AWS Microservices Cheat Sheet

Key Patterns, Services and Concepts

BlowStack

Basics

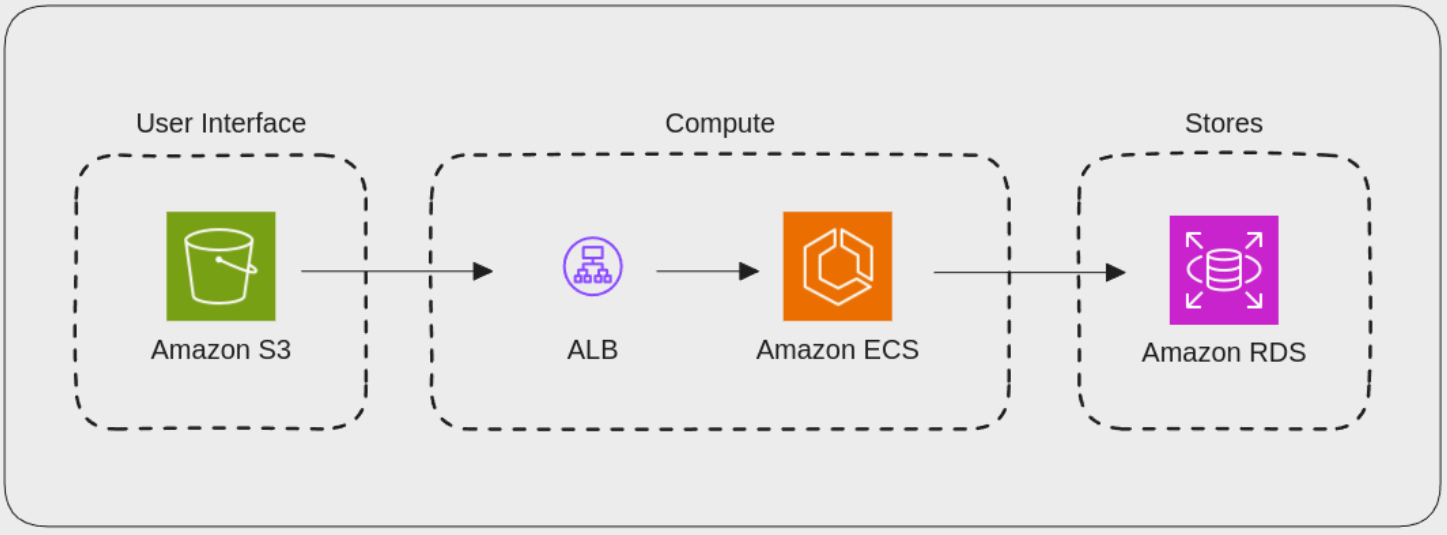
What are Microservices

Microservices are an architectural style where an application is composed of small, independently deployable services, each dedicated to a specific business function and interacting via APIs. This approach to software development speeds up deployment, promotes innovation, improves maintainability, and increases scalability. It hinges on the use of loosely coupled services with well-defined APIs, managed by autonomous teams.

Microservices vs Monolithic

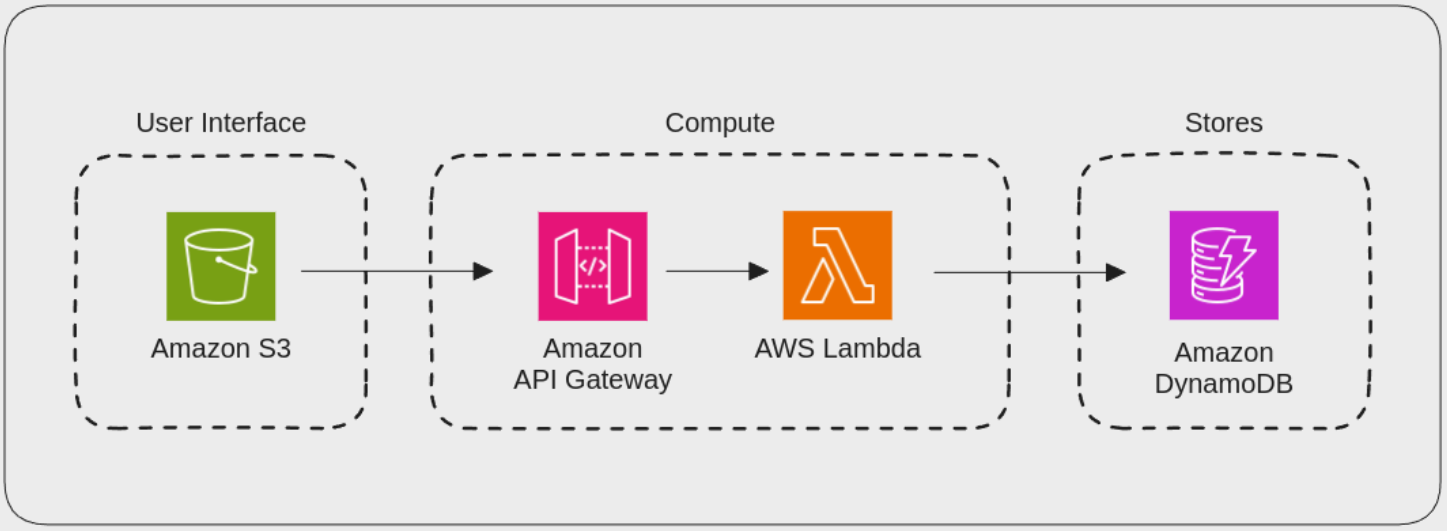
Monolithic architecture builds an application as a single, unified unit, making it easier to develop initially but harder to scale and maintain as it grows. Microservices architecture breaks the application into smaller services, allowing for easier scaling, independent deployments, and better manageability. While monoliths require redeploying the entire application for changes, microservices enable updating individual services without affecting the whole system.

Basic microservices app - Containerization



The diagram illustrates a basic microservices architecture on AWS, where requests are routed through an **Application Load Balancer (ALB)** to microservices managed by **Amazon ECS** (alternatively by EKS), with microservices placed in **ECS tasks**. The interface is hosted on Amazon S3, and data storage is handled by Amazon RDS. Both the store and UI can be omitted depending on the use cases. Using **Fargate** instead of **EC2** for underlying ECS hardware makes the microservices also serverless.

Basic microservices app - Serverless



The diagram demonstrates a **serverless microservices architecture** on AWS, highlighting AWS Lambda's role. **AWS Lambda** serves as the compute layer, running discrete functions in response to **API Gateway** requests. Each Lambda function can represent an independent microservice, handling specific business logic or processing tasks. These functions are stateless and can be developed, deployed, and scaled independently, embodying the microservices principles of modularity, scalability, and autonomous deployment. Alternatively to Lambda, **AWS Fargate** can be used to host microservices.

Service Discovery

What is Service Discovery

Service discovery in AWS microservices architecture dynamically tracks and manages the network locations of service instances, allowing microservices to find and communicate with each other without hardcoding addresses. Using AWS Cloud Map or integrated services like ECS with Elastic Load Balancing, instances register themselves at startup and deregister on shutdown. This enables dynamic scaling, resilience, and simplified configuration management. Health checks ensure traffic is routed only to healthy instances, enhancing fault tolerance and maintaining a decoupled, flexible system.

Client-Side Discovery

Clients query a service registry to obtain service instances and handle the routing logic themselves.

Server-Side Discovery

Clients send requests to a load balancer or router, which forwards the requests to the appropriate service.

DNS-Based Discovery

Services use DNS to resolve the network locations of other services,

Service Mesh

A dedicated infrastructure layer manages service-to-service communication, including discovery, using sidecar proxies and a control plane.

DNS-based

ECS Service Discovery

Service discovery through its DNS-based method or by integrating with AWS Cloud Map.

Amazon Route 53

Works with ECS and EKS. For ECS, Route 53 utilizes the ECS Service Discovery feature, which employs the Auto Naming API to automatically manage services discovery.

AWS Cloud Map

AWS Cloud Map provides a dynamic API-based service discovery solution, ensuring that changes are consistently propagated across your services.

Mesh-based

Amazon VPC Lattice

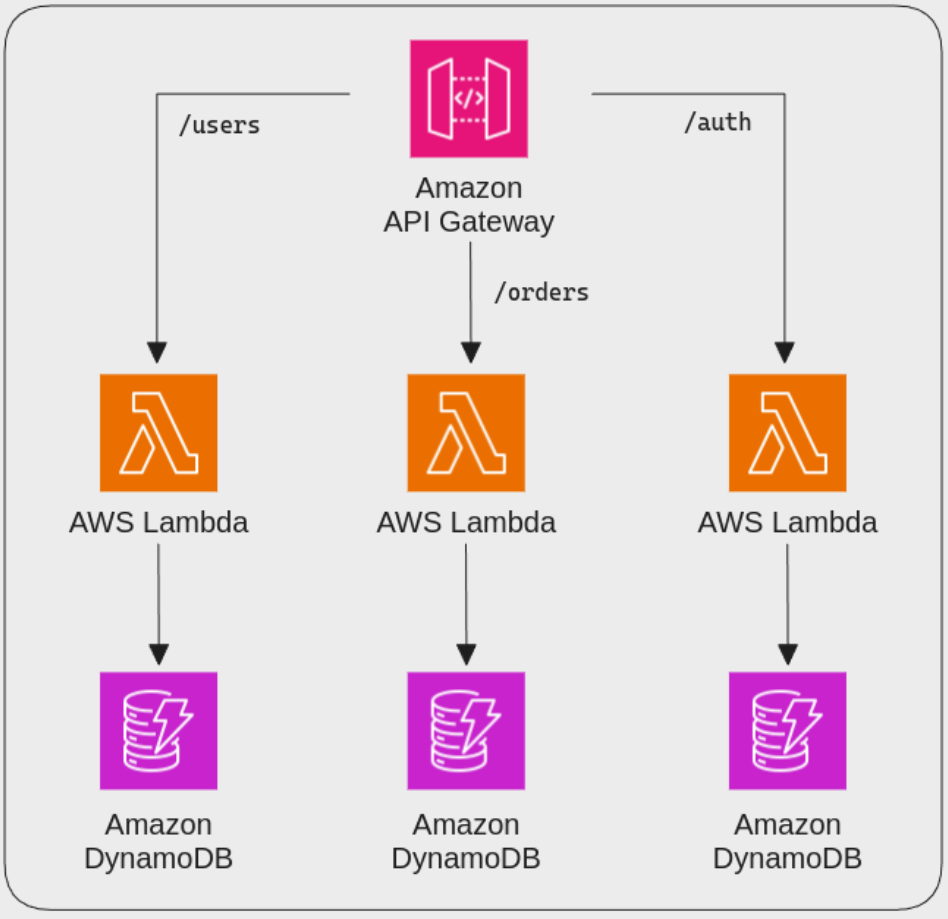
An application networking service that connects, monitors, and secures communications between services, allowing developers to focus on building business-critical features while managing network traffic, access, and monitoring across various compute services.

AWS App Mesh

Built on the open-source Envoy proxy, addresses advanced needs with sophisticated routing, load balancing, and comprehensive reporting, and unlike Amazon VPC Lattice, it supports the TCP protocol.

Distributed data management patterns

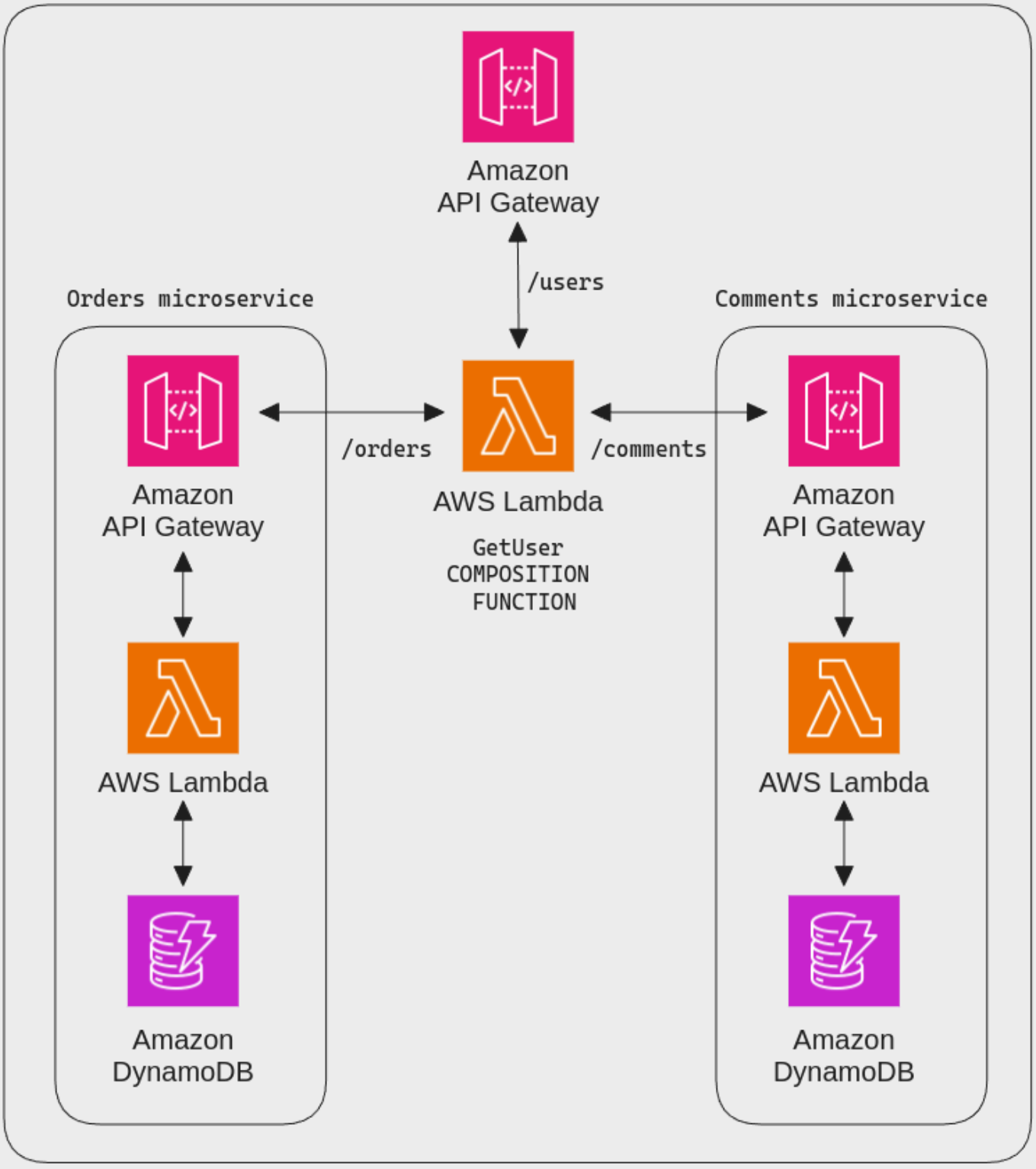
Database-per-service pattern



Endpoints route to individual Lambdas, deployed as microservices. Each interacts with its own DynamoDB table, ensuring isolation.

Use this pattern if loose coupling, different compliance or security requirements, or more granular control of scaling are needed between microservices. However, it can be challenging to manage multiple databases, implement complex transactions, and meet two of the CAP theorem requirements.

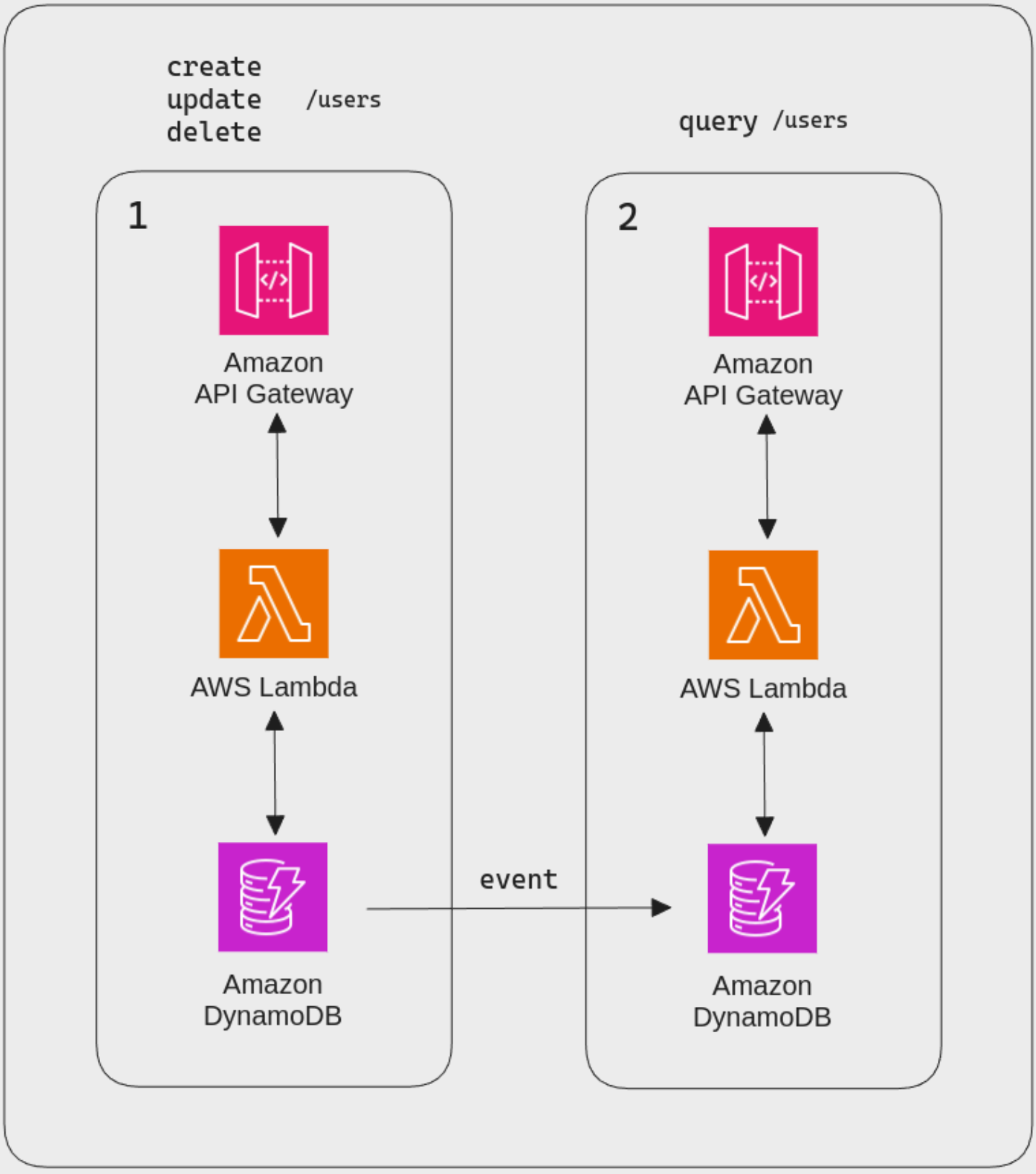
API composition pattern



Idea of the API composition pattern is to handle and process data from multiple microservices in memory, enabling efficient data retrieval and aggregation before sending the response back to the client.

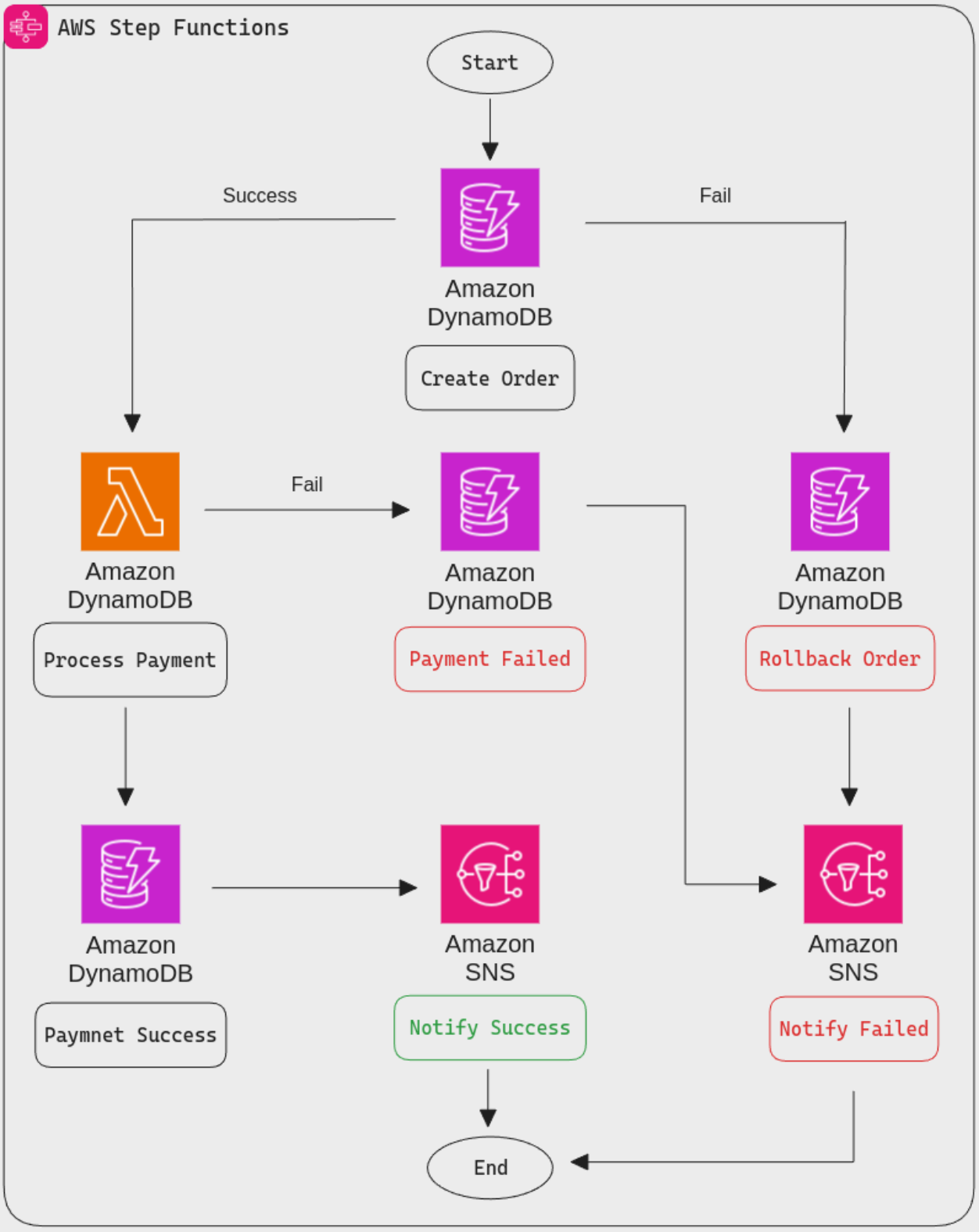
The API composition pattern offers a simple way to gather data from multiple microservices but may struggle with complex queries, reduce system availability with more connected microservices, and increase operational costs due to higher network traffic.

CQRS pattern



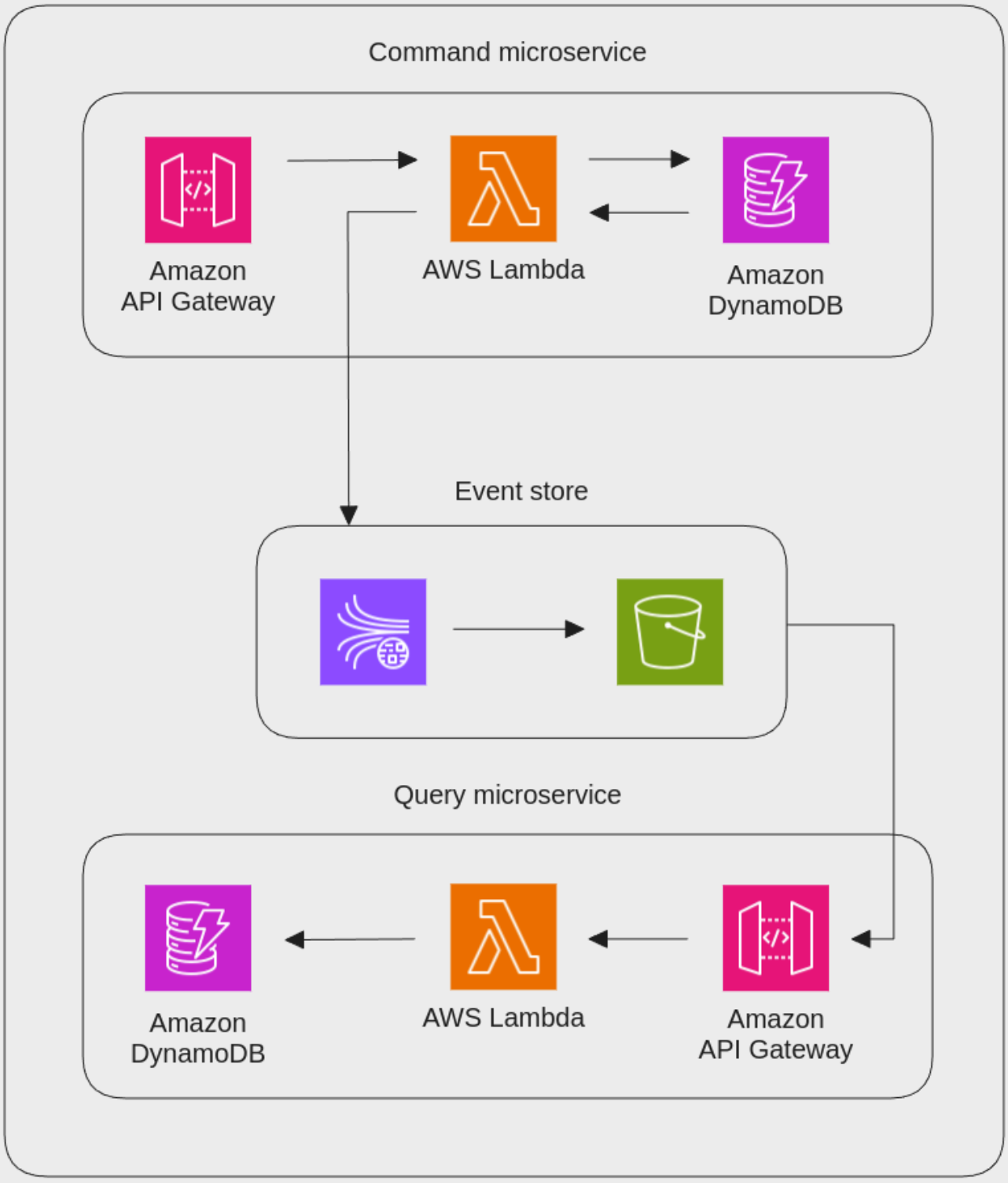
The Command Query Responsibility Segregation (CQRS) pattern separates data mutation (commands) from data retrieval (queries). It is useful when updates and queries have different requirements for throughput, latency, or consistency. CQRS divides the application into two parts: the command side handles create, update, and delete requests, while the query side manages data retrieval, often using read replicas

Saga pattern



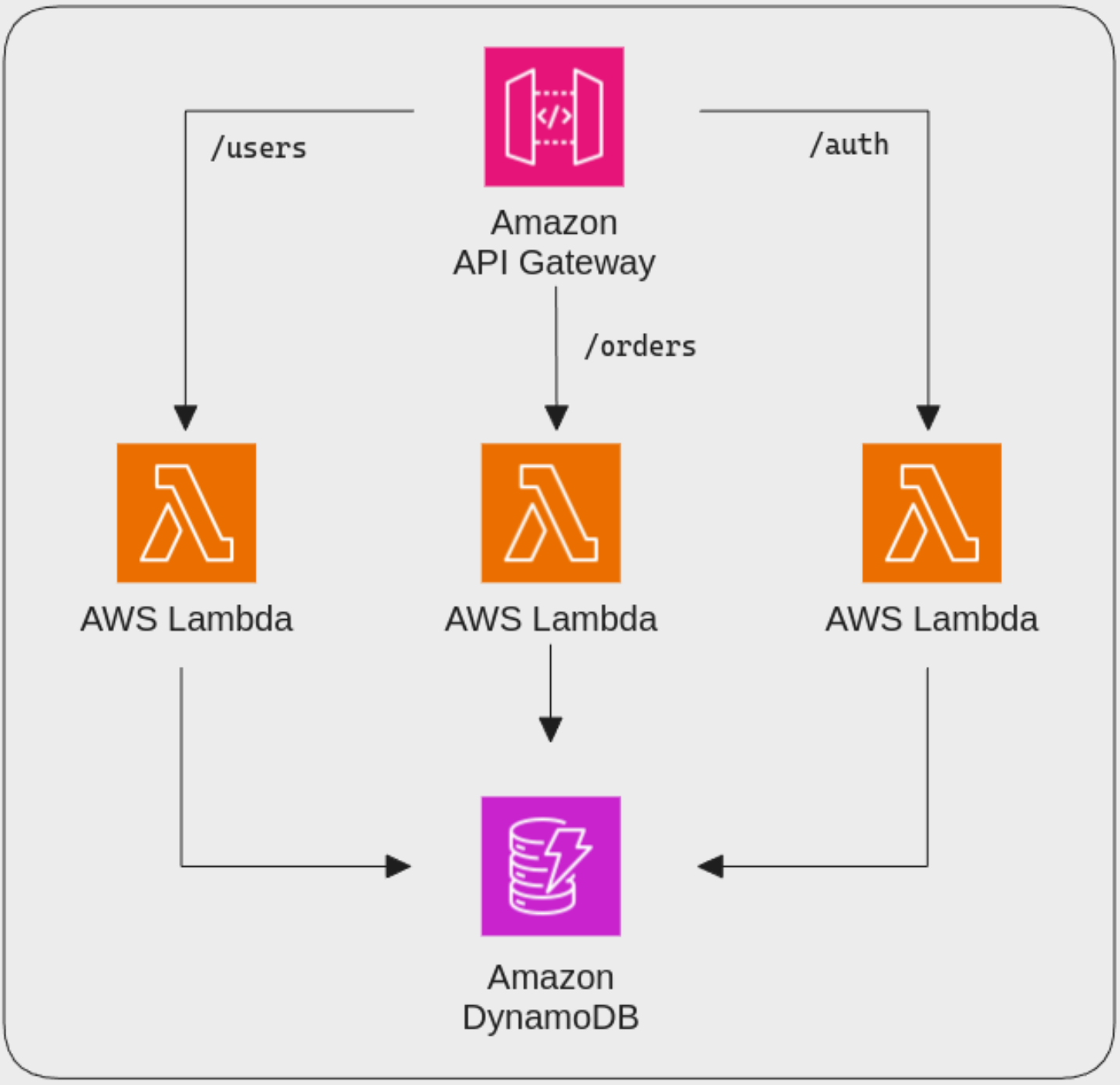
The saga pattern is a failure management strategy that ensures data consistency in distributed applications by coordinating transactions across multiple microservices. Each microservice publishes an event after a transaction, triggering the next transaction based on the event's outcome. This pattern is ideal for maintaining consistency without tight coupling, handling long-lived transactions, and enabling rollbacks. However, it is complex to debug and requires a sophisticated programming model for compensating transactions and undoing changes.

Event sourcing pattern



The event sourcing pattern, often paired with CQRS, optimizes performance, scalability, and security by storing data as events. Microservices replay these events to determine their states, providing current state visibility and historical context. This pattern enables state reconstruction at any time, creating an audit trail and simplifying debugging, though it results in eventual consistency. Implementable with Amazon Kinesis Data Streams or Amazon EventBridge, it requires the Saga pattern for data consistency across microservices.

Shared-database-per-service pattern



In the shared-database-per-service pattern, multiple microservices share the same database. Before adopting this pattern, assess the architecture to avoid hot tables and ensure all database changes are backward-compatible. Consider this pattern if you want minimal code refactoring, enforce ACID transactions for data consistency, prefer maintaining a single database, face difficulties with database-per-service due to interdependencies, and aim to avoid a complete data layer redesign.