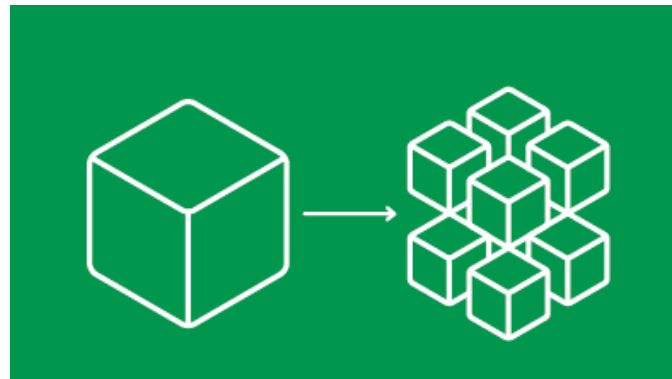




# Microservices


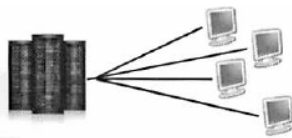

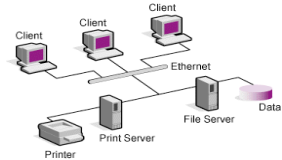
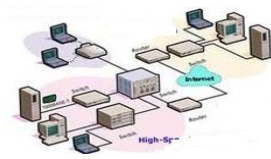



2018.11  
김영기 수석  
resious@gmail.com

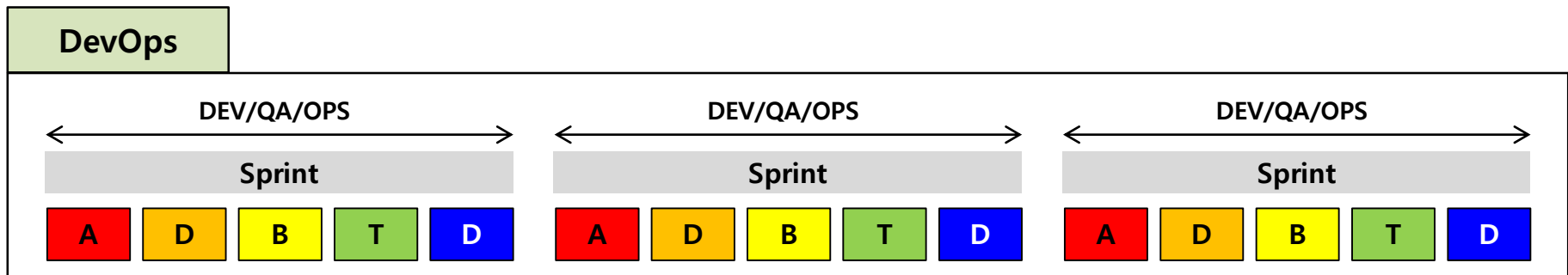
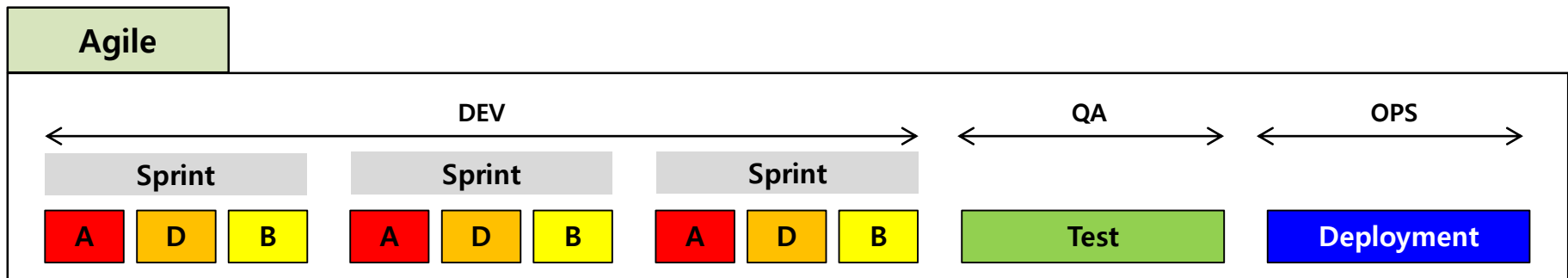
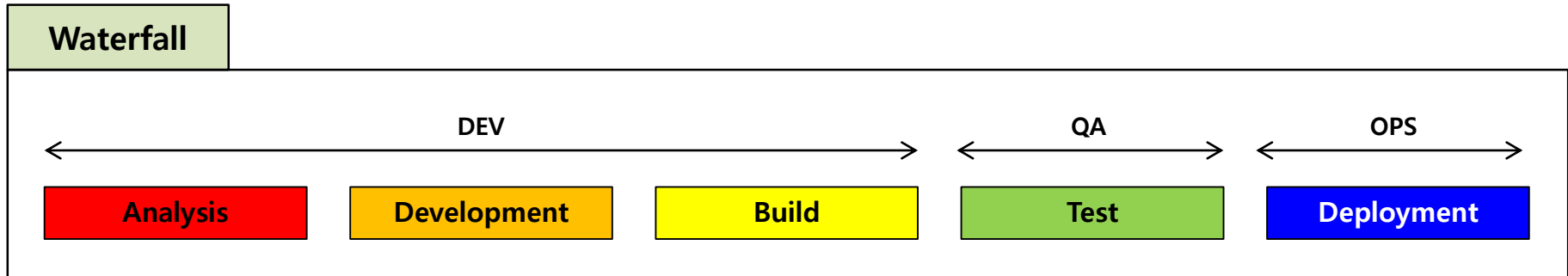


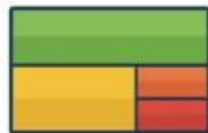
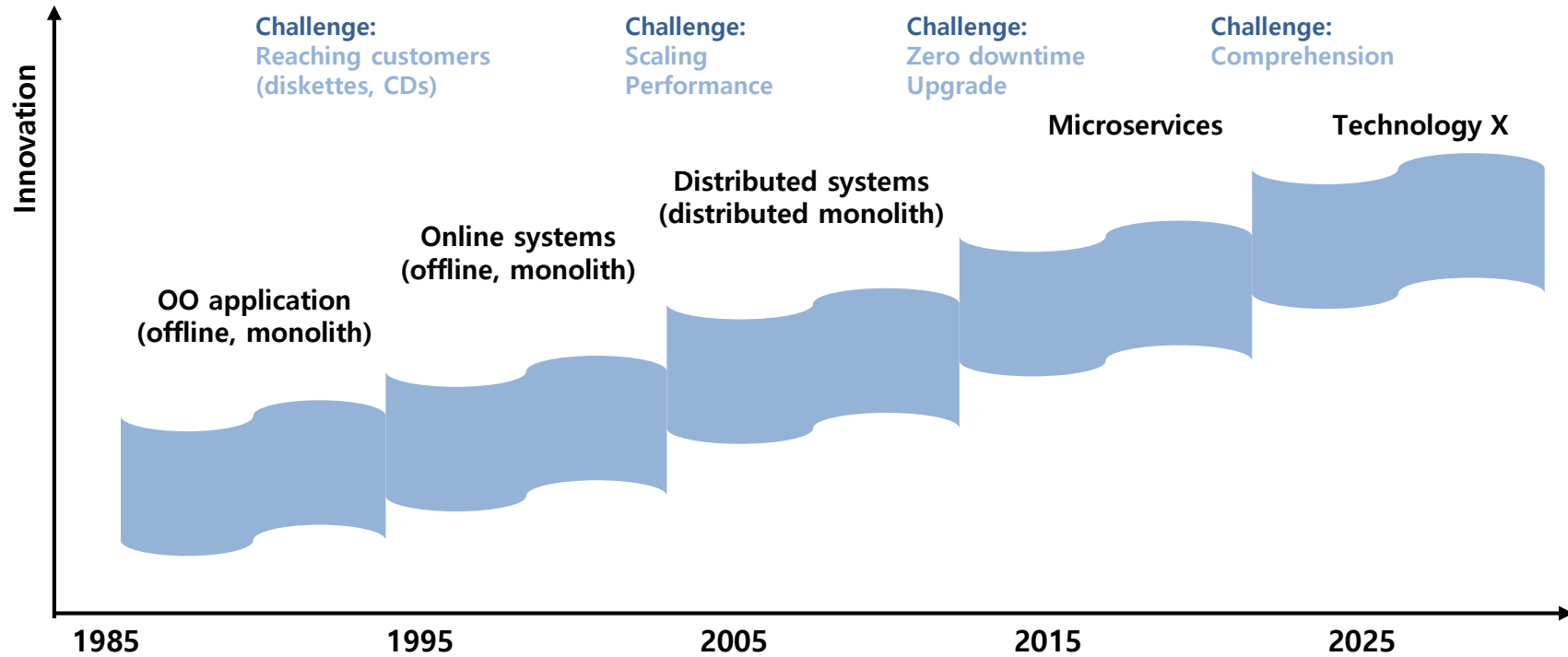
0	Warm-up
1	마이크로서비스의 정의
2	마이크로서비스의 구조
3	마이크로서비스 사용 이유 및 문제점
4	마이크로서비스의 통합 및 통신
5	마이크로서비스 개발 환경 및 테스트
6	MSA 사례

## ❖ IT 인프라스트럭처의 진화

<b>Electronic accounting machine</b>	<b>1930 ~ 1950</b>	<ul style="list-style-type: none"> <li>데이터 처리를 위한 크고 복잡한 컴퓨터 사용</li> <li>하드 와이어드 소프트웨어</li> </ul>	
<b>General-Purpose Mainframe and minicomputer</b>	<b>1950 ~ 현재</b>	<ul style="list-style-type: none"> <li>1958: IBM 메인프레임</li> <li>1965: DEC 미니컴</li> </ul>	
<b>Personal computer</b>	<b>1981 ~ 현재</b>	<ul style="list-style-type: none"> <li>1981: IBM PC</li> <li>80~90년대 PC 및 SW 급증</li> </ul>	
<b>Client Server</b>	<b>1983 ~ 현재</b>	<ul style="list-style-type: none"> <li>서버와 클라이언트의 네트워킹을 통한 작업 분할</li> <li>N-tier</li> <li>다양한 유형의 서버: 네트워크, 애플리케이션, Web</li> </ul>	
<b>Enterprise Internet</b>	<b>1992 ~ 현재</b>	<ul style="list-style-type: none"> <li>분산 네트워크</li> <li>인터넷 표준 및 엔터프라이즈 애플리케이션</li> </ul>	
<b>Cloud and mobile computing</b>	<b>2000 ~ 현재</b>	<ul style="list-style-type: none"> <li>컴퓨팅 및 애플리케이션이 인터넷이나 네트워크를 통해 제공</li> <li>가장 빠르게 증가하는 컴퓨팅 형태</li> </ul>	

## ❖ Waterfall → Agile → DevOps





Monolith  
Single unit

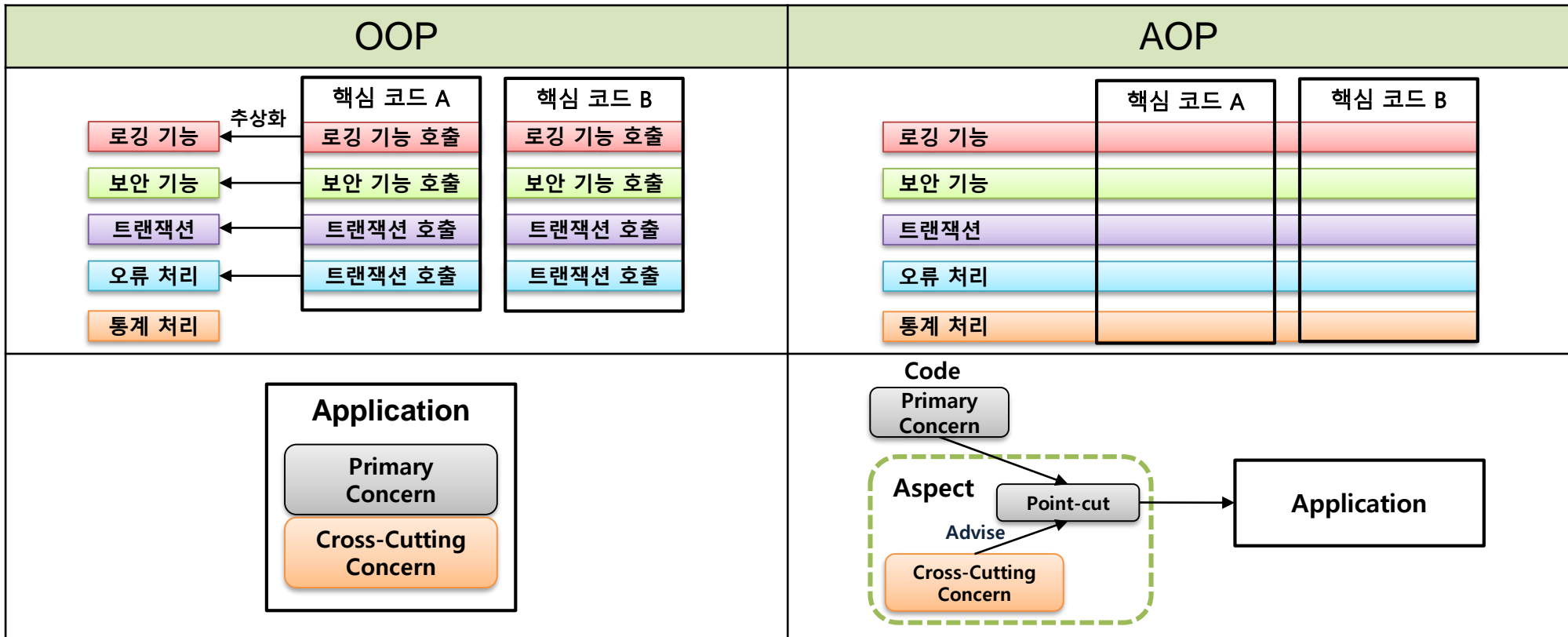


SOQ  
Coarse-grained



Microservices  
Fine-grained

**AOP**(Aspect Oriented Programming)은 기능을 핵심 비즈니스 로직과 공통 모듈로 구분하고, 핵심 로직에 영향을 주지 않으면서 로직 사이 사이에 공통 모듈을 효과적으로 삽입해 개발하는 방법이다. 코드 외부에서 공통 모듈을 만든 후, 공통 모듈을 비즈니스 로직에 삽입하는 것이 핵심이다.



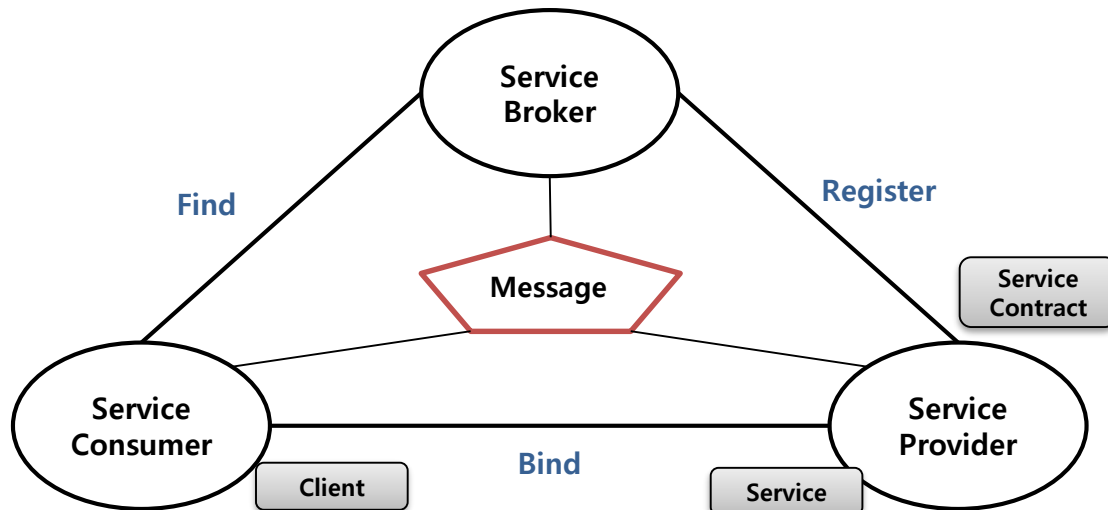
Primary(Core) Concern: 비즈니스 로직을 구현한 부분  
 Cross-Cutting Concern: 부가적인 기능으로 시스템 전반에 산재되어 사용되는 기능  
 Code: primary Concern으로 구현해 놓은 코드

Join Point: 횡단 관심 모듈 기능이 삽입되어 동작할 수 있는 실행 가능한 특정 위치  
 Point-Cut: 어느 클래스의 Join Point를 사용할 것인지 결정하는 선택 기능  
 Advise: 각 Join-Point에 삽입되어 동작할 수 있는 코드

## ❖ Service Oriented Architecture ...

- 서비스 요청자(Client)와 제공자로 이루어진 애플리케이션 설계 방식
- 표준 기반의 공통 모듈(재사용)로 가능한 서비스들을 느슨한 관계 모델링을 통해 소프트웨어의 서비스화를 지향하는 아키텍처

Service Orientation	Architecture
<ul style="list-style-type: none"> <li>▪ Open 된 상호 호환성을 갖는 프로토콜을 사용</li> <li>▪ 애플리케이션 서비스 Description과 이를 지원하는 Dynamic Discovery 시스템을 통해 조합 가능한 환경</li> </ul>	<ul style="list-style-type: none"> <li>▪ 전체적인 목표를 달성하기 위해 컴포넌트를 조합하는 프로세스</li> <li>▪ Layer에 의해 구성되는 컴포넌트들을 통해 이루어지는 Blueprint로 컴포넌트간 특성 및 관계, 상호작용, 제약 사항을 포함</li> </ul>

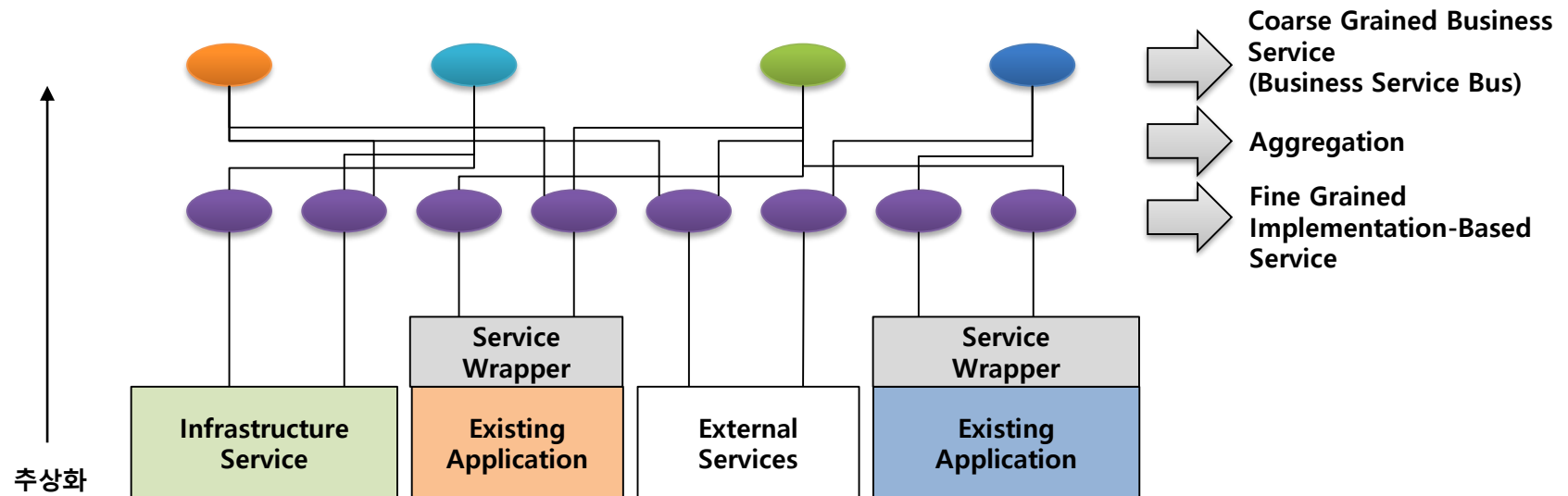


자기 설명적인 인터페이스를 가진 서비스는 플랫폼 독립적인 XML 형식으로 정의되어, Service Broker를 통해 서비스로 등록되고 관리되며, 애플리케이션이 증가해도 서비스를 찾을 수 있는 방법이 제공된다.

## ❖ 서비스



- 논리적으로 하나의 단위를 이루는 업무를 자체적으로 처리할 수 있는 SW 컴포넌트
  - ✓ 플랫폼 독립적, 동적 검색 및 호출, Self-contained
- 비즈니스 업무의 논리 단위를 구현한 것으로, 개방형 인터페이스로 설계되어 다른 서비스에서 접근이 가능한 소프트웨어 컴포넌트



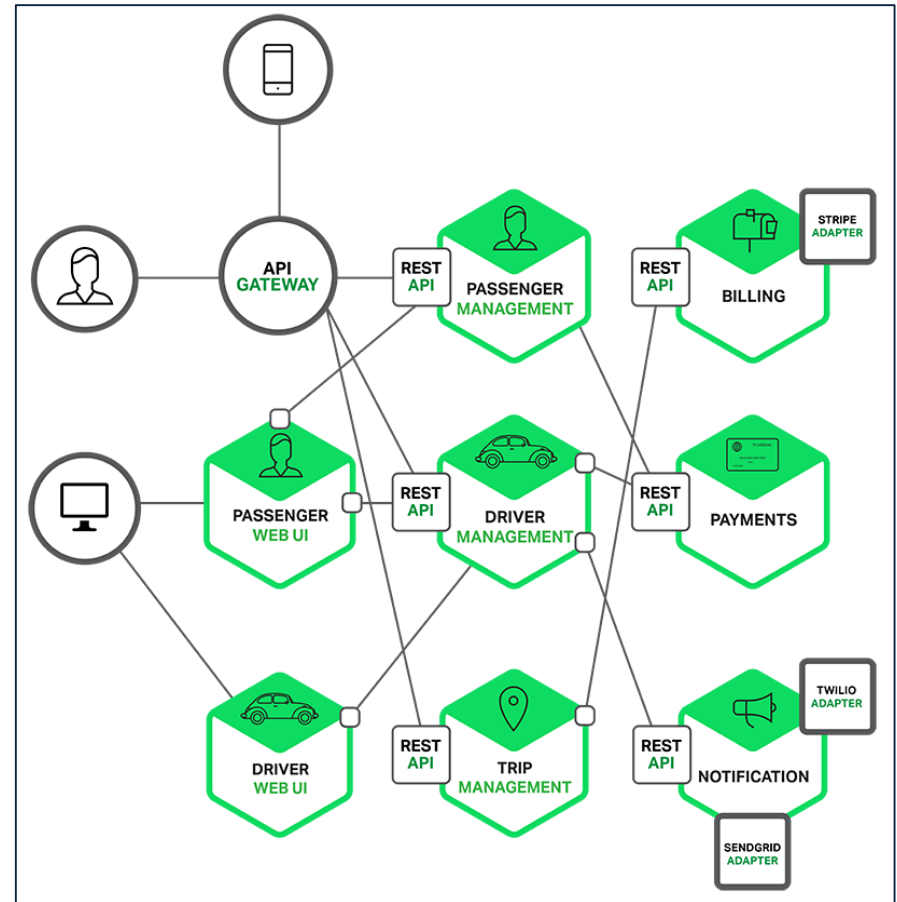
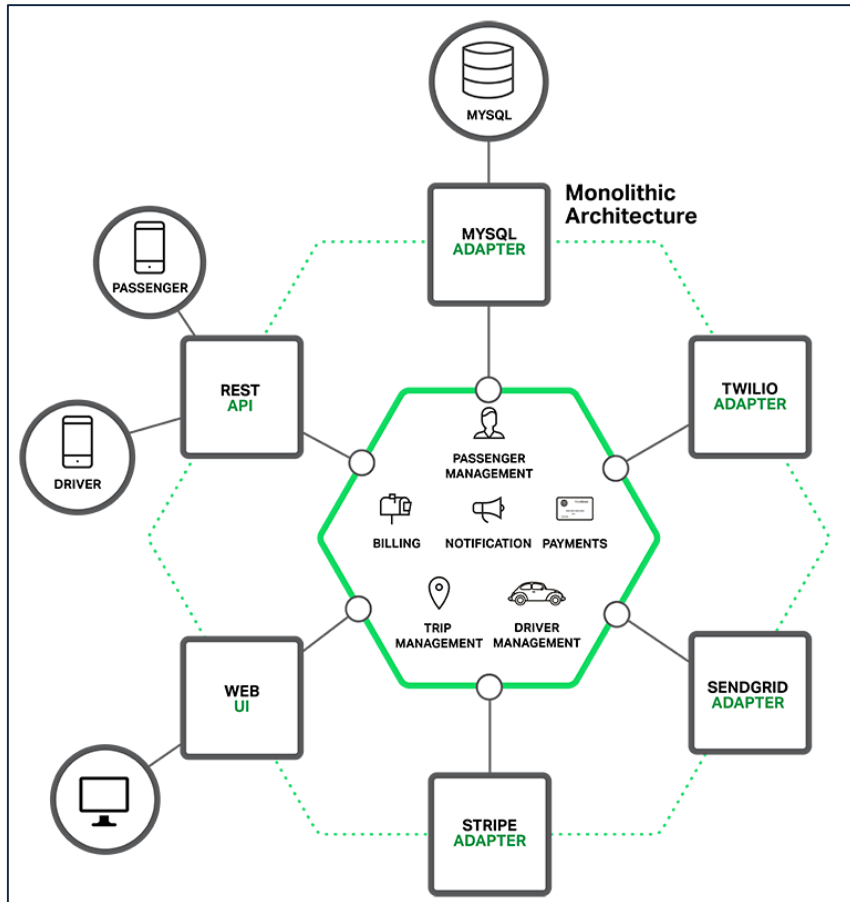




# 마이크로서비스의 정의



# Monolithic Architecture vs. MSA



## ❖ Microservices is ...

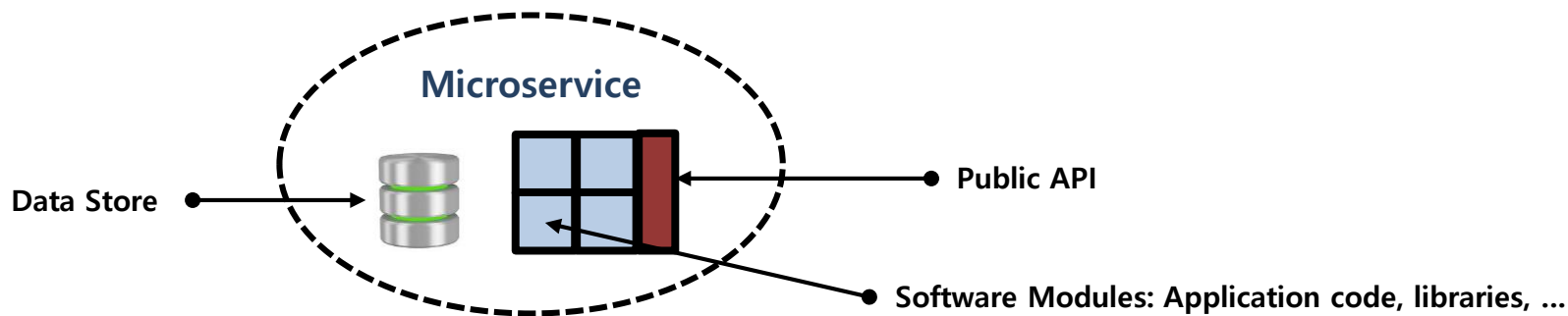
A software architecture style in which complex applications are composed of small, Independent process communicating with each other using language-agnostic APIs.

These services are small, highly decupled and focus on doing a small task, facilitating a modular approach to system-build

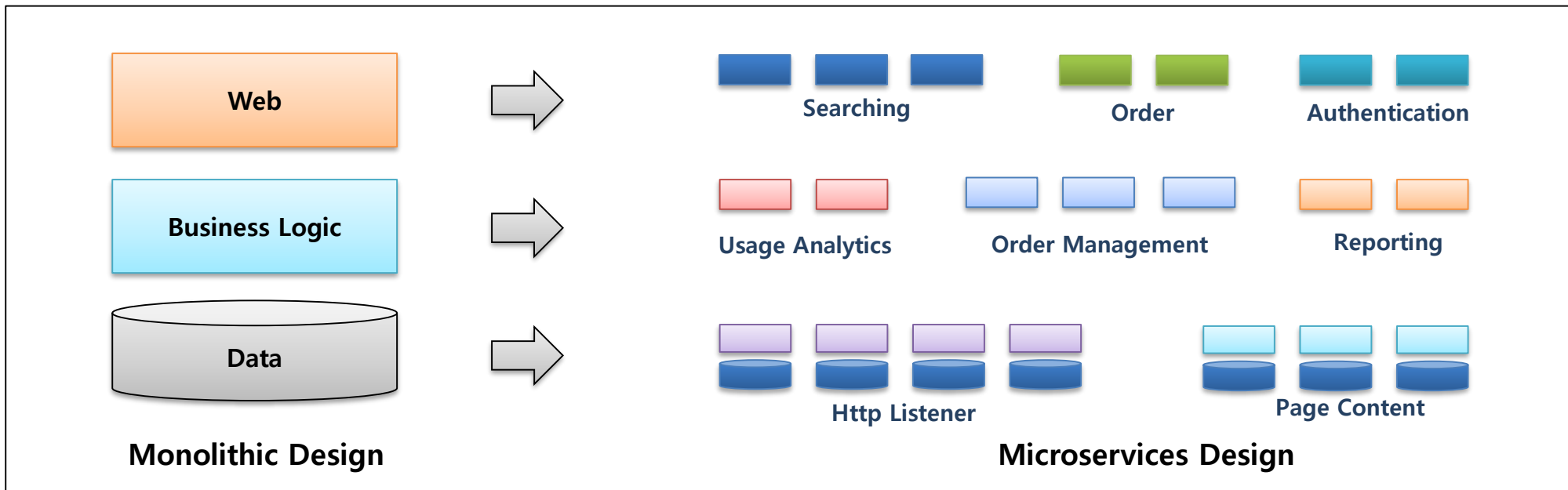
- Wikipedia

서비스 디자인 스타일로서 작은 서비스의 결합을 통해 하나의 응용프로그램을 개발하는 방법으로, 각각의 서비스는 독립적인 로직으로 구성되며, 완전히 자동화 된 개발/배포환경에 의해 각각 독립적으로 배포될 수 있다. 최소한의 중심적인 관리 체계가 있으며, 이 시스템은 각각 다른 프로그래밍 언어, 다른 데이터 스토리지 기술로 작성이 가능하다.

- Martin Fowler at ThoughWorks



Microservices = Fine-grained service-oriented architecture + “small” public API



Monolithic Style	Microservices
Single Unit with all components interwoven	Applications are suite of service
Overall Scaling of the structure required	Each service independently scalable
Overall deployment	Independently deployable
Increases complexity for developers	More developer friendly
One errant component crashes the whole system	Fault diagnosis and isolation is easier
Testing is streamlined	Difficult to test

## ❖ 마이크로서비스의 크기에 영향을 주는 요소들



마이크로서비스의  
이상적인 크기 ?



팀 크기 (조직 구조)

마이크로서비스는 여러 팀의 작업이 필요한 만큼 커져서는 안된다  
팀들은 독립적으로 작업하고 소프트웨어를 출시 할 수 있어야 한다

모듈화

개발자가 이해하고 추가적인 개발을 허용하는 크기가 좋다

교체 가능성

교체 가능성은 마이크로서비스의 크기를 감소시킨다

트랜잭션과 일관성

트랜잭션과 일관성은 하나의 마이크로서비스에서만 보장된다

인프라스트럭처

필요한 인프라 제공이 힘든 경우, 마이크로서비스의 개수가 제한된다

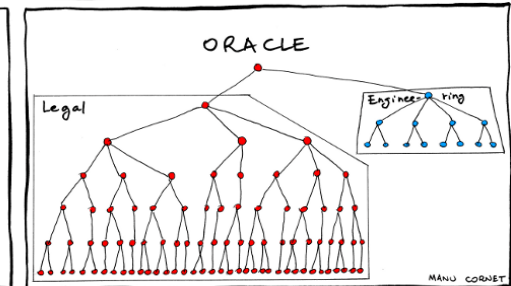
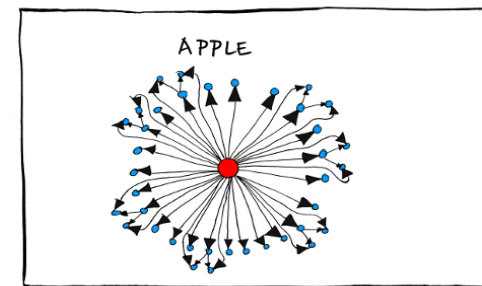
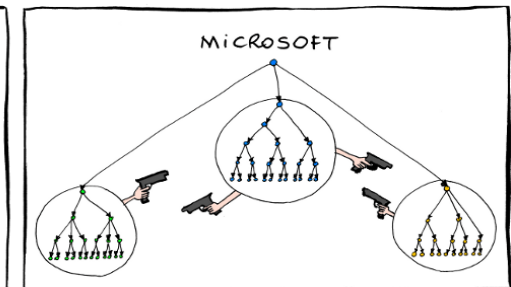
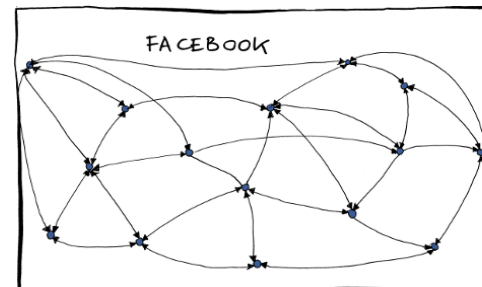
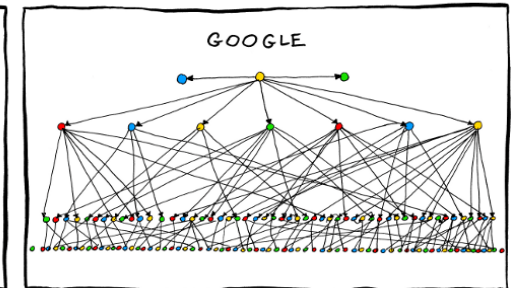
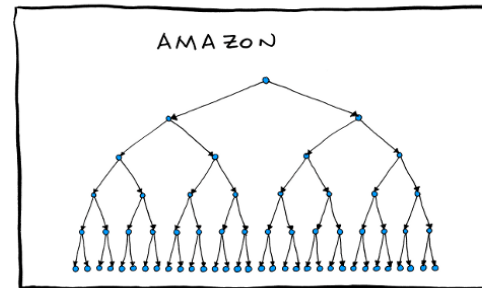
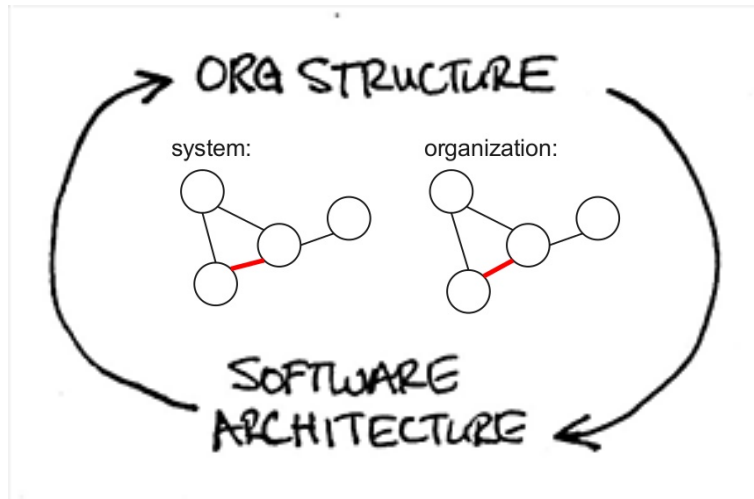
분산 통신

마이크로서비스가 증가하면 분산 통신이 증가한다



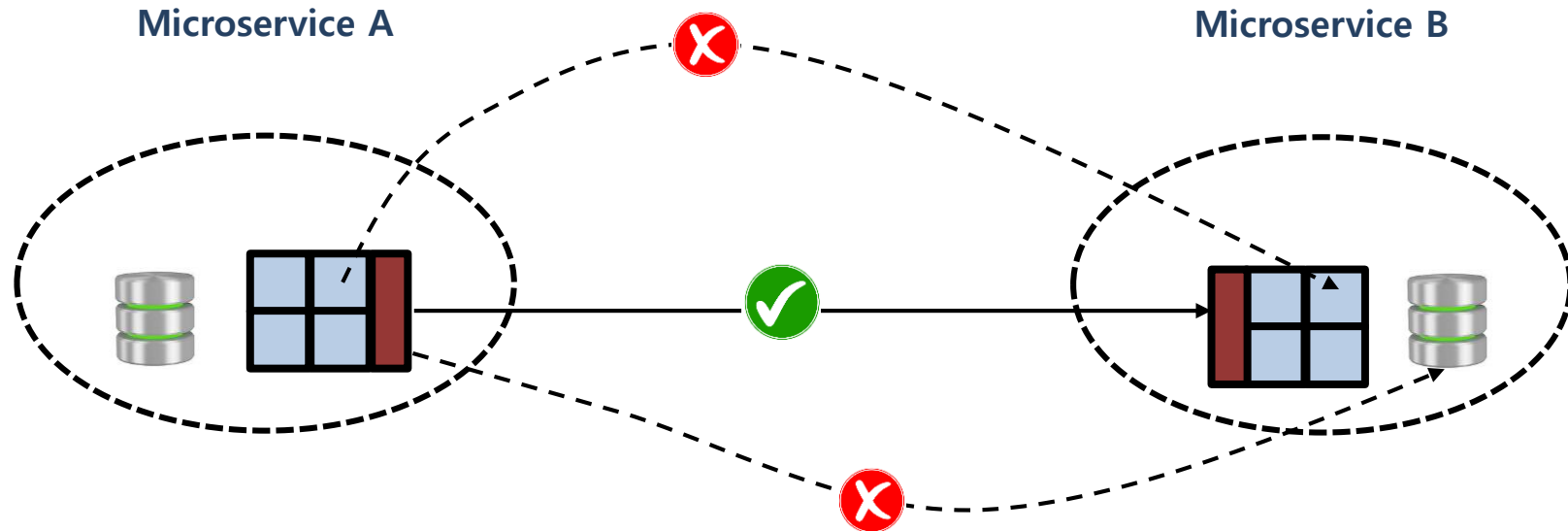
Organizations which design system (in the board sense used here) are constrained to produce design which are copies of the communications structures of these organizations

- Melvin Conway, Datamation, 1968



## ❖ Only rely on each other's public API

- 모든 팀은 서비스 인터페이스를 통해 자신들의 데이터와 기능을 노출한다
- 팀들은 반드시 인터페이스를 통해 서로 통신한다
- 모든 서비스 인터페이스는 예외 없이 처음부터 노출이 가능하도록 설계되어야 한다





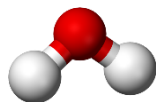
# 마이크로서비스의 구조





## ❖ Microservices Architecture

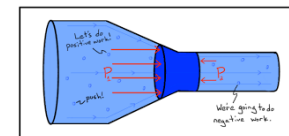
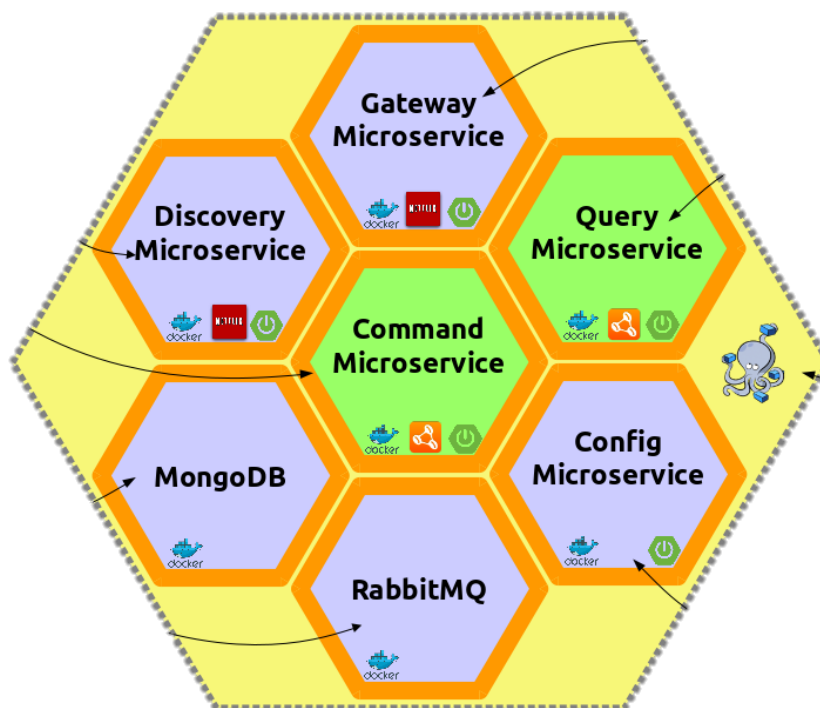
- 서비스를 기능별로 작게 분할한 서버 아키텍처의 디자인 패턴
  - ✓ Inner Architecture – 개발 마이크로서비스의 아키텍처
  - ✓ Outer Architecture – 마이크로서비스로 구성된 시스템의 아키텍처



미시적 관점

### Inner Architecture

- Single Purpose
- Loosely coupled
- Independently deployable
- Independently disposable
- Focused Business Capabilities
- Decentralized Data Management
- Heterogeneous Technology
- Logging & Monitoring
- Stateless



거시적 관점

### Outer Architecture

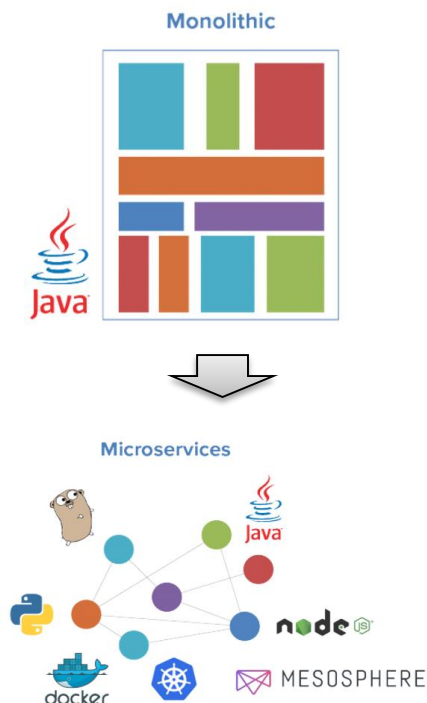
- Managed APIs
- Expose services
- Route messages
- Asynchronous communication
- Load Balancing
- Service Discovery
- Shared Configuration
- Dependency management
- Transaction Analysis
- Logging & Monitoring

## ❖ MSA 구현

- 일반적으로 아래와 같은 4단계를 거침

1

기능별로 서비스  
애플리케이션 분리

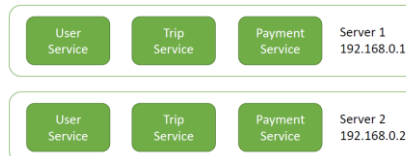


하나의 서비스는  
한 가지 일에 초점을 맞춘다

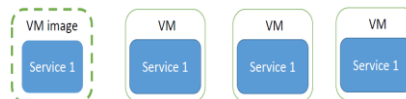
2

서비스 이미지 개발

① 서버마다 전체 서비스 실행



② 서비스를 VM에서 실행



③ 서비스를 컨테이너에서 실행



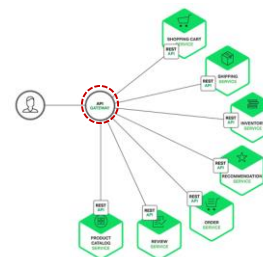
각 서비스는 적절한 자원을  
제공 받아야 하며, 배포가 빠  
르고 효율적이어야 한다

3

클라이언트에서  
필요한 서비스 선택

① API Gateway

- 비동기/논블로킹 I/O



② IPC

- 동기(메시지 기반)  
- 비동기(Request)



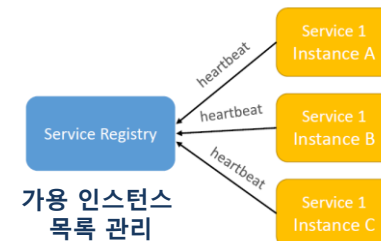
너무 많은 요청은  
비효율적이며, 클라이언트 코  
드가 복잡해진다

4

인스턴스  
선택 및 관리

하나의 서비스는 여러 인스턴스로  
실행되고, 인스턴스들의 IP가 변함

➡ Service Discovery



① Client Side Discovery Pattern

② Server Side Discovery Pattern  
- 로드 밸런서가 존재

서비스 =  $\Sigma$  가상화 인스턴스  
서비스 검색 패턴을  
정의해야 한다

## Monolithic



## Microservices








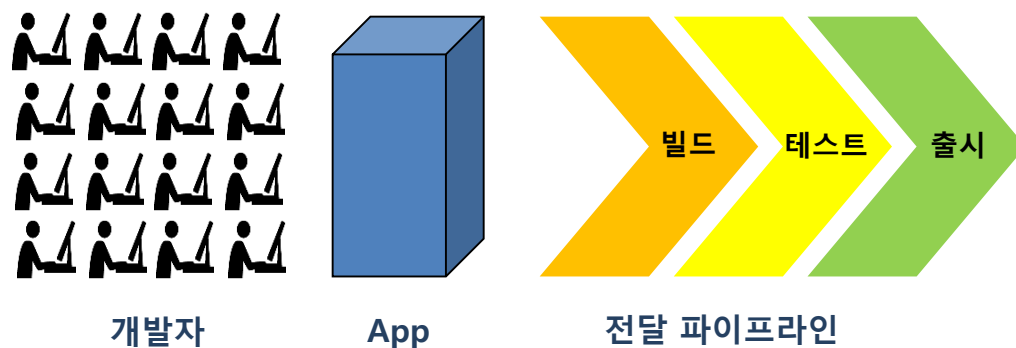
## 마이크로서비스 사용 이유 및 문제점



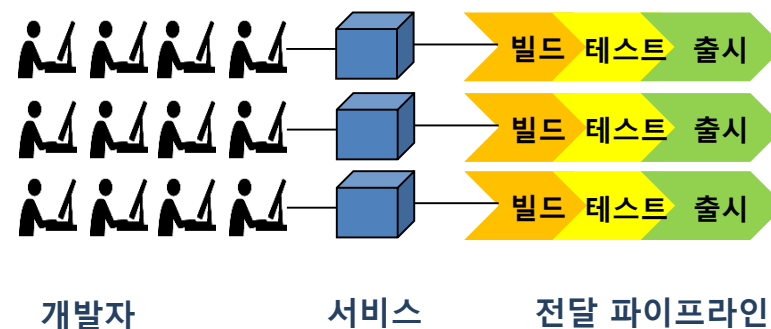
## ❖ 마이크로서비스 사용 시 혜택

기술 측면	조직 측면	비즈니스 측면
<ul style="list-style-type: none"> <li>교체 가능성</li> <li>지속 가능한 개발</li> <li>레거시 처리</li> <li>지속적인 통합 활용</li> <li>확장성</li> <li>견고성</li> <li>자유로운 기술 선택</li> <li>독립성</li> </ul> 	<ul style="list-style-type: none"> <li>팀의 독립성</li> <li>자기 조직화</li> <li>소규모 프로젝트</li> </ul> 	<ul style="list-style-type: none"> <li>병렬 기능 개발</li> <li>애자일 프로세스의 확장</li> <li>과도한 조정이나 소통을 요구하지 않음</li> </ul> 

모놀리식 개발 수명주기

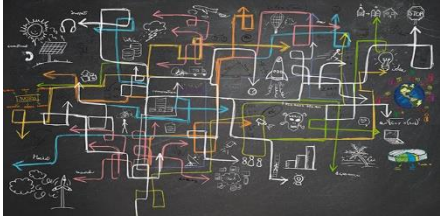


마이크로서비스 개발 수명주기

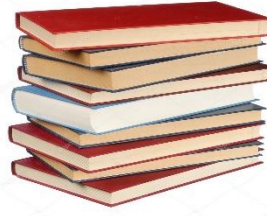


## ❖ 마이크로서비스 개발 시 고려사항

장애진단



다양한 기술에 대한 표준 관리



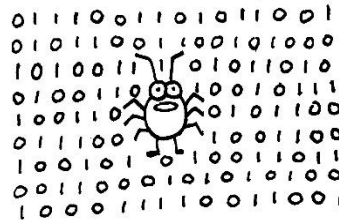
API를 통한 통신 → 성능 문제



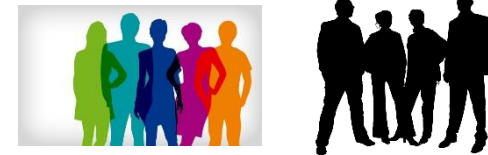
상이한 설정 및 기동 방법



테스팅



팀 역량에 따른 일정 및 품질 문제



모니터링



트랜잭션 관리



서비스 조정

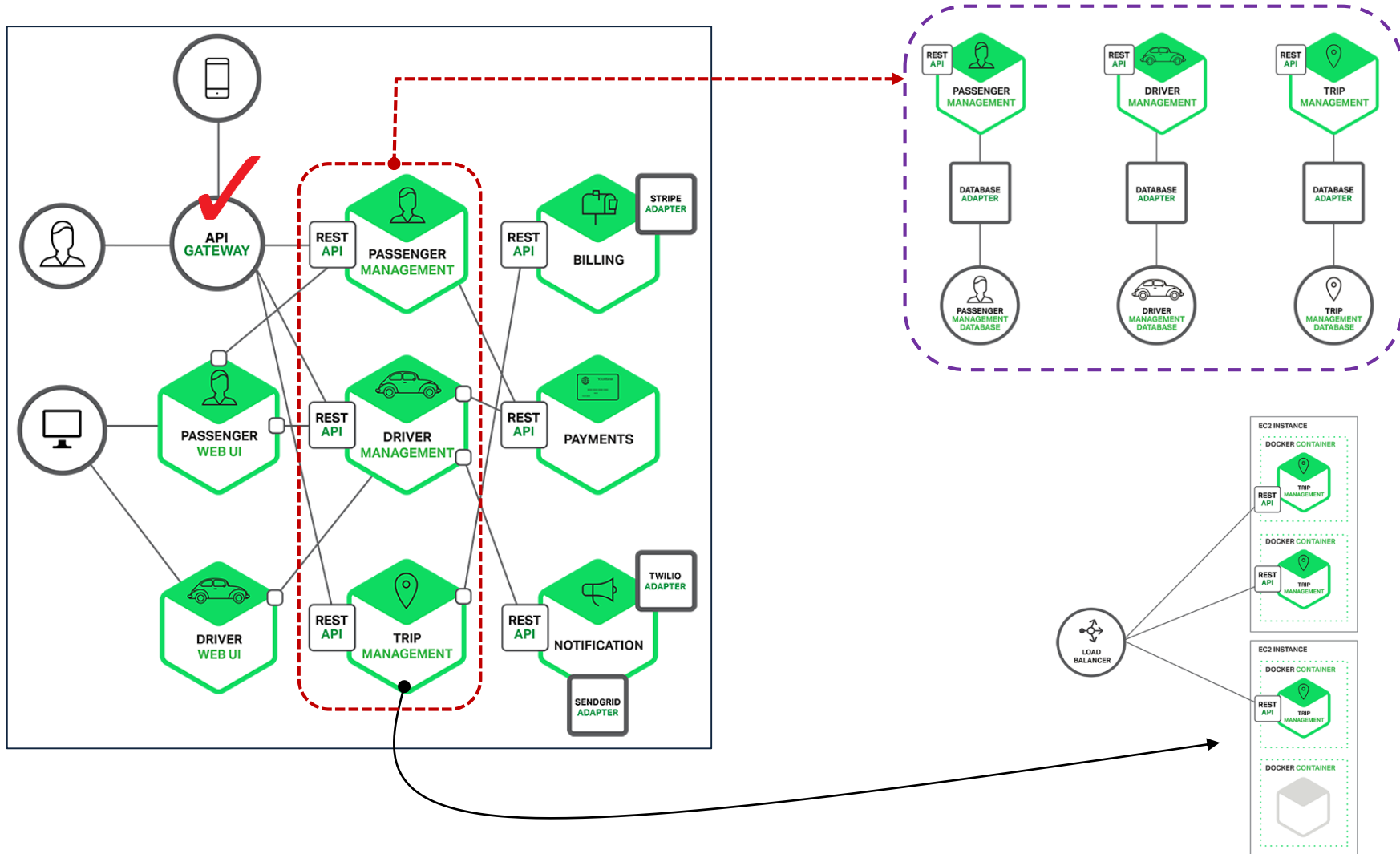




# 마이크로서비스의 통합 및 통신



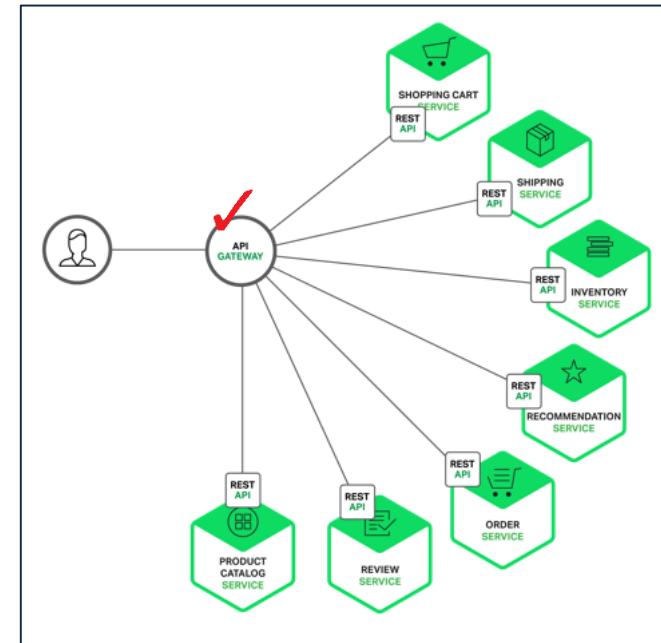
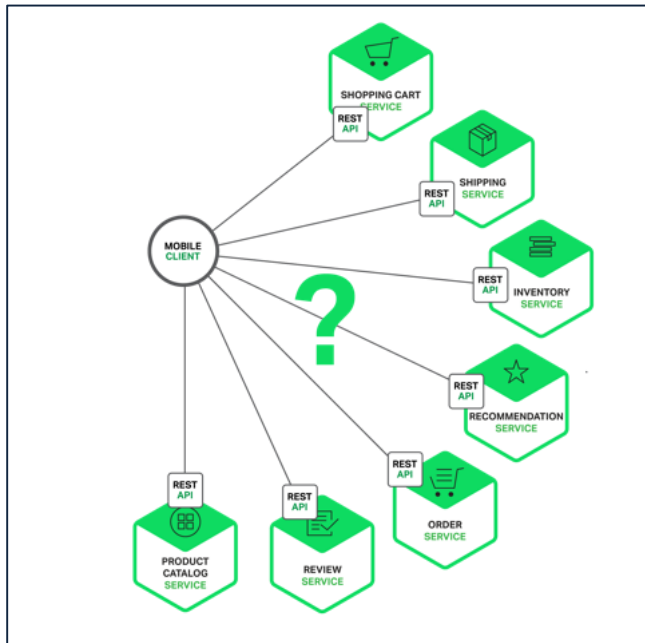
# Microservice Architecture





## ❖ API Gateway

- 요청에 따라 필요한 서비스로 라우팅
- 많은 요청을 처리하기 위해, 비동기 및 논블록킹 기반으로 설계
- API Gateway 장애 발생 시 서비스 전체가 다운
- API Gateway는 로드밸런싱, 캐싱, 모니터링 등 다양한 기능을 가질 수 있음  
→ 병목 지점이 될 가능성이 있다.

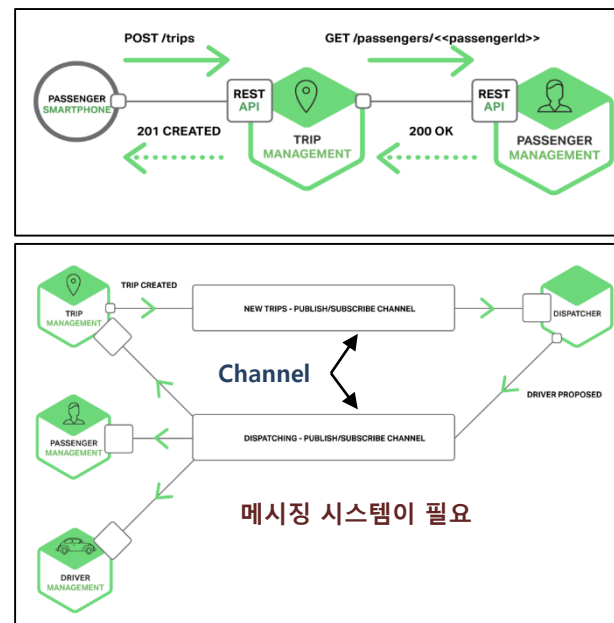


## ❖ Inter-Process Communication

- 마이크로서비스 시스템 = 분산 시스템
  - ✓ 서비스 간 통신이 필요(1:1, 1:N)

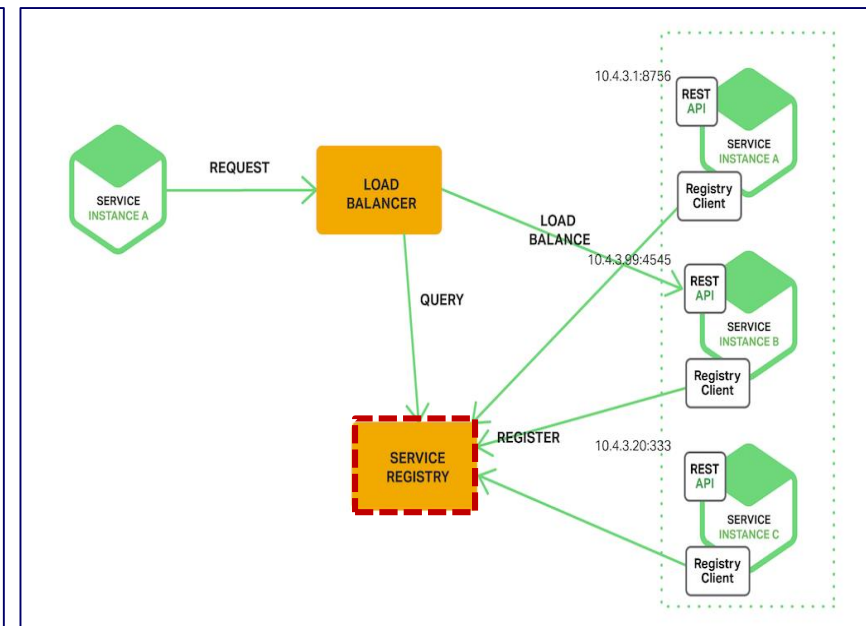
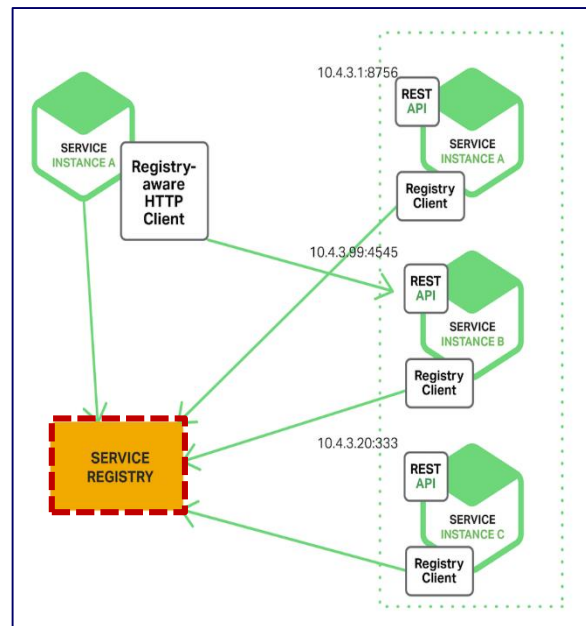
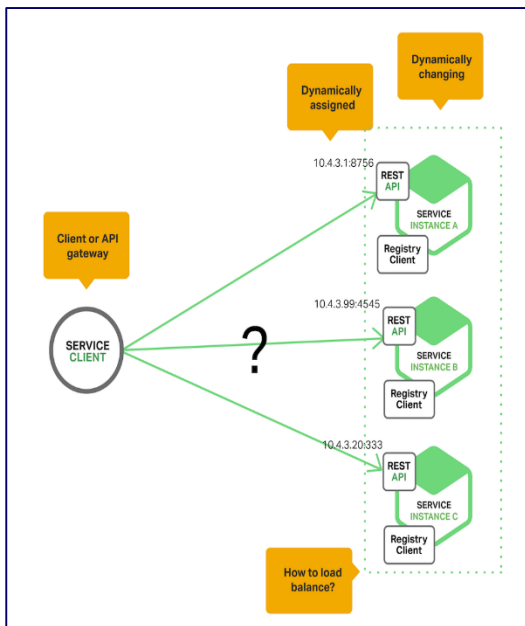
	1:1	1:N
<b>Synchronous</b>	Request/response	-
<b>Asynchronous</b>	Notification Request/async response	Publish/subscribe Publish/async response

- IPC 기술
  - ✓ 동기 요청/응답 기반 : REST, ...
  - ✓ 비동기 메시지 기반 : AMQP, ...
- 데이터 포맷
  - ✓ 텍스트 기반 : JSON, XML, ...
  - ✓ 바이너리 : Protocol Buffers, ...



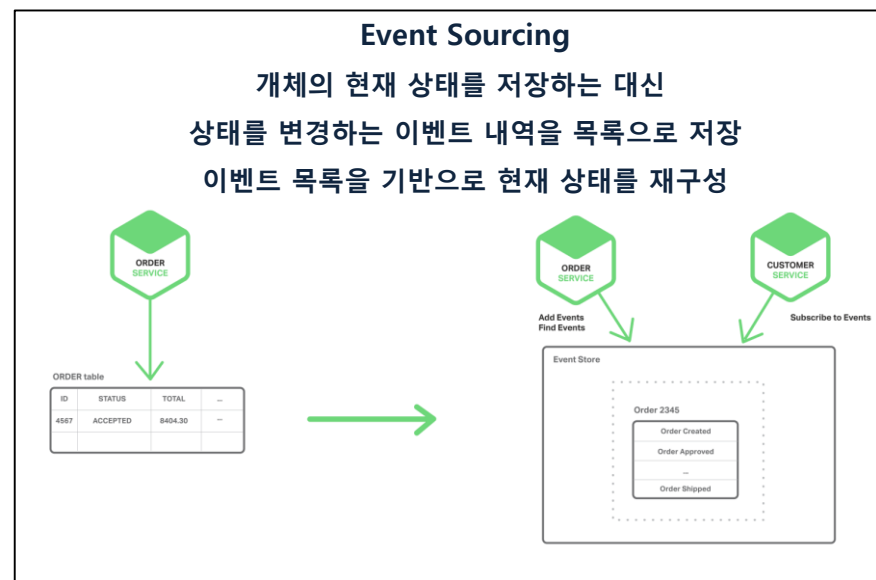
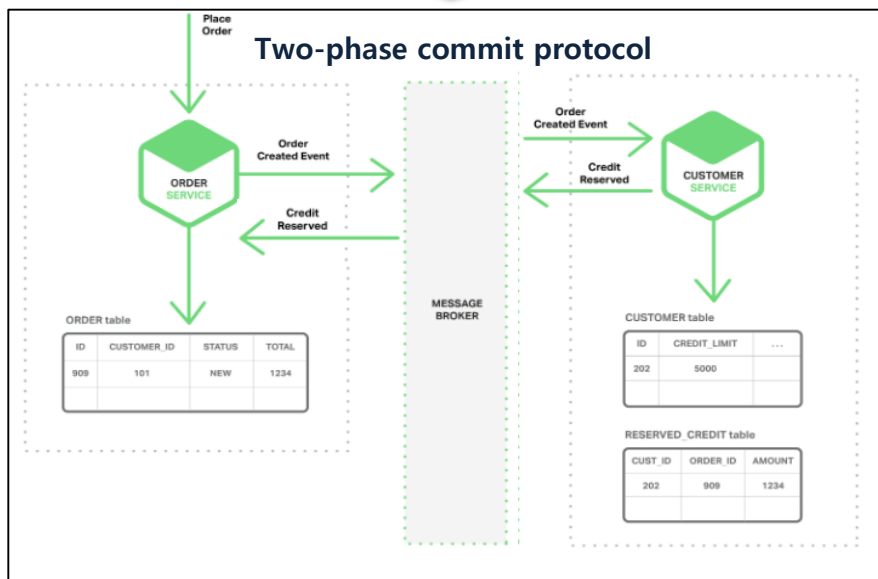
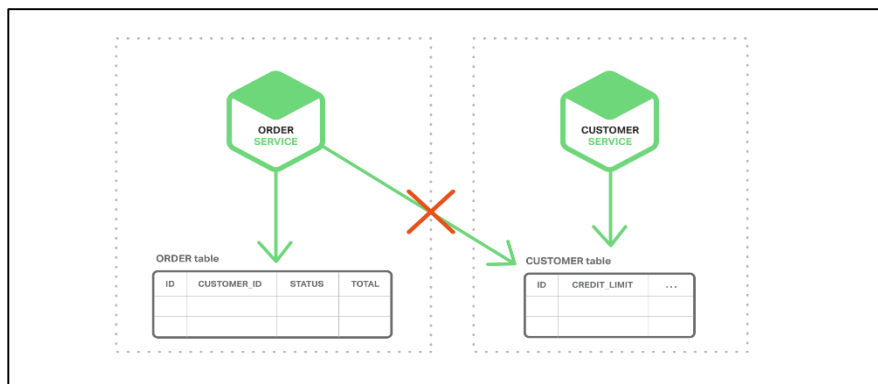
## ❖ Why ?

- Request를 보내기 위해서는 서비스 인스턴스가 네트워크 어디에 있는지 알아야 함  
→ IP와 Port 정보가 필요
- 서비스 레지스트리는 각 서비스 인스턴스의 네트워크 정보를 저장하는 데이터베이스  
→ 항상 최신 정보를 유지하고,고가용성이 필요  
→ Netflix Eureka, etcd, Consul, Apache Zookeeper  
→ 서비스 등록은 Self-Registration Pattern과 Third-Party Registration Pattern



## ❖ 분산 데이터

- CAP(Consistency, Availability, Partition tolerance)를 만족하지 못함



## CQRS(Command Query Responsibility Segregation)

명령(데이터를 변경하는 Create/Delete/Update)와  
쿼리(데이터를 조회하는 Read)를 분리



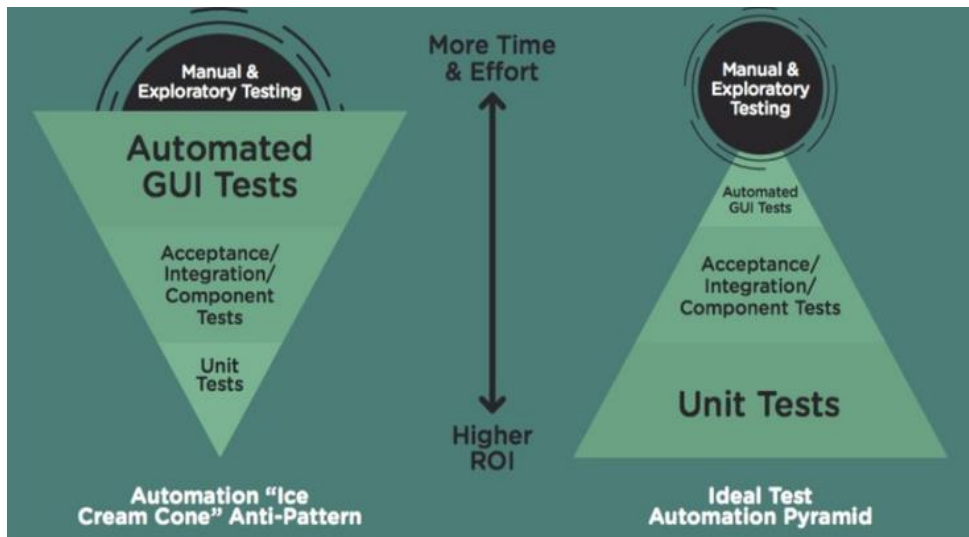
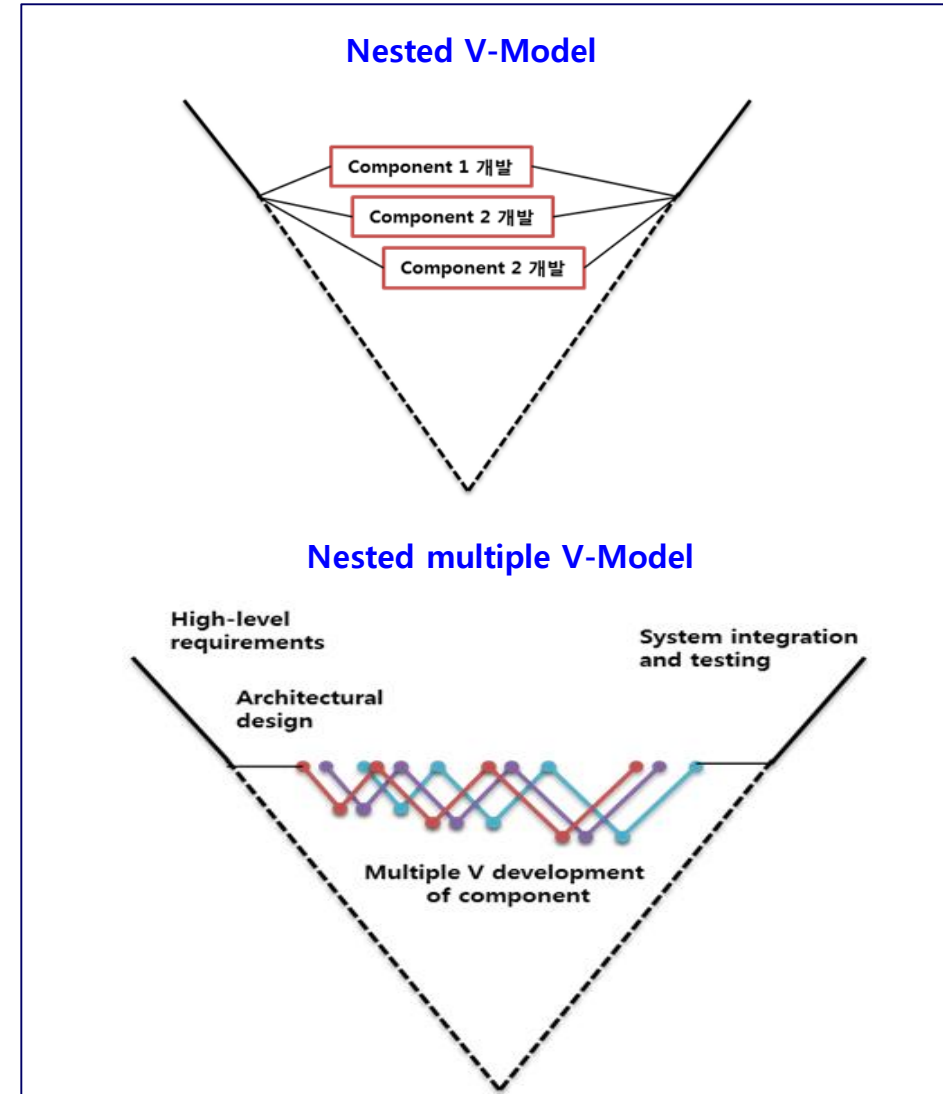
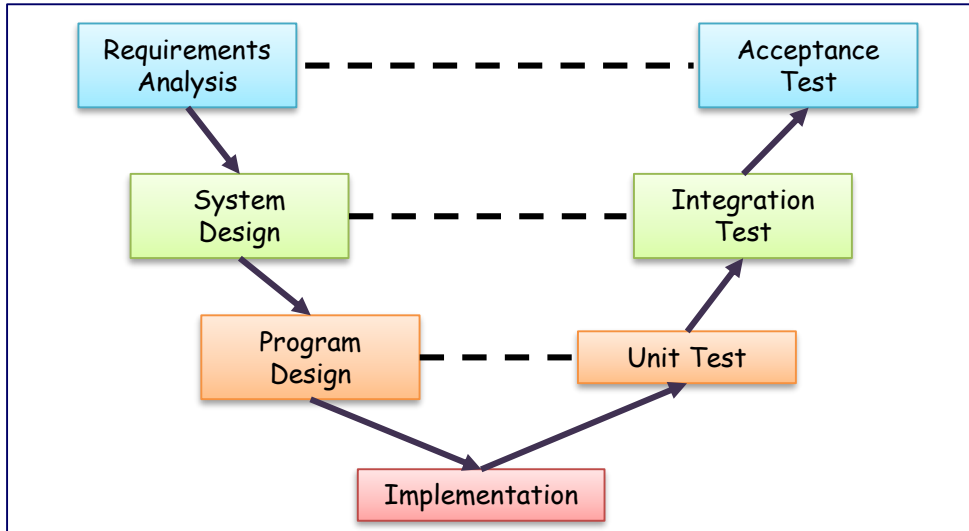
# 마이크로서비스 개발 환경 및 테스트



# V-Model & Test Pyramid

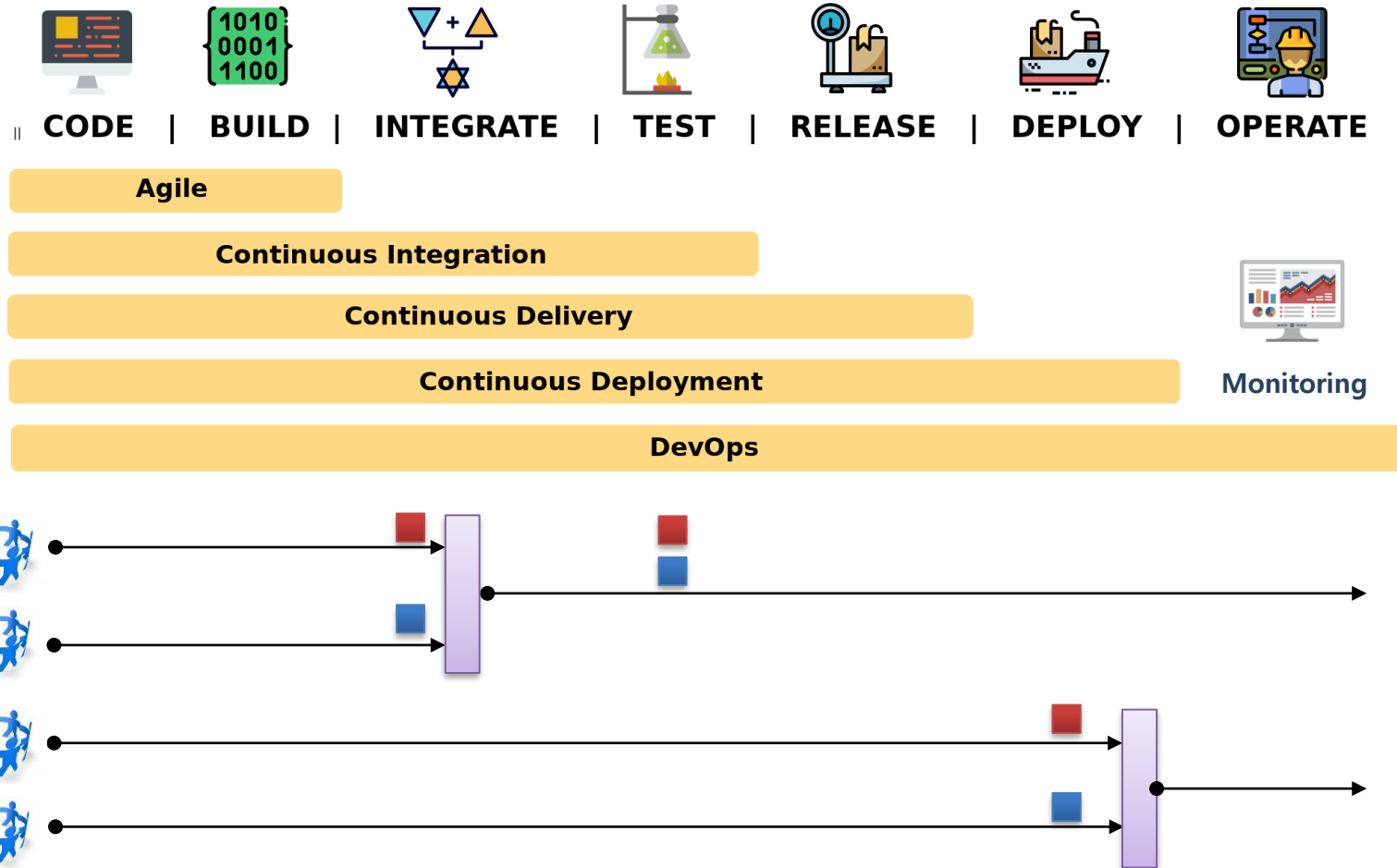


- ❖ Each phase has corresponding test or validation counterpart

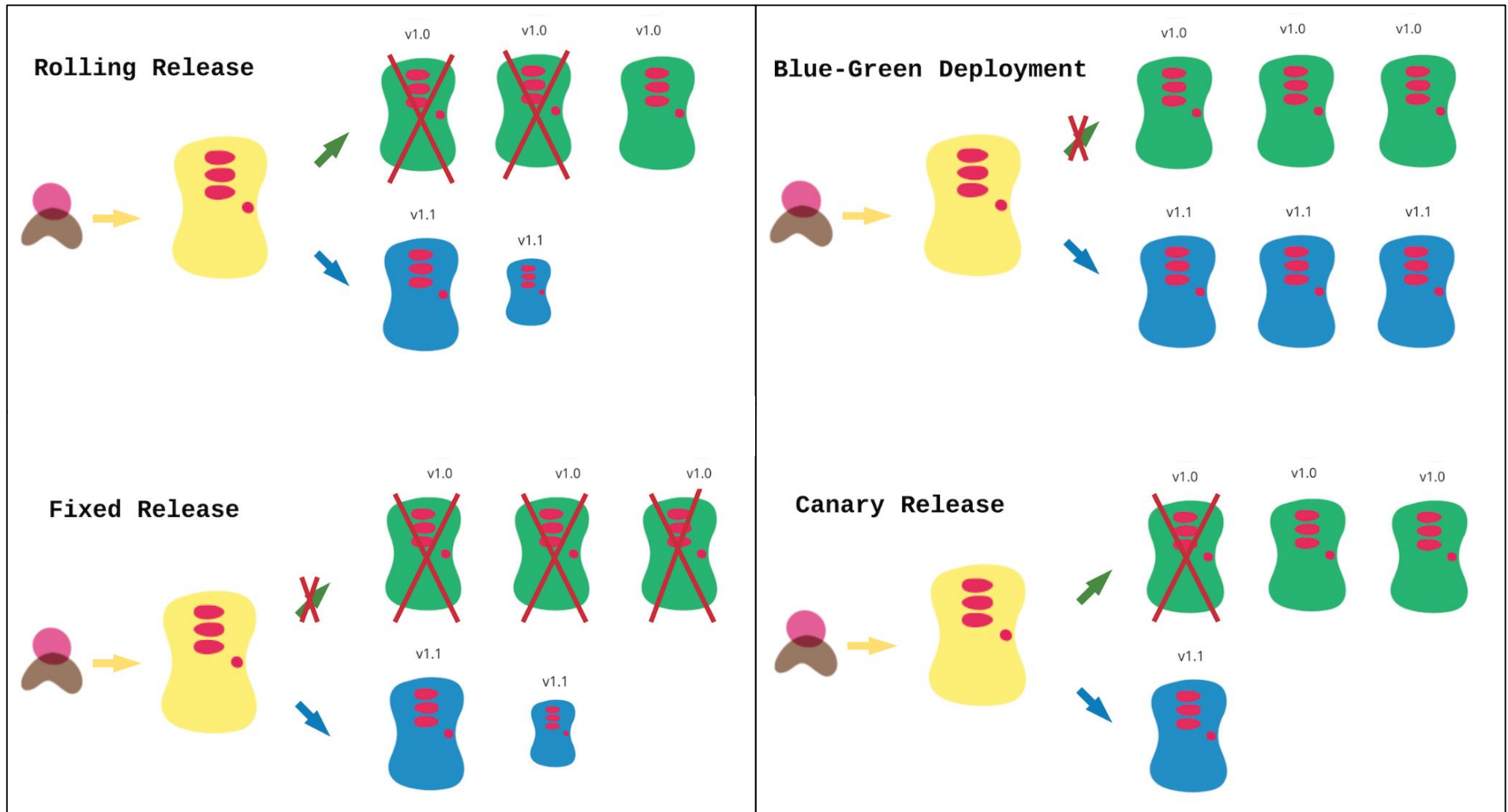


## ❖ Use Continuous Integration/Delivery/Deployment

- Decreased lead times and higher quality with smaller batches



# Deployment Strategy



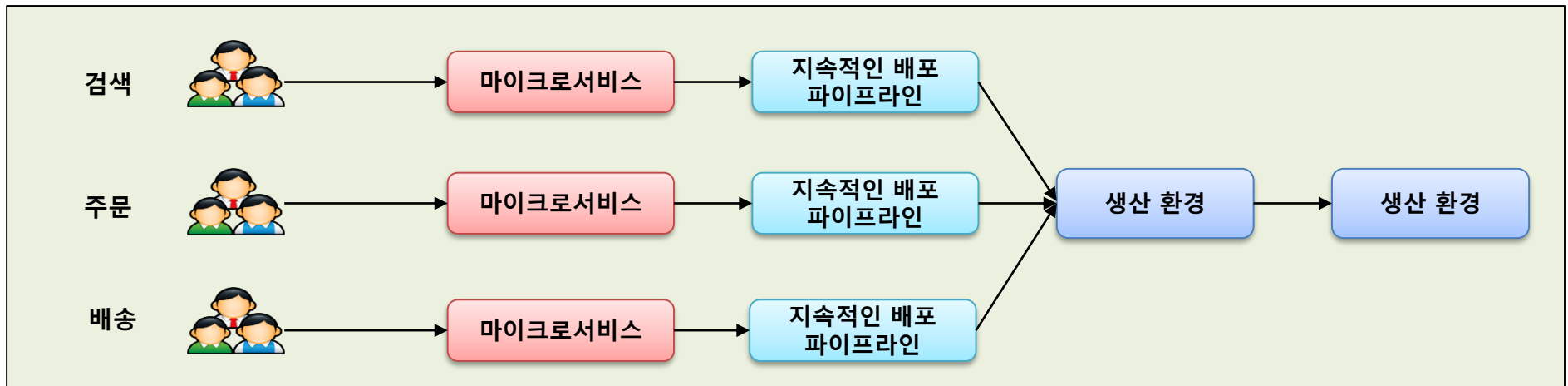
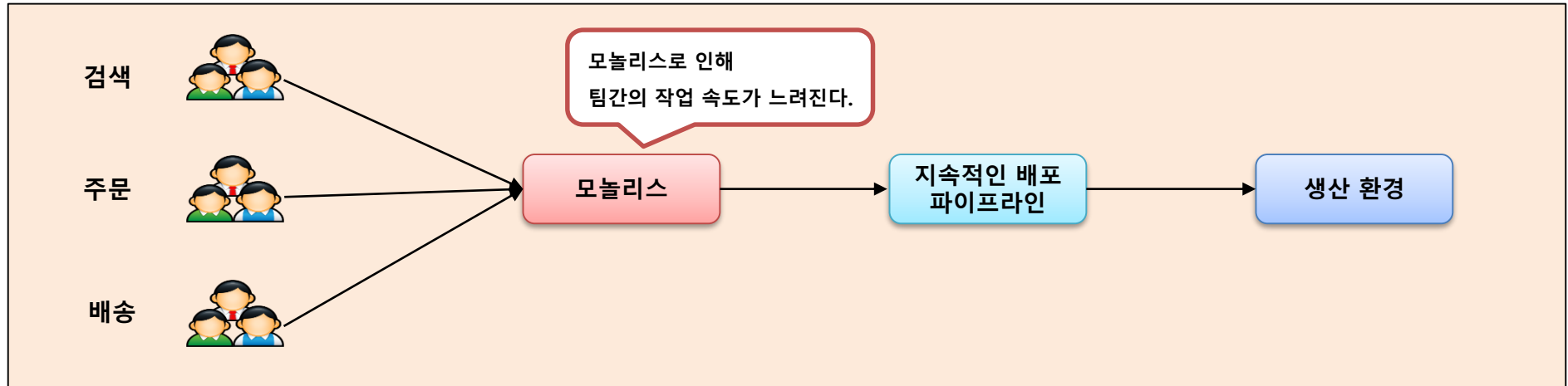




# MSA 사례

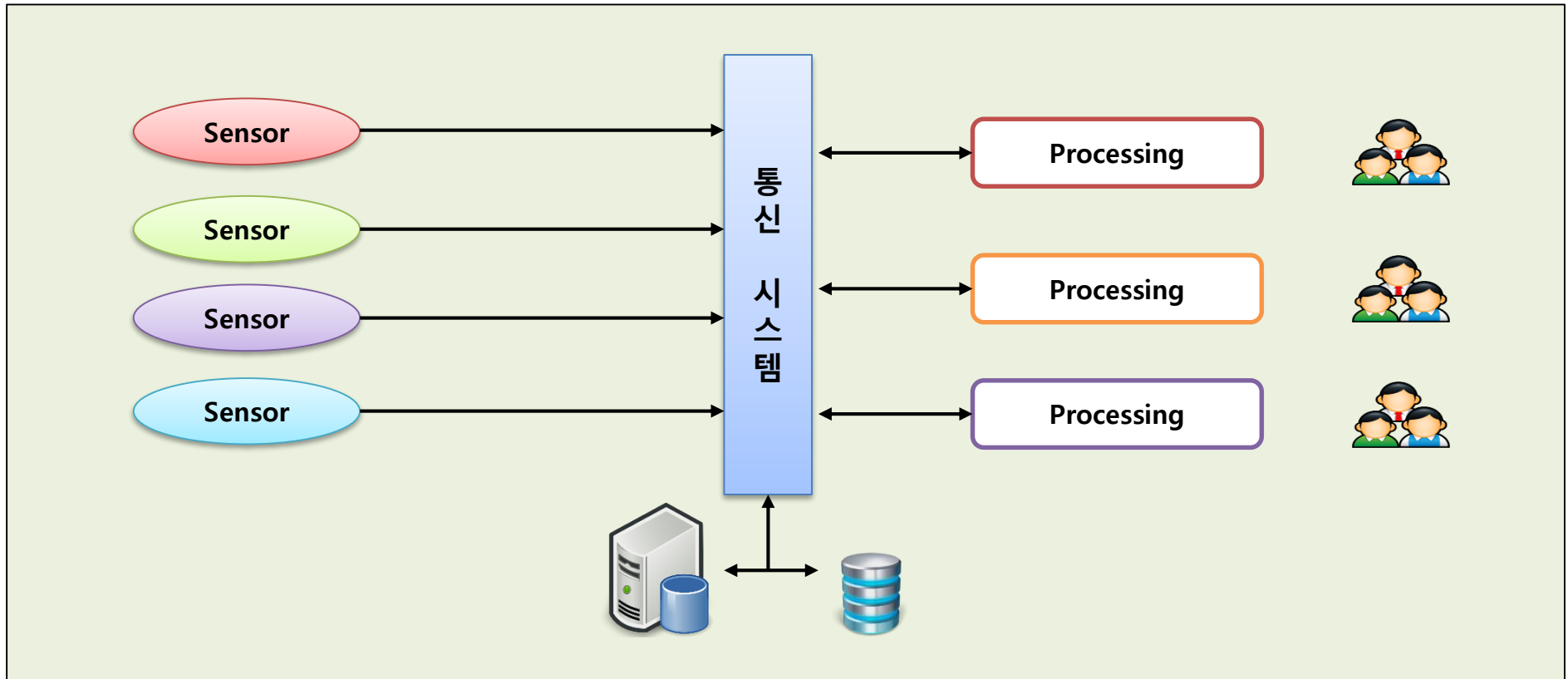


## ❖ A사 유사 사례



## ❖ 신호처리 시스템(Signal Inc.)

- 다양한 종류의 HW -> 이기종 시스템의 통합
- 서로 다른 데이터 처리 작업 → 팀 별 기술 스택이 다름
- 분산 시스템



# Microservice Reference Model

