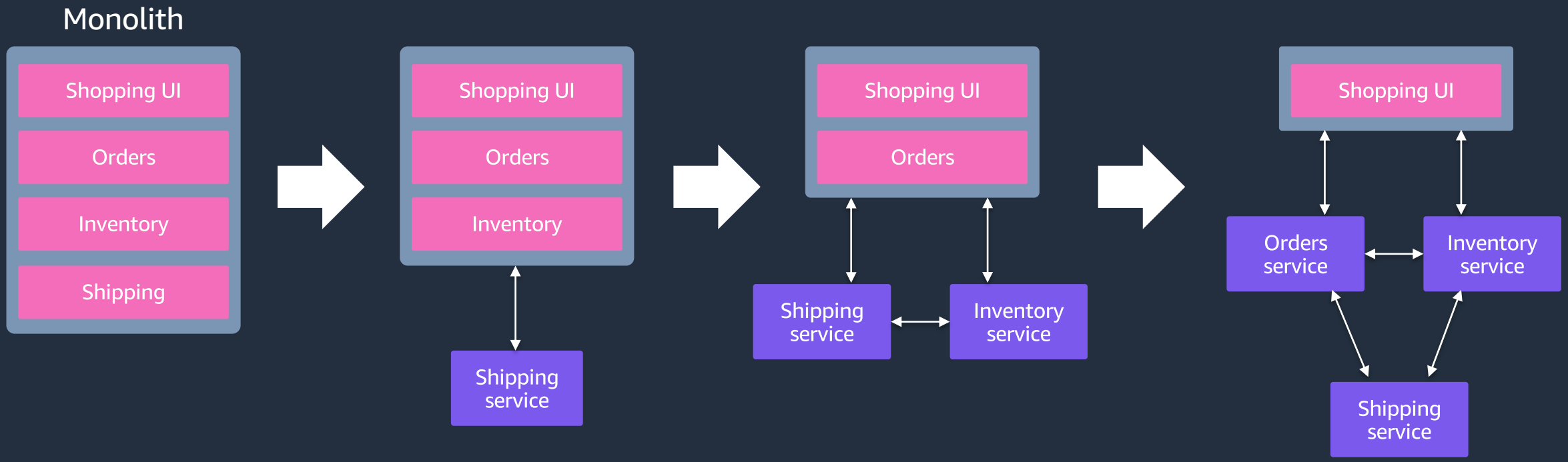
AWS Lambda

Amazon API Gateway

Amazon Simple Storage Service

The Strangler Fig

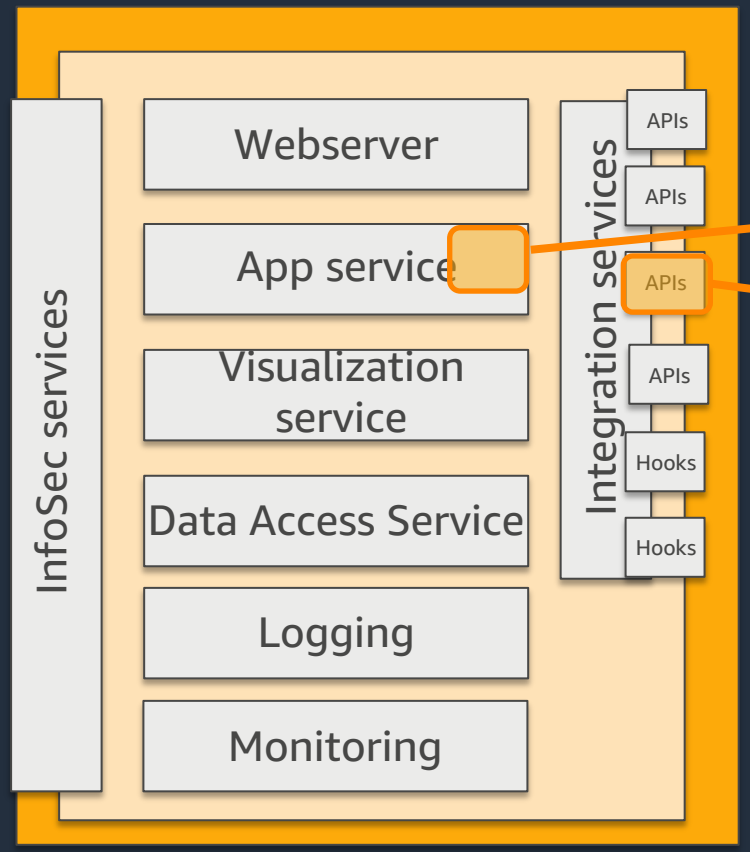
Recommended pattern



Strangler application pattern:

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

Design, develop, deploy - A pilot



Technical requirements

- API-driven
- Independent DBs
- Containerized or serverless

Organizational requirements

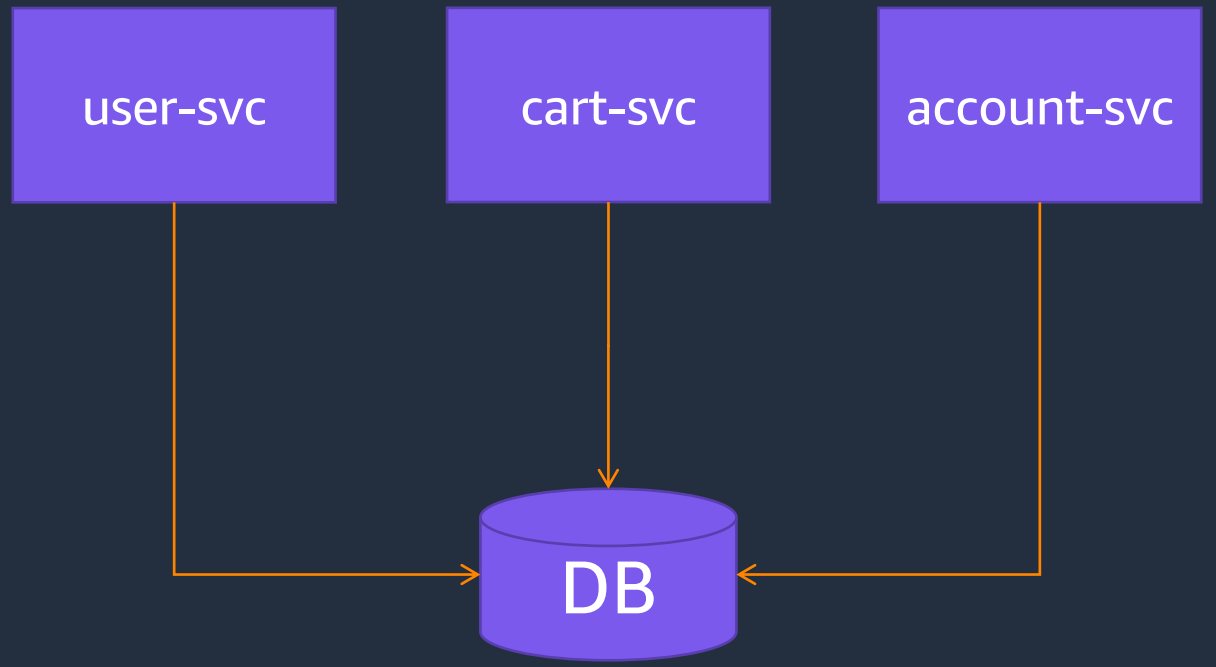
- Dedicated product team
- Small frequent incremental changes

Challenges and learnings

Challenge - Centralized database

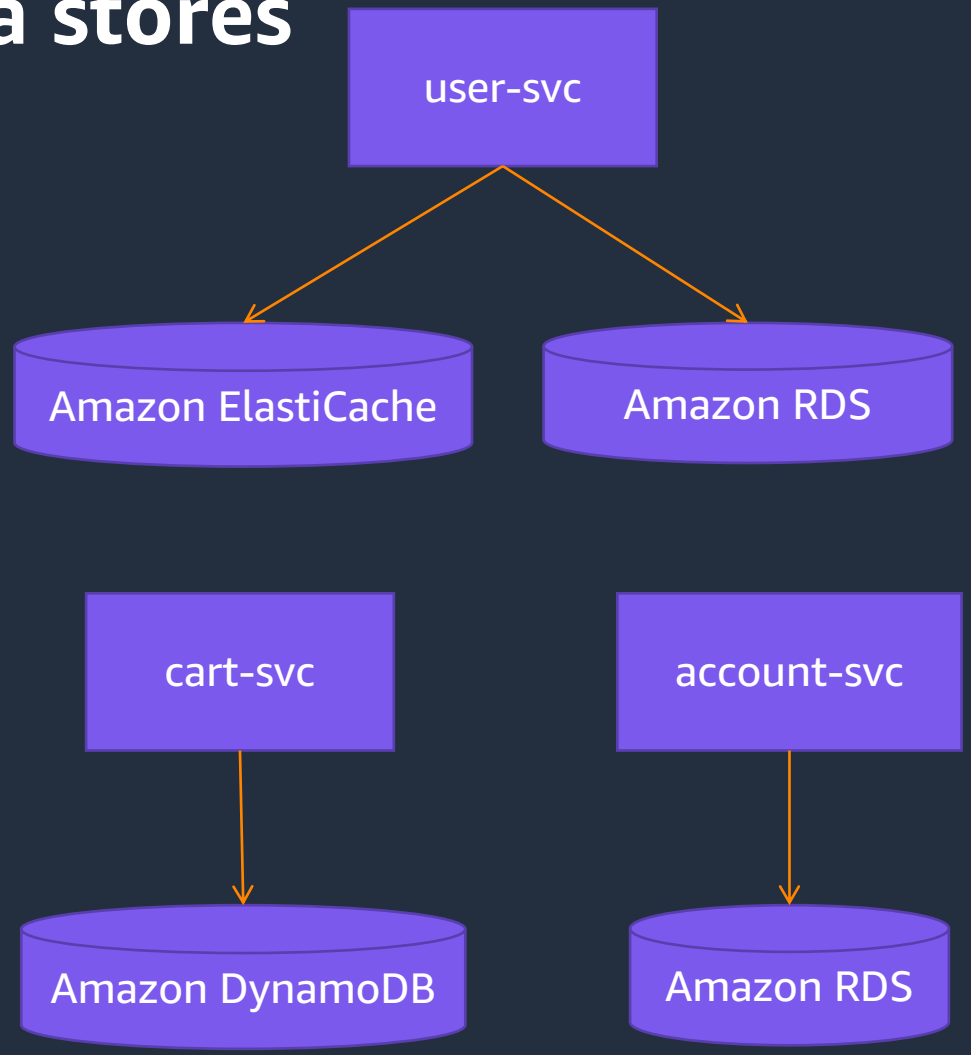
Applications often have a **monolithic** data store:

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



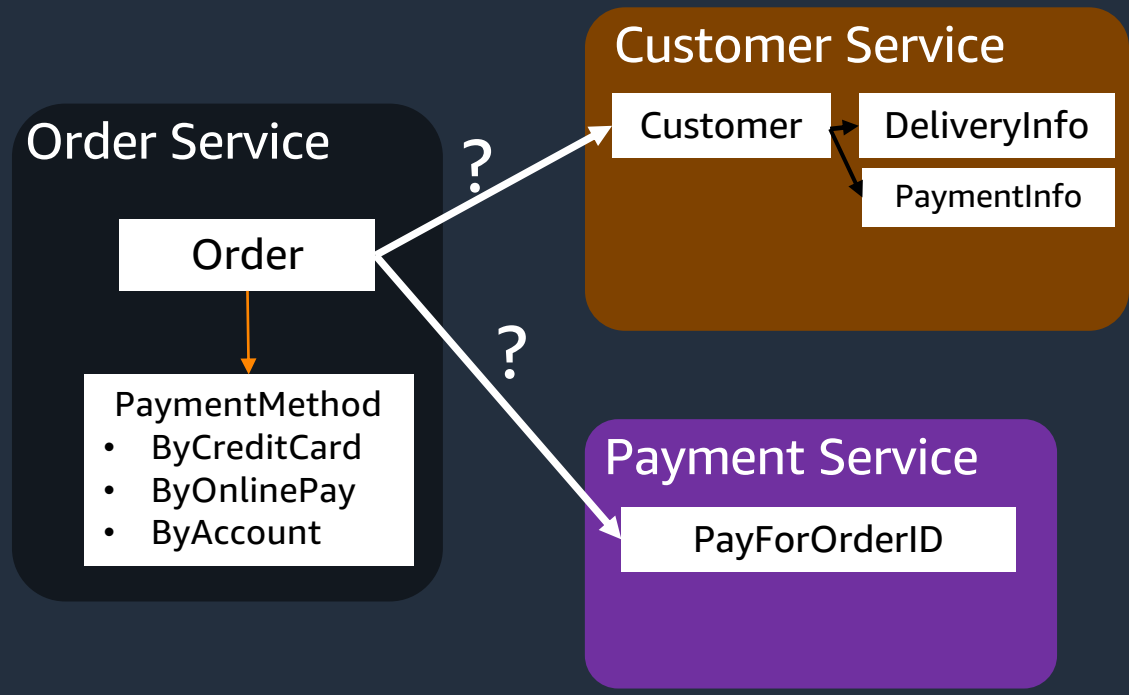
Best practice: decentralized data stores

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

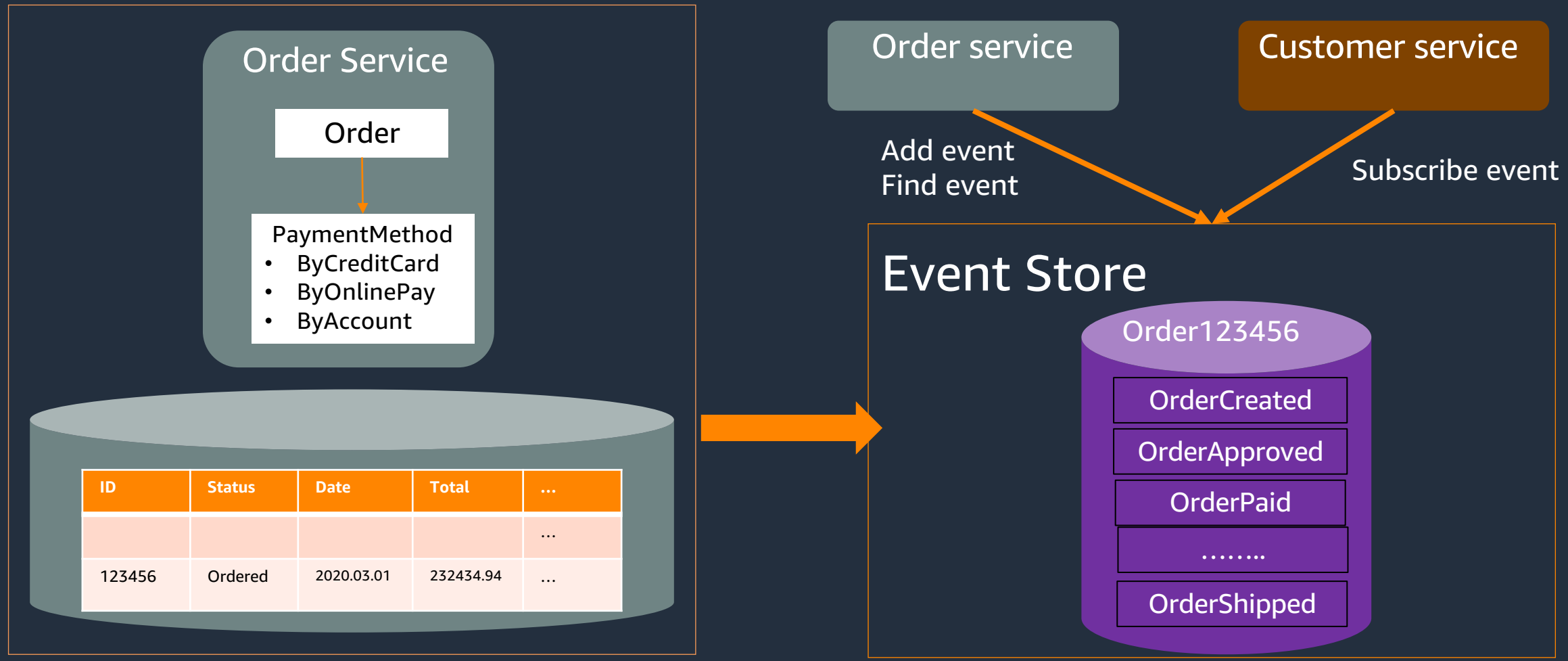


Challenge - transactional integrity

- Moving to microservices introduces polyglot persistence and asynchronous behaviors
- How do we handle **transactional integrity?**
  - Event-sourcing: Capture changes as sequence of events
  - Staged commit
  - Rollback on failure

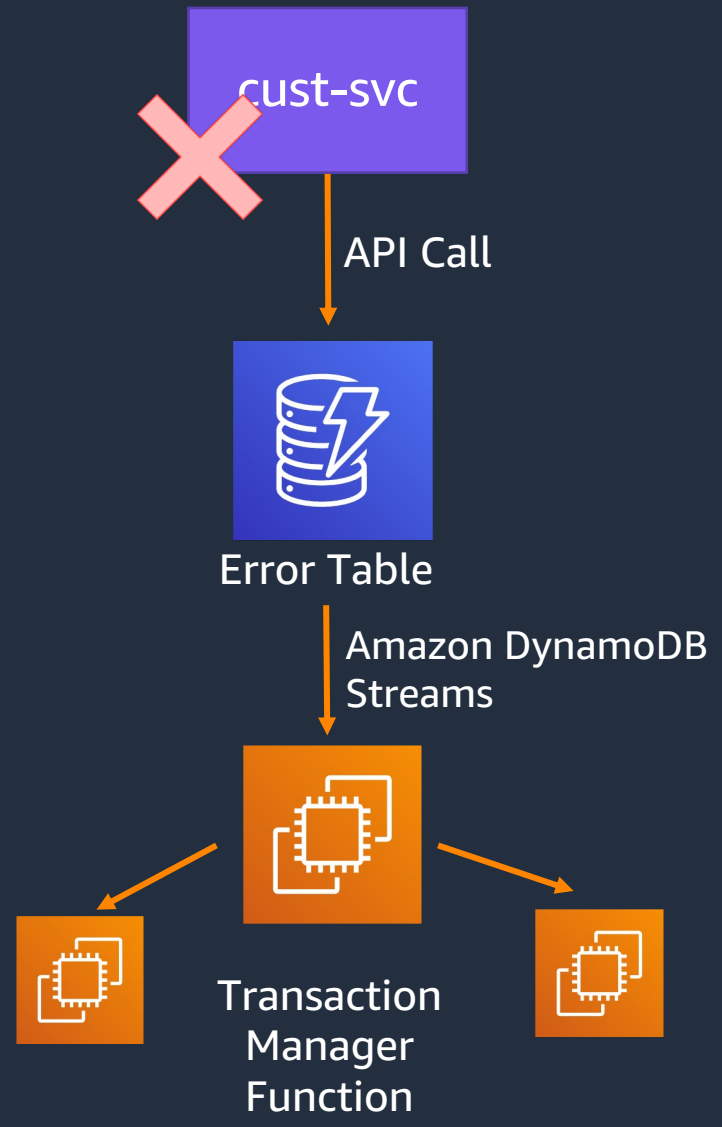


Best practice: event sourcing pattern

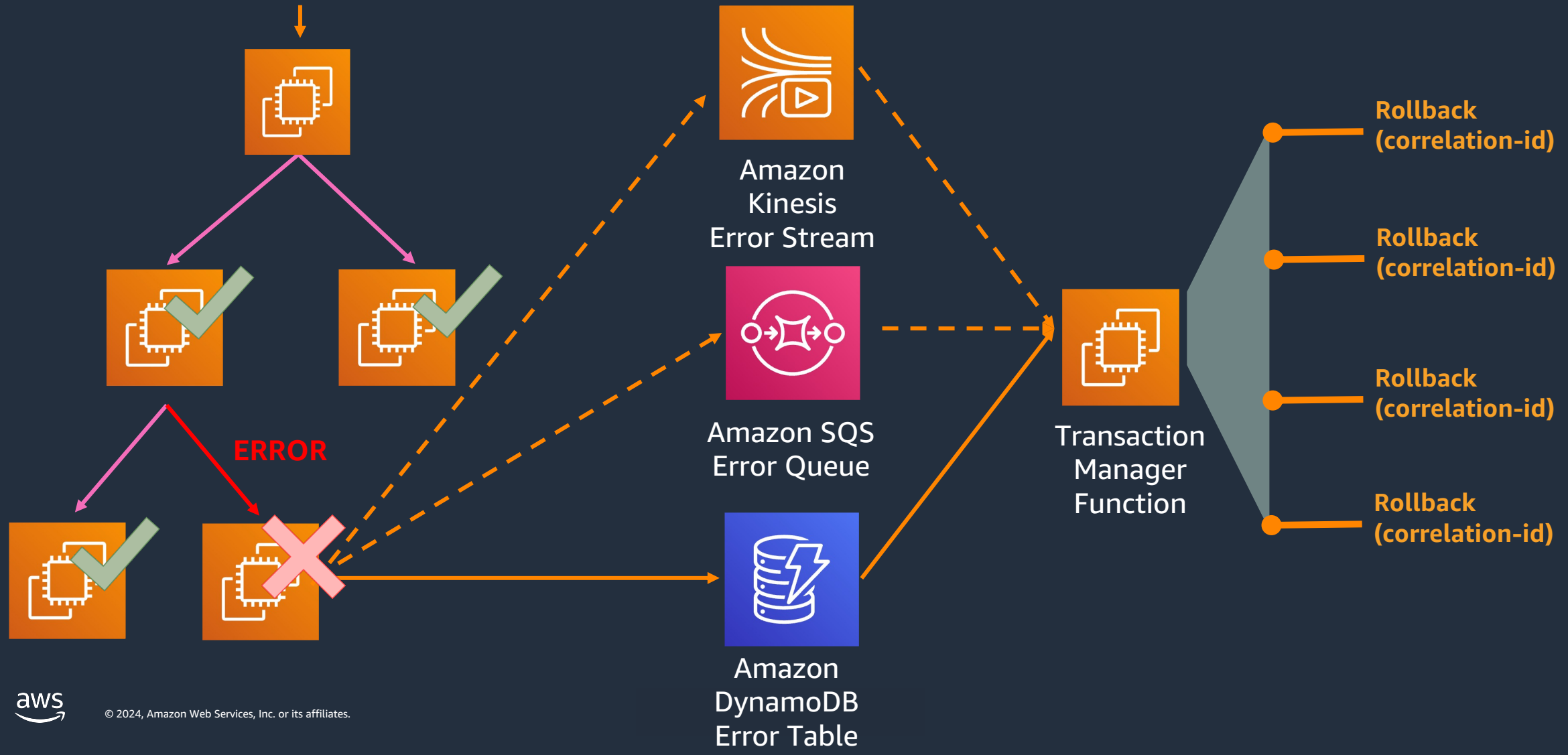


Challenge - report errors / rollback

- 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

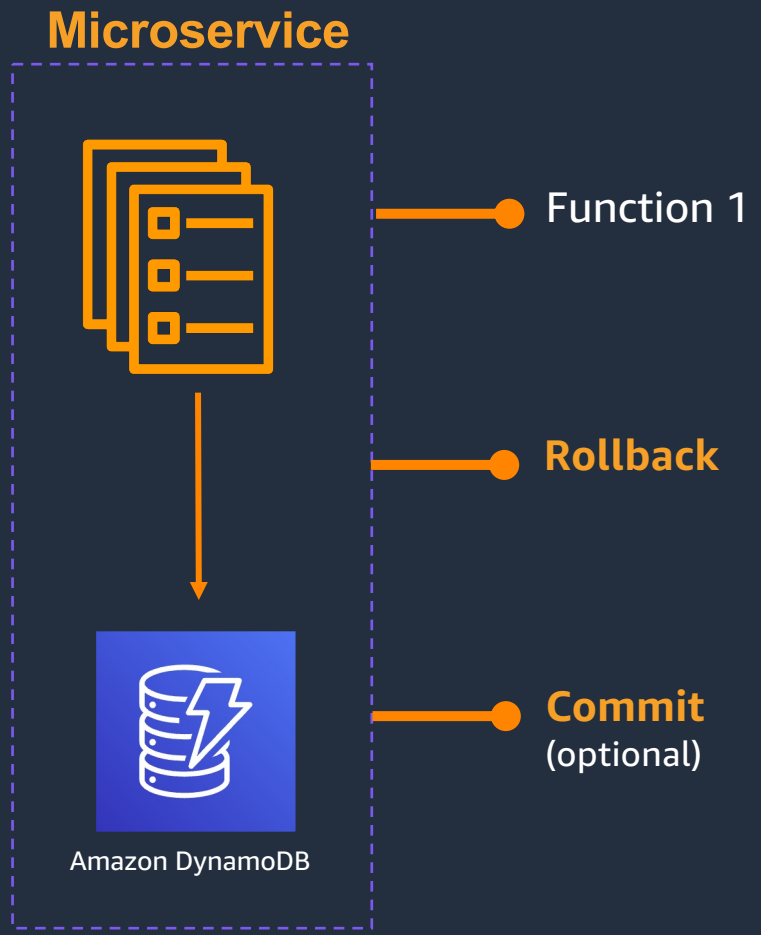


Challenge: report errors / rollback

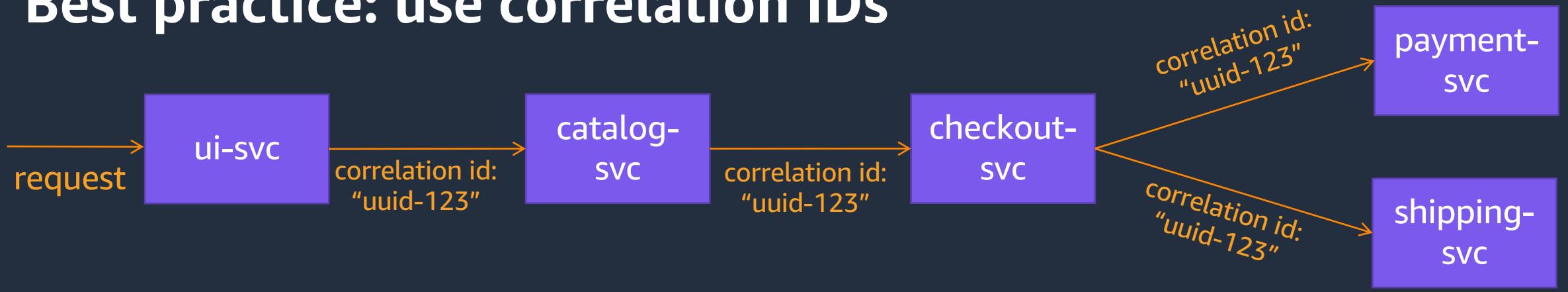


Best practice: microservice owns rollback

- 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: use correlation IDs

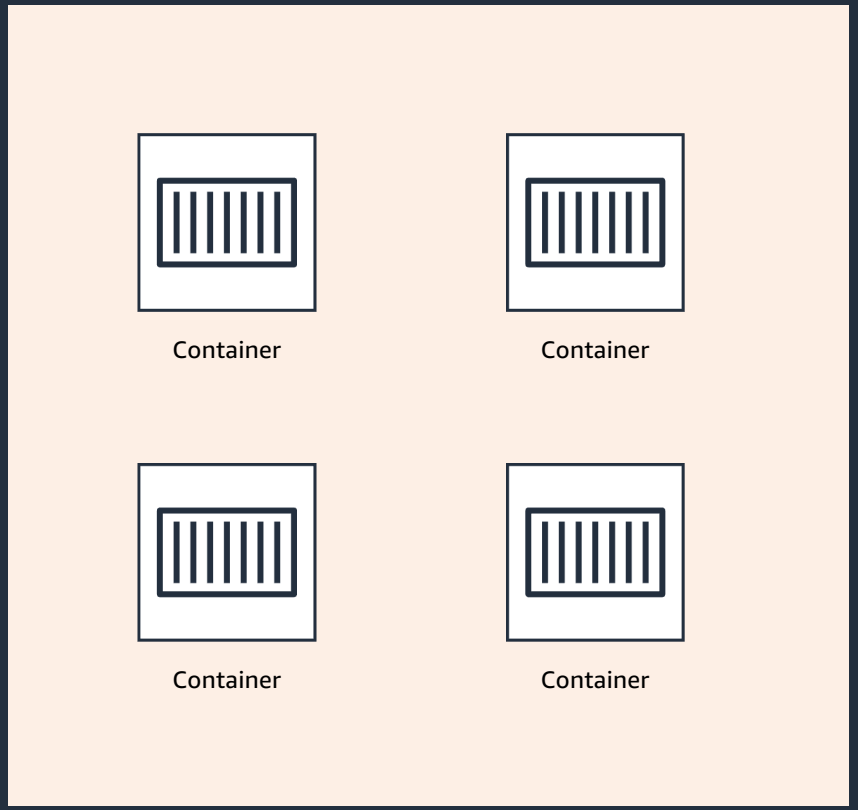



```
09-02-2015 15:03:24 ui-svc INFO [uuid-123] ……
09-02-2015 15:03:25 catalog-svc INFO [uuid-123] ……
09-02-2015 15:03:26 checkout-svc ERROR [uuid-123] ……
09-02-2015 15:03:27 payment-svc INFO [uuid-123] ……
09-02-2015 15:03:27 shipping-svc INFO [uuid-123] ……
```

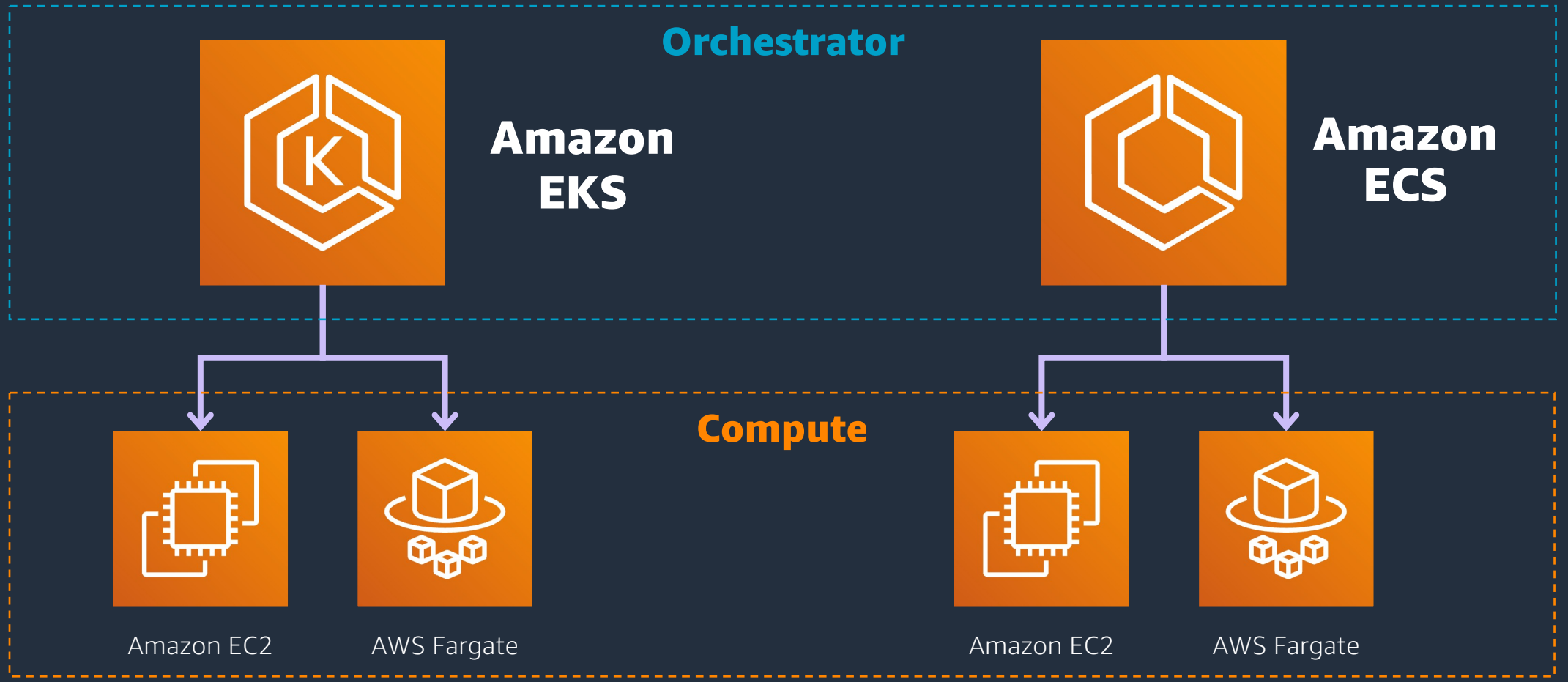
Using Containers

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



Choices for container workloads



Arkansas Admin Office of the Courts

Goal: Modernize the Court Management System.

Solution: Containerize .NET applications using service-oriented application design.

"A core component of the Arkansas Judiciary's Strategic Plan is to Embrace Technology. 'The courts must respond to the changing technological environment by providing court users remote access to information, records, and services.' AWS is helping us to provide those services through this project." – **Marty Sullivan, Arkansas State Court Administrator**

Total cost of ownership
reduced 40 percent

https://aws.amazon.com/blogs/modernizing-with-aws/ar-admin-office-courts-reduced-tco-dotnet-modernization/

Useful resources

Amazon ECS workshop – https://ecsworkshop.com/

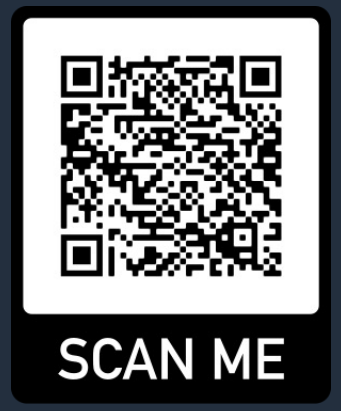
Amazon EKS workshop – https://www.eksworkshop.com/

Monolith to microservices workshop – https://aws.amazon.com/getting-started/hands-on/break-monolith-app-microservices-ecs-docker-ec2/

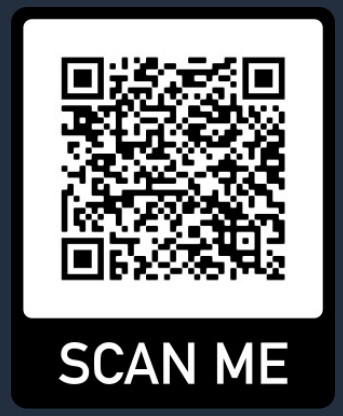
Strangler application pattern – https://martinfowler.com/bliki/StranglerFigApplication.html

Next steps

- AWS Free Tier
- Training / workshops
- AWS Ask-an-Expert booth



Public Sector Blog



SLG Webpage

Contact Us

Application modernization, security, and governance – up next

11:40a	Resilience best practices: Well-architected application on AWS
1:30p	One Observability Workshop
3:10p	Security is top priority

Thank you!

David Stielstra
Solutions Architect
dstiel@amazon.com

Kiran Palsam
Solutions Architect
kapalsam@amazon.com

Please complete the session survey by scanning the QR code

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