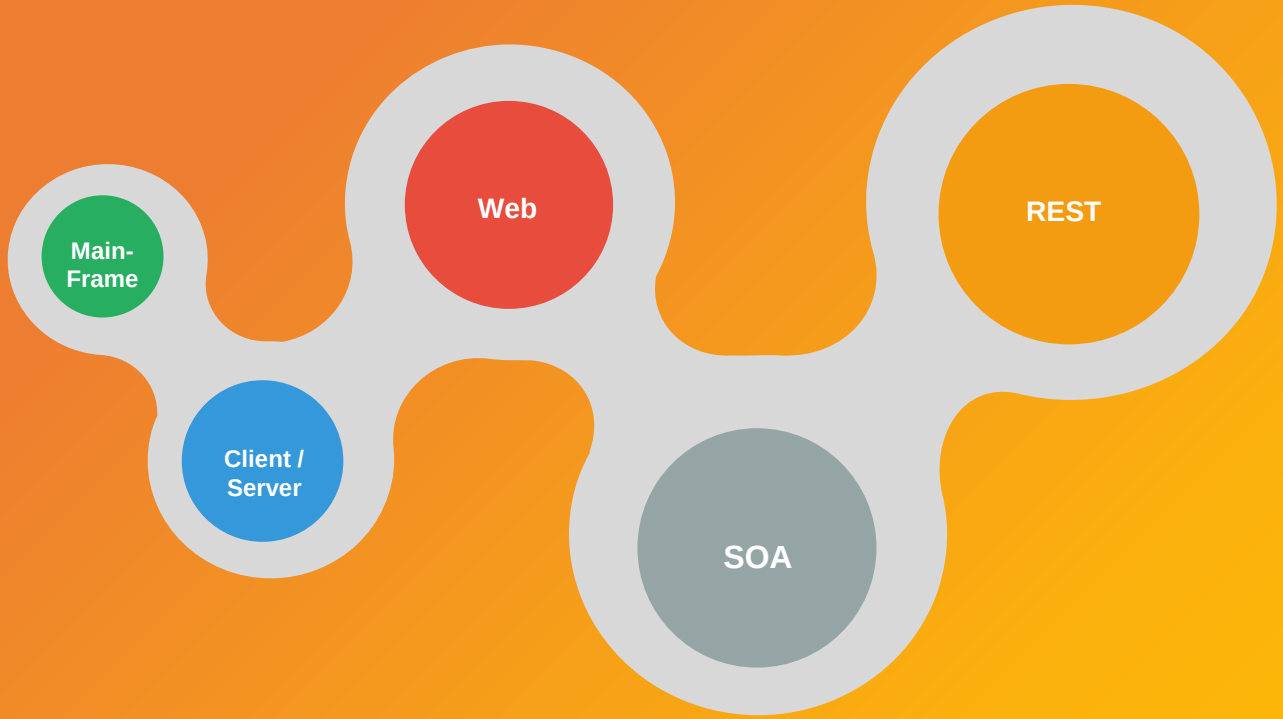
Micro Service Modeling

3rd Jan 2020, ver2.0

uEngine

소프트웨어 아키텍처의 성장 여정



AS-IS: Pain-points

A 사 의료분야 SaaS 운영

- 서비스 **업그래이드가 수시로 요청**이 들어와 거의 **매일 야근**중. 개발자 행복지수가 매우 낮음.
- 테넌트별 다형성 지원을 제대로 하지 못하여 가입 고객이 늘 때마다 전체 관리 비용이 급수로 올라가는 한계에 봉착함
- **자체 IDC를 구성하여 하드웨어, 미들웨어 구성**을 직접해야 하는 비용문제.

B 사 제조분야 SaaS 운영

- **운영팀과 개발팀이 분리되어 개발팀의 반영을 운영팀이 거부하는 사례** 발생
- 개발팀은 새로운 요건을 개발했으나, 이로인해 발생하는 오류가 두려워 **배포를 꺼려함**
- 현재 미국, 일본, 유럽 등 수요가 늘어나는 상황이나, 상기한 문제로 신규 고객의 요구사항을 받아들이지 못하는 상황
- **수동 운영의 문제로, SLA 준수가 되지 못**하여 고객 클레임이 높은편
- 기존 모놀로씩 아키텍처의 한계로 장기적인 발전의 한계에 봉착

Agile Delivery

Amazon, Google, Netflix, Facebook, Twitter는 얼마나 자주 배포할까요?

고객이 요구하는 기술적 요구사항

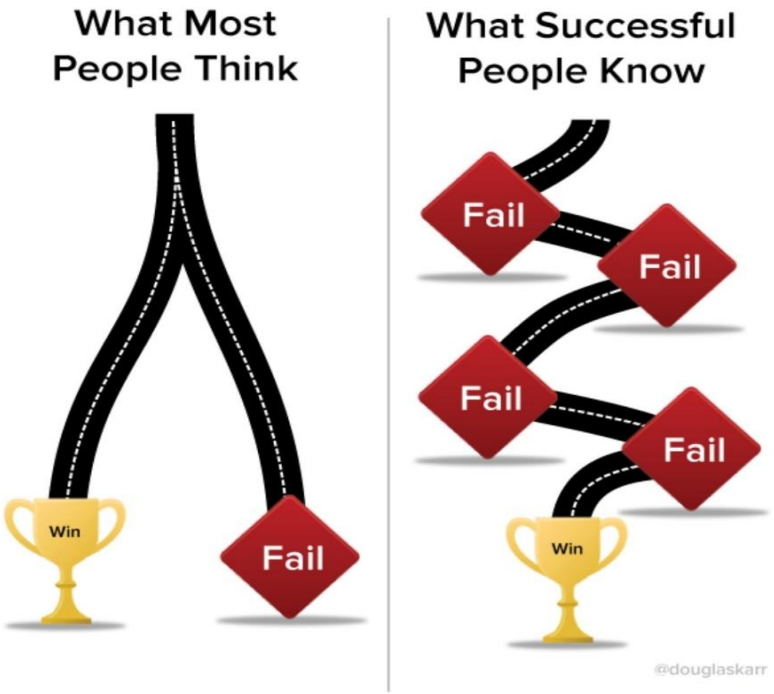
Company	배포 주기 Deploy Frequency	배포 지연 시간 Deploy Lead Time	안정성 Reliability	고객 요구 응답성 Customer Responsiveness
아마존	23,000 / 일	몇 분 (Minutes)	높음	높음
구글	5,500 / 일	몇 분 (Minutes)	높음	높음
넷플릭스	500 / 일	몇 분 (Minutes)	높음	높음
페이스북	1 / 일	몇시간 (Hours)	높음	높음
트위터	3 / 주	몇시간 (Hours)	높음	높음
일반회사	9개월 주기	수개월~수분기별	낮음 / 보통	낮음 / 보통

출처: 도서 The Phoenix Project

Agile 의 정의

Planning is important!

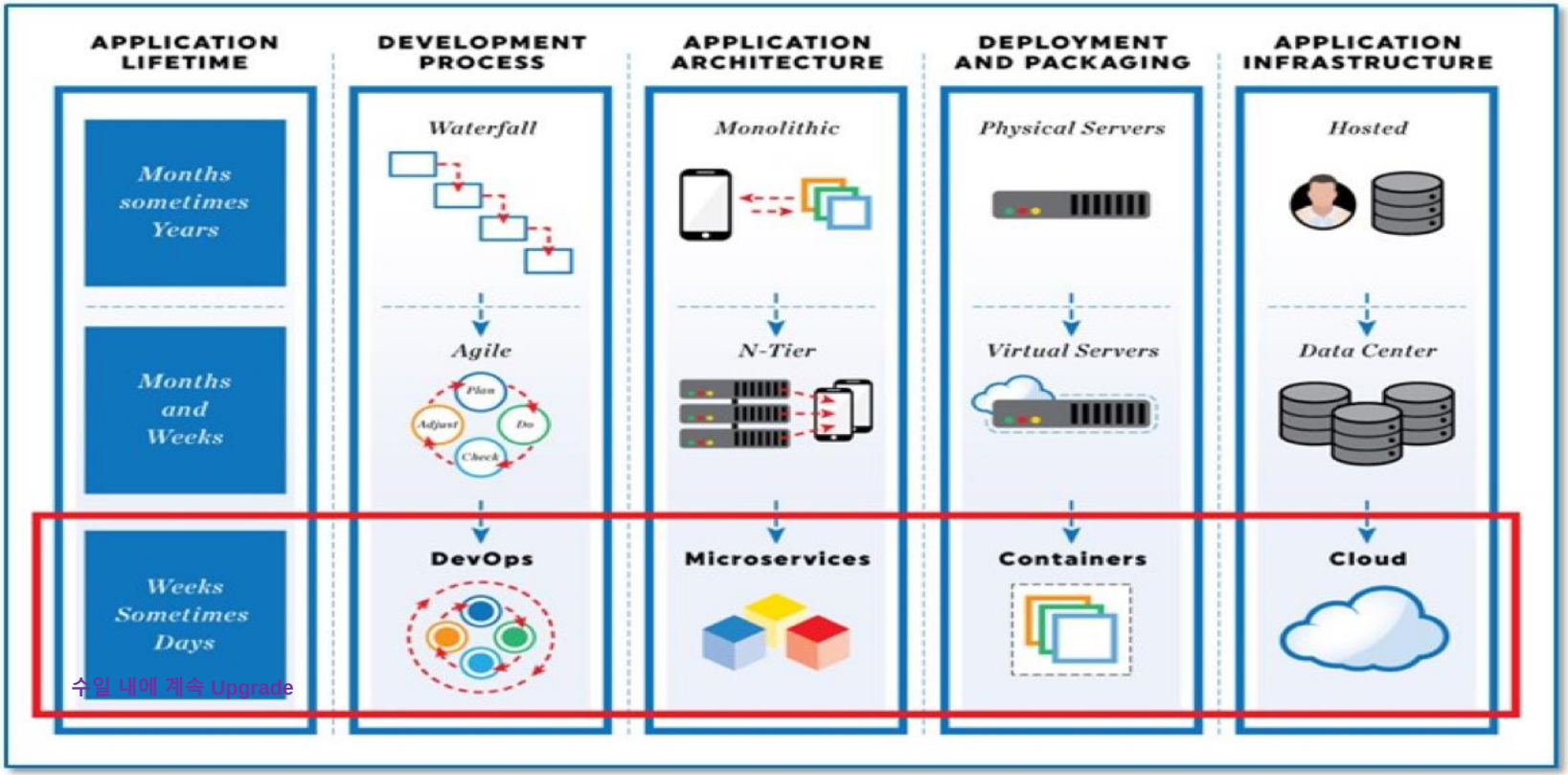
- 계획 중심
- 실패는 나쁜 것
- 비용 중심 사고



Agility is important

- 성장 마인드셋
- Fail Cheap, Fail Fast, Fail Often
- 고객 중심 사고

Agile 에 필요한 것들



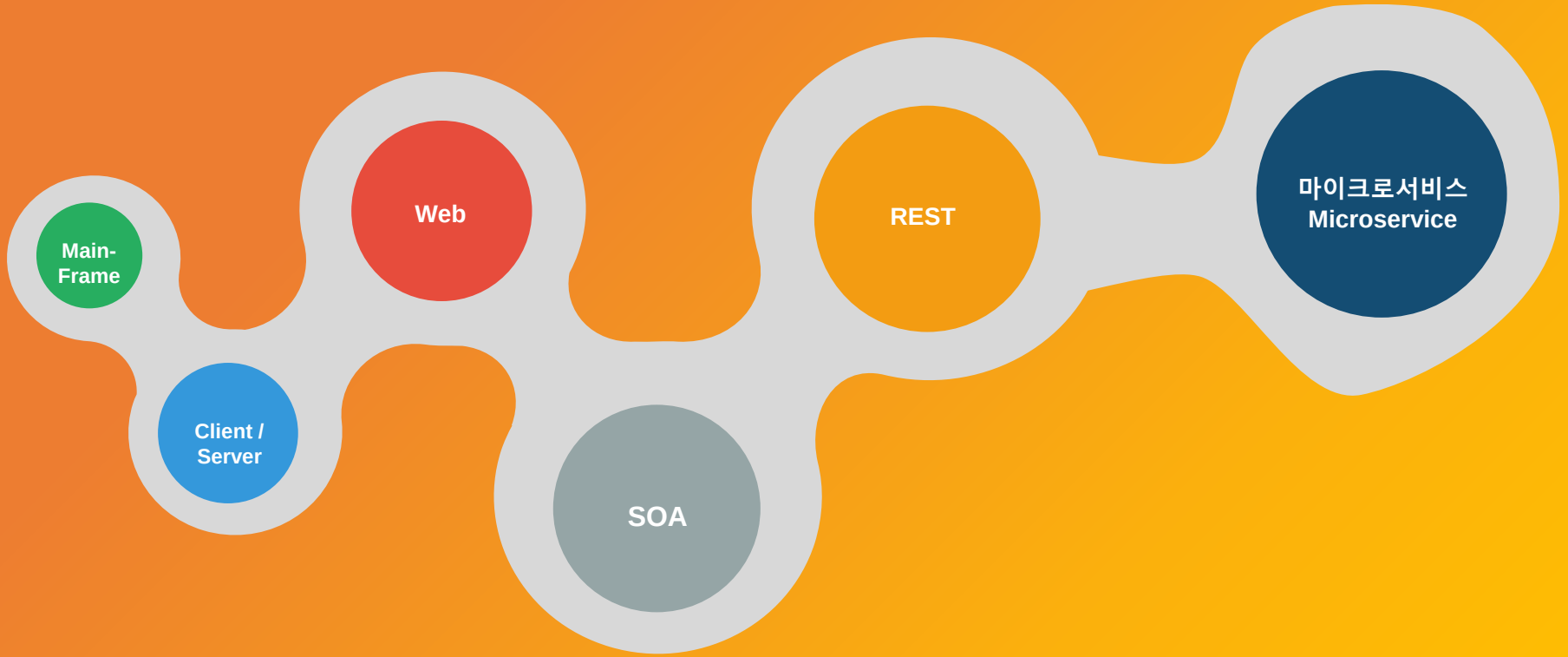
제프베조스의 의무사항

1. All teams will henceforth expose their data andfunctionality through service interfaces.
2. Teams must communicate with each other through these interfaces.
3. **다음과 같은 그 어떠한 직접적 서비스간의 연동은 허용하지 않겠다:**

   no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

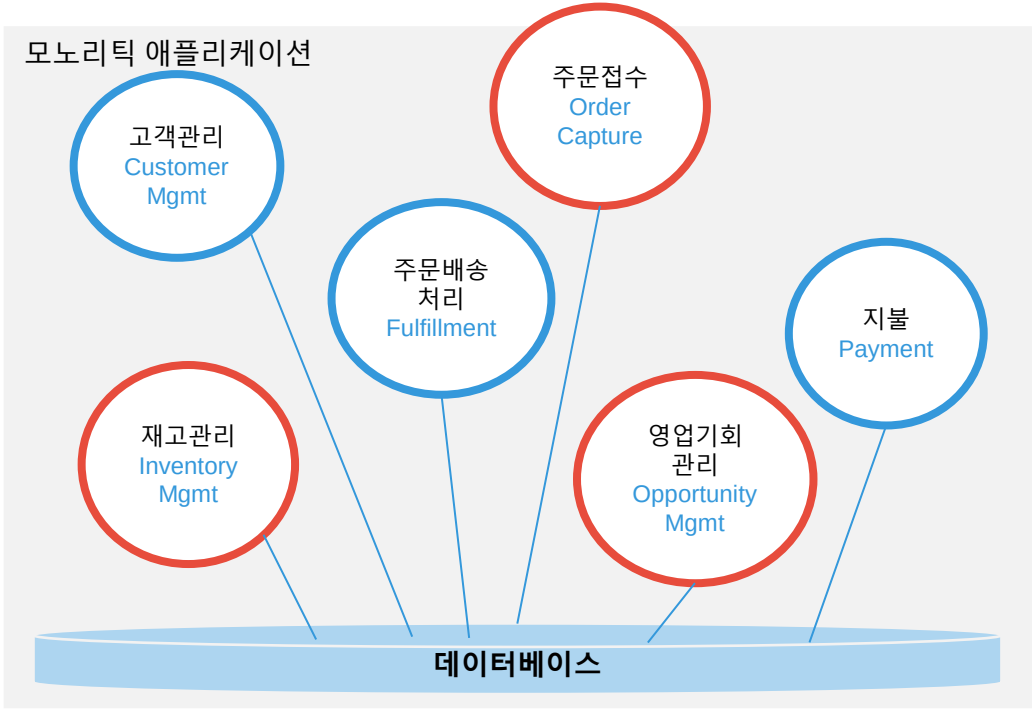
   …
4. The team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
5. Anyone who doesn't do this will be fired.

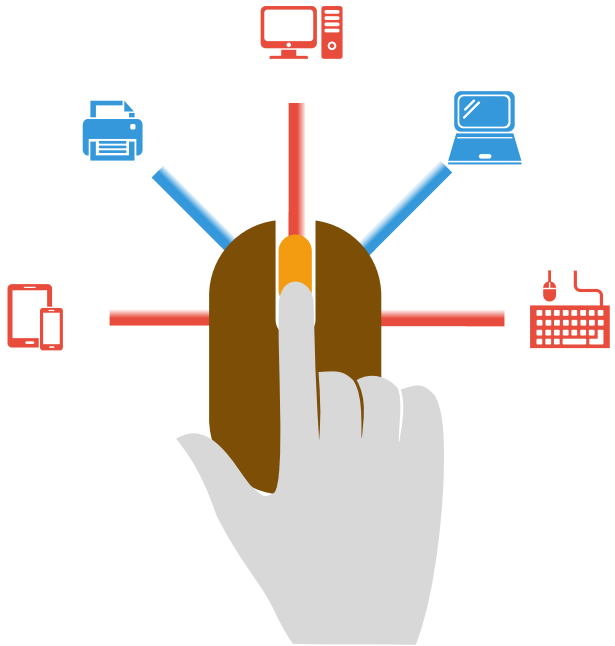
소프트웨어 아키텍처의 성장 여정



첫번째, 마이크로서비스가 아닌 것: 모노리틱 아키텍처 (A Monolithic Architecture)

An Enterprise Application or Suite





모노리틱 아키텍처의 분석

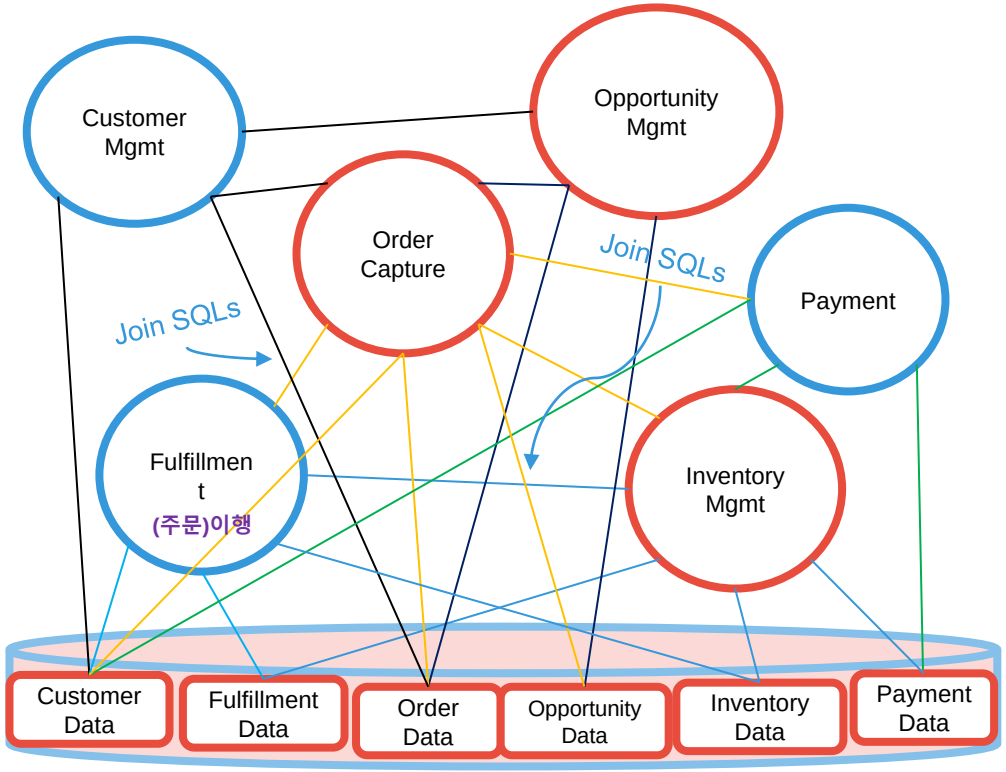
상호 데이터 참조가 용이하여 빨리 개발하기 위한 초기 아키텍처로 적합

그러나, 쉬운 상호연동은 상호의존성을 높힘
But, ease of interaction results in many inter-dependencies

시간이 갈수록, 커플링(의존성)은 강해지고 강해짐
Over time, coupling becomes tighter and tighter

하나의 컴포넌트를 수정하는 일은…
e.g. Order Data Table 의 field 변경

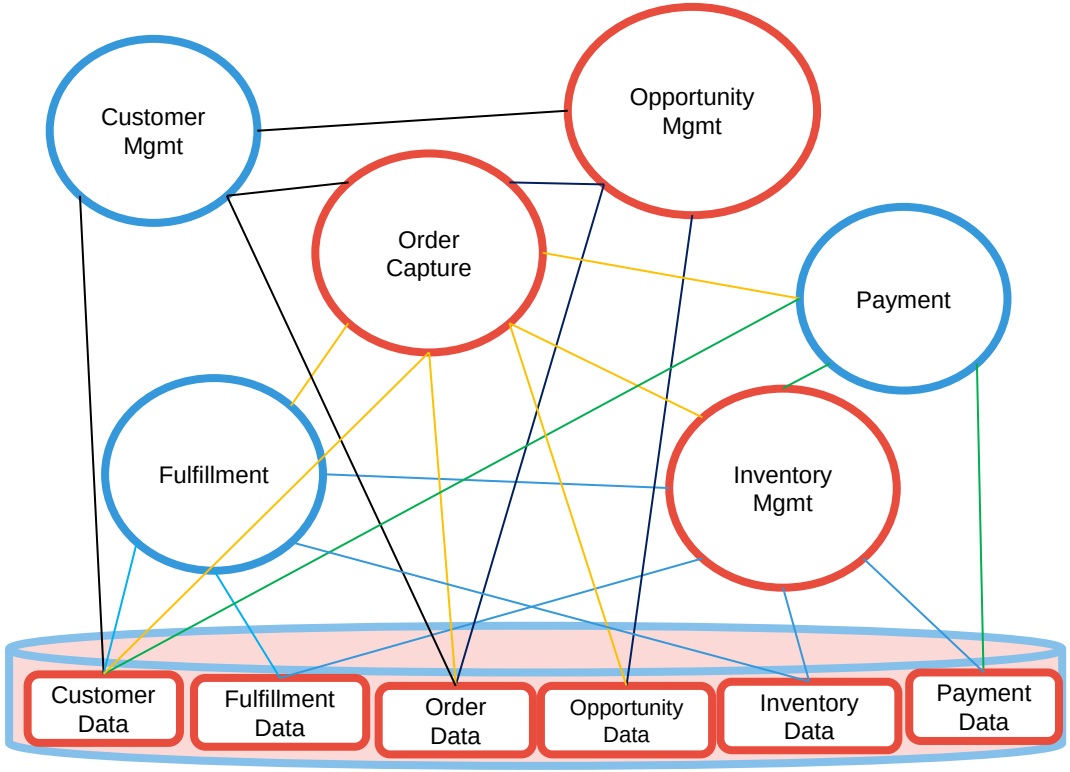
많은 다른 컴포넌트의 수정을 동반하게 됨
e.g. 다른 모듈의 Join SQL 들



모노리틱 아키텍처의 단점

Drawbacks of a Monolithic Architecture

Size matters, costs grow as they grow



코드량이 방대함 (오류발생시 바닷가에서 바늘 찾기)
Large code base

개발환경이 무거움 (IDE, WAS 등)
Overloaded IDE and development environment (Web container, etc.)

하나의 변경이 나머지의 모든 재배포를 유발함
Deployment of any change requires redeploying everything

선별적 확장이 용이하지 않음
Only scales in one dimension

하나의 기술 스택 만을 선택 가능 e.g. Java 1.8 + Oracle
You are committed to the technology stack

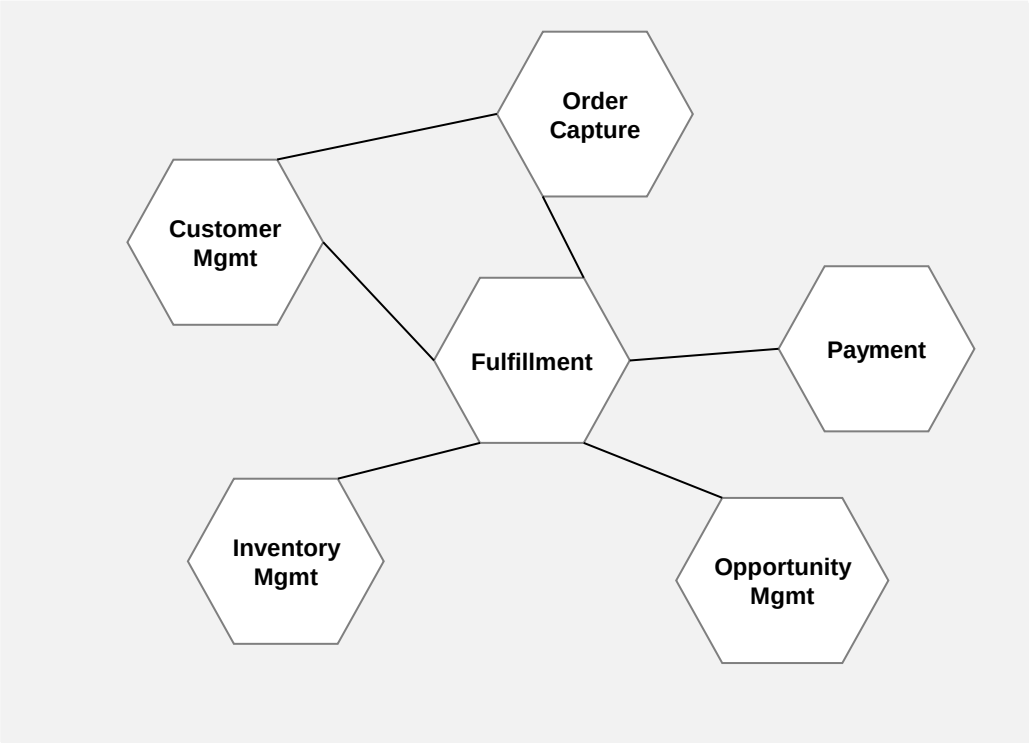
새로운 개발팀을 추가하는데 어려움이 있음
Becomes an obstacle to scaling development

반면: A Microservice Architecture

Service-oriented architecture of loosely coupled elements with bounded contexts

모든 기능을 각각의 배포 스택으로 분리
Break each function into separate deployment stacks

- Separate database
- Separate Servers running any technology
- Local or wide-area network



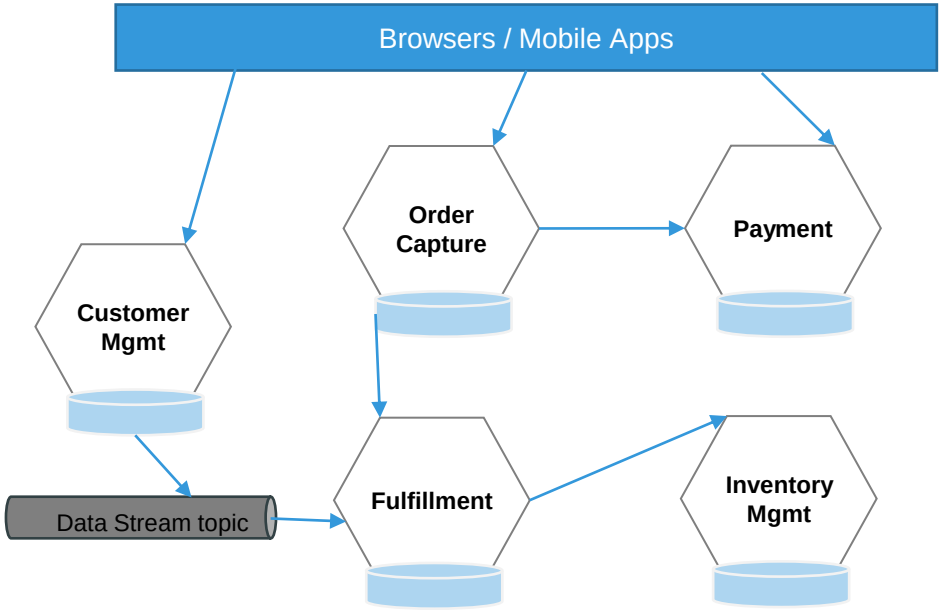
Microservice Architecture

Isolation is the name of the game

Service-oriented architecture

Loosely coupled elements

- Interaction only through HTTP/REST
- Or asynchronous streaming / messaging



Principles of Microservices

Isolation is the name of the game

독립성과 자치성을 코드의 재사용성 보다 높게 본다
Independence and autonomy are more important than code re-usability

마이크로 서비스는 코드와 데이터를 공유하지 않는다
Microservices should not share code or data

불필요한 서비스와 소프트웨어 컴포넌트(라이브러리) 간의 커플링을 피한다
Avoid unnecessary coupling between services and software components

각 마이크로서비스는 각자의 단 하나의 기능에 초점을 맞추어 구현된다
Single responsibility

운영시에는 SPOF (단일 실패 지점) 을 만들어서는 안된다
There should be no single point of failure

HTTP/REST 를 이용한 약결합 연동

(Loosely-Coupled Interaction via HTTP/REST)

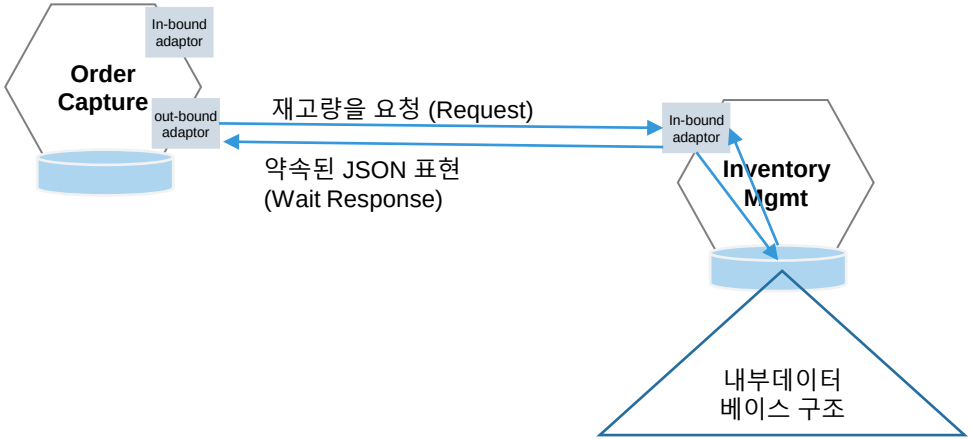
REST APIs must be stable and hide internals

클라이언트-서버 REST API 는 내부구현을 숨길 수 있으면서 연동
Client-oriented REST APIs hide the internal implementation of the service

동기식 연동 (Synchronous, Request-Response)

SOAP 에 비하여 REST 는 API 정의 및 관리 비용이 낮으나, 각 클라이언트에 대한 요청에 충분한 API 를 정의하고 관리하는 비용이 높아질 수 있음

API 는 하위호환성을 유지하기 위하여 추가적인 수정만을 허용
Only additive changes allowed



➔ 빌드타임에서만 약결합을 제공함

비동기 메시징을 통한 약결합 연동

Loosely-Coupled Interaction via Asynchronous Messaging

Data streaming can decouple database with shared data

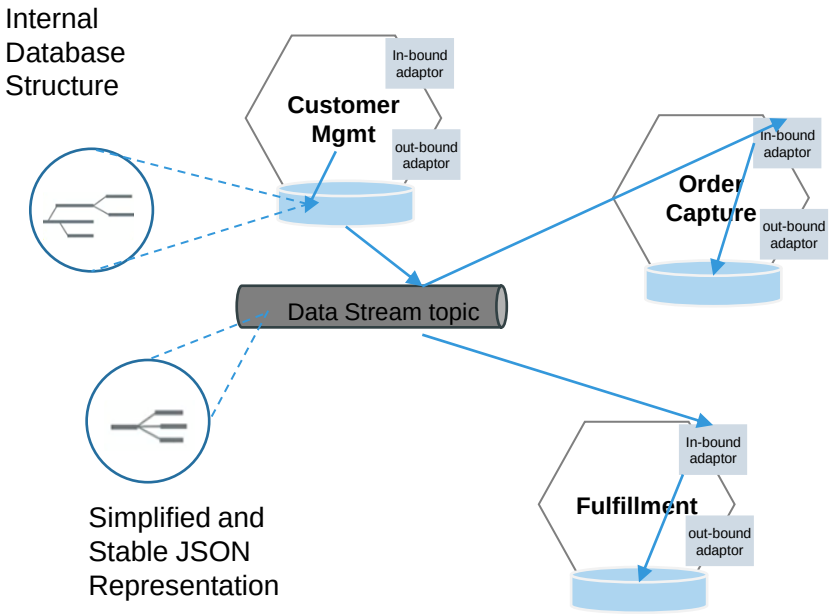
상대의 수신을 기다리지 않는 비동기식 메시징
Asynchronous messaging(streaming)

- Publish-Subscribe
- 준실시간으로 동작함
- **Decouples <u>"timing"</u> = Non-blocking**

Data streams hide the internal implementation of the service

Client ignore parts of the stream they don't understand

➔ 런타임에서도 약결합을 제공함



Microservice Architecture benefits

Smaller is better

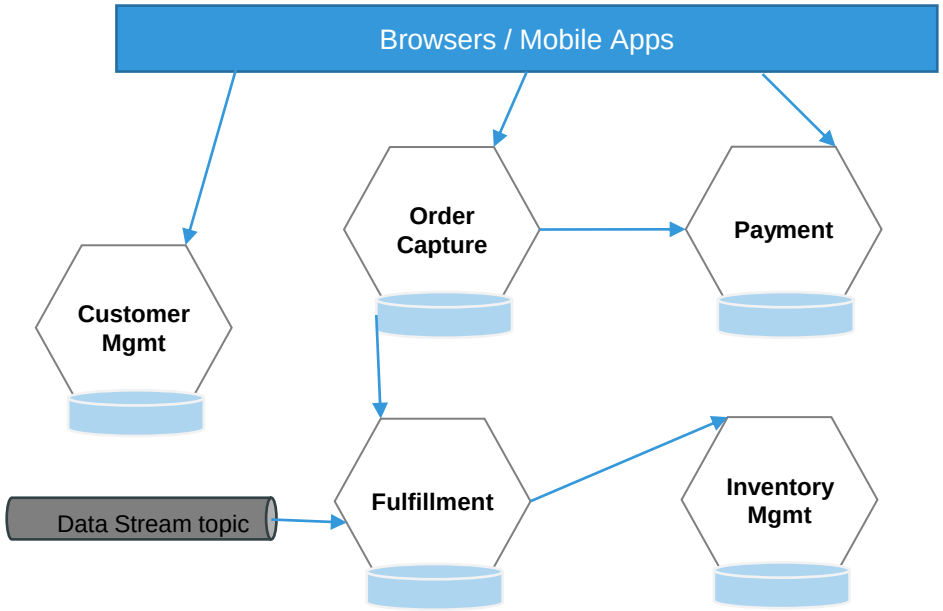
작은 코드량은 이해하기 쉽고, 오류를 찾기 쉽다
(버그가 살기 힘든 공간 = 버그가 숨을 공간이 적다)
Smaller, independent code base is easier to understand

수정된 서비스에 대한 국지적 단위 디플로이와 테스팅이 용이해진다
Simpler deployment and testing of just the changed service

장애격리가 좋아진다
Improved fault isolation

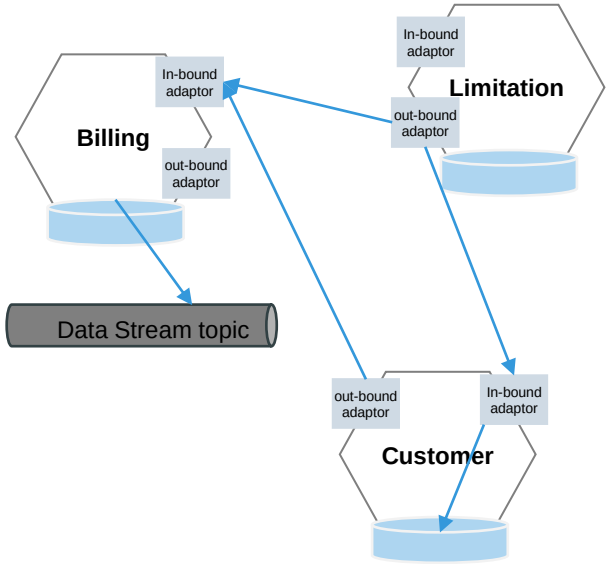
여러가지 혼재된 기술 스택 (신기술)을 사용하기 쉽다
Not committed to one technology stack

개발환경을 구축하기 쉽고 빨리 기동된다
IDE and app startup are faster



Microservice Architecture

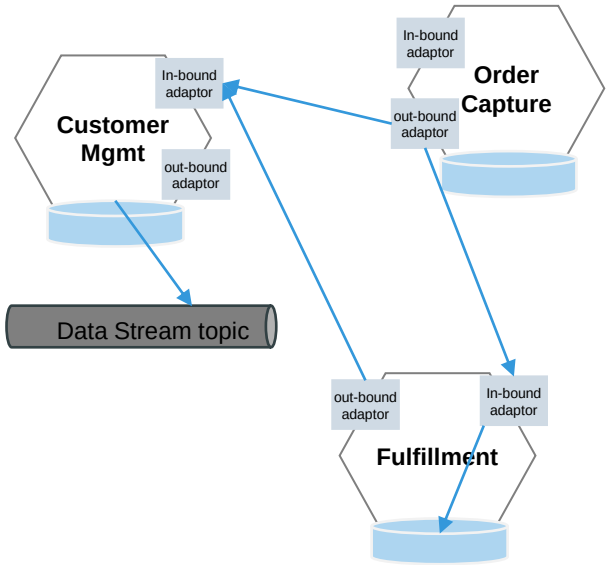
- Updating one service doesn't require changing others
- Ability to upgrade the tech stack (HW/SW/DBMS/NW) Of each service independently



- Good fault isolation
- Smaller, simpler code base
- Options for scaling

Drawbacks to a Microservice Architecture

Distributed computing adds complexity and slow down initial development



개발도구들이 아직 최적화되지 않았다
Dev tools not optimized for distributed services

전체적인 테스트가 복잡해진다
Testing can be more complicated

운영과 디플로이가 복잡해져 쿠버네티스 와 같은 DevOps 환경이 필수적이다
Deployment and operations are more complex

서비스를 어떻게 쪼갤 것인가?
Where/How to decompose the services?

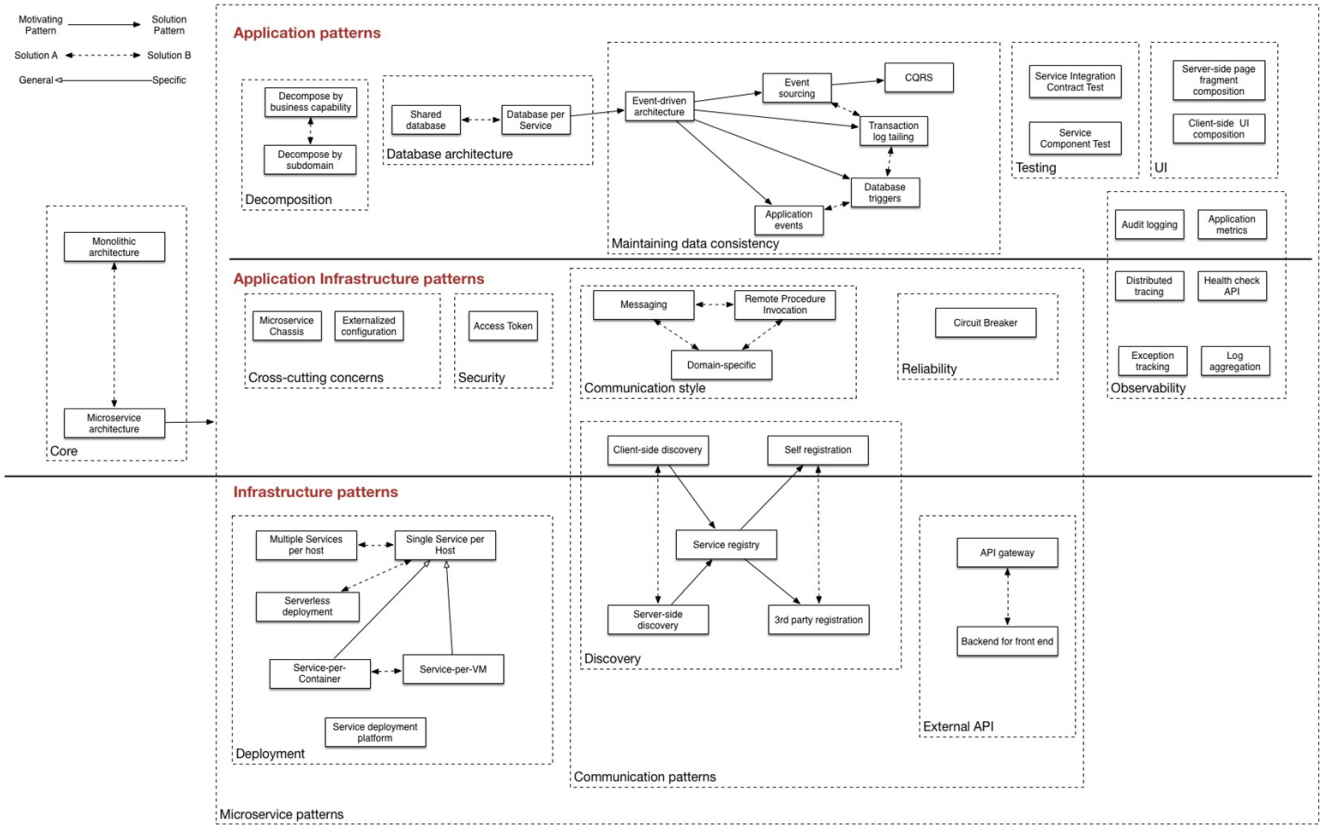
서비스간 연동을 통해서만 구현하는 비용의 상승
Inter-service communication complicates development

분산트랜잭션 (데이터 일관성 등)을 어떻게 보장할 것인가?
Maintaining consistency with distributed transactions is hard

서비스 개수가 많아진 상황의 보안처리를 어떻게 할 것인가?
What about inter-service security? Identity management?

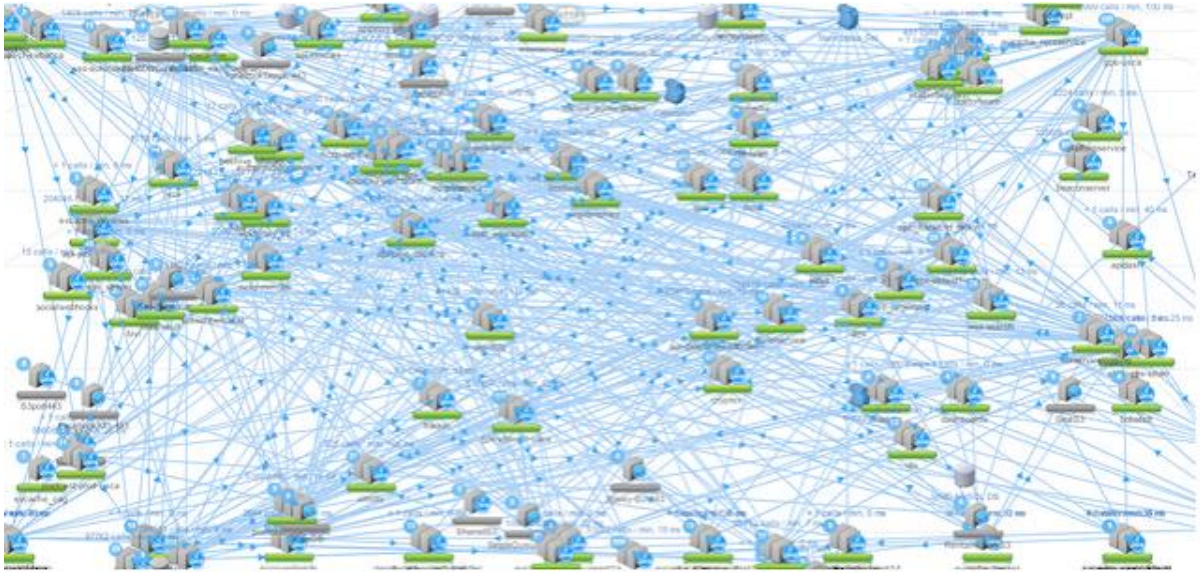
해결책: MSA 디자인 패턴

- microservices.io



넷플릭스 OSS

넷플릭스는 수백개의 마이크로서비스를 운영하고 있는 것으로 유명하다. 각 마이크로서비스간에는 REST 방식의 호출을 통하여 연동되며 이들간의 자동화된 식별과 동적 연동을 위하여 자체적인 플랫폼을 구축했고 이를 Netflix OSS 라는 이름으로 오픈소스화 하였다.



마이크로 서비스 전환 사례

Video & Broadcasting

Mobile Apps and
Serverless Microservices

Pure Play Video OTT- A

From Monolithic to
Microservices

Gaming Platform

IoT Service

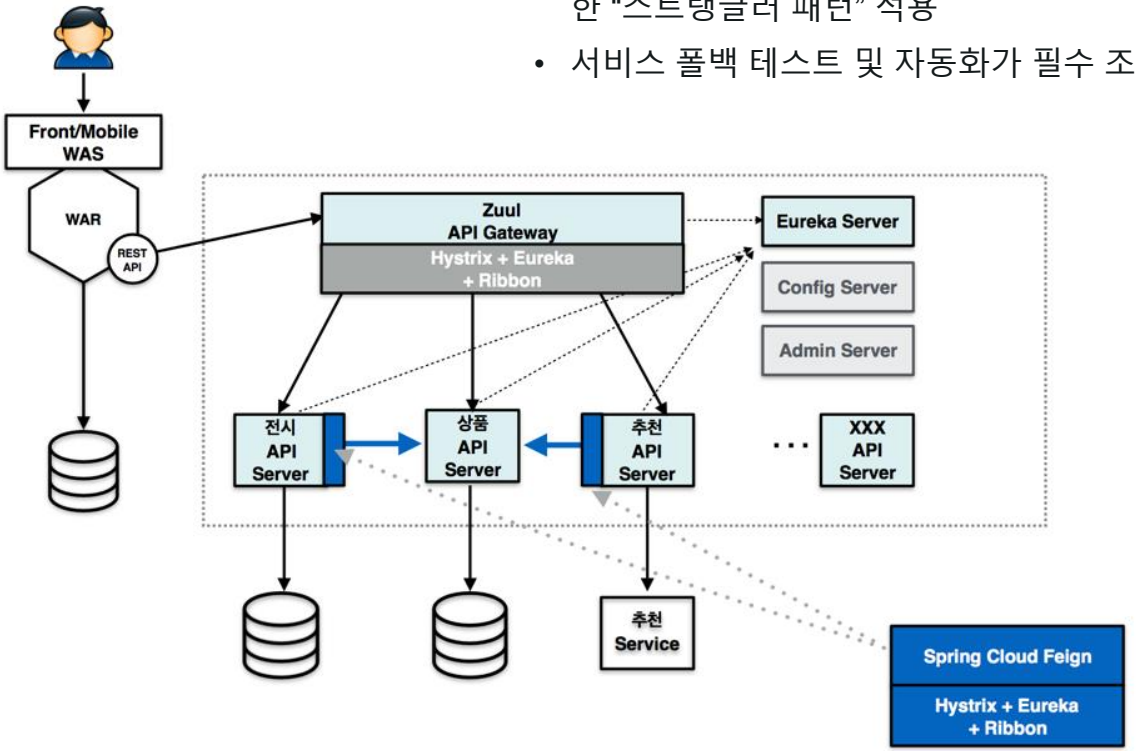
Serverless Microservices

당근마켓의 코어도메인은 채팅!
사용자 입장에서 Critical한 부분은 채팅.

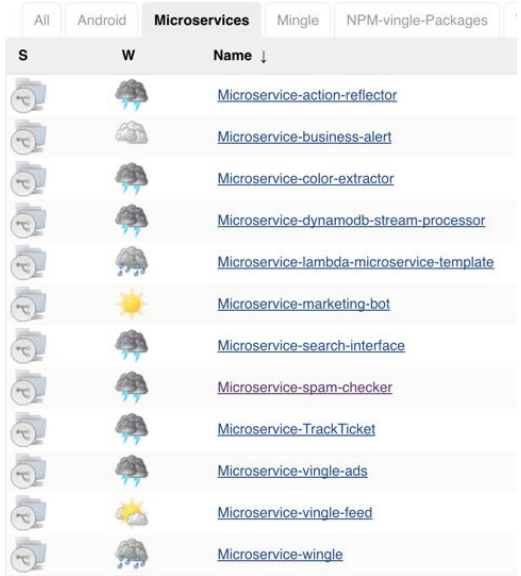
당근마켓

https://youtu.be/mLIthm96u2Q?t=50

- 가장 작고 독립적인 서비스로 부터 마이그래이션
- 순차적 레거시의 마이크로서비스 전환을 위한 API GW 적용한 “스트랭글러 패턴” 적용
- 서비스 폴백 테스트 및 자동화가 필수 조건



- 새로 추가/변경 필요한 기능부터 분리
  - 안정된 레거시는 가능한 손대지 않음
  - 적은 리스크 업무 영역부터
  - 기술보다 노하우와 팀의 문화 정립 우선시
- Business Domain 단위 서비스구성
  - 팀단위 분리 X, 기술적 단위 분리 X
  - 비즈니스 서브 도메인 단위로의 구성
    - Separation of Concerns
  - 응집도
- 서비스 존재 목적은 재사용되어지는 것
  - 표준화된 API I/F
  - 자동화된 문서화



Tip: 10 Attributes of Cloud Native Applications

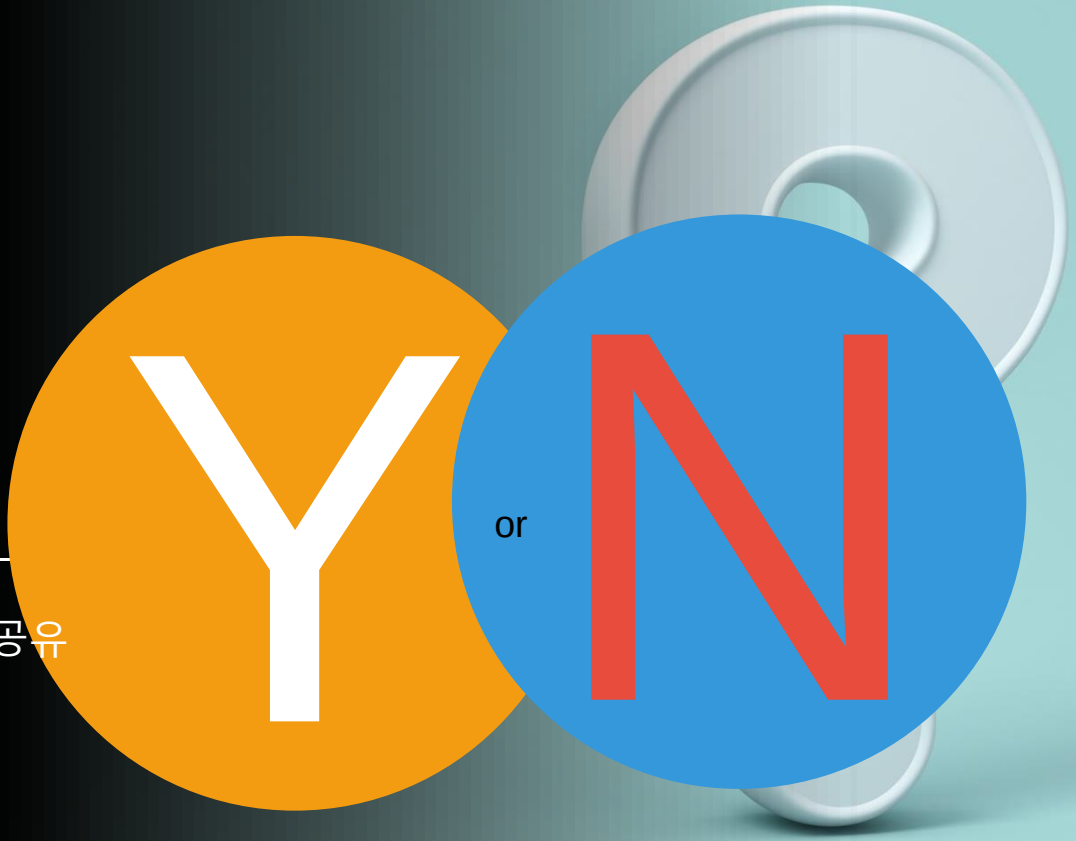
1. Packaged as lightweight containers
2. Developed with best-of-breed languages and frameworks
3. Designed as loosely coupled microservices
4. Centered around APIs for interaction and collaboration
5. Architected with a clean separation of stateless and stateful services
6. Isolated from server and operating system dependencies
7. Deployed on self-service, elastic, cloud infrastructure
8. Managed through agile DevOps processes
9. Automated capabilities
10. Defined, policy-driven resource allocation

https://thenewstack.io/10-key-attributes-of-cloud-native-applications/

26

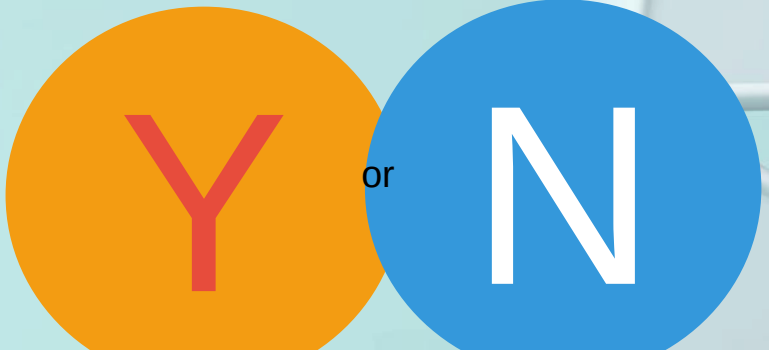
Quiz

마이크로서비스간에 코드를 공유하는 것은 좋은 사례이다



Quiz

마이크로서비스가 실패할 때는 독립적으로 실패해야 한다



Quiz

• 왜 단일 Database 접근은 마이크로서비스에서 안티-패턴 (하지 말아야 할 접근) 인가?

1. 마이크로서비스는 관계형 데이터베이스 (R-DBMS)와 호환되지 않기 때문이다
2. **하나의 데이터베이스만 사용했을 때는 이것이 실패단일지점 (Single Point of Failure)가 될 수 있기 때문이다**
3. 단일 데이터베이스 시스템은 보통 큰 시스템을 만들 때만 쓰이기 때문이다.

Table of content

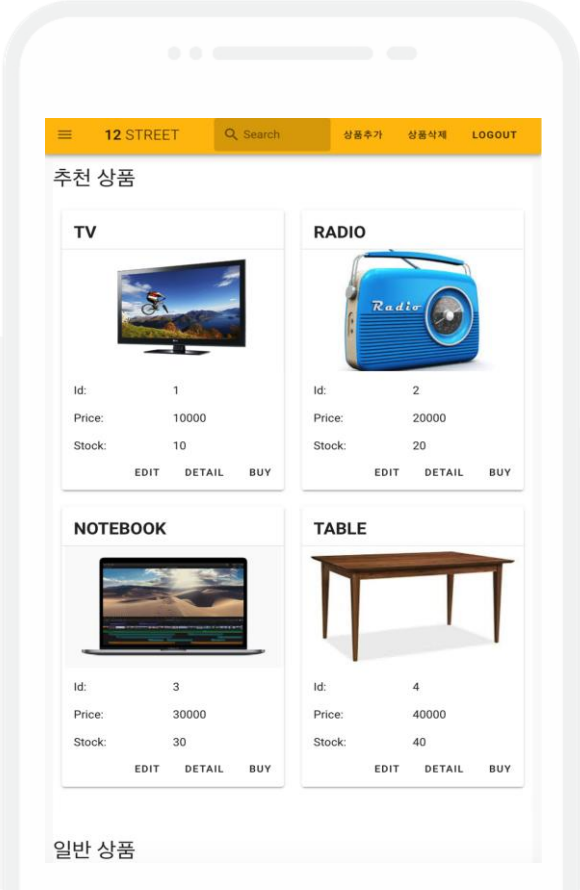
Microservice and
Event-storming-Based
DevOps Project

UEngine

무정지 배포 : 신규 기능 배포할 때 배포 전략을 잘 세워야 함.

Target Domain

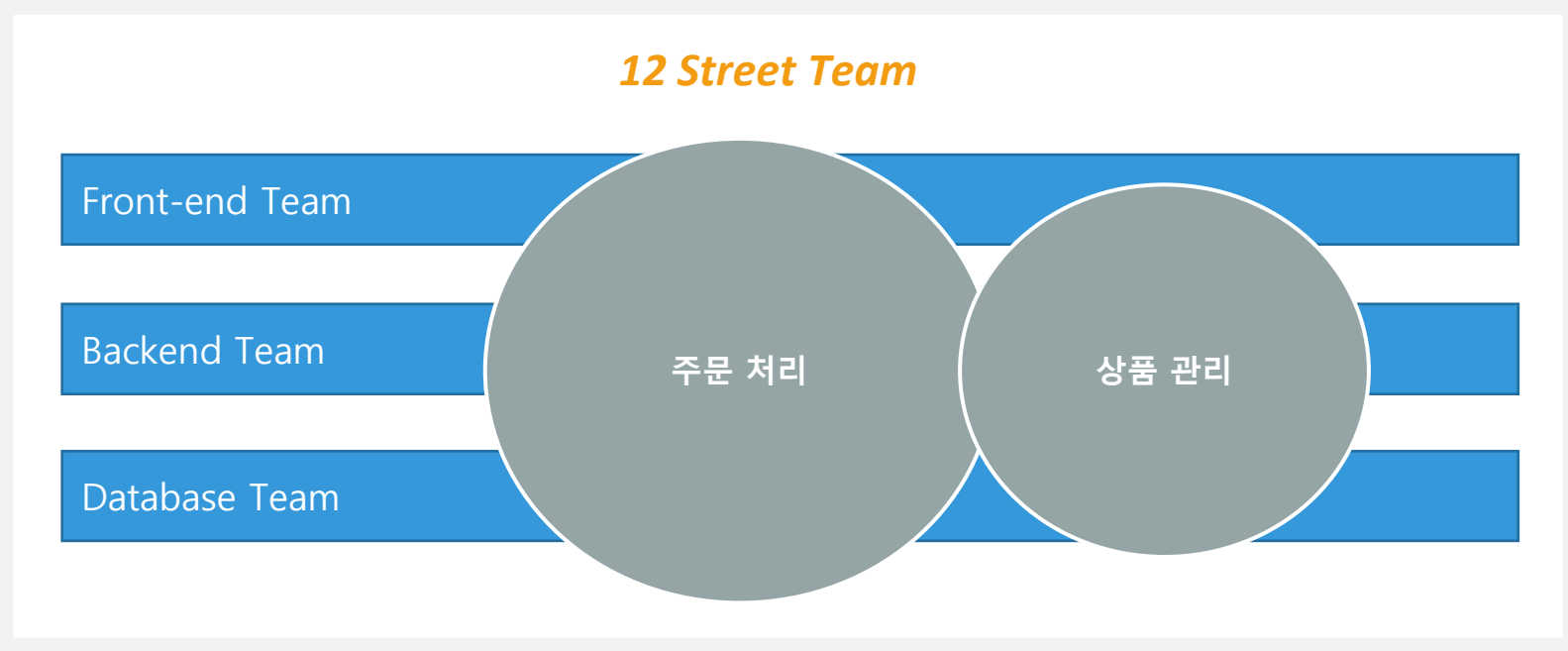
: Online Shopping Mall (12 STREET)



- Vision & Mission

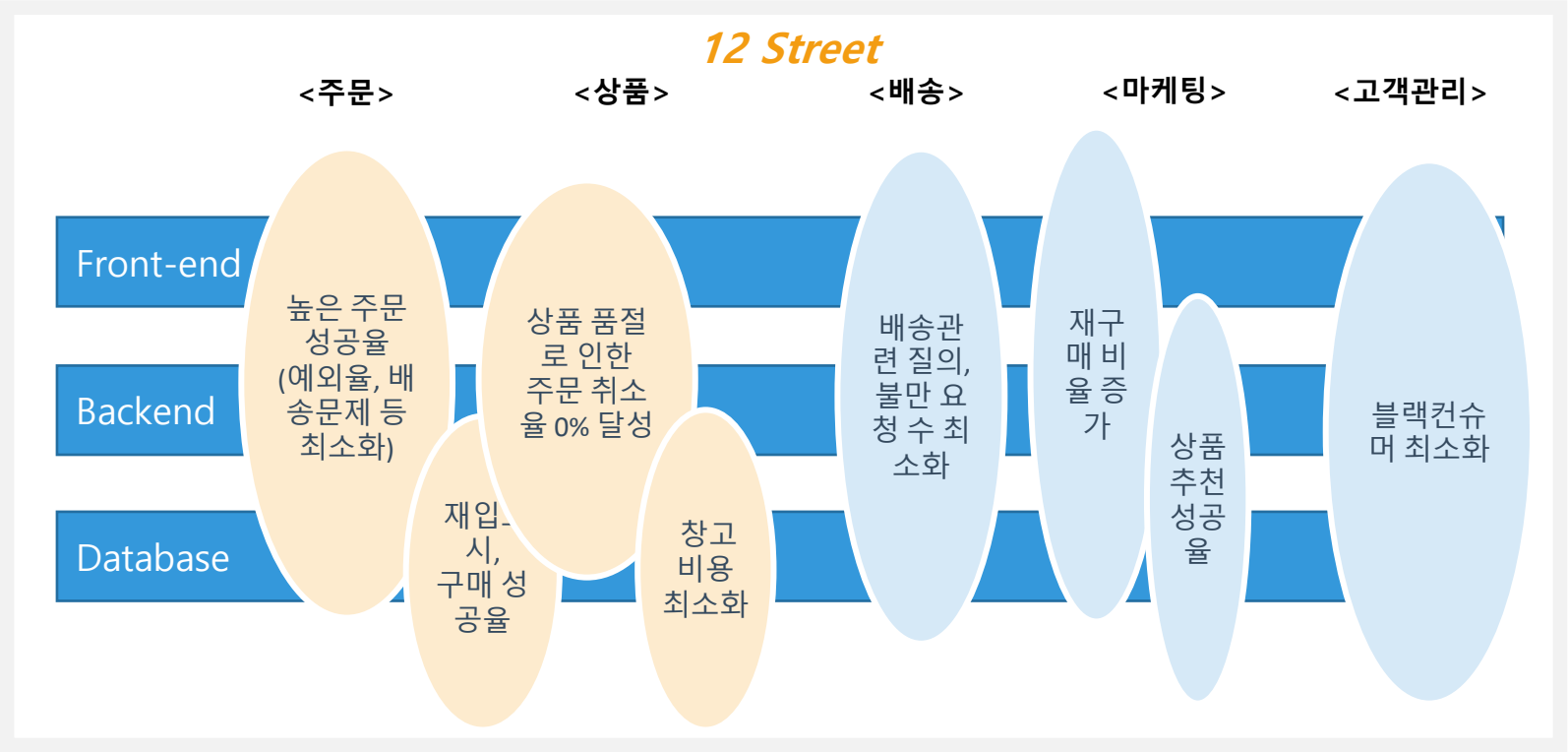
- **Service Resiliency**
  24시간 365일 접속과 주문이 가능
  : **자동화된 회복**, 장애전파최소
- **Customer Responsiveness**
  다양한 고객 기능 요구사항의 탐색과 반영
  : 무정지 재배포, A/B 테스팅
- **Scalability**
  조직, 기능 및 데이터의 확장에 열려 있는 아키텍처
  : EDA, Feature-driven-development

Organization & KPI Definition - 창업시기



- Multiple Concerns Single Organization → no problem
- Horizontally aligned

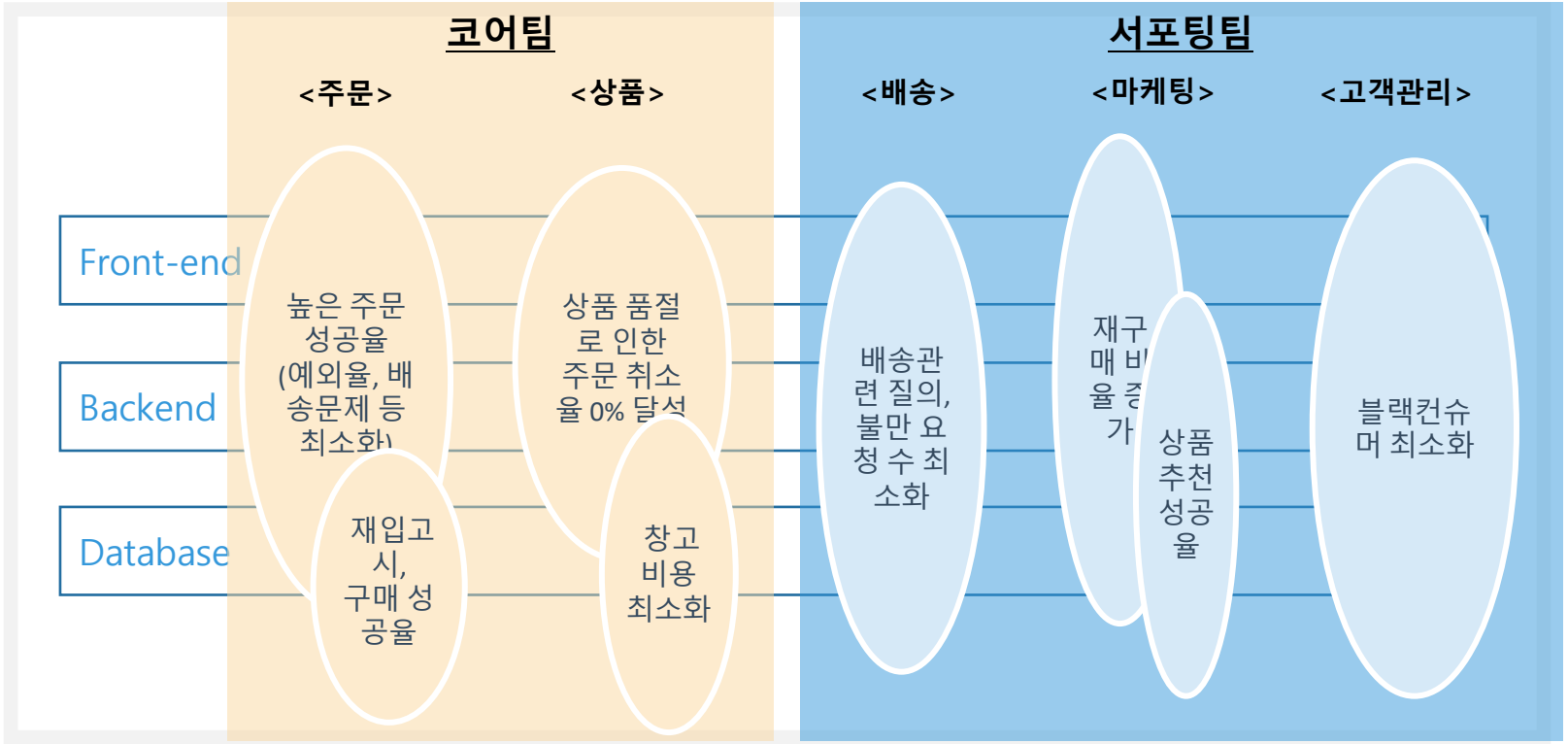
회사의 성장 – 너무 많은 Concerns



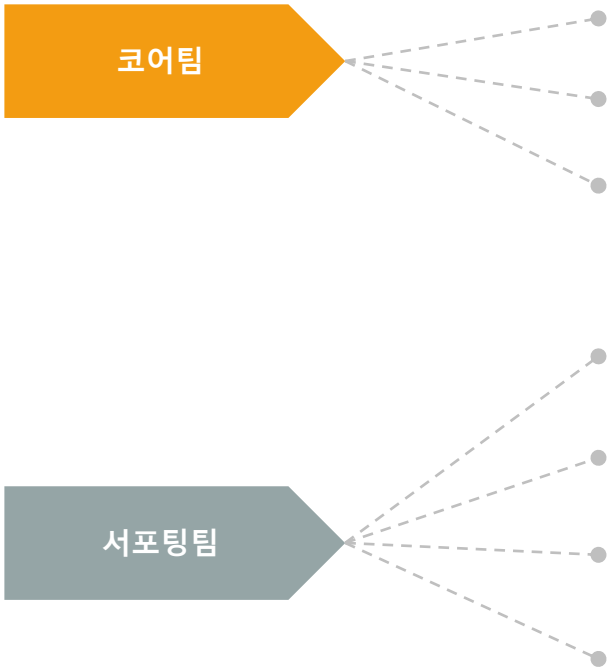
User Stories 와 책임소재 → 너무 많은 Concerns

1. The customer enters the number of products to order and clicks the order button.
2. When an order is placed, the delivery team arranges for delivery of the product.
3. As the order or cancellation is completed, the product stock is changed(+/-).
4. Customers can also purchase products that are out of stock and This can be resolved after registering products later
5. A notification is sent to the customer whenever the delivery status changes.
6. Customers who repeat orders and cancellations frequently are blacklisted, and customers with high order purchase rates are whitelisted.
7. The marketing team mails recommendation products tailored to the customers at 9 am daily.

회사의 성장 - Separation of Concerns



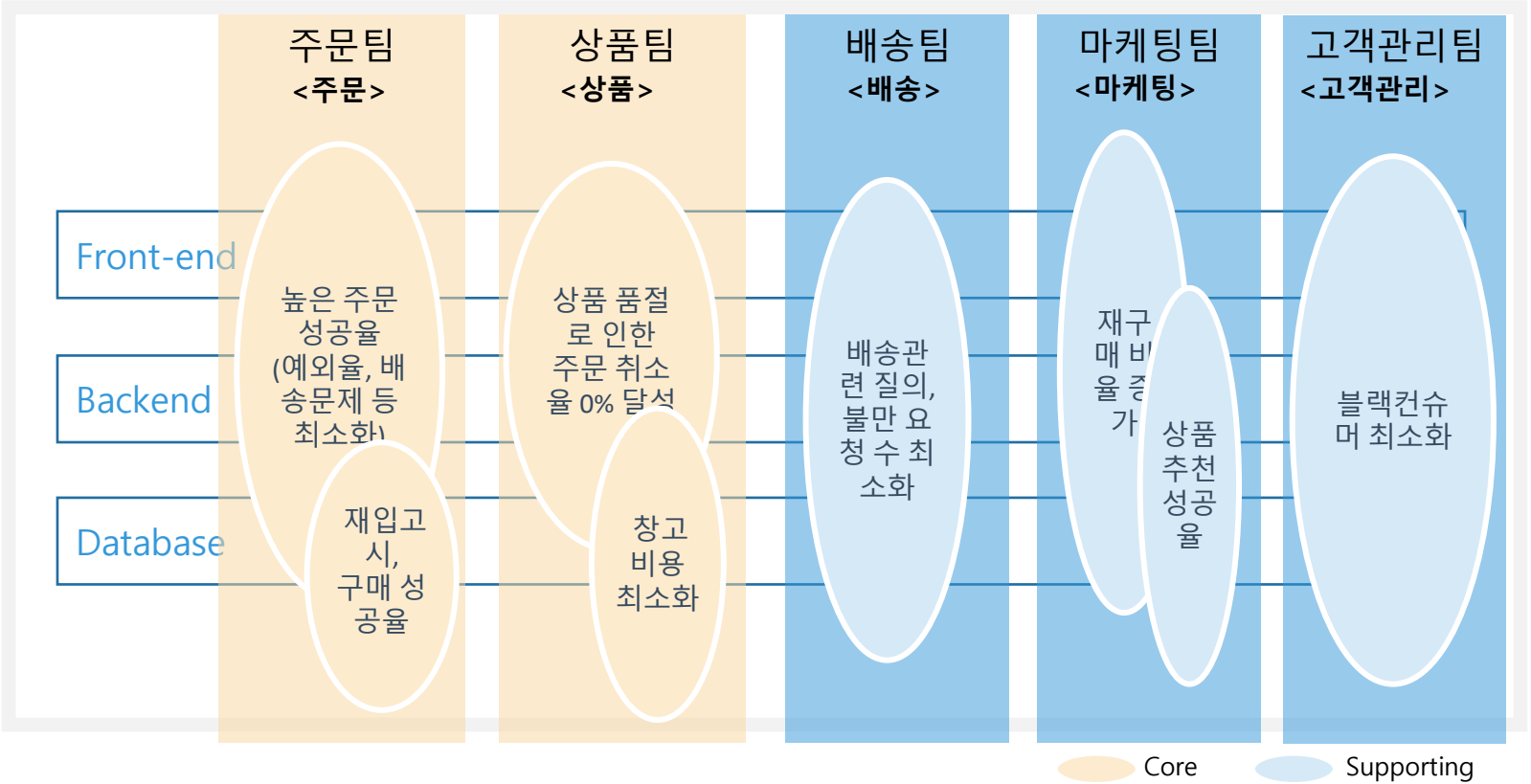
책임소재 – 중요한 것과 덜 중요한 것의 분리



- The customer enters the number of products to order and clicks the order button.
- When an order is placed, the delivery team arranges for delivery of the product.
- As the order or cancellation is completed, the product stock is changed(+/-).

- Customers can also purchase products that are out of stock and This can be resolved after registering products later
- A notification is sent to the customer whenever the delivery status changes.
- Customers who repeat orders and cancellations frequently are blacklisted, and customers with high order purchase rates are whitelisted.
- The marketing team mails recommendation products tailored to the customers at 9 am daily.

회사의 성장 – 자치성과 동기부여

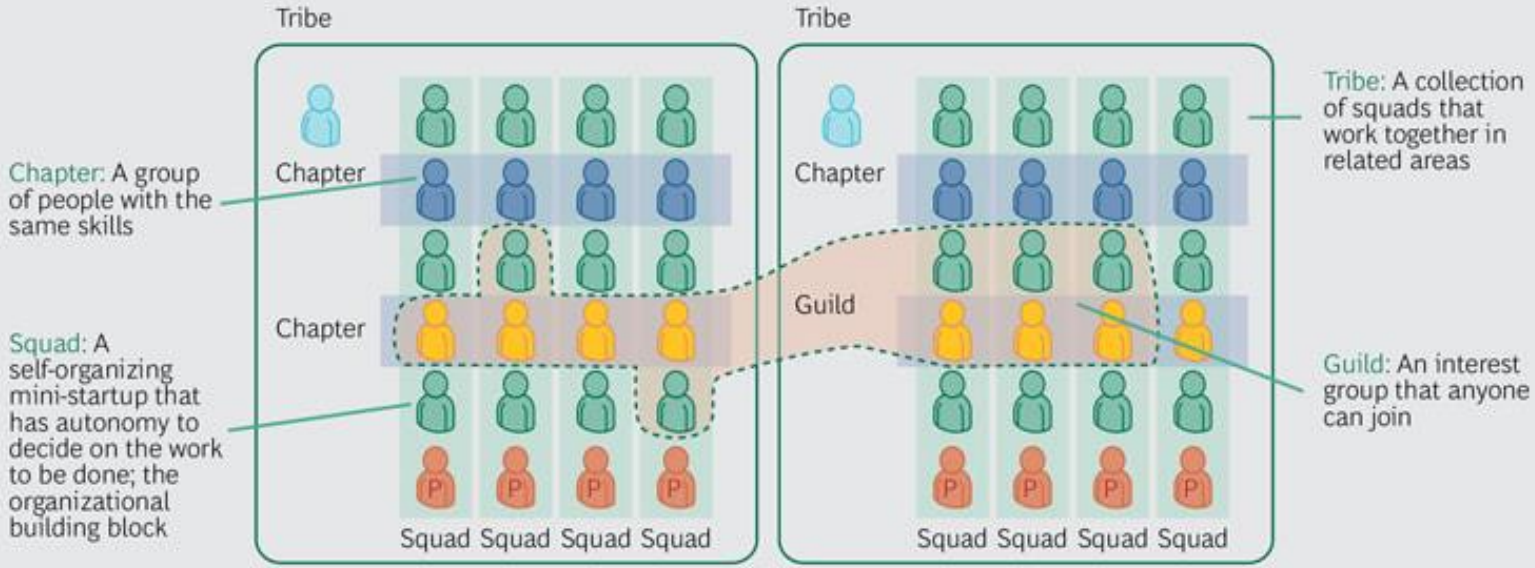


책임소재 – 자치성과 동기부여

- The customer enters the number of products to order and clicks the order button.
- When an order is placed, the delivery team arranges for delivery of the product.
- As the order or cancellation is completed, the product stock is changed(+/-).
- Customers can also purchase products that are out of stock and This can be resolved after registering products later

- A notification is sent to the customer whenever the delivery status changes.
- Customers who repeat orders and cancellations frequently are blacklisted, and customers with high order purchase rates are whitelisted.
- The marketing team mails recommendation products tailored to the customers at 9 am daily.

조직 전환 방안 – TO-BE



Source: Spotify

사례1 – ING BANK

This way of working is based on 8 important principles...

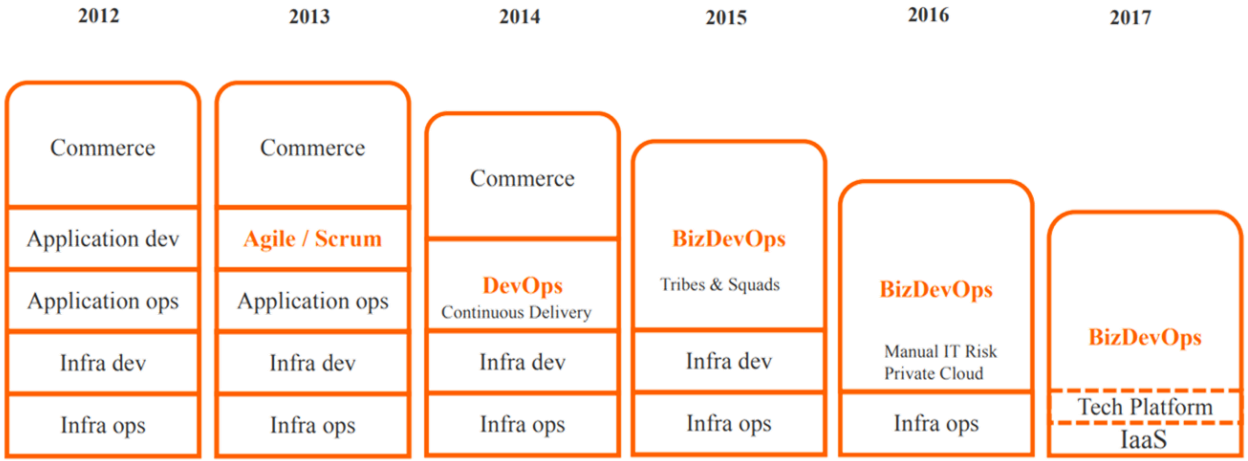
Principles

Inspired by the Agile methodology, eight principles are at the heart of our Way of Working:

1. We work in high **performing** teams
2. We **empower** teams
3. We care about talent and **craftmanship**
4. We continously **learn** from customers and apply learning to improve
5. We set **priorities** with the big picture in mind
6. We are consistent in our **organisational** design and way of working
7. We organise for **simplicity**
8. We **re-use** instead of reinvent

사례1 – ING BANK

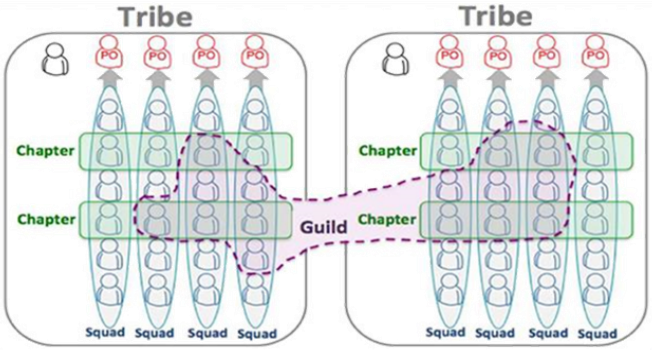
In the past five years, ING has been reorganizing for speed and skill. Roles and responsibilities have shifted radically



Engineer: From single discipline to **full stack engineers**: designing, coding, test engineering, infra engineering, etc
Product Owner: From writing PIDs to product vision and backlog to **end to end bizdevops responsibility**
IT Manager: from delivery manager to perhaps the most differntiating role: **skill and competency coach**.

사례1 – ING BANK

And this is how we organize ourselves

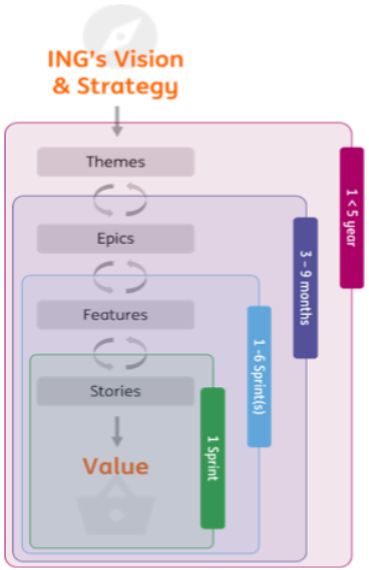


- **Squads**, are high performing "stable" teams who provide the execution power to the One WoW
- A **Tribe** is a collection of Squads organized around the same purpose
- **Chapters** are formal "groups" of members with the same background / expertise deciding on how things need to be done within a Tribe, regarding their area of expertise
- A **Guild** is a group of experts or community, which can be set up across Tribes (and even across countries) typically based on a shared interest in a technology or product / service

Squad over I, Tribe over Squad, Company over Tribe, Customer over Company

사례1 – ING BANK

These are the fundamentals of One Way of Working Agile



- A **Theme** connects the dots between Strategy and execution. Hence, a Theme represents an (investment) area that supports the strategic vision. Themes are representing one to five year(s).
- A Theme is broken down in multiple **Epics**: large scale bank initiatives that will be delivered in one to three quarters. At the Epic level, a.o. the benefits should be defined.
- An Epic can be broken down in one or more **Features**. A Feature reflects **part of the value** (both functional and non-functional) for a stakeholder that could be delivered within multiple (typically one to six) Sprints.
- Features have to be refined to **Story** level. Stories are explicitly defined by teams themselves. They have to be small enough to be picked up by one team and delivered within one Sprint.

A structured way to manage the demand and focus on the minimum viable products

사례1 – ING BANK

Our way of working in the new organisation

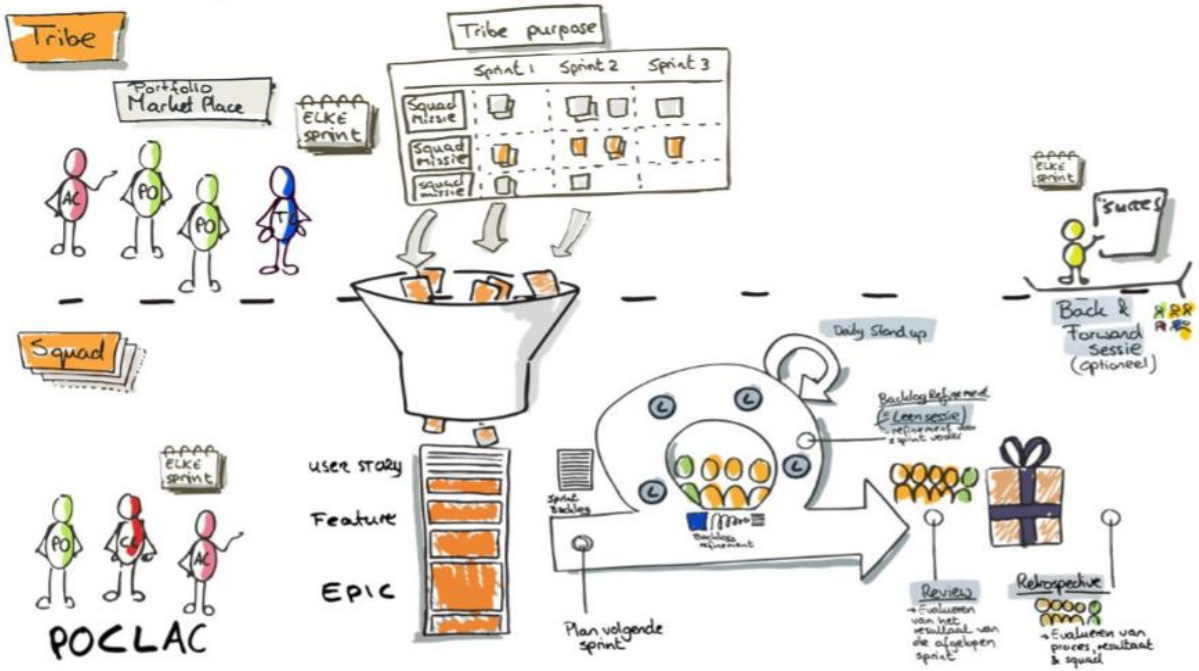


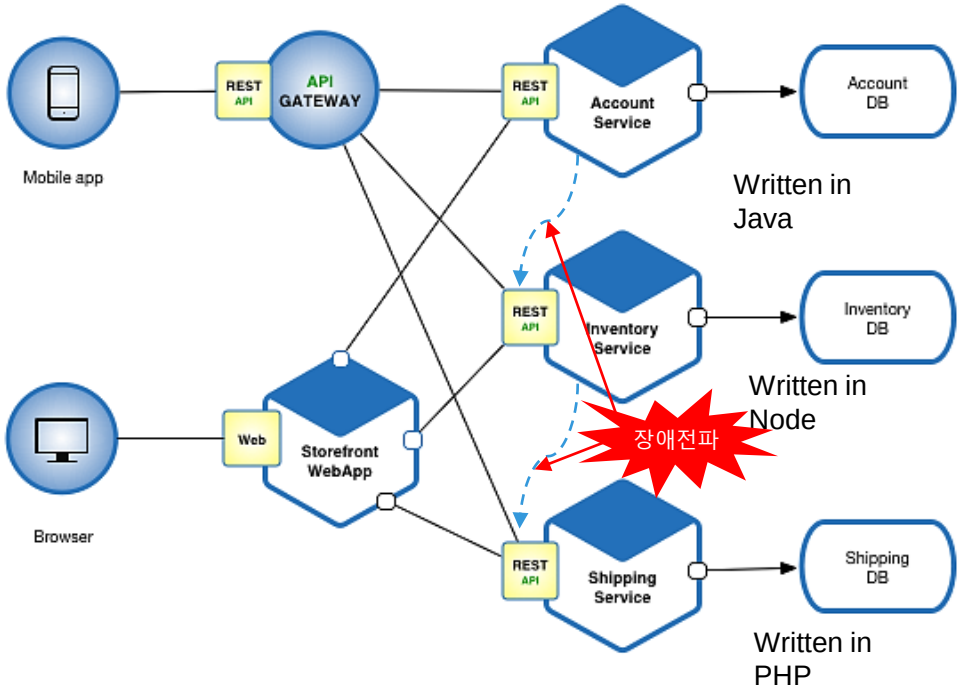
Table of content

Microservice and
Event-storming-Based
DevOps Project

UEngine

Approach #1 : Micro Service Architecture

Contract based, Polyglot Programming
** Separation of Concerns, Parallel Development, Easy Outsourcing**



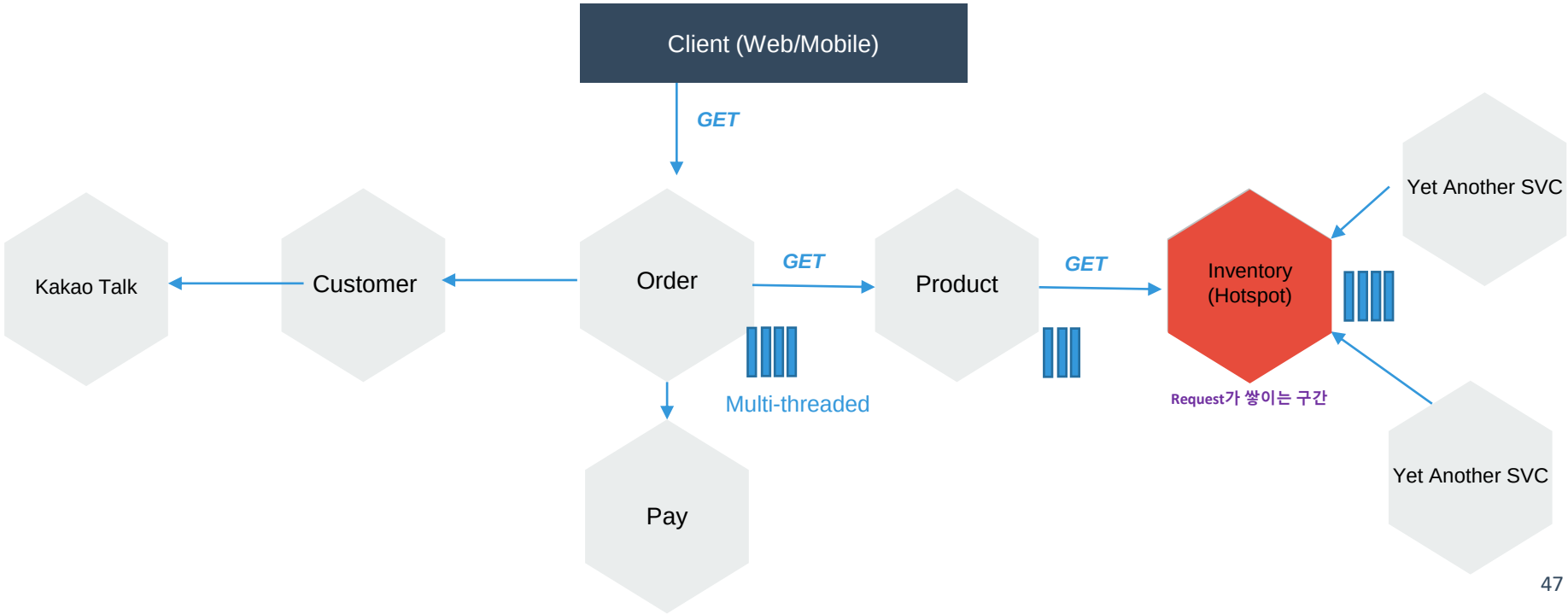
[Limitation]

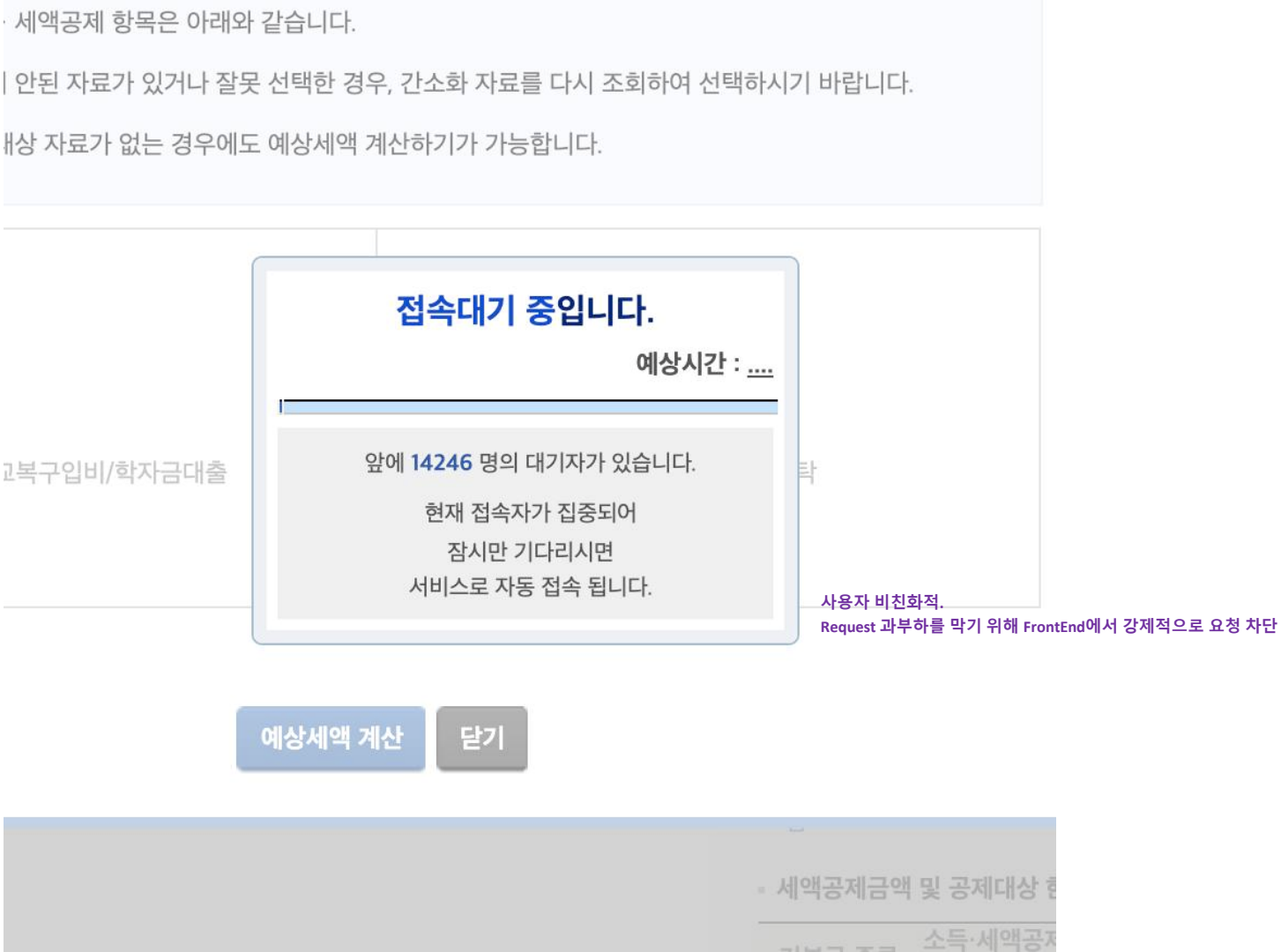
1. Code Coupling 해소
   But, Time Coupling 잔존
2. 서비스 Blocking 가능성 내재
3. 장애 전파 우려
4. Point to Point 연결에 따른 복잡한 스파게티 네트워크

Data Projection in Request-Response model

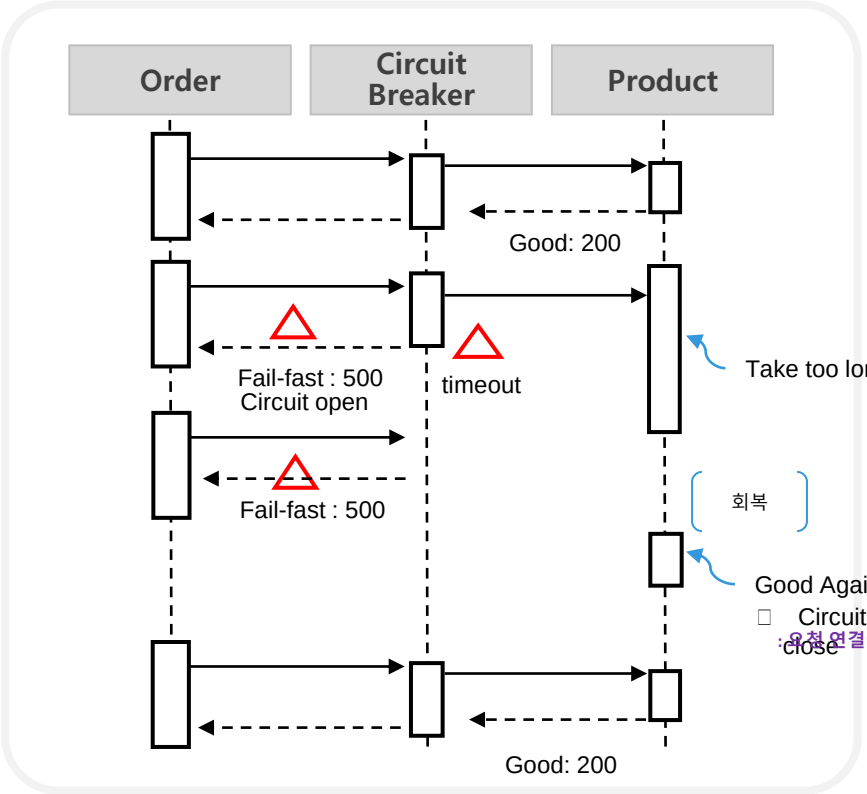
서비스구성 (Request-Response, Sync 호출)

- 성능저하
- 장애전파




세액공제 항목은 아래와 같습니다.
안된 자료가 있거나 잘못 선택한 경우, 간소화 자료를 다시 조회하여 선택하시기 바랍니다.
상 자료가 없는 경우에도 예상세액 계산하기가 가능합니다.
접속대기 중입니다.
예상시간 : ....
앞에 14246 명의 대기자가 있습니다.
현재 접속자가 집중되어
잠시만 기다리시면
서비스로 자동 접속 됩니다.
사용자 비친화적.
Request 과부하를 막기 위해 FrontEnd에서 강제적으로 요청 차단
예상세액 계산
닫기

장애전파 차단: 서킷브레이커 패턴



하나의 트랜잭션이 가능한 블로킹이 발생하지 않도록 미리 차단, Fail-Fast 전략

- 메모리 사용 폭주 막음

장애 감지 시, 차단기 작동(Open)

↓

일시 동안 Fallback으로 서비스 대체

↓

일부 트래픽으로 서비스 정상여부 확인

↓

정상 확인 시, 차단기 해제(Close)

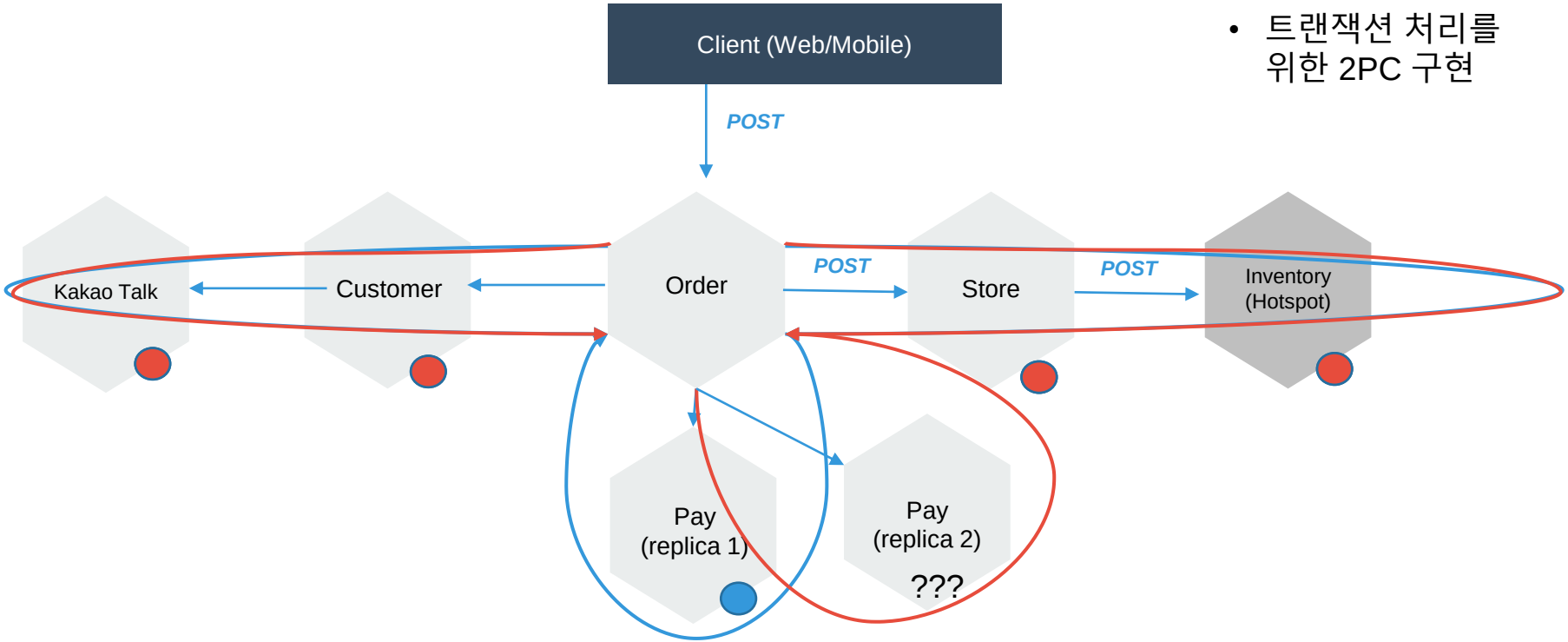
Write-side using Req-Resp:
2-Phase Commit

2-Phase Commit (2단계 커밋)
1) 실제 요청 처리 전 리허설 단계를 미리 둠.
2) 리허설 단계에서 정상적으로 처리가 되면 실제 요청을 처리함.

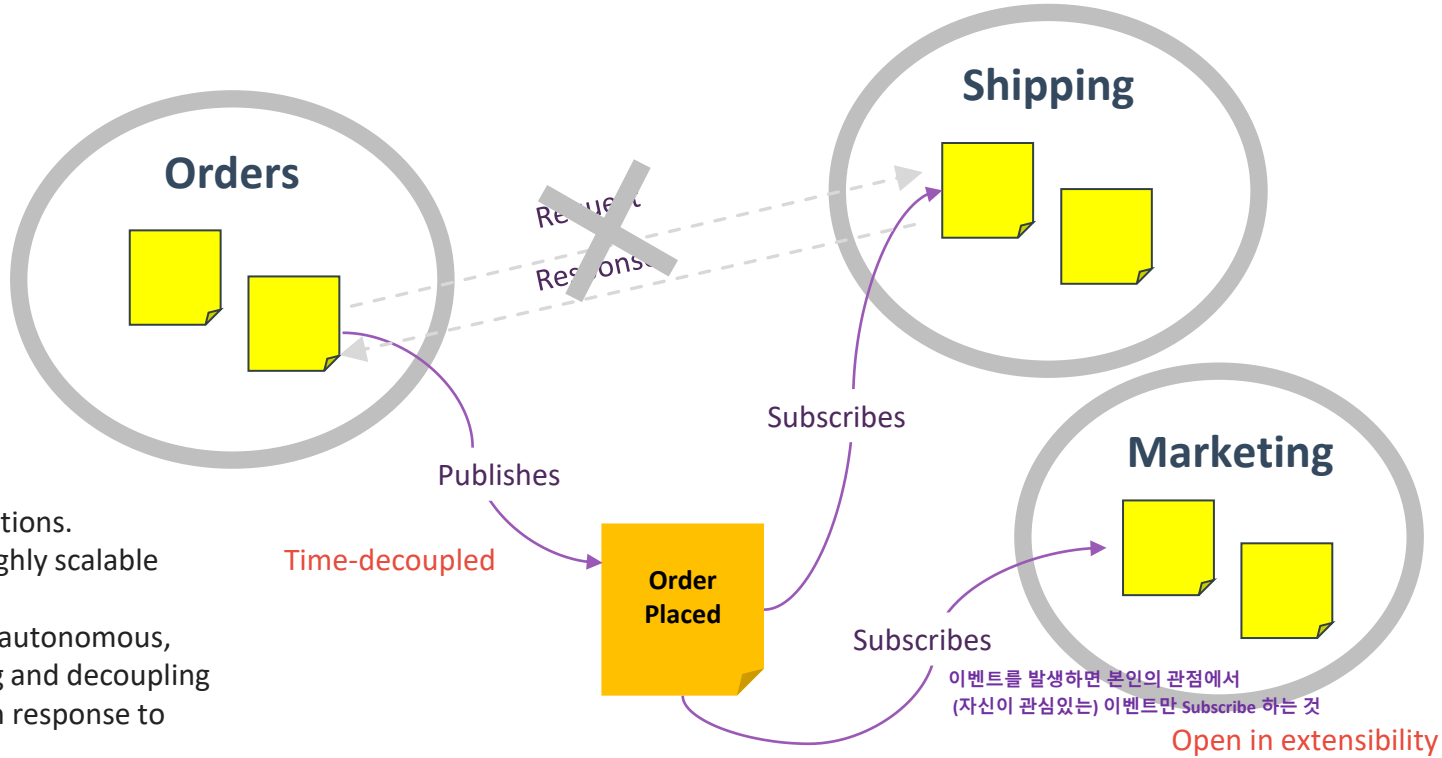
단, 이중으로 리소스 사용이 발생하여 과부화 가능성이 높음. (성능저하)
따라서, MSA에서는2-Phase Commit은 사용하지 않는 방식임.

서비스구성 (Request-Response, Sync 호출)

- 성능저하
- 장애전파
- 트랜잭션 처리를 위한 2PC 구현



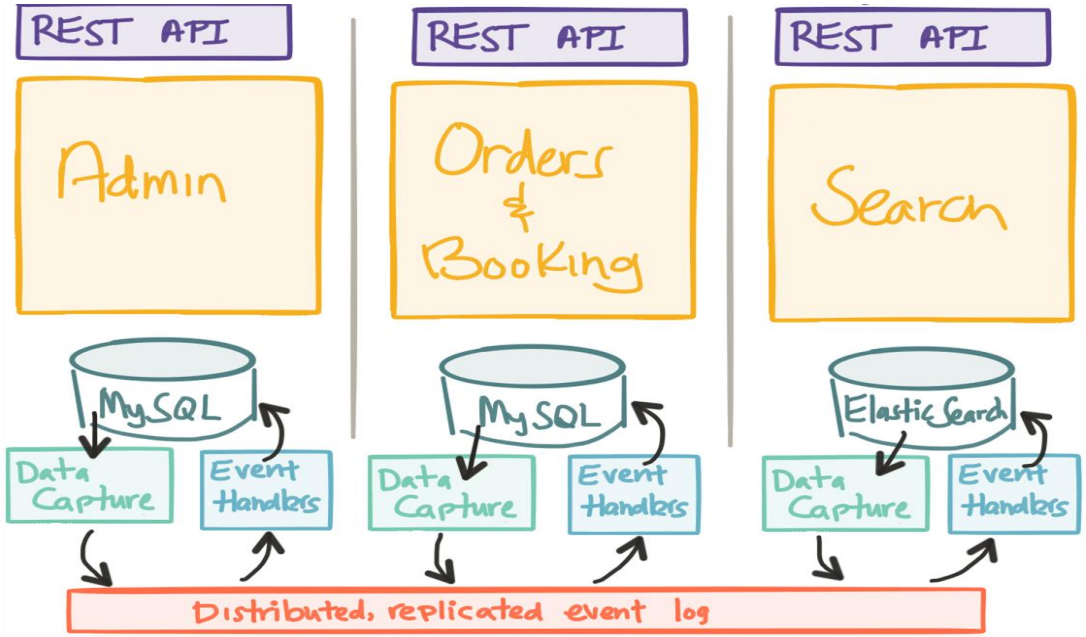
Approach #2 : Event Driven Architecture



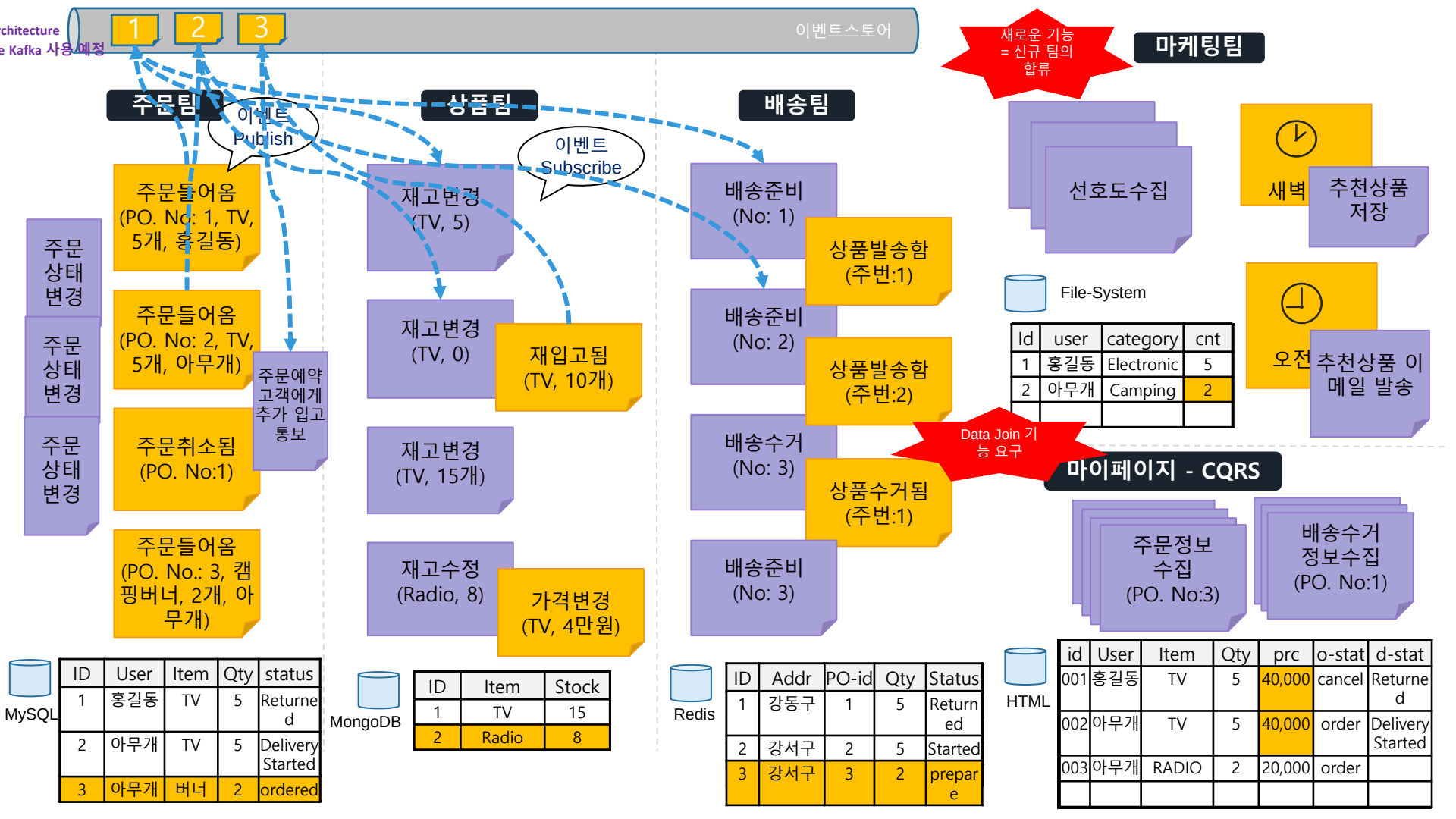
- No point-to-point integrations.
- High performance and highly scalable systems.
- Components can remain autonomous, being capable of coupling and decoupling into different networks in response to different events.
- Advanced analytics can be done using complex event processing.

Approach #2 : Event Driven Architecture

Polyglot-persistence



Source: https://blog.redelastic.com/corporate-arts-crafts-modelling-reactive-systems-with-event-storming-73c6236f5dd7, Kevin webber



Id	user	category	cnt
1	홍길동	Electronic	5
2	아무개	Camping	2

ID	User	Item	Qty	status
1	홍길동	TV	5	Returned
2	아무개	TV	5	Delivery Started
3	아무개	버너	2	ordered

ID	Item	Stock
1	TV	15
2	Radio	8

ID	Addr	PO-id	Qty	Status
1	강동구	1	5	Returned
2	강서구	2	5	Started
3	강서구	3	2	prepare

id	User	Item	Qty	prc	o-stat	d-stat
001	홍길동	TV	5	40,000	cancel	Returned
002	아무개	TV	5	40,000	order	Delivery Started
003	아무개	RADIO	2	20,000	order	

비동기 메시징 - 런타임 약결합

Loosely-Coupled Interaction via Asynchronous Messaging

Data streaming can decouple database with shared data

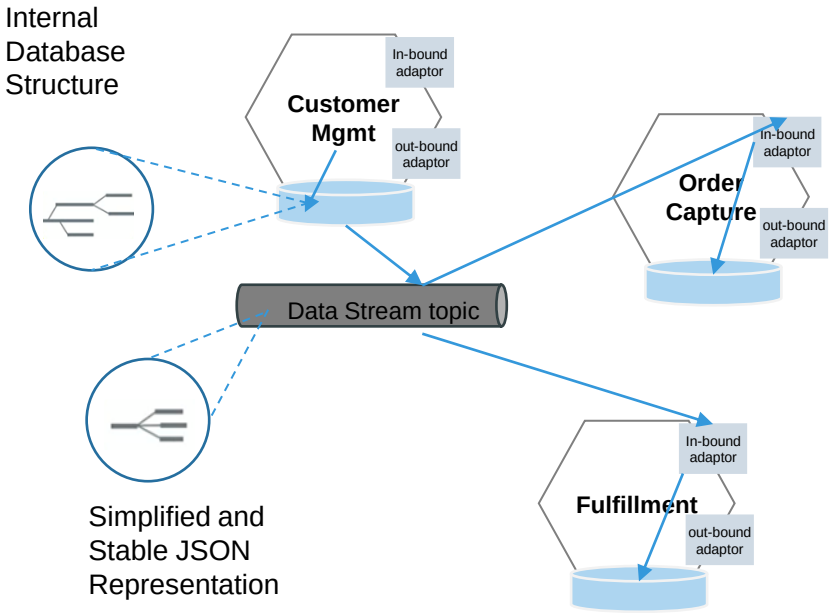
상대의 수신을 기다리지 않는 비동기식 메시징
Asynchronous messaging(streaming)

- Publish-Subscribe
- 준실시간으로 동작함
- **Decouples <u>"timing"</u> = Non-blocking**

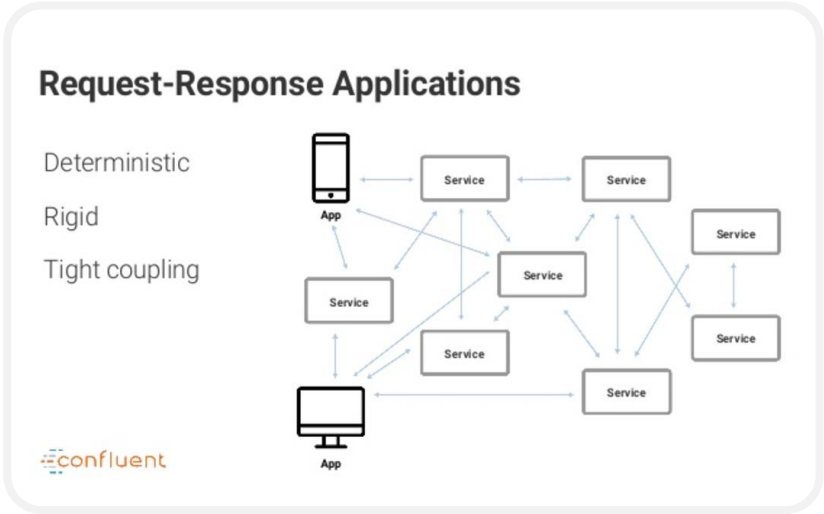
Data streams hide the internal implementation of the service

Client ignore parts of the stream they don't understand

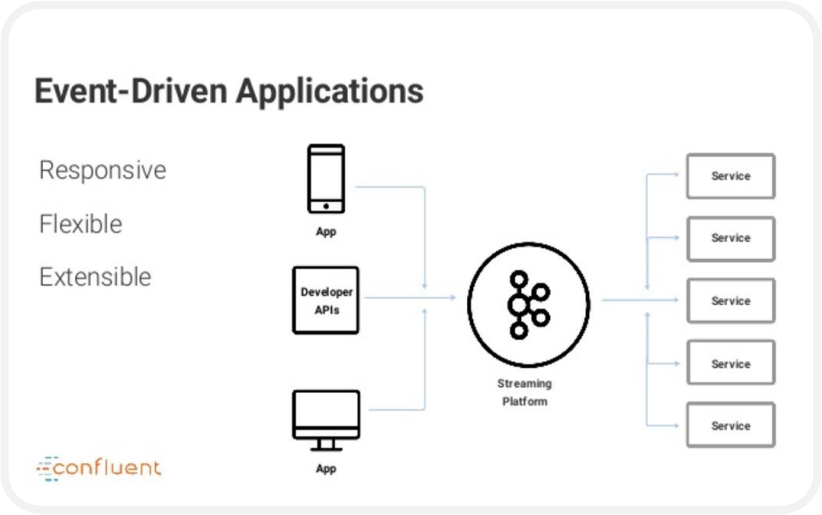
➔ 런타임에서도 약결합을 제공함



Microservice Integration with Event-driven Architecture



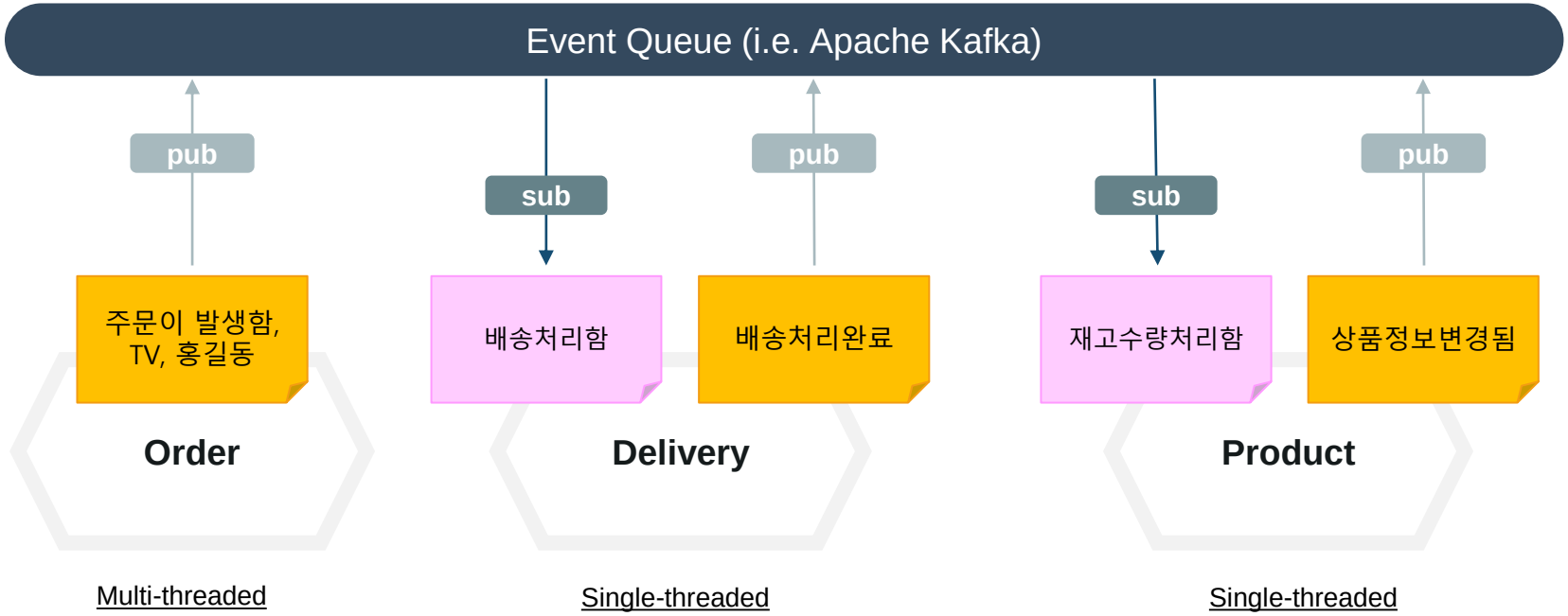
VS



- Point to Point Spagetti network
- Blocking Model
- System fault spread

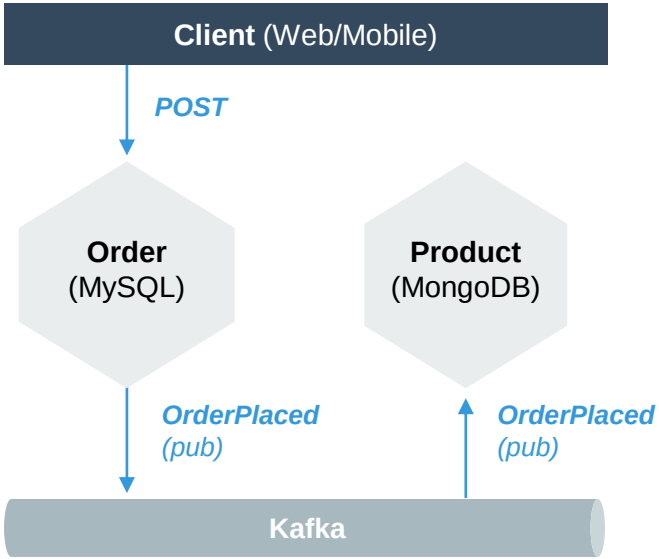
- Broadcasting
- Non-Blocking Model
- System fault isolation

Inter-microservice Call - Event Publish / Subscribe (PubSub)



Eventual Consistency를 통한 분산 트랜잭션 처리

- 서비스구성 (Eventual TX)



- 조회화면

-. 단점
Kafka(이벤트 스토어)에 이벤트를 Publich 하는 것이 트랜잭션 1개.
Kafka(이벤트 스토어)에서 이벤트를 수신하는 것이 트랜잭션 1개.

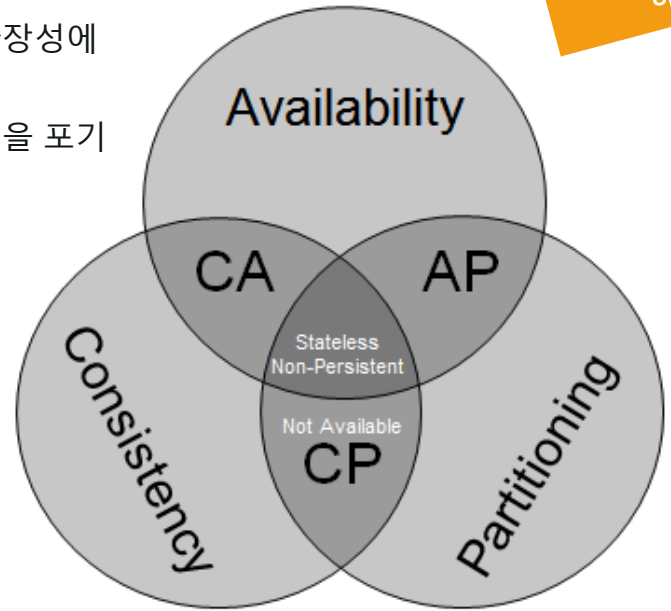
각각 트랜잭션으로 동작하므로, 순간 불일치할 수도 있음.

여기서 선택의 길을 만남…

- 내 서비스에서 데이터 불일치가 얼마나 미션 크리티컬 한가?
- 크리티컬 하다고 응답한다면, 내 서비스의 성능과 확장성에 비하여 크리티컬 한가?
- 성능과 확장성 보다 크리티컬 하다면, 성능과 확장성을 포기할 수 있는가?

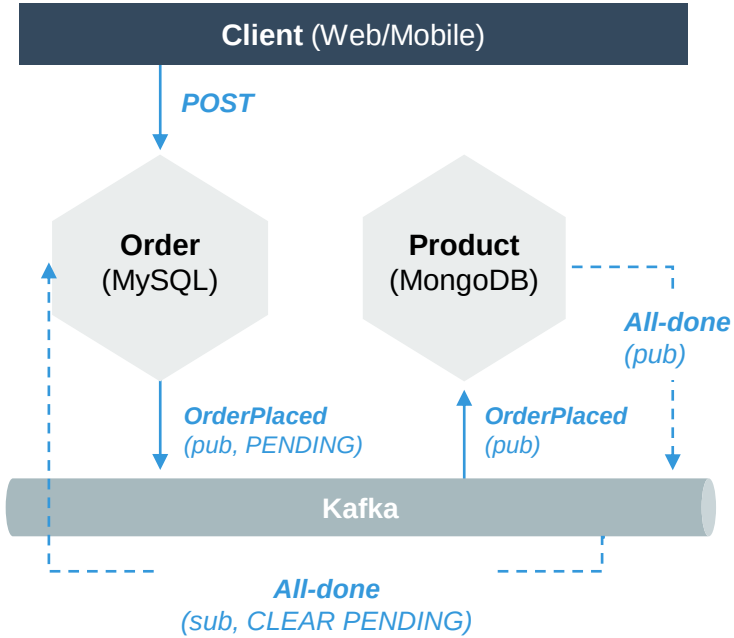
"성공한 내 서비스의 경쟁자들은 무엇을 포기했는가?"

- 강력한 의사결정 필요
  (무엇으로 태어날 것인가…)



Eventual TX – 불일치가 Mission Critical 한 경우

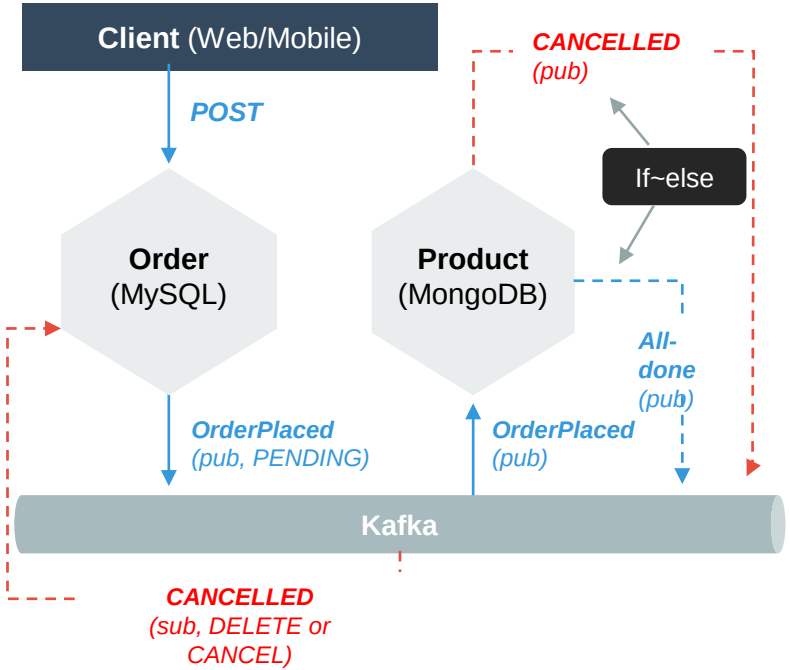
- 서비스구성 (Eventual TX)



- 조회화면

Eventual TX – Rollback (Saga Pattern)

- 서비스구성 (Eventual TX)



- 조회화면

Consistent In-consistent

주문량 : 9 , 재고량 : 2

주문량 : 9 , 재고량 : 2 (PENDING)

주문량 : 10 , 재고량 : 1

주문량 : 11 , 재고량 : 1 (PENDING)

주문량 : 10 , 재고량 : 0 □ **취소됨**

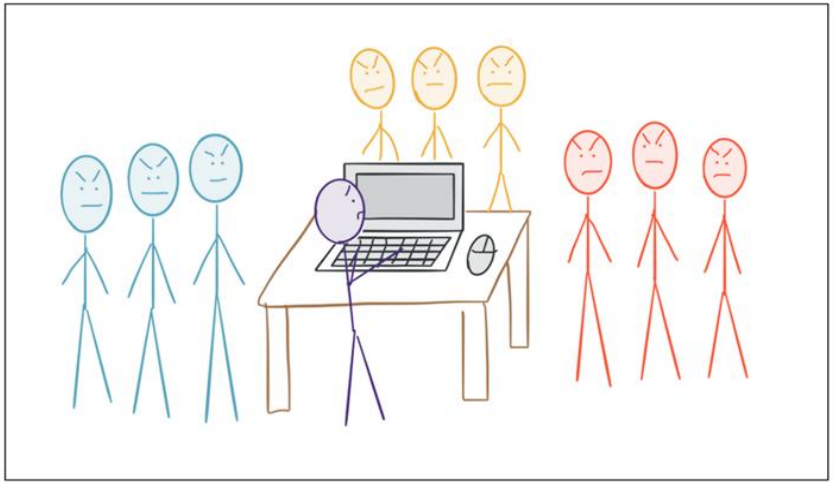
복구 &
결국 일치

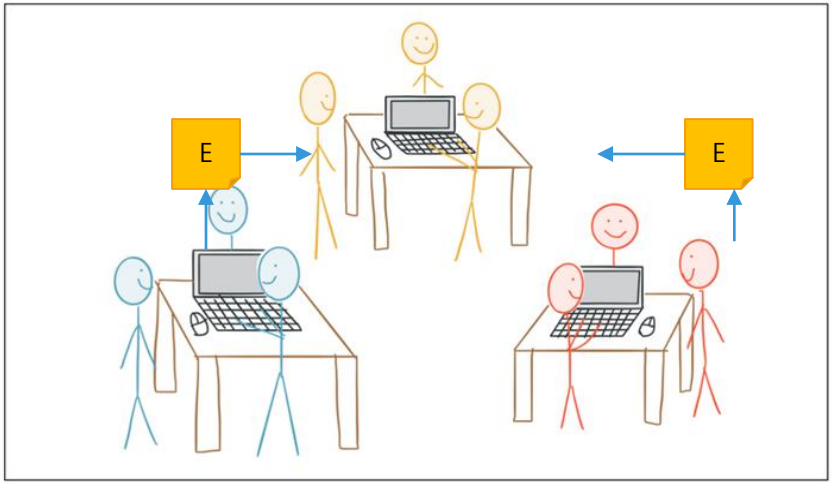
Benefits of Event-Sourcing?

- The event store provides a complete log of every state change, effectively creating an audit trail of the entire system.
- The state of any object can be recreated by replaying the event store.
- It removes the need to map relational tables to objects.
- An event store can feed data into multiple read databases.
- In the case of failure, the event store can be used to restore read databases.

장점)
주요한 사실들을 이벤트 스토어에 보관하긴 때문에, 백업 및 복구에 용이함.
이전 단계로 언제든지 복구 가능함.
이전 과거 데이터를 모두 갖고 있기에 미래 방향성 및 예측이 가능함. (데이터의 가치가 높아짐)

Event Driven MSA Architecture





Monolith:

- With Canonical Model
- (“Rule Them All” model)

Reactive Microservices:

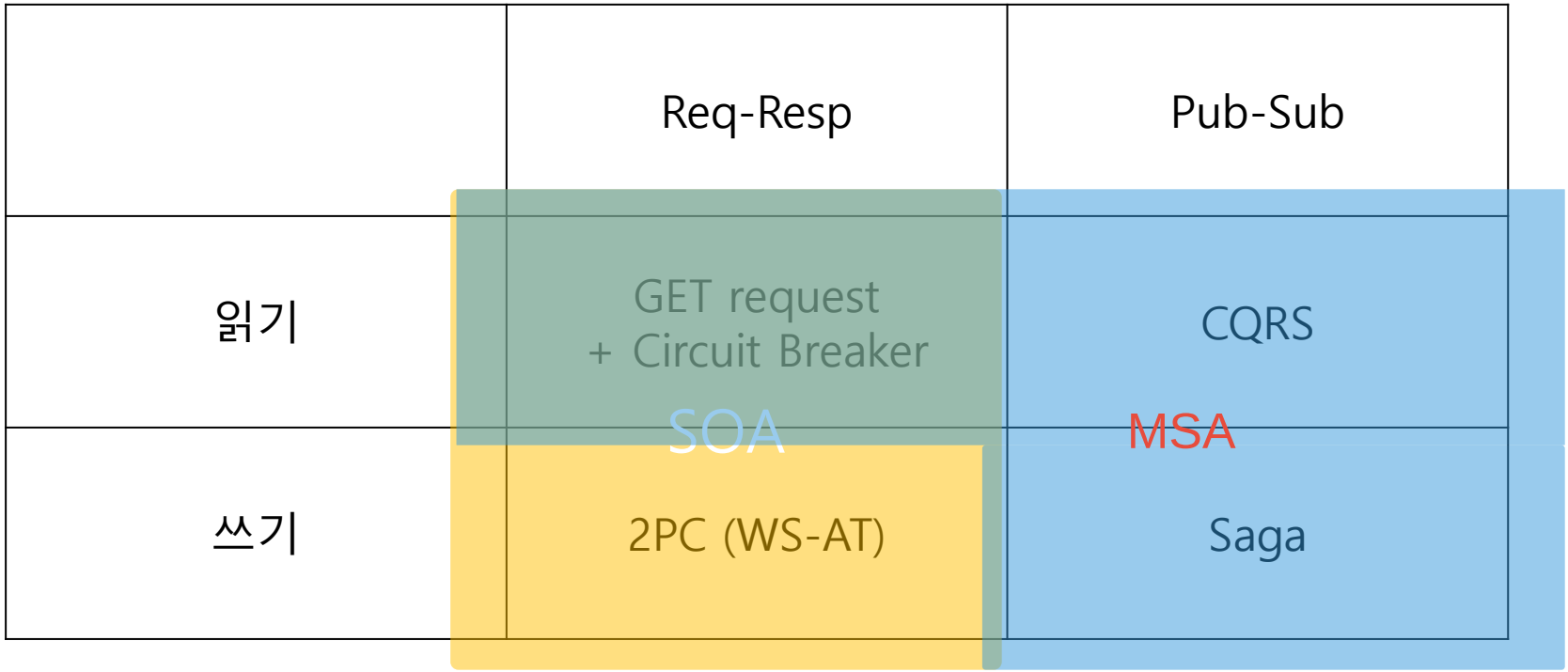
- Autonomously designed Model(Document)
- Polyglot Persistence

Source: https://blog.redelastic.com/corporate-arts-crafts-modelling-reactive-systems-with-event-storming-73c6236f5dd7, Kevin webber

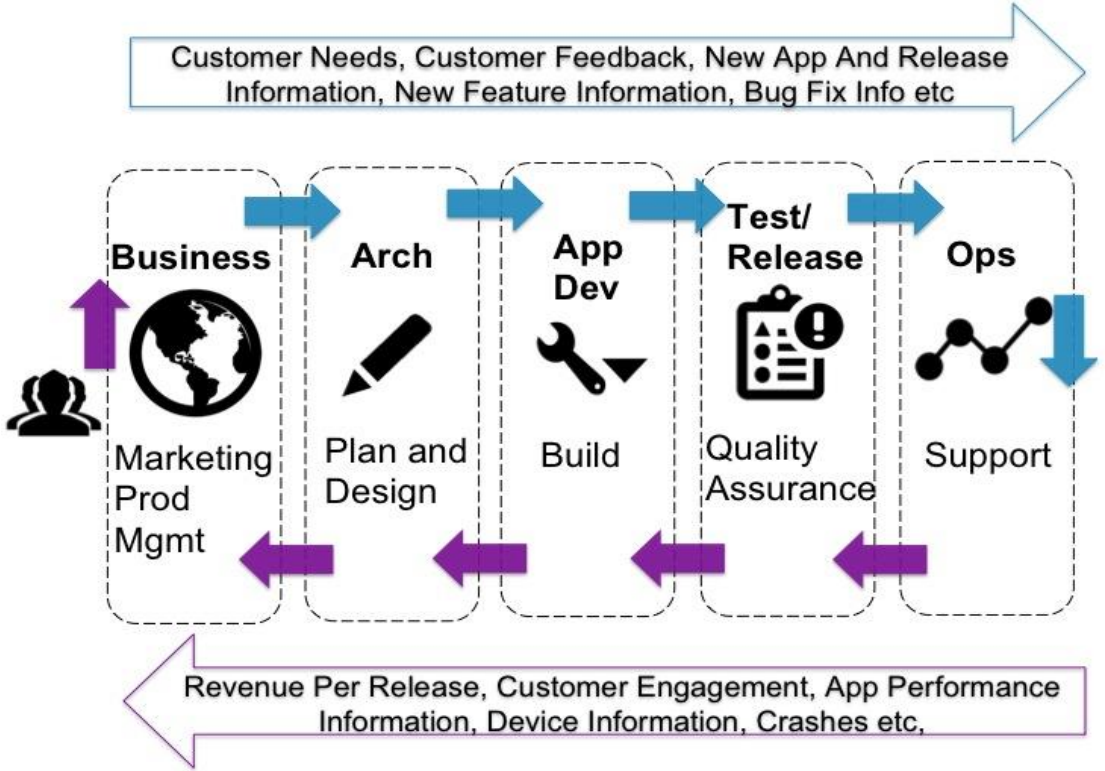
Case Study – 11번가

https://tv.naver.com/v/11212897

Integration Strategies Compared

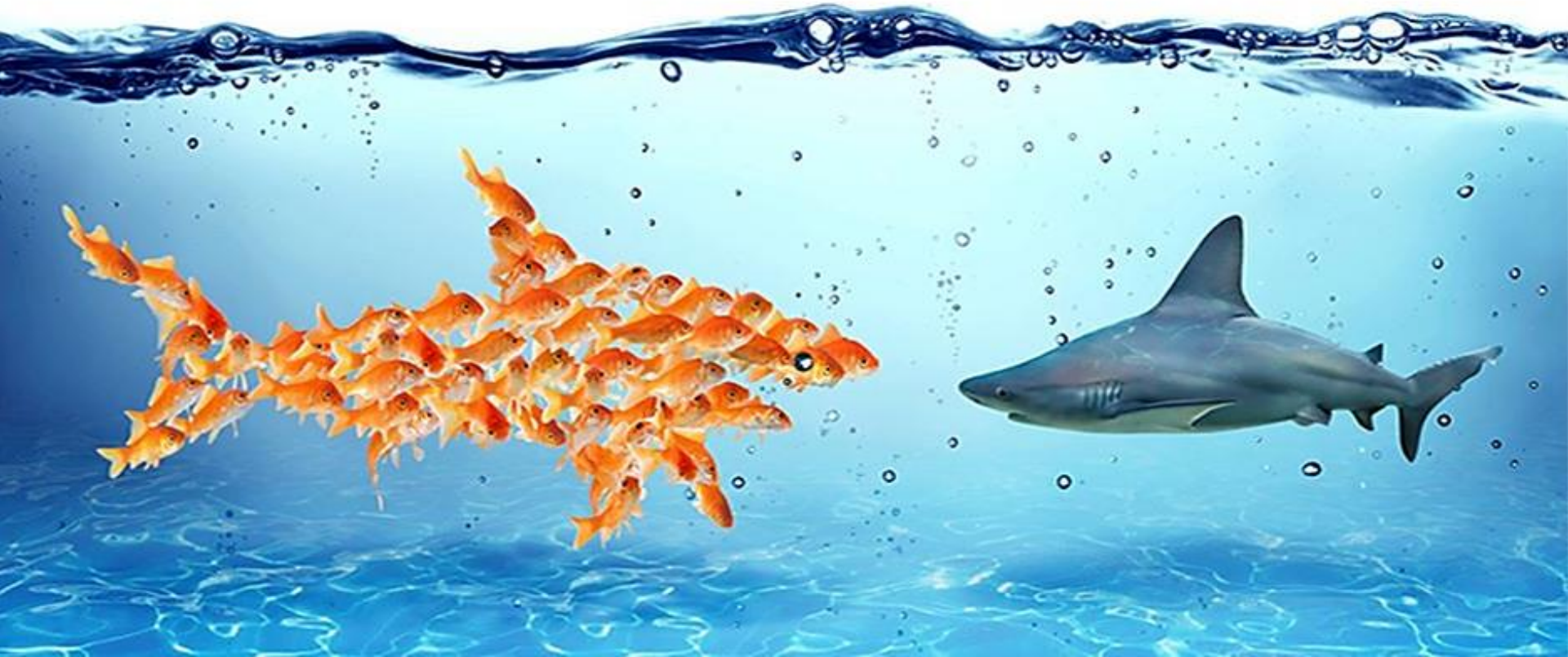


Approach #3: BizDevOps Process & Infra



- 1 Sprint = 1~2 Weeks Cycle 기반
- 지속적인 Needs 탐색

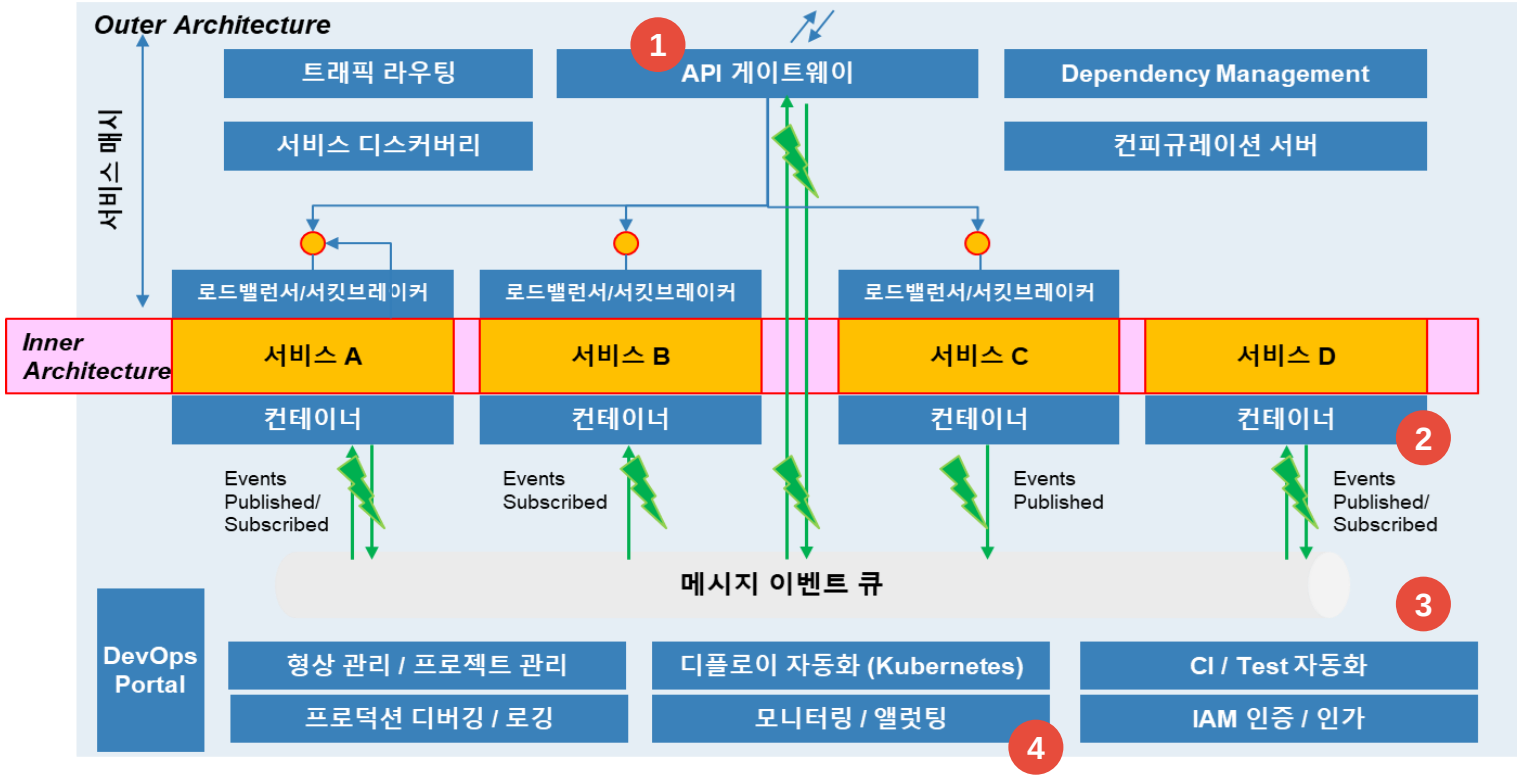
서비스를 죽지 않게 하려면...



항상성이 (Self-Healing) 자동으로 유지되면 어떨까?

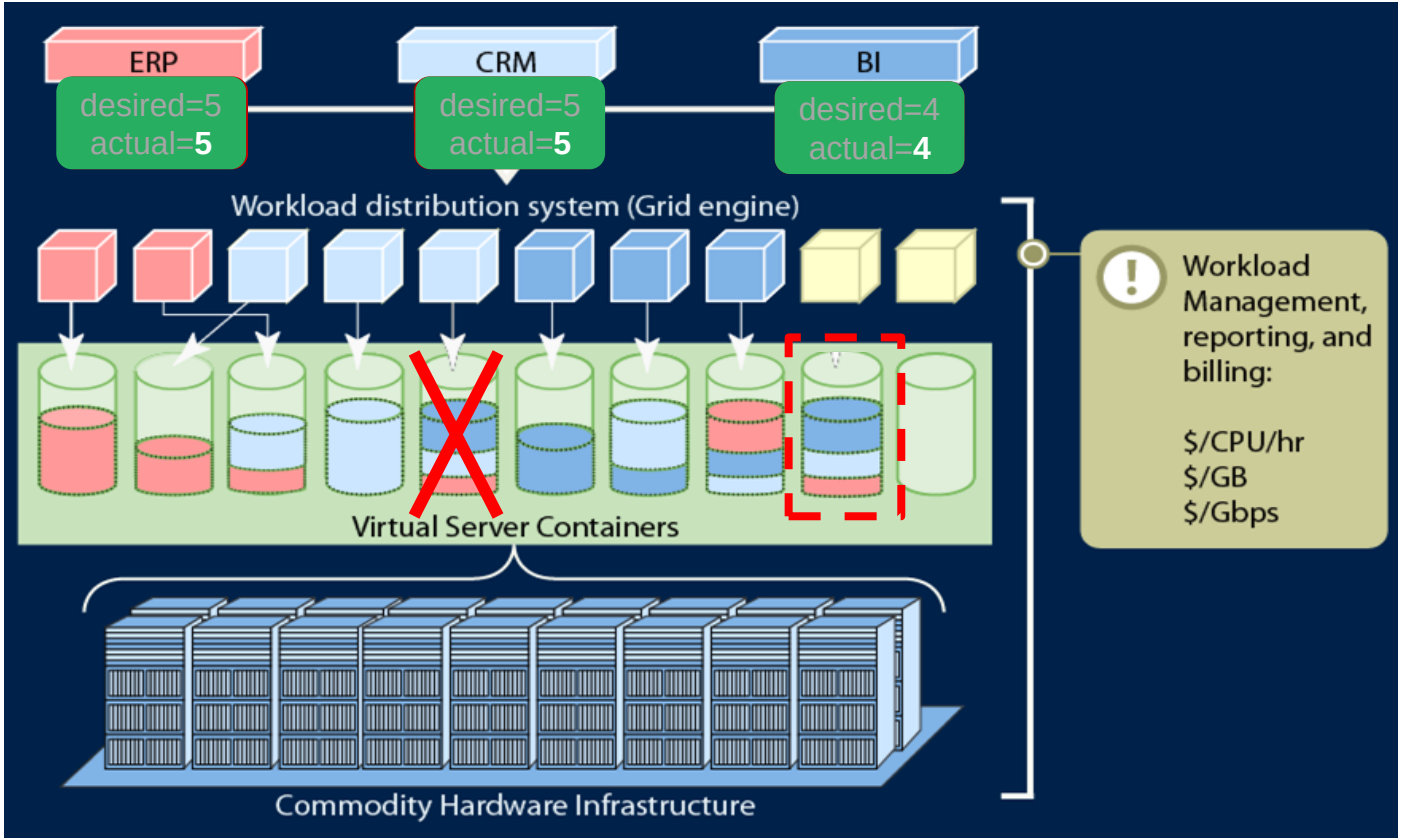
Kubernetes

Outer & Inner Architecture



1. API Gateway : API를 사용하여 통신하는 모든 서비스들의 인증 및 제어, 트래픽 모니터링 등 수행
2. Managed Container System : 서비스들의 중앙 관리 및 인증, 라우팅등의 기능 수행
3. Backing Services : 스토리지(Block, Object), Cache, Message Queue 등의 기능 영역
4. Observability Services : 서비스들의 이벤트 모니터링 및 알림, 로그 취합, 진단 등 수행

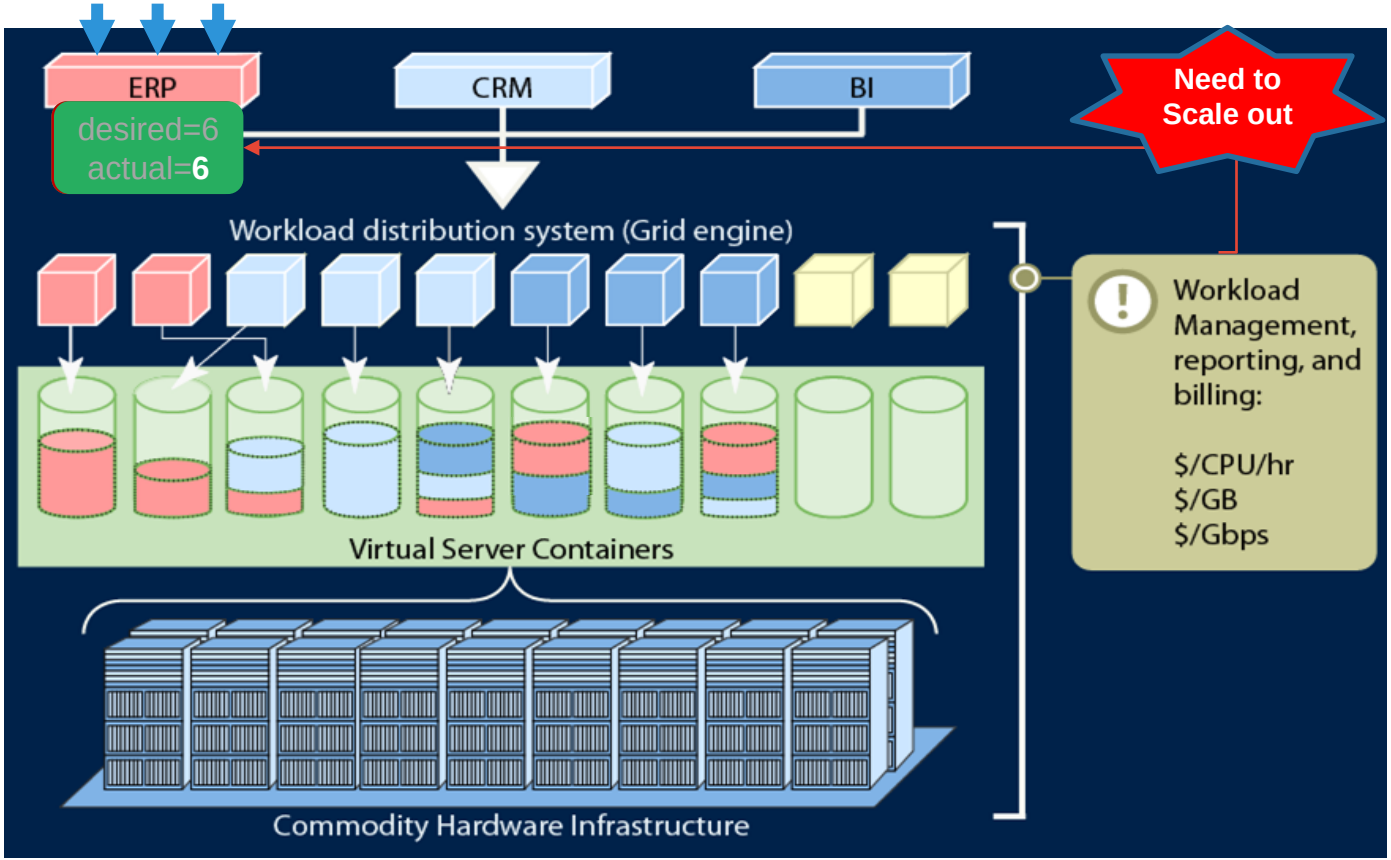
Container Orchestration – Self Healing Mechanism



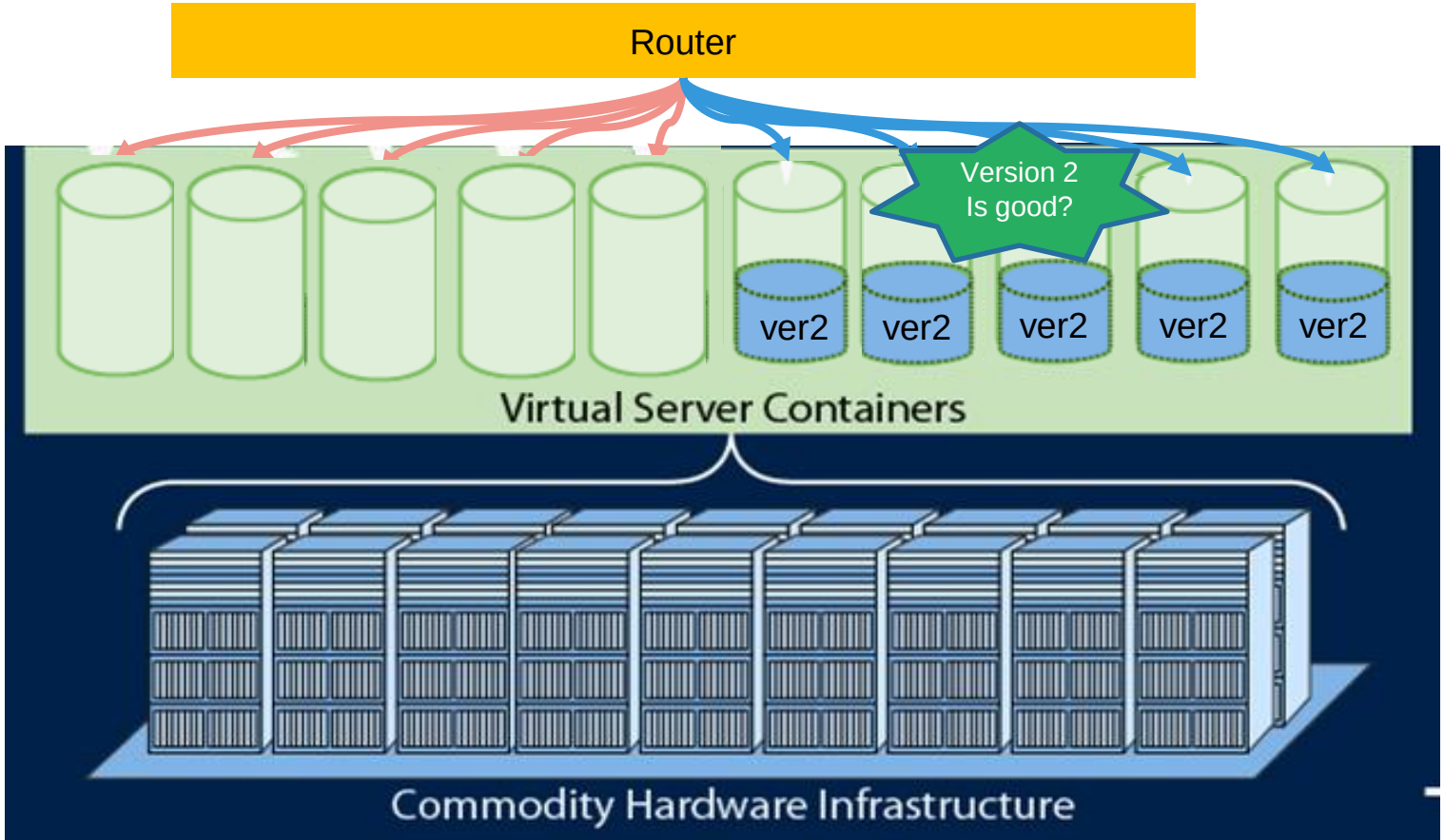
가 서비스 장애가 나도
수 있도록 분산하여 저장

만약, H/W 정의해놓을 배분율
다르게 되면 (쿠버네티스가 계
이에 해당하는 배분율을 지키
하나 더 할당하여 배분율 유지

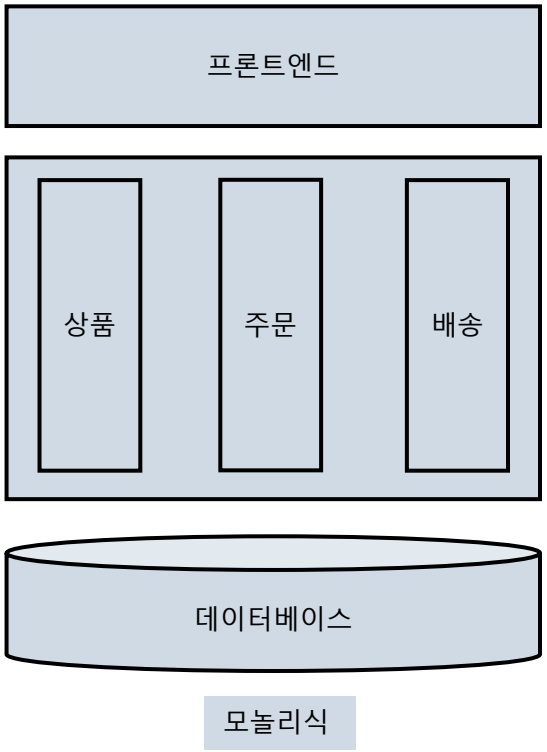
Container Orchestration – Scale out

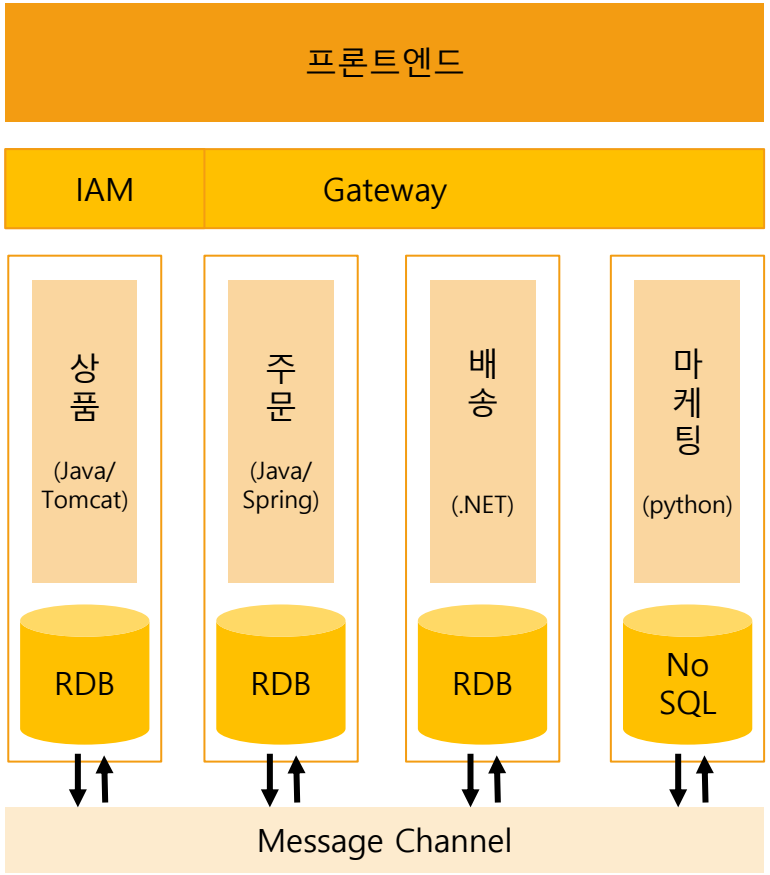


Container Orchestrator – Zero Down Time Deploy

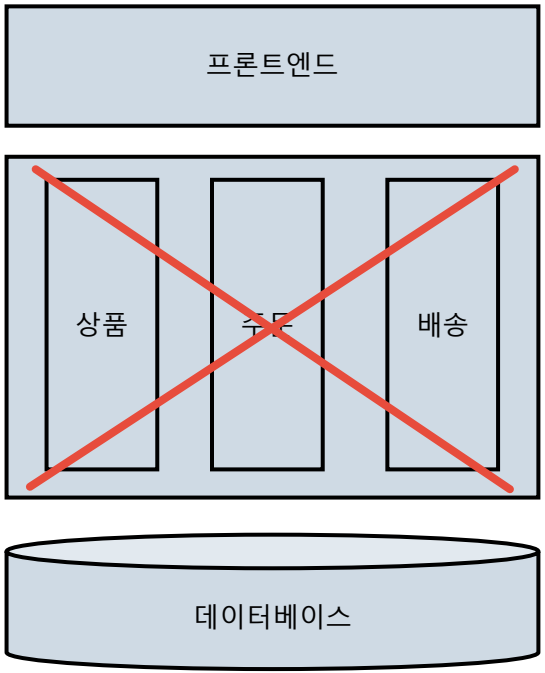


적용 아키텍처

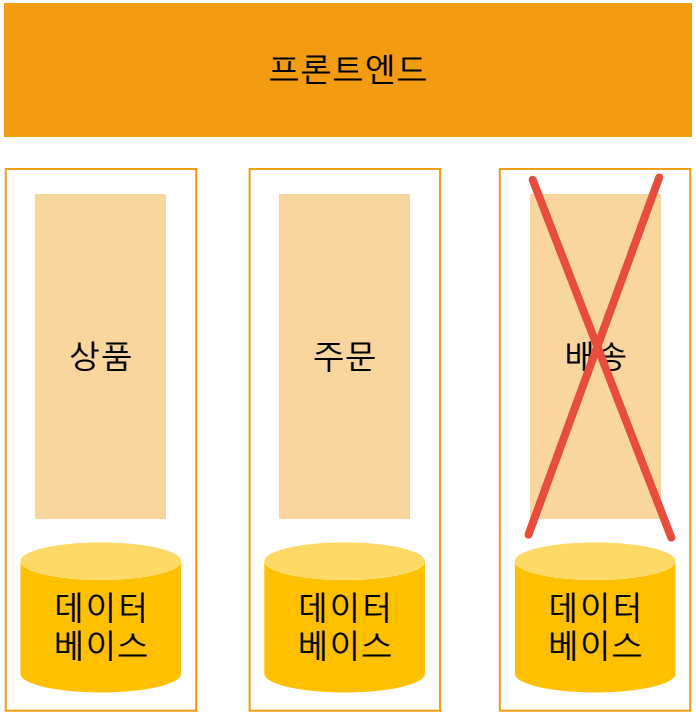

프론트엔드
상품
주문
배송
데이터베이스
모놀리식


프론트엔드
IAM
Gateway
상품
(Java/
Tomcat)
주문
(Java/
Spring)
배송
(.NET)
마케팅
(python)
RDB
RDB
RDB
No SQL
Message Channel

효과 – 장애 최소화

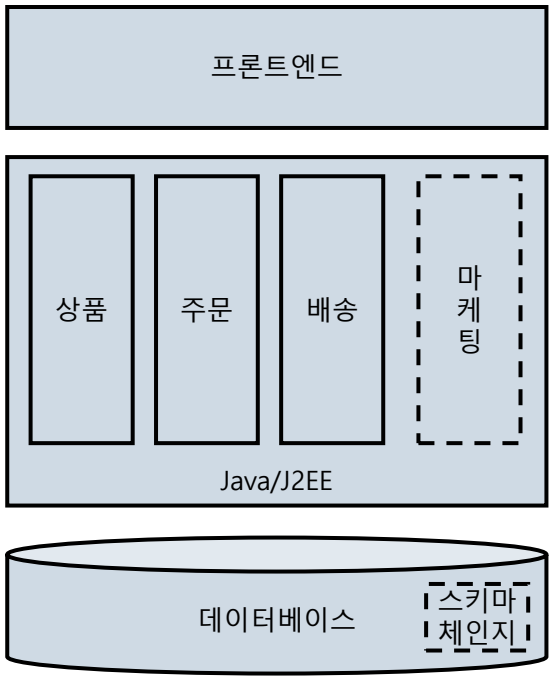


장애가 전파되며, 수동 복구로 장애 지연

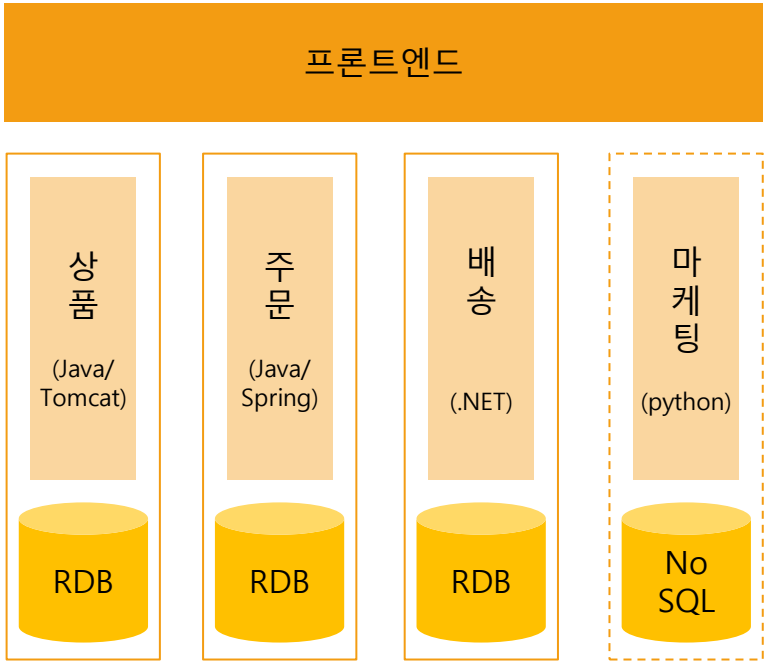


장애가 격리되며, 자동으로 복구됨(이전버전)

효과 – 기능 확장

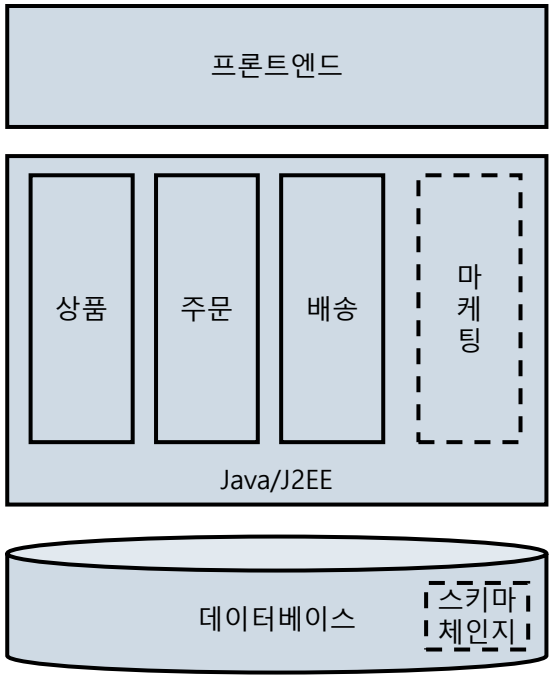


기존 코드의 수정 □ Big Impact

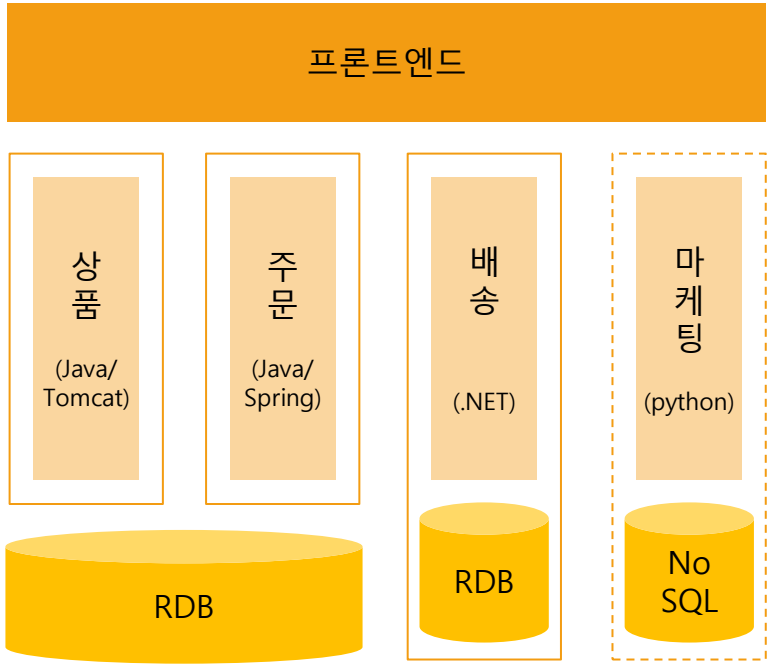


신규 마이크로서비스의 추가 □ Minimal Impact
자율적 기술선택(플랫폼 & 데이터레이아웃)

효과 – 기능 확장

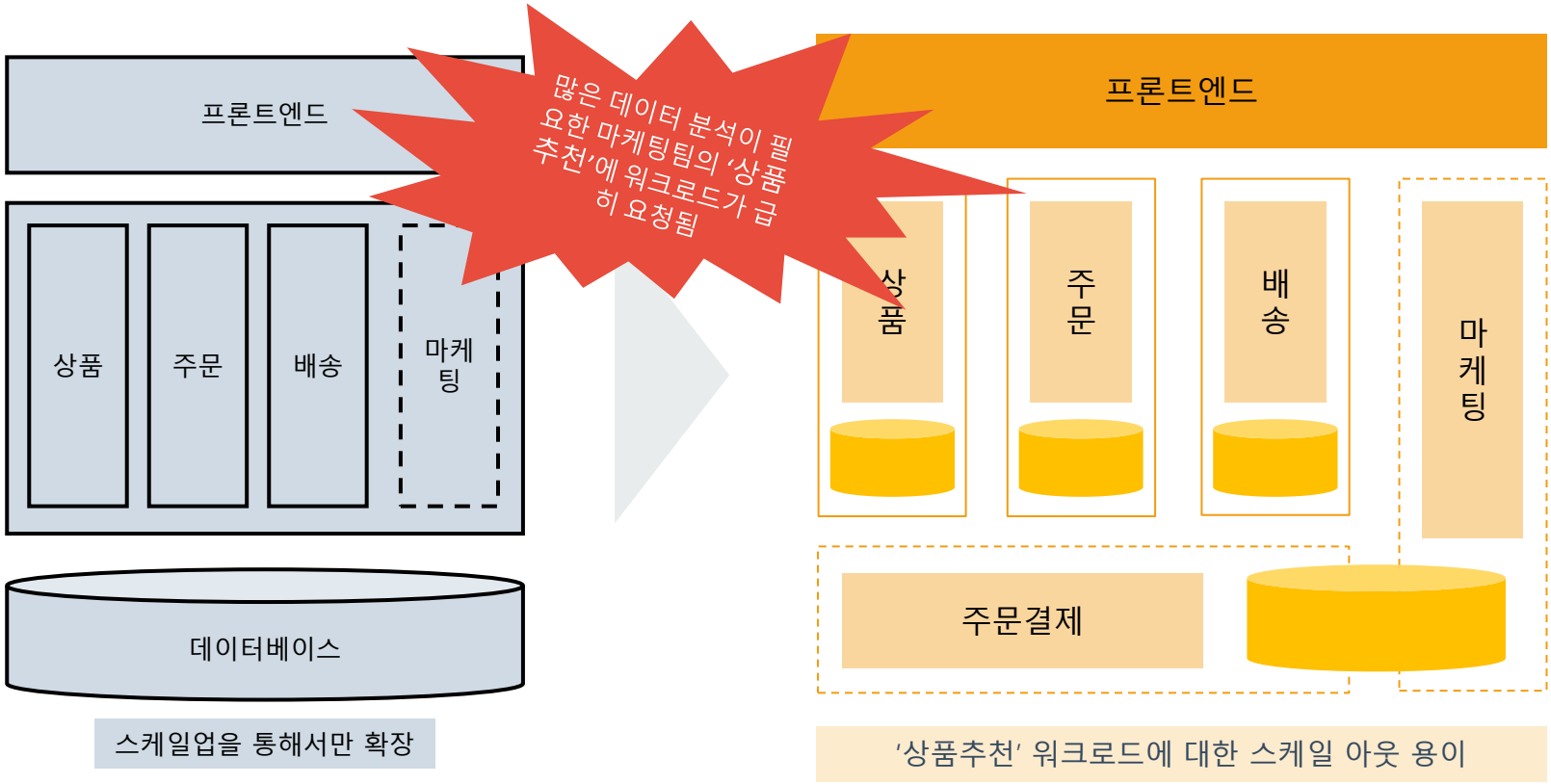


기존 코드의 수정 □ Big Impact



신규 마이크로서비스의 추가 □ Minimal Impact
자율적 기술선택(플랫폼 & 데이터레이아웃)

효과 – 성능 배분



스케일업을 통해서만 확장

'상품추천' 워크로드에 대한 스케일 아웃 용이

목표수준수립과 비용

늘어나는고객수를 처리하기 위한 확장

- → Core/Supporting 분리
- → 컨테이너 적용
- → 팀의 확장
- → 팀 자율성

팀자율성 개선

→ 팀수준 (KPI or Business Capability) 수준의 분리

새로운 언어와 기술의 도입

→ 컨테이너 기술

시스템 작동 중단 시간 감소

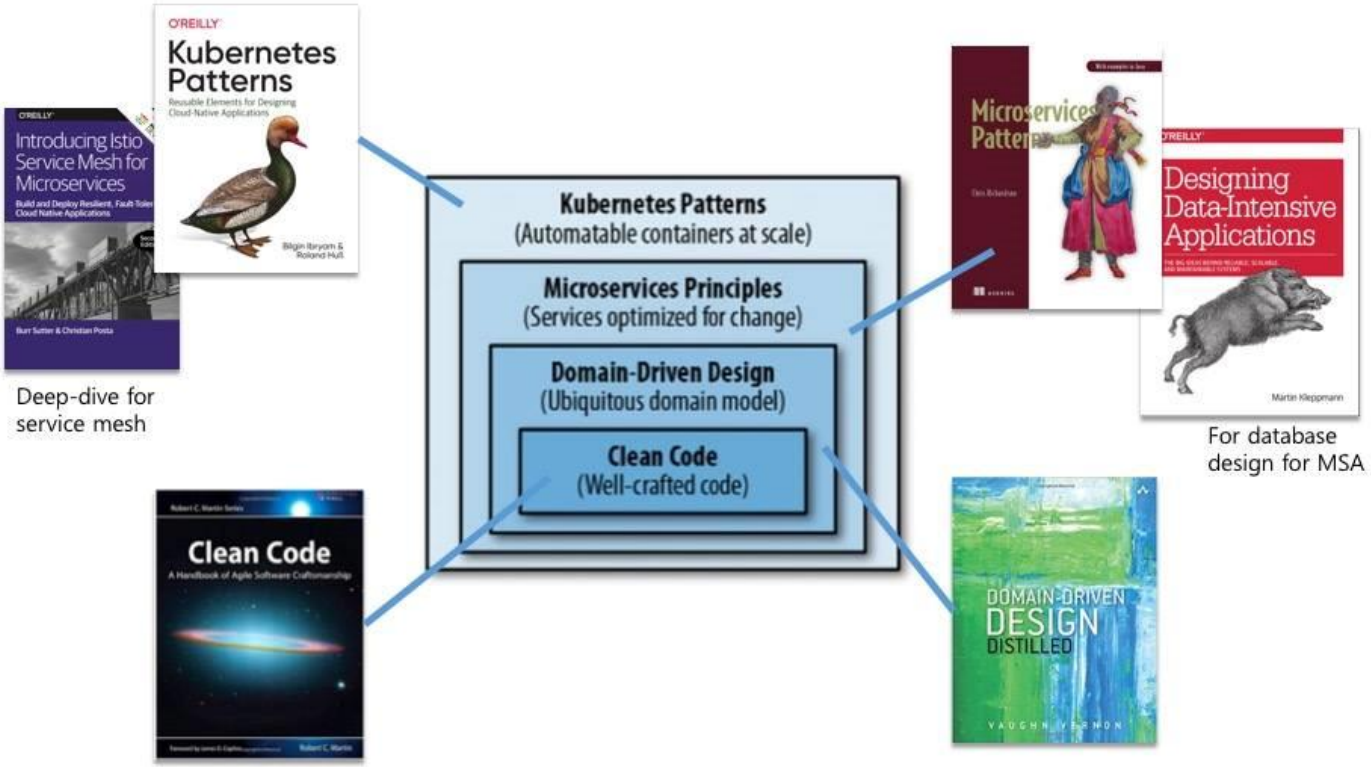
- → Core/Supporting 의 분리
- → 무정지 재배포
- → 컨테이너 기술

Table of content

Microservice and
Event-storming-Based
DevOps Project

UEngine

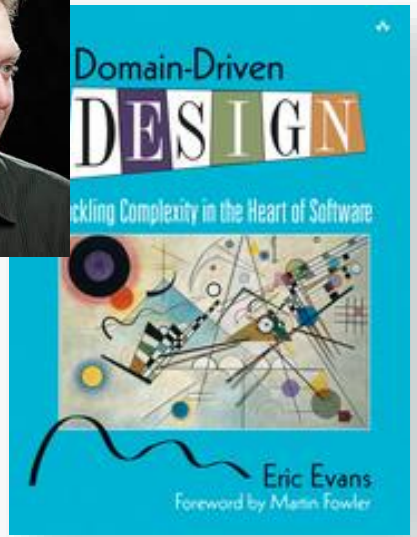
Learning Path to Cloud Native



DDD Patterns

Domain-Driven Design & MSA

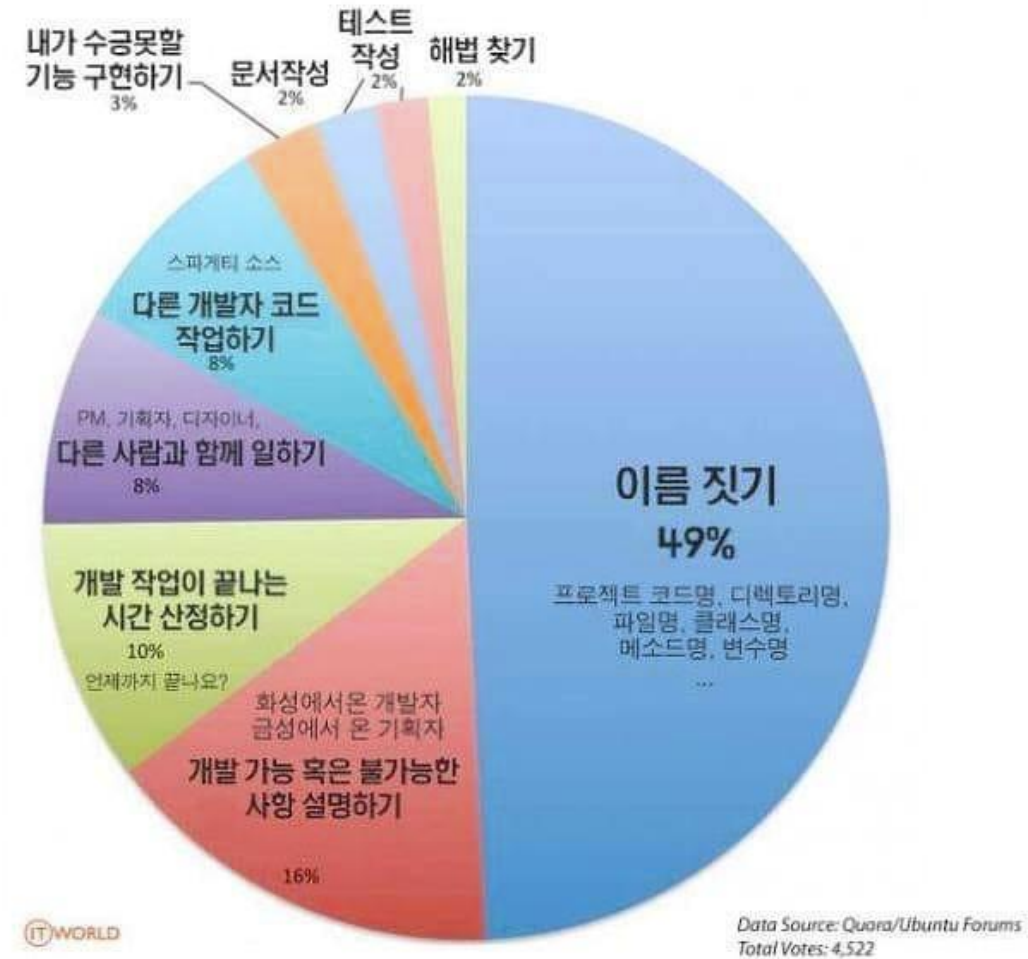
DDD for MSA



- **Bounded Context 와 Ubiquitous Language**
  - 어떤 단위로 마이크로 서비스를 쪼개면 좋은가?
- **Context Mapping**
  - 서비스를 어떻게 결합할 것인가?
- **Domain Events**
  - 어떤 비즈니스 이벤트에 의하여 마이크로 서비스들이 상호 반응하는가?

The key to controlling complexity is a good domain model
– Martin Fowler

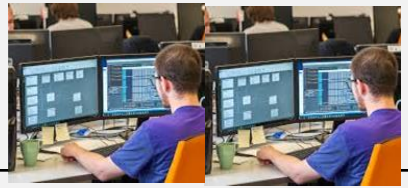
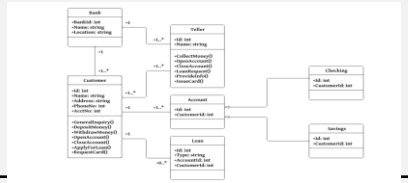
프로그래머가 가장 힘들어하는 일은?



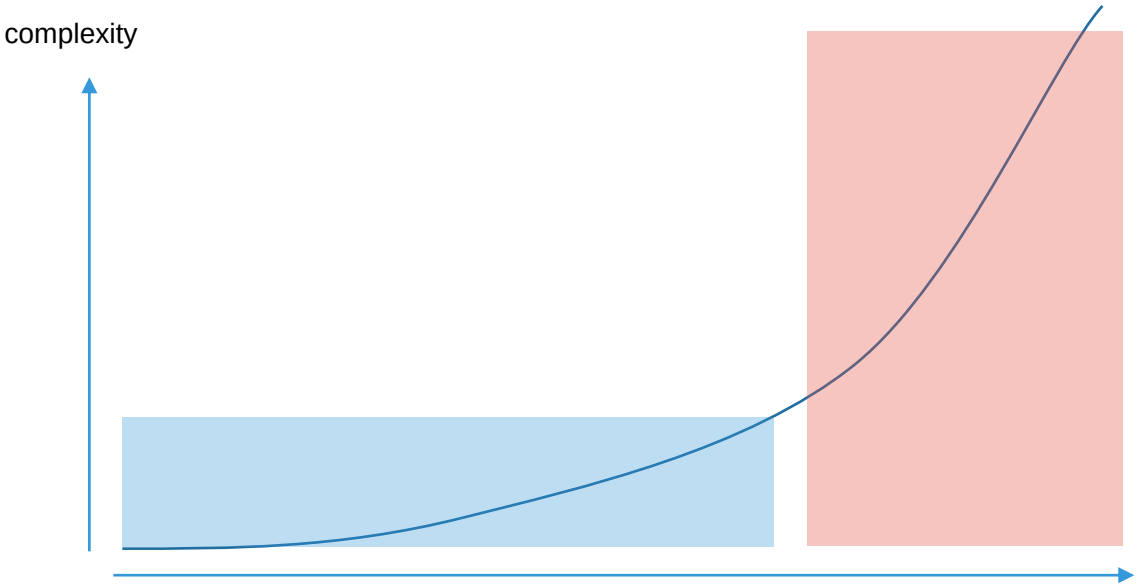
Bounded Context
(한정된 맥락)

Ubiquitous Language
(도메인 언어)

Bounded Context and Ubiquitous Language

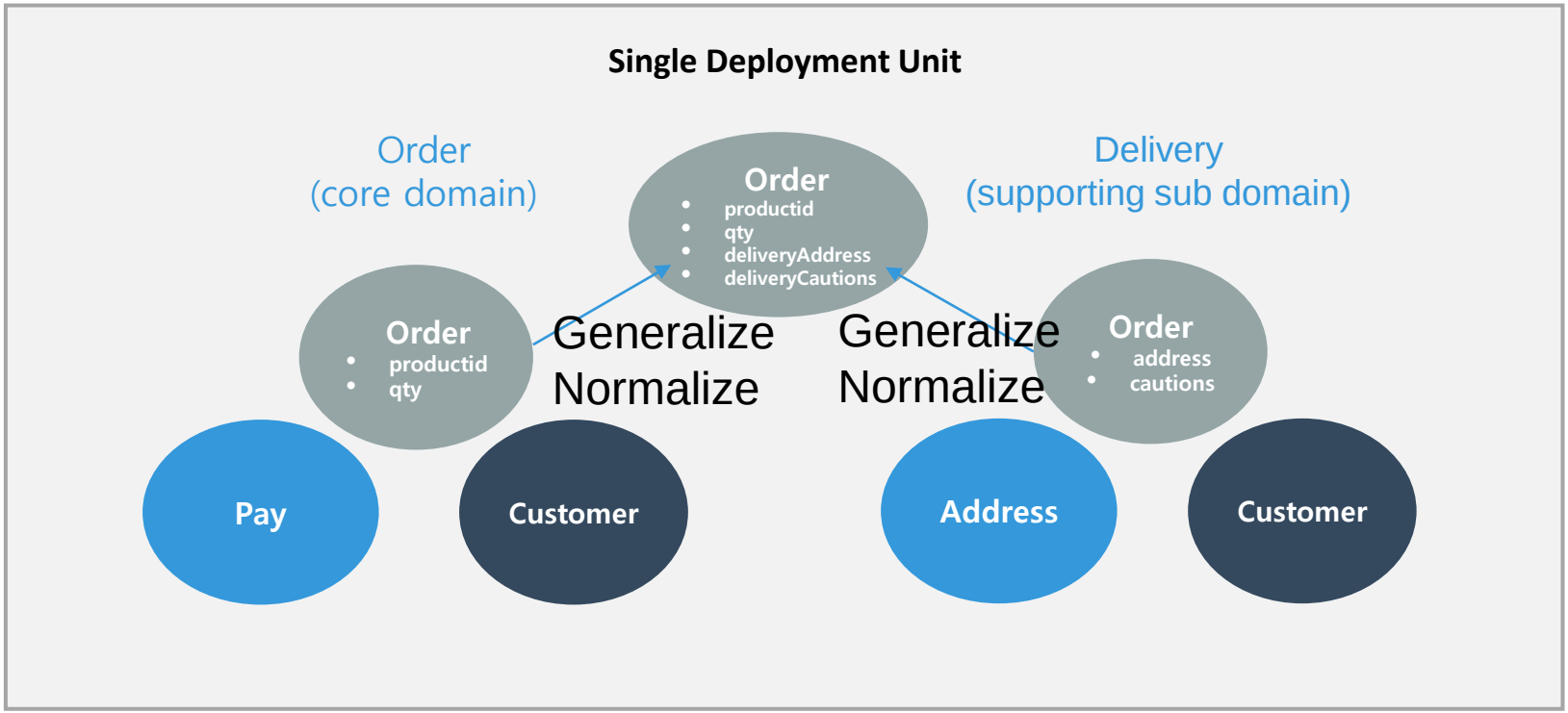
	SW	CONSTRUCTION
A Project		
An Architecture		
A Developer		

Cognitive Cost



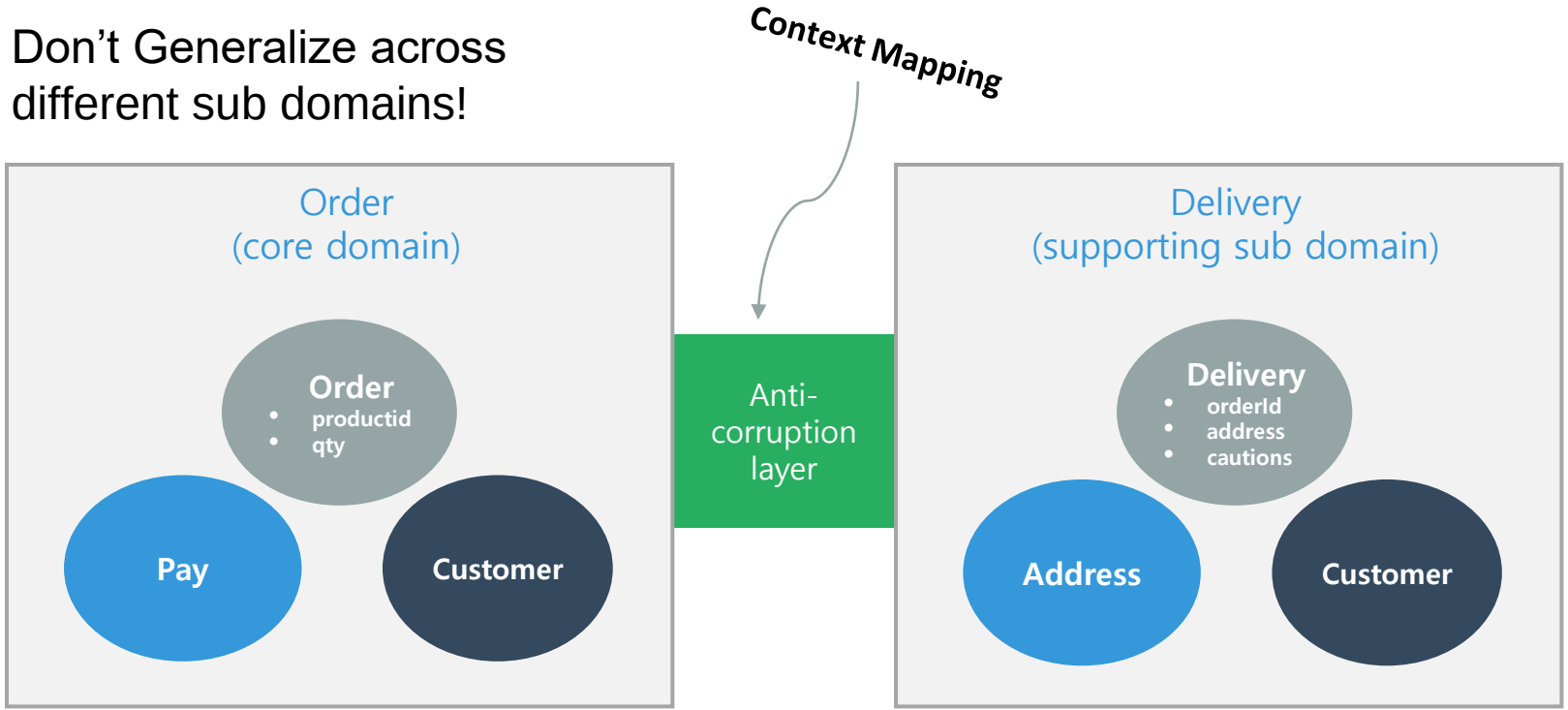
소스코드의 영역을 뜻함

In Monolith : Single Model Fits All



In MSA : Multiple Models

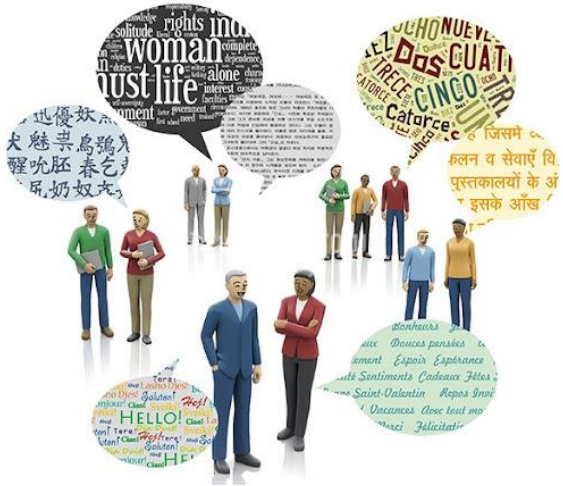
Don’t Generalize across different sub domains!



Comparison of Total Communication Cost

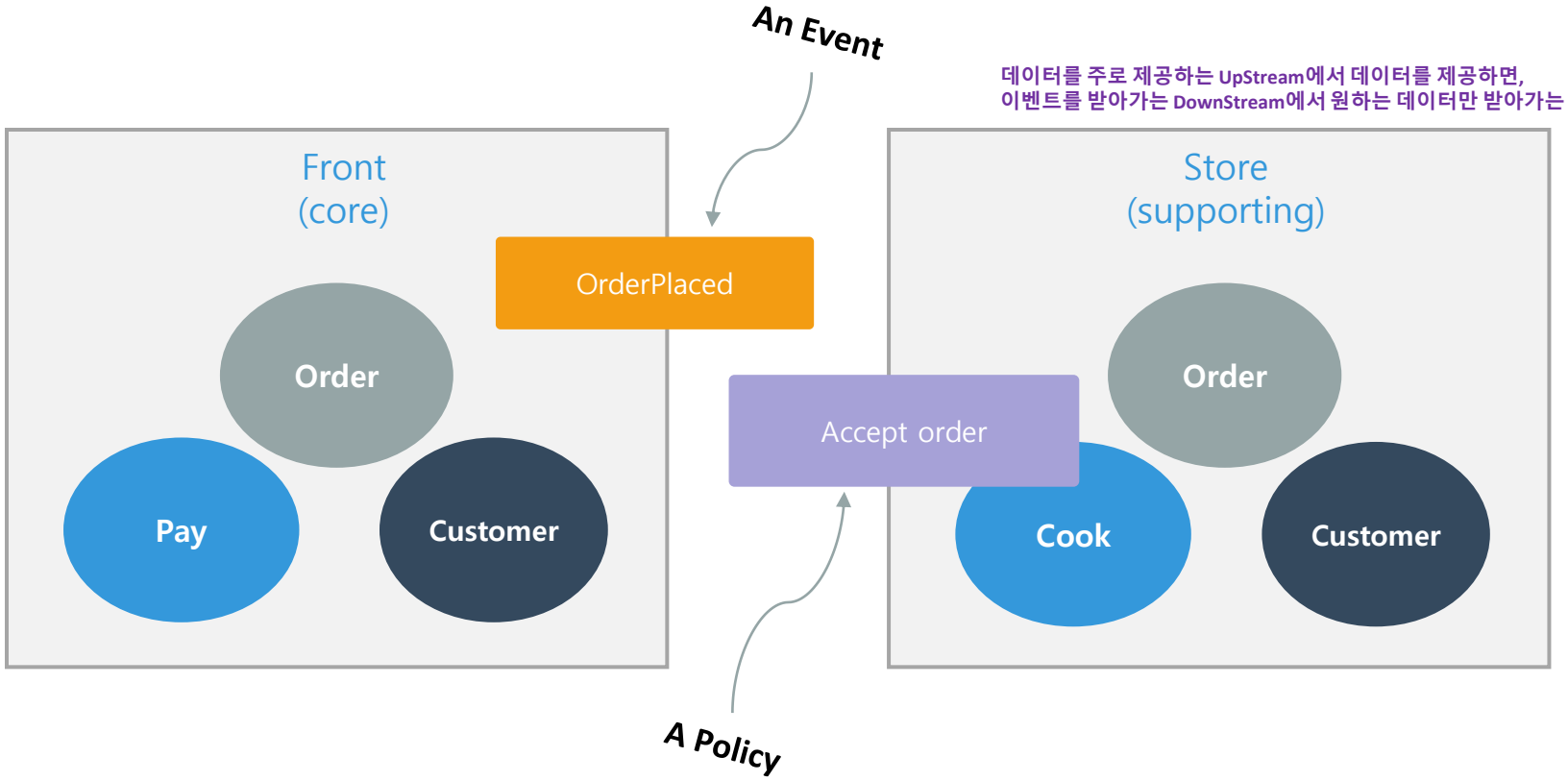
- Generalization and Normalization of single language

- Translators between multiple language
- Anti-corruption between Multiple languages



DDD 관련 추천 도서
- DDD Distilled
- Leaning DDD

In Microservices :



마이크로 서비스간의 서열과 역학관계

" 무엇을 우선적으로 챙길 것인가? "

1순위 : Core Domain

- 버릴 수 없는. 이 기능이 제공되지 않으면 회사가 망하는

예) 쇼핑몰 시스템에서 주문, 카탈로그 서비스 등

2순위 : Supporting Domain

- 기업의 핵심 경쟁력이 아닌, 직접 운영해도 좋지만 상황에 따라 아웃소싱 가능한
- 시스템 리소스가 모자라면 외부서비스를 빌려쓰는 것을 고려할만한

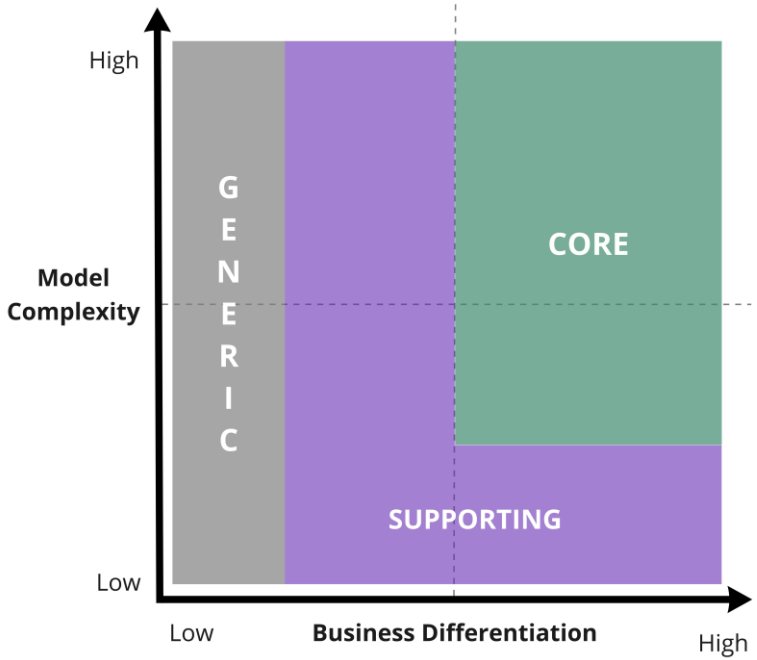
예) 재고관리, 배송, 회원관리 서비스 등

3순위 : General Domain

- SaaS 등을 사용하는게 더 저렴한, 기업 경쟁력과는 완전 무관한

예) 결재, 빌링 서비스 등

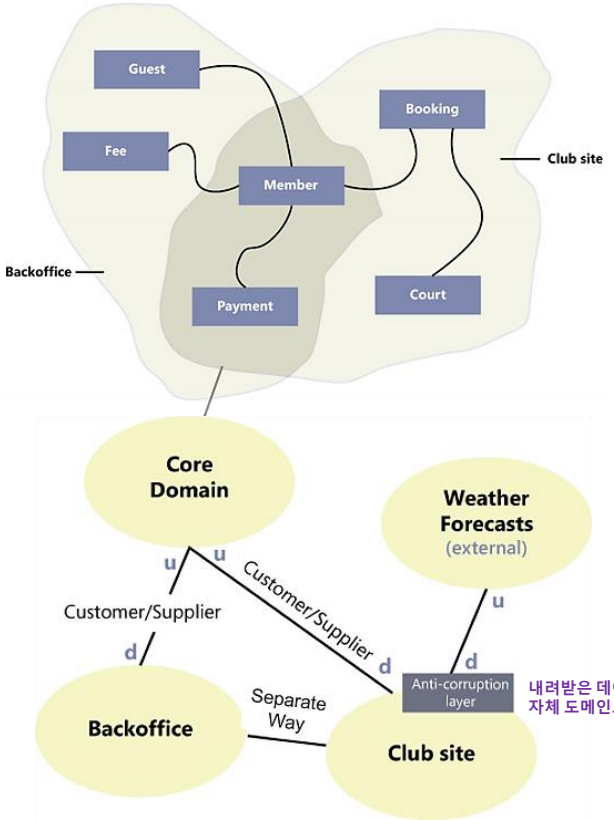
Diff. between Subdomain Types



서브도메인 유형	경쟁적 우위	복잡도	변화성	구현	문제
Core	Yes	High	High	In-house	재미있는
Generic	No	High	Low	구매/적용	해결된
Supporting	No	Low	Low	In-house/아웃소싱	뻔한

Ref : Relation types between services

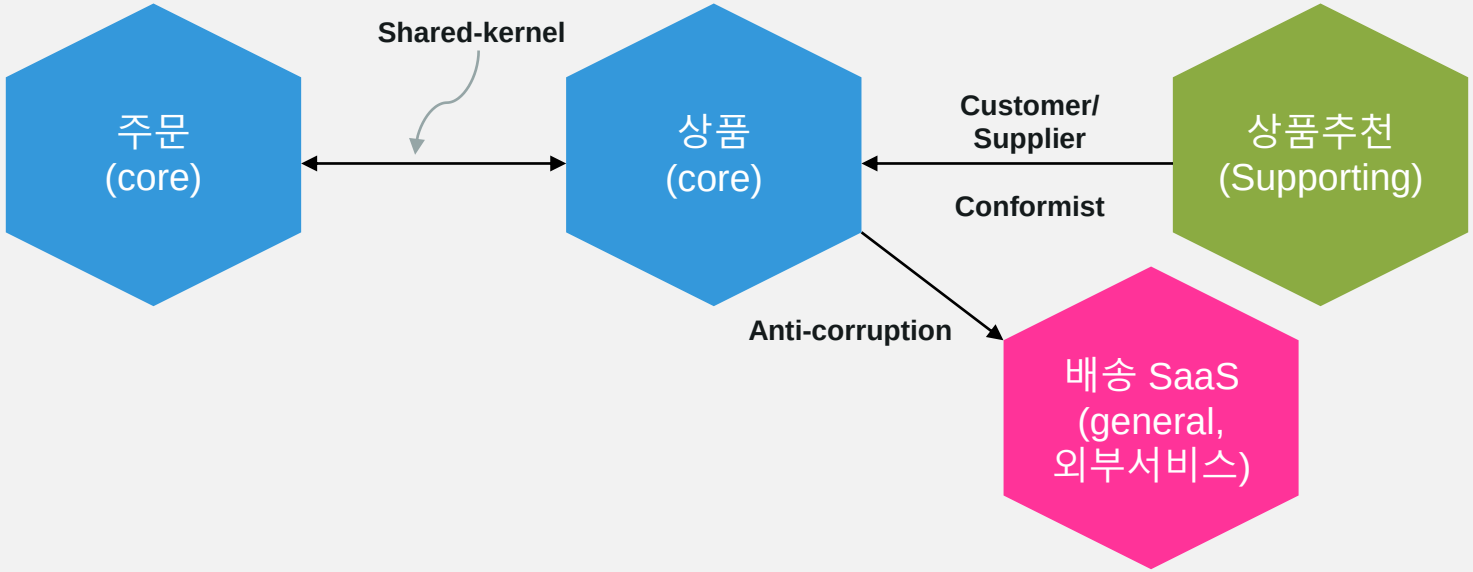
- Shared Kernel
  - Designate a subset of the domain model that the two teams agree to share.
- Customer/Supplier
  - Make the downstream team play the customer role to the upstream team.
- Conformist
  - Eliminate the complexity of translation between bounded contexts by slavishly adhering to the model of the upstream team.
- Separate Ways
  - Declare a bounded context to have no connection to the others at all. The feat ures can still be organized in middleware or the UI layer.
- Open Host Service
  - Open the protocol so that all who need to integrate with you can use it. Other t eams are forced to learn the particular dialect used by the host team. In some situations, using a well-known published language as the interchange model c an reduce coupling and ease understanding.



Eric Evans, Domain-Driven Design, 2003.

마이크로 서비스간의 서열과 역학관계

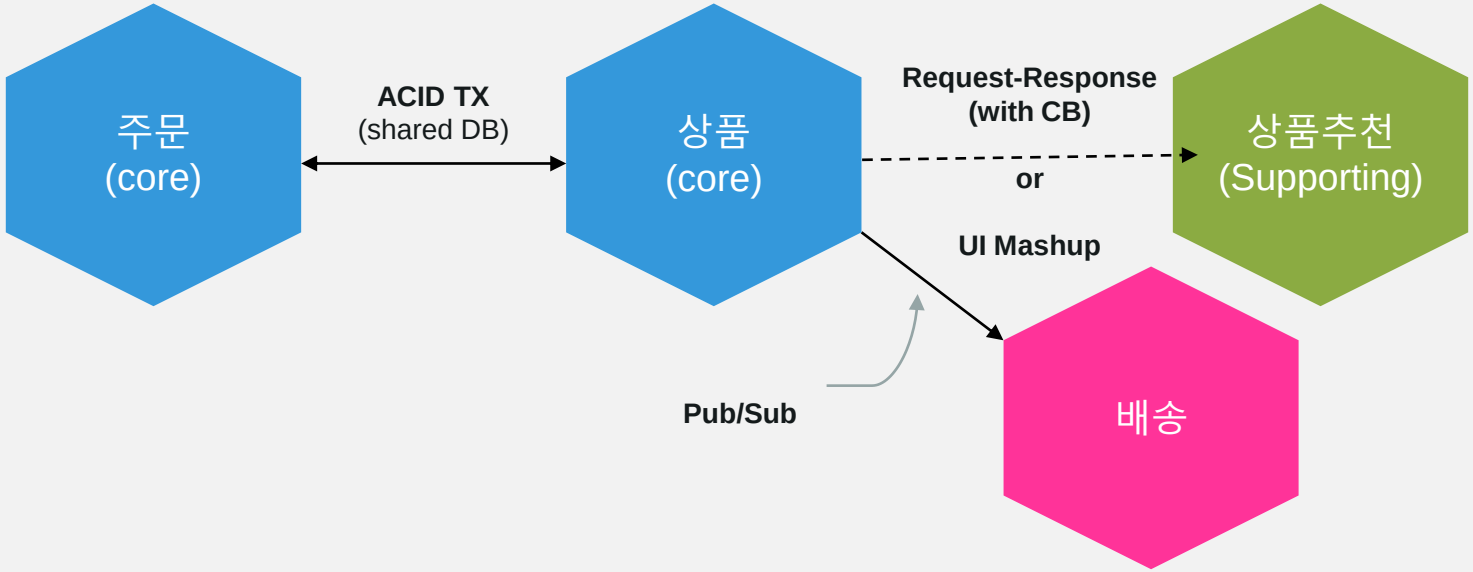
" 어느 마이크로 서비스의 인터페이스를 더 중요하게 관리할 것인가? "



Core Domain 간 (높은 서열끼리) 에는 Shared-kernel 도 허용가능하다.
하지만, 중요도가 낮은 서비스를 위해 높은 서열의 서비스가 인터페이스를 맞추는 경우는 없을 것이다.

마이크로 서비스간의 서열과 역학관계

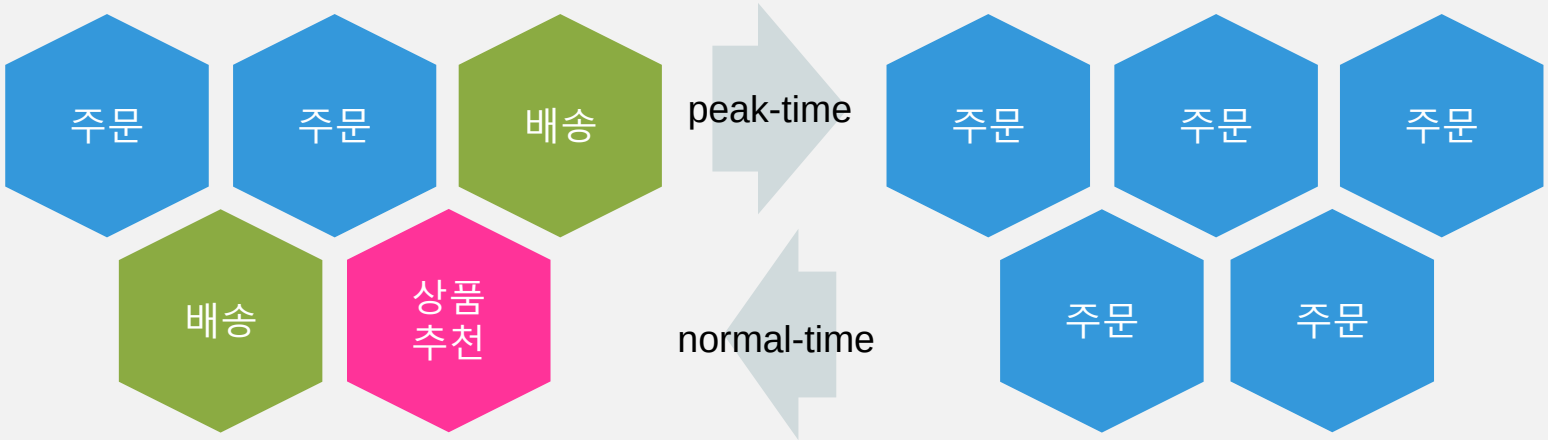
" 마이크로서비스간 트랜잭션의 묶음을 어떻게 할 것인가? "



배송기사가 없다고 주문을 안받을 것인가? 상품 추천이 안된다고 주문버튼을 안보여줄 것인가?
Core Domain 간에는 강결합이 요구되는 경우가 생길 수 있다.
하지만 우선순위가 떨어지는 비즈니스 기능을 위해 강한 트랜잭션을 연결할 이유는 하등에 없다.

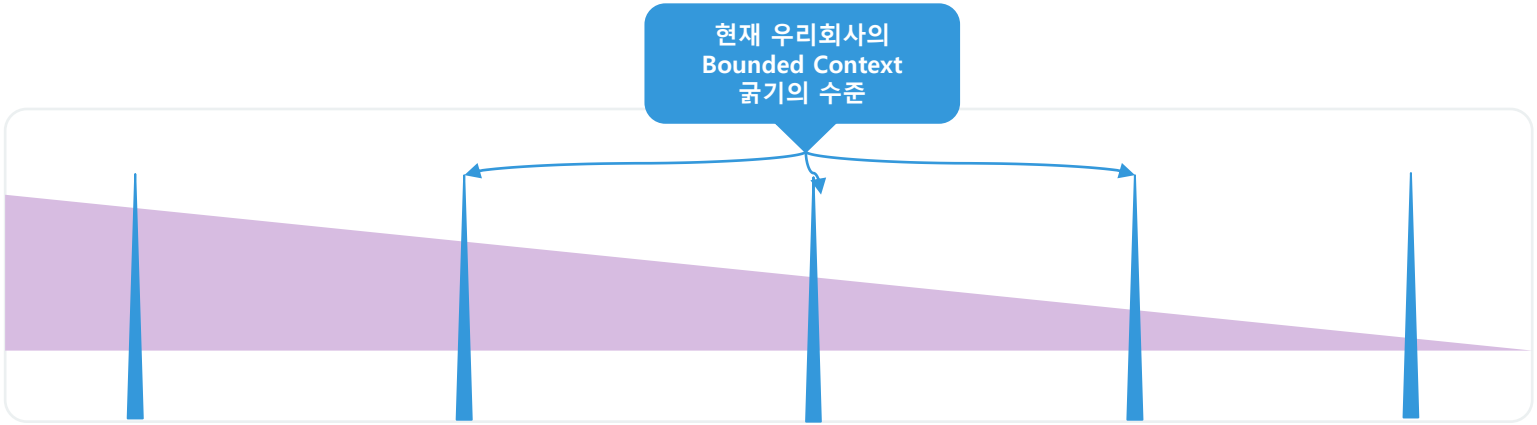
마이크로 서비스간의 서열과 역학관계

“ 손님이 많이 와서 집이 좁아졌다.. 무엇을 우선적으로 줄일 것인가? ”



배송서비스는 야간에 올라와서 후에 처리되어도 된다.
주문이 몰려드는 시간에 주문을 못 받으면 배송은 의미가 없다

분리수준 - 어느 굵기로 쪼갤 것인가?



Monolith

Core/Supporting

간섭이 가장 높은 부분을
경계로 한. 시급한.

구현이 용이한 단위
- 점진적 MSA 전환이 용이한 단위

Business Capability

비즈니스적
목표에 따라 분리.

– 팀단위 KPI와 연결하
기 좋은
- 자치적 동기부여.

BC Original

커뮤니케이션이 원
활해지는.

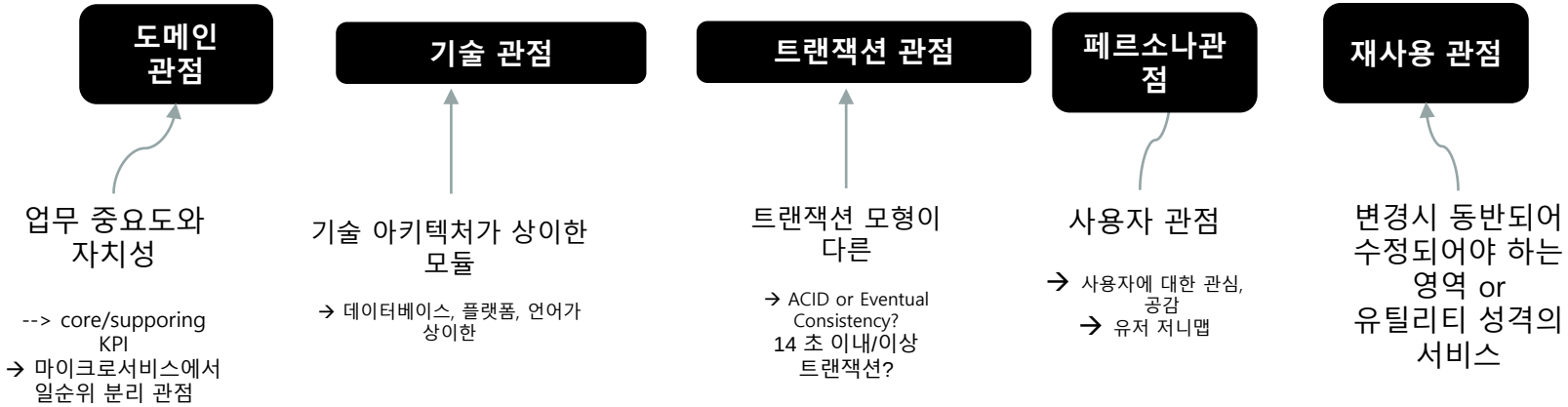
– Ubiquitous Language
유지

Aggregate

더이상 쪼갤 수 없는.

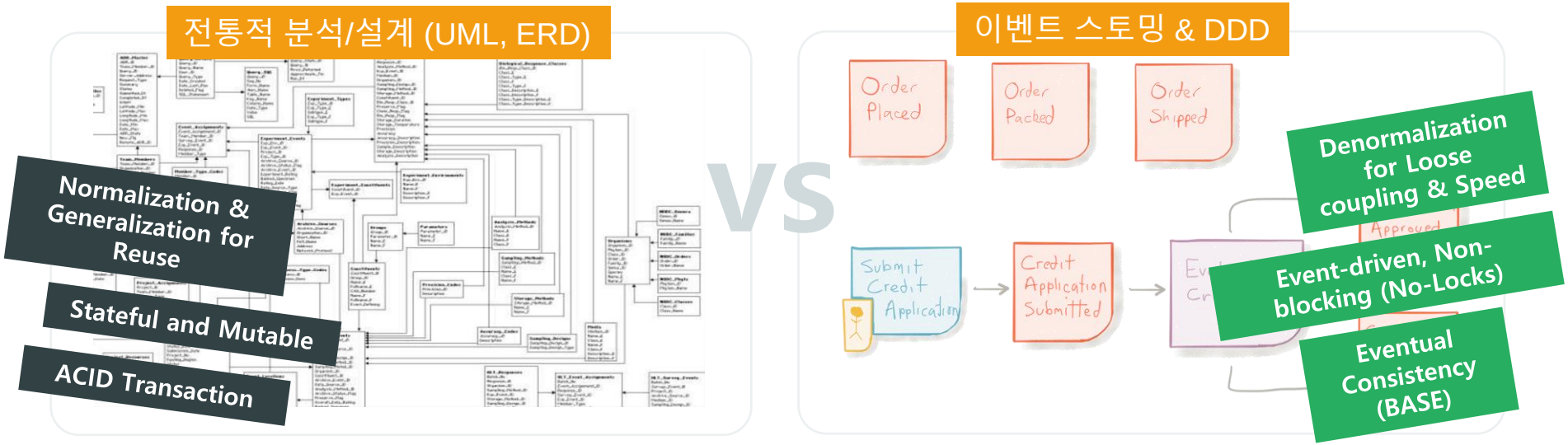
– ACID 가 준수되어야 하는
원자적 단위
- Serverless, FaaS

분리관점 - 어떤 관점으로 쪼갤 것인가?



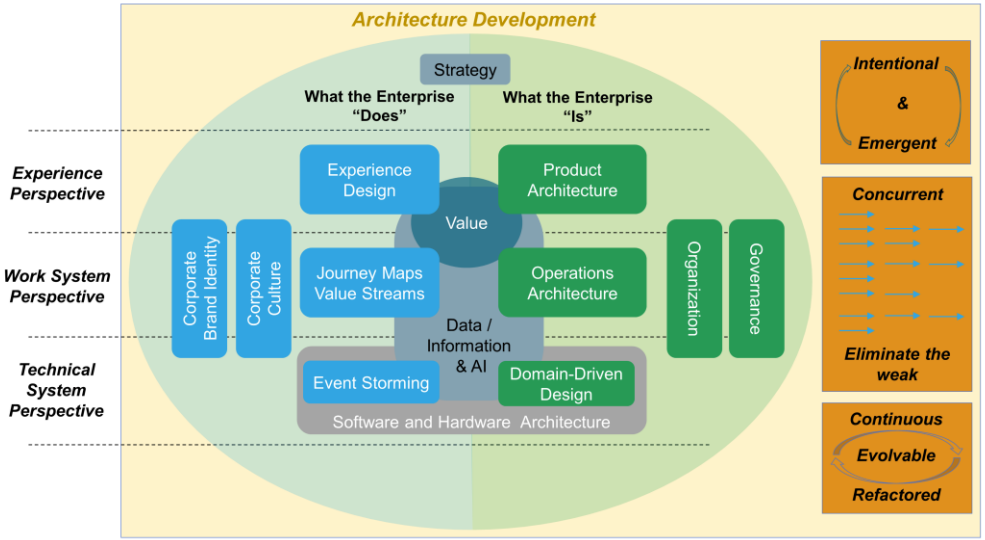
Event Storming : DDD 를 쉽게 하는 방법

- **이벤트스토밍은** 시스템에서 발생하는 **이벤트를 중심 (Event-First) 으로** 분석하는 기법으로 특히 Non-blocking, Event-driven 한 **MSA 기반 시스템을 분석에서 개발까지 필요한 도메인에 대한 탁월하게 빠른 이해를 도모**하는데 유리하다.
- 기존의 유즈케이스나 클래스 다이어그래밍 방식과 다르게 별다른 **사전 훈련된 지식과 도구 없이 진행할 수 있다**.
- 진행과정은 참여자 워크숍 방식의 방법론으로 결과는 **스티키 노트를 벽에 붙힌 것으로 결과가 남으며, 오랜지색 스티키 노트들의 연결로 비즈니스 프로세스가 도출**되며 이들을 이후 BPMN과 UML 등으로 정재하여 전환할 수 있다.



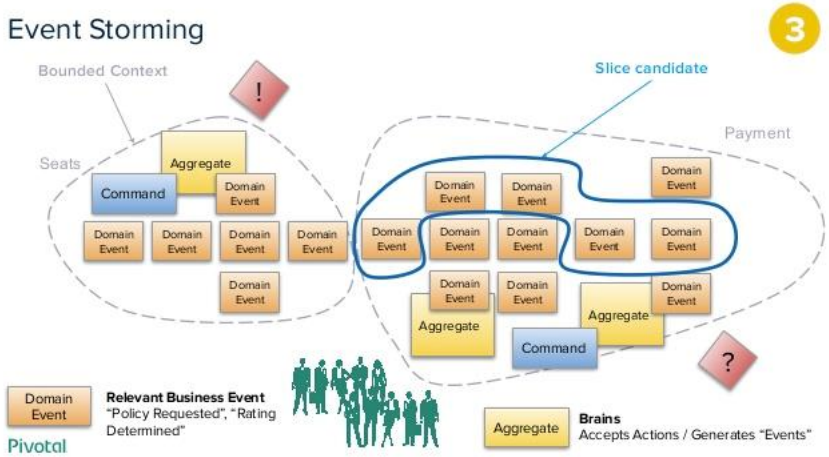
Event Storming 의 위상

The Open Group 의 OAA 의 빌딩블록



https://pubs.opengroup.org/architecture/o-aa-standard/#Introduction

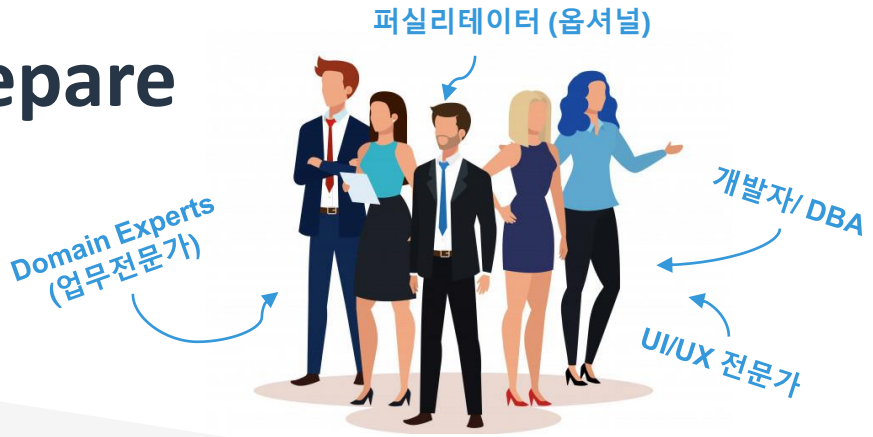
Pivotal 의 AppTX 방법론



https://www.slideshare.net/Pivotal/pivotals-secret-sauce

IBM 의 Garage 방법론… etc

Event Storming : Prepare

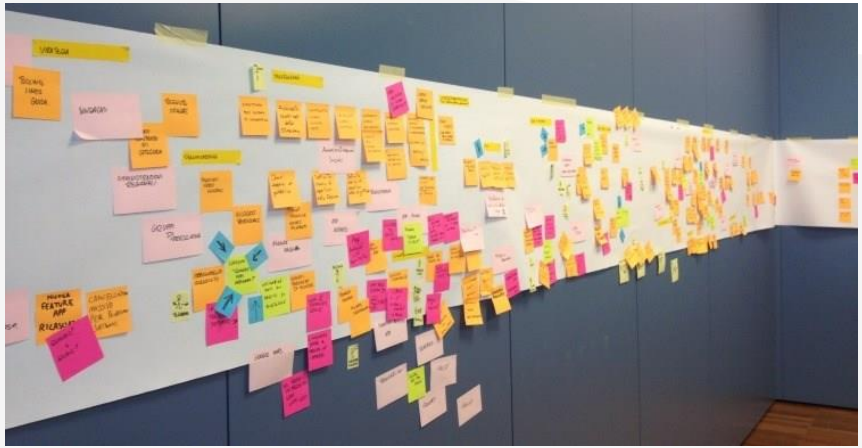


People

- 모든 팀 및 프로세스, UI, DB, 아키텍처를 책임질 팀당 2명 이상 구성
- 이벤트스토밍 전문가 퍼실리테이터가 초기에는 필요할 수 있음

Tools

- 큰 종이 시트 및 종이 시트를 여러 장 붙일 수 있는 충분한 벽이 있는 넓은 공간
- 여러 종류의 색깔 스티커, 검은색 펜, 검은색 or 파란색 테이프
- 서서 하는 방식으로 의자 필요 없음.



Type of stickers

Domain Event (Orange)

과거형(P.P)으로 작성

발생한 사실, 결과, Output

도메인 전문가가 정의
이벤트 퍼블리싱

사용자, 페르소나, 스테이크 홀더

유저 인터페이스를 통해 데이터를 소비하고 명령을 실행하여 시스템과 상호 작용

Definition

정의

도메인에 대한 용어 등의 설명,기술

Command (Sky Blue)

의사결정, Input, API, UI 버튼

현재형으로 작성,
행동, 결정 등의 값들에 대한 UI 혹은 API

Aggregate (Yellow)

구현체 덩어리, 시스템, 모듈

비즈니스 로직 처리의 도메인 객체 덩어리. 서로 연결된 하나 이상의 엔터티 및 value objects의 집합체

Read Model (Green)

의사결정에 필요한 자료, View, UI

행위와 결정을 하기 위하여
유저가 참고하는 데이터, 데이터 프로젝션이 필요 : CQRS 등으로 수집

External System (Pink)

외부 시스템

시스템 호출을 암시 (REST)

Policy (Lilac)

정책, 반응(Reaction)

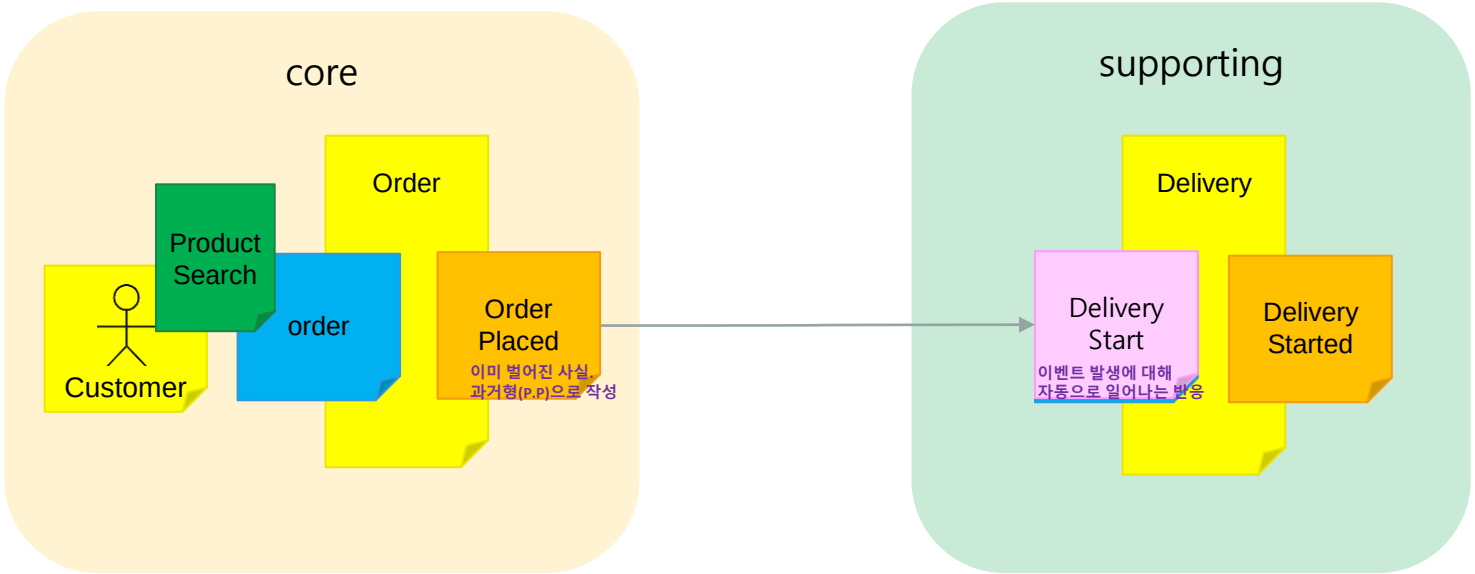
이벤트에 대한 반응
(서브스크라이브)
비즈니스 룰 엔진 등

Comment Or Question (Purple)

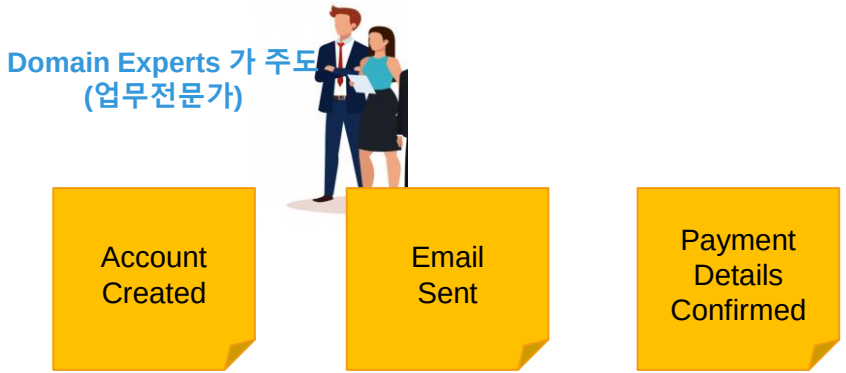
의견 또는 질문

추기적인 내용 입력,
예측되는 Risk

Examples


core
Order
Product Search
order
Order Placed
이미 벌어진 사실, 과거형(P.P)으로 작성
Customer
supporting
Delivery
Delivery Start
이벤트 발생에 대해 자동으로 일어나는 반응
Delivery Started

Firstly, Event Discovery

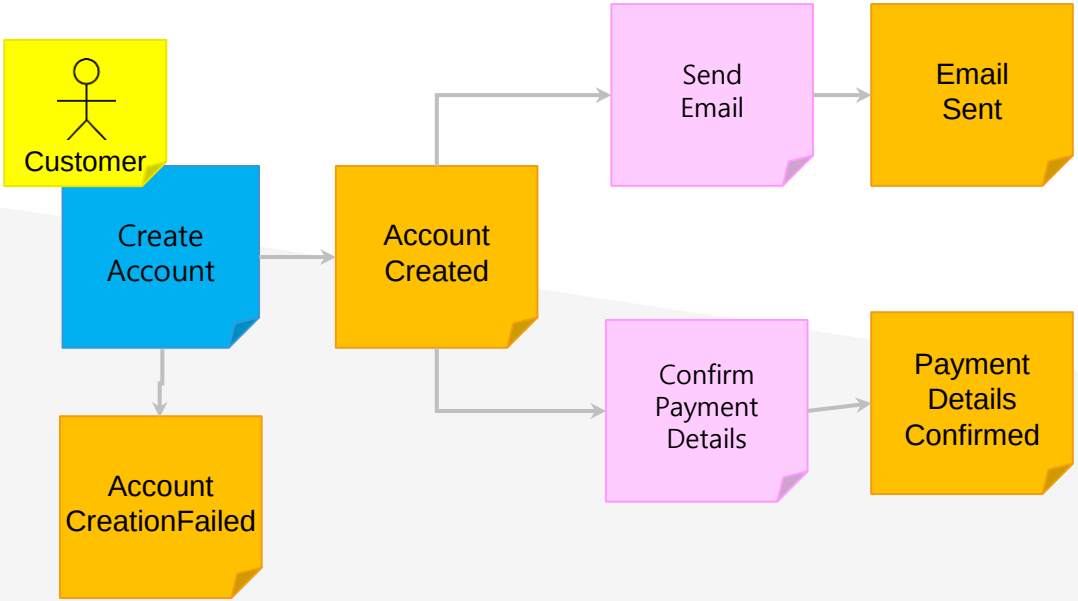


- 오렌지(주황색) 스티커 사용
- 각 도메인 전문가들이 개별 도메인 이벤트 목록 작성
- 이벤트는 도메인 전문가와 비즈니스 관계자 서로 이해할 수 있는 의미 있는 방식(유비쿼터스 랭귀지)으로 표현하고 동사의 과거형 (p.p.)으로 표현한다.
- 시작 및 종료 이벤트를 식별하고 스티커를 붙일 벽의 시작과 끝의 타임라인에 배치
- 이벤트를 페르소나와 관련시키는 방법에 대해 논의
- 중복된 이벤트를 발견하면 중복된 이벤트를 벽에서 제거
- 불분명한 경우 다른 색상의 스티커 메모를 사용하여 질문이나 의견을 추가(빨간색 스티커)
- 이벤트에 대해서 동사를 과거 시제로 입력하고 다른 이벤트와 명확하게 구분되는 용어를 사용

Command, Policy, Actor 도출

Domain Experts + UI/UX 가 주도
(업무전문가)

- 하늘색 or 라일락 색의 스티커 사용
- Command 는 사용자의 의사결정에 의하여 (UI) 결과이벤트에 이르게 하는 Input
- Command 는 어떠한 상태의 변화를 일으키는 서비스*
- Policy는 이벤트가 발생한 후 발생하는 반응형 논리 (Input)
- Policy는 결과로서 Event 를 트리거 할 수 있고, 다른 Command를 호출 할 수도 있음
- Policy는 시스템에 의하여 자동화 될 수 있다.



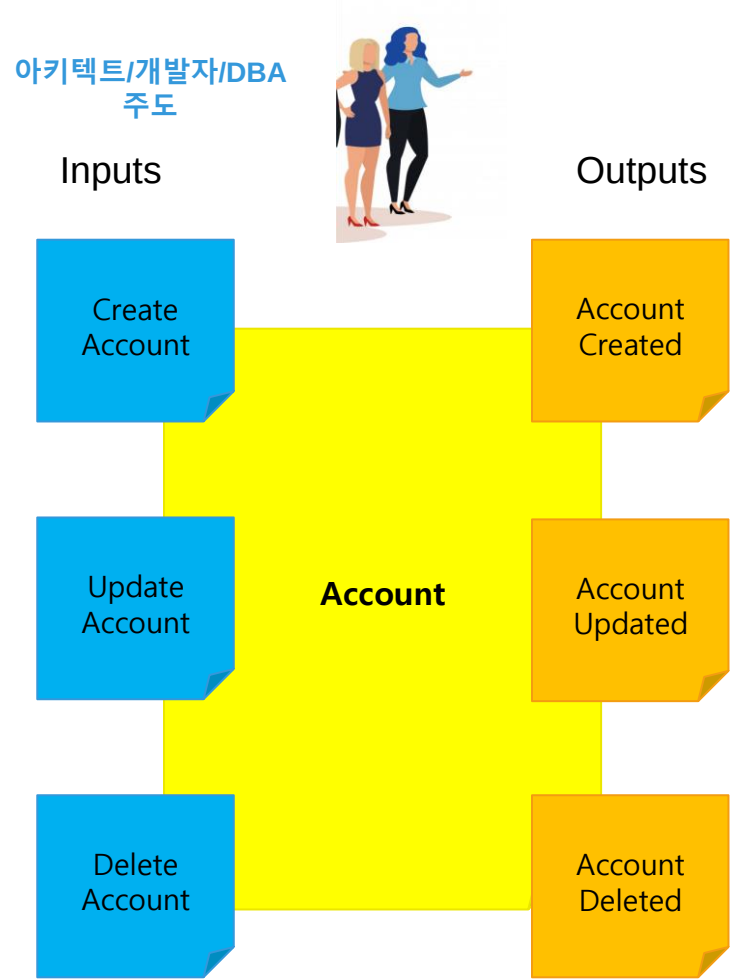
*"Whenever a **new user account is created** we will **send her an acknowledgement by email.**"*

* Command 라는 용어는 CQRS 에서 유래했으며,
쓰기서비스인 Command 와 읽기행위인 Query는 구분됨.

Aggregate 도출

- 노란색 스티커 사용
- 같은 Entity를 사용하는 연관 있는 도메인 이벤트들의 집합
- 관련 데이터 (Entity 및 value objects)뿐만 아니라 해당 Aggregates의 Life Cycle에 의해 연결된 작업(Command)으로 구성

도메인 이벤트가 발생하는 데는, 어떠한 도메인 객체의 변화가 발생했기 때문이다.
하나의 ACID 한 트랜잭션에 묶여 변화되어야 할 객체의 묶음을 도출하고, 그것들을 커맨드, 이벤트와 함께 묶는다.



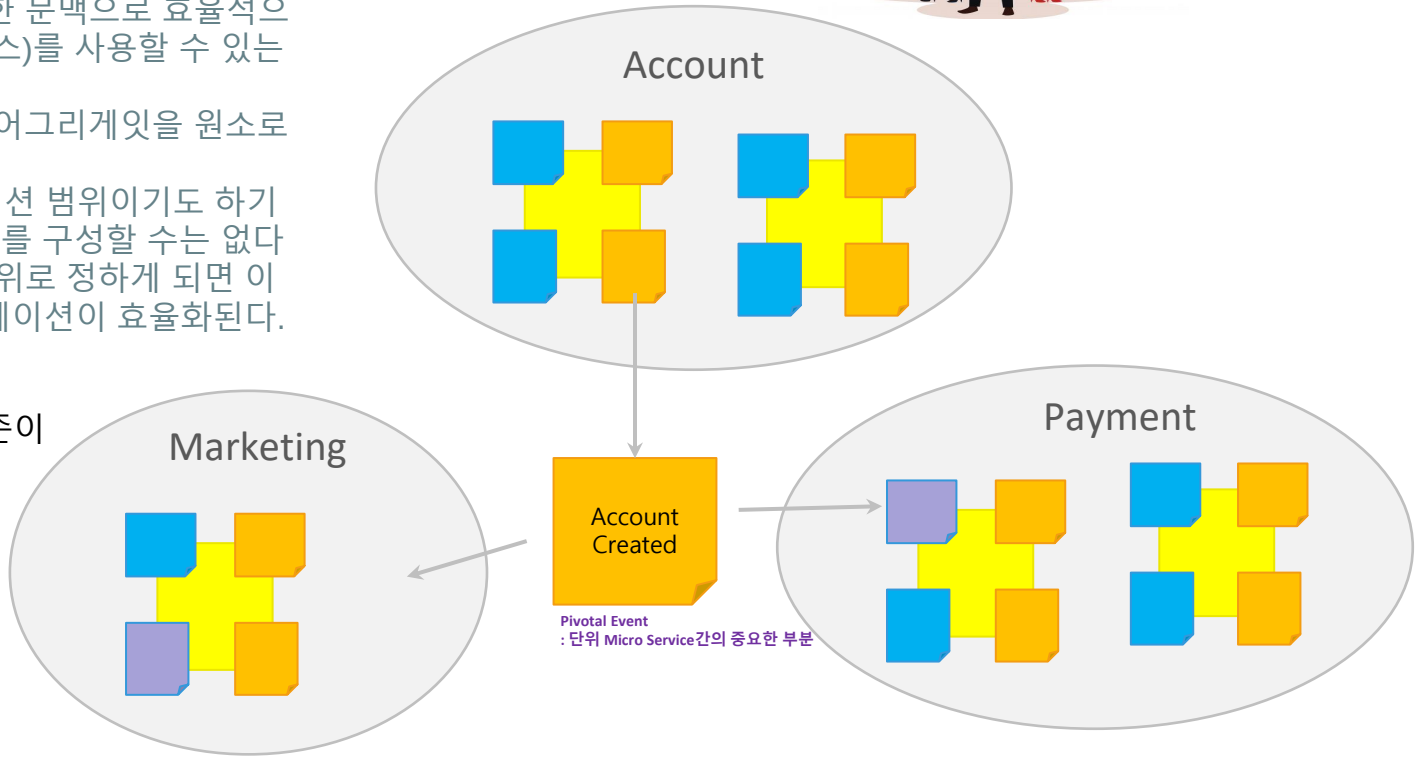
Bounded Contexts 도출

- Bounded Context 는 동일한 문맥으로 효율적으로 업무 용어 (도메인 클래스)를 사용할 수 있는 범위를 뜻한다.
- 하나의 BC 는 하나이상의 어그리게잇을 원소로 구성될 수 있다.
- 어그리게잇은 ACID 트랜잭션 범위이기도 하기 때문에 이를 더 쪼개서 BC 를 구성할 수는 없다
- BC를 Microservice 구성단위로 정하게 되면 이를 담당한 팀 내의 커뮤니케이션이 효율화된다.

찾는 방식은 여러가지 기준이 있다

- Domain Boundary
- Transaction Boundary
- Technical Stack

...



Source: https://blog.redelastic.com/corporate-arts-crafts-modelling-reactive-systems-with-event-storming-73c6236f5dd7, Kevin webber

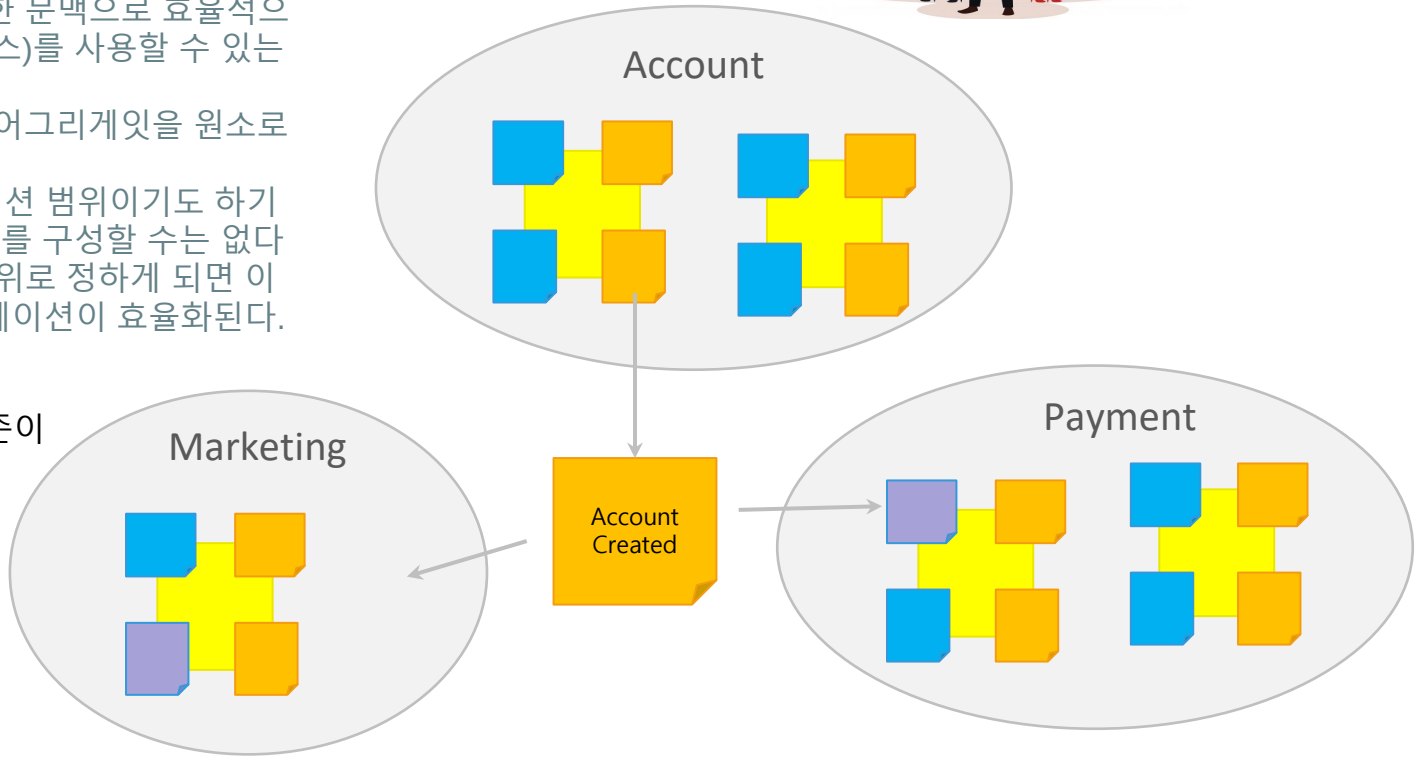
Bounded Contexts 도출

- Bounded Context 는 동일한 문맥으로 효율적으로 업무 용어 (도메인 클래스)를 사용할 수 있는 범위를 뜻한다.
- 하나의 BC 는 하나이상의 어그리게잇을 원소로 구성될 수 있다.
- 어그리게잇은 ACID 트랜잭션 범위이기도 하기 때문에 이를 더 쪼개서 BC 를 구성할 수는 없다
- BC를 Microservice 구성단위로 정하게 되면 이를 담당한 팀 내의 커뮤니케이션이 효율화된다.

찾는 방식은 여러가지 기준이 있다

- Domain Boundary
- Transaction Boundary
- Technical Stack

...

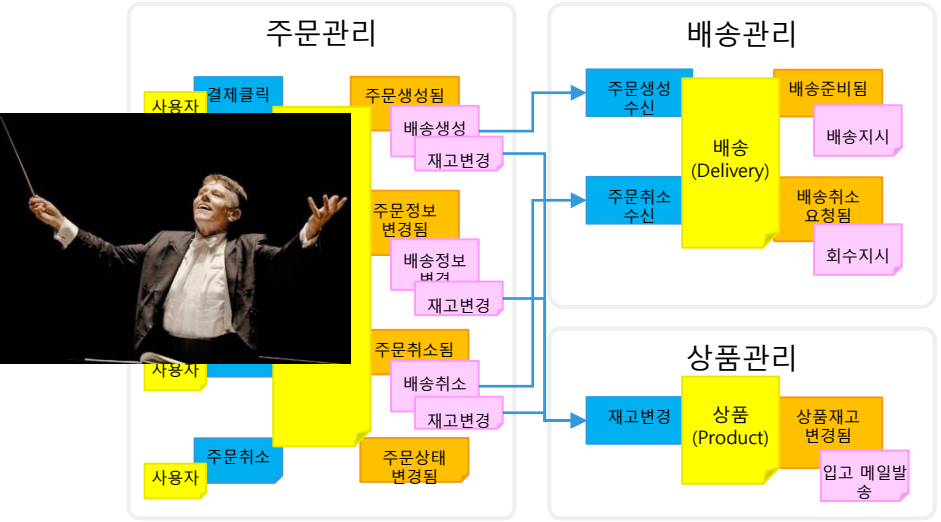


Source: https://blog.redelastic.com/corporate-arts-crafts-modelling-reactive-systems-with-event-storming-73c6236f5dd7, Kevin webber

Context Mapping

라이락 스티커 (Policy) 를 어디에 둘 것인가? - Orchestration or Choreography? Or Mediation?

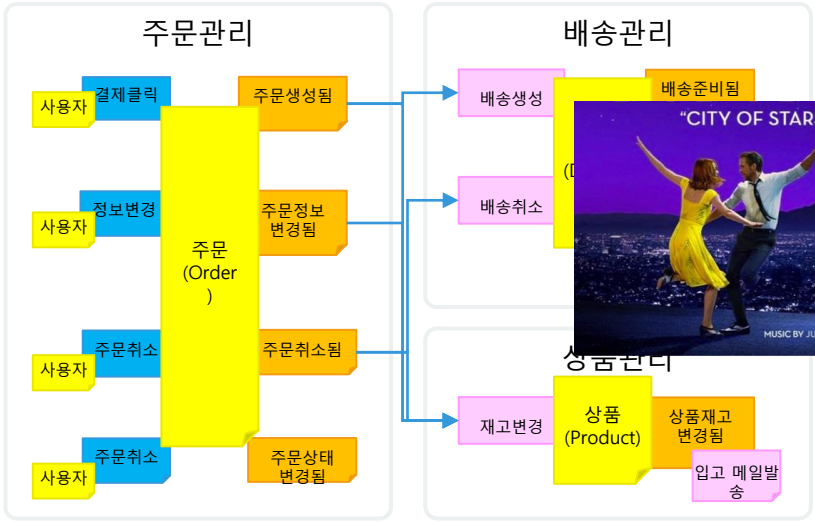
Orchestration



Originator should know how to handle the policy

- Coupling is High

Choreography



More autonomy to handle the policy

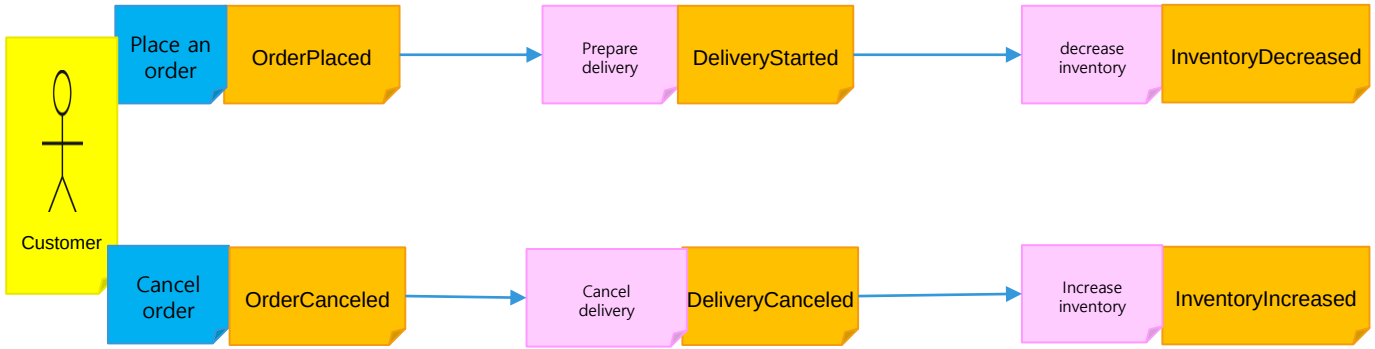
- Low-Coupling
- Easy to add new policies

Lab Time – 12 번가 예제 – 최초 Event Discovery

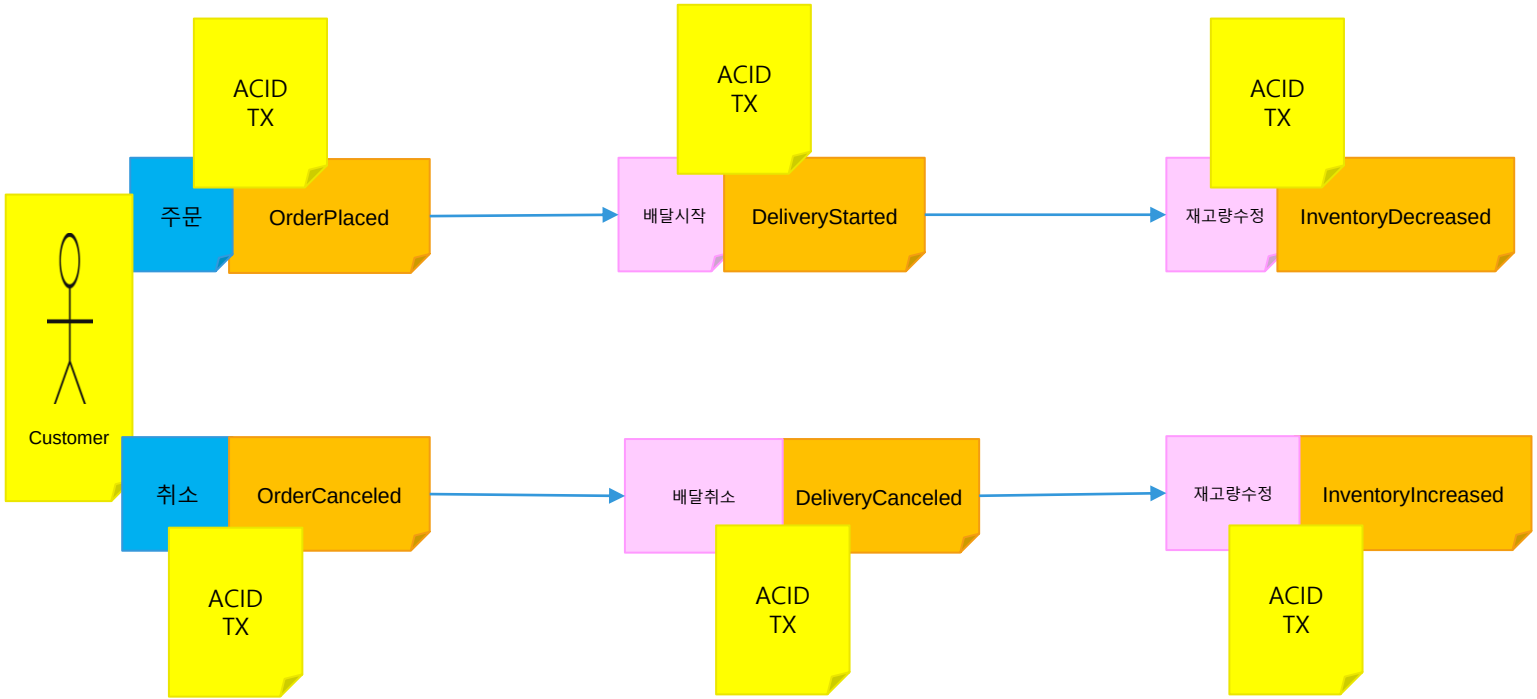
- Customers search products and clicks the order button to **placed an order**.
- When an **order is placed**, the delivery team prepares for delivery of the product and **the delivery will start**.
- When delivery has been started, the product **inventory is decreased**.
- Customers can **cancel their orders.**
- When an order has been canceled, **delivery belongs to the order must be canceled** as well.
- When delivery has been canceled, the product **inventory is increased**.

Lab Time – Command, Policy, Actor

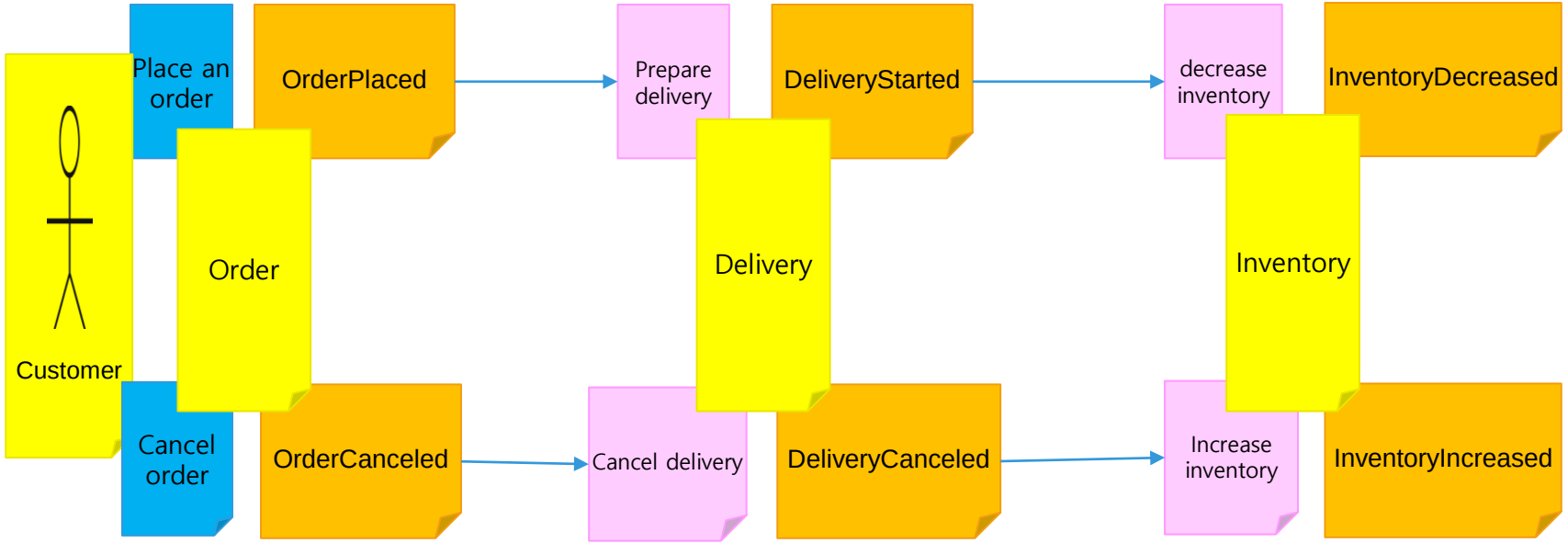
- **Customers** search products and clicks the order button to **placed an order**.
- When an **order is placed**, the **delivery team prepares for delivery** of the product and **the delivery will start**.
- **When delivery has been started, the product inventory is decreased.**
- **Customers** can **cancel their orders.**
- When an order has been canceled, **delivery belongs to the order must be canceled** as well.
- When delivery has been canceled, the product **inventory is increased**.



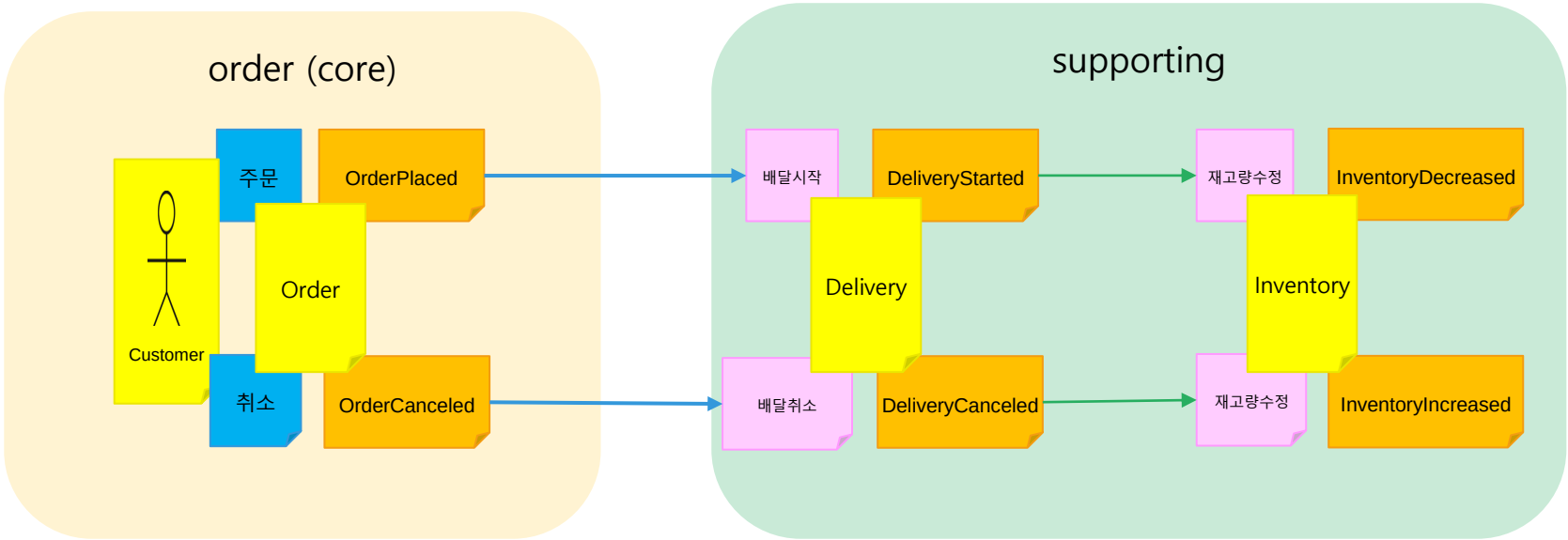
Lab Time – Aggregate



Lab Time – Aggregate (2)



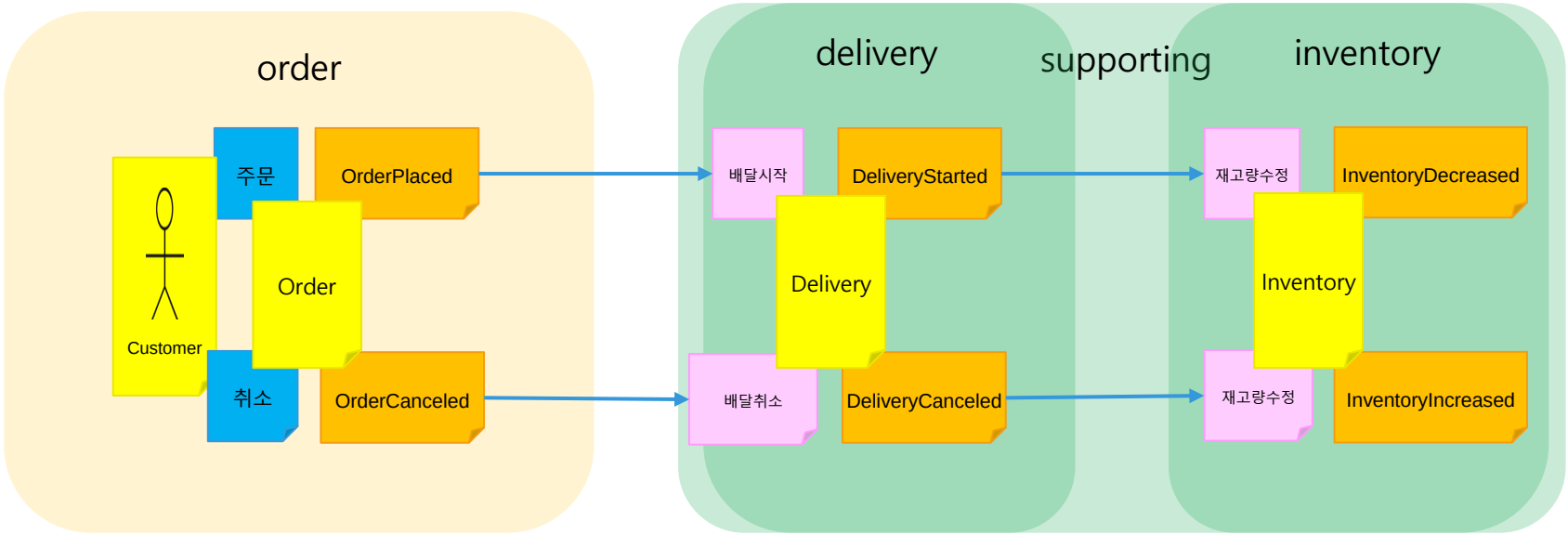
Lab Time – Bounded Context 분리



PubSub via Event Stream
(e.g. Kafka, Axon Server)

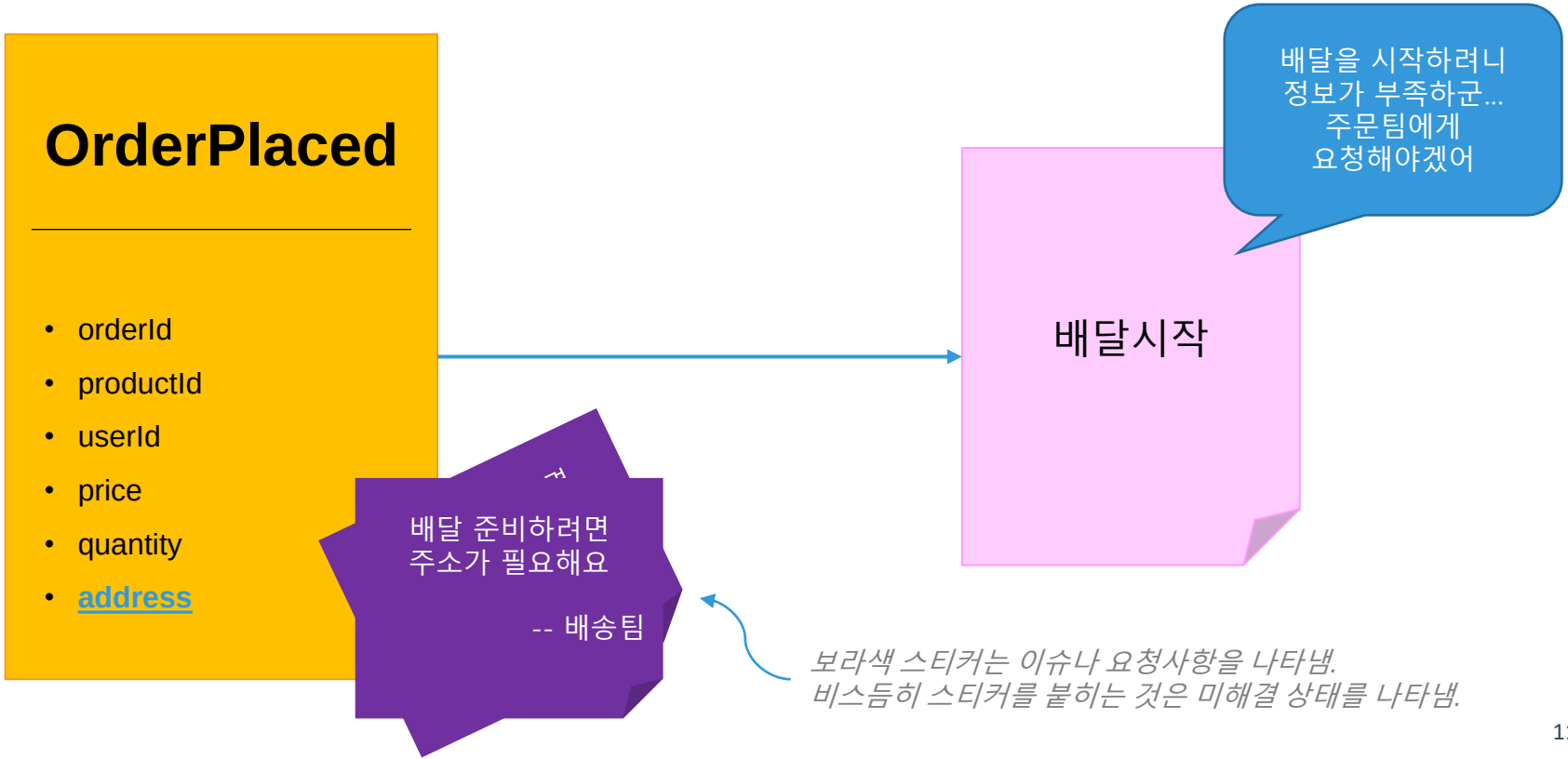
PubSub via Local Event Design Pattern
(e.g. White-board Pattern)

Lab Time – Bounded Context 분리 (다음스프린트)

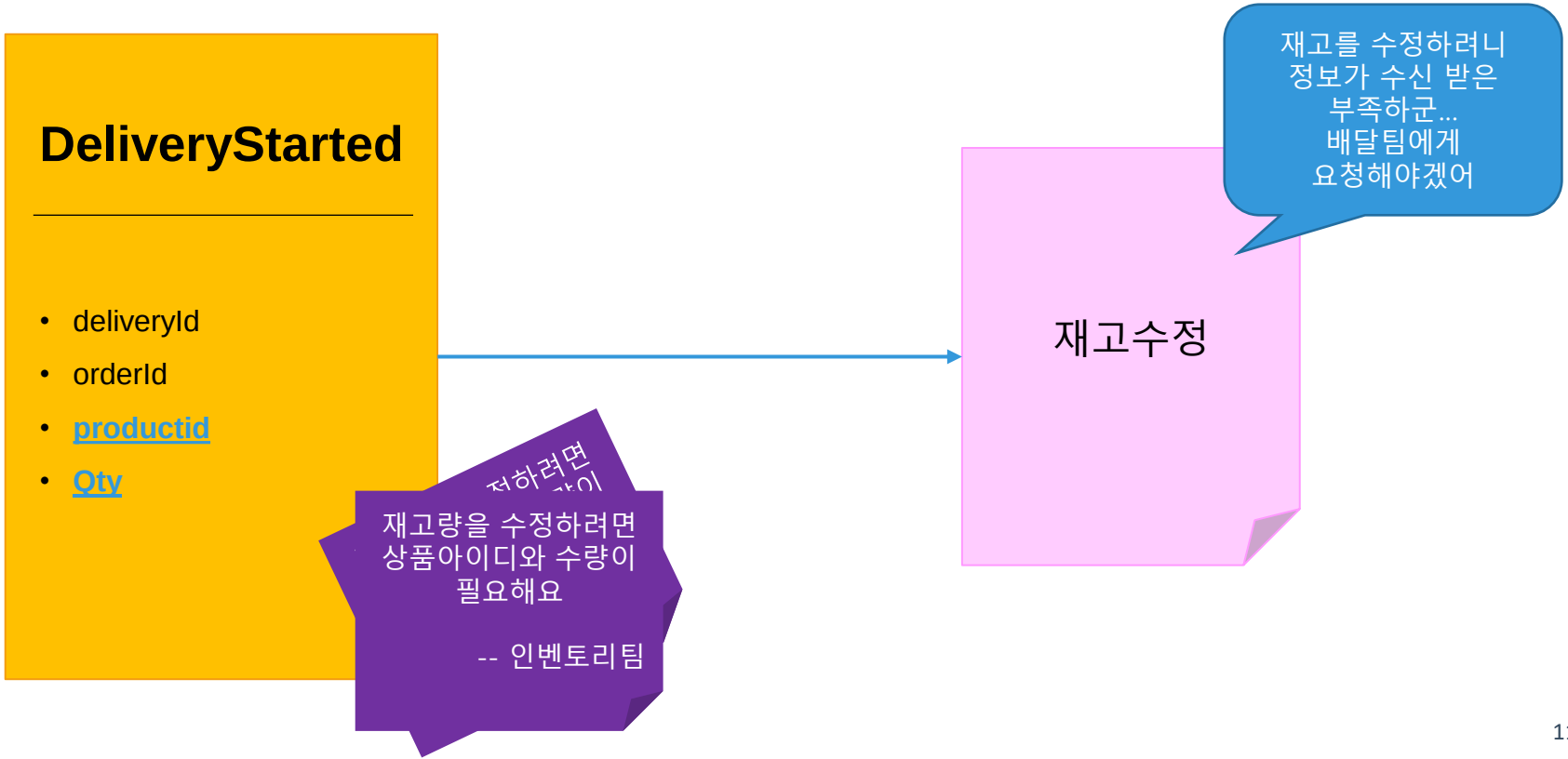


PubSub via Event Stream
(e.g. Kafka, Axon Server)

Lab Time – 도메인 이벤트의 속성 선언



Lab Time – 도메인 이벤트의 속성 선언(2)



Quiz. 다음중 도메인 이벤트로 부적절한 것은?

O

다른 팀에서 관심 가질만함

다른 팀에서 관심 가질만함

적당한 사이즈임

X

It's a command

It doesn't make any state change

Too technical and too fine-grained

Be business level

Nobody will be interested in this event

Tools around Event Storming and DDD

Collaboration & Code Generation

- www.msaez.io
- https://docs.vlingo.io/xoom-designer
- https://contextmapper.org/

Frameworks

- Xoom Vlingo Framework: https://docs.vlingo.io/
- Axon Framework: https://axoniq.io/

Collaboration Tools

- https://miro.com

Table of content

Microservice and
Event-storming-Based
DevOps Project

UEngine

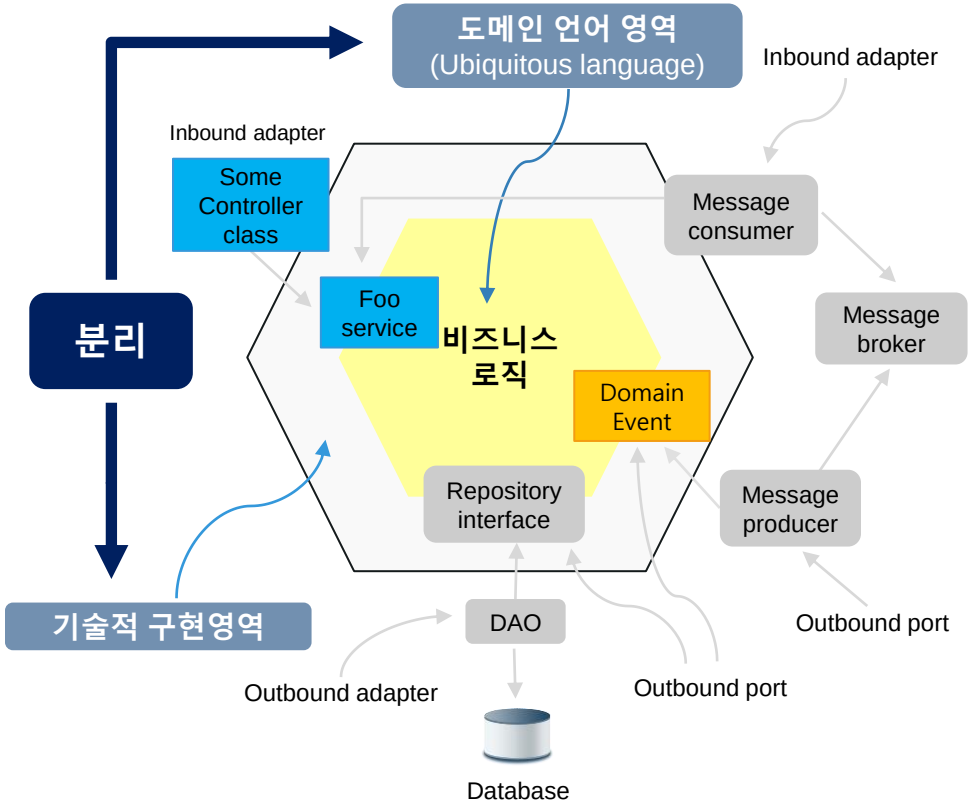
Microservice Implementation Pattern (1)

- **Hexagonal Architecture**

Create your service to be independent of eit her UI or database and to provide adapters f or different input/output sources such as GUI, DB, test harness, RESTful resource, etc.

Implement the publish-and-subscribe messa ging pattern: As events arrive at a port, an ad apter (a.k.a. service agent) converts it into a p rocedure call or message and passes it to the application. When the application has someth ing to publish, it does it through a port to an a dapter, which creates the appropriate signals needed by the receiver.

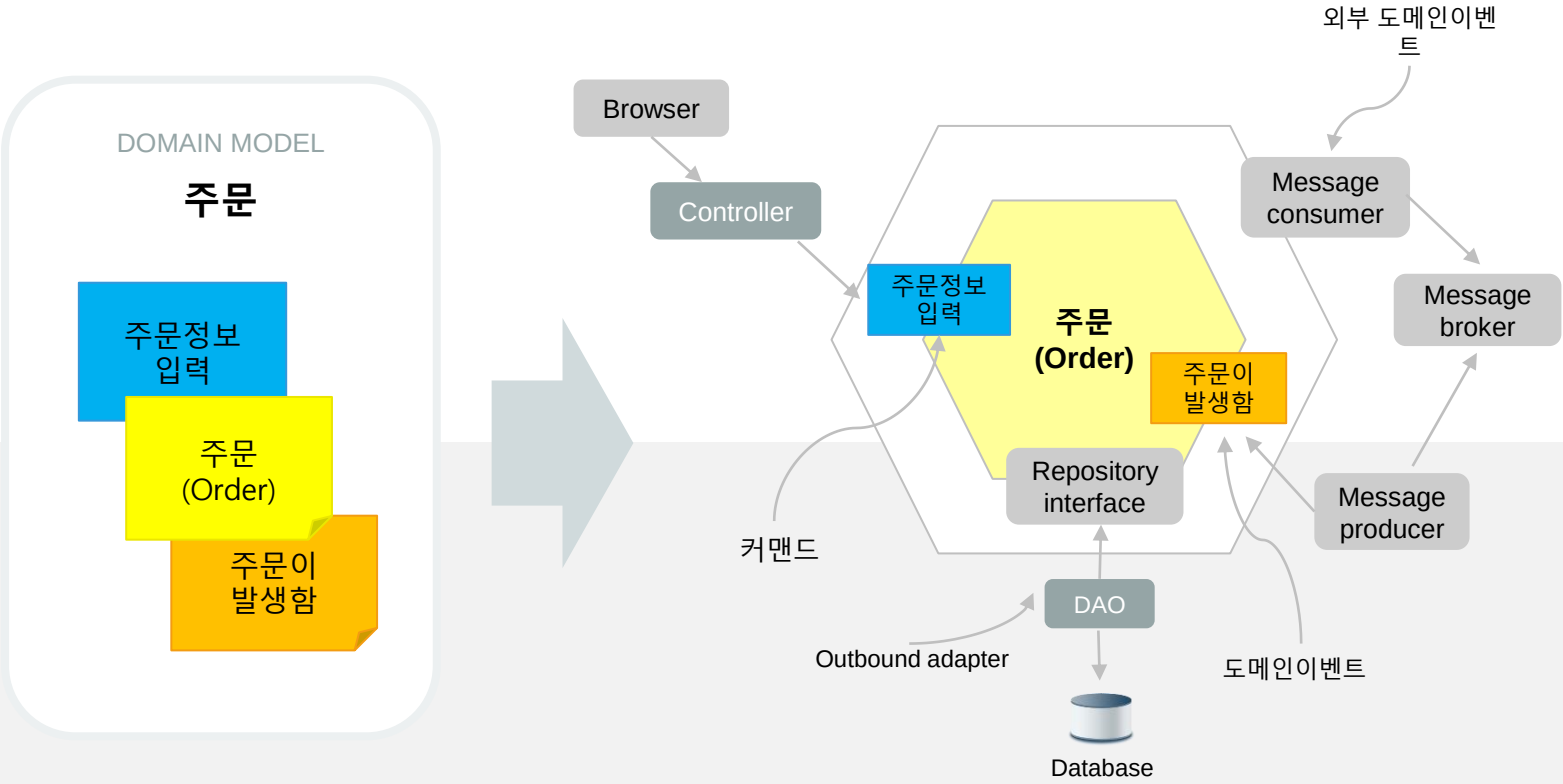
https://alistair.cockburn.us/Hexagonal+architecture



Service Implementation

- You have Powerful Tool:
  Domain-Driven Design and Spring Boot / Spring Data REST
- Domain Classes : Entity or Value Object
- Resources can be bound to Repositories □ Full HATEOAS service can be generated!
- Services can be implemented with Resource model firstly
- Low-level JAX-RS can be used if not applicable above

Applying Hexagonal Architecture

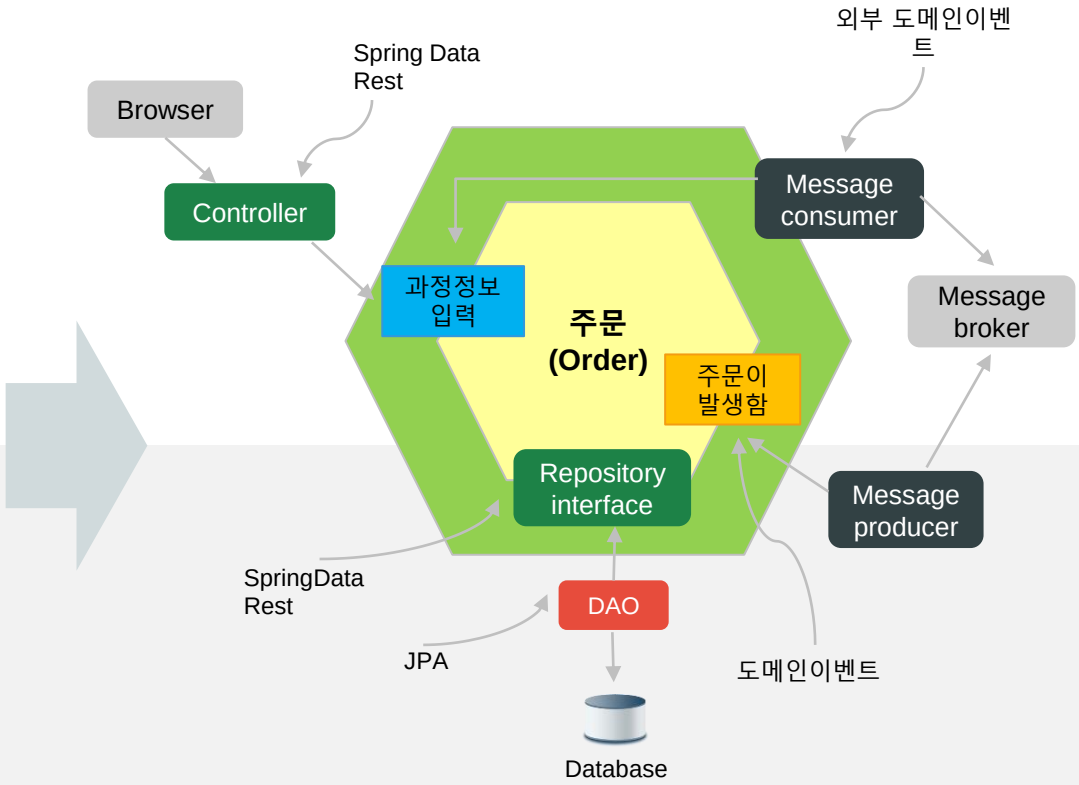


Applying MSA Chassis

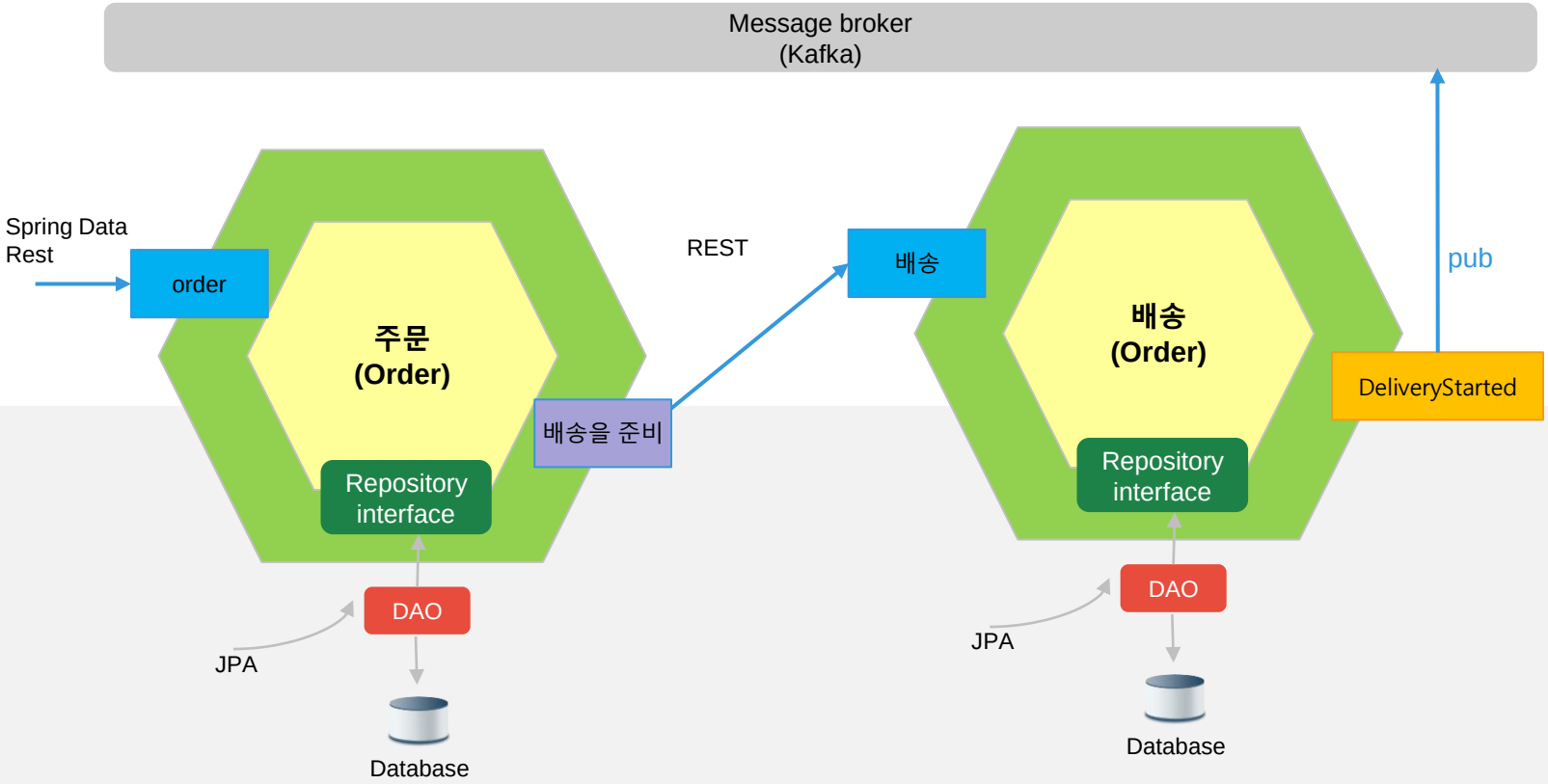
DOMAIN MODEL

Spring Boot / Cloud

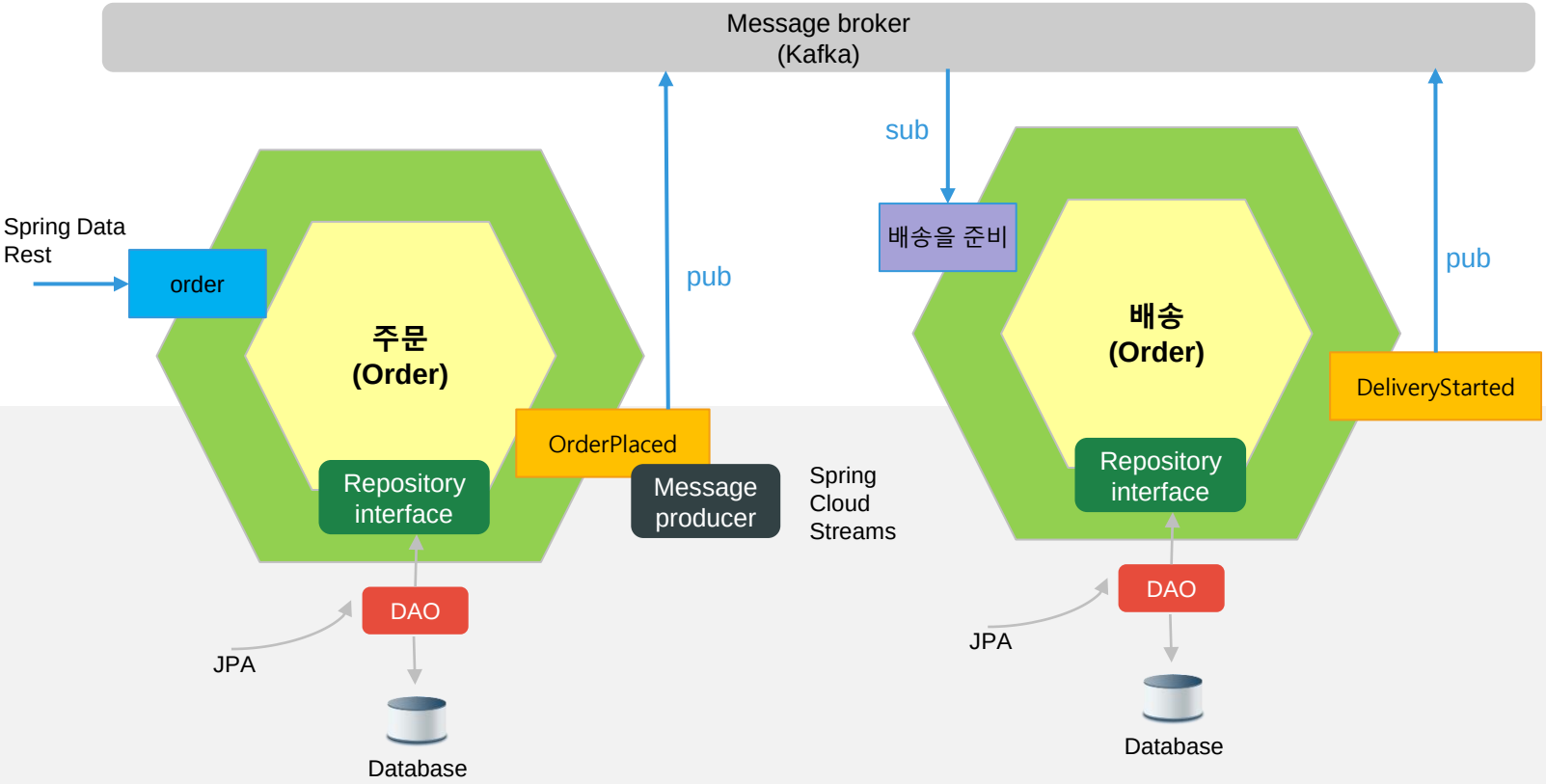
- Spring Boot
- Spring Data Rest
- JPA
- Spring Cloud Stream



Applying MSA Chassis



Applying MSA Chassis



이벤트 스토밍 결과에서 구현 기술 연동

요소	디자인패턴	미들웨어/프레임워크/라이브러리
이벤트 (Domain Event)	• 도메인 이벤트	• 카프카, Rabbit MQ, Active MQ ... • CDC (Change Data Capturing)
커맨드 (Command)	• 애그리거트 내의 메서드 • 서비스 • 레포지토리	• Spring Data REST • REST-Easy
결합물 (Aggregate)	• 애그리거트 루트 + 도메인 클래스 묶음 • 엔티티 (ORM) • Value Objects	• JPA • My batis
정책 (Policy)	• 이벤트 리스너 • 애그리거트 내의 메서드를 호출	• Spring Cloud Stream
바운디드 컨텍스트	• 마이크로 서비스 (후보)	• Spring Boot • Container (Docker) • OSGi or VM (옛말)

Aggregate

Order

- orderId
- productId
- userId
- price
- quantity
- telephone

Aggregate Root

```
@Entity
public class Order{

  @Id  Long id;
  Long productId;
  String customerId;
  Money price;
  int quantity;
  PhoneNumber telephone;
… setter/getters …

}
```

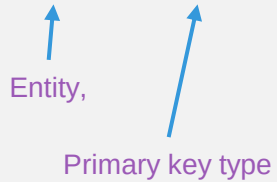
Aggregate Members

```
@Entity
public class OrderDetail {

….

… setter/getters …

}
```

```
//Value Objects
public class Money {

….

… setter/getters …

}
```

Repository

```
public interface OrderRepository extends JpaRepository<Order, Long>{

    // leave it blank

}
```



Command

주문취소

```
@Service
public class ProductService {

    @RequestMapping(
        method = RequestMethod.DELETE,
        path="orders/{orderId}/"
    )

    public void cancelOrder(@PathVariable("productId") Long orderId){

        orderReposioty.findById(orderId).ifPresent(order -> {
            order.setStatus(Status.CANCELLED);
            order.setOrderDetails(null);
            orderRepository.save(order);
        })

    }
}
```

Anemic Domain Model

비즈니스 로직은 Aggregate 내부의 메소드에 작성해야함.

Command

주문취소

Command → Aggregate 내부의 메서드

```
@Entity
public class Order {
 …
    protected void setStatus(Status status){…}

    public void cancelOrder() {
            setOrderDetails(null);
            setStatus(Status.CANCELLED);
    }
}
```

Command

주문취소


```
@Service
public class ProductService {

    @RequestMapping(
          method = RequestMethod.DELETE,
          path=/orders/{orderId}/”
    )

    public void cancelOrder(@PathVariable(”productId") Long orderId){

        orderReposioty.findById(orderId).ifPresent(order -> {
            order.cancelOrder();
            orderRepository.save(order);
        })

    }
}
```

Domain Event

OrderPlaced

- orderId
- productId
- userId
- price
- quantity
- ~~Telephone~~
- timestamp

개발 (POJO)

```
public class OrderPlaced{

  Long orderId;
  Long productId;
  String userId;
  double price;
  int quantity;

… setter/getters …

}
```

카프카

실행 (JSON)

```
{
  type: “OrderPlaced”,
  name: “캠핑의자”,
 userId : “1@uengine.org”,
  orderId: 12345,
  price:  100
 quantity :   10
}
```

Publishing Domain Events

Order

- orderId
- productId
- userId
- price
- quantity
- Telephone

- publishOrderPlaced()

Event □ 이벤트 발사로직

```
@Entity
public class Order {
 …

   @PostPersist   // 주문이 저장된 후에
   private void publishOrderPlaced() {
      OrderPlaced orderPlaced = new OrderPlaced();

      orderPlaced.setOrderId(id);
          …

      orderPlaced.publish();
   }
}
```

Subscribing Domain Events

(Whenever OrderPlaced)

배송을 준비함

```
@StreamListener(topics = "shopping")
public void onOrderPlaced(OrderPlaced orderPlaced)

    deliveryService.start(orderPlaced.getOrderId());

}
```

How to develop “Ubiquitous Language”

Bounded Contexts, Teams, and Source Code Repositories

There should be one team assigned to work on one *Bounded Context*. There should also be a separate source code repository for each *Bounded Context*. It is possible that one team could work on multiple *Bounded Contexts* , but multiple teams should not work on a single *Bounded Context*. Cleanly separate the source code and database schema for each *Bounded Context* in the same way that you separate the *Ubiquitous Language*. Keep acceptance tests and unit tests together with the main source code.

It is especially important to be clear that one team works on a single *Bounded Context*. This completely eliminates the chances of any unwelcome surprises that arise when another team makes a change to your source code. Your team owns the source code and the database and defines the official interfaces through which your *Bounded Context* must be used. It’s a benefit of using DDD

- On the other hand, DDD emphasizes embracing such differences by segregating the differing types
- into different Bounded Contexts. Admit that there are different languages, and function accordingly.
- Are there three meanings for policy? Then there are three Bounded Contexts , each with its own
- policy, with each policy having its own unique qualities. There’s no need to name these
- UnderwritingPolicy , ClaimsPolicy , or InspectionsPolicy . The name of the
- Bounded Context takes care of that scoping. The name is simply Policy in all three Bounded
- Contexts.

You may be wondering how you can make the transition from a written scenario to some sort of artifact that can be used to validate your domain model against the team's specifications. There is a technique named *Specification by Example* [Specification] that can be used; it's also called *Behavior-Driven Development* [BDD] . What you are trying to achieve with this approach is to collaboratively develop and refine a *Ubiquitous Language* , model with a shared understanding, and determine whether your model adheres to your specifications. You will do this by creating acceptance tests. Here is how we might restate the preceding scenario as an executable specification:

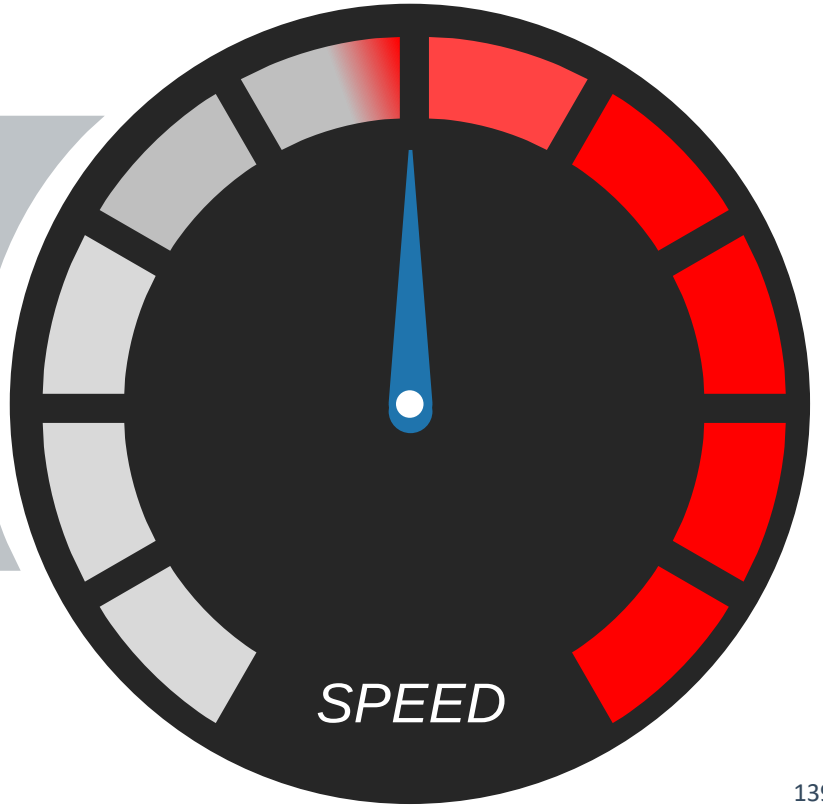
Click here to view code image

```
Scenario:
  The product owner commits a backlog item to a sprint
   Given a backlog item that is scheduled for release
   And the product owner of the backlog item
   And a sprint for commitment
   And a quorum of team approval for commitment
   When the product owner commits the backlog item to the sprint
   Then the backlog item is committed to the sprint
   And the backlog item committed event is created
```

With a scenario written in this form, you can create some backing code and use a tool to execute

Speed to digital transformation

Software development is a learning process, working code is a side effect - Martin Fowler



Thank you

- 수고 많으셨습니다!

The key to controlling complexity is a good domain model

– Martin Fowler

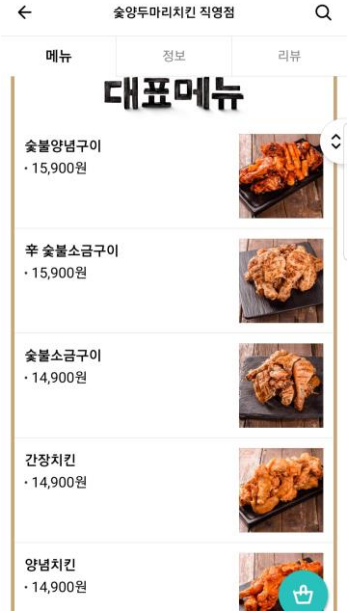
실습 예제

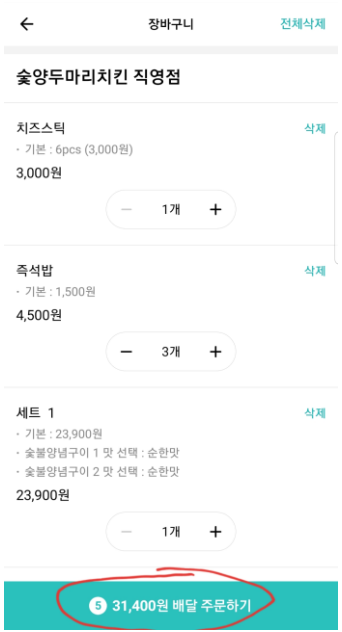
Food Delivery

https://github.com/msa-ez/example-food-delivery

Context: Food Delivery App – Front

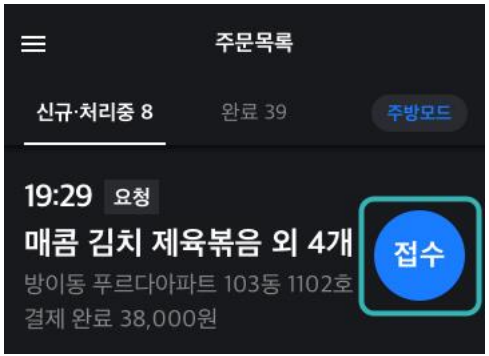
https://github.com/msa-ez/example-food-delivery

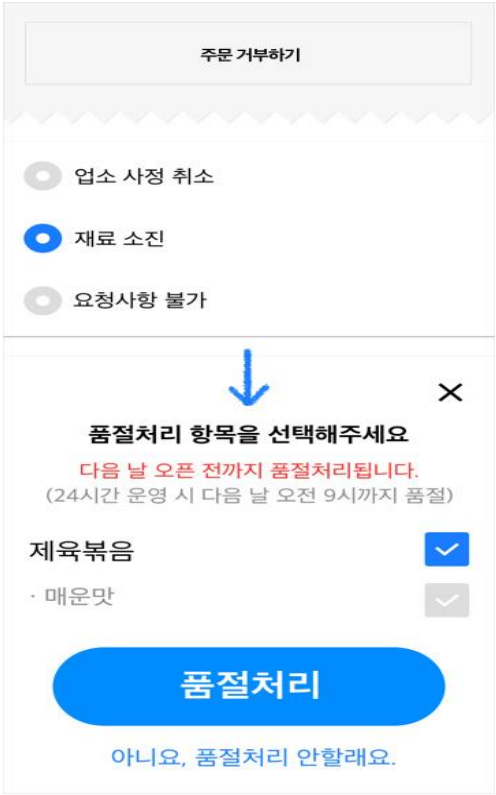




Context: Food Delivery App - Store

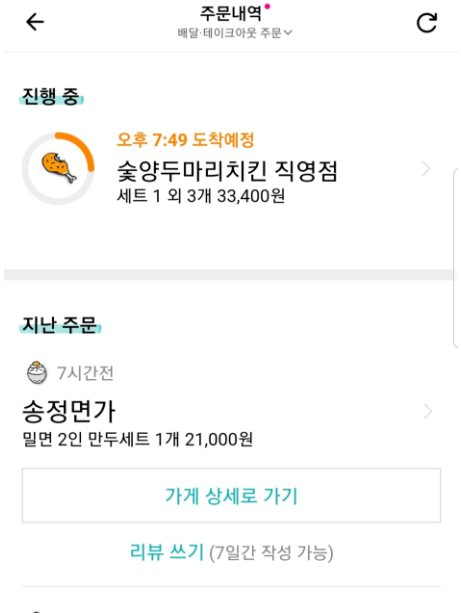
https://github.com/msa-ez/example-food-delivery

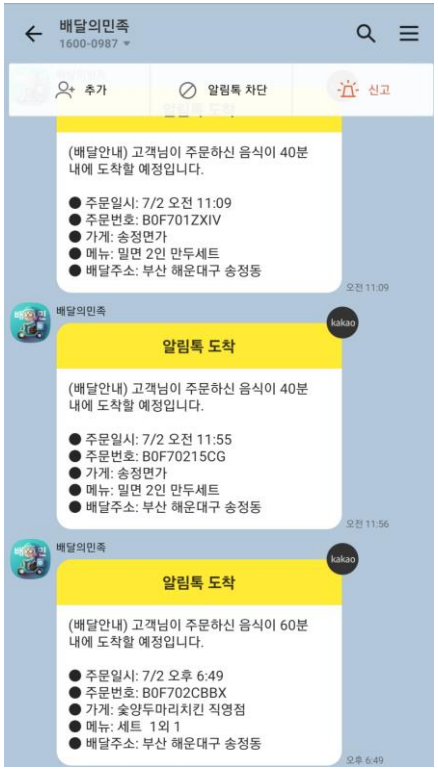




Context: Food Delivery App - Customer

https://github.com/msa-ez/example-food-delivery





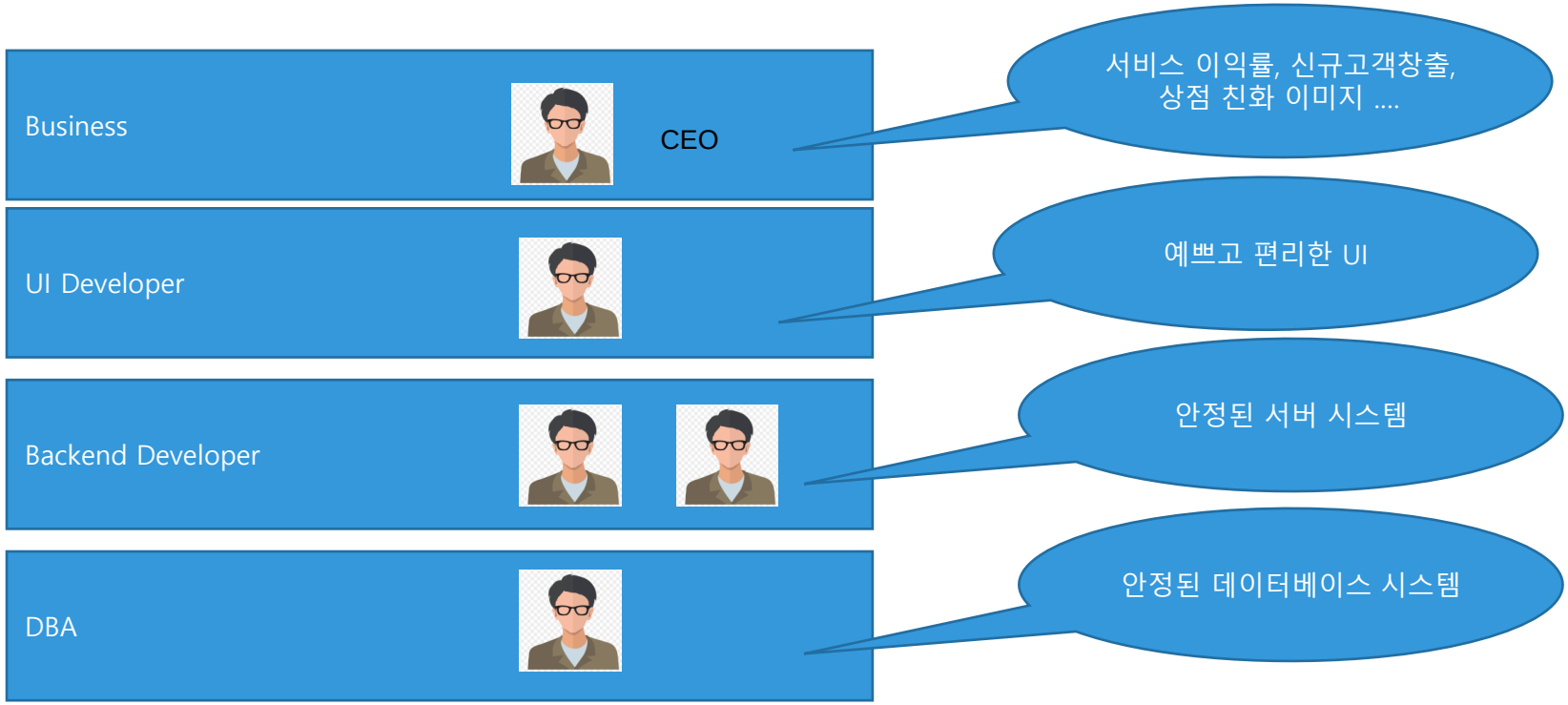
분석/설계

“배달의 민족” MSA 로 설계/구현하기

시나리오

1. 고객이 메뉴를 선택하여 주문한다
2. 고객이 결제한다
3. 주문이 되면 주문 내역이 입점상점주인에게 전달된다
4. 상점주인이 확인하여 요리해서 배달 출발한다
5. 고객이 주문을 취소할 수 있다
6. 주문이 취소되면 배달이 취소된다
7. 고객이 주문상태를 중간중간 조회한다
8. 주문상태가 바뀔 때 마다 카톡으로 알림을 보낸다

창업시기 조직구조 – Horizontal



조직구조 – Vertical

CEO

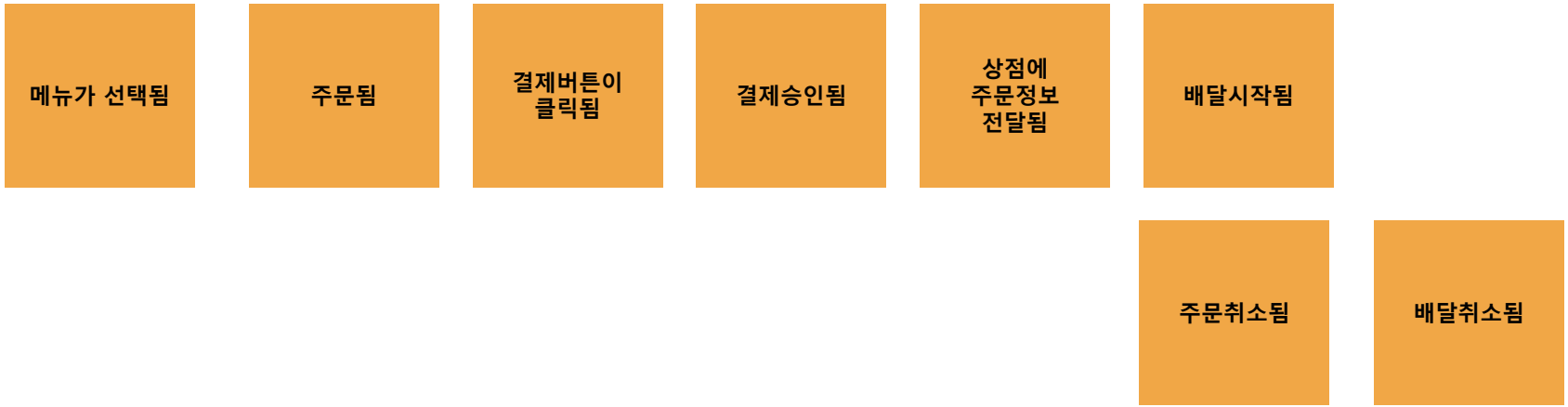
CTO

비기능적 요구사항

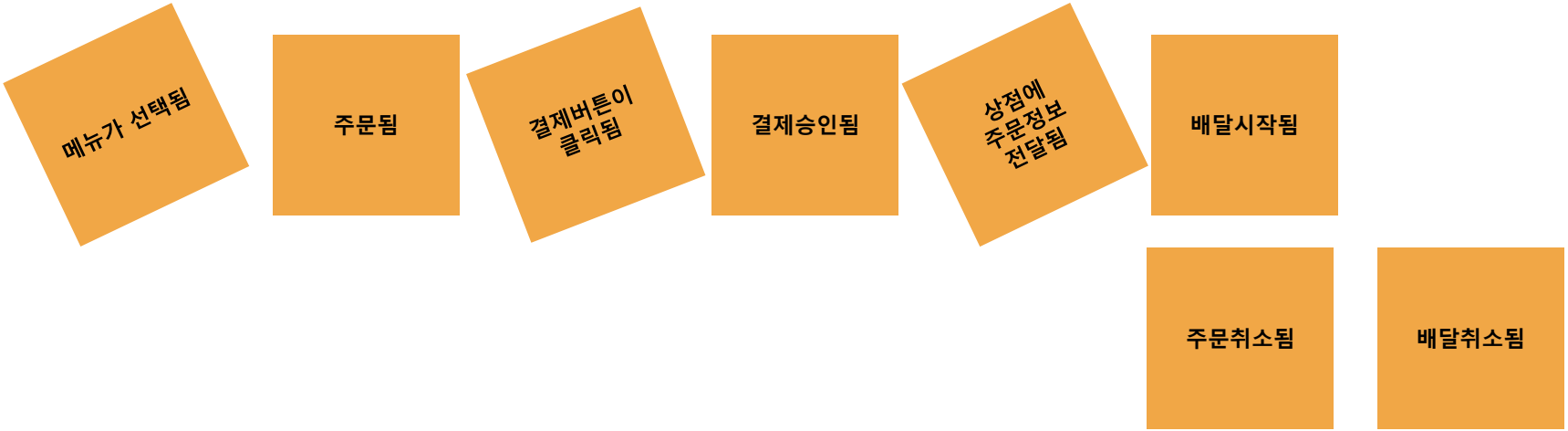
1. 트랜잭션
    1. 결제가 되지 않은 주문건은 아예 거래가 성립되지 않아야 한다 → Sync 호출

2. 장애격리
    1. 상점관리 기능이 수행되지 않더라도 주문은 365일 24시간 받을 수 있어야 한다 → Async (event-driven), Eventual Consistency
    2. 결제시스템이 과중되면 사용자를 잠시동안 받지 않고 결제를 잠시후에 하도록 유도한다 → Circuit breaker, fallback
3. 성능
    1. 고객이 자주 상점관리에서 확인할 수 있는 배달상태를 주문시스템(프론트엔드)에서 확인할 수 있어야 한다 → CQRS
    2. 배달상태가 바뀔때마다 카톡 등으로 알림을 줄 수 있어야 한다 → Event driven

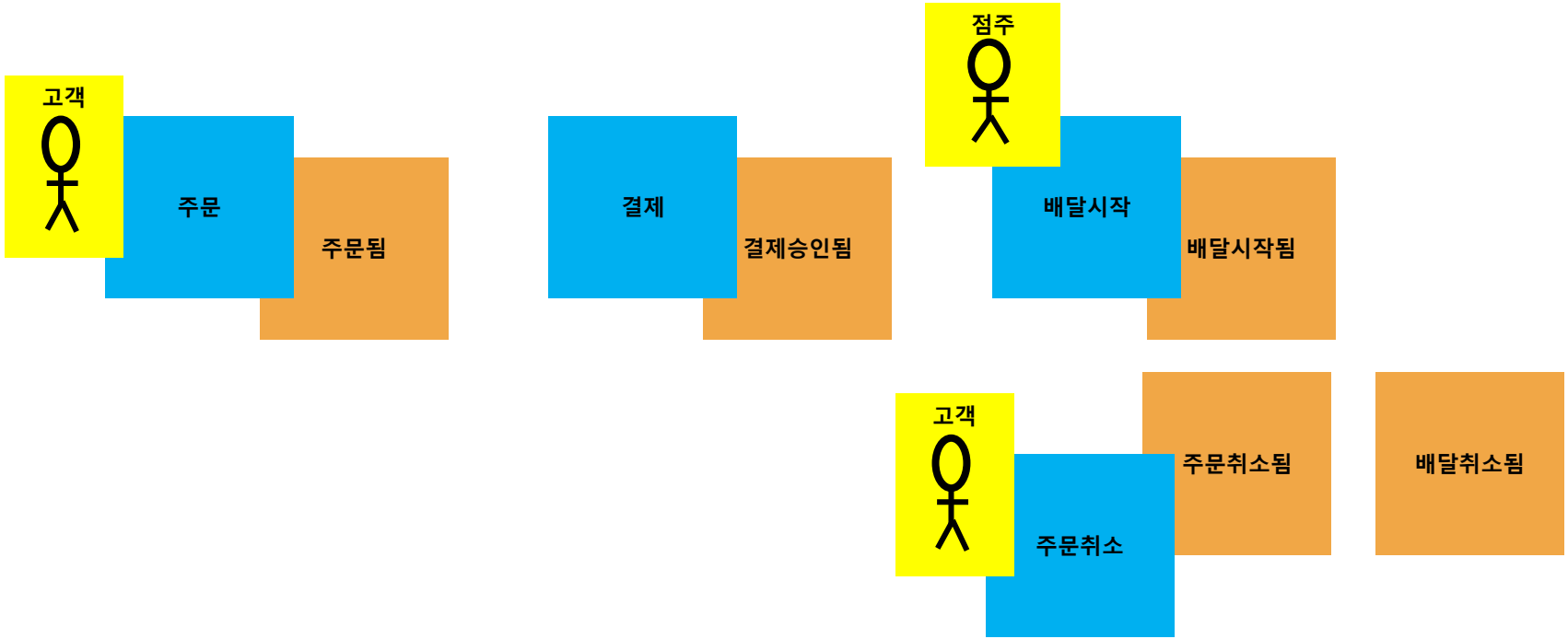
이벤트스토밍 - Event



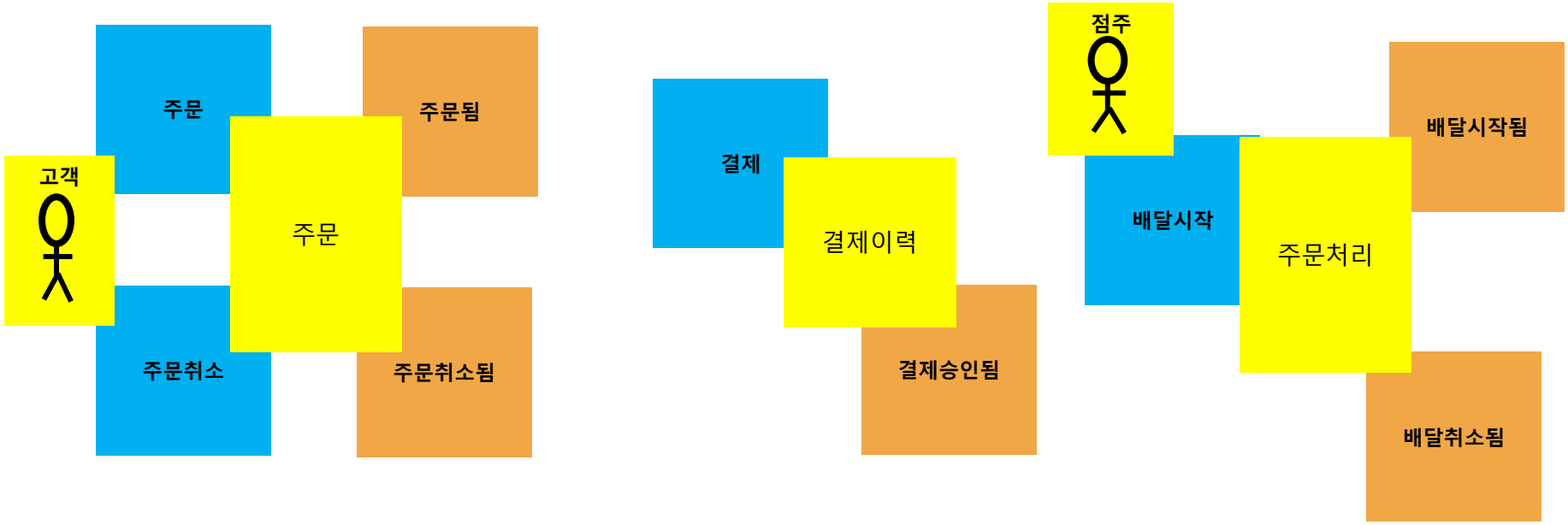
이벤트스토밍 – 비적격 이벤트 제거



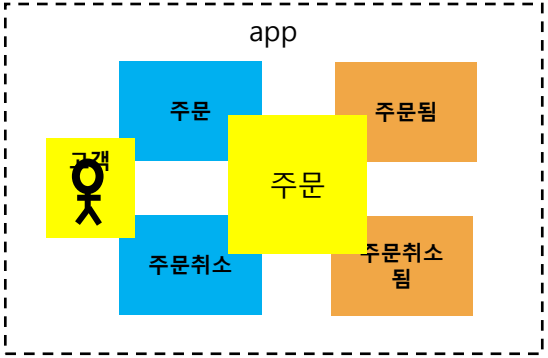
이벤트스토밍 – Actor, Command

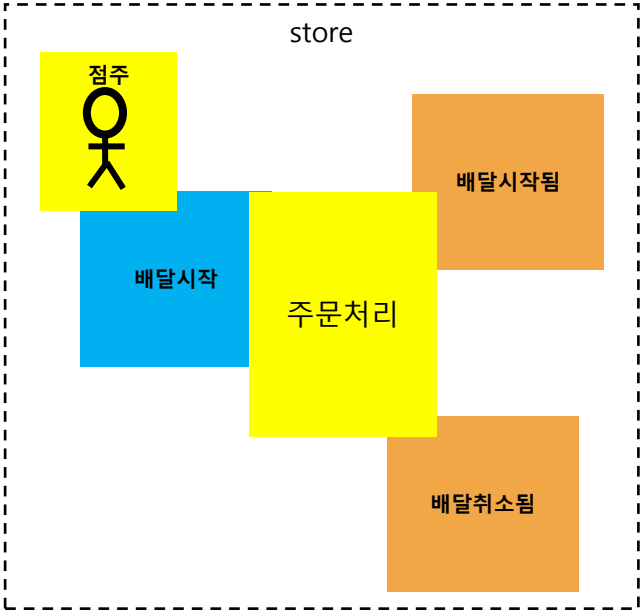


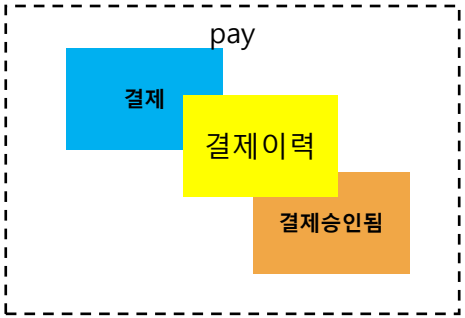
이벤트스토밍 – Aggregate



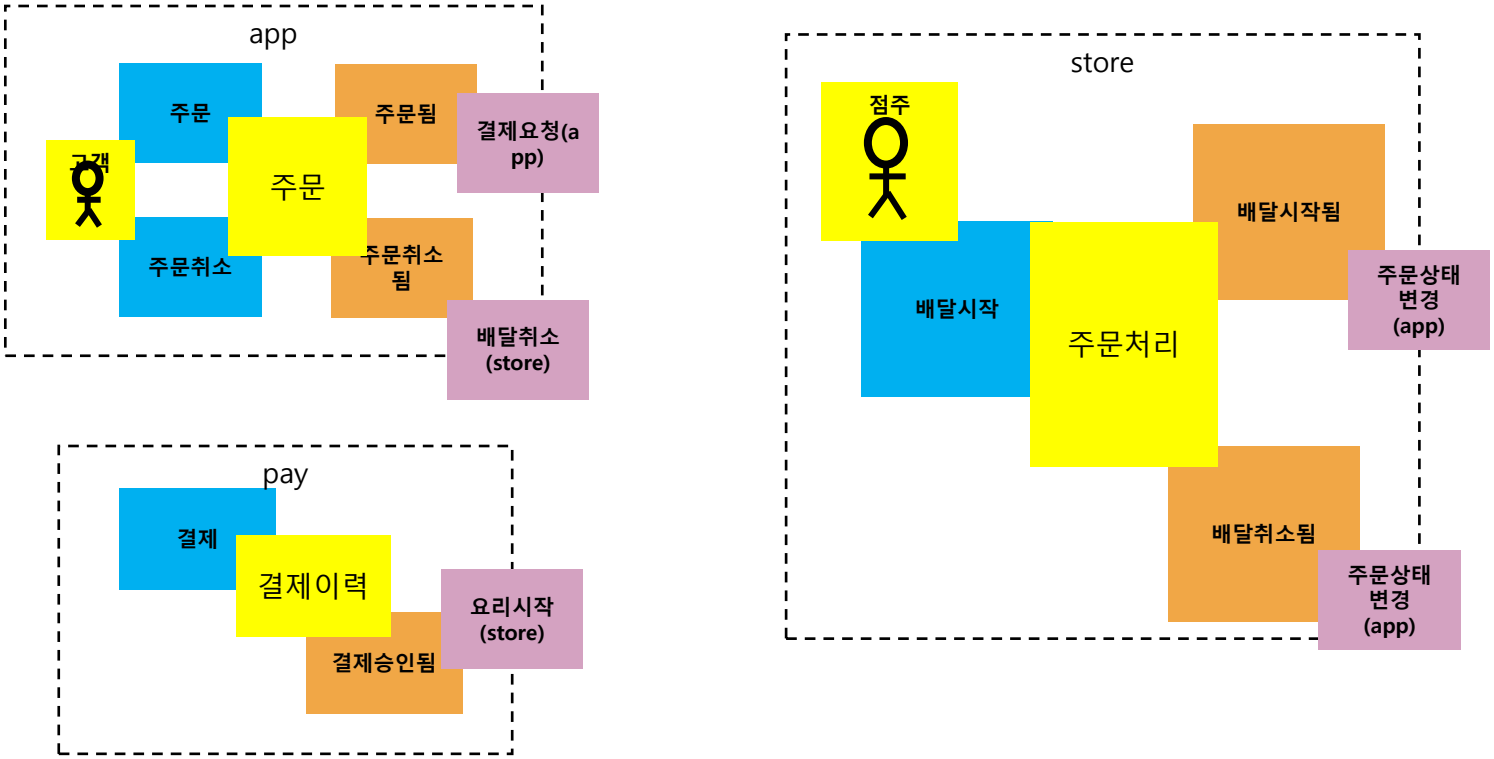
이벤트스토밍 – Bounded Context



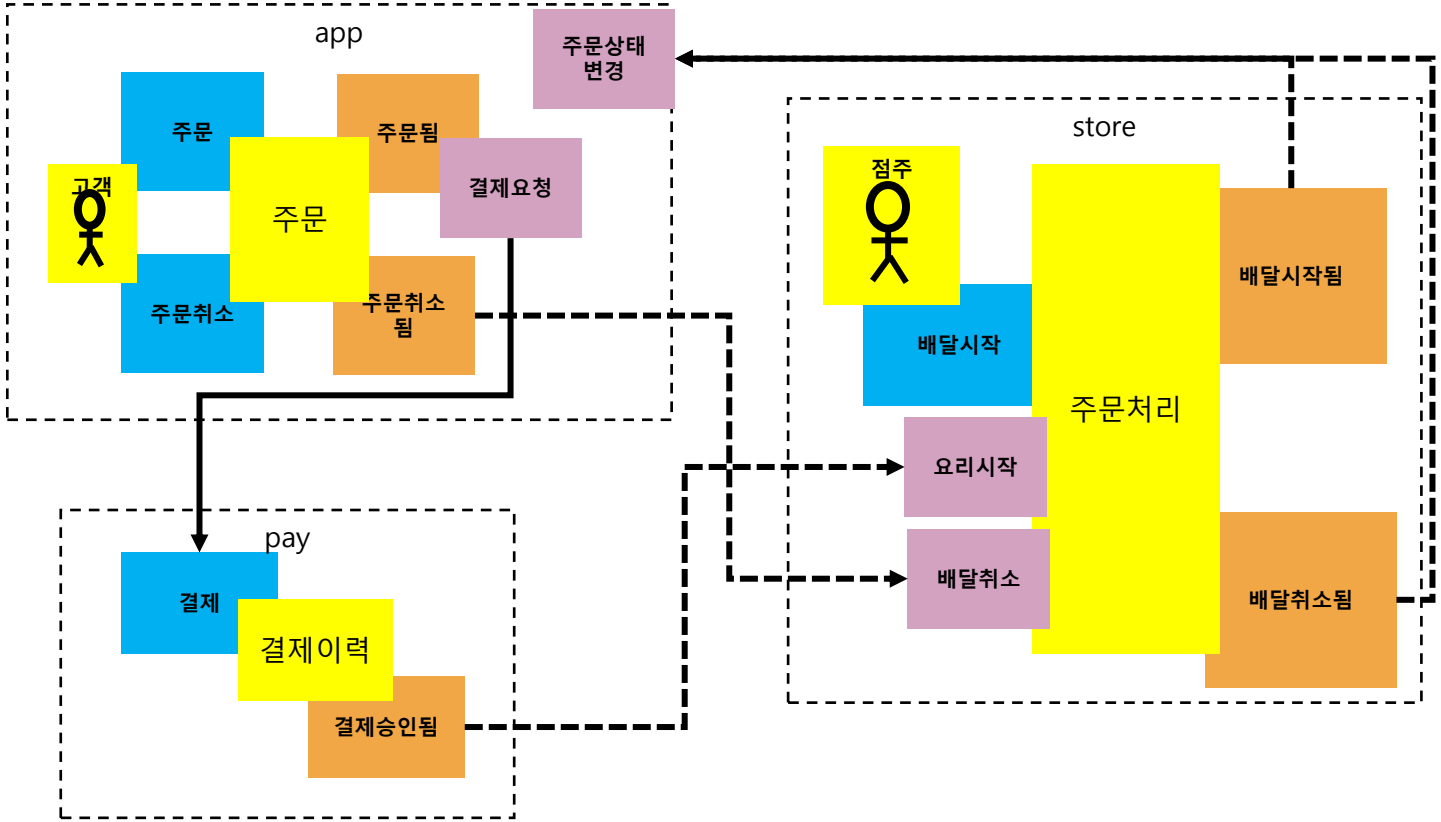




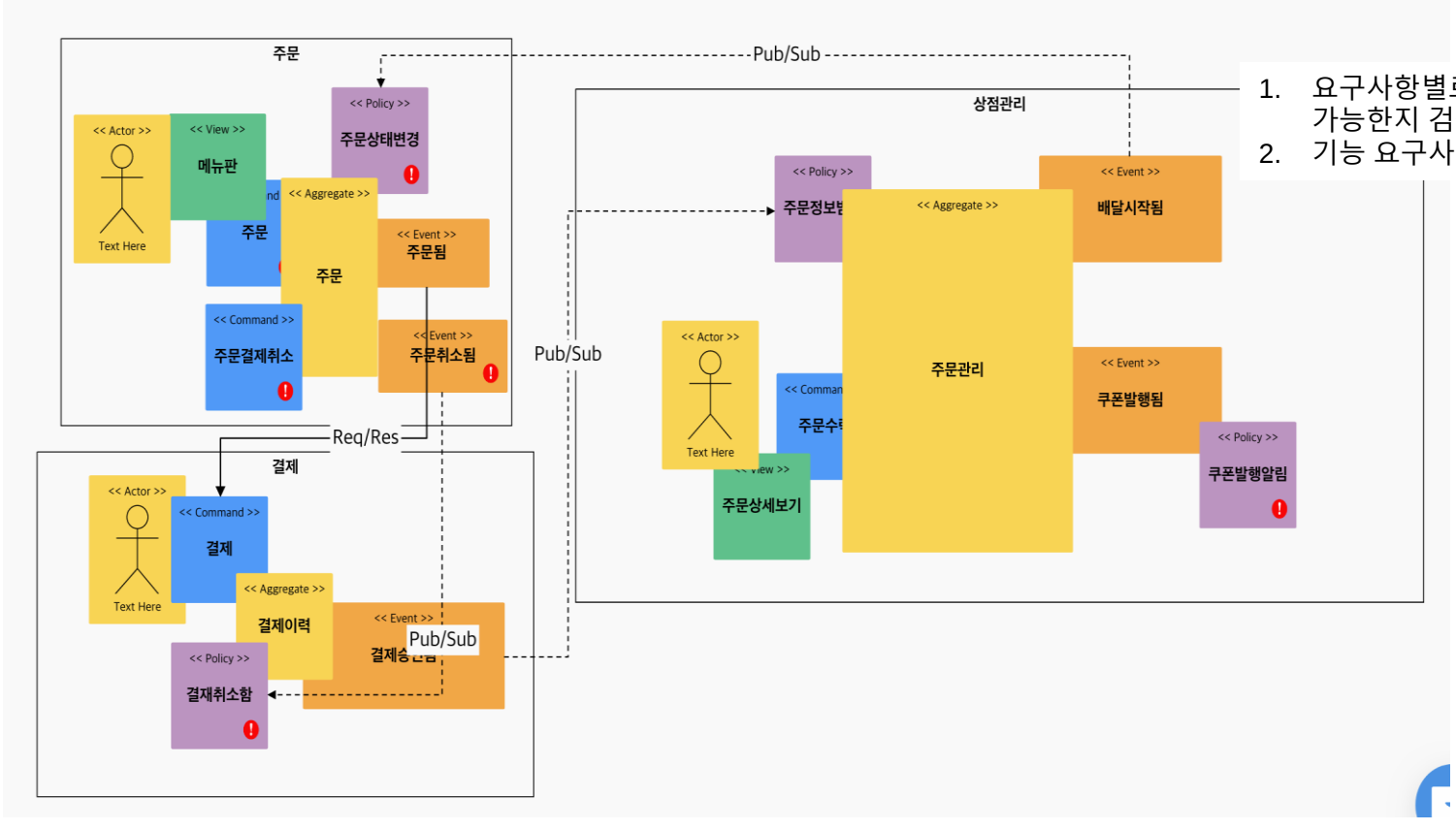
이벤트스토밍 – Policy (괄호 - 수행주체)



이벤트스토밍 – Policy 를 수행주체로 이동



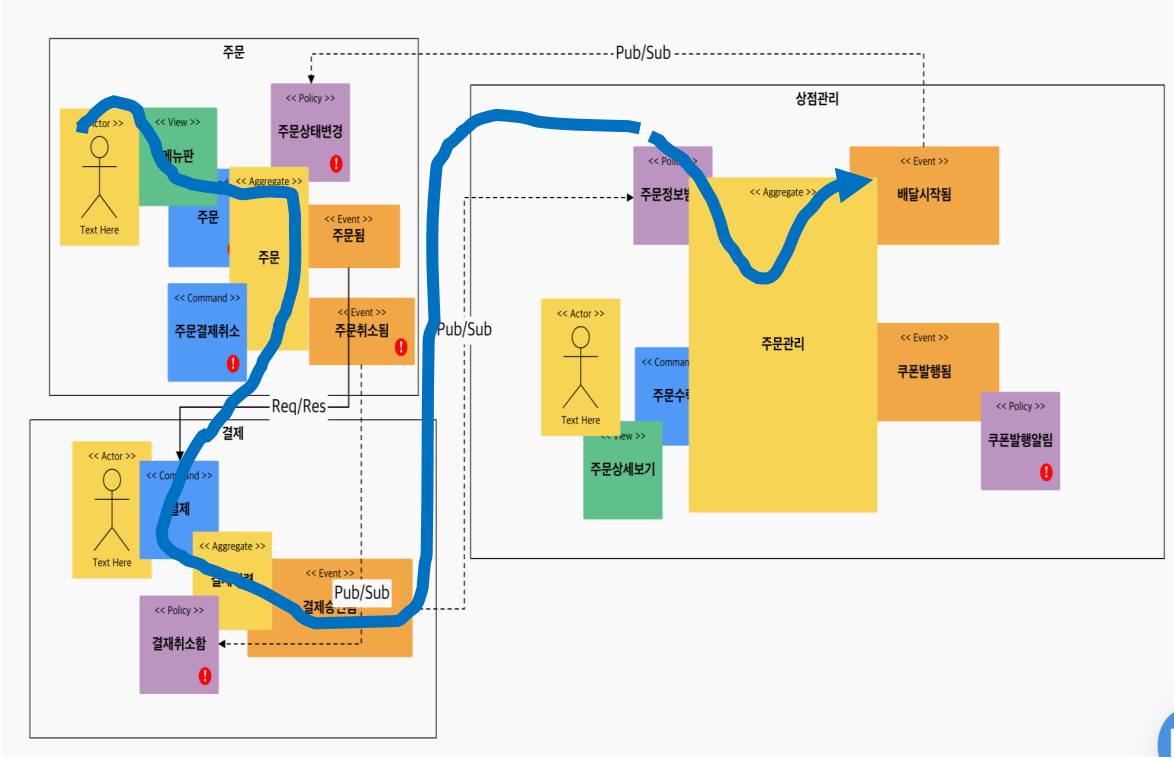
기능 요구사항 coverage



1. 요구사항별로 모든 나래이션이 가능한지 검증함
2. 기능 요구사항별로 패스 표시

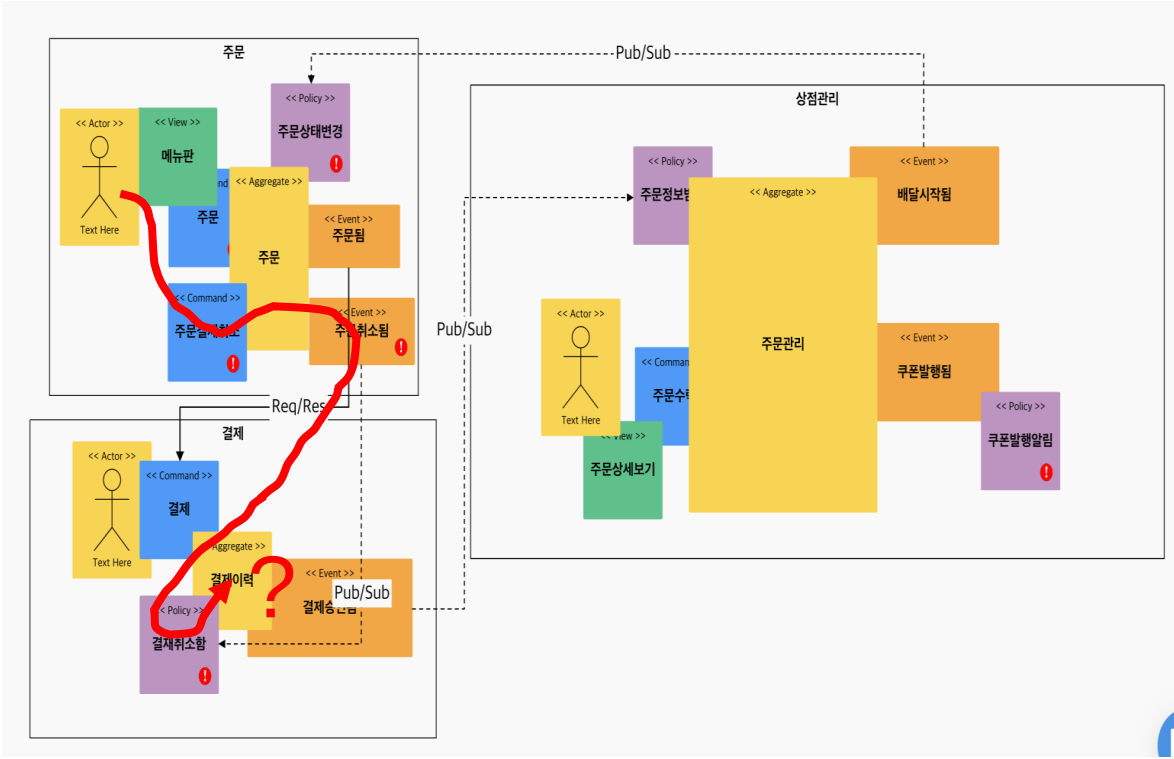
시나리오 Coverage Check (1)

1. 고객이 메뉴를 선택하여 주문한다
2. 고객이 결제한다
3. 주문이 되면 주문 내역이 입점상점주인에게 전달된다
4. 상점주인이 확인하여 요리해서 배달 출발한다



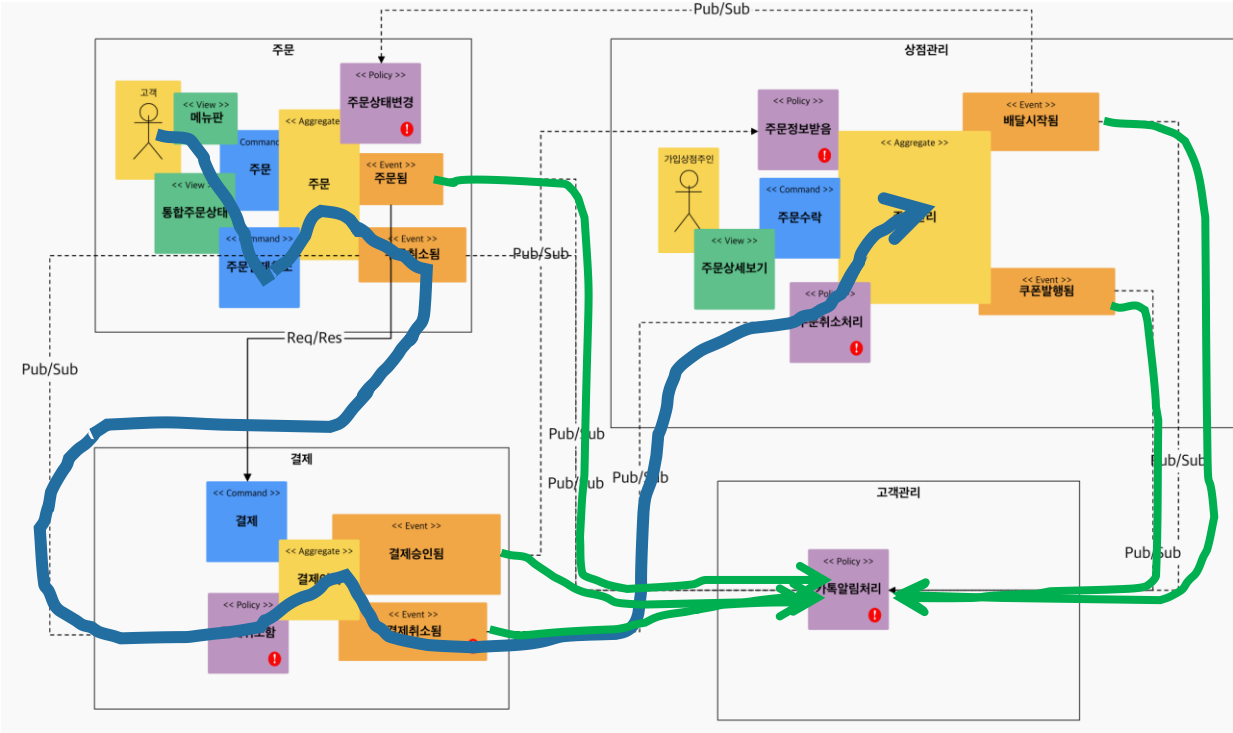
시나리오 Coverage Check (2)

5. 고객이 주문을 취소할 수 있다
6. 주문이 취소되면 배달이 취소된다 ➔ 배달 취소는?
7. 고객이 주문상태를 중간중간 조회한다
8. 주문상태가 바뀔 때마다 카톡으로 알림을 보낸다 ➔ ?



모델 업그래이드 – 요구사항 커버 확인

5. 고객이 주문을 취소할 수 있다
6. 주문이 취소되면 배달이 취소된다
7. 고객이 주문상태를 중간중간 조회한다
8. 주문상태가 바뀔 때마다 카톡으로 알림을 보낸다

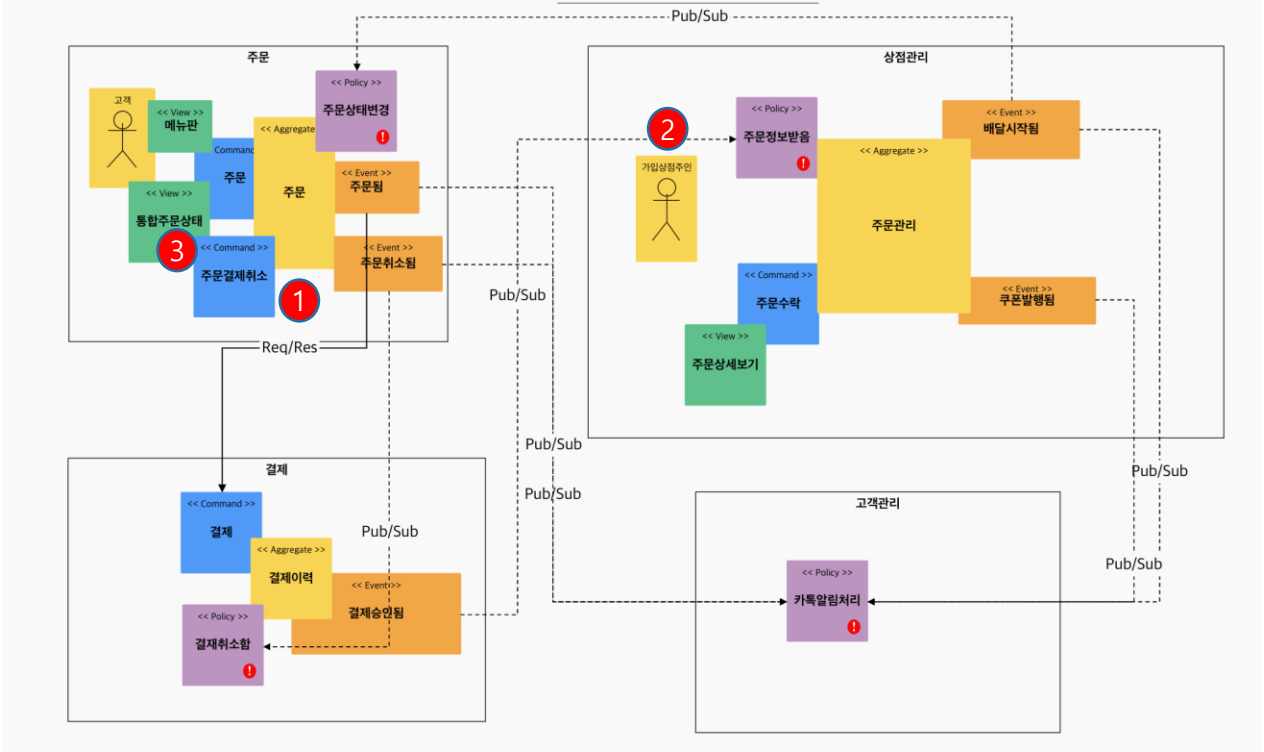


비기능 요구사항 coverage

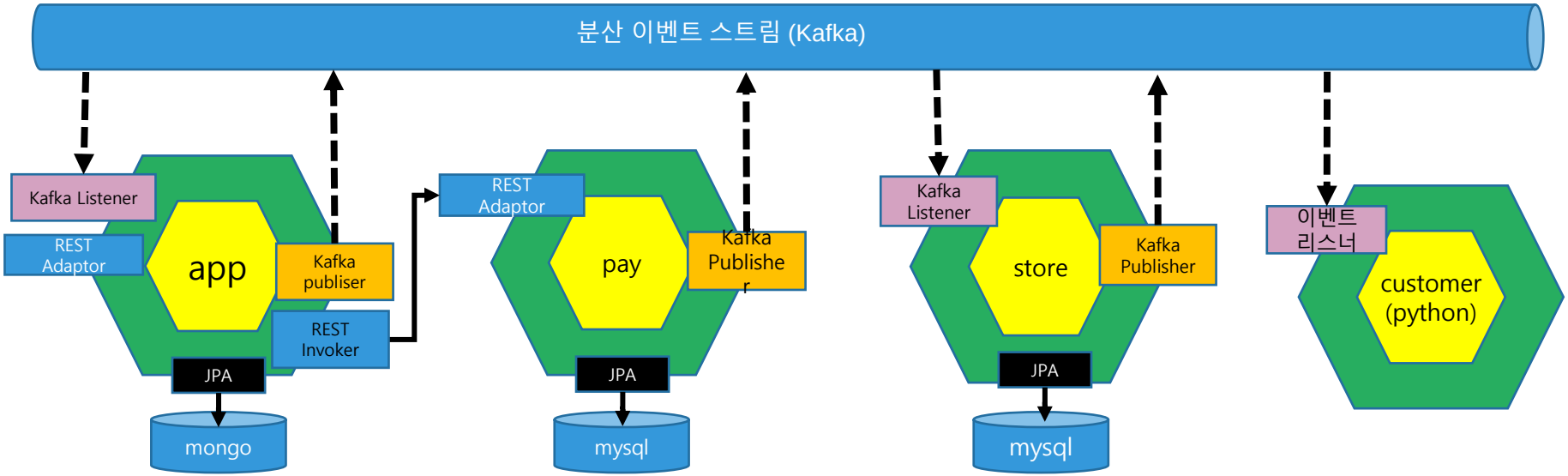
1. 주문에 대해서는 결제가 처리되어야만 주문 처리하고 장애격리를 위해 CB를 설치함 (트랜잭션 > 1, 장애격리 > 2)

2. 결제승인 이벤트를 수신하여 상점의 주문정보 변경을 수행함 (장애전파 > 1)

3. 상점의 배달관련 이벤트를 주문에서 수신하여 View Table 을 구성 (CQRS) (성능 > 1)



헥사고날 아키텍처


분산 이벤트 스트림 (Kafka)
Kafka Listener
REST Adaptor
app
Kafka publiser
REST Invoker
JPA
mongo
REST Adaptor
pay
Kafka Publisher
JPA
mysql
Kafka Listener
store
Kafka Publisher
JPA
mysql
이벤트 리스너
customer (python)

구현

- Unit Microservice Implementation

  Spring Boot

- Inter-microservice Communication

  Publish-Subscribe (Saga)

  - Event Sourcing
  - Correlation
  - Compensation

  Request-Response (RPC)

  - Service Registry
  - Circuit Breaker

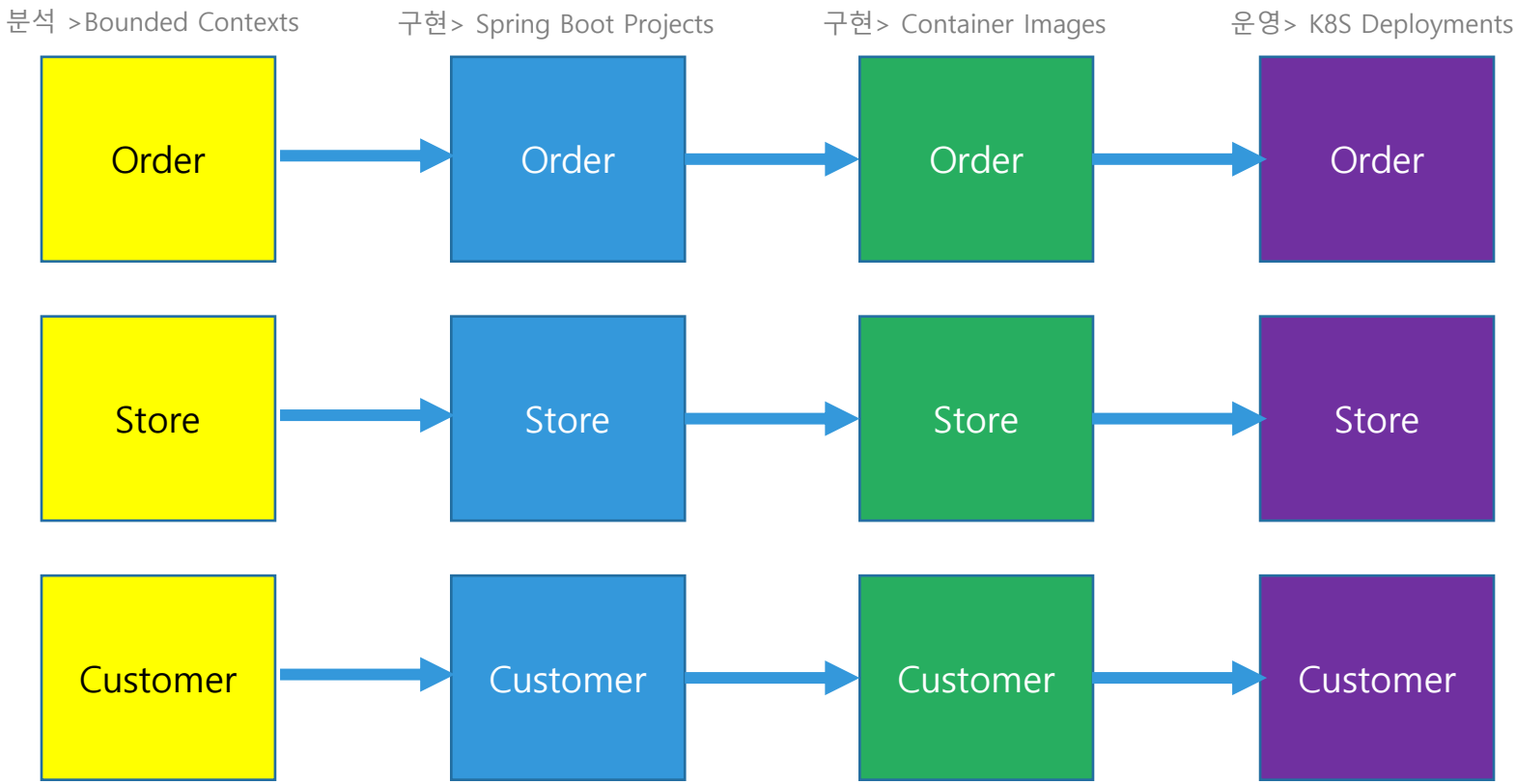
- Client-Server Communication

  API Gateway

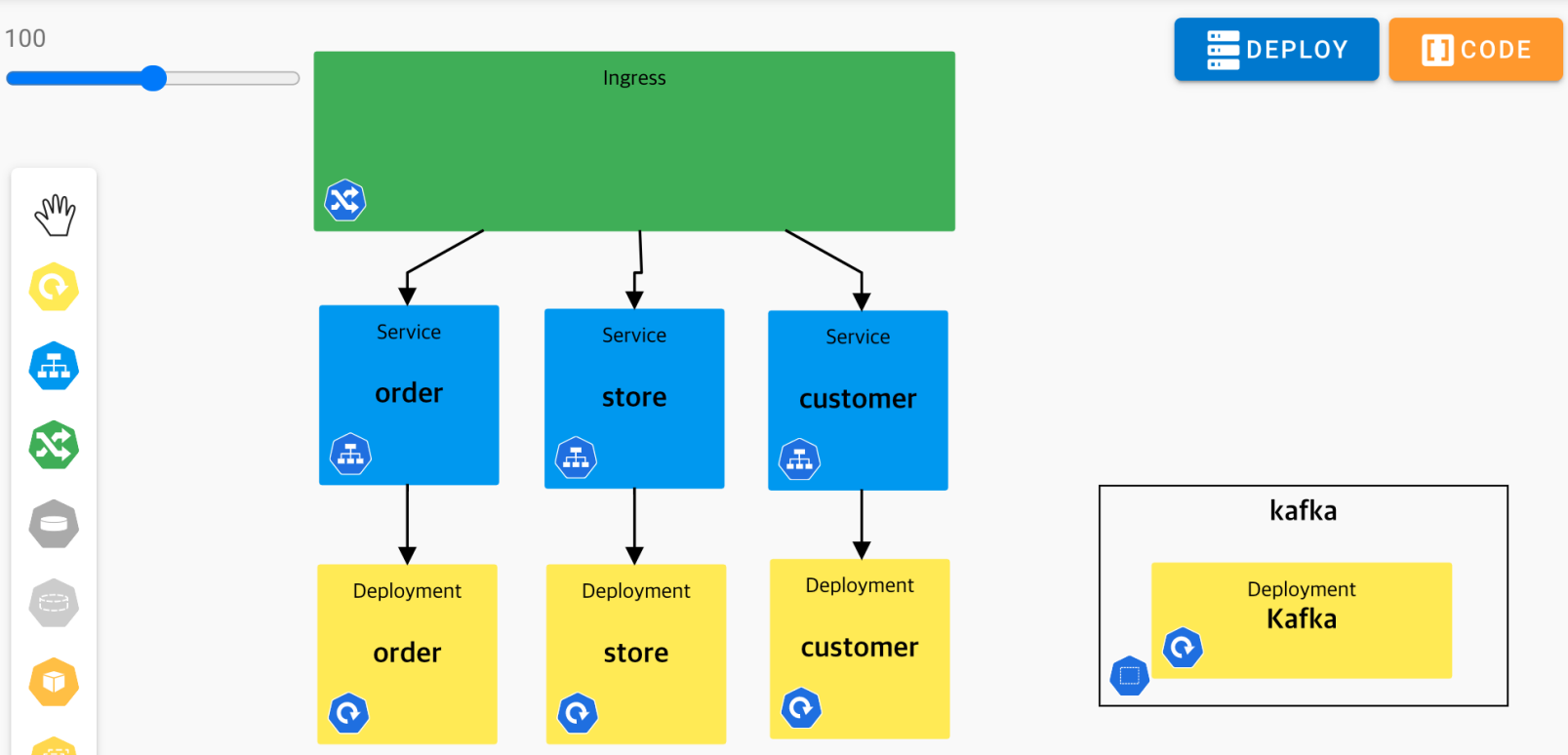
  Front-end Development

운영

Biz-Dev-Ops


분석 >Bounded Contexts
구현> Spring Boot Projects
구현> Container Images
운영> K8S Deployments
Order
Order
Order
Order
Store
Store
Store
Store
Customer
Customer
Customer
Customer

K8S deploy model for Food-Delivery



지속적 개선

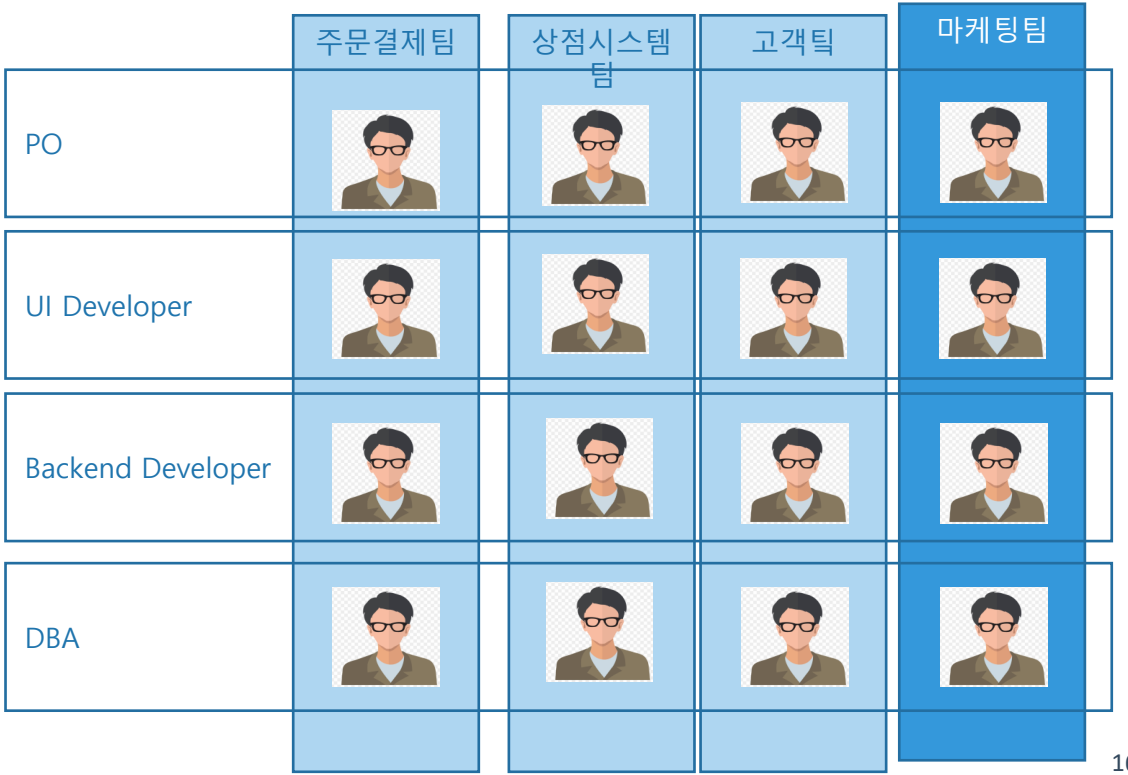
간섭없는 개발 조직의 추가

마케팅팀의 추가

KPI:
신규고객유
입률

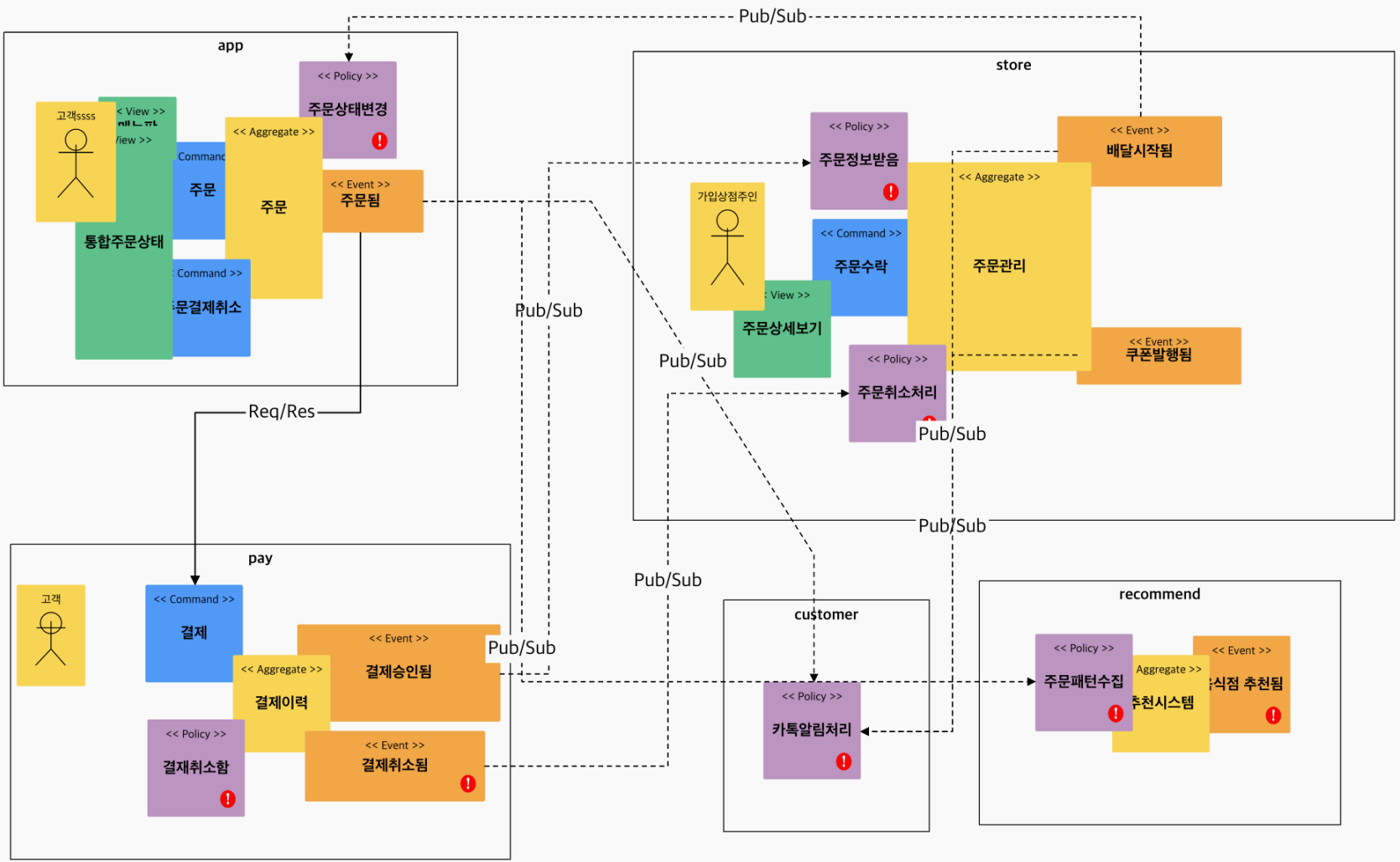
CEO

CTO

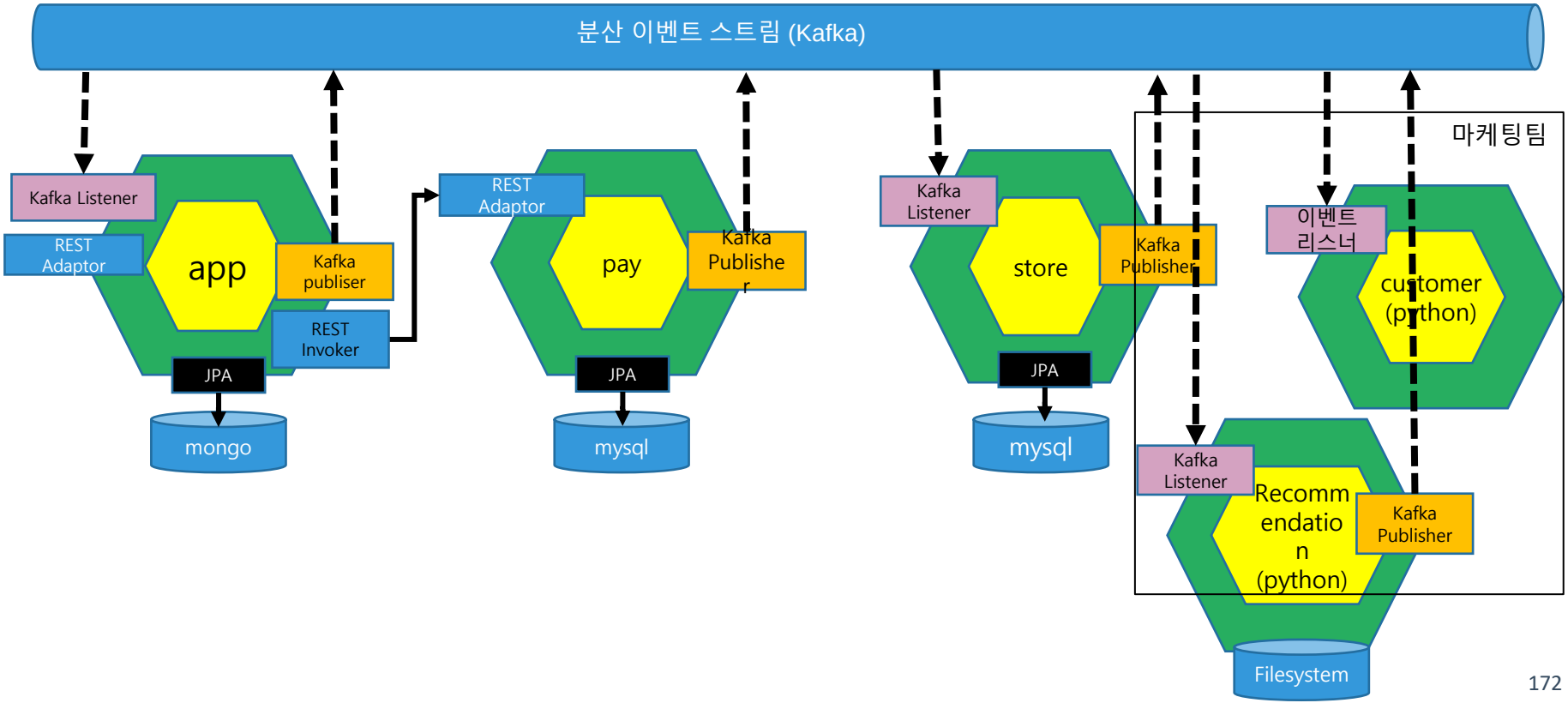


시나리오 – 마케팅 팀

- 우수고객에게 쿠폰을 발행한다
- 자주 주문하는 상품에 대한 유사 제품을 홍보한다


Pub/Sub
app
고객ssss
< View >>
Command
주문
<< Aggregate >>
주문
<< Policy >>
주문상태변경
<< Event >>
주문됨
통합주문상태
Command >>
문결제취소
Pub/Sub
Req/Res
store
<< Policy >>
주문정보받음
<< Event >>
배달시작됨
가입상점주인
<< Aggregate >>
주문관리
<< Command >>
주문수락
View >>
주문상세보기
<< Event >>
쿠폰발행됨
Pub/Sub
<< Policy >>
주문취소처리
Pub/Sub
Pub/Sub
pay
고객
<< Command >>
결제
<< Event >>
결제승인됨
Pub/Sub
<< Aggregate >>
결제이력
<< Policy >>
결재취소함
<< Event >>
결제취소됨
customer
<< Policy >>
카톡알림처리
recommend
<< Policy >>
주문패턴수집
Aggregate >>
추천시스템
<< Event >>
식점 추천됨

헥사고날 아키텍처


분산 이벤트 스트림 (Kafka)
Kafka Listener
REST Adaptor
app
Kafka publiser
REST Invoker
JPA
mongo
REST Adaptor
pay
Kafka Publisher
JPA
mysql
Kafka Listener
store
Kafka Publisher
JPA
mysql
마케팅팀
이벤트 리스너
customer (python)
Kafka Listener
Recommendation (python)
Kafka Publisher
Filesystem

THANKS!

Any Question?

You can find me at:

jyjang@uengine.org
https://github.com/jinyoung
https://github.com/TheOpenCloudEngine

Recommended Books

- **Overall MSA Design patterns:**
  https://www.manning.com/books/microservices-patterns

- **Microservice decomposition strategy:**
  - DDD distilled: https://www.oreilly.com/library/view/domain-driven-design-distilled/9780134434964/
  - Learning DDD: https://www.amazon.com/Learning-Domain-Driven-Design-Aligning-Architecture/dp/1098100131
  - Event Storming: https://leanpub.com/introducing_eventstorming

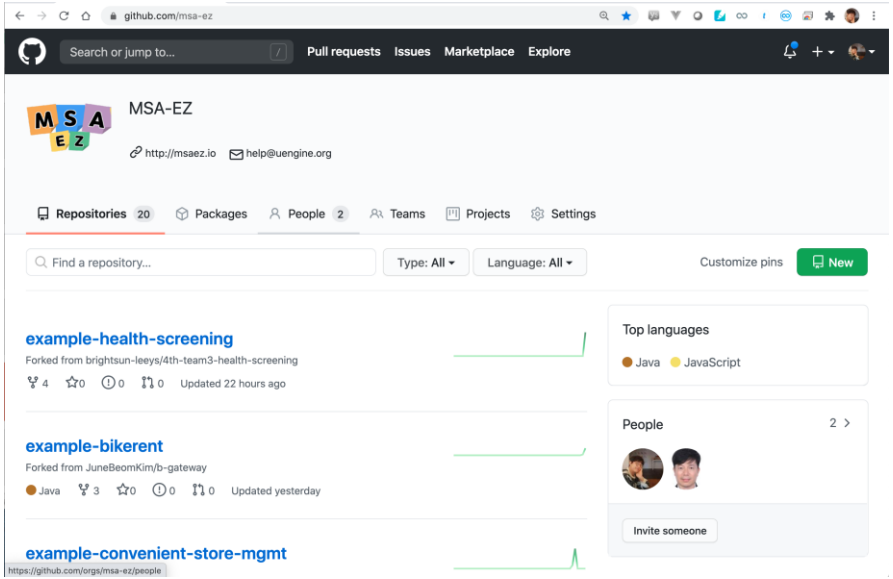
- **API design and REST:**
  http://pepa.holla.cz/wp-cont.../.../2016/01/REST-in-Practice.pdf

- **Database Design in MSA:**
  - Lightly:
- https://www.confluent.io/wp-content/uploads/2016/08/Making_Sense_of_Stream_Processing_Confluent_1.pdf
  - Deep dive:
- https://dataintensive.net/?fbclid=IwAR3OSWkhqRjLI9gBoMpbsk-QGxeLpTYVXIJVCSaw_A5eYrBDc0piKSm4pMM

github.com/msa-ez

Reference MSA Projects

github.com/msa-ez



MSA School

msaschool.io