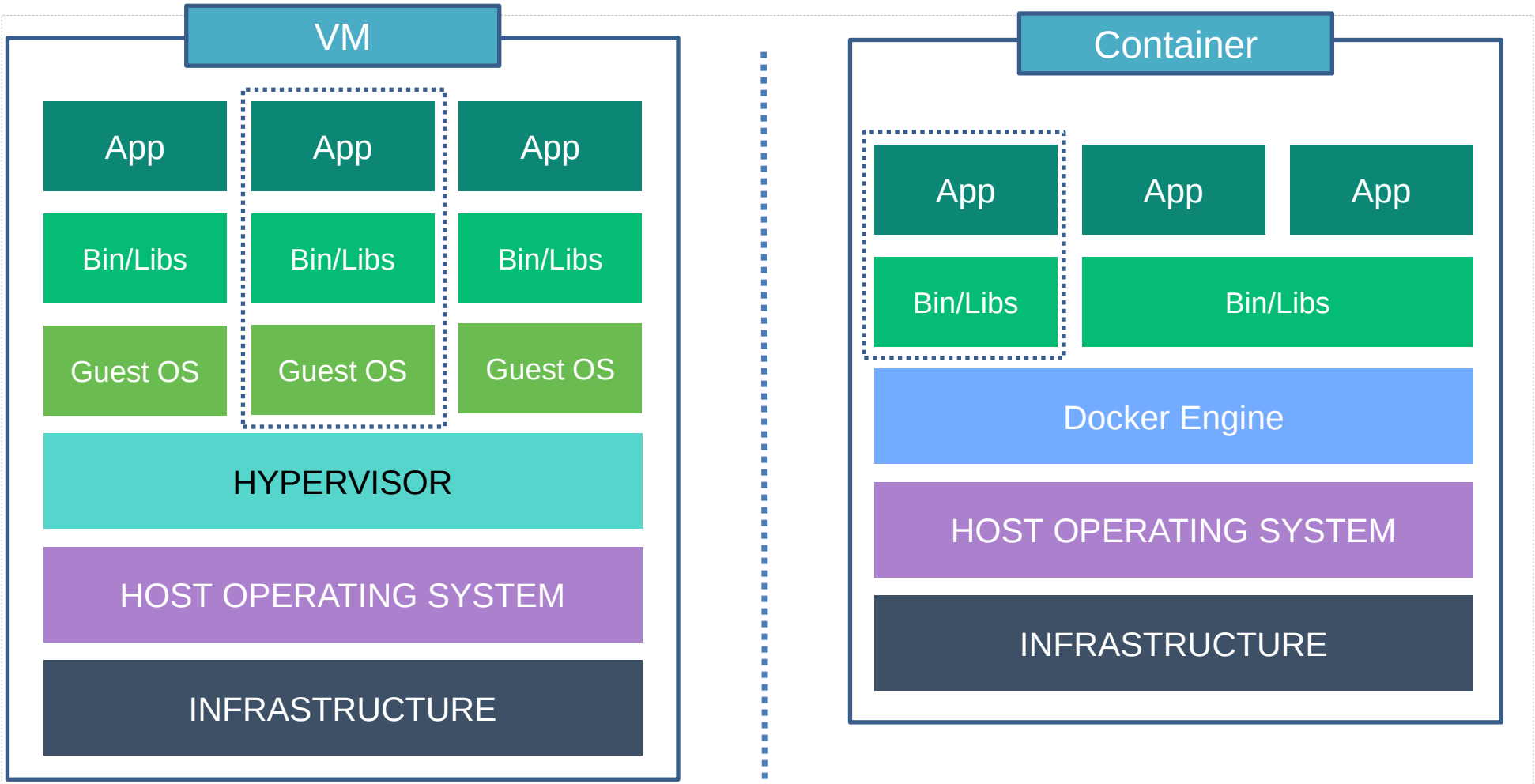
CLOUDZ LABS

Cloud Application 기본설계 가이드

SK 주식회사

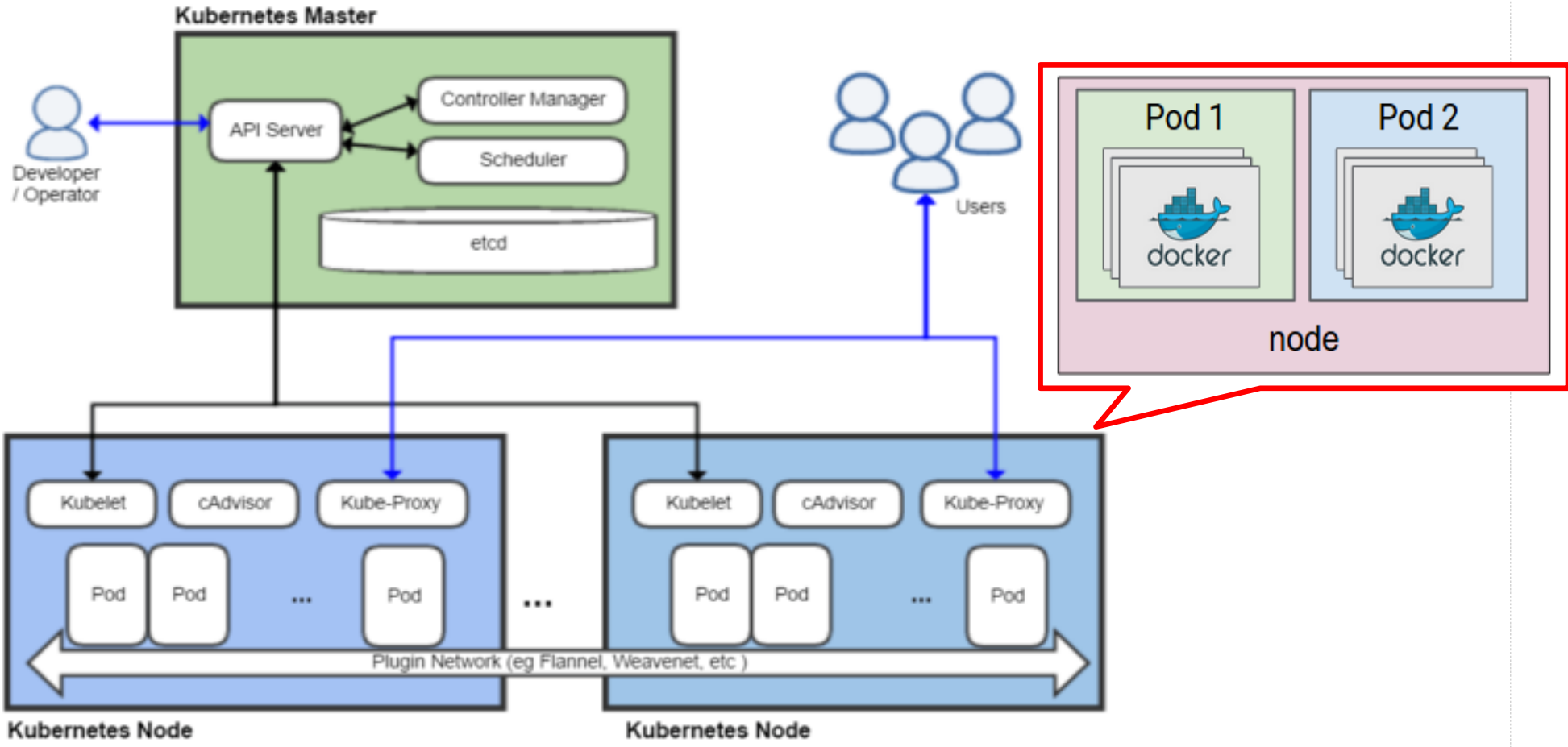
Table of Contents

VM vs Container

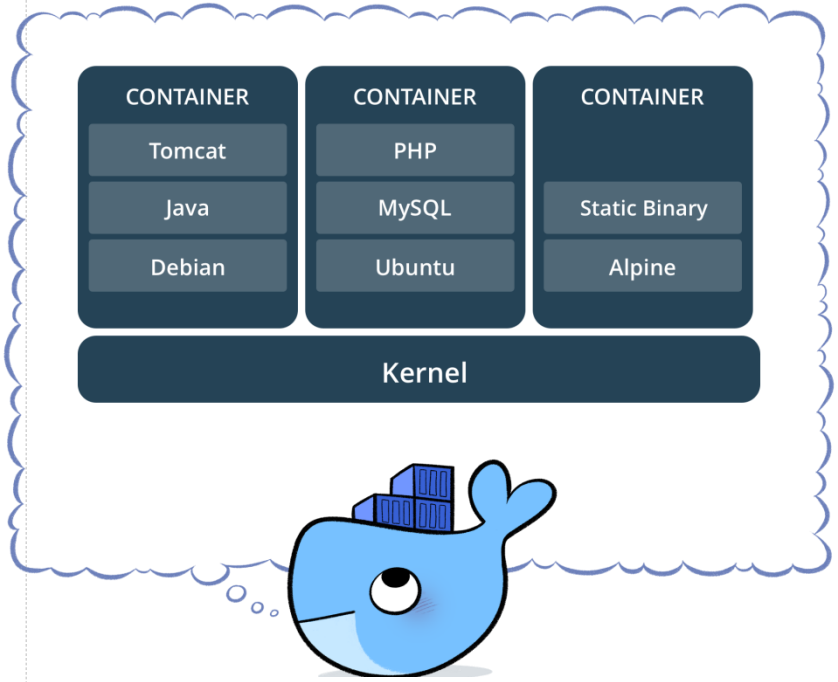


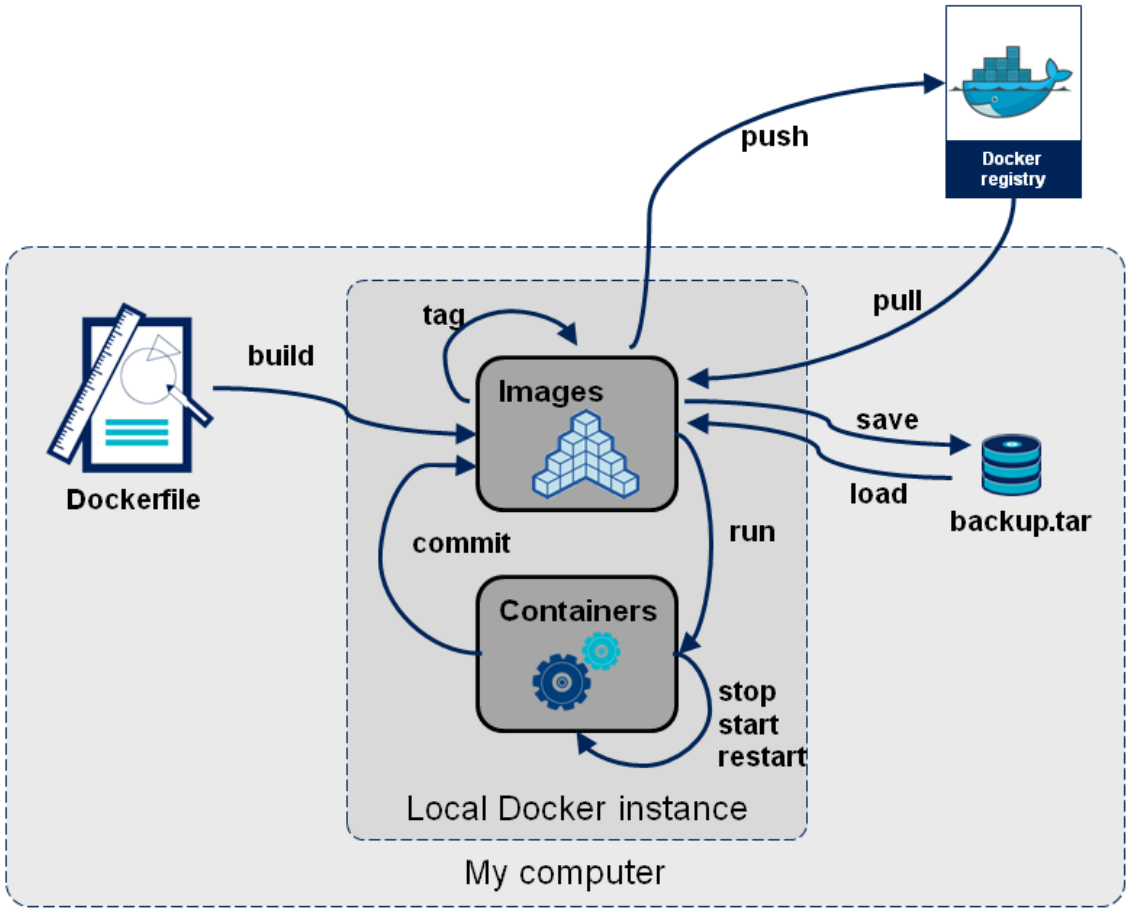
1 Kubernetes 특징

Kubernetes 아키텍처



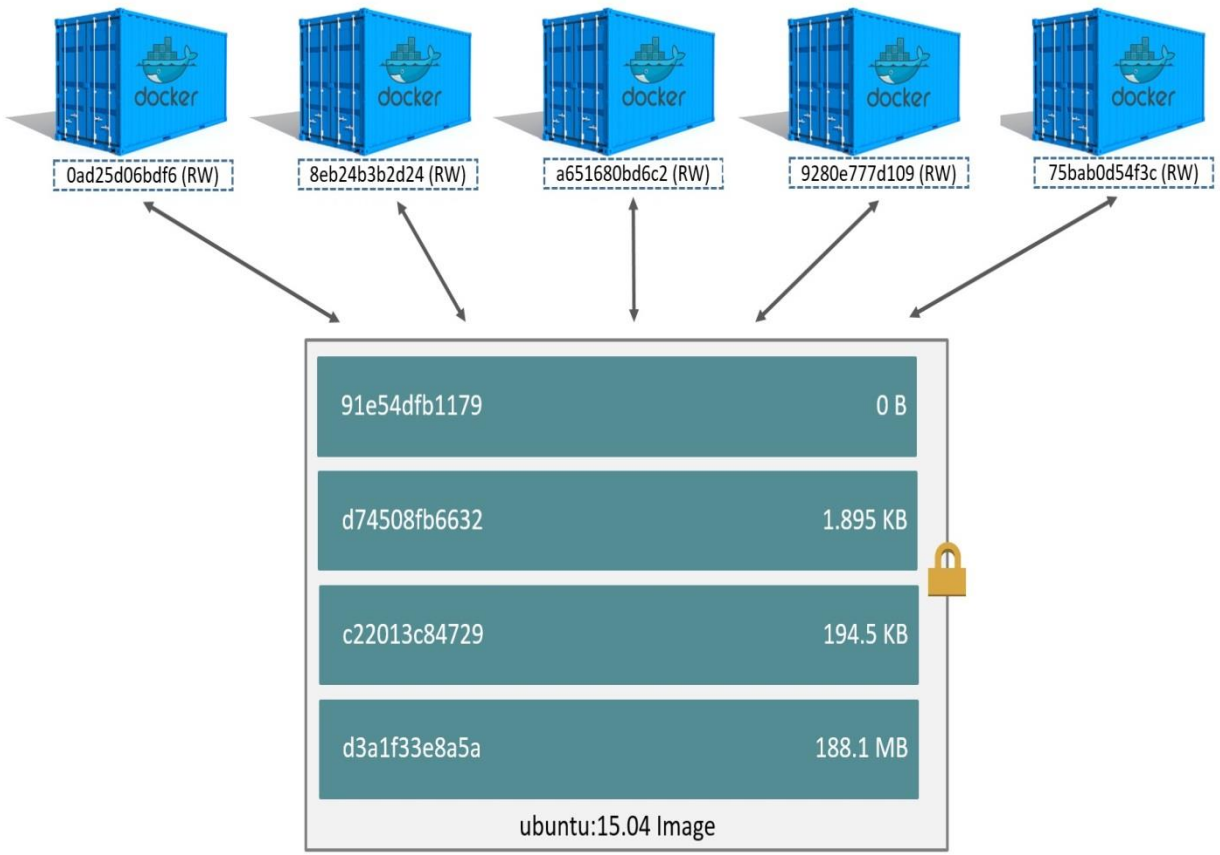
Docker 아키텍처





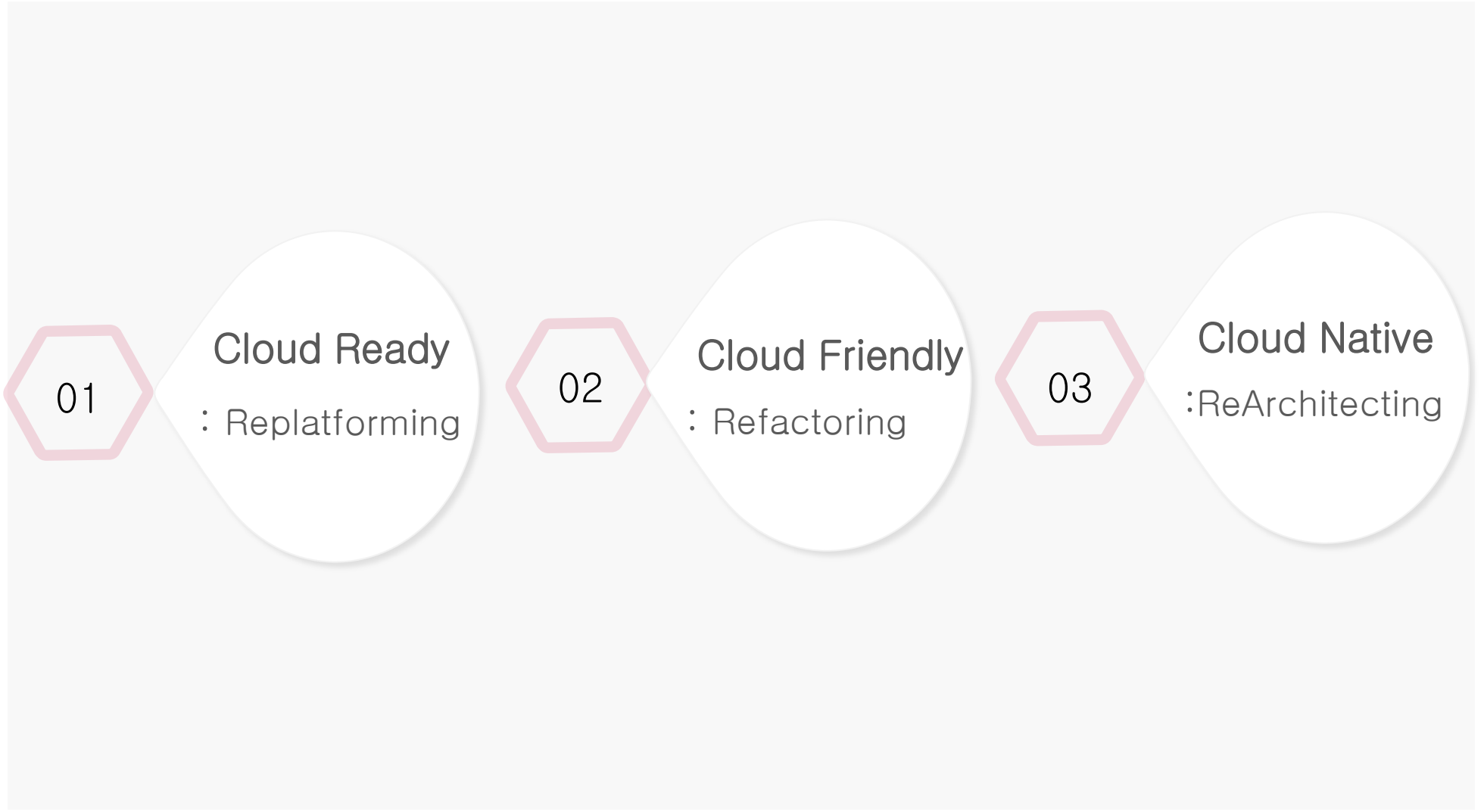
1.2 Kubernetes 특징

Container as a Service



- **Docker Container** Orchestration
- Docker image를 활용한 컨테이너 구성
  - 애플리케이션은 물론 서비스 (DB, Redis 등)까지 컨테이너 화 가능
- Container 단위 Scaling

PART2 클라우드 애플리케이션 유형별 개발 가이드



2. Cloud Application 유형

- 기존의 시스템을 Cloud (PaaS)로 전환 할 때 고려 할 수 있는 애플리케이션 유형.
- 애플리케이션의 특성에 따라서 아래 유형 중 선택 한다.
- 각 유형 별 애플리케이션으로 전환 시 고려해야 할 내용을 뒤에서 자세히 가이드 한다

특징	Cloud Application 유형		
	Cloud Ready	Cloud Friendly	Cloud Native
전환 유형	Replatforming	Refactoring	ReAchitecturing
전환 이점	PaaS 플랫폼을 사용하는 이점	자동으로 스케일링 가능 설정자동화절차 체계화 해서 새로운 환경이나 개발자가 프로젝트에 참여하는데 시간과 비용 최소화	새로운 요구사항이나, 장애등에 빠르게 대응 할 수 있음
전환 고려사항	PaaS에 배포하기 위해서 수정해야 되는 내용 도출	Scalabliity , 설정 자동화, 코드와 환경에대한 내용 분리 , CI/CD	Microservice
기술 요건	PaaS 플랫폼 특징 이해	12factor , 배포 파이프라인	Design for failure, API first Design , Event driven Design
전환 목적	진화된 Cloud Applcation으로 발전하기 위한 단계로 활용	현재 아키텍처 수준에서 클라우드 효과를 최대로 보기위한 전환	변화나 수정에 유연한 구조의 애플리케이션으로 전환

- Cloud Application 유형분리는 절대적인 기준이 아니며, Cloud에 최적화되어 최대의 효과를 얻을 수 있는 애플리케이션으로 전환을 한번에 이루기 어렵기 때문에 단계적으로 진화 가능한 수준이라는것을 보여주기 위한 분리이다.
- 실제로 애플리케이션 전환 시 기존시스템의 특성에 따라 최대의 효과를 볼 수 있는 전환 수준을 도출 해야 한다 .

  (예〉 Cloud ready수준에 scalability만 확보하면 최소의 노력으로 최대의 효과를 볼 수 있다고 분석되면 Cloud Ready단계의 고려사항과 Cloud Friendly 단계의 고려사항 중 scalability 확보를 위한 기술까지만 적용.)
- **Cloud Ready 〈 Cloud Friendly 〈 Cloud Native**

  (Cloud Native 애플리케이션 으로 전환 설계시 Cloud Ready , Cloud Friendly 의 특징을 모두 고려해야함)

[백업] Application Architecture : 12 Factors

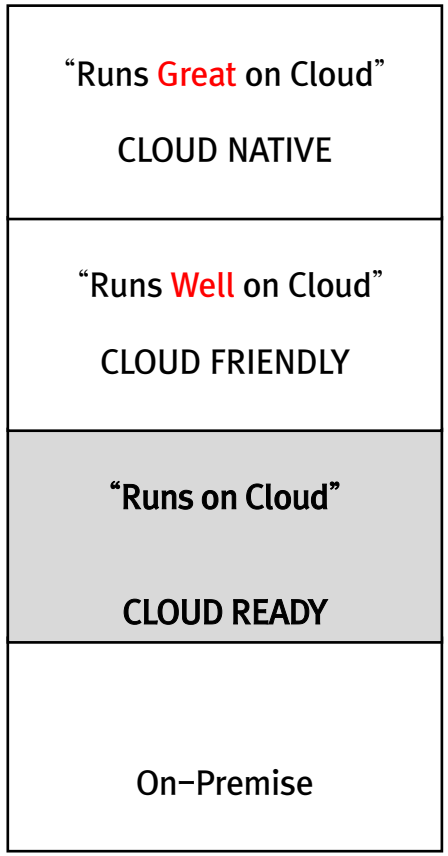
– Cloud Application 구축을 위한 표준 설계 원칙으로 개발 방법 및 개발/운영 환경에 대한 내용을 포함한다.

- https://12factor.net/ko/

12 Factors		달성 방법	PaaS	DevOps	Refactor
1. 동일한 Repo	개발/운영을 하나의 저장소로 관리	DevOps Pipeline		✓	
2. 의존성 설정	소스가 아닌 설정으로 의존성 관리	Maven			✓
3. 환경변수	소스가 아닌 환경변수로 관리	ENV, Spring Config			✓
4. 서비스바인딩	DB, 연계는 서비스바인딩으로 간주	Backing Services	✓		
5. 릴리스 분리	빌드/릴리스/실행의 완전한 분리	DevOps Pipeline		✓	
6. 무상태	무상태 처리를 기본으로 함	Stateless/Shared Nothing			✓
7. Self-Contained	컨테이너가 아닌 Appl. 자체가 URL	Menifest & Container	✓		
8. 동시성	스케일 아웃을 위한 프로세스 모델	Router, Microservices			✓
9. 빠른 Reboot	빠른 부팅과 Graceful 셧다운	Cloud Foundry Container	✓		
10. 실행환경일치	개발계/검증계/운영계 환경 일치	DevOps Pipeline		✓	
11. 중앙 로깅	플랫폼에서 이벤트 로그를 수집	Metric & Logging	✓		
12. JOB 관리	DB변경과 같은 JOB을 소스에 포함	Infrastructure as a Code			✓

2. 1 Cloud Ready

Cloud Ready 수준의 애플리케이션으로 전환시 고려 사항을 가이드 한다.



Goal

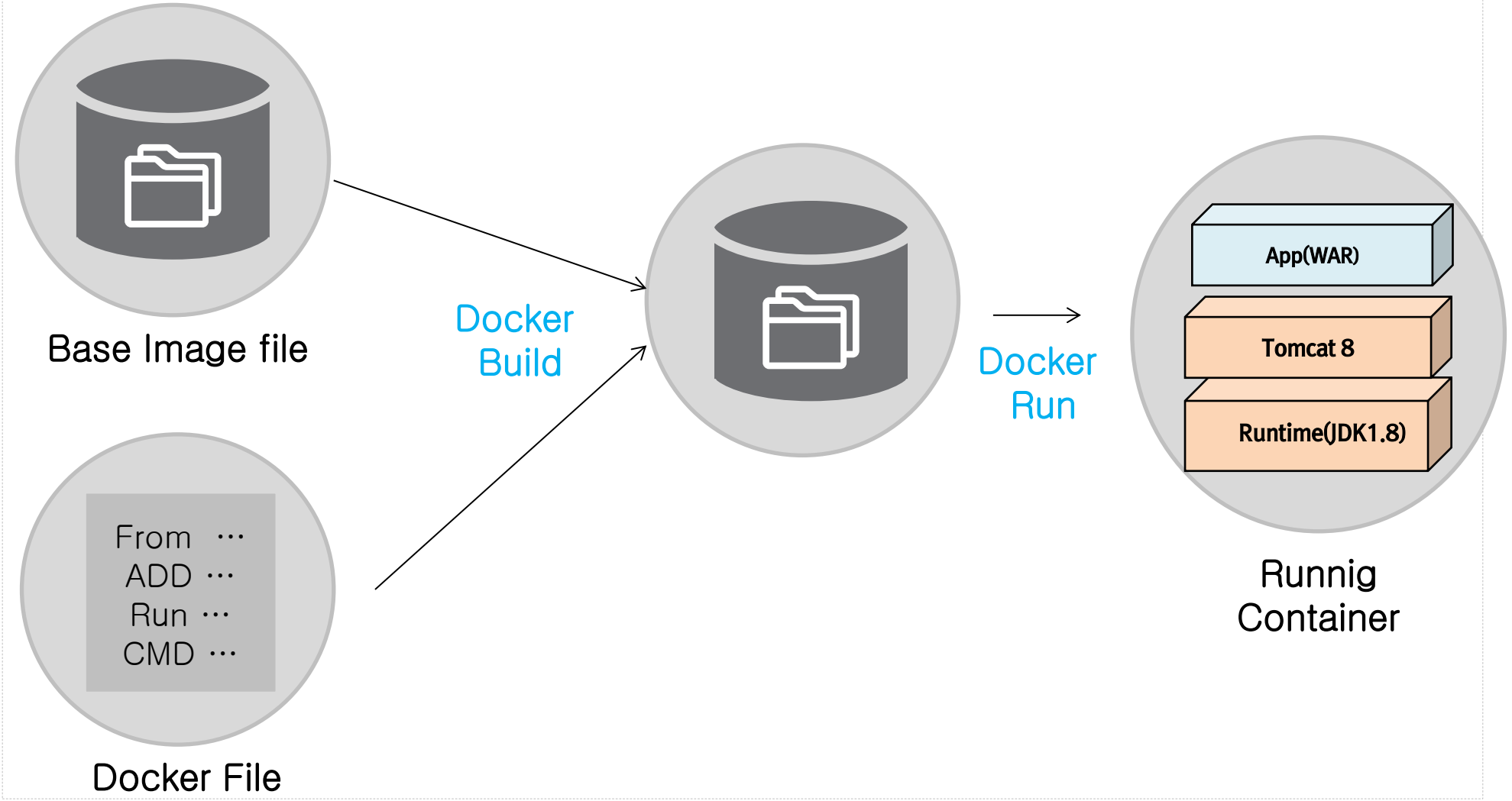
- PaaS에 애플리케이션 정상적으로 배포

목차

- Replatforming 고려 사항
- Replatforming Path를 활용한 Replatforming 계획 수립
- Buildpack을 고려한 애플리케이션 마이그레이션
- PaaS 플랫폼 특성을 고려한 애플리케이션 마이그레이션

2.1 Cloud Ready : Replatforming 고려사항

시스템을 기존 운영환경과 다른 환경인 cloud 환경으로 전환하기 위해서 Replatforming 과정이 필요하다.
Replatforming 시에는 애플리케이션을 배포하면 런타임 환경을 구성해주는 Docker Container Base Image 를 고려해야 한다



2.1. Cloud Ready : Application Replatfoming

기존 시스템을 **PaaS** 환경으로 마이그레이션할 때 컨테이너를 구성할 **WAS**의 종류와 버전을 확인하고 마이그레이션 해야하며 성공적인 마이그레이션을 위해 아래와 같은 내용을 고려해야 한다.

WAS

고려 사항	설명
WAS 종류 및 버전	빌드팩에서 지원되는 WAS 종류와 WAS버전을 확인하고 마이그레이션 해야 함
특정 WAS 및 WAS 버전에 대한 Dependency	WAS 변경 시 발생하는 Character Encoding 오류 수정
	특정 WAS의 라이브러리를 이용한 API (JNDI, Sevlet 등)가 구현되어 있는 경우 해당 로직 수정 및 대체 필요
	J2EE (EJB, JTA, JMS, JCA 등) API를 사용하고 있는 경우 해당 API가 지원 되는 WAS로 마이그레이션 해야함
	WAS 변경 시 JSP파일 등의 사소한 문법 오류 수정
	어플리케이션 구동을 위해 WAS 설정 파일에 추가할 내용이 있는지 확인 ex) JNDI 설정 등 추가할 내용이 있는 경우 최대한 소스 내에서 해결하고 빌드팩 커스터마이징은 지양함
Environment 설정	WAS 구동을 위해 추가할 환경 변수가 있는지 확인 ex) 톰캣 버전 명시가 필요한 경우 manifest.xml 파일이나 CF CLI를 통해 설정 cf set-env my-application JBP_CONFIG_TOMCAT '{tomcat: { version: 7.0.+ }}' https://github.com/cloudfoundry/java-buildpack/blob/master/docs/container-tomcat.md

2.1. Cloud Ready : Application Replatfoming

JDK 및 라이브러리는 Docker Base Image에서 지원하는 버전을 확인하여 마이그레이션 해야 하며 성공적인 마이그레이션을 위해 아래와 같은 내용들을 고려해야 한다.

JDK

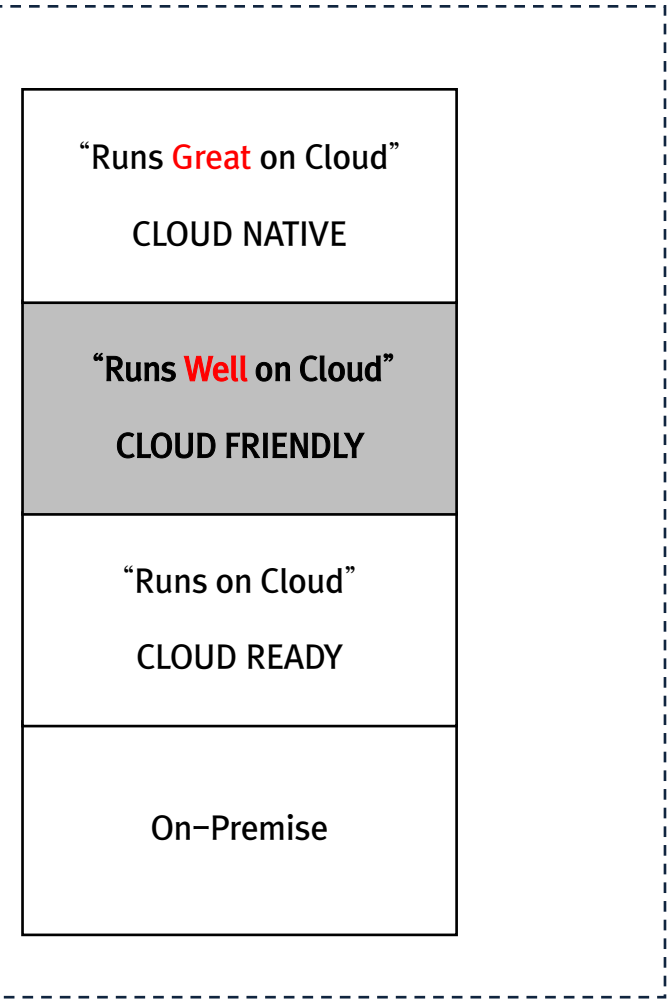
고려 사항	설명
JDK 버전	Base Image JDK 버전을 확인하고 업그레이드 해야함

Library

고려 사항	설명
Spring Framework 버전	Spring FWK 사용 시 Base Image에서 지원되는 버전을 확인하고 업그레이드 해야함
WAS 및 JAVA에 대한 Dependency	WAS, JDK 변경 및 업그레이드에 따른 라이브러리 오류 발생 시 최신 library로 대체하거나 불필요한 library는 삭제하고 필요한 경우 소스를 수정함 ex) JDK 버전 업그레이드시 JDBC library 오류가 발생할 경우 JDBC도 최신 library로 변경해야 함
3rd party library	애플리케이션에서 사용하고 있는 3rd party library가 PaaS에서 구동 가능한지 확인 필요에 따라 PaaS 환경에 맞는 라이브러리로 교체 (ex. 설치형 라이브러리)
기타	사용하고 있는 3rd party library의 라이선스가 Cloud 환경에서 적합한지 확인

2.2 Cloud Friendly

Cloud Friendly 수준의 애플리케이션으로 전환 시 고려사항을 가이드 한다.



Goal

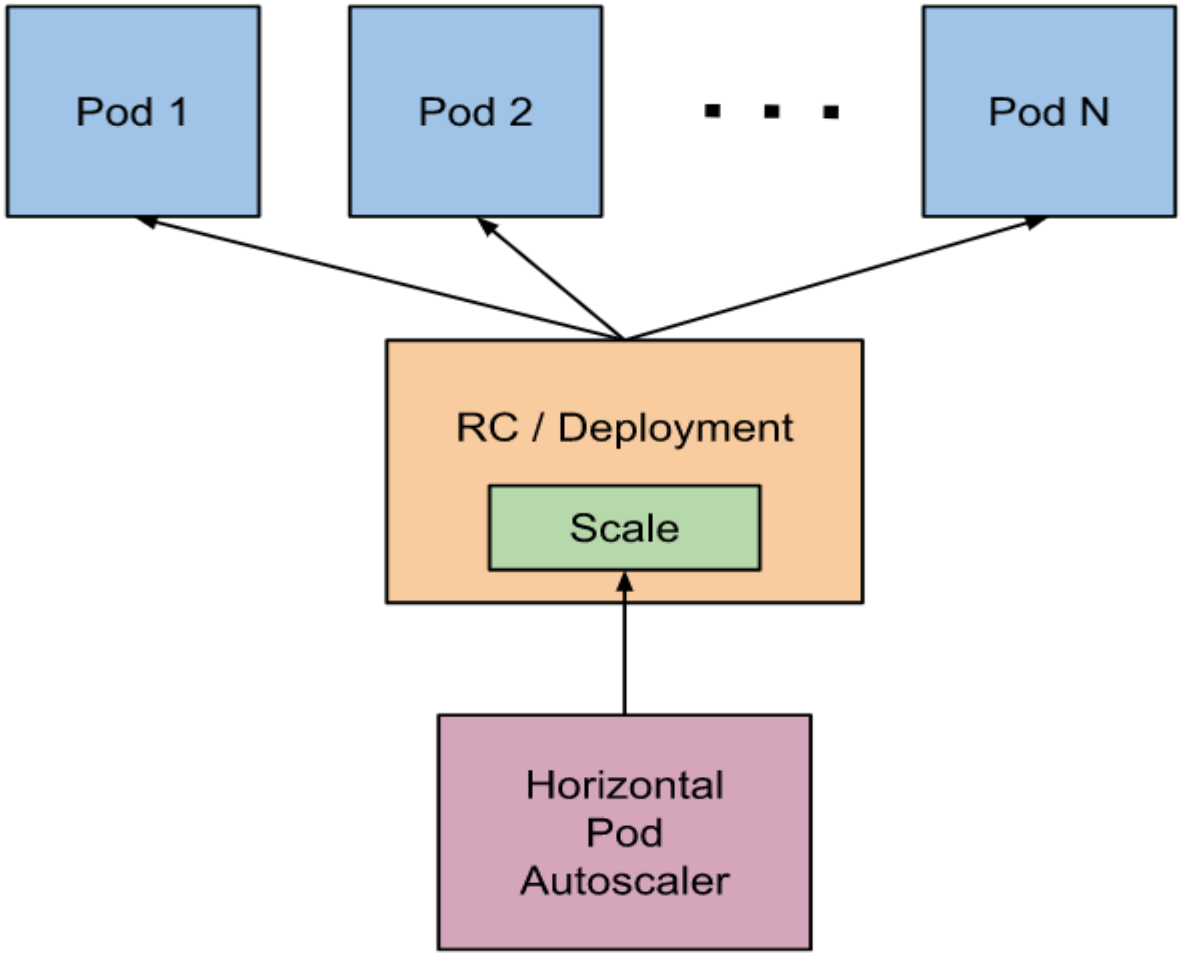
클라우드의 장점을 활용 할 수 있는 애플리케이션 배포

목차

- **Scalability** 확보를 위한 애플리케이션**Refactoring** 고려사항
- 새로운 환경이나, 새로운 개발자가 빠르게 개발에 투입될 수 있는 애플리케이션으로 **Refactoring** 고려사항
  - **Configuration**
  - 의존성 관리
  - **Backing Service**
- 지속적인 배포를 위한 배포파이프라인 구성
  - **CI/CD**

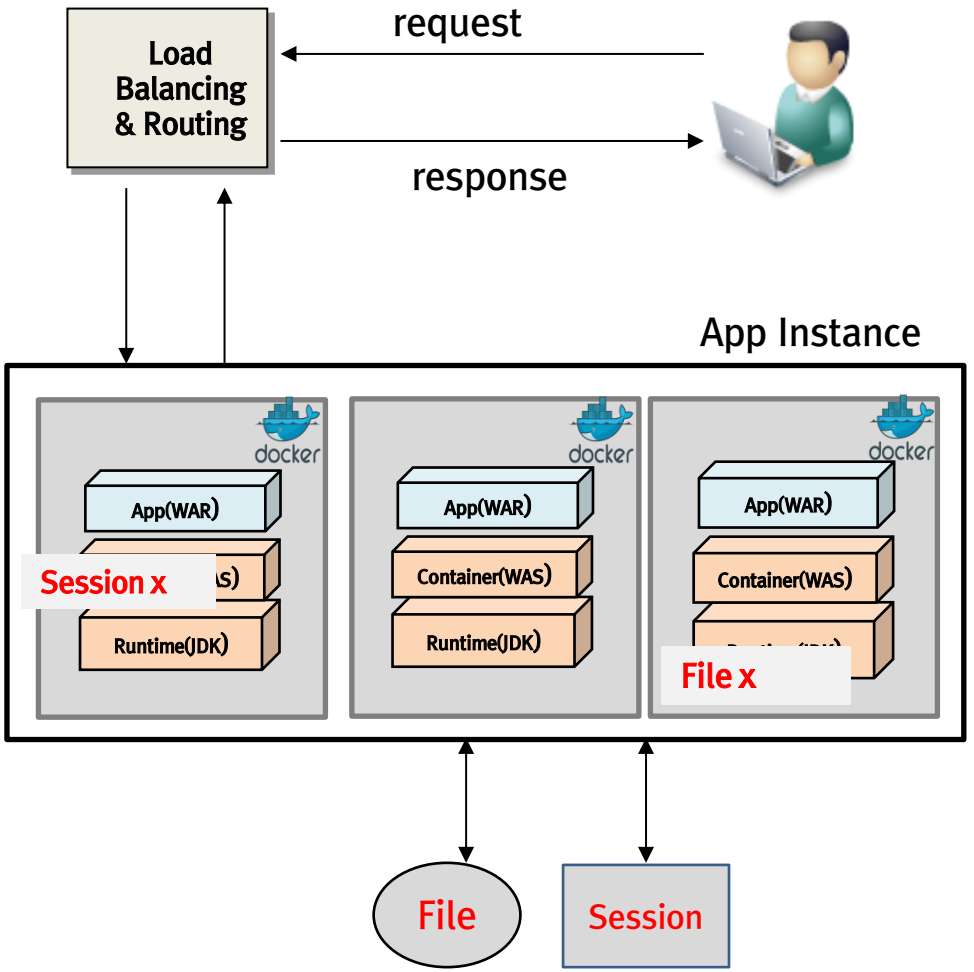
2.2 Autoscaling

Horizontal Pod Autoscaler (HPA)를 활용하여 애플리케이션을 자동으로 다중화 구성한다.



2.2 Cloud Friendly : Scalability확보

12factors의 요건으로 PaaS에서 애플리케이션이 정상적으로 확장(scale out)되서 운영 되기 위해서는 아래의 조건을 고려해야 한다.



- ✓ 애플리케이션의 **Scablilty** 확보를 위해 아래와 같은 조건을 만족시켜야 한다.
  - Stateless
  - Shared nothing
- ✓ **Stateless & Shared nothing** 해야하는 이유?
  - 프로세스가 재실행될 때 (repush, restage) 로컬의 상태(메모리, 파일 등)를 초기화 함
  - 현재 나의 요청이 미래에도 같은 프로세스에서 동작하리라는 보장 없음(애플리케이션이 하나이상의 프로세스로 동작 가능함: scale out 상태)
- ✓ 구현방법
  - 메모리/파일을 사용할 경우 단일 트랜잭션 내에서 읽고,쓰고 등의 모든 작업을 처리
  - 세션 상태 데이터의 경우 애플리케이션 외부 서비스 (redis, gemfire등) 에 저장 – sticky session X
  - 유지될 필요가 있는 데이터(파일) 외부서비스(object storage, DB등) 에 저장
  - 기존에 전역변수로 저장해서 사용하는 다음 트랜잭션까지 사용하던 데이터도 외부서비스에 저장

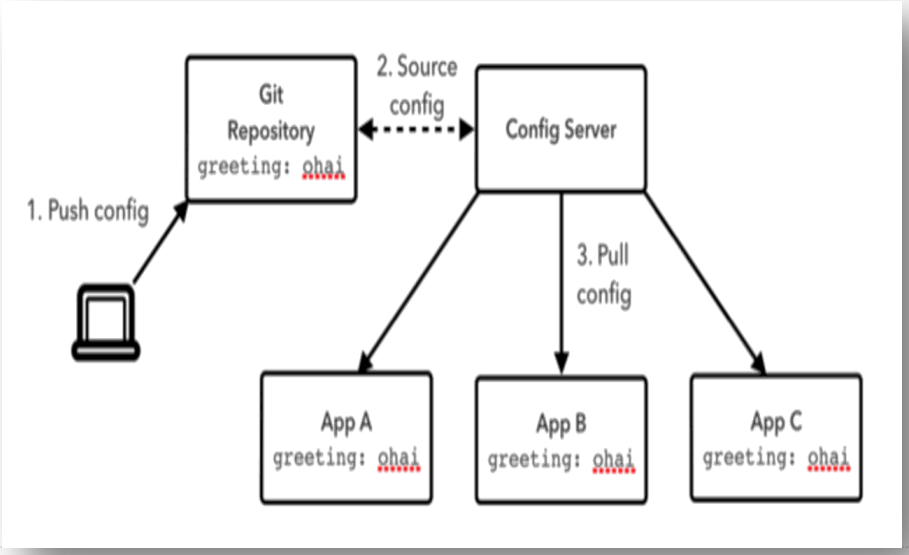
2.2 Cloud Friendly : Configuration

12factors 요건 중 하나로 애플리케이션은 하나의 코드 베이스를 유지하고 배포환경에 따라 달라질 수 있는 내용은 설정 파일로 작성해 코드로 부터 분리하고 런타임 때 코드에 의해 읽혀야 한다.

배포 환경에 따라 파일을 아래와 같은 방식으로 분리하고 런타임 시에 알맞은 프로퍼티 파일을 사용할 수 있도록 액티브 프로퍼티 값을 외부에서 환경변수로 주입한다

- application.yml
- application-**default**.yml
- application-**dev**.yml
- application-**stg**.yml
- application-**prod**.yml

manifest.yml
env:
Spring_PROFILES_ACTIVE: prod



〈spring config 서버〉

- ✓ **분리해야 하는 설정 정보**
  - DB, Cache 등 **Backing Service**를 처리하는 리소스
  - 외부 서비스(**S3, Twitter** 등)의 인증 정보
  - 각 배포마다 달라지는 값(**canonical hostname** 등)
- ✓ **변경되는 설정 정보가 존재하면 안 되는 곳**
  - 코드
  - 단일 프로퍼티 파일
  - 빌드 (한번의 빌드로 여러 번 배포가 가능하므로)
  - 앱 서버 정보 (**JNDI** 데이터소스 등)
- ✓ **구현 방법**
  - 환경변수에 저장 (**manifest.yml, vcap_services** 활용)
  - 설정 파일을 중앙화하여 관리하기 위한 서비스인 **Config**서버 구현(**Spring cloud config** 서버 활용가능)

2.2 Cloud Friendly : Backing Service

12factors 요건 중 하나로 애플리케이션에서 네트워크를 통해 이용하는 모든 서비스들을 코드 변경 없이 자유롭게 연결 및 분리 할 수 있도록 **URL**과 같은 엔드포인트를 통해 접근하도록 한다.

✓ **Backing service란?**

- 네트워크를 통해 이용하는 모든 서비스 (DB, Cache, Messaging/Queueing System 등)

✓ **사용 목적**

- 서비스의 정보는 환경에 따라 변경될 수 있는데 , 이 때 코드 변경 없이 자유롭게 연결하거나 분리할 수 있도록 하기 위함

✓ **활용방법**

- PaaS의 서비스바인딩 기능을 이용하여 쉽게 활용 가능

1. 사용할 서비스 인스턴스 생성 (cf cli를 통해 아래 cf command로 생성)

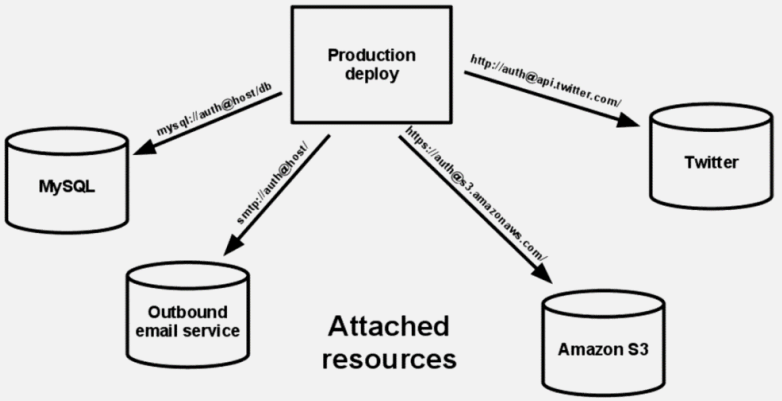
   1-1. PaaS 마켓플레이스에 서비스 존재 :cf create-service [service] [plan] [서비스명]

   1-2. PaaS 마켓플레이스에 서비스 미존재 : cf create-user-provided-service [서비스명] -p "[url]"

2. 서비스 바인딩

cf bind-service [애플리케이션명] [서비스명]

3. 애플리케이션은 서비스 사용시 **서비스명에만** 의존



2.2 Cloud Friendly : 의존성 관리

12factor 요건 중 하나로 애플리케이션이 동작하는 모든 시스템(로컬, 개발, 운영등) 에서 동일한 라이브러리(버전)를 사용할 수 있도록 하고, 새로운 개발자가 투입되었을 때 소스코드만 체크 아웃 받아서 바로 개발환경을 구축할 수 있도록 의존성 도구를 이용해서 라이브러리를 관리해야 한다

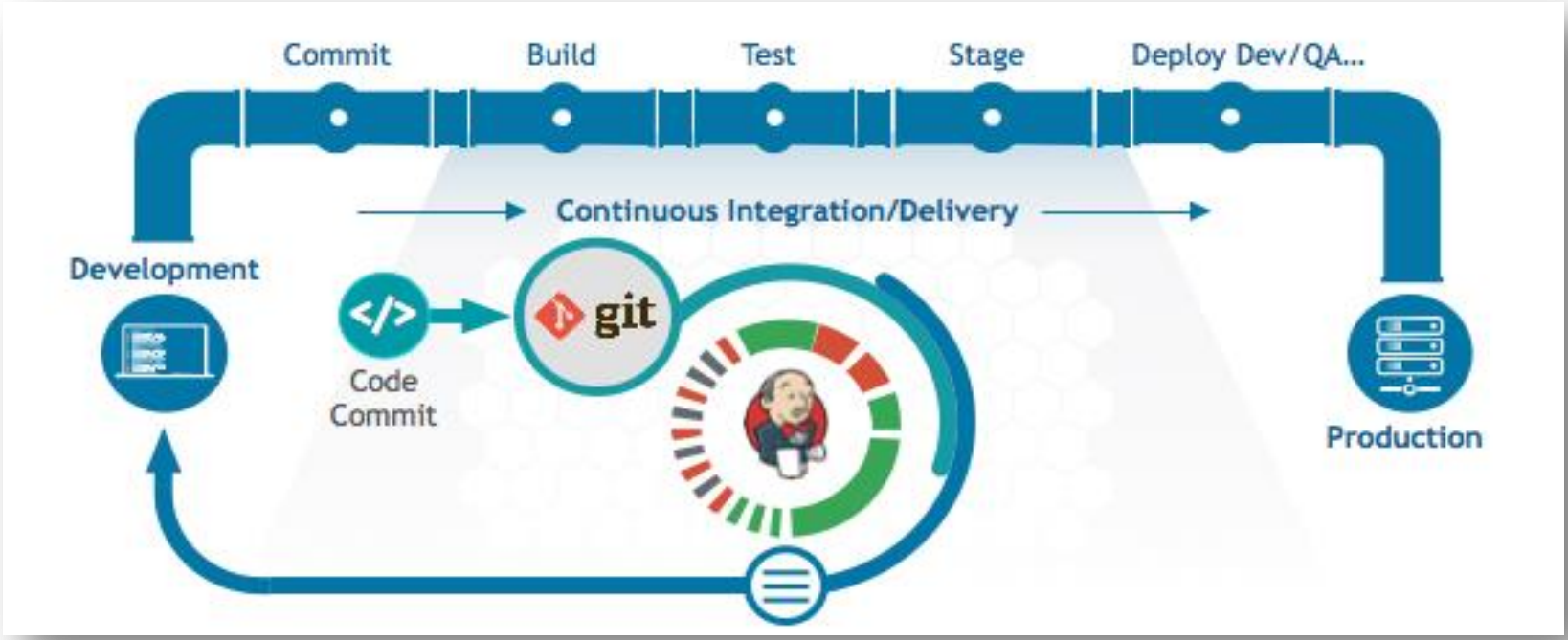
Maven, Gradle 같은 의존성 관리 도구 활용

```
<!-- webjars -->
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>3.2.0</version>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>jquery</artifactId>
    <version>2.1.1</version>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>materializecss</artifactId>
    <version>0.97.5</version>
</dependency>
<!-- end of webjars -->

<!-- jackson -->
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-mapper-asl</artifactId>
    <version>1.9.7</version>
</dependency>
<!-- end of jackson -->

<!-- swagger -->
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.2.2</version>
    <scope>compile</scope>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.2.2</version>
    <scope>compile</scope>
</dependency>
```

〈maven 의 pom.xml 파일 〉

2.2 Cloud Friendly : CI/CD

빠르고 지속적으로 질 좋은 시스템을 제공하기 위해 자동화된 빌드, **TEST**, 배포를 파이프라인으로 구성해
지속적 통합(**Continuous Integration**) 과 지속적 배포 (**Continuous Delivery**)가 가능하도록 해야 한다



빌드 파이프라인

버전관리 도구에 소스를 커밋하면 CI 서버가 이를 감지해서 빌드하고, 자동화 테스트 후 배포하는 단계가
파이프라인처럼 연결되어 있는 것을 의미한다.

2.2 Cloud Friendly : CI/CD

배포 파이프라인을 구축 할 때 아래와 같은 항목이 고려되어야 한다.

항목	설명	활용도구
CI Server	빌드 프로세스를 관리 하는 서버	**Jenkins** , Hudson
SCM (source Codemanagement)	소스코드 형상관리 시스템으로 **CI** 서버에서 소스 코드 변경을 폴링해서 감지할 수 있도록 소스코드는 형상관리 시스템으로 관리되어야 함	**SVN , Git**
Build Tool	형상관리 시스템의 코드를 배포가능한 형태로 빌드 하는 도구	**Maven,** ant , gradle
Test Tool	작성된 테스트 코드에 따라서 자동으로 테스트를 수행해 주는 도구로 **Build Tool** 의 스크립트에서 실행됨	**Junit**
Inspection Tool	프로그램을 실행하지 않고, 소스코드 자체로 품질을 판단할 수 있는 정적 분석도구	Checkstyle, findbug
CF CLI (배포)	**PaaS**에 빌드된 애플리케이션이 배포될 수 있도록 **CF CLI** 가 설치되어야 함	CF CLI
배포 파이프라인 Dash board	배포 파이프라인의 상태를 시각화 할 수 있는 대시 보드	Jenkins Delivery Pipeline Plugin

2.3 Cloud Native

Cloud Native 수준의 애플리케이션으로 전환 시 고려사항을 가이드한다.

Goal

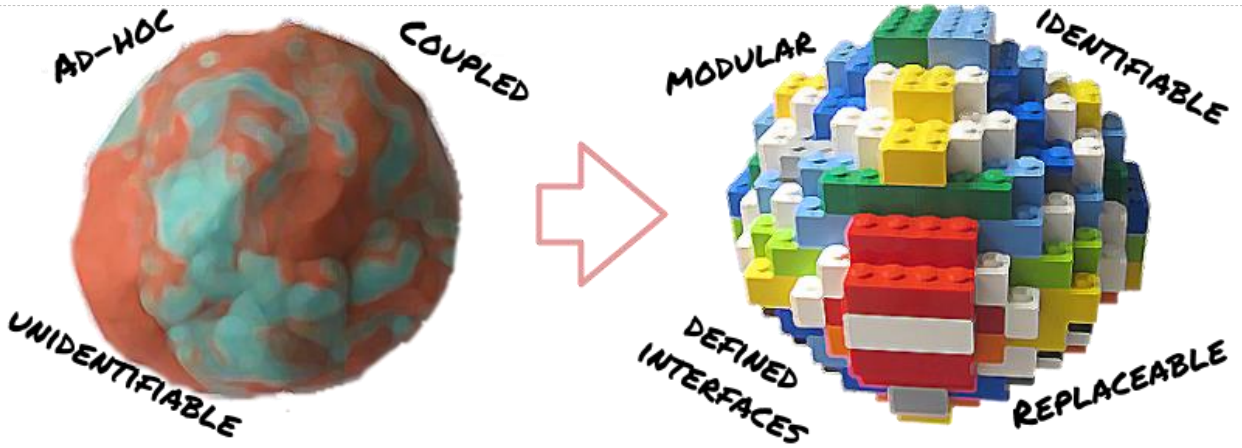
마이크로 서비스로 애플리케이션 배포

목차

- 마이크로 서비스란
- 마이크로 서비스 분해 원칙
- 마이크로서비스 분리 워크샵
- 마이크로 서비스 **Architecture**

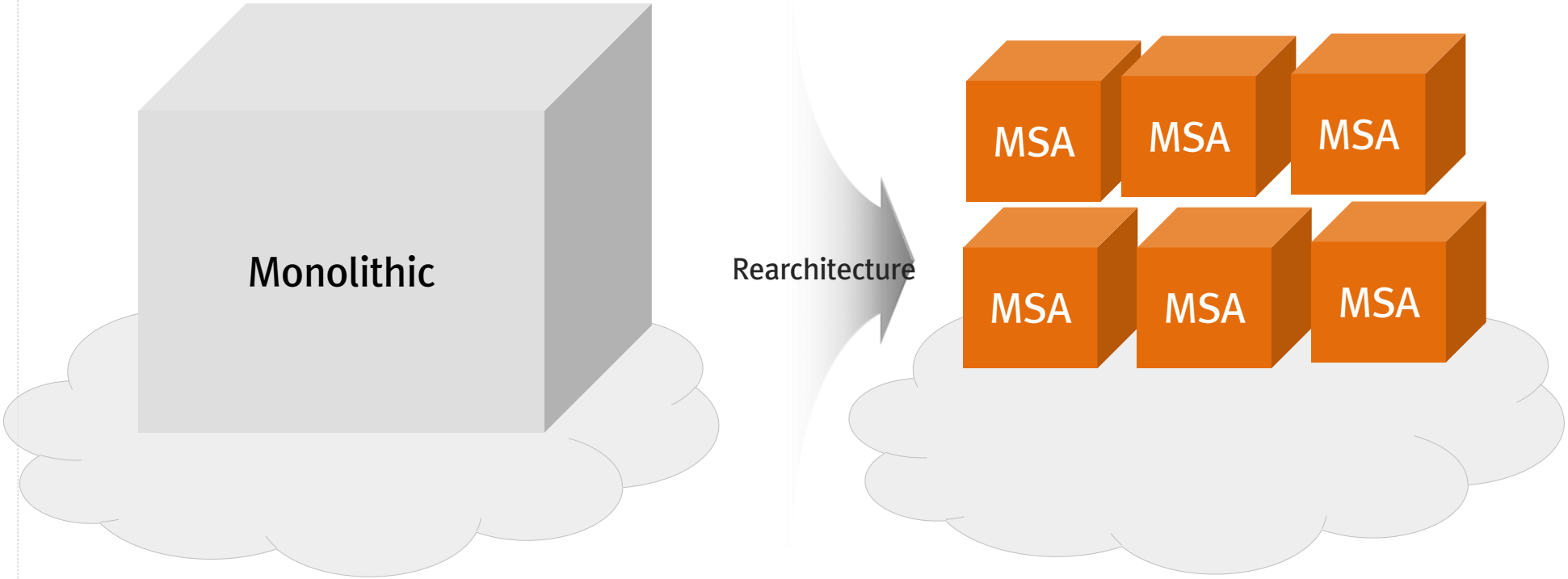
2.3 마이크로서비스란

하나의 아키텍처 접근 방식으로, 시스템은 작은 서비스들로 구성되어 구축되며, 각 서비스는 각자의 프로세스를 보유하고, 가벼운 프로토콜을 통해 통신한다". – 마틴 파울러, 제임스 루이스 –



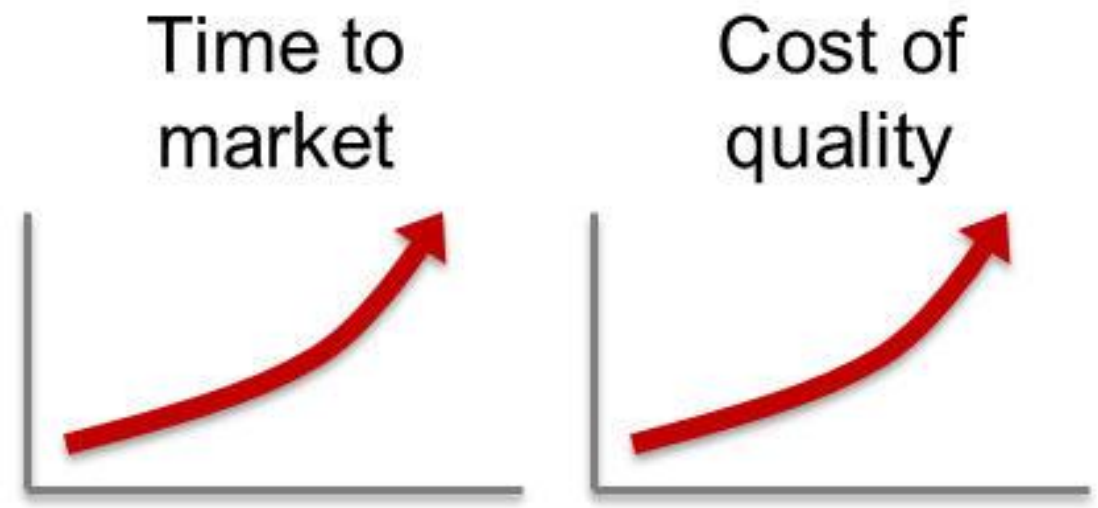
2.3 마이크로서비스란

기존의 모놀리식 애플리케이션을 작은 단위로 쪼개서 운영하는 방식입니다.



2.3 마이크로서비스 목적

마이크로서비스 아키텍처로 달성 해야 하는 비즈니스 요구사항이 명확히 정의되어야 함!



"작고, 독립적인"

마이크로 서비스 적용 대상 애플리케이션

어떤 애플리케이션이 마이크로서비스로
전환되어야 할까?

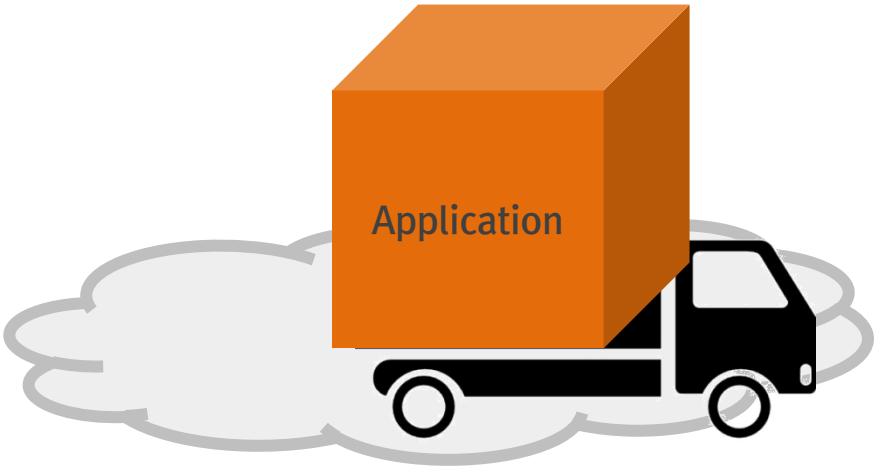
Pain Point 도출해 보기

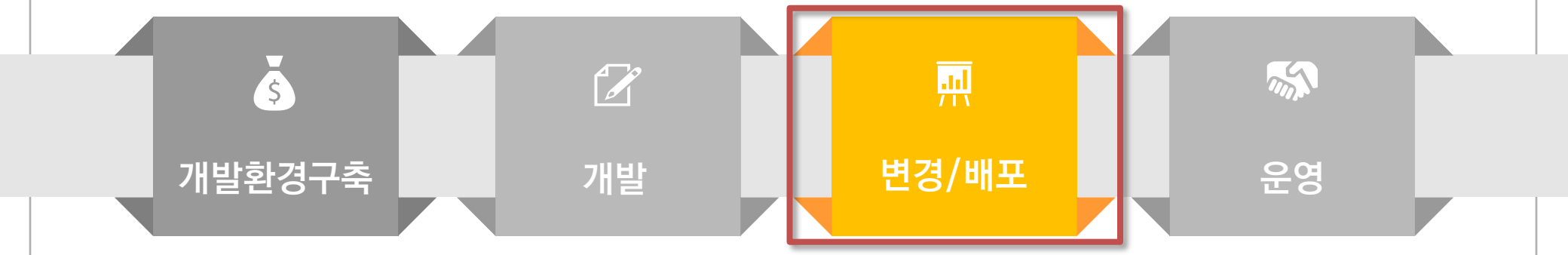
2.3 마이크로 서비스가 답이 될 수 있는 Pain Point

모놀리식 애플리케이션은 변경 주기가 상대적으로 큼 .

적시 배포(서비스 출시)

서비스 출시시기가 더 빨라질 수 없나?
변경 영향도를 최소화 할 수는 없나?



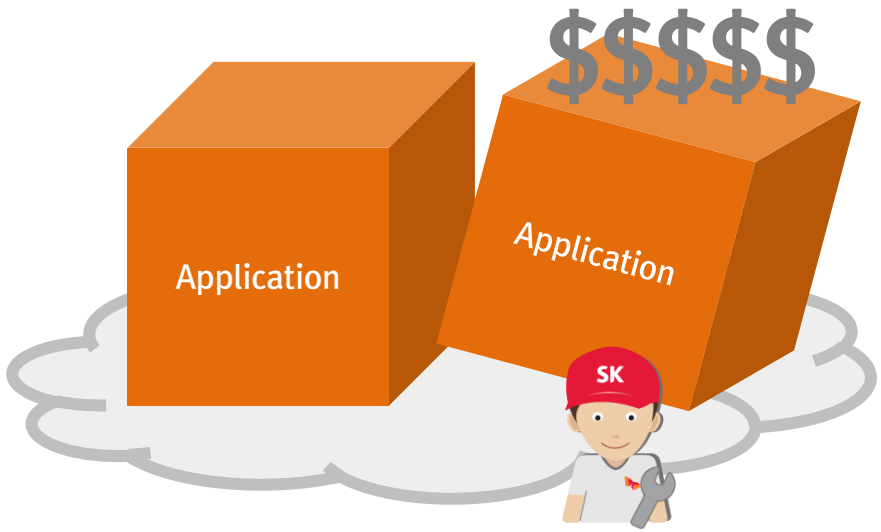


2.3 마이크로 서비스가 답이 될 수 있는 **Pain Point**

전체 VM을 스케일링하기보다는 작은 단위의 스케일링이 요구됩니다.

용이한 스케일링과 효율적인 이중화

Appl.이 원하는 자원이 즉시 제공 가능한가?
자원 추가 시 서비스 영향도는 없는가?
필요한 서비스만 이중화/삼중화 할 수 없는가?



개발환경구축 → 개발 → 변경/배포 → 운영

장애 분리

장애 영향도

장애 발생시 영향도 범위를 줄일 수 있는가?

개발환경구축 → 개발 → 변경/배포 → 운영

2.3 마이크로 서비스 분해과정

모놀리식한 애플리케이션을 마이크로 서비스로 전환하기 위해서 아래와 같은 단계를 고려해야 한다.

1. Bounded Context 분리

- 업무 담당자의 Best Guesses 도출을 위한 Workshop
- 단독으로 출시 가능한 최소한의 업무 기능으로 분리

2. Context Map 작성

- 1번에서 분리된 Context의 기술적 관점 검토
- Transaction 관계 도출, dependency 확인
- Tight coupling 업무 도출
- DB 의존성 확인(ERD로 data 관계 확인)
- Class 의존성 확인(JDepend 등 Tool로 Reverse Engineering)
- 기술적으로 복잡한 연관관계를 갖는다고 판단되는 Context에 대해서 1-2번 과정 반복

3. Picking : 분리된 Context중에 refactoring 우선 순위 결정 (pain , gain을도출 해서 결정)

- Pain: Refactoring Cost
- Gain
  - 빈번한 스케일링
  - 잦은 배포

4. Refactoring

2.3 마이크로 서비스 분리 워크샵 (Bounded Contxt 분리)

업무 담당자의 **Best Guesses** 도출을 위한 **Workshop**

1. 현재 시스템의 요구사항 나열해보기

2. 현재 시스템의 비즈니스 목표 도출하기

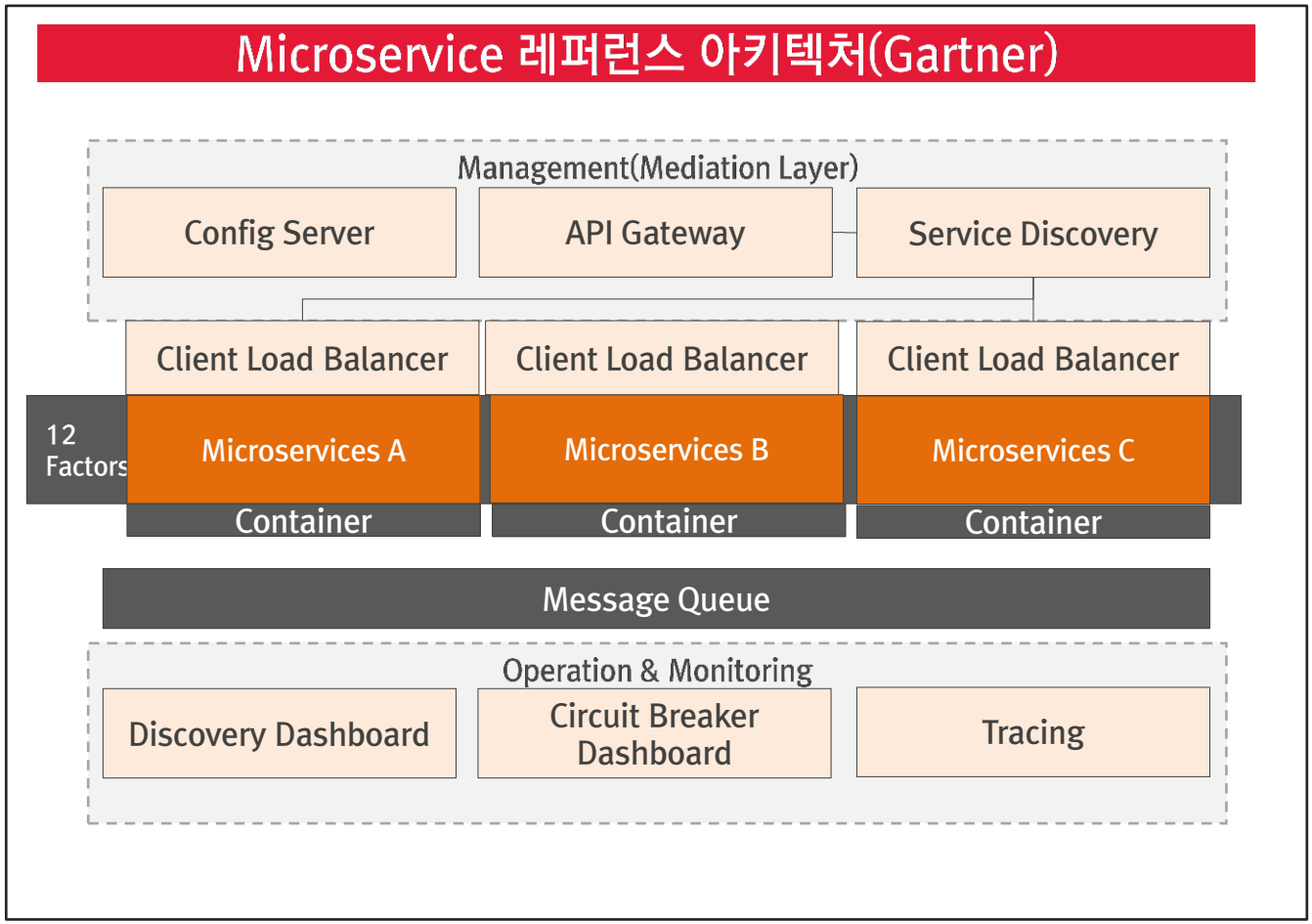
3. **Bounded Context**분리

- 비즈니스 목표를 달성 할수 있는 단위로 **Bounded Context** 묶어보기
  - 돈과 관련된 장애에 예민한 업무
  - 외부 연동 인터페이스
  - 이벤트 성 업무 (생성되었다사라지는 업무)
  - 확장 가능성
  - 배치성 업무

4. Bounded Context 결정

2.3 마이크로 서비스 Outer Architecture 설계

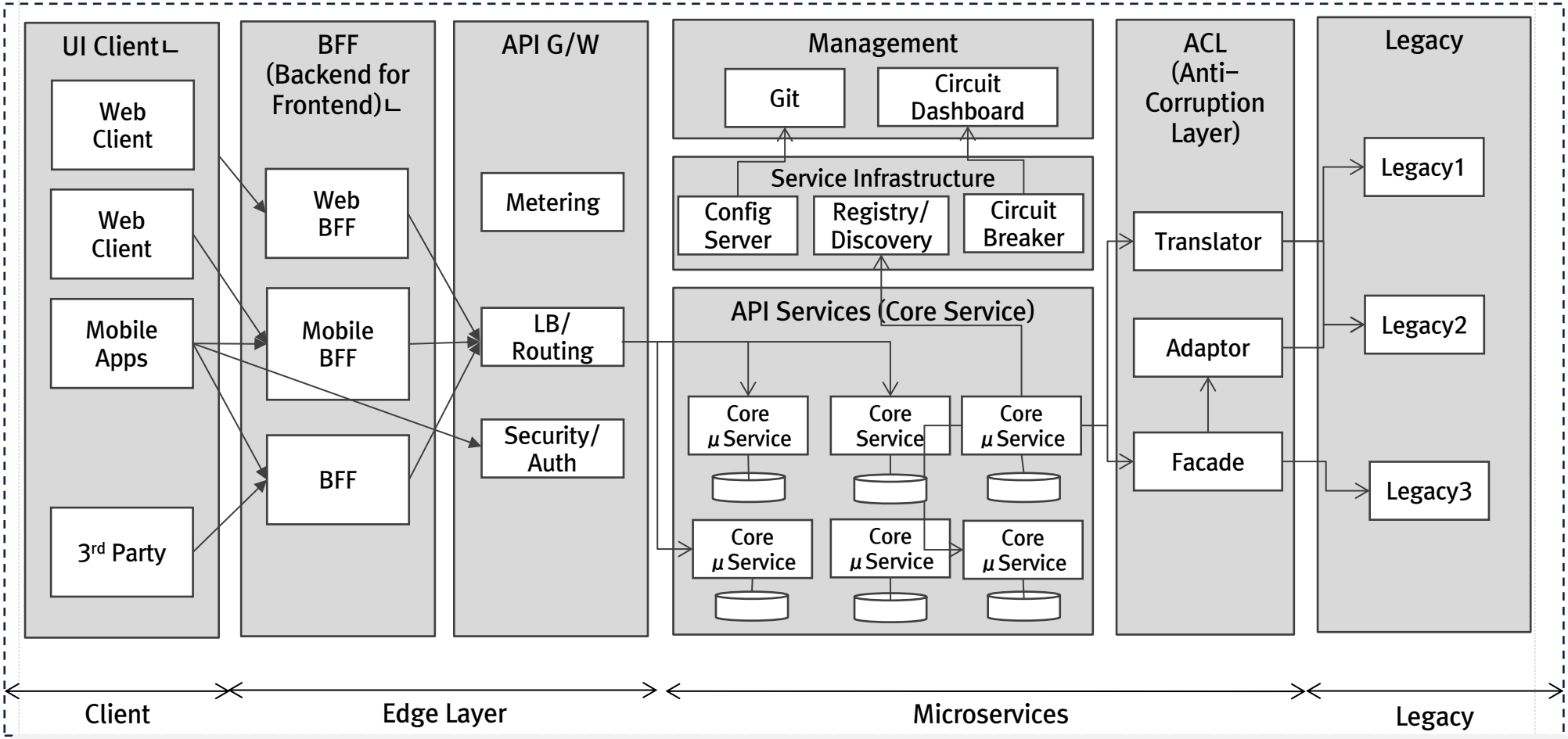
마이크로서비스의 효과적인 운영을 위한 전체 아키텍쳐(Outer Architecture) 를 수립합니다.



Source : Microservices Reference Architecture (Gartner)

[백업] 마이크로 서비스 아키텍처

일반적인 웹 서비스의 마이크로서비스 레이어 구성으로, 프로젝트의 특성을 고려해서
레이어를 구성하여야 한다



- ✓ UI를 위한 서비스(BFF)와 API를 위한 서비스(Core Service)를 분리
- ✓ Core Service간 호출은 API G/W를 거치지 않고, Discovery를 통한 내부 통신으로 연계

감사합니다