
openmaru
APM
Microservices Architecture

Microservice Architecture?

What's Microservices?

"마이크로 서비스 아키텍처"라는 용어는 소프트웨어 응용 프로그램을 독립적으로 배치 가능한 서비스 조합 (suite)으로 설계하는 특정 방법을 가리키는 것으로, 지난 몇 년 동안 빠르게 확산되고 있습니다. 이 아키텍처 스타일에 대한 정확한 정의는 없지만, 비즈니스 수행(Capability)과 관련된 조직, 배포 자동화, 엔드 포인트에서의 인텔리전스 (intelligence) 그리고 프로그래밍 언어와 데이터의 분산 제어에 대한 명확한 공통적인 특징들이 존재합니다. "

Martin Fowler's blog - https://martinfowler.com/articles/microservices.html

Microservice Architecture?

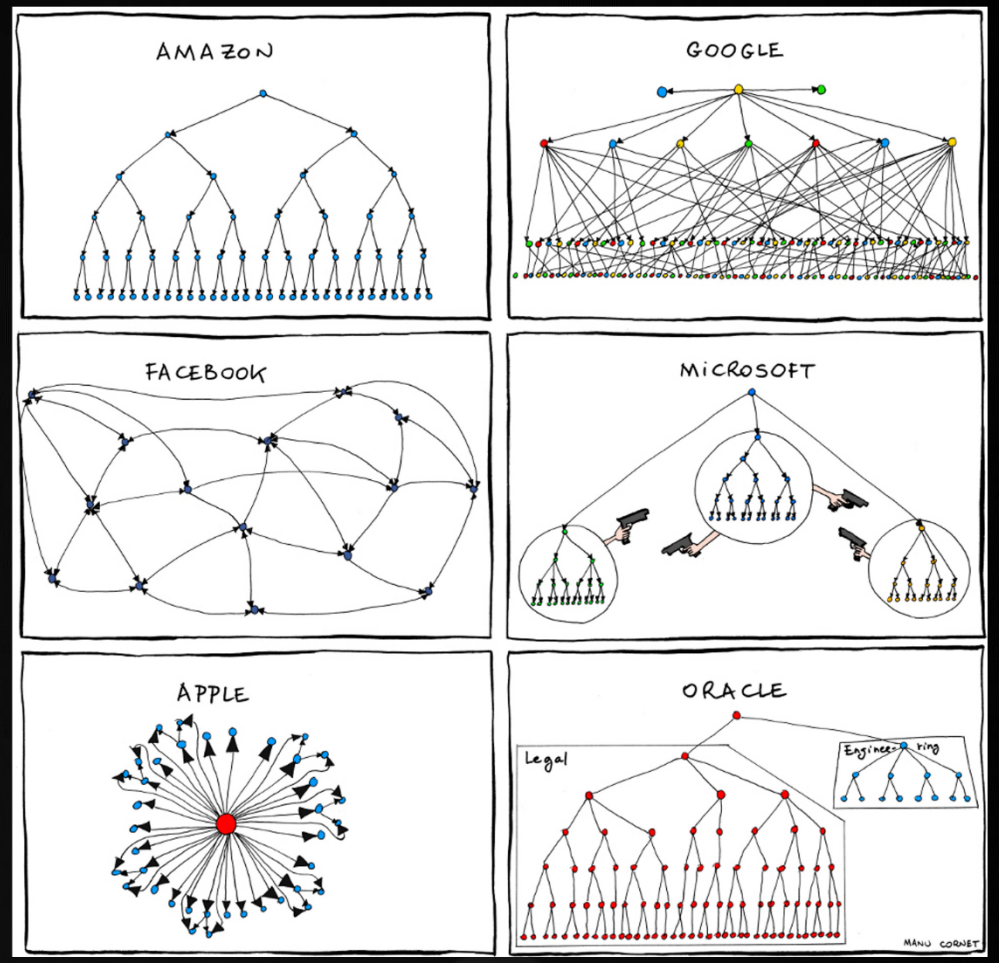
- When? Microservices Architecture?
  - 2012년 ThoughtWorks의 James Lewis가
    Java, the Unix Way라는 발표를 하면서 처음으로 언급
    (발표링크: https://www.infoq.com/presentations/Micro-Services/)
  - 2014년 3월 James Lewis와 Martin Fowler가
    Microservices 라는 내용을 담은 기사를 발표
- 독립적이고 단순한 서비스를 묶어서 전체 서비스를 제공
  - 데이터, 팀조직, 아키텍쳐가 있는 완전한 독립 형태
- 독립적인 팀이 각 서비스의 개발과 운영을 담당
  - 책임과 권한에 있어서 완전한 독립을 추구

Conway's Law

"Any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure."

Melvin Conway , Datamation , 1968

소프트웨어 아키텍처는 그것을 개발 한 조직 구조를 반영한다.



MEL CONWAY'S HOME PAGE - http://www.melconway.com/Home/Conways_Law.html

Picture credit: Manu Cornet / www.bonkersworld.net

서비스 지향 아키텍처(SOA)와의 차이점

- 서비스 기반은 같지만 목표로 하는 수준이 다름
  - SOA는 다수의 시스템이 상호작용하는 방법의 정의함
- 마이크로서비스는 독립적인 환경
  - SOA는 UI나 서비스의 변경이 필요한 경우 둘 이상의 팀에 대한 조정이 필요
  - 마이크로서비스는 개별 팀이 독립적인 환경에서 적용시키려 함

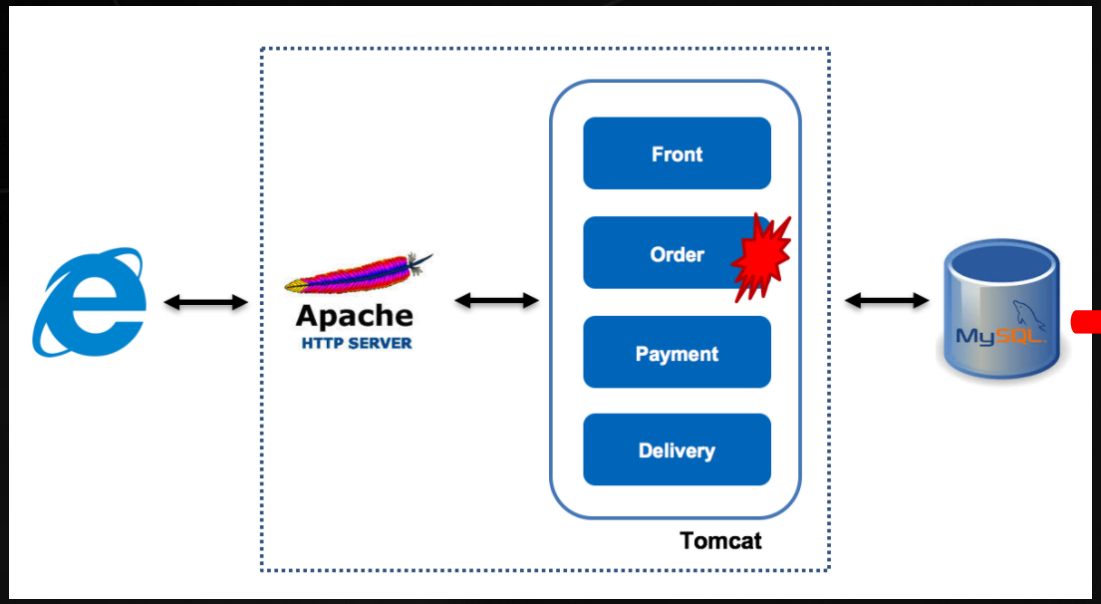
	서비스 지향 아키텍처	마이크로 서비스 아키텍처
범위(scope)	전사적 아키텍처	한 프로젝트에 대한 아키텍처
유연성(Flexibility)	오케스트레이션에 의한 유연성	빠른 배포와 신속하고 독립적인 개발에 의한 유연성
조직(Organization)	다양한 조직 단위에 의한 서비스 구현	동일 프로젝트 내의 팀들에 의한 서비스 구현
배포(Deployment)	여러 서비스들의 단일 배포	각 서비스들의 독립적인 배포
사용자 인터페이스(UI)	모든 서비스들을 위한 보편적인 UI로서의 포털	서비스가 UI를 포함

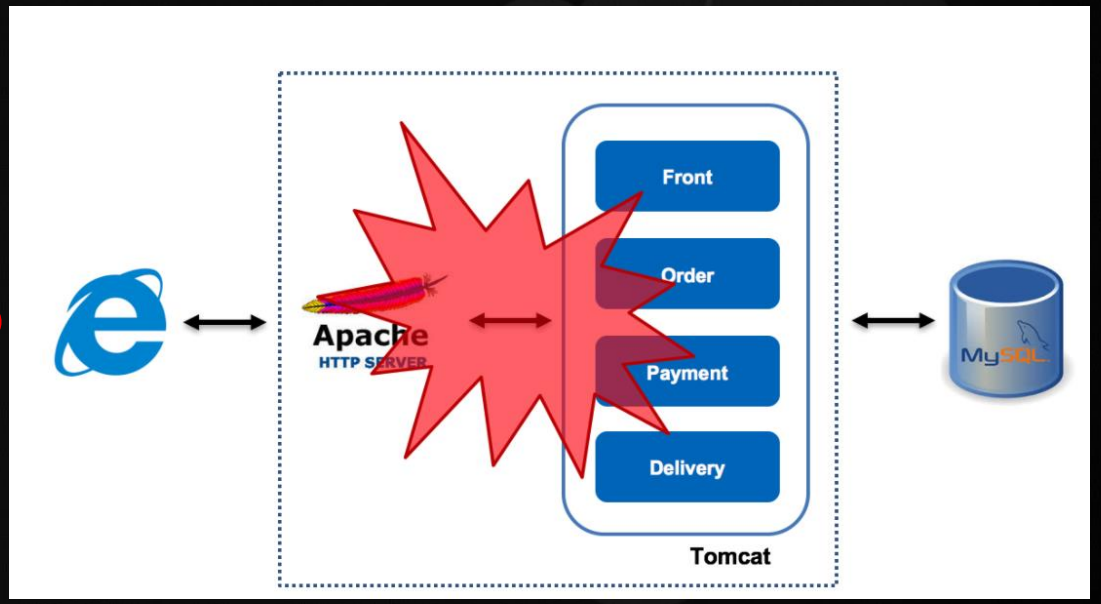

Microservice Architecture
Understanding Microservices
openmaru
APM

Coupang의 MSA 전환사례- Monolith편

기술 블로그에서 발췌/정리
https://medium.com/coupang-tech/%ED%96%89%EB%B3%B5%EC%9D%84-%EC%B0%BE%EA%B8%B0-%EC%9C%84%ED%95%9C-%EC%9A%B0%EB%A6%AC%EC%9D%98-%EC%97%AC%EC%A0%95-94678fe9eb61

Coupang의 MSA 전환사례- Monolith편

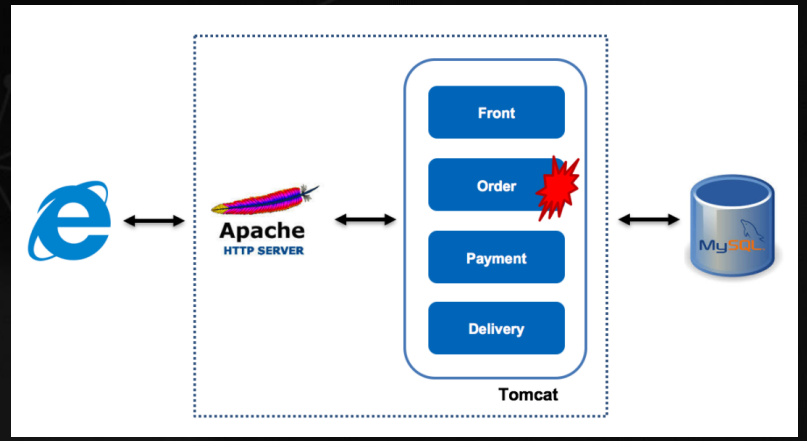


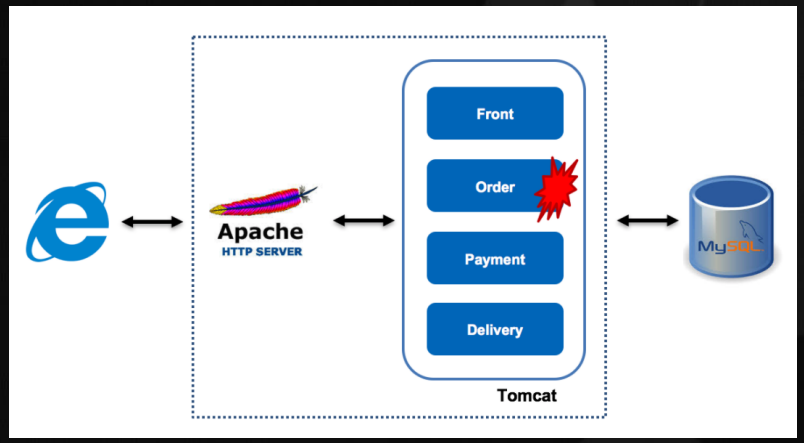


Coupang의 MSA 전환사례-MSA편

기술 블로그에서 발췌/정리
https://medium.com/coupang-tech/%ED%96%89%EB%B3%B5%EC%9D%84-%EC%B0%BE%EA%B8%B0-%EC%9C%84%ED%95%9C-%EC%9A%B0%EB%A6%AC%EC%9D%98-%EC%97%AC%EC%A0%95-94678fe9eb61

Coupang의 MSA 전환사례-MSA편



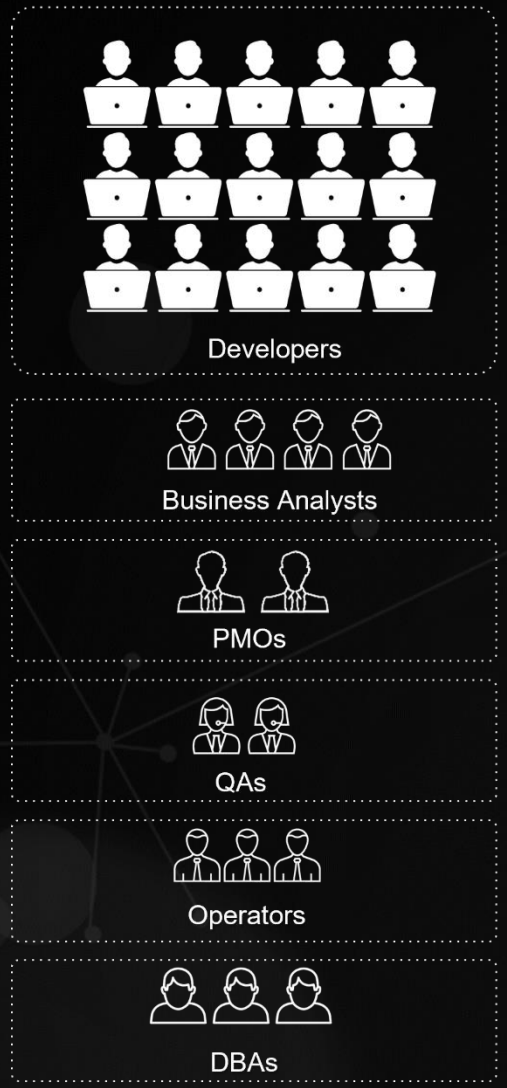


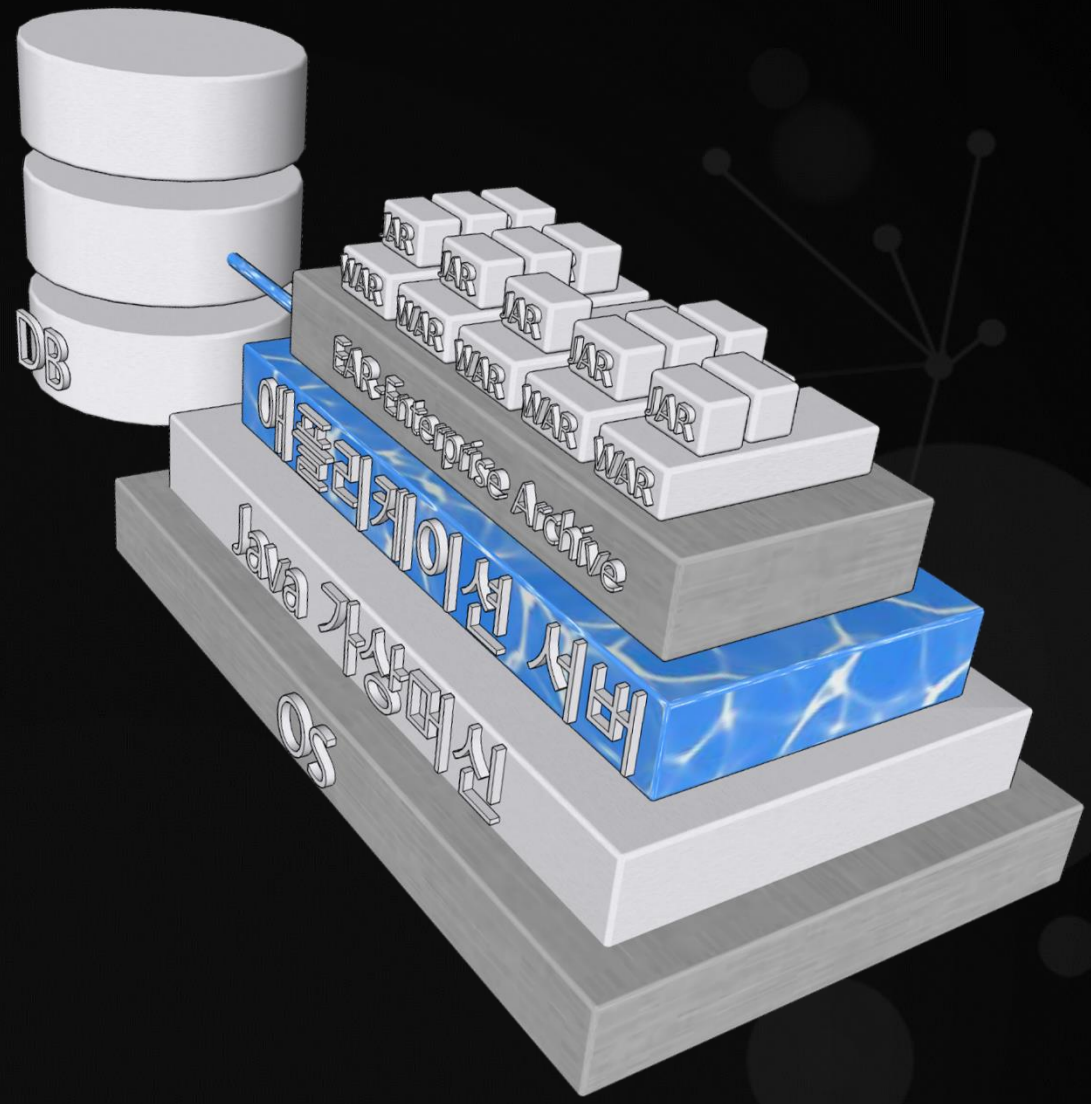
기술 블로그에서 발췌/정리

https://medium.com/coupang-tech/%ED%96%89%EB%B3%B5%EC%9D%84-%EC%B0%BE%EA%B8%B0-%EC%9C%84%ED%95%9C-%EC%9A%B0%EB%A6%AC%EC%9D%98-%EC%97%AC%EC%A0%95-94678fe9eb61

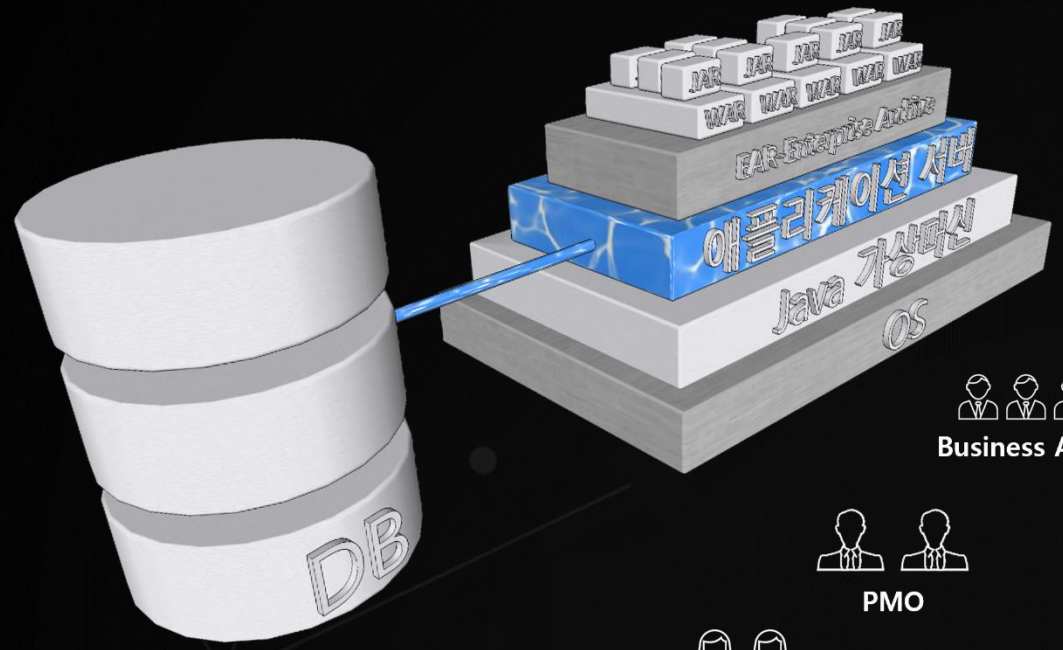

Microservice Architecture
Microservices vs Monolith
openmaru
APM

Monolith Architecture Team





Microservices Architecture Team



Developers

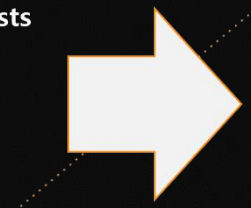
Business Analysts

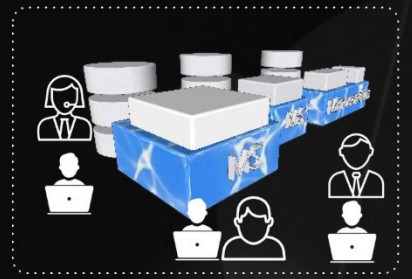
PMO

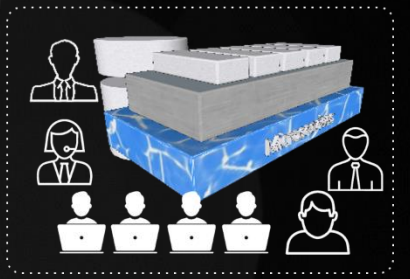
QA

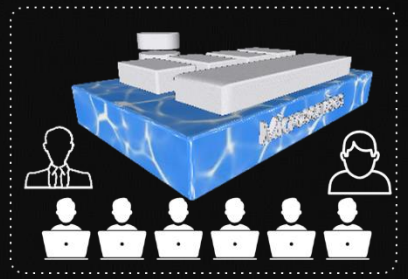
Operators

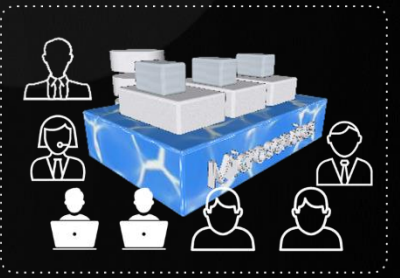
DBAs











3 티어 아키텍처 – 2000년대

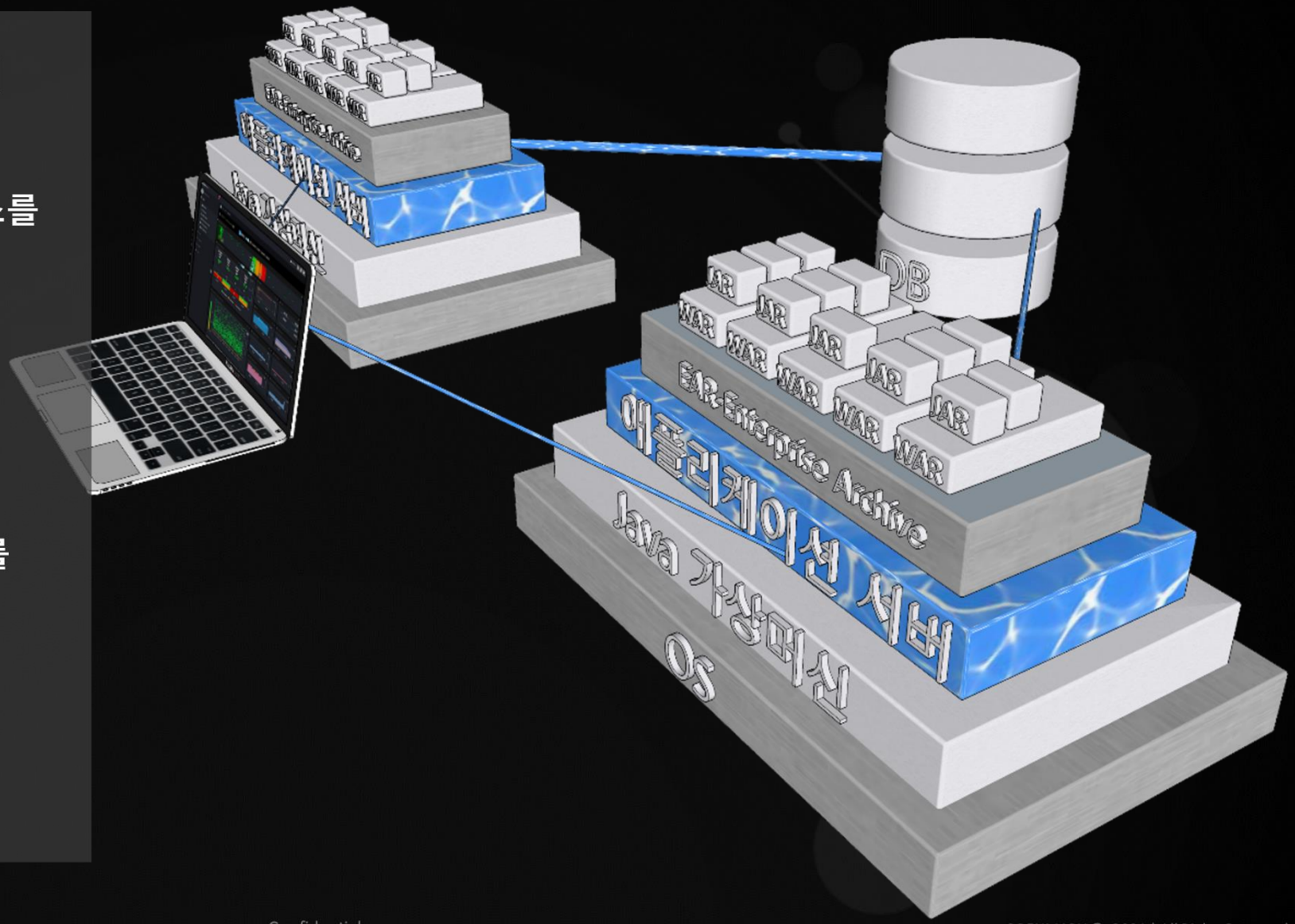
클라이언트는 주로 웹브라우저를 사용하며, 서블릿이나 JSP 를 사용하여 서버 구현

H/W와 N/W자원을 효율적으로 사용하기 위하여 App Server 인스턴스를 최소화하고 세션정보도 최소화

데이터 내용과 중요도에 관계없이 주요 저장소로 RDBMS를 사용

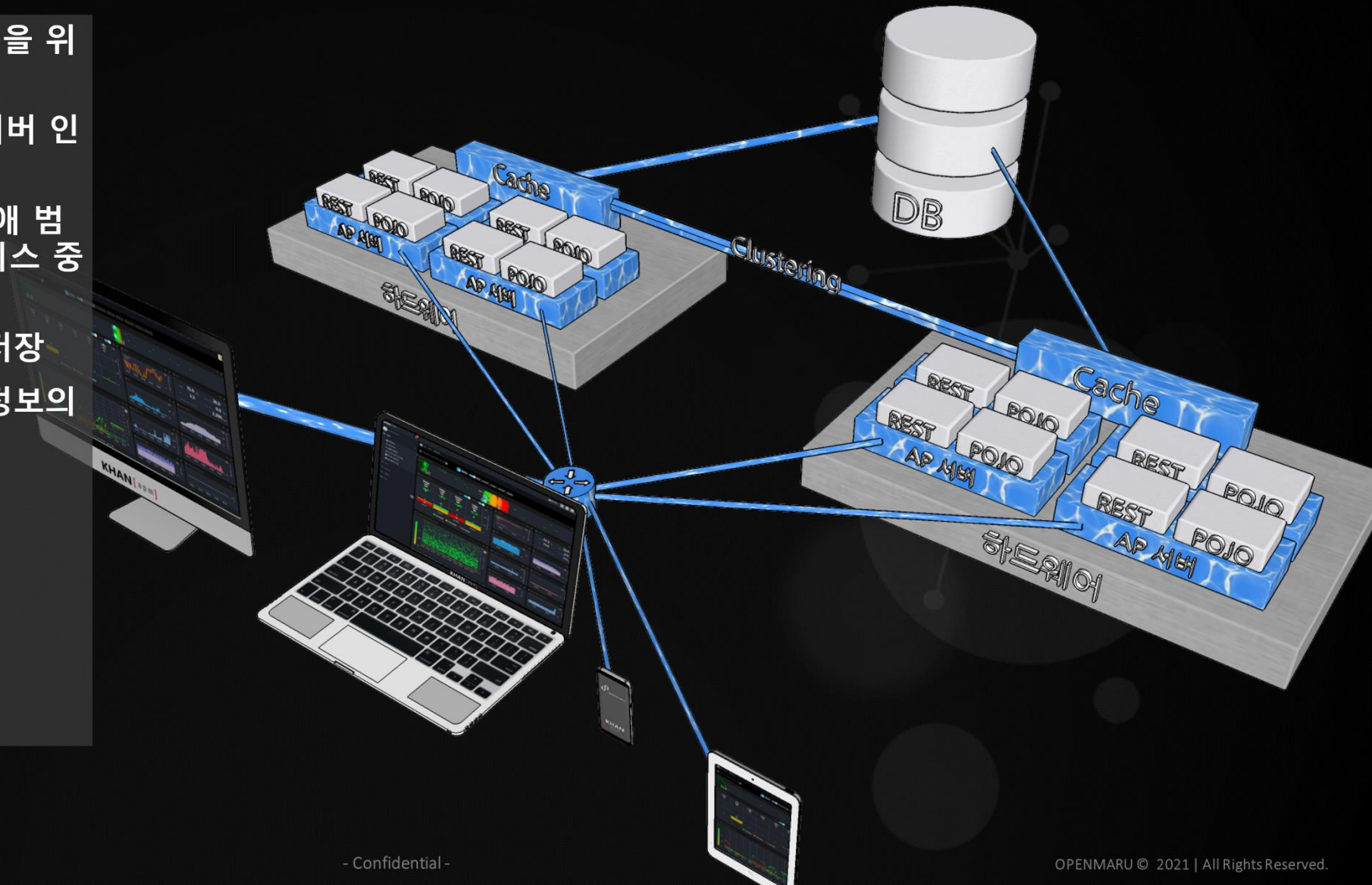
아키텍처 이슈

1. 인스턴스가 적기 때문에 장애의 영향이 큼
2. Hot Deploy 등을 사용하여 서버를 중지하지 않고 운영하는 구조
3. 많은 데이터가RDBMS에 저장되기 때문에 RDBMS 튜닝에 의존
4. 확장하는 것이 거의 불가능



4티어 아키텍처 – 2010년 이후

- 멀티 디바이스와 옴니 채널 대응을 위한 REST가 주류
- H/W와 N/W자원이 저렴해져 서버 인스턴스를 여러 대 구성
  - 인스턴스가 많기 때문에 장애 범위가 작고, 서버 장애는 서비스 중지가 아님
- 세션 정보는 기본 캐시 서버에 저장
- 데이터 내용 및 중요도에 따라 정보의 저장 위치를 선택
- 확장 가능한 아키텍처
- 관리 오버헤드 발생

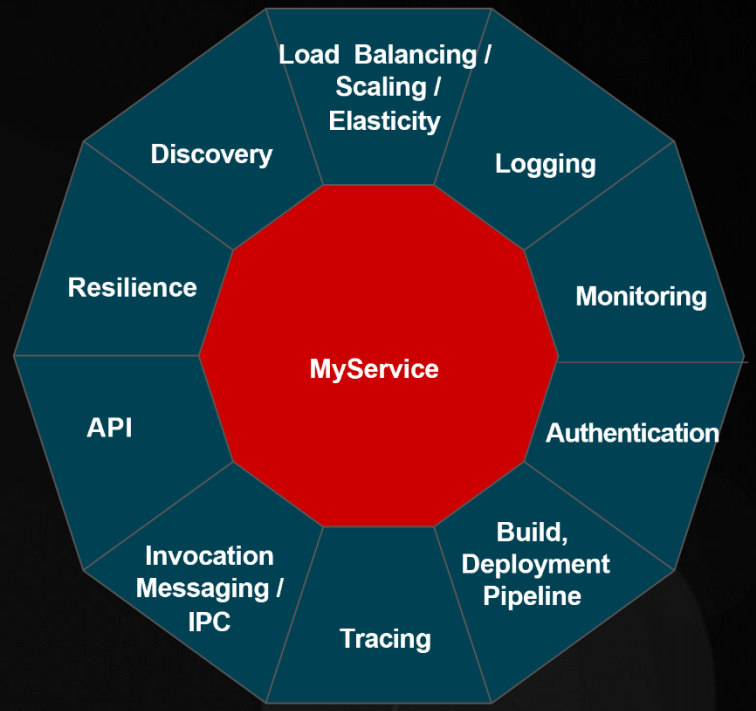


마이크로서비스 아키텍처 등장의 배경

- 기술 환경의 변화
  - 하드웨어의 저렴화와 고속화
  - 네트워크의 고속화 및 보급
  - 클라우드 환경의 충실
- 웹 서비스 레거시 화
  - 기업의 IT 환경이 보다 거대하고 복잡하게 됨
  - 서비스마다 특성과 변화 속도가 크게 다르기 때문에 표준화 어려움
    - 빅뱅이 아닌 개별 서비스의 재구성
  - 거대한 웹 서비스를 관리하기 위한 필연적인 선택이 MSA
- 서비스 공유의 일반화
  - SDx 흐름에 따른 다양한 가상화 기술
  - 서비스에 대한 성능과 가용성을 보장
  - API-Gateway 를 통하여 서비스의 공유화를 실현

마이크로 서비스 과제

- 구축
  - 서비스 경계 / 입도 / 빌드 / 배포 효율화
  - 서비스 간 연계 (서비스 검색 오케스트레이션, 코레오 그래피)
  - 테스트 (서비스 단위 / 엔드 투 엔드)
  - 자율적 인 회복
- 운용
  - 서비스의 건전성 관리 / 모니터링
  - 서비스 장애 대응 (로그 집중 관리, 추적)
  - 서비스 간의 의존 관계 관리 (버전 관리, 데이터 마이그레이션) / 성능 관리 / 자원 관리
  - API 관리 (특히 외부 공개용)
- 보안
  - 분산 환경에서의 인증 /인가
  - 암호화
  - 감사



마이크로 서비스로 향하는 길

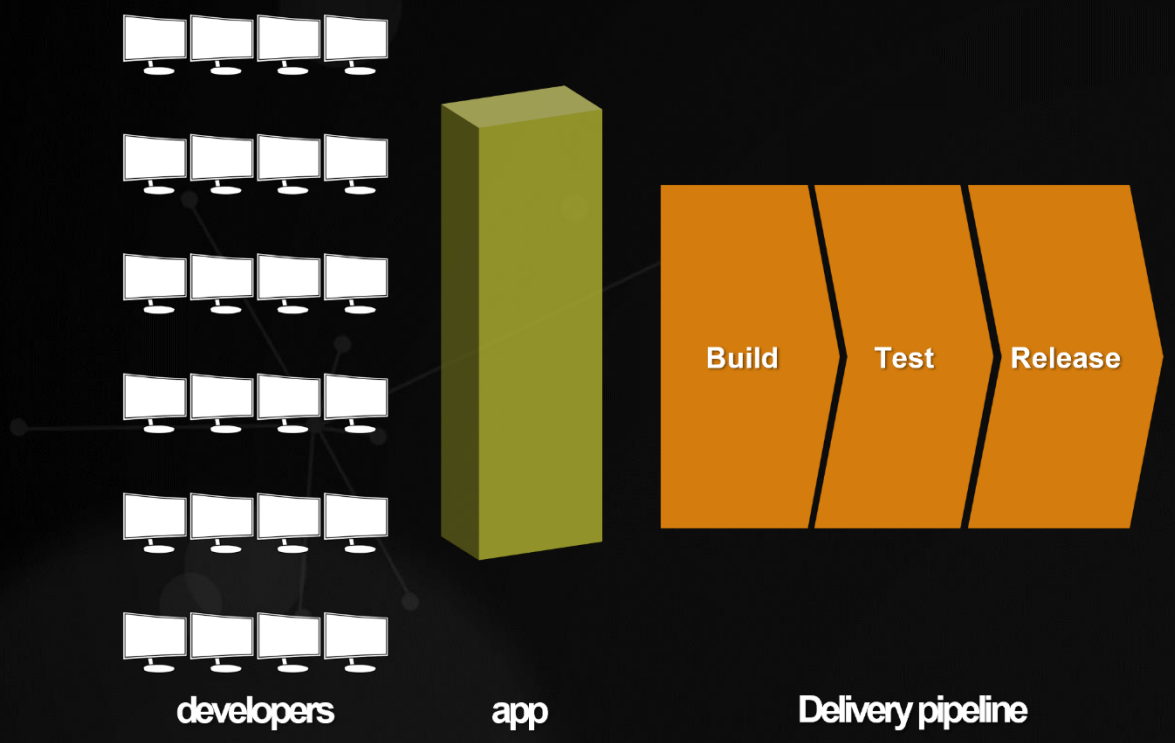
- 모놀리스가 만능 해결책은 아니다
  - 기존의 모놀리스로도 충분한 경우, 초기 서비스 단계인 경우
- 모놀리스를 단계적으로 다수의 마이크로 서비스로 전환하기
  - 기존 시스템에서 분할이 가능한 항목 또는 신규 기능을 마이크로 서비스로 분리하 여 추가하기
- 의존성 낮추기
  - 독립적인 배포가 가능하도록 서비스를 분리하기
- 조직 문화의 변화 수용
  - 마이크로 서비스는 만능 해결책이 아니므로, 구성원들의 skill-up에 대한 관심과 투자가 뒷받침 되어야 함

개발 프로세스 차이

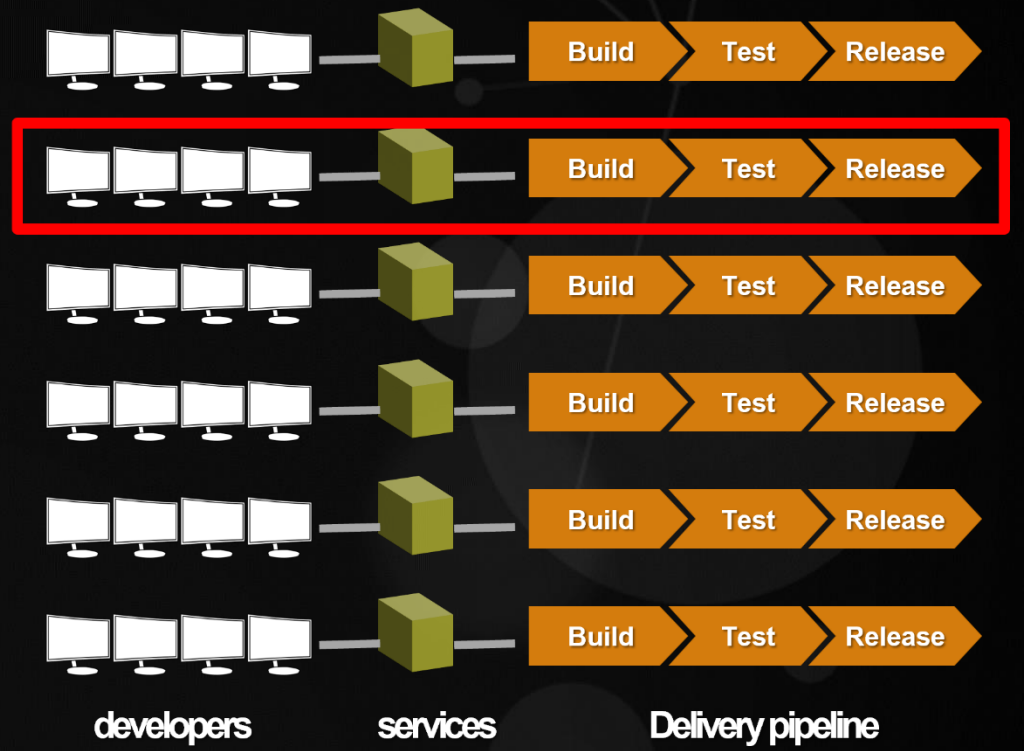
Monolith vs. Microservices

openmaru

- 모놀리식 개발 프로세스

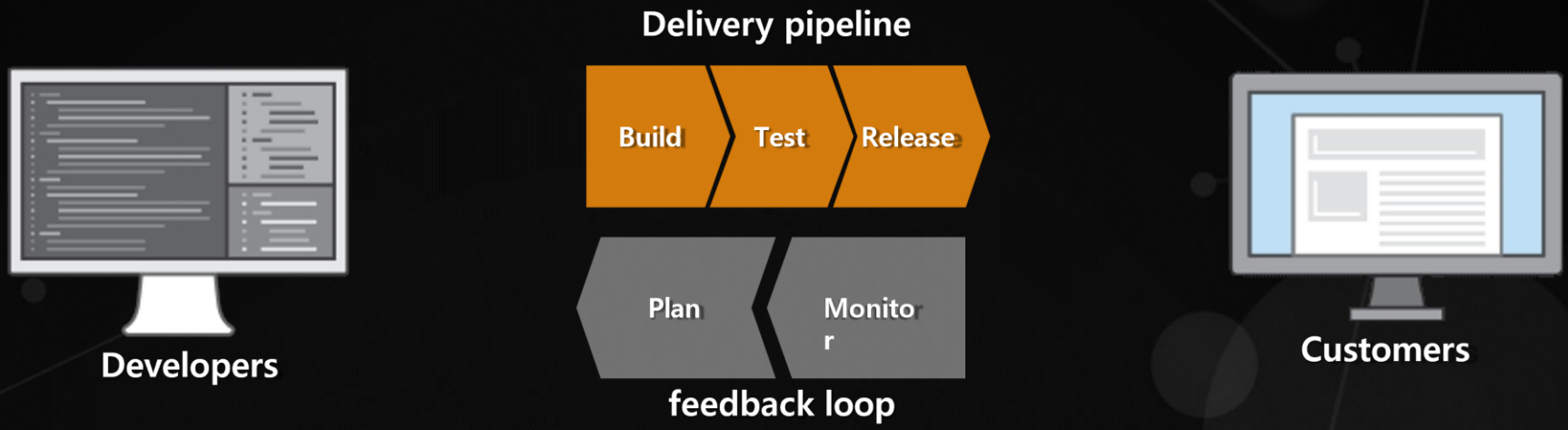


- 마이크로 서비스 개발 프로세스



DevOps

소프트웨어 개발 라이프 사이클

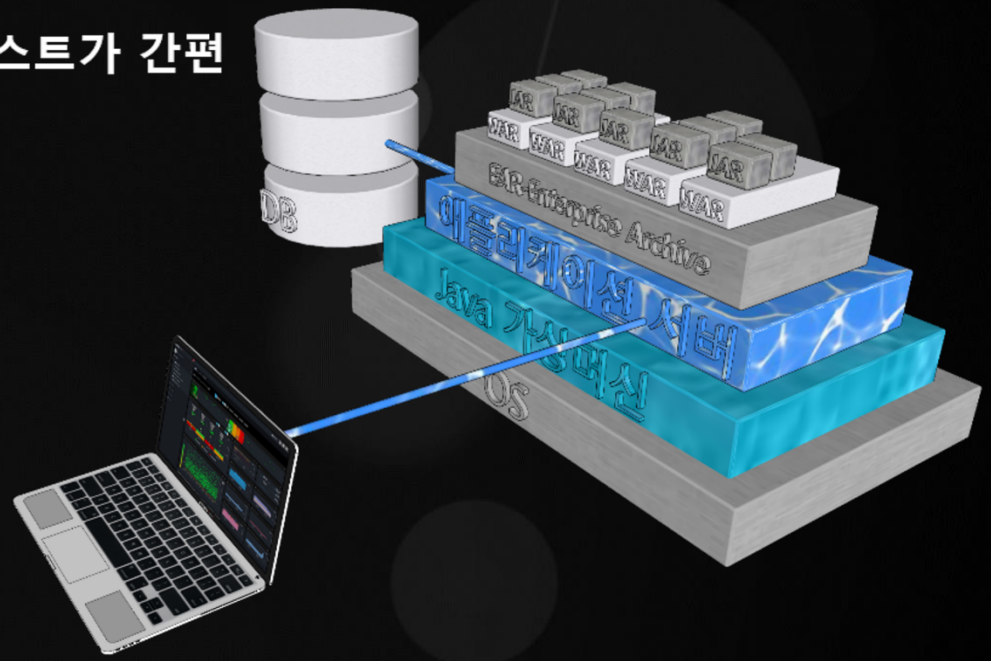


DevOps = 라이프 사이클을 효율적으로 단축

Monolith Architecture

- 단점
  - 배포가 전체 응용 프로그램에 영향을 미치
  - 개발의 영향 범위, 책임 분해 점이 어려운
  - 학습 장벽이 높음
  - 확장성 – 스케일업
  - **신기술 도입이 어려움**
  - 여러 개의 기술 혼용
  - **배포 및 재기동 시간이 오래 걸림**
  - **수정이 용이하지 않음**

- 장점
  - 기술 단일화
  - 관리 용이성
  - 심플한 구조
  - 간편한 코드관리
  - 간편한 트랜젝션관리
  - 설계/테스트가 간편



Application Performance Management

Microservices Architecture 특징

openmaru
APM

Microservices 9 가지 특징 (1/2)

1. 서비스 단위의 컴포넌트 화 (Componentization via Services)
   - 서비스를 HTTP 나 RPC에 연계하는 컴포넌트로 구현
2. 비즈니스 기능 중심의 구성 (Organized around Business Capabilities)
   - 기술이 아닌 비즈니스 기능에 따라 서비스의 분할을 실시 - 콘웨이 법칙
3. 프로젝트가 아니라 제품 (Products not Projects)
   - 기능을 제공하고 끝이 아니라 사용자가 있는 제품수준으로 개발
4. 스마트 엔드포인트와 간단한 파이프 (Smart endpoints and dumb pipes)
   - 복잡한 프로토콜이 아닌 RESTful 한 HTTP API 및 경량 메시징 프로토콜을 사용

Martin Fowler's blog - https://martinfowler.com/articles/microservices.html

Microservices 9 가지 특징 (2/2)

5. 개발의 분권화 (Decentralized Governance)
   - 특정 기술에 고착되고 중앙 집권적인 관리에서
     가볍고 최적화된 기술 선택으로 전환
6. 데이터 관리의 분권화 (Decentralized Data Management)
   - 개별 서비스마다 데이터베이스를 소유
7. 인프라 자동화 (Infrastructure Automation)
   - 프로비저닝을 통한 인프라 구성에 대한 자동화
8. 장애를 전제로 한 설계 (Design for failure)
   - 장애가 서비스에 영향을 주지 않도록 하며, 모니터링 체계 구축
9. 변화에 대응하는 설계 (Evolutionary Design)
   - 변화에 쉽게 대응할 수 있도록 작은 규모의 조직 구성

Martin Fowler's blog - https://martinfowler.com/articles/microservices.html

Microservices 특징

- 기술측면 : 분산과 통합
  - 서비스 단위의 컴포넌트 화
  - 스마트 엔드포인트와 간단한 파이프
  - 데이터 관리의 분권화
  - 인프라 자동화
  - 장애를 전제로 한 설계
- 문화측면 : 지속성과 분권
  - 비즈니스 영역에 따른 조직화
  - 프로젝트가 아니라 제품
  - 개발의 분권화
  - 변화에 대응하는 설계

서비스를 서비스로 구성

•정적 결합에서 동적으로 서비스를 조합

메시지에 의한 통합

•개별 서비스는 표준 프로토콜로 통신

서비스 관리

•모니터링, 의존성 관리, 장애 감지 및 복구, 버전 관리

서비스를 지속적으로 운영

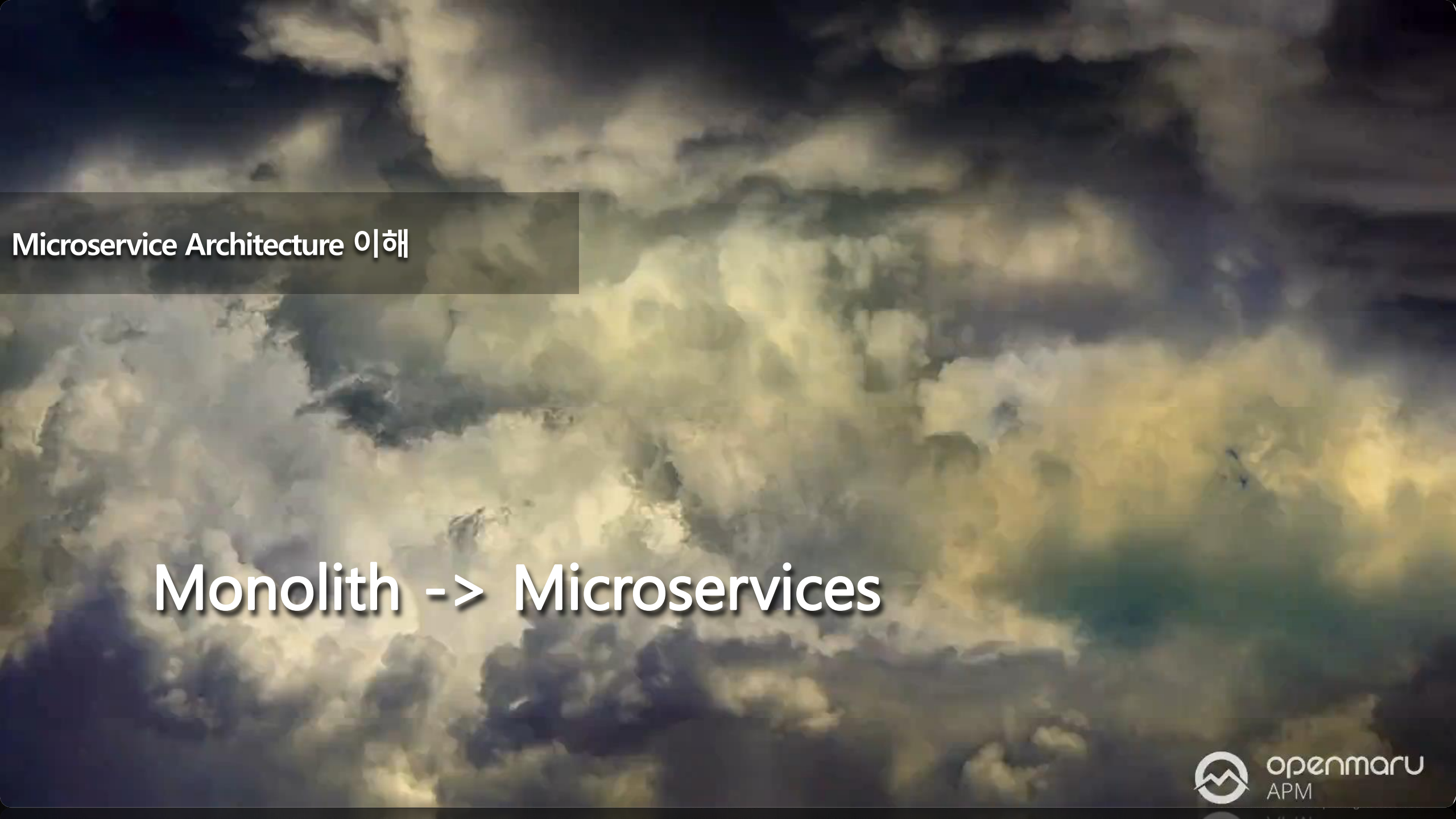
• 소프트웨어 개발에서 IT 서비스 운영으로 전환

도메인 관련 기술과 운영

• 각 도메인 별로 자주성을 인정

도메인 별 라이프 사이클 관리

• 개별적인 구성을 허용


Microservice Architecture 이해
Monolith -> Microservices
openmaru
APM

마이크로서비스 고려사항

- 개발 및 운영 복잡성
  - 서비스 단위로 분리하고 실제 서비스 가능한 상태로 만든다는 것은 그만큼 시스템
    이 늘어나며, 운영 복잡도가 증가함
- 코드 의존성
  - 개별 서비스를 독립적으로 배포할 수 있기 때문에 서로 다른 라이브러리 버전들이 바이너리 의존성 때문에 더 이상 호환되지 않을 수 있음
- 성능 저하
  - 서비스가 분리되어 늘어난 통신들로 인해 성능이 저하될 수 있음
- 인프라 스트럭처와 운영
  - 모니터링, 배포, 테스트가 자동화되어 있어야 함

마이크로서비스 아키텍처와 거버넌스

- SOA : 하향식 vs. MSA : 상향식

  SOA : 이상적이며 전체 변경 vs. 현실적이며 부분 최적화 변경

  SOA : 전체 시스템을 통제 vs. 전체 서비스를 분할하여 통제

- 봉건 → 군주제 → 민주주의로의 변화와 유사

봉건제 :
지역 영주에
의한 분리 통치

Legacy : 연결성 없음

군주제 :
왕에 의한 중앙
집권제 통치

SOA : 벤더 중심

민주주의:
국민에 의한 통치

MSA : 개별 시스템 중심

There’s no silver bullet.

도메인 = 변화의 경계선

- 변화의 경계선을 찾는다
  - 모듈화는 변화의 경계선에 의해 일어난다
- 변화의 요인 (외부 / 내적)는 품질 특성에서 이해하기
  - 기능뿐만 아니라 비 기능도 중시해야 한다
- 변화의 경계에 선을 긋는다
  - 변화의 경계는 불분명하지만 선을 그릴 수 밖에 없다
- 도메인 경계를 유지한다
  - 패키지 문제 또는 재 구축 문제

12 Factor App 방법론

- 최근에는 소프트웨어가 서비스로 제공되며, 웹 애플리케이션과 Software as a Service 라고 함
- Twelve-Factor App은 다음과 같은 Software as a Service를 만들기 위해 방법론
  - 설치 자동화에 대한 선언적인 형식을 사용하여 프로젝트에 추가 된 새로운 개발자가 시간과 비용을 최소화
  - 하부 OS에 의존 관계를 명확히 하고 실행 환경 사이의 이식성을 최대화
  - 현대적인 클라우드 플랫폼 에서의 배포에 적합하며, 서버 관리 및 시스템 관리가 불필요
  - 개발 환경과 운영 환경의 차이를 최소화 하고 민첩성을 극대화하는 지속적인 배포가 가능
  - 아키텍처 개발 사례를 크게 변경하지 않고 확장
- Twelve-Factor 방법론은 어떤 프로그래밍 언어로 작성된 응용 프로그램에도 적용가능
- 백엔드 서비스 (데이터베이스, 메시지 서버, 메모리 캐시 등)와 조합하여 적용

The 12 Factor App (1/2)

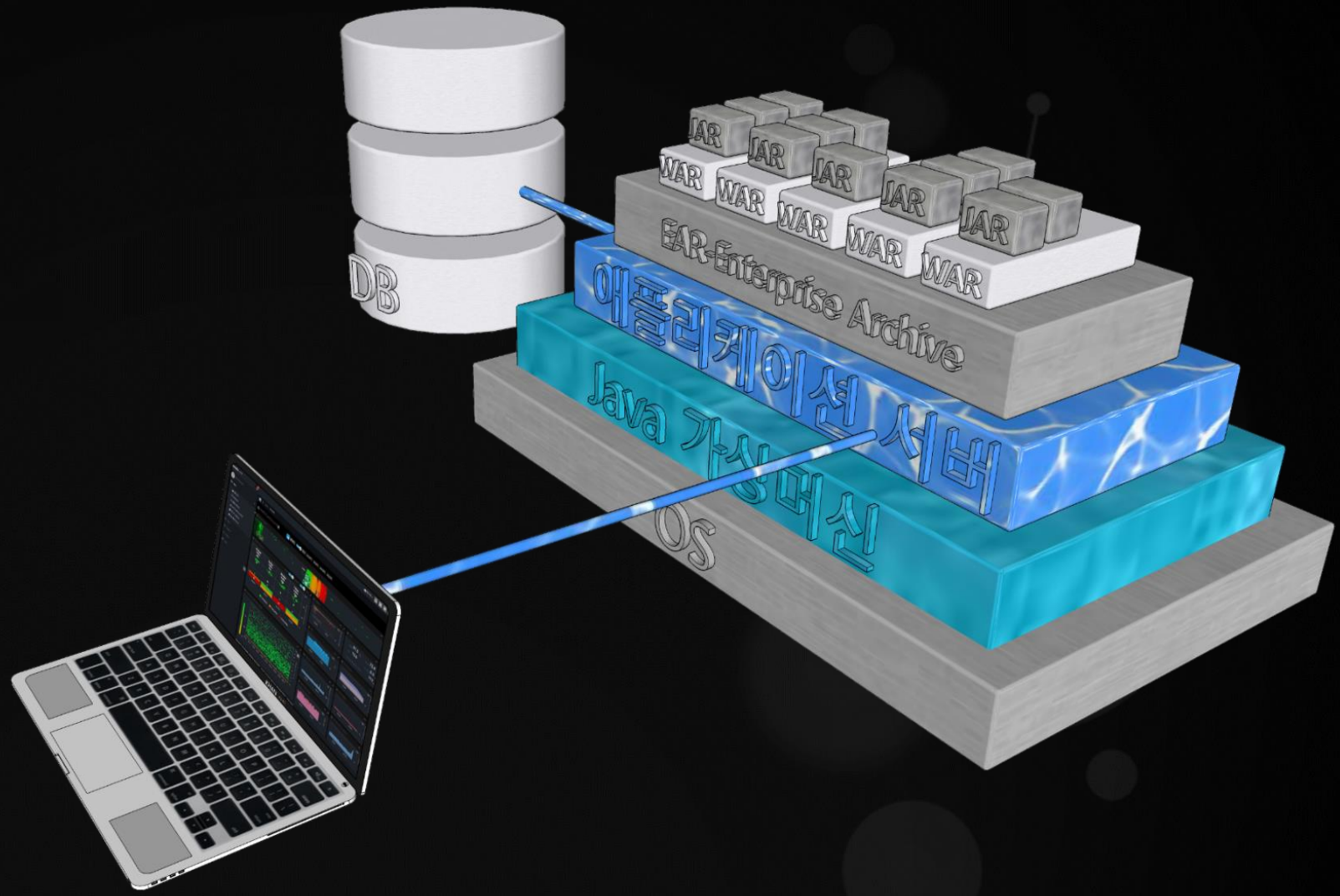
- I. 코드베이스
  - 버전 관리되는 하나의 코드베이스로 여러 곳에 배포
- II. 종속성
  - 의존 관계를 명시적으로 선언하고 분리
  - 환경에 의존하지 않도록 함
- III 설정
  - 설정을 환경 변수에 저장하기
- IV. 백엔드 서비스
  - 백엔드 서비스를 연결된 리소스로 취급
- V. 빌드 릴리스, 실행 (Build, release, run)
  - 빌드, 릴리스, 실행 3 단계를 엄격하게 분리
- VI 프로세스
  - 응용 프로그램을 하나 또는 여러 개의 독립적인 프로세스로 실행
- VII. 포트 바인딩
  - 포트 바인딩을 통해 서비스를 공개

The 12 Factor App (2/2)

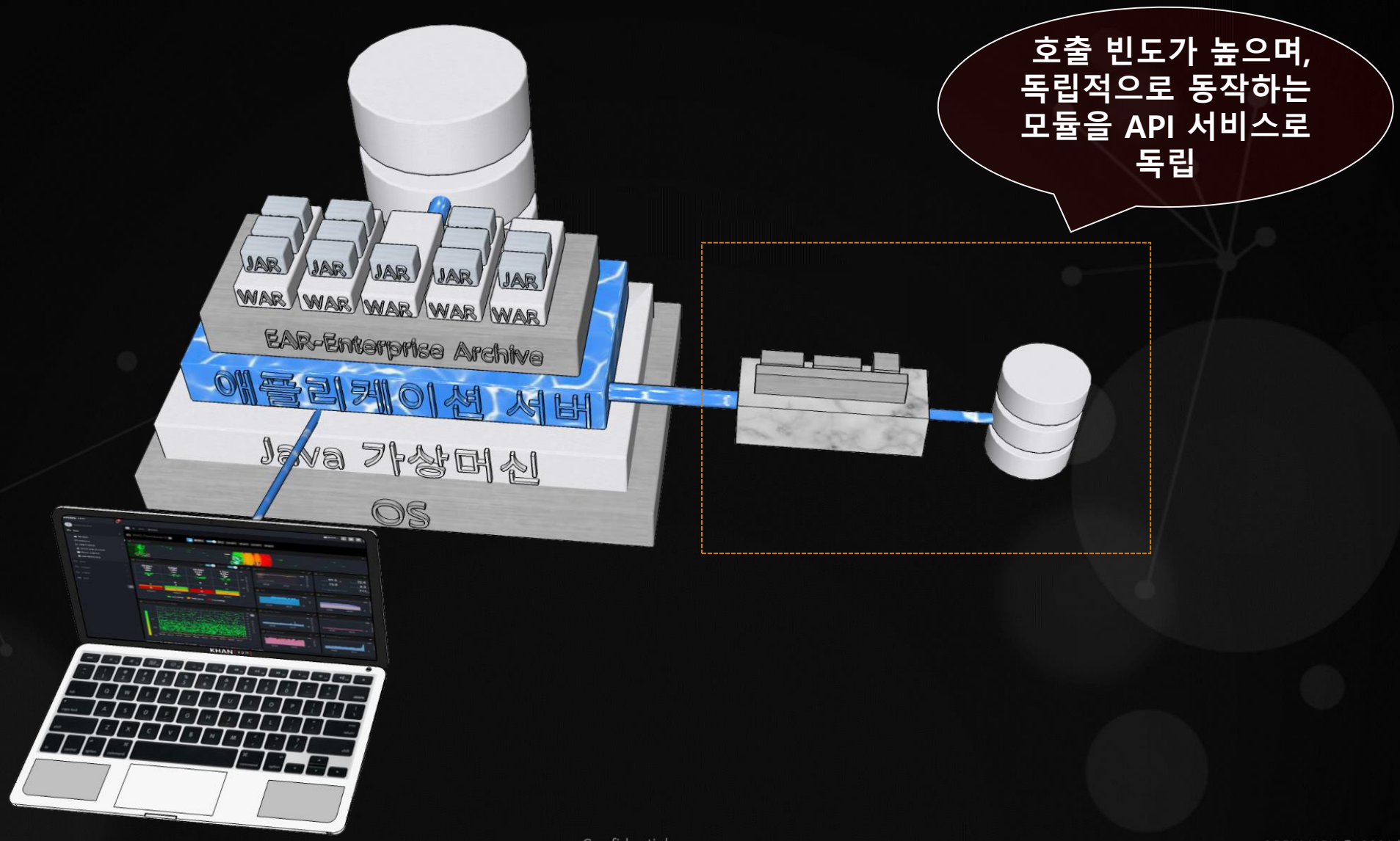
- VIII. 동시성
  - 프로세스 모델에 따라 확장
- IX. 폐기 용이성
  - 빠른 시작이 가능하며, Graceful Shutdown시 서비스에 영향을 미치지 않도록 함
- X. 개발 / 운영 일치
  - 개발 준비 프로덕션 환경과 최대한 일치된 상태를 유지
  - CI / CD (Continuous Integration/Continuous Delivery)환경이 갖춰져 있어야 함
- XI. 로그
  - 로그를 이벤트 스트림으로 취급함
  - 중앙 집권적인 서비스를 통해 로그 이벤트를 수집하고, 인덱싱하여 분석하는 환경이 가능해야 함
- XII. 관리 프로세스
  - 관리 작업을 일회성 프로세스로 실행

Monolith Architecture

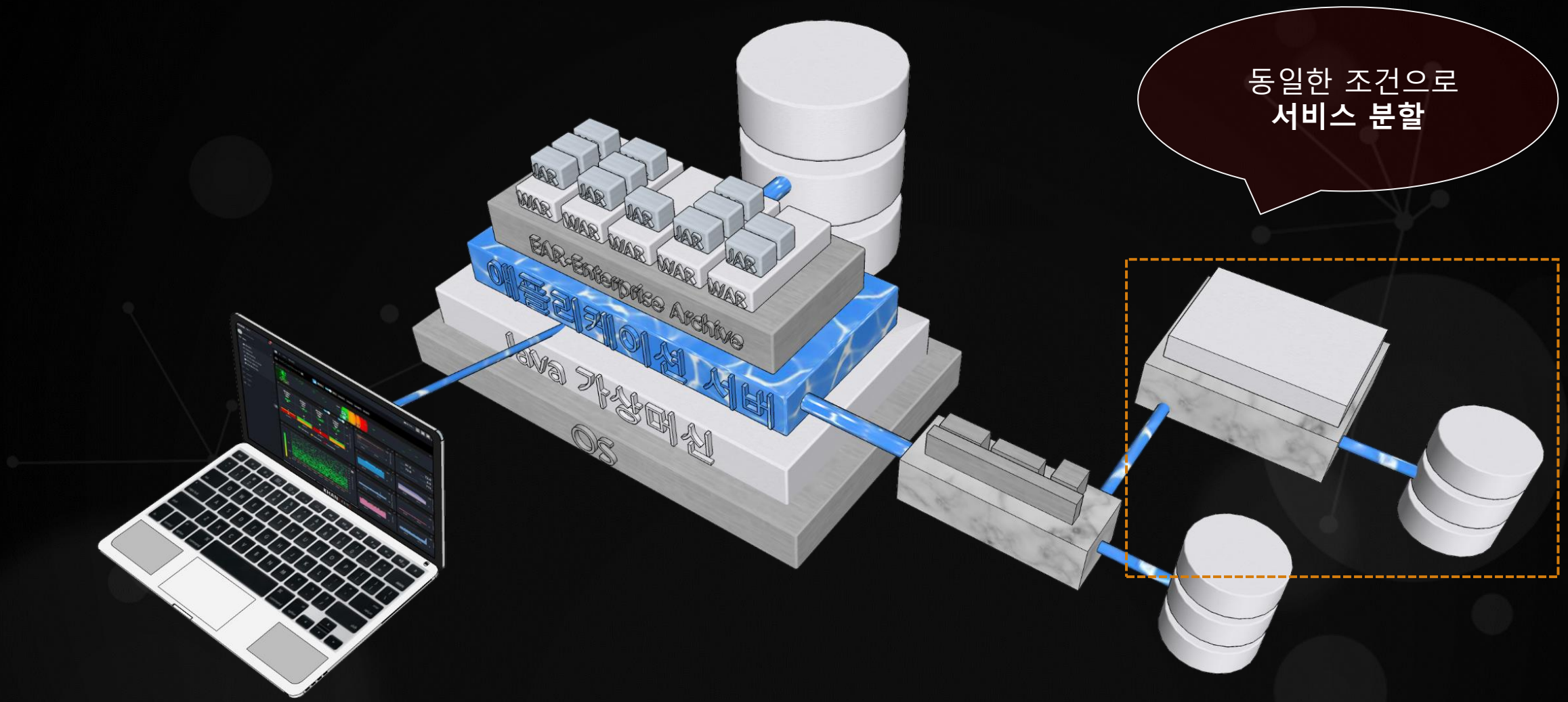
- 모노리스 아키텍처의 특징
  - 견고함
  - 표준화
  - 단일 기술



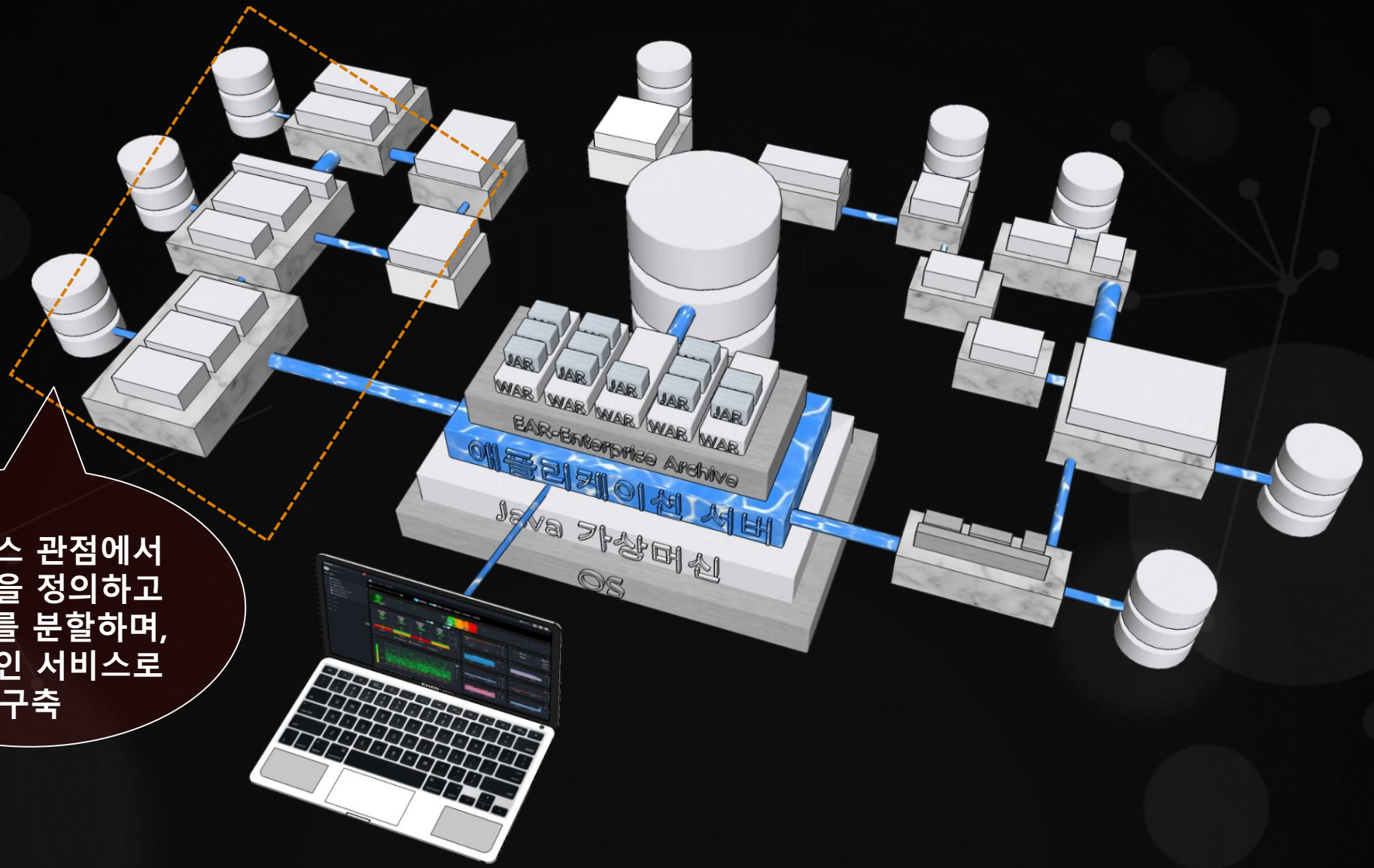
Monolith -> Microservices – Phase 1



모노리스에서 마이크로서비스로 전환 – Phase 2



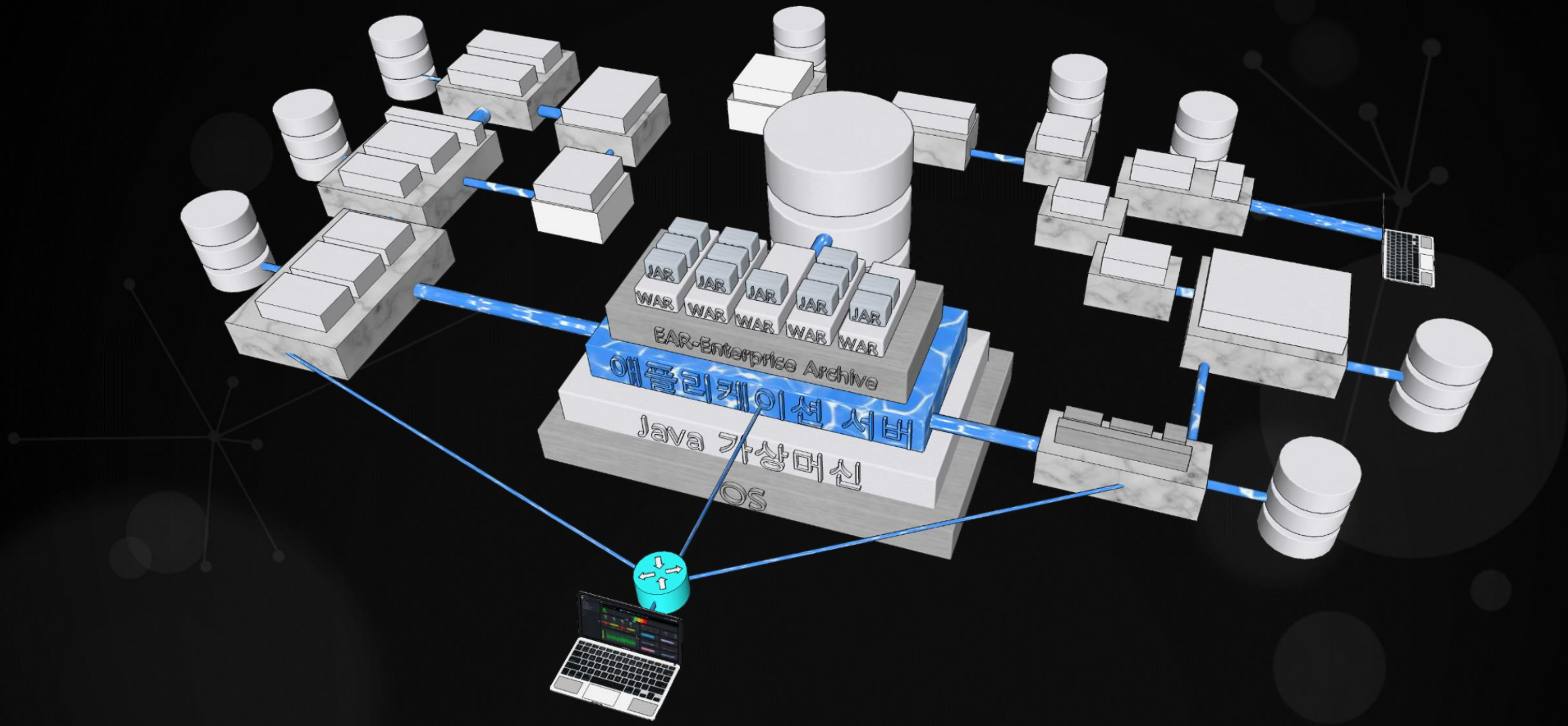
모노리스에서 마이크로서비스로 전환 – Phase 3

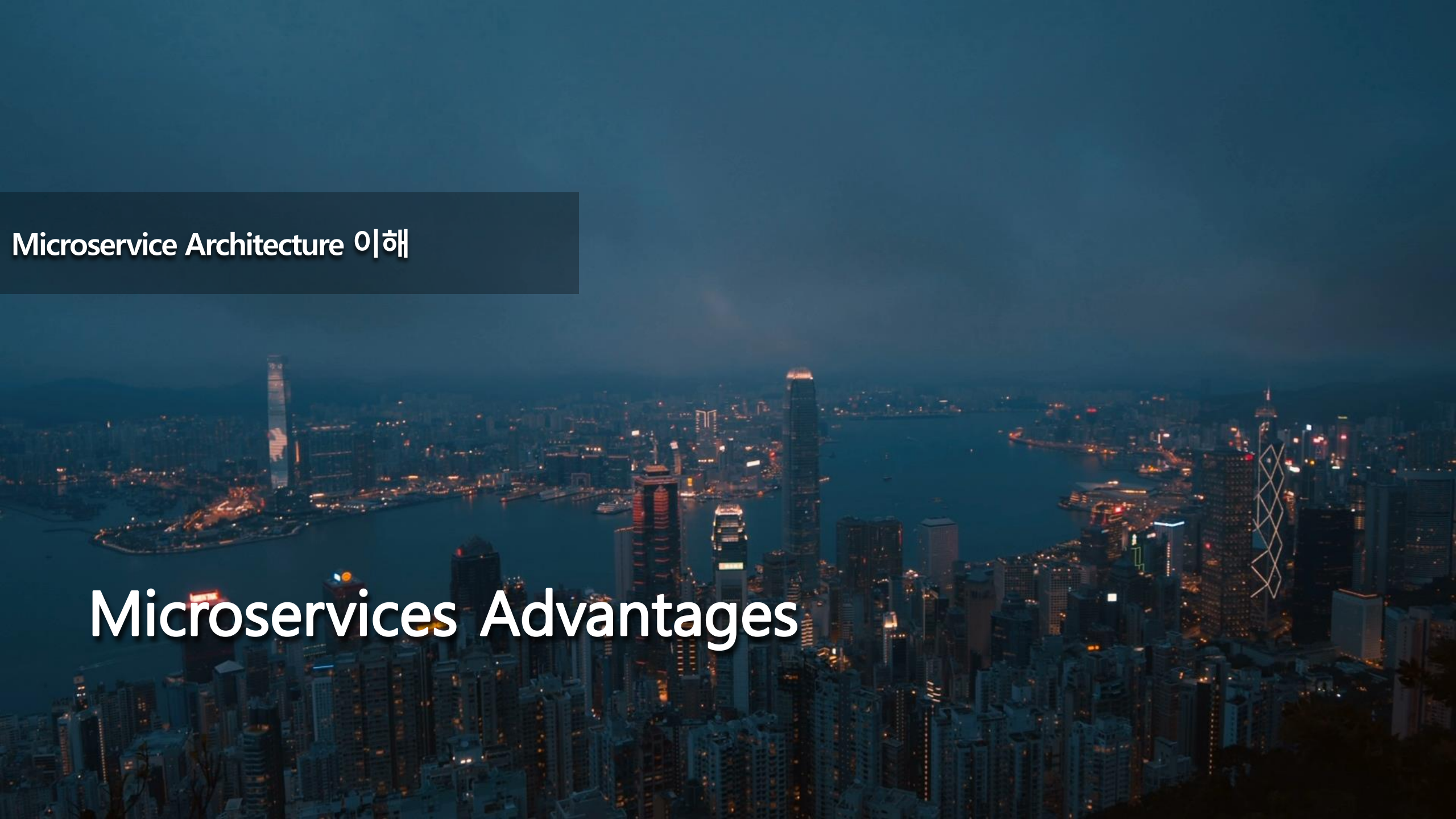


비즈니스 관점에서
도메인을 정의하고
서비스를 분할하며,
독립적인 서비스로
구축

모노리스에서 마이크로서비스로 전환 – Phase 4

- 독립적이고 단순한 서비스가 묶여서 **전체 서비스**를 제공





Microservices Advantages

마이크로서비스 장점

기존 모노리스 아키텍처 탈피	• 개별 마이크로서비스로 확장 가능
서비스 별로 독립적인 운영	• 서비스에 대해 유연한 추가, 변경이 가능 • 서비스의 변화나 장애가 전체 시스템에 영향을 주지 않음 • 서비스마다 성능을 확장
서비스 별로 독립적인 개발	• 모든 서비스에 대해 동일한 프로그램 언어나 DB 등을 사용할 필요 없음 • 서비스에 최적이라고 생각되는 것을 선택 • 최신 기술의 도입/실험에 부담이 없어 개발팀 역량 강화
서비스 간에 독립성이 유지	• 개별적인 문제가 전체에 영향을 주지 않음 • API 디자인 능력 향상
비즈니스 경계와 더 밀접하게 정렬됨	• 변화에 강한'시스템을 실현하기 쉬어짐 • 좀더 유연한 비즈니스 지원 체계 구축
유연한 개발과 운영	• 병렬 개발과 배포 지원 • 신속한 배포


openmaru