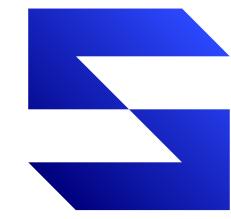


# 서비스의 장점을 극대화하는 이벤트 기반의 응용 아키텍처 “3factor app” 의 소개 및 AWS 서비스로 구현해보기

Serverless Operations, Inc

김선우

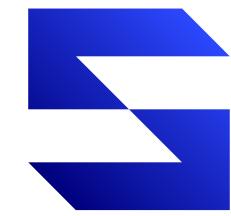


## 간단한 자기소개



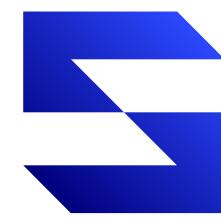
- 김선우 / Sonu Kim
- COO at Serverless Operations, Inc.  
Full-stack AWS Serverless Engineer
- AWS Community Builders (2023 - )
- 일본 Serverless 커뮤니티 및  
글로벌 AWS 커뮤니티에서 활동



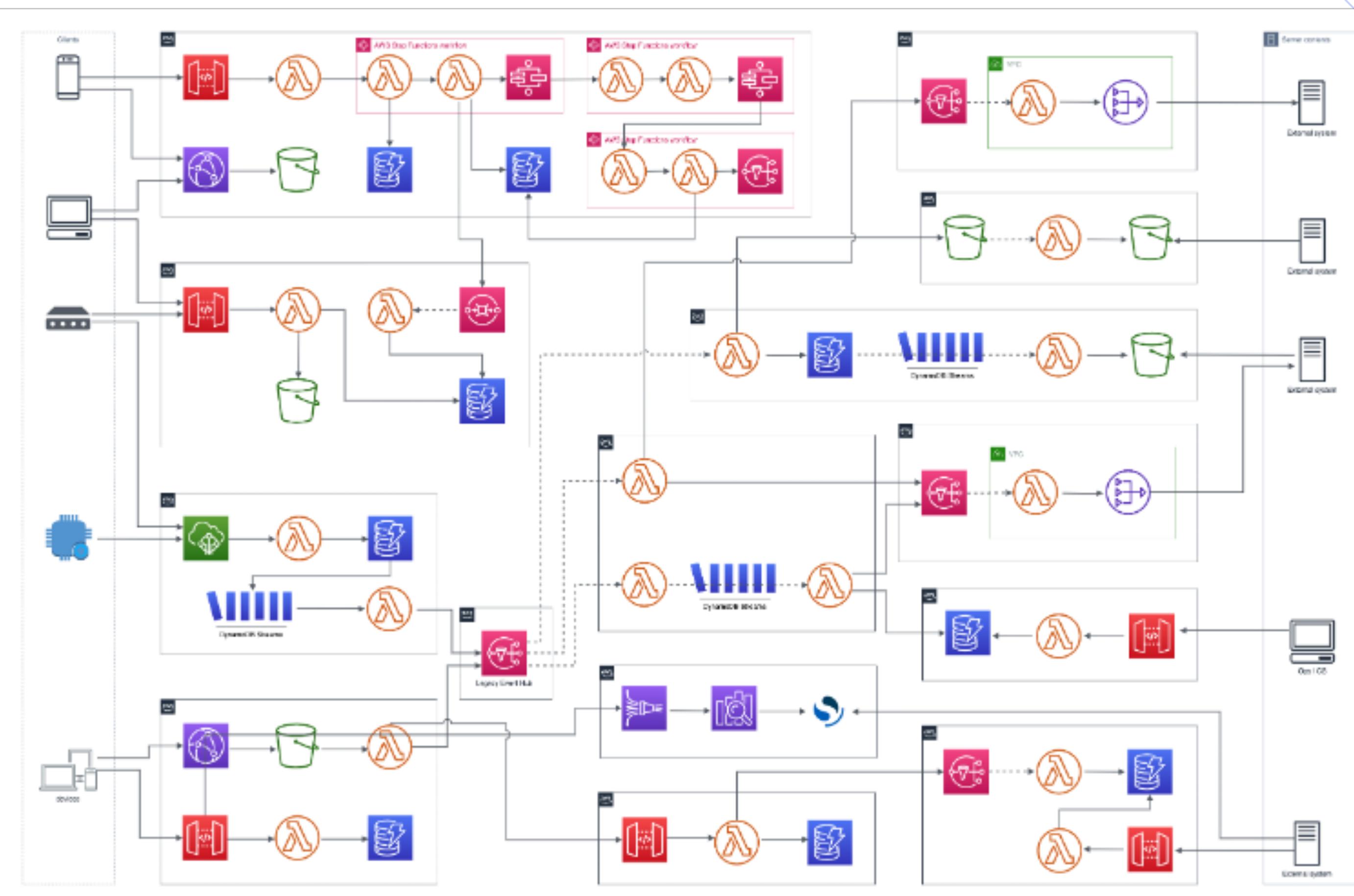


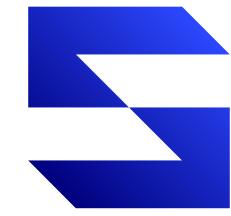
## Agenda

- 3factor app 이 필요한 배경 및 정의
- 3factor app 의 구성 및 설계
- AWS 서비스로 구축해보기

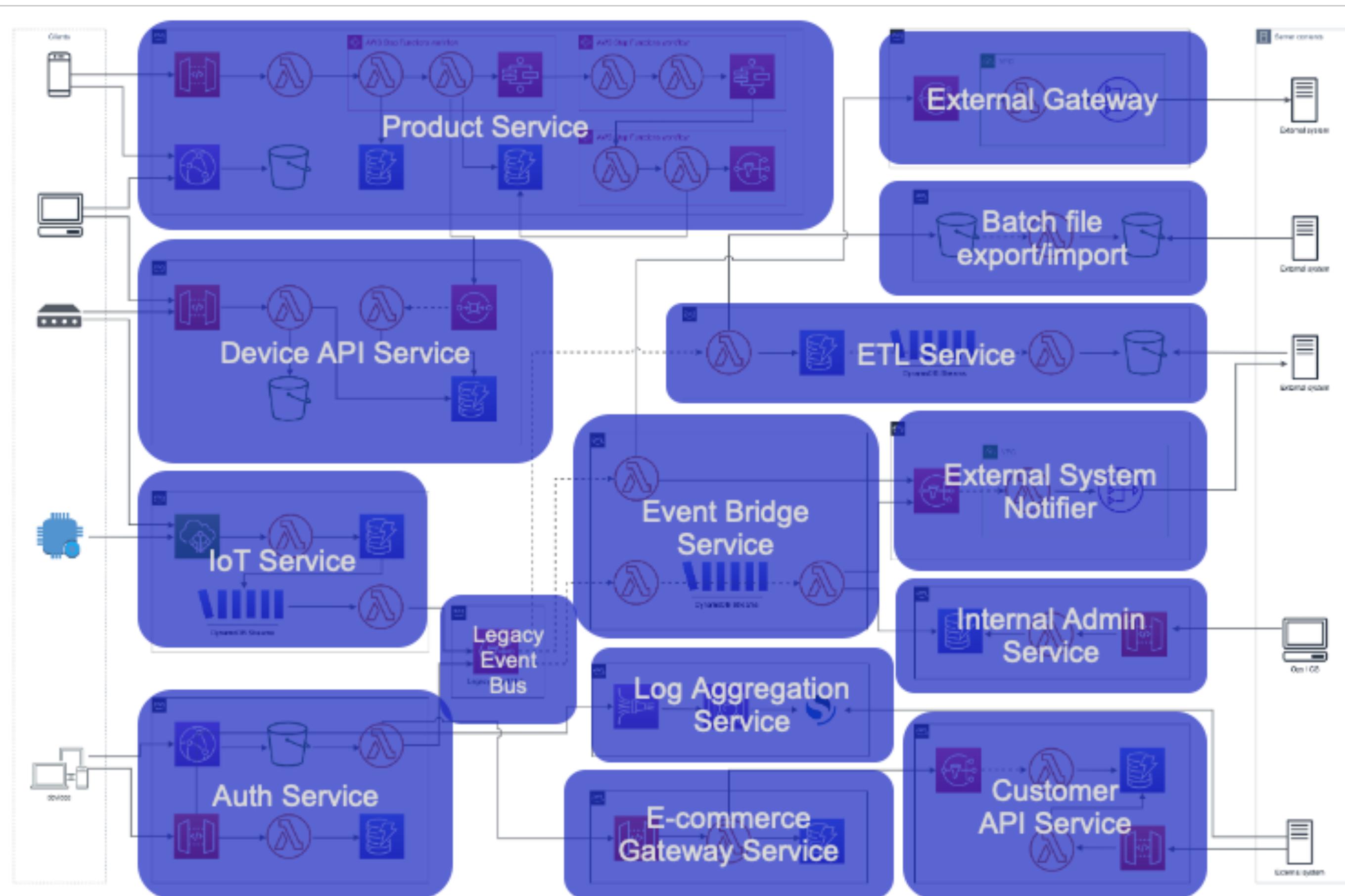


# 서비스의 복잡성 문제

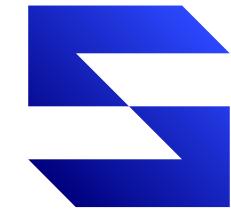




# 무엇이 서비스를 복잡하게 만들까?

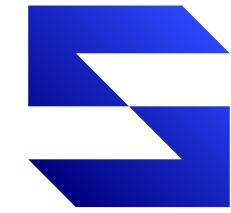


- 너무 많은 마이크로 서비스와 연속적인 비동기 처리
- 비대해진 워크플로우 및 응답시간이 길 어진 API 호출 프로세스
- 많은 외부시스템과의 결합 및 cross-account 접근 설정으로 인한 번거로움



## 서비스 아키텍처를 보다 심플하게 하기 위한 동기부여

- 비즈니스 로직의 간소화, 합리화 (합리적인 비즈니스 로직 = 좋은 서비스 애플리케이션 설계)
- 서비스 구성을 유지 함으로서 서버나 컨테이너 등 인프라에 대한 운용부담을 지지 않는 것
- 팀의 에너지와 시간을 절약해서 계속되는 비즈니스 요구에 즉각 대응하기 위한 작업에 몰두하는 것

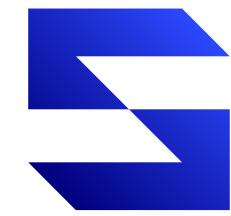


## 3factor app 이란?

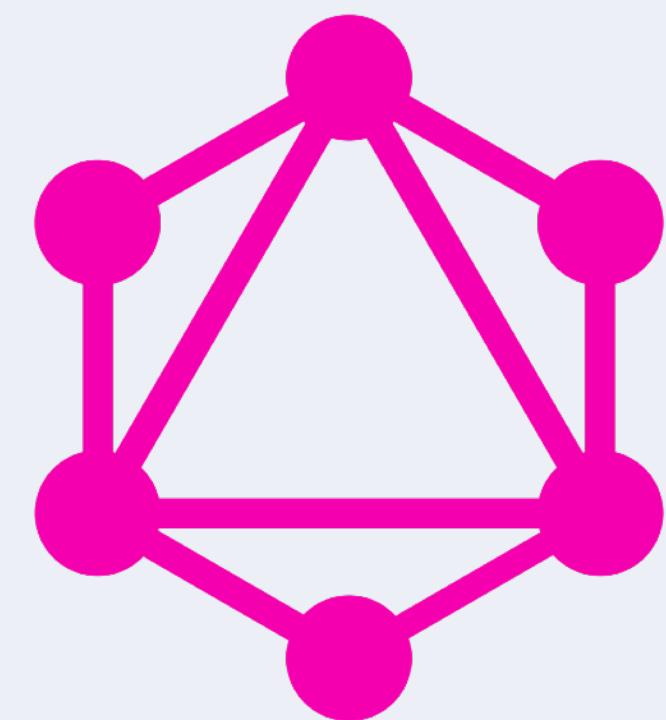
*3factor app is an architecture pattern for modern full-stack apps. 3factor enables building apps that are robust and scalable from the get go by using modern API architectures along with the power of Cloud.*

3factor app 은 모던 풀스택 애플리케이션을 위한 아키텍처 패턴입니다. 모던 API 설계와 클라우드의 힘을 이용해서 처음부터 견고하고 확장성이 있는 애플리케이션 구축이 가능합니다.

<https://3factor.app/>

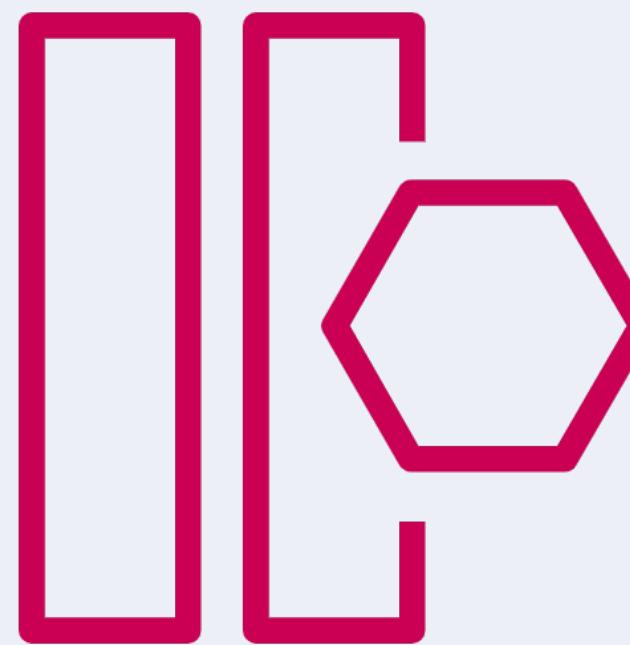


## 세가지 팩터 “3 factors” 란?



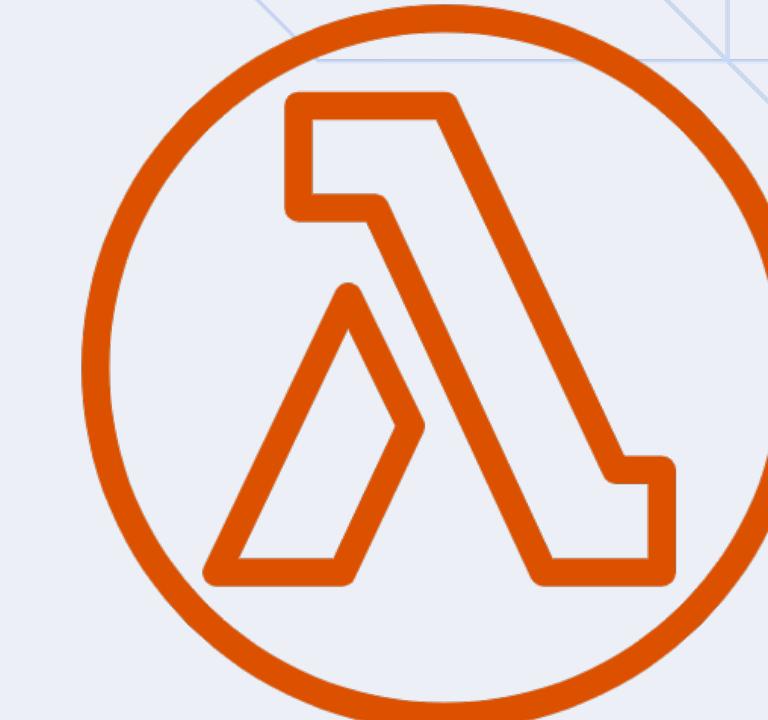
GraphQL

Subscription을 이용한  
리얼타임 상호작용



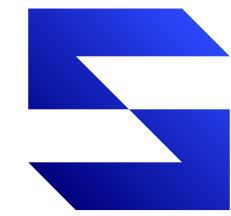
Event

신뢰성 있는 이벤트처리



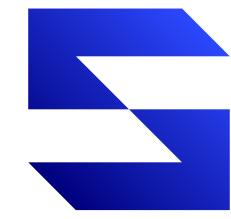
Serverless

비동기 서비스 함수를  
사용한 비즈니스 로직 실행

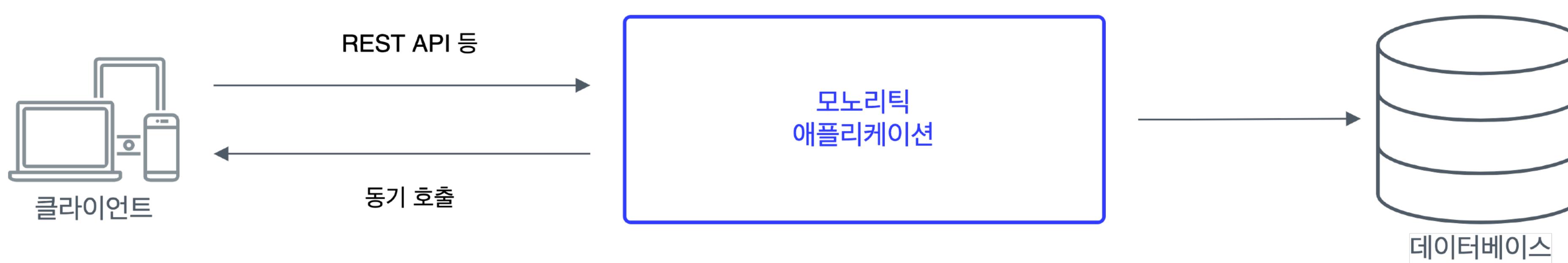


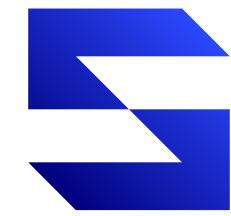
## 3factor app 의 기본적인 구성

- 클라이언트는 GraphQL API 를 경유해서 상태를 입력
- 상태 변경에 따라 이벤트를 작성 및 발행
- 비동기적으로 발행되는 이벤트에 따라 비즈니스 로직을 호출 및 처리
- 클라이언트는 가장 최신의 상태를 구독(subscription) 해서 실시간으로 반영

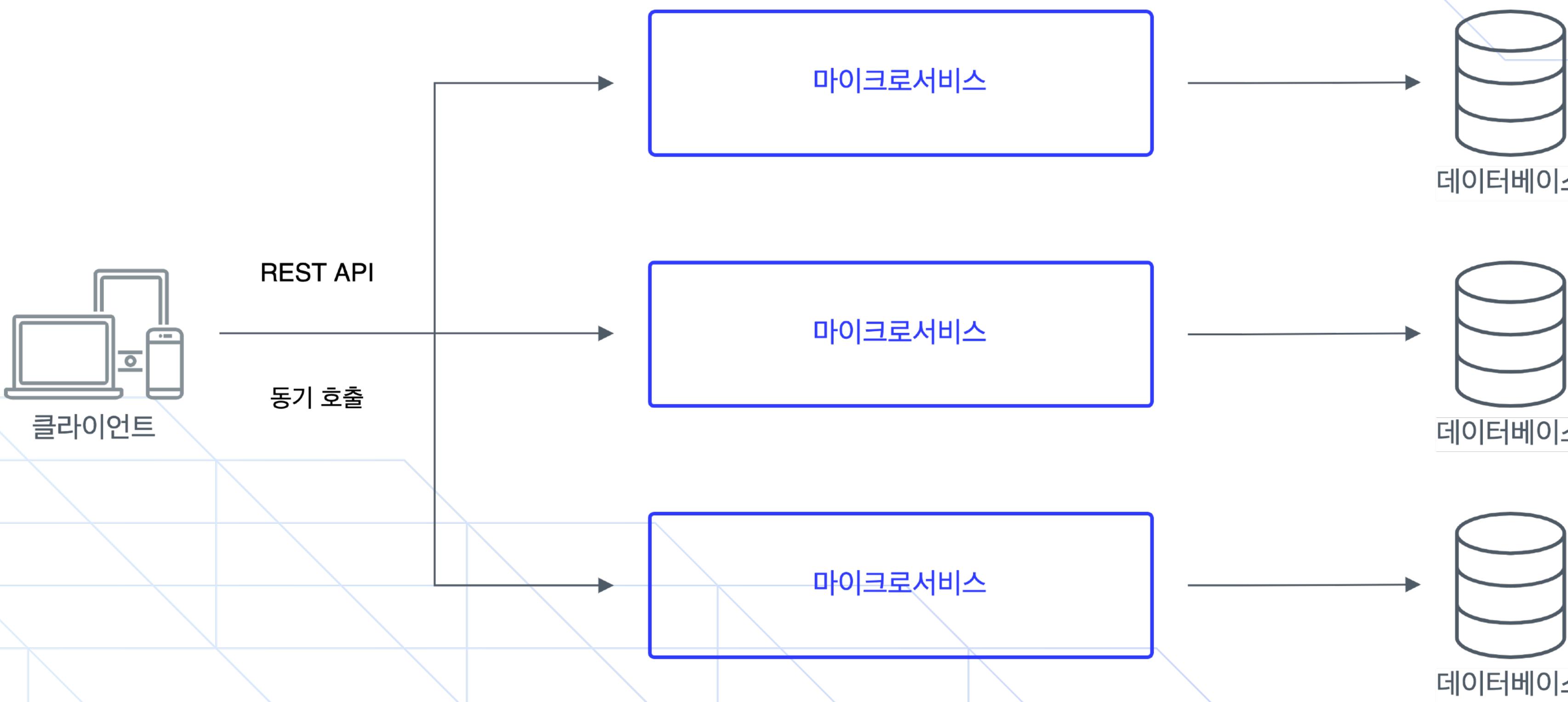


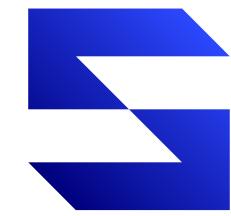
## 일반적인 모노리스 구성의 예시



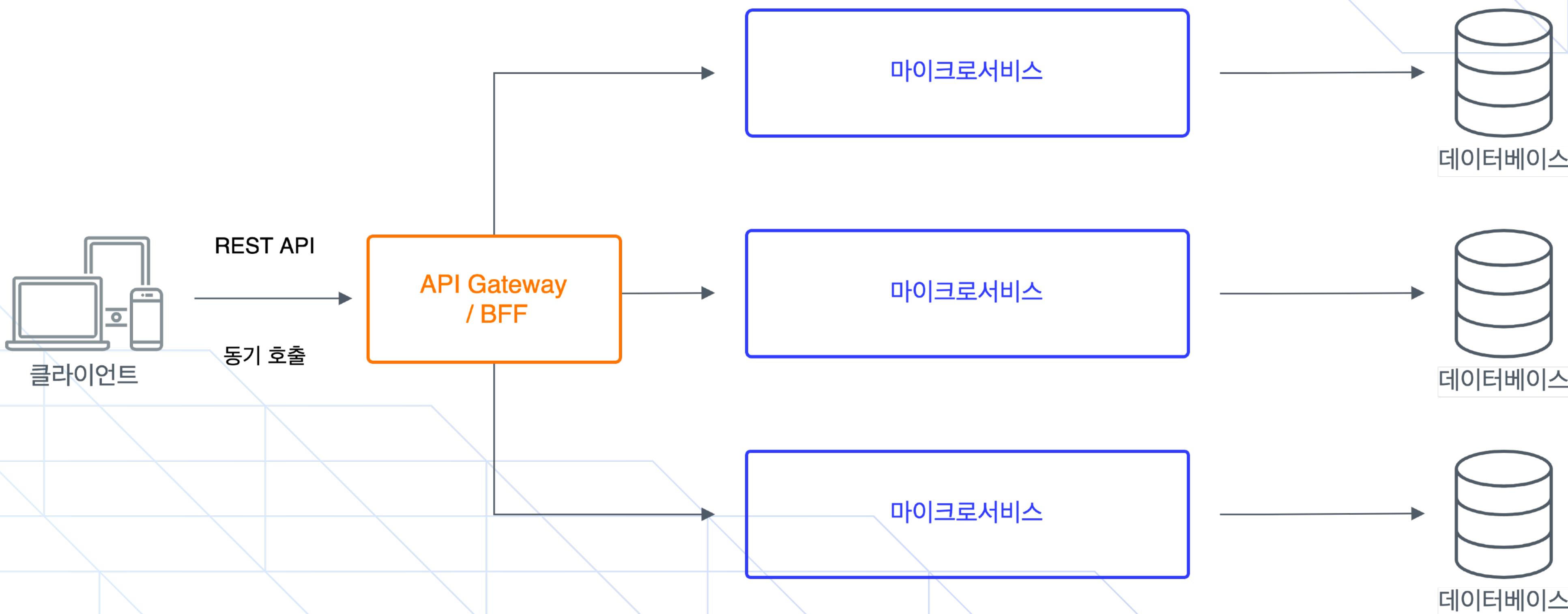


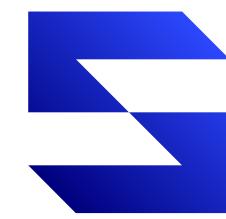
# 마이크로 서비스 구성이라도 기본적으로 동일



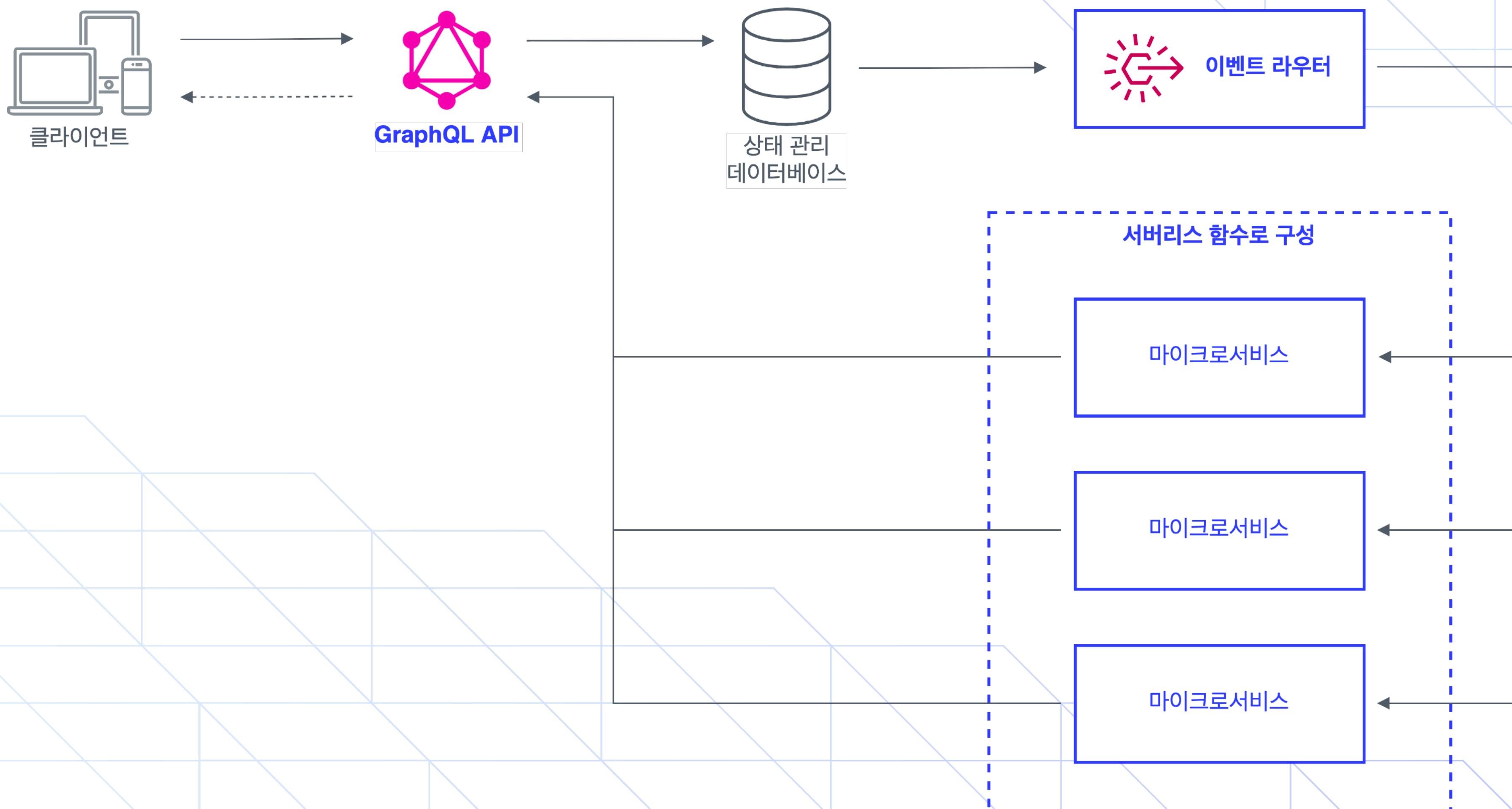


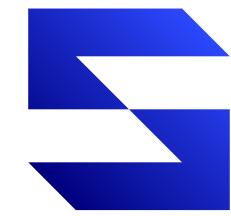
## BFF(Backend for Frontend) 를 구성해도 기본적으로 동일



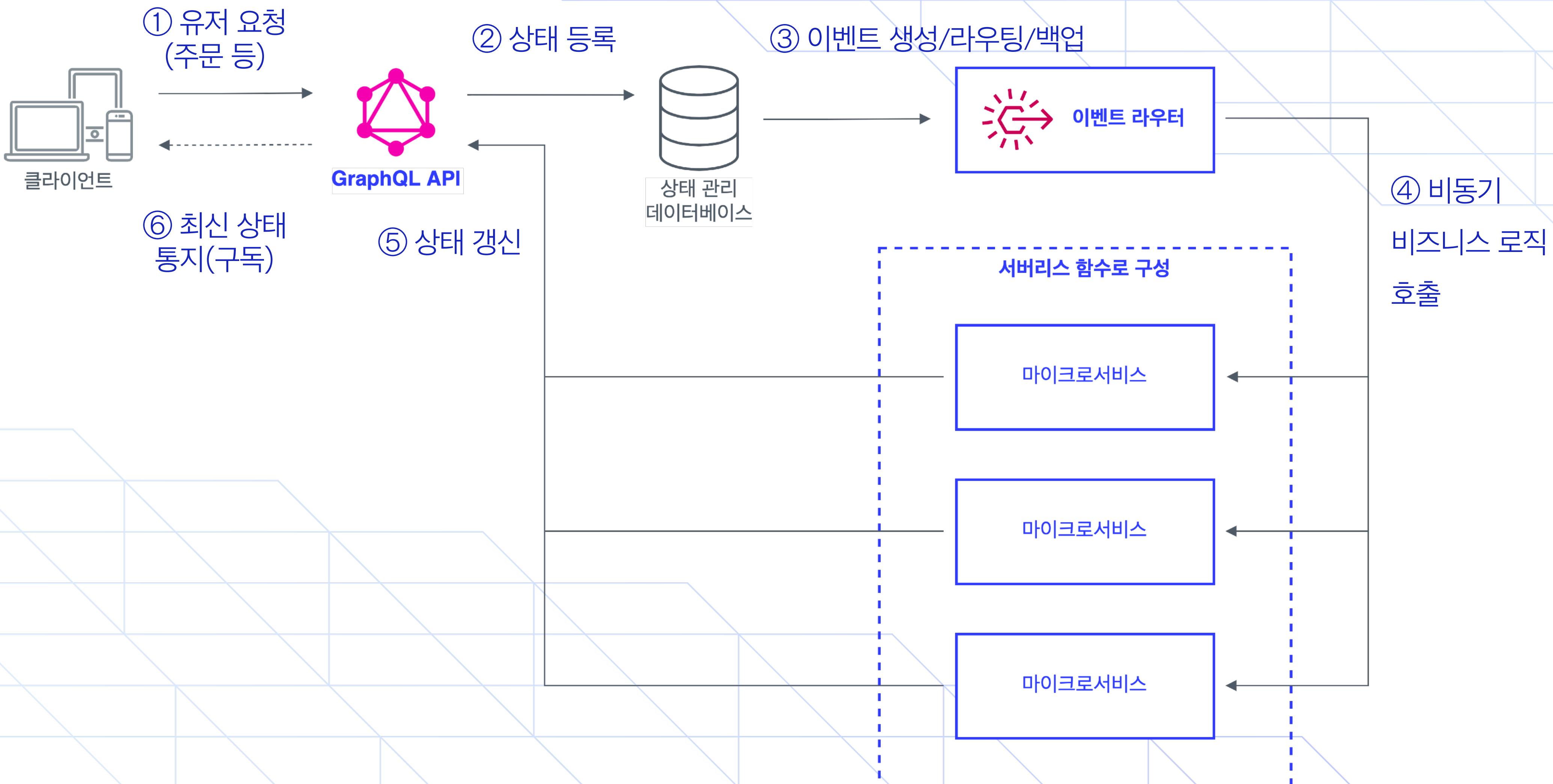


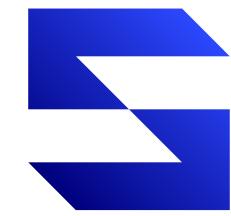
## 3factor 아키텍처





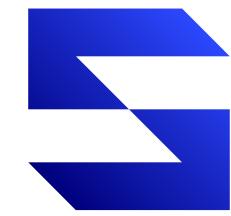
## 3factor 아키텍처



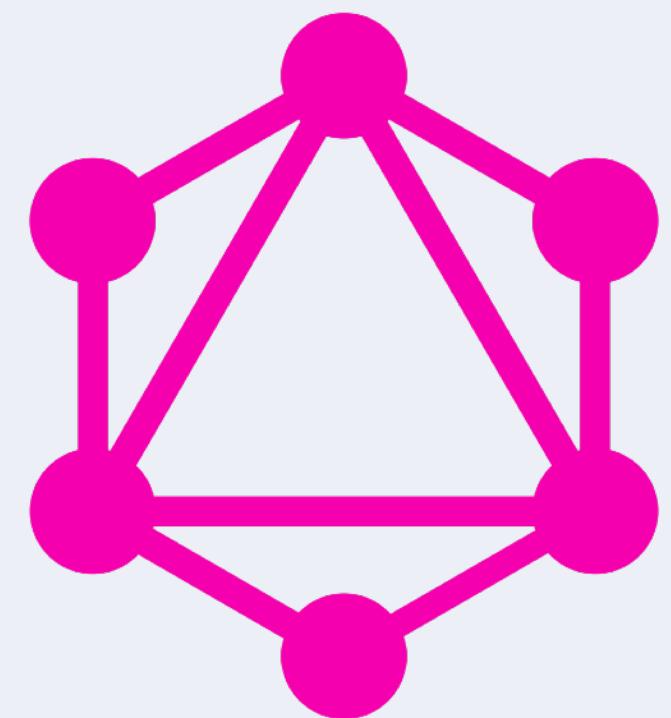


## 3factor 아키텍처의 이점

- 확장성 (Scalability)
- 탄력성 (Resilience)
- 사용자 경험 (User Experience)
- 개발자 경험 (Developer Experience)

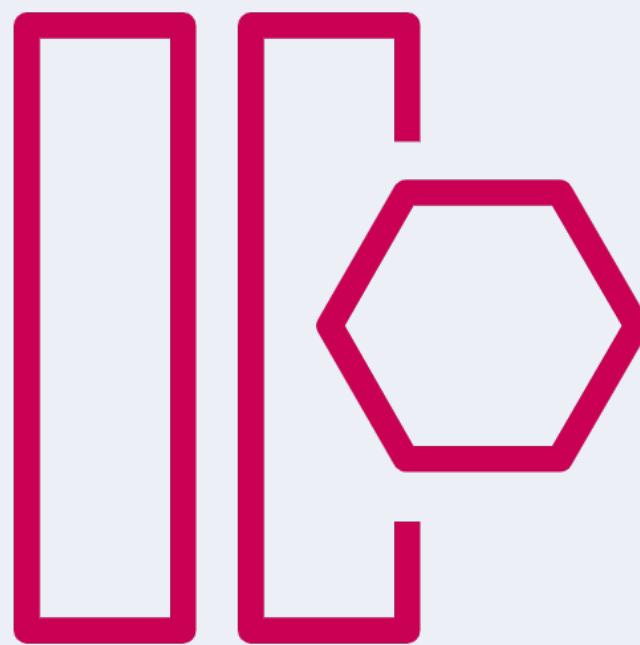


## 세가지 팩터 “3 factors” 란?



GraphQL

Subscription을 이용한  
리얼타임 상호작용



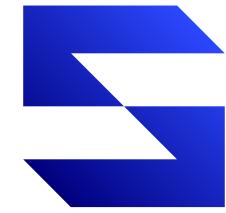
Event

신뢰성 있는 이벤트처리



Serverless

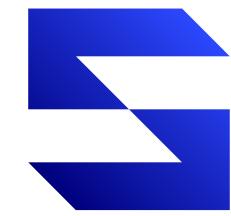
비동기 서비스 함수를  
사용한 비즈니스 로직 실행



## GraphQL이란?

- 오픈소스의 API 쿼리를 위한 언어 및 런타임
- 스키마 정의에 의한 타이핑, 필요한 항목을 골라서 요청할수 있는 쿼리, 데이터 구독(Subscription) 기능에 의한 실시간 상호작용 개발 가능
- Facebook, GitHub, Shopify, Netflix, Uber eats 등 적용 사례 다수





# GraphQL의 세가지 기본 기능

```
query GetNews {  
  getNews(id: "news_1") {  
    id  
    title  
    author  
    date  
  }  
}
```

Queries

작성한 스키마의  
데이터를 요청

```
mutation createNews {  
  createNews(input: {...}) {  
    id  
    author  
    date  
  }  
}
```

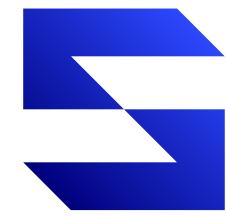
Mutations

데이터 작성/업데이트/삭제

```
subscription onCreateNews {  
  onCreateNews {  
    id  
    title  
    author  
  }  
}
```

Subscriptions

실시간으로 변화(mutation)된  
데이터를 구독(통지)



# GraphQL의 이점

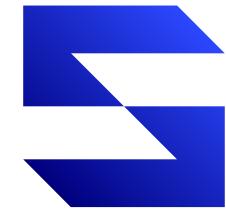
- 정적 타이핑과 스키마 정의

```
type Restaurant {  
    id: ID!  
    name: String!  
    address: String!  
    type: String!  
    menu: [Menu]!  
}  
  
type Menu {  
    name: String!  
    price: Int!  
    isAllergyFree: [String]!  
    isForVegetarian: Boolean!  
}
```

- 특히 프론트엔드에서의 개발자경험 및 개발 속도 향상

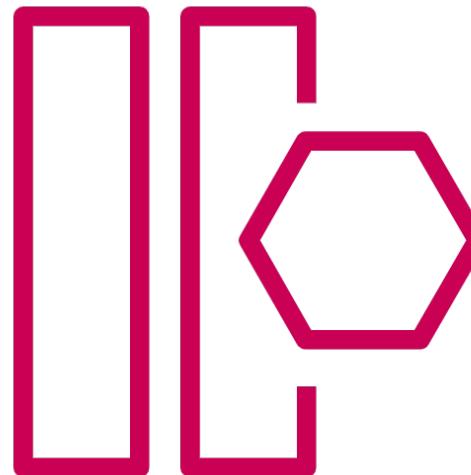
- 유연하며 확장성이 있는 데이터 취득 및 스키마 구조의 변경이 가능

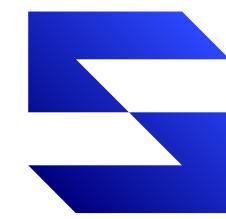
- 구독(Subscription)에 의한 실시간 데이터 통지 및 반영



## 이벤트란 무엇일까?

- 시스템의 “상태”가 변화 했다는것을 가르키는 신호 및 변화한 상태를 서비스간에 공유하기 위한 매커니즘
- 이벤트에는 의미를 나타내기 위한 의도가 포함되어 있으며 주로 과거형의 동사로 표현  
“news\_created”, “order\_cancelled”, “payment\_approved”
- 서비스간의 공통항목을 가지고 컨텍스트를 구성  
“order\_id”, “booking\_id”, “apply\_id”





## 푸드 딜리버리 서비스의 예시



고객



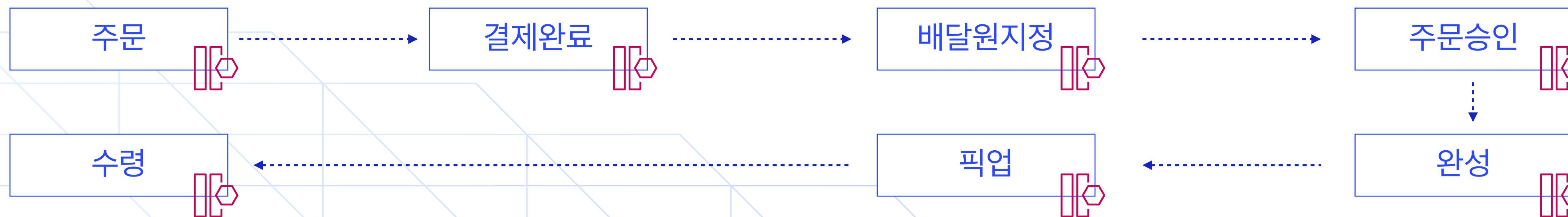
결제 서비스

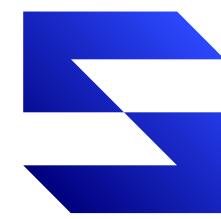


배달원



레스토랑





## 푸드 딜리버리 서비스의 예시



고객



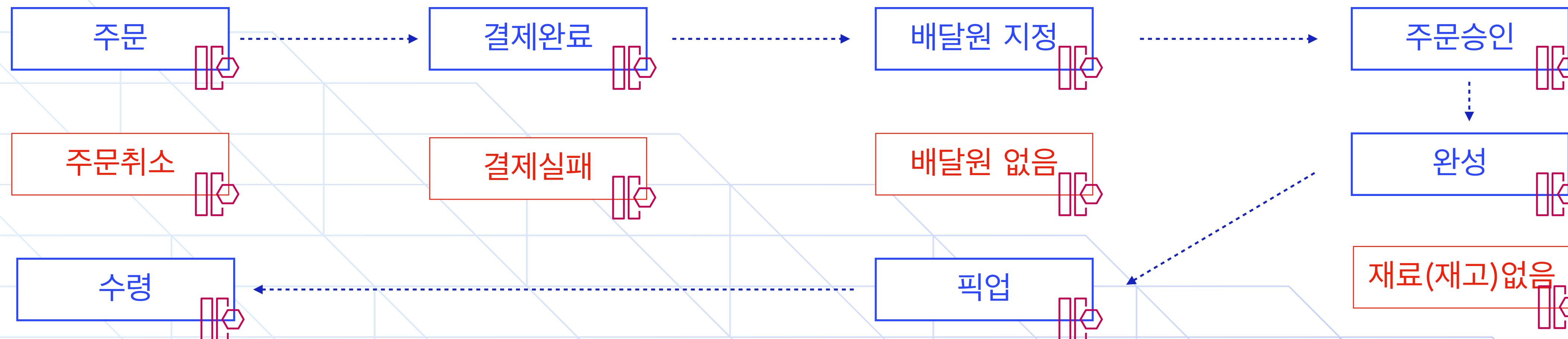
결제 서비스

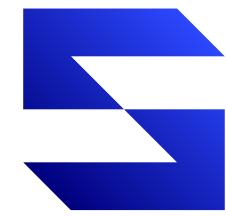


배달원

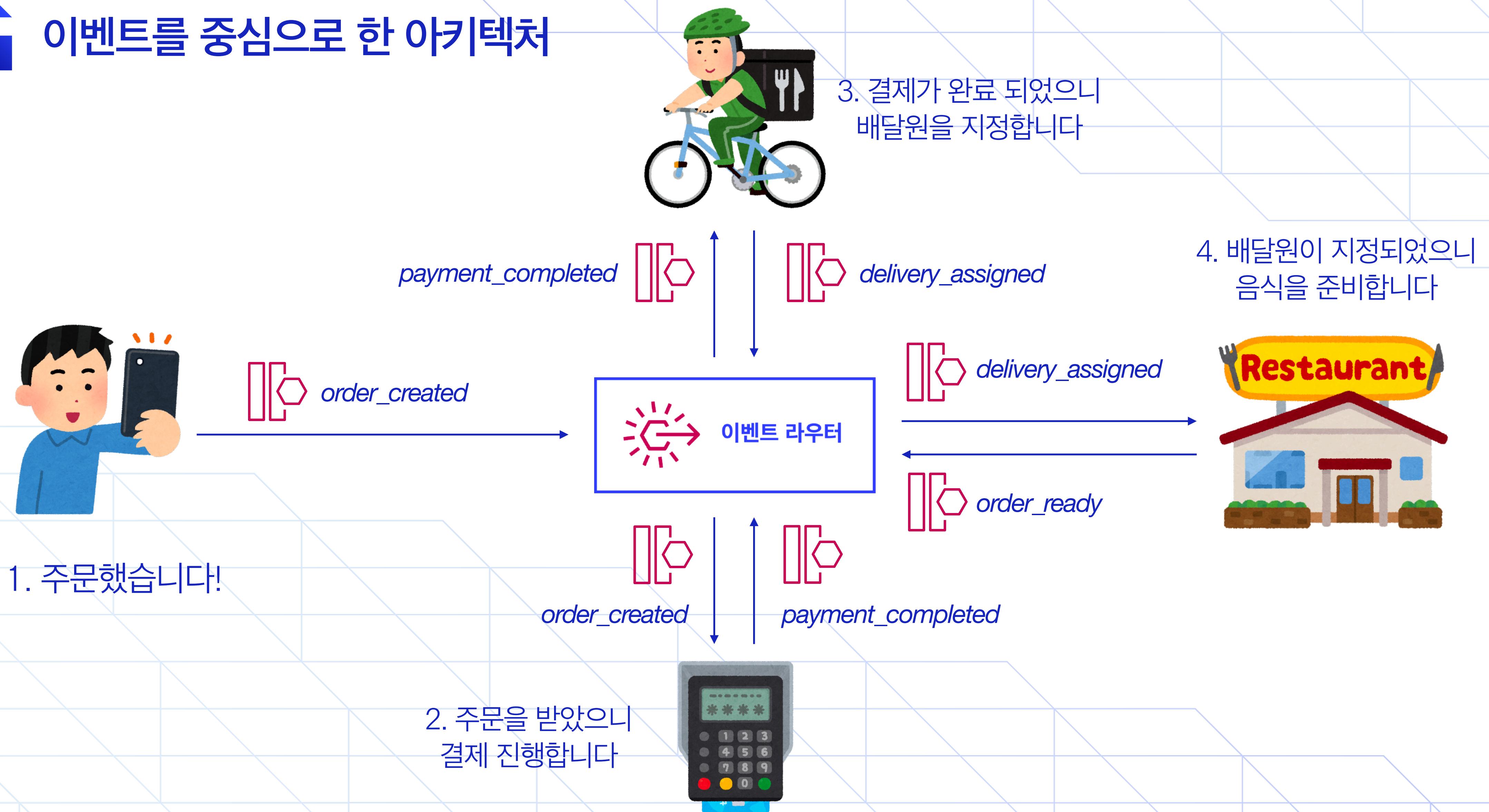


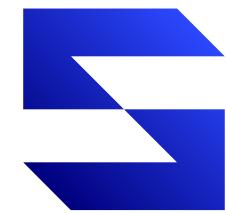
레스토랑





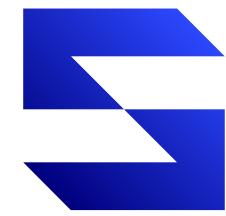
## 이벤트를 중심으로 한 아키텍처





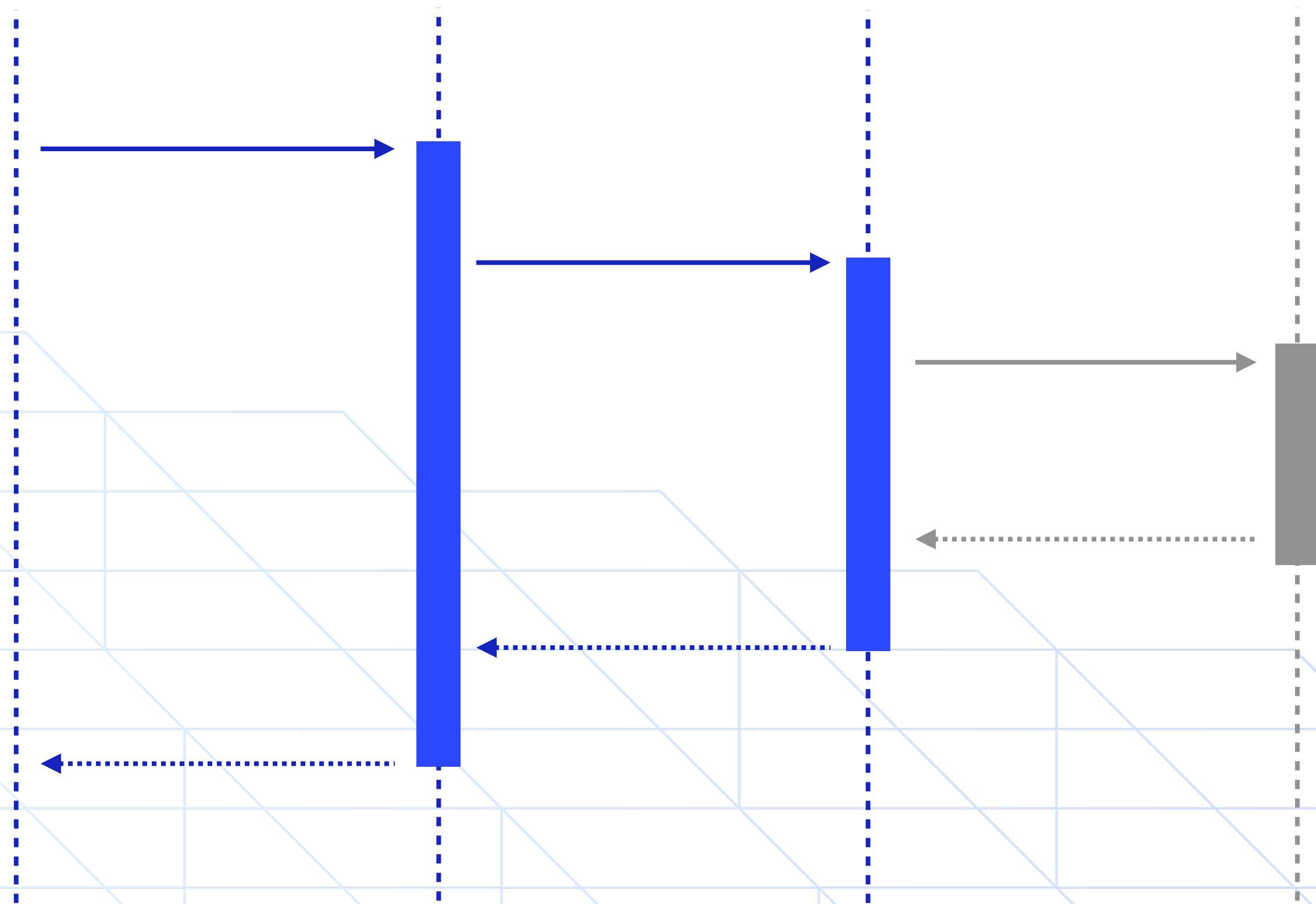
## 이벤트는 “명령” 이 아닌 “관측” 하는 대상



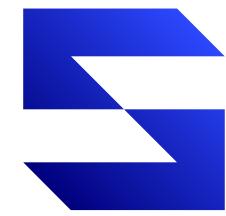


## 동기적인 명령수행 구조의 단점

클라이언트      서비스 A      서비스 B      서비스 C (추가)

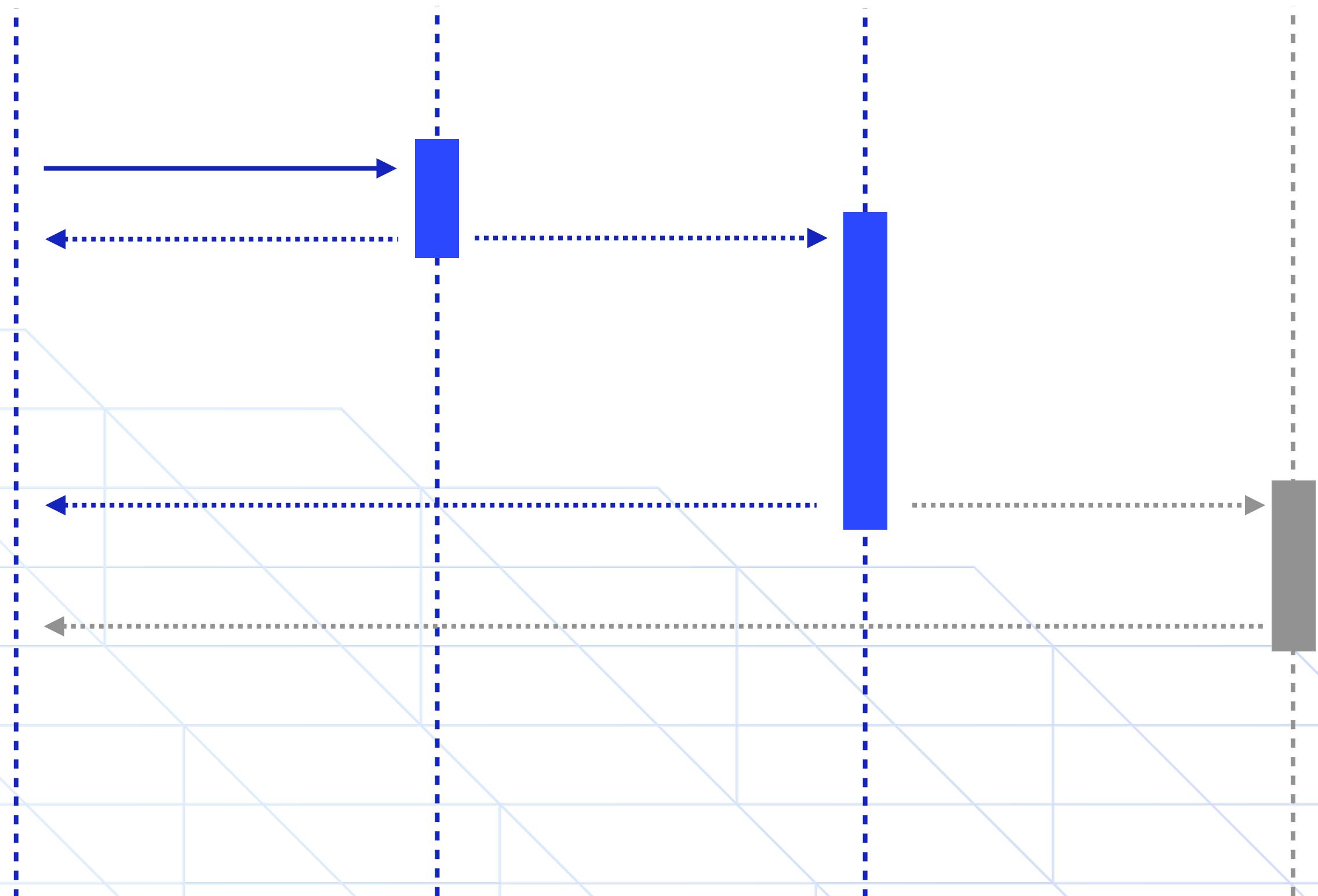


- 다운스트림에 서비스를 추가 할수록 응답시간 및 장애 포인트가 증가
- 추가 개발시 기존 서비스에 코드 수정이 필요
- 각 서비스간이 밀접하게 결합하여 개발 및 배포 의존관계가 증가

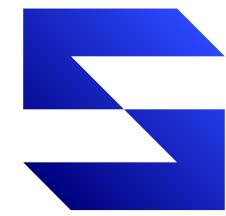


## 비동기적 이벤트에 의한 이점

클라이언트      서비스 A      서비스 B      서비스 C (추가)



- 각 서비스를 독립적으로 관리운용 할 수 있음
- 다른 서비스에 영향을 최소화하여 비교적 쉽게 서비스를 추가 가능
- 실시간 상호작용과 느슨한 결합에 의해 사용자경험과 개발자경험을 둘다 만족 시킬수 있음



# 이벤트 라우터에 의한 느슨한 결합

## Producer

이벤트를 투입, 송신



## Event Router

이벤트를 필터링해서  
적절한 타겟에 송신



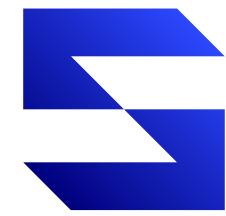
Consumer  
수신한 이벤트를  
적절하게 처리하는 함수



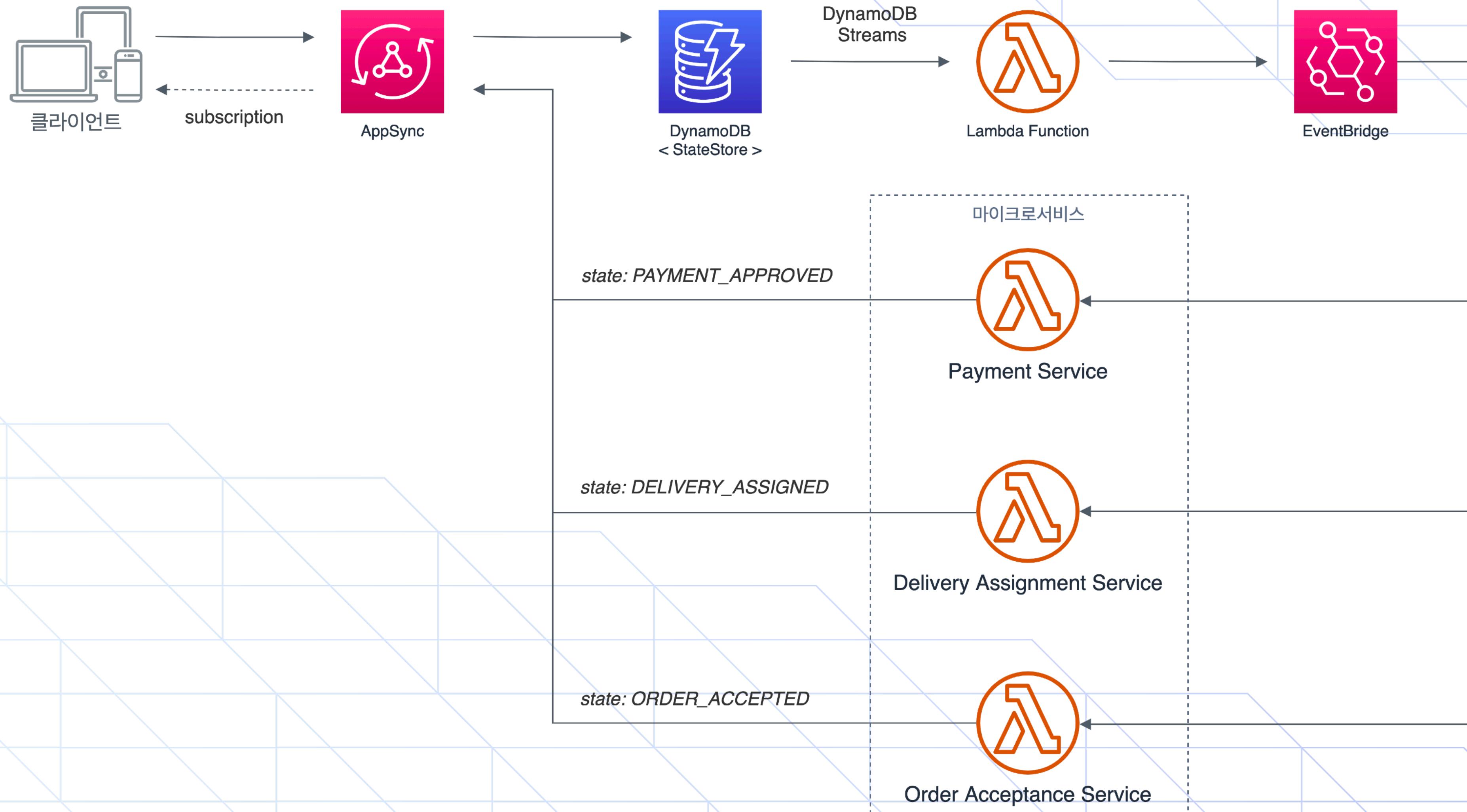
과거 이벤트를 로깅 및 아카이빙하는  
이벤트 스토어의 역할도 가능

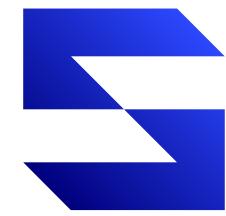


데이터베이스

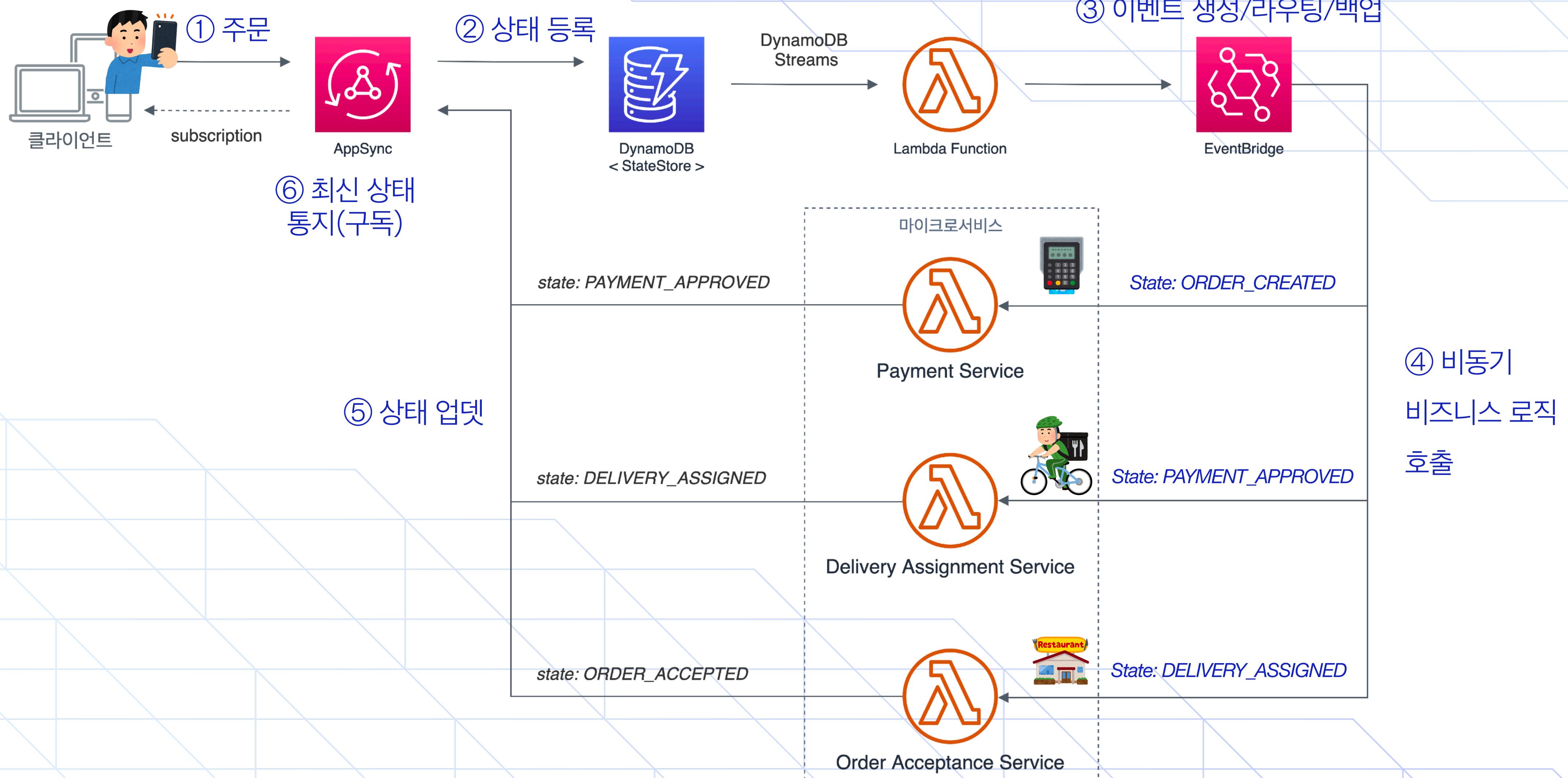


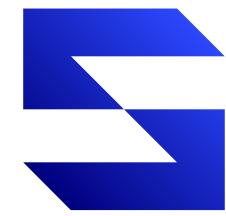
# AWS 서비스로 구축해보기



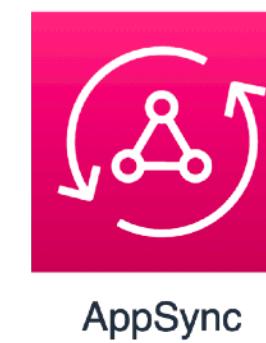
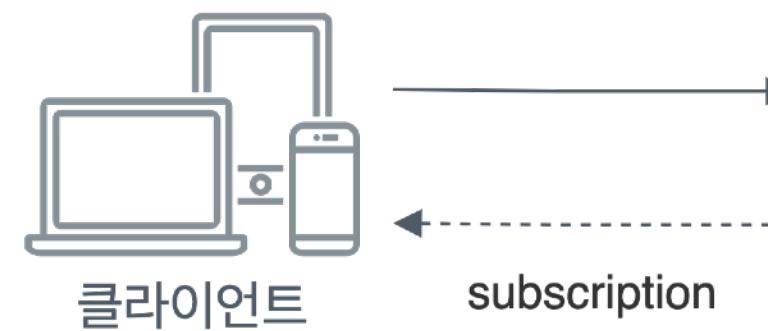


# AWS 서비스로 구축해보기

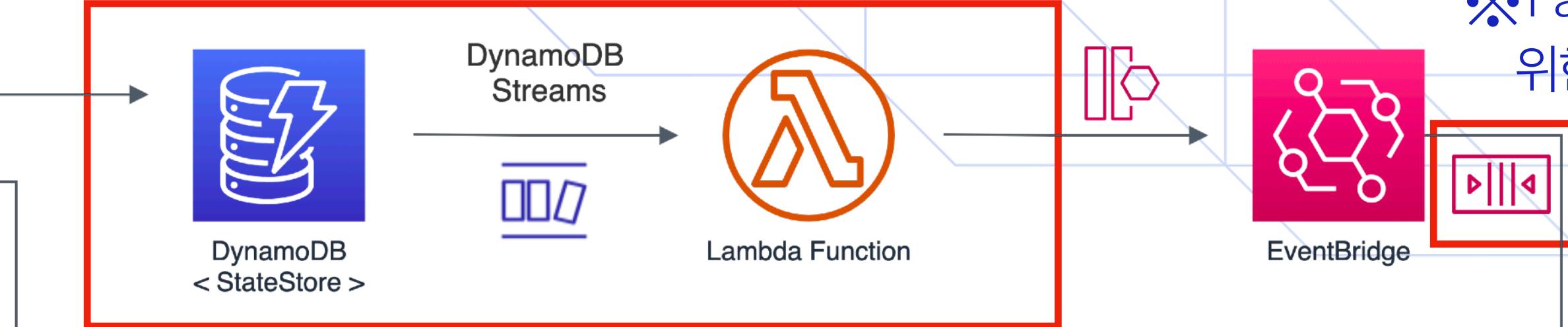




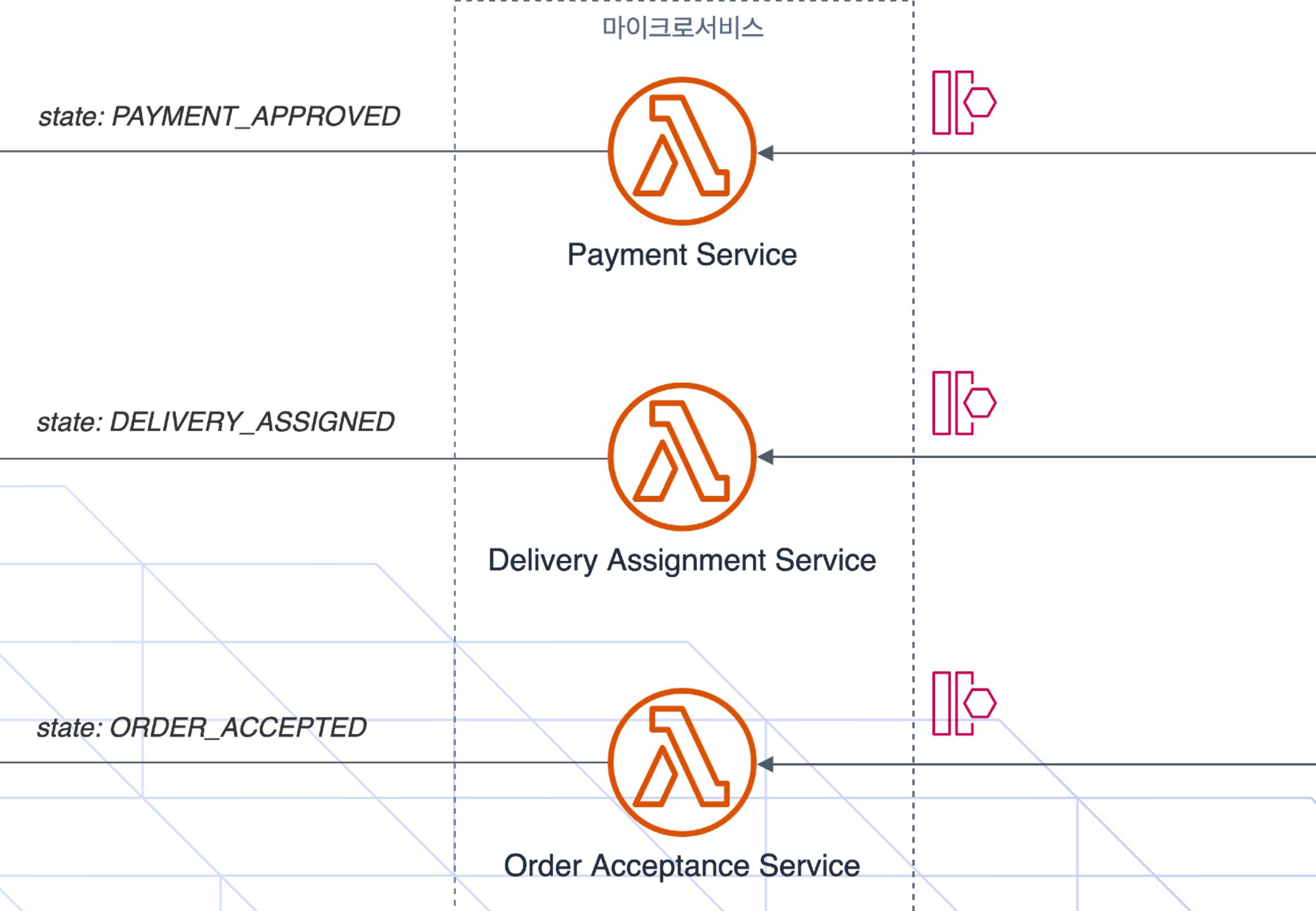
# AWS 서비스로 구축해보기

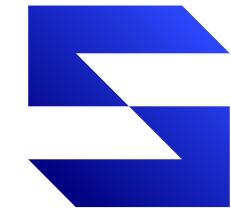


\*EventBridge pipes 혹은 SQS 도 사용가능



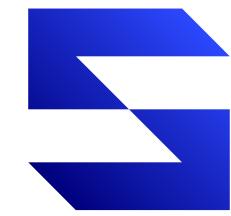
\*Fault Tolerance 를 위한 DLQ 설정





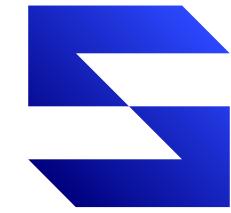
## AWS 서비스 구성의 데모

슬라이드에서 설명드린 푸드 딜리버리 서비스를 가정한  
간단한 데모 및 AWS 설정을 보여드립니다.



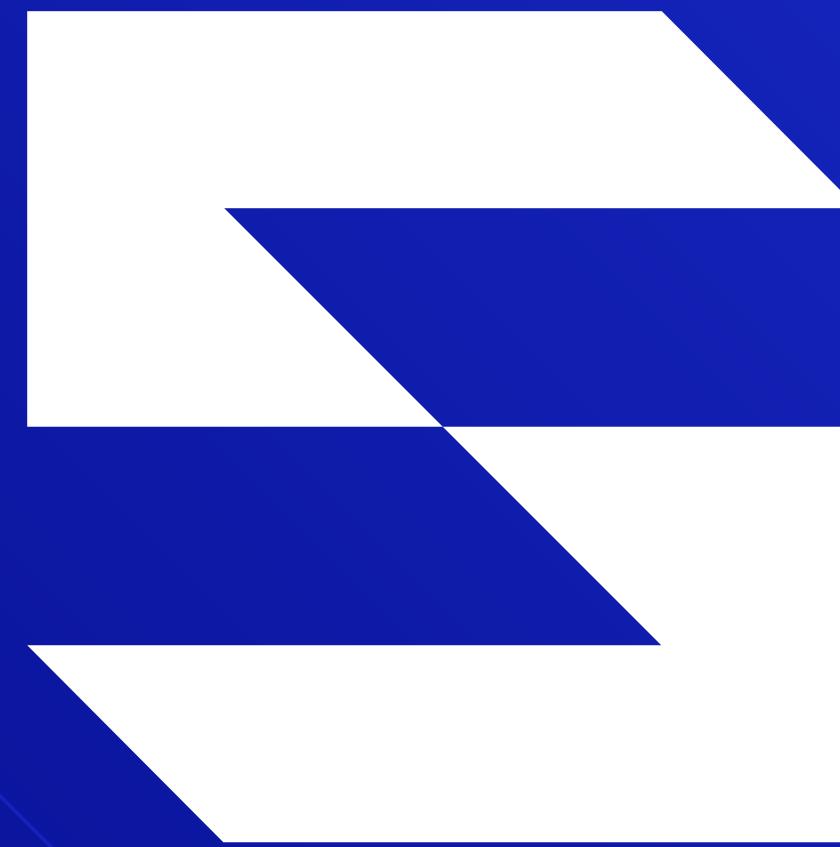
## 3factor app은 완전히 새로운 기술은 아님

- WebSocket, MQTT, REST API로 폴링하기 등, 대체 가능한 기술 및 프랙티스가 이미 많이 존재하고 있음
- 도메인구동, 이벤트구동 등 기존의 설계 방식을 답습하여 응용한 설계 패턴(Design Pattern) 중의 하나
- 결과정합성(Eventually consistent) 및 비동기 마이크로서비스 등, 서비스를 전제로 둔 패턴

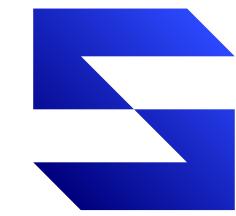


## 3factor app은 모든 면에서 성공적인 기술 요소가 아닐 수 있음

- 리얼타임 요소를 배제한다면 반드시 3factor app 의 이점을 살릴 수 있는지 고민해야 함
- 비동기 설계 및 처리에 익숙하지 않다면 테스트나 에러 처리가 힘들 수 있을 가능성
- 새로운 학습 요소에 대한 비용을 따져볼 것
  - e.g. RESTful에서 벗어난 설계, 비동기/이벤트구동, 서비스/매니지드 서비스에 대한 인식 및 사용, GraphQL 및 프론트엔드에서의 실시간 처리 등

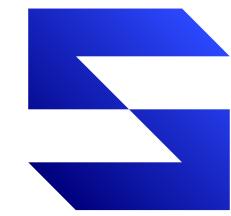


# Serverless Operations



## Appendix

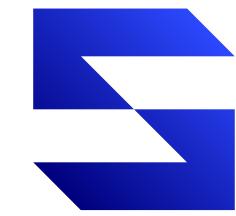
### 부록



## 비동기 테스트의 어려움

Serverless Meetup Virtual 2020 #4 자료참조

<https://riotz.works/slides/2020-serverless-meetup-japan-virtual-4/#22>



## DR 및 멀티리전 구성

AWS Community Day Singapore 2023 자료 참조 (p.25-p.26)

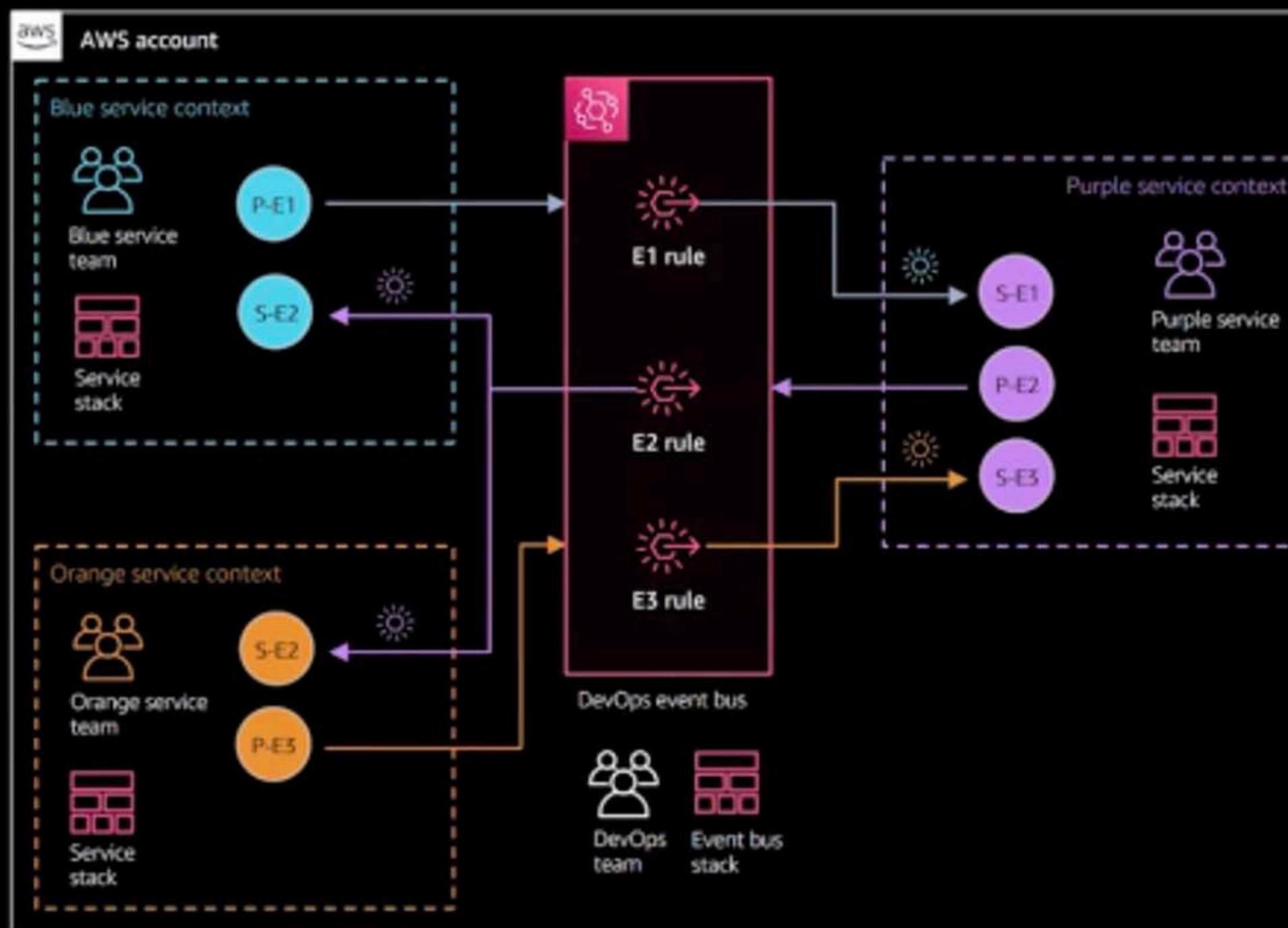


# COMMUNITY DAY

## Architectural approaches with EventBridge - centralized vs distributed

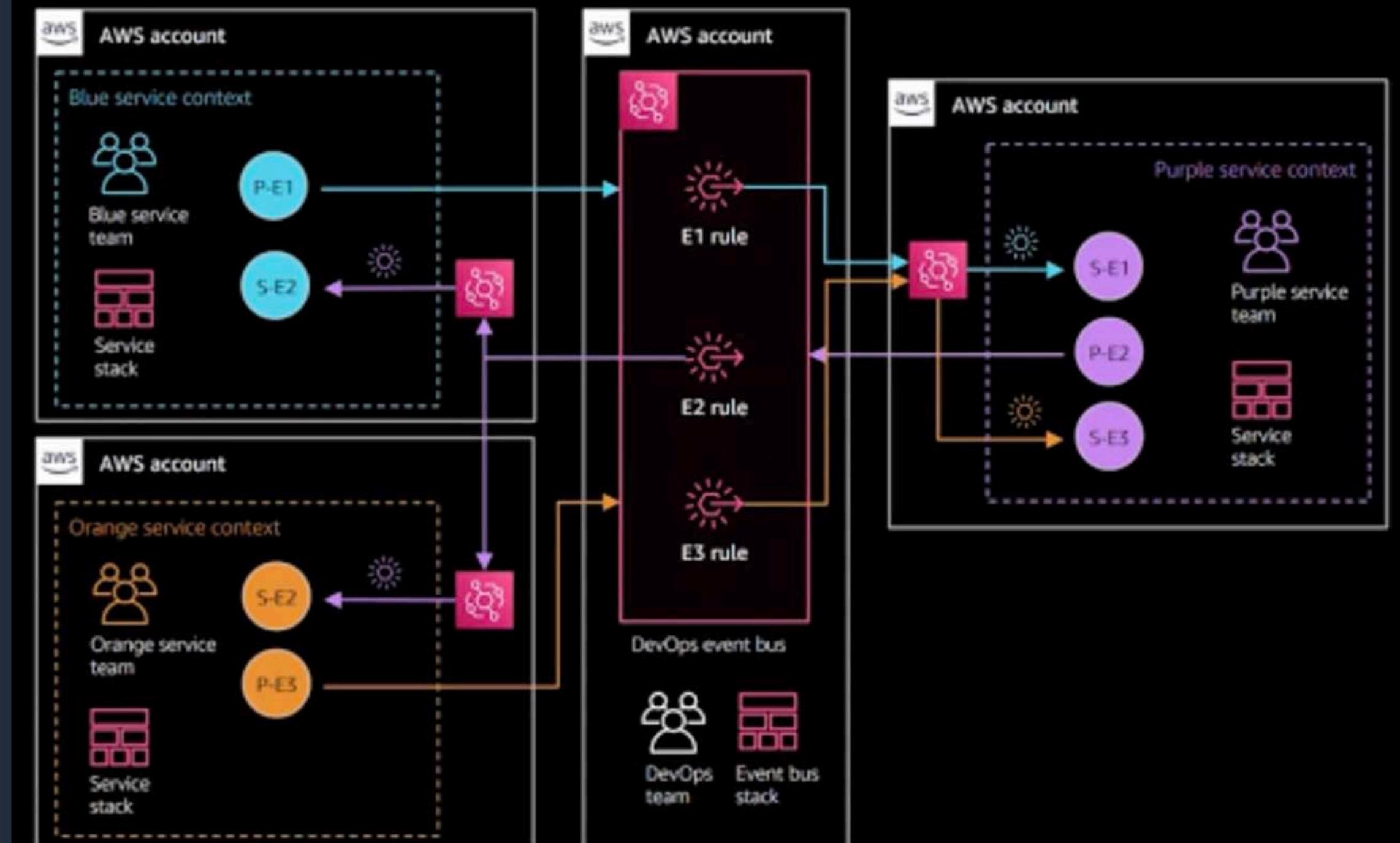
### Single-bus, single-account pattern

CENTRALIZED



### Single-bus\*, multi-account pattern

CENTRALIZED



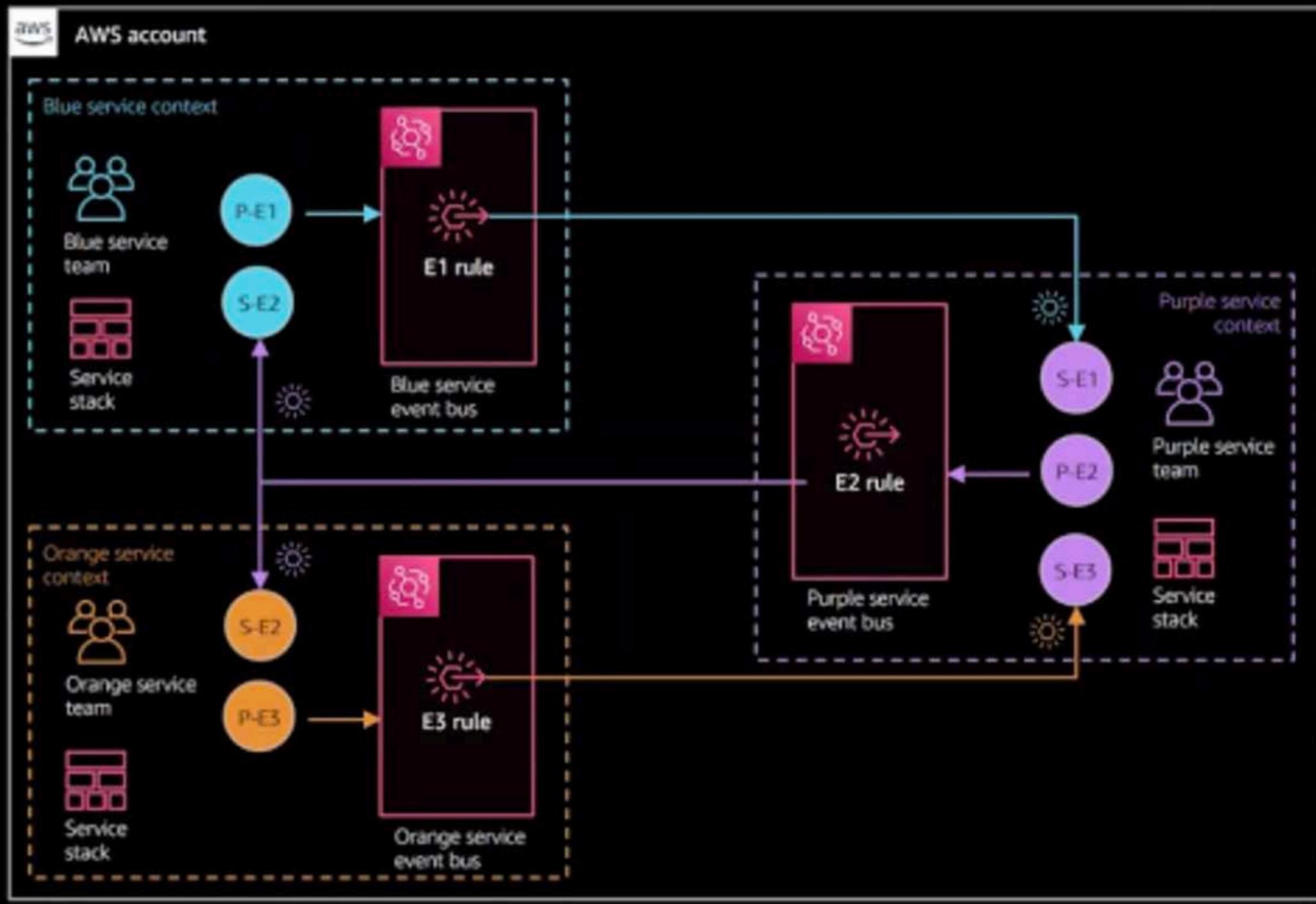


# COMMUNITY DAY

## Architectural approaches with EventBridge - centralized vs distributed

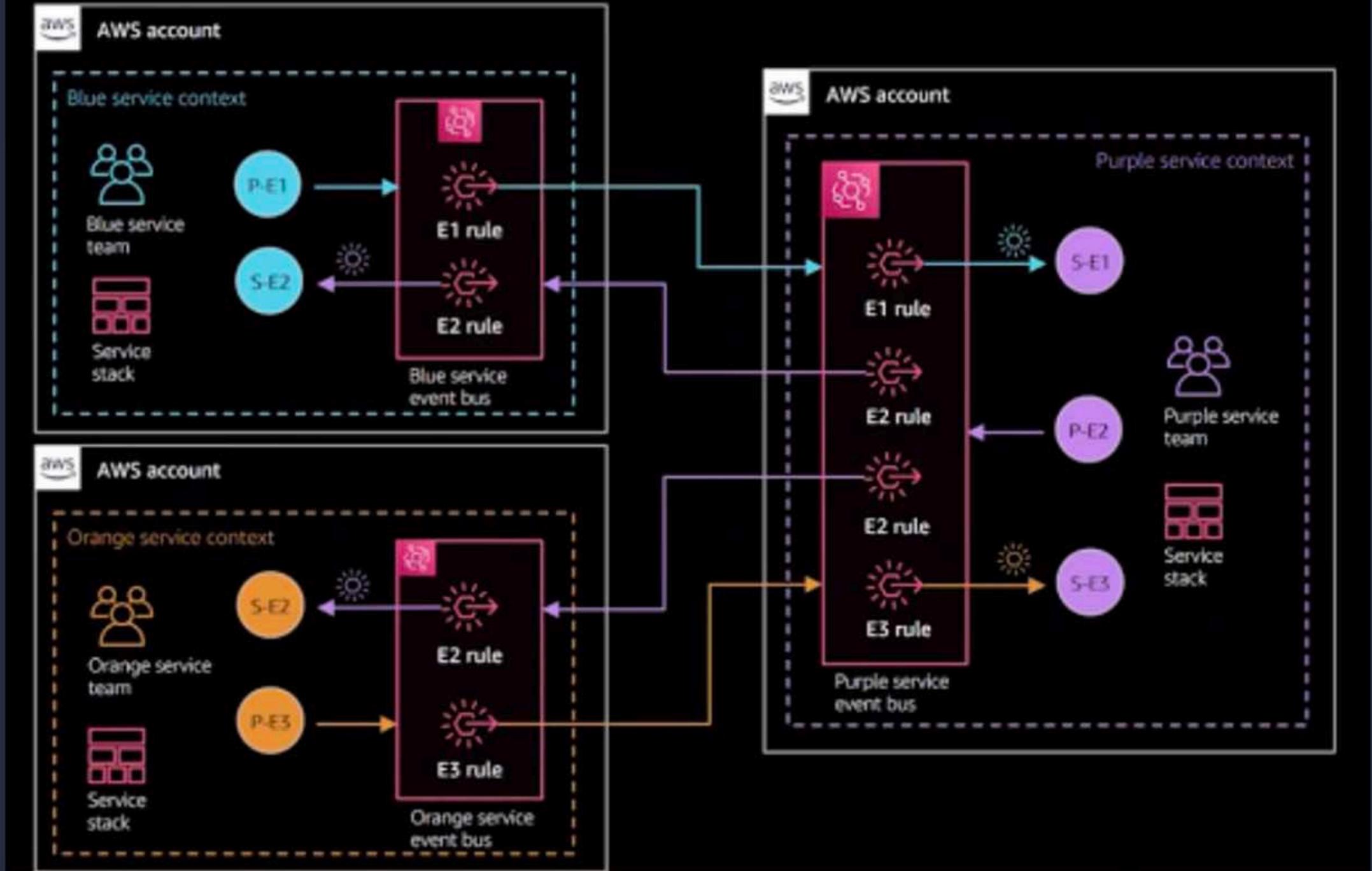
### Multi-bus, single account pattern

DISTRIBUTED



### Multi-bus, multi-account pattern

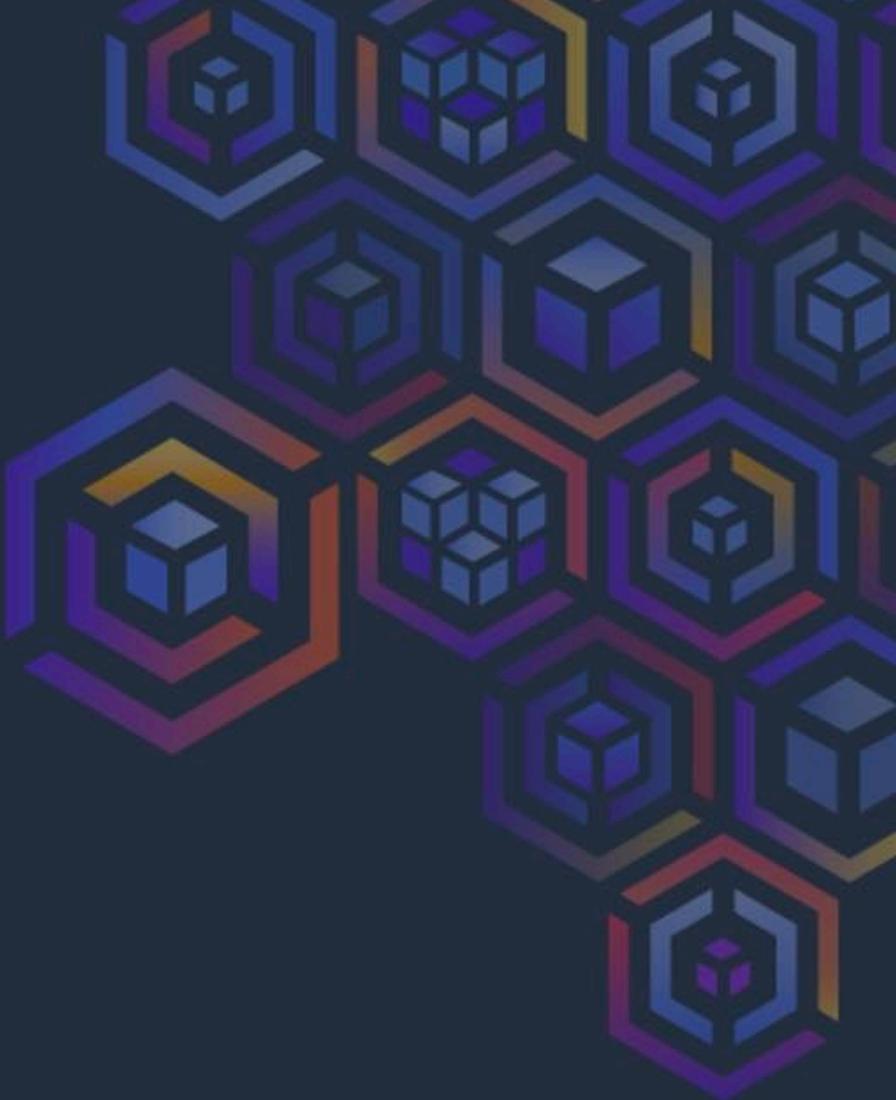
DISTRIBUTED





# COMMUNITY DAY

## Architectural approaches - centralized vs distributed



### Advantages

#### Centralized

- Easily integrate applications with minimal changes

### Disadvantages

- Single point of failure

#### Distributed

- No single point of failure
- Greater support service autonomy

- More difficult to design distributed solutions
- More resources to manage



serverless.co.jp