CLOUDZ LABS

Cloud Application 기본설계 가이드

SK 주식회사

Table of Contents

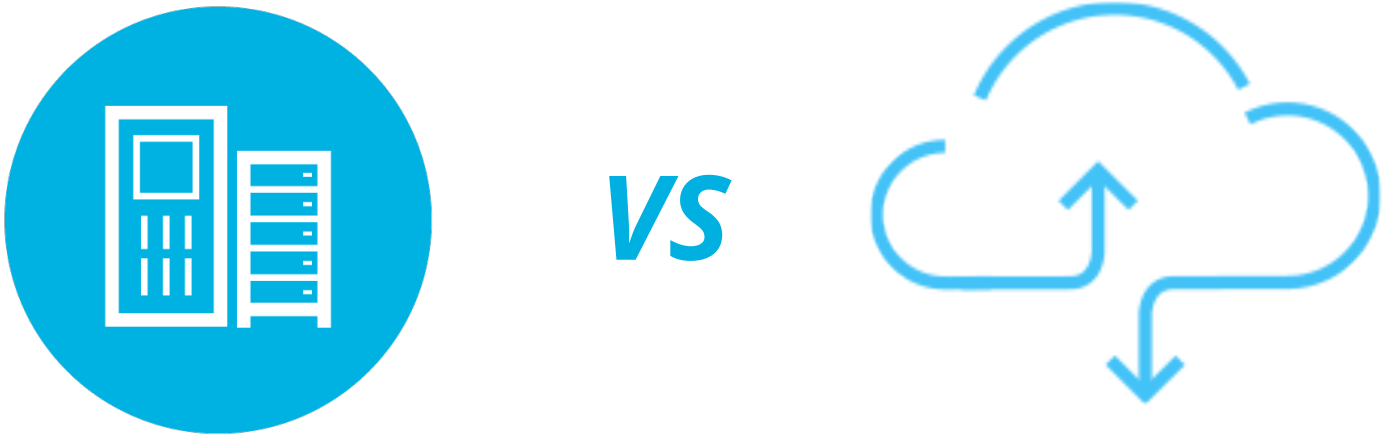
PART 1. 클라우드 플랫폼 특징

01 **PaaS** 특징

02 **CaaS** 특징

1.1 PaaS 특징

On-premise vs Cloud



1.1 PaaS 특징

On-premise 만 있다면 고민해야 될 것이 너무 많다**!**

OS 선택? 버전?

서버 구입 비용은?

Network 구성은?

WAS는?

Framework은?

스토리지는?

메모리는?

인스턴스 갯수는?

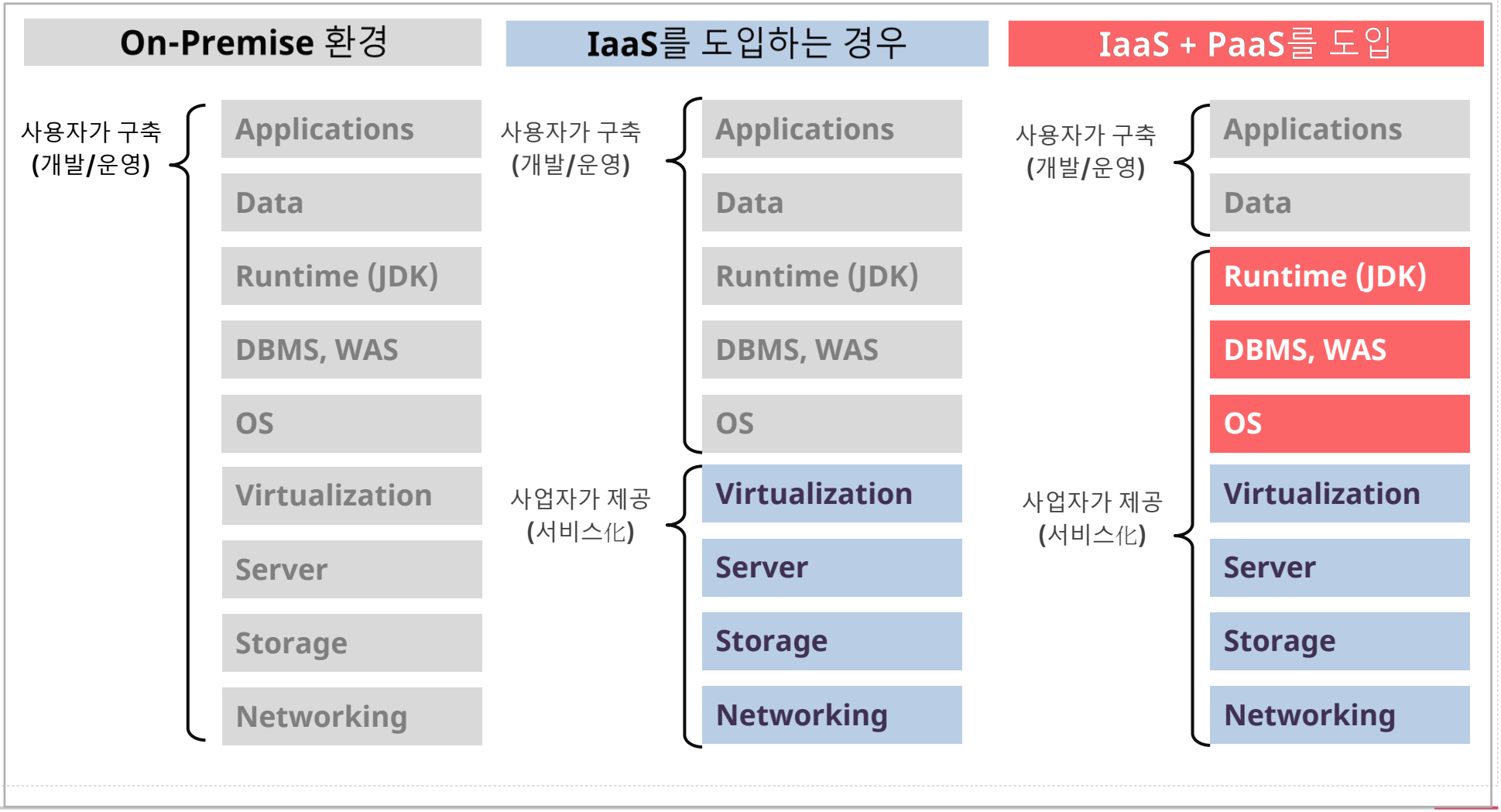
DB 연동은?

보안?

운영은?

1.1 PaaS 특징

IaaS는 IT Infra를 제공하고, PaaS는 시스템 S/W 제공(OS, DB, WAS, JDK)



1.1.1 개발환경 구성

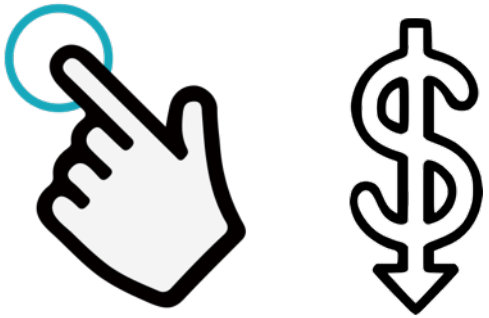
개발환경 구성의 신속성으로 개발에만 집중할 수 있음

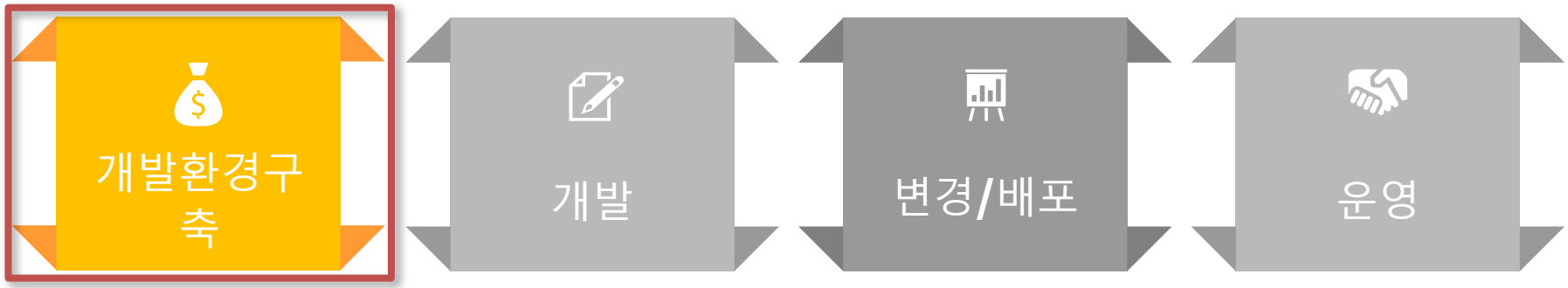
개발 환경 제공의 신속성

개발 환경 구축은 손쉽게 할 수 있어야 함

초기 구축 비용을 절감 해주어야 함

불필요한 고민을 줄여야함





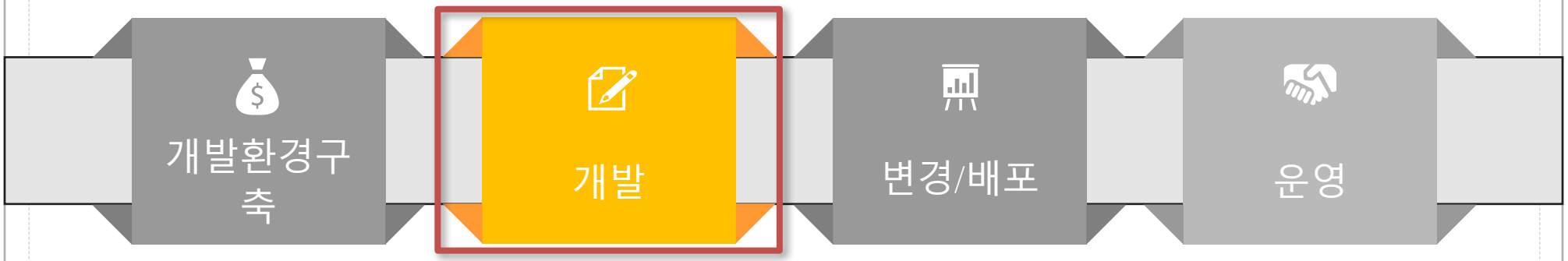
1.1.2 서비스 마켓플레이스

필요한 서비스들을 바로 사용하기 위해서는 서비스 마켓 플레이스 개념이 필요

개발 생산성

개발에 필요한 서비스들을 바로

사용할 수 있어야 함



1.1.3 Auto Scaling

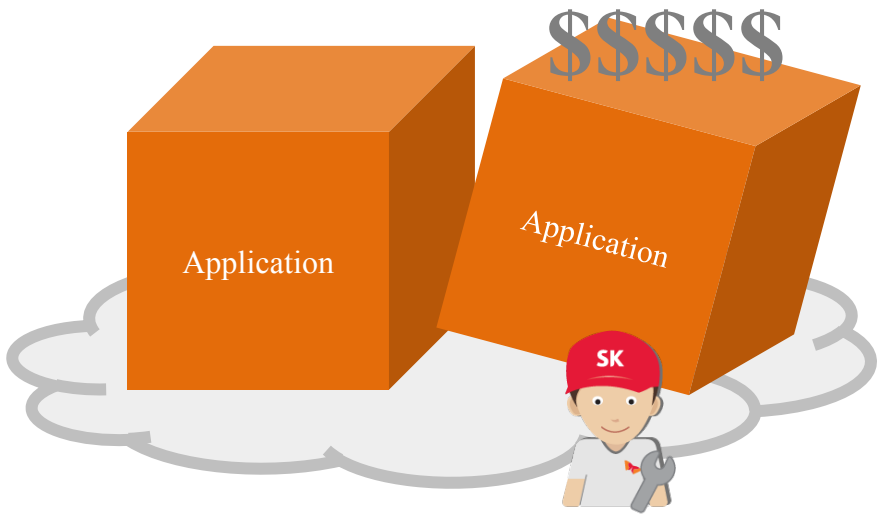
전체 VM을 스케일링하기보다는 작은 단위의 스케일링이 요구됨

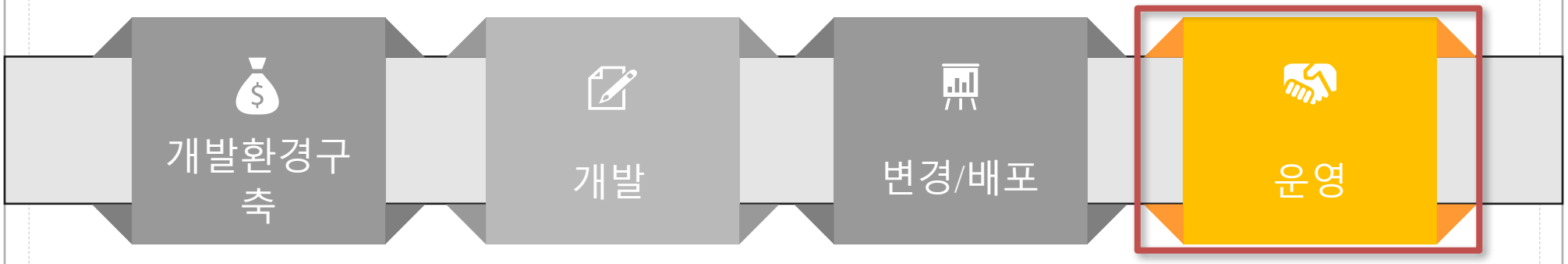
용이한 스케일링과 효율적인 이중화

Appl.이 원하는 자원이 즉시 제공 가능

자원 추가 시 서비스 영향도가 감소

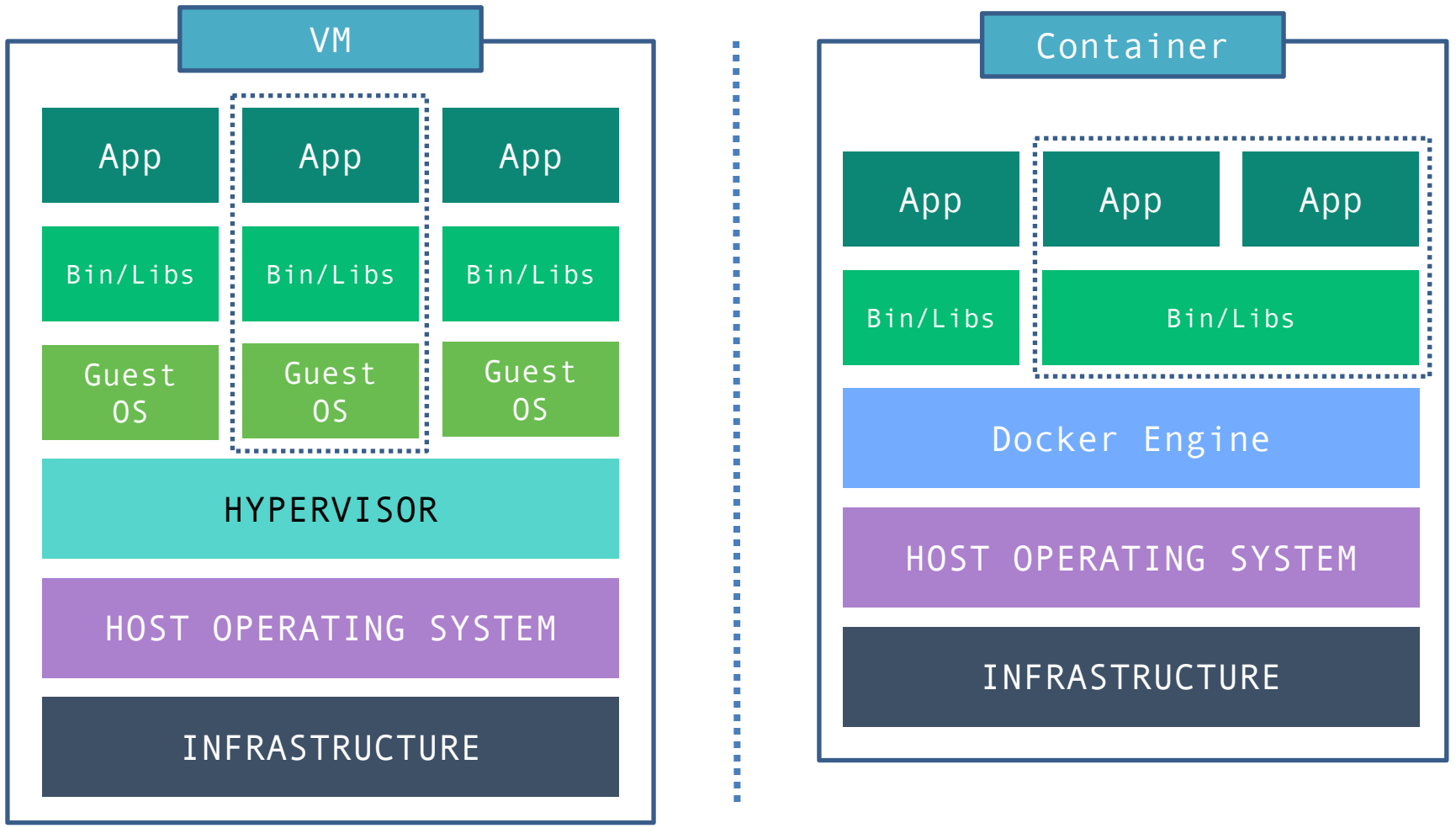
필요한 서비스만 이중화/삼중화





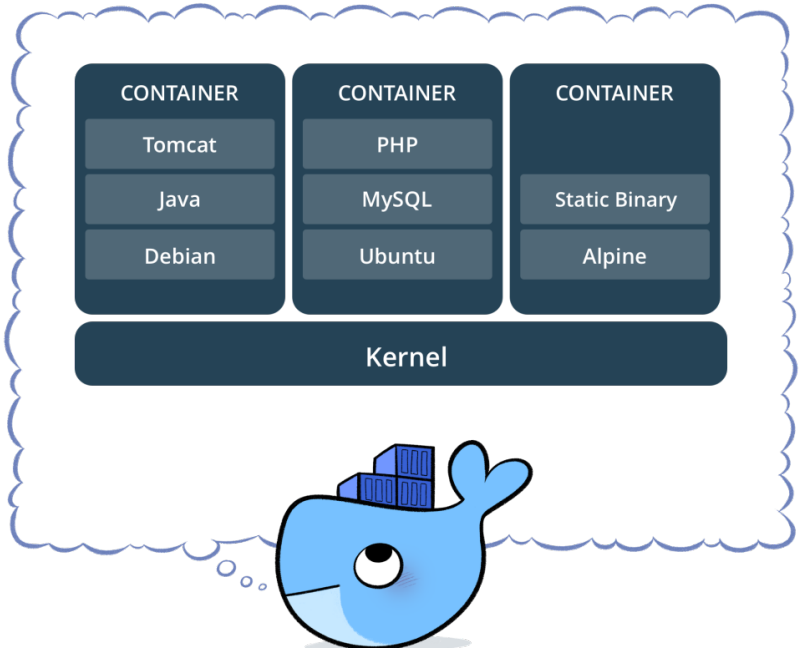
1.2 CaaS 특징

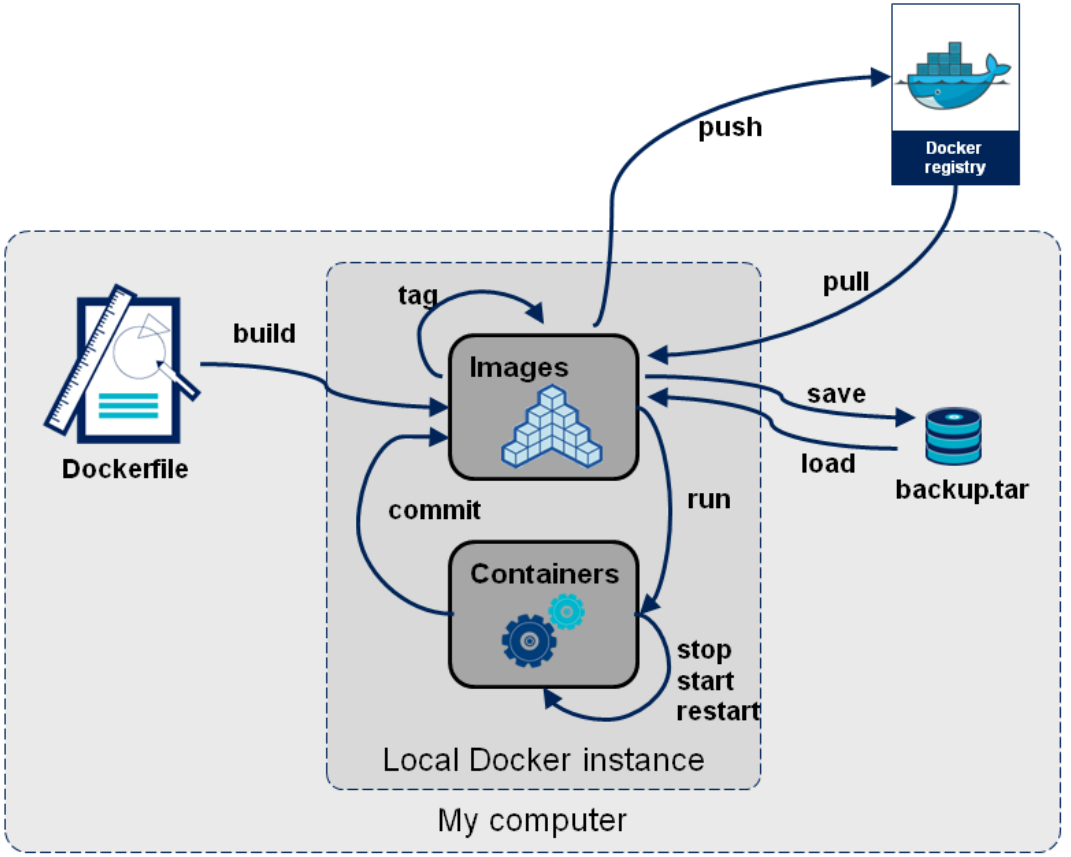
VM vs Container



1.2 CaaS 특징

Docker 아키텍처





1.2 CaaS 특징

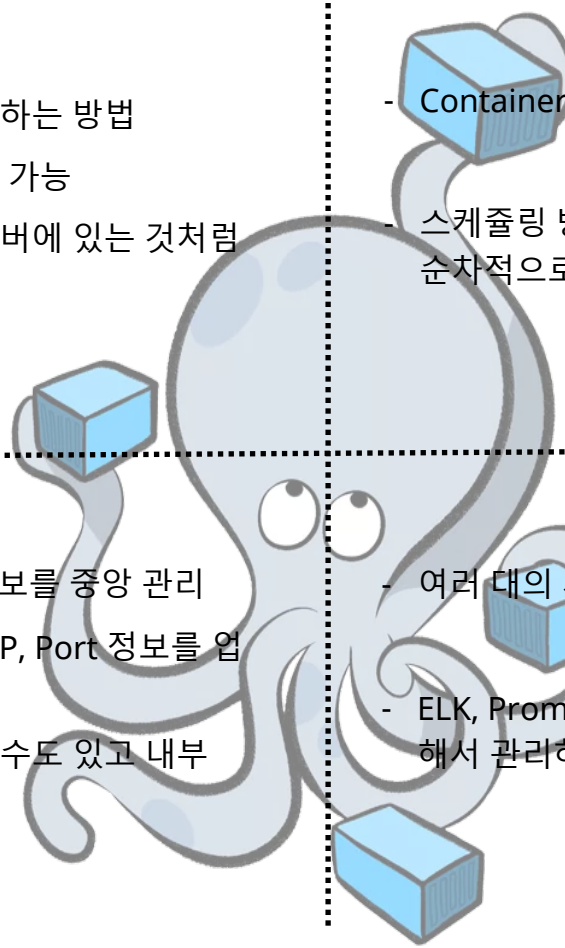
Container Orchestration

Clustering

- 여러 개의 서버를 하나의 서버처럼 사용하는 방법
- 클러스터에 새로운 서버를 추가 및 제거 가능
- 가상 네트워크를 이용하여 마치 같은 서버에 있는 것처럼 쉽게 통신할 수 있음

Scheduling

- Container를 적절한 서버에 배포해주는 작업
- 스케쥴링 방법은 툴에 따라서 지원하는 전략(랜덤배포, 순차적으로 배포 등등...)을 선택할 수 있음



Service Discovery

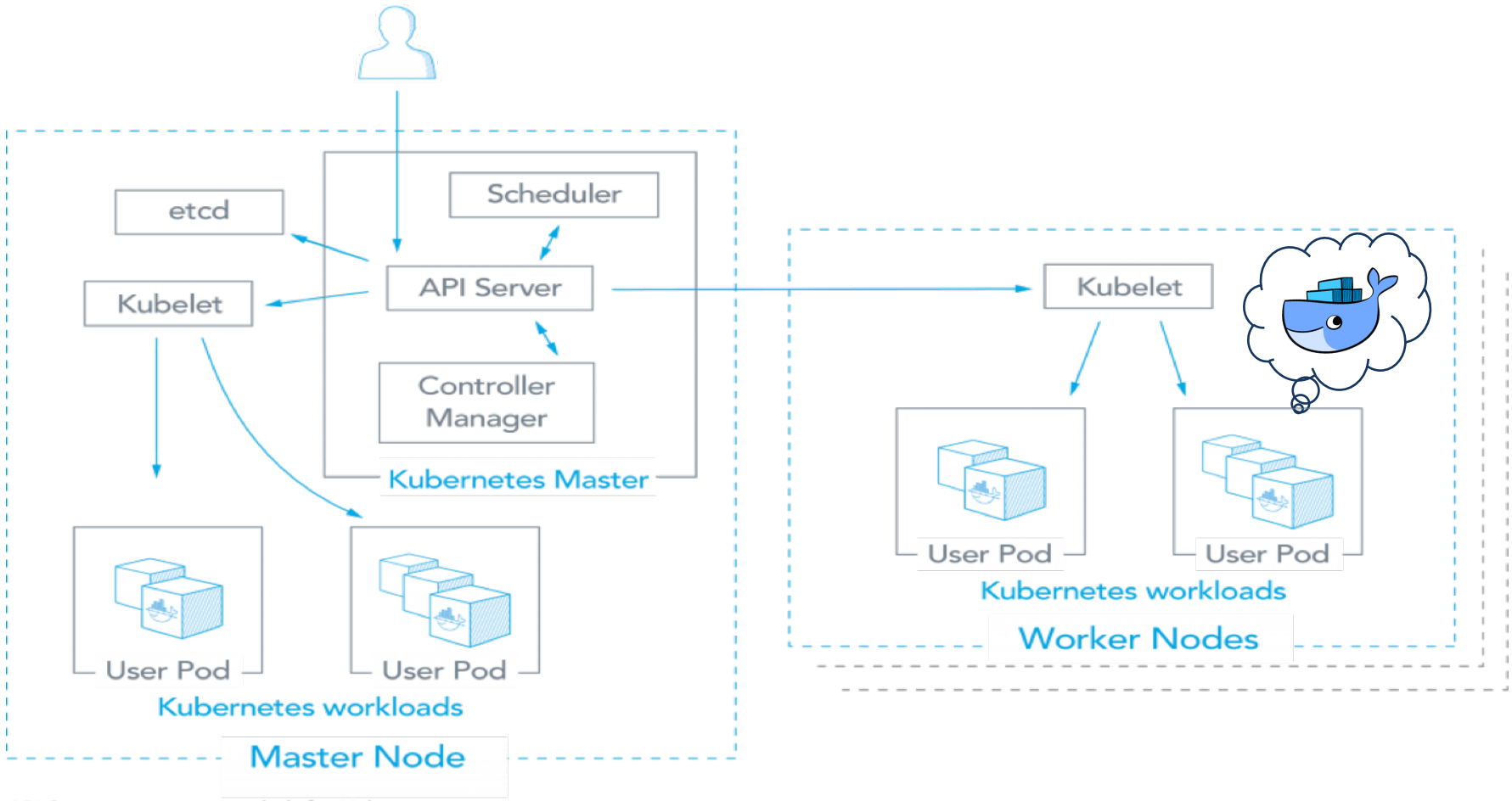
- 여러 Node에 생성되는 서비스들의 정보를 중앙 관리
- 서비스가 생성되거나 변경될 때 해당 IP, Port 정보를 업데이트
- Key-Value 스토리지에 정보를 저장할 수도 있고 내부 DNS 서버를 활용할 수 있음

Logging, Monitoring

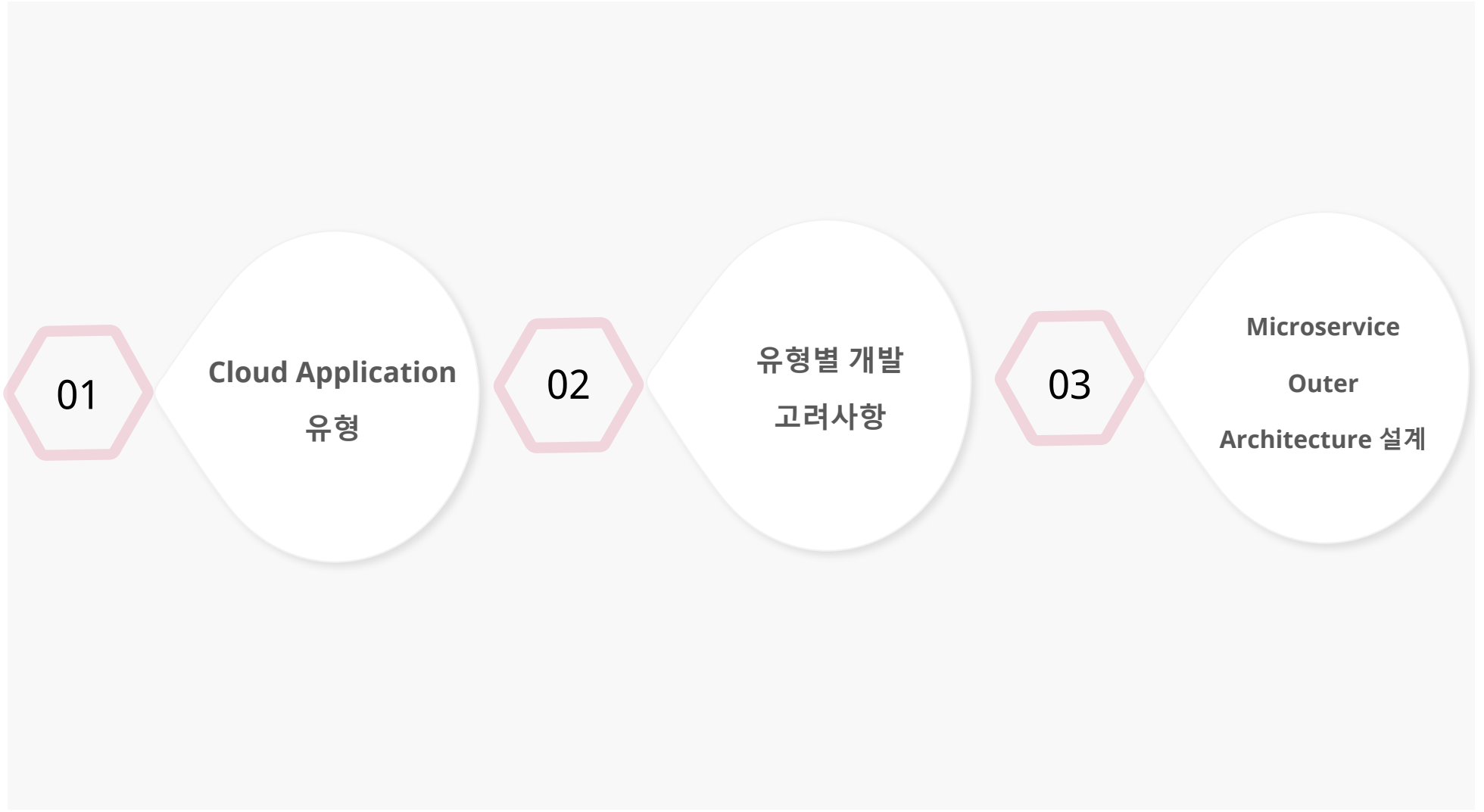
- 여러 대의 서버의 로그를 모아서 한 곳에서 관리함
- ELK, Prometheus등과 같은 모니터링 툴을 별도로 설치해서 관리하기도 함

1.2 CaaS 특징

Kubernetes 아키텍처

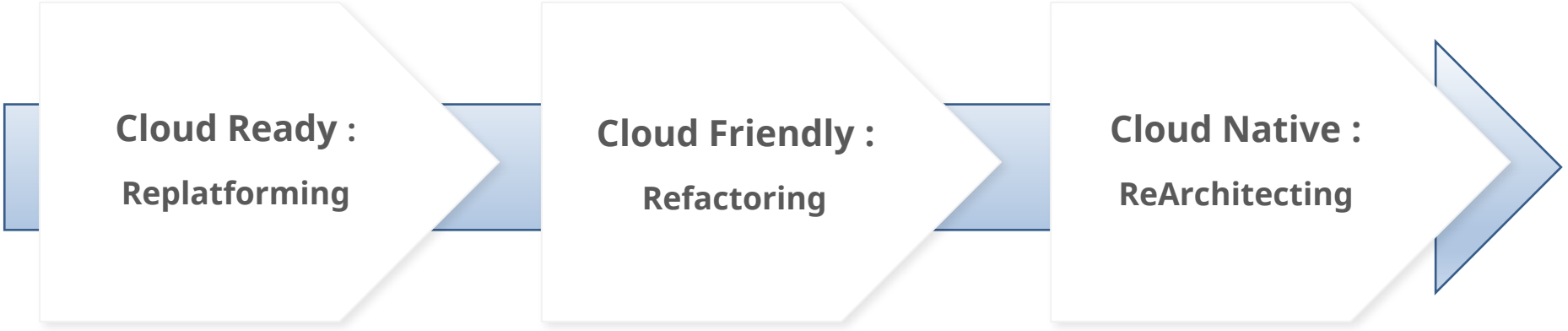


PART2 클라우드 애플리케이션 유형별 개발 가이드



2.1 Cloud Application 유형

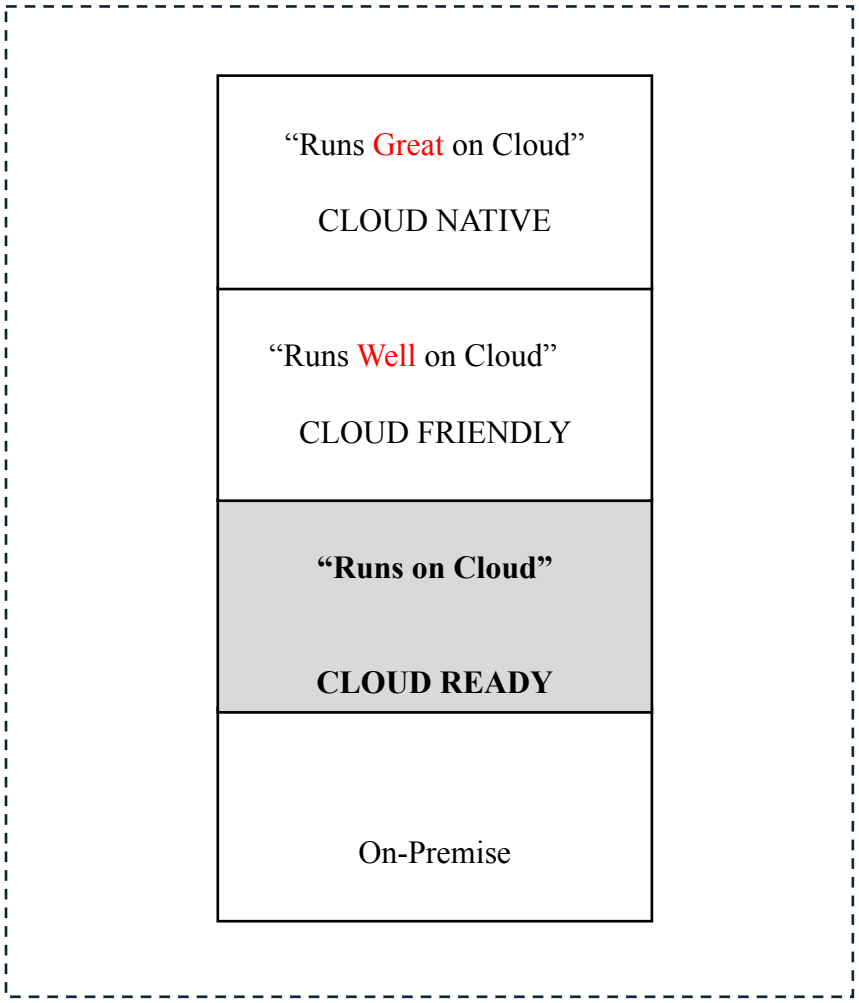
Cloud Application 전환을 위한 Step



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 보기 위한 전환	변화나 수정에 유연한 구조의 애플리케이션으로 전환

2.2.1. Cloud Ready : Replatforming

Cloud Ready 수준의 애플리케이션으로 전환시 고려 사항을 가이드 한다.



Goal

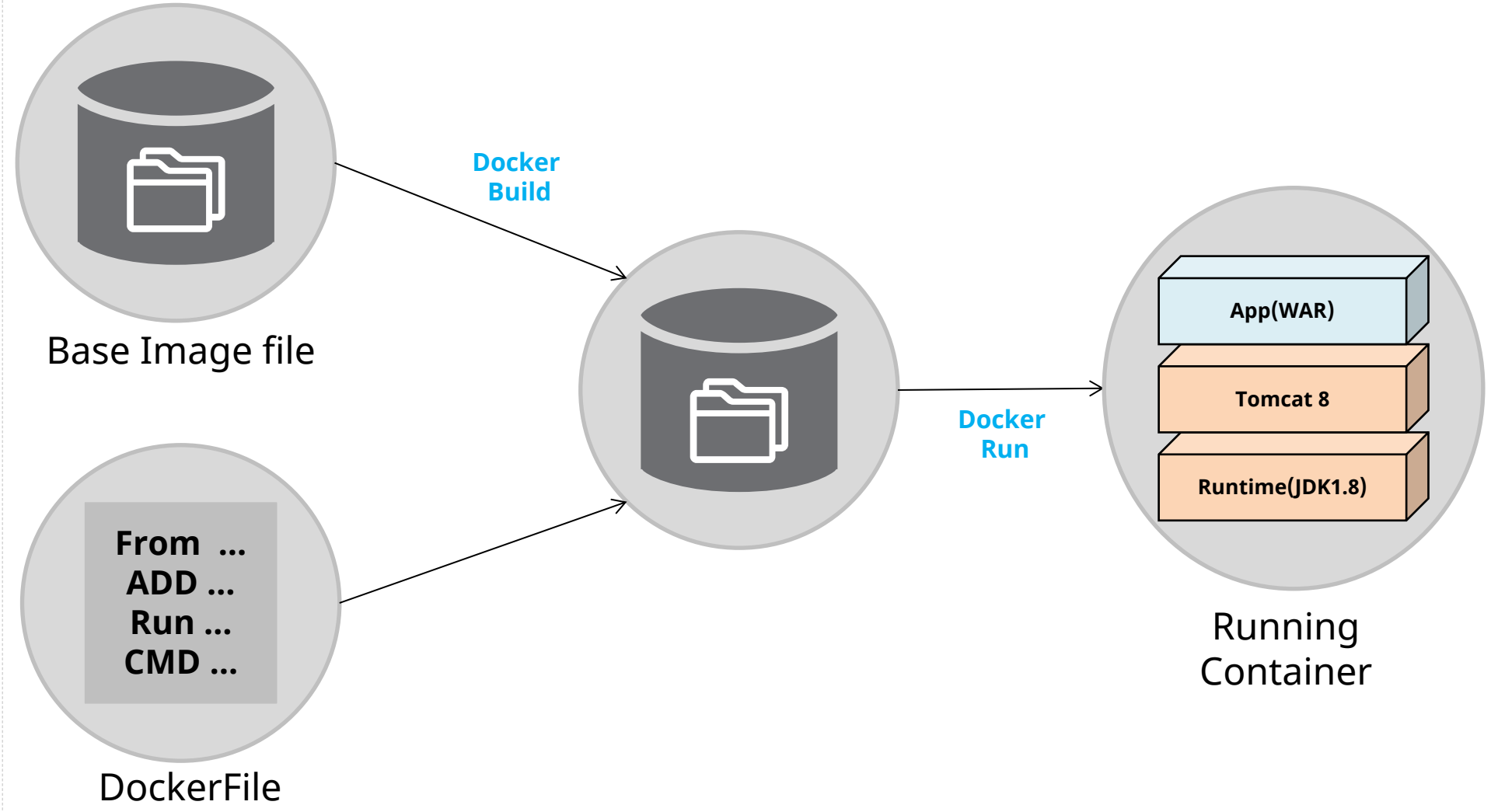
- **Cloud에 애플리케이션 정상적으로 배포**

목차

- **Replatforming** 고려 사항
- 플랫폼 특성을 고려한 애플리케이션 마이그레이션

2.2.1. Cloud Ready : Replatforming

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



2.2.1. Cloud Ready : Replatforming

파일을 저장하는 로직이 있을 경우, 애플리케이션 외부 서비스를 활용 하도록 수정해야 한다.
(**Cloud**에서 애플리케이션 재 배포시 컨테이너 재 생성 으로 저장된 파일 초기화 됨)

파일

로컬 파일 시스템 사용 여부 확인

- 임시 파일이 아닌 영구적 파일 시스템을 사용하는 경우 Object Storage같은 외부 서비스를 사용
- 외부서비스 사용을 위해 파일 사용 로직에 대한 소스 변경 필요

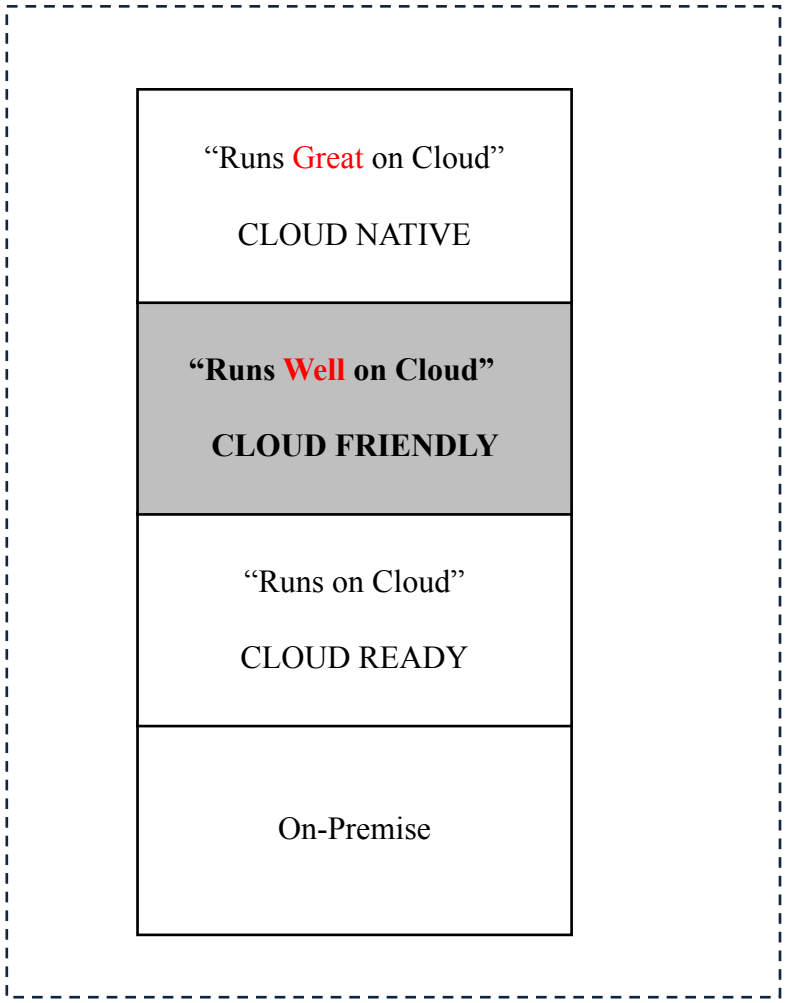
로그

기존 시스템의 로그 파일 생성 여부 확인

- 파일 로그를 사용하는 경우 console로 로그를 출력하도록 변경
- ELK, EFK와 같은 외부 서비스를 통해 로그를 수집, 저장함

2.2.2. Cloud Friendly : Refactoring

Cloud Friendly 수준의 애플리케이션으로 전환 시 고려사항을 가이드 한다.



Goal

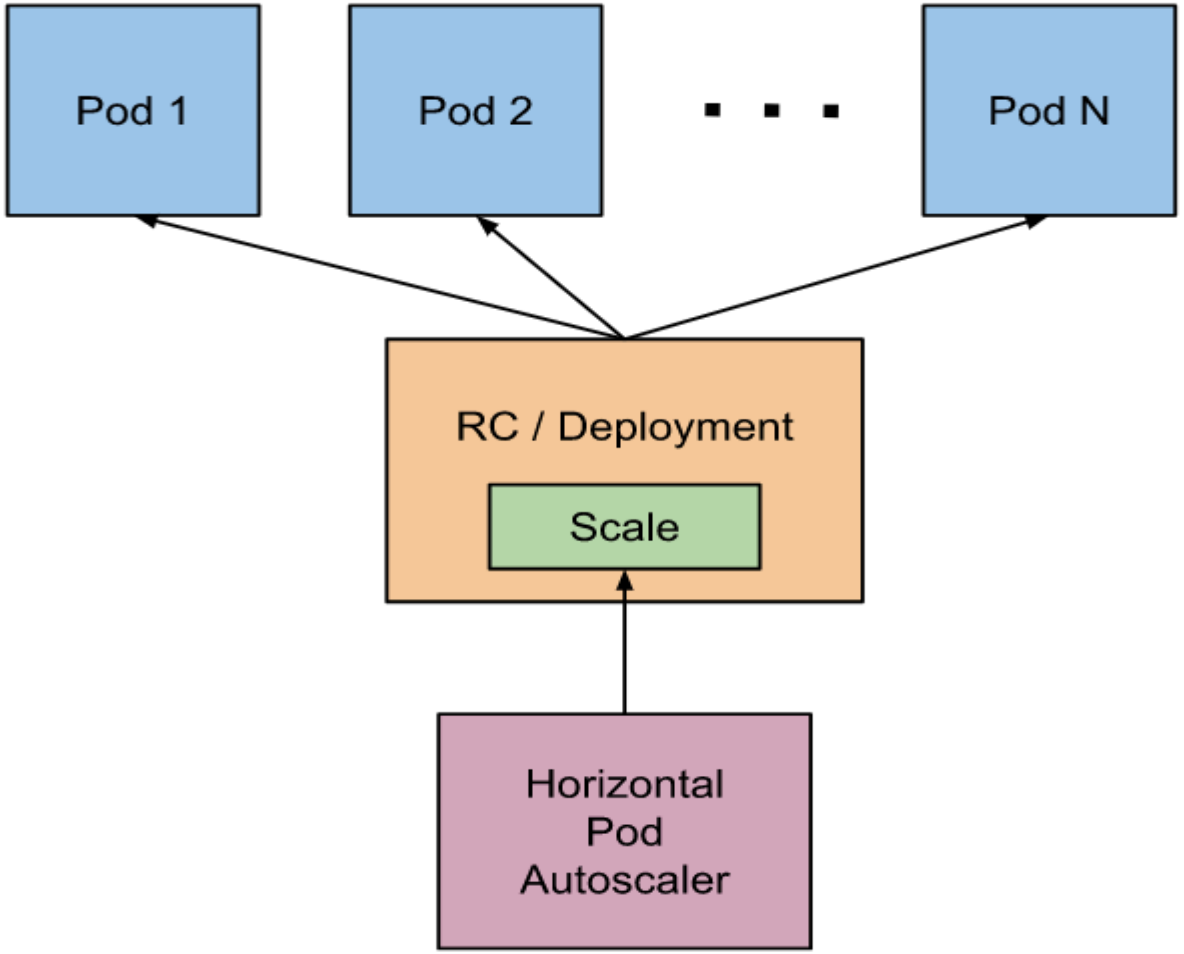
클라우드의 장점을 활용 할 수 있는 애플리케이션 배포

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

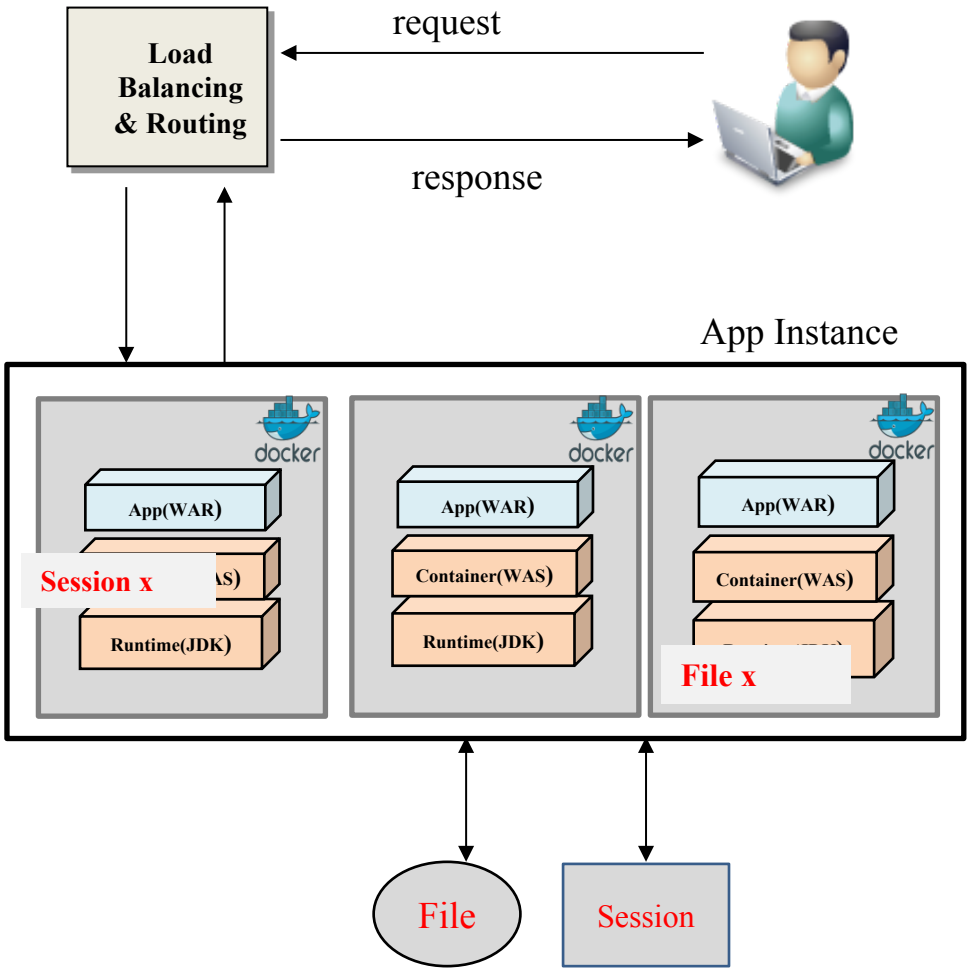
2.2.2. Cloud Friendly : Refactoring

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



2.2.2. Cloud Friendly : Refactoring

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



- ✓ **애플리케이션의 Scalablilty 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 하나 이상의 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

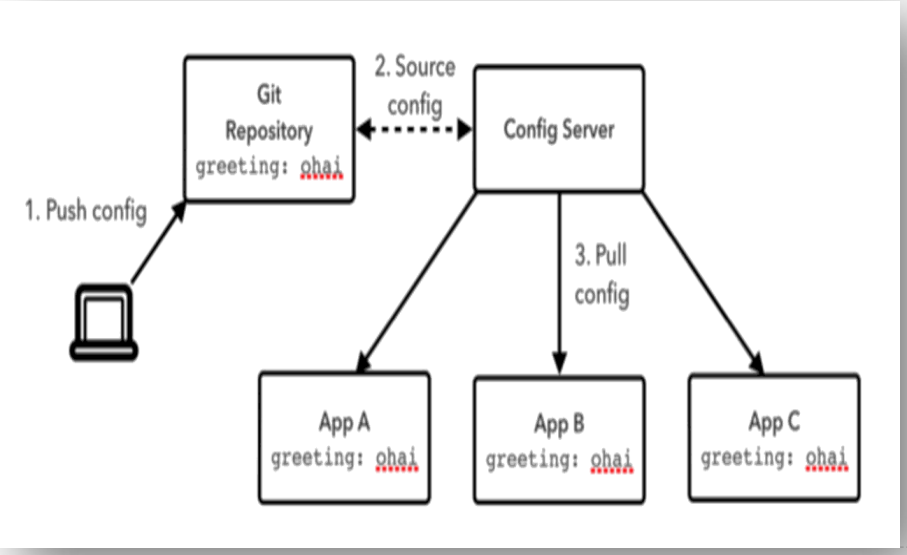
2.2.2. Cloud Friendly : Refactoring

12factors 요건 중 하나로 애플리케이션은 하나의 코드 베이스를 유지하고 **배포환경에 따라 달라질 수 있는 내용은 설정 파일**로 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 Service를 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 Config서버 구현(Spring cloud config 서버 활용가능)

2.2.2. Cloud Friendly : Refactoring

12factors 요건 중 하나로 애플리케이션에서 네트워크를 통해 이용하는 모든 서비스들을 코드 변경 없이 자유롭게 연결 및 분리 할 수 있도록 URL과 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)

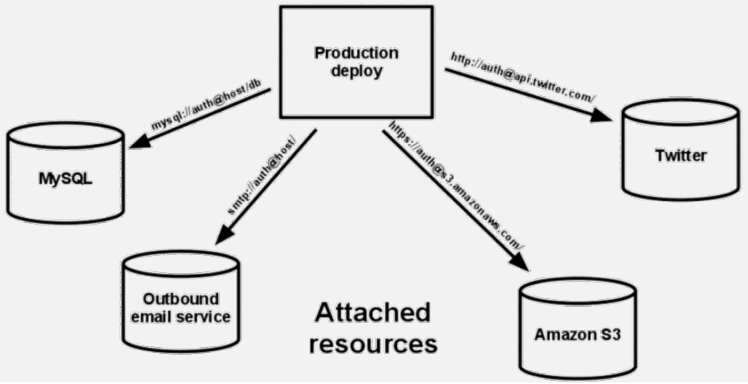
   1-1. PaaS 마켓플레이스에 서비스 존재 :cf create-service [service] [plan] [**서비스명**]

   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



2.2.2. Cloud Friendly : Refactoring

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

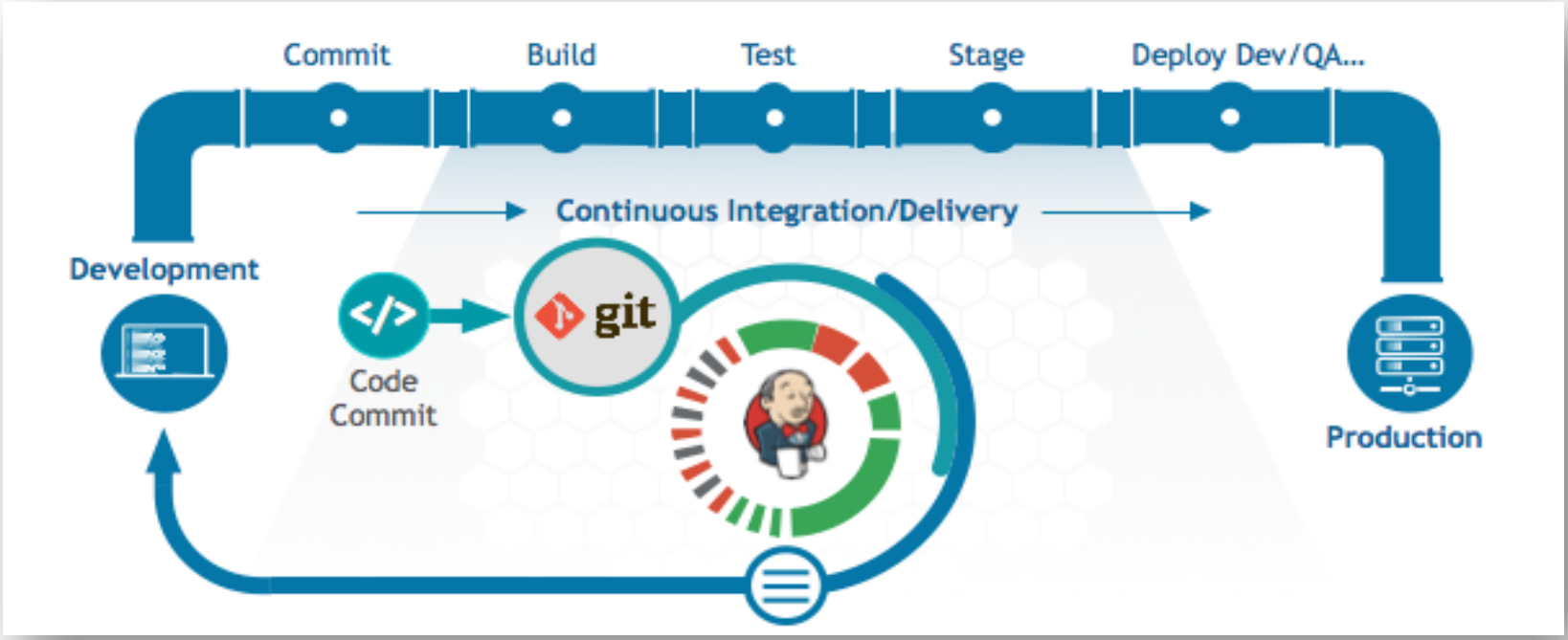
Maven, Gradle 같은 의존성 관리 도구 활용

```
<!-- 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>

2.2.2. Cloud Friendly : Refactoring

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.

2.2.2. Cloud Friendly : Refactoring

배포 파이프라인을 구축 할 때 아래와 같은 항목이 고려되어야 한다.

항목	설명	활용도구
CI Server	빌드 프로세스를 관리 하는 서버	**Jenkins** , Hudson, Travis CI
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
배포 파이프라인 Dash board	배포 파이프라인의 상태를 시각화 할 수 있는 대시보드	Jenkins Delivery Pipeline Plugin, Spinnaker

2.2.3. Cloud Native : Reachitecturing

Cloud Native 수준의 애플리케이션으로 전환 시 고려사항을 가이드한다.

Goal

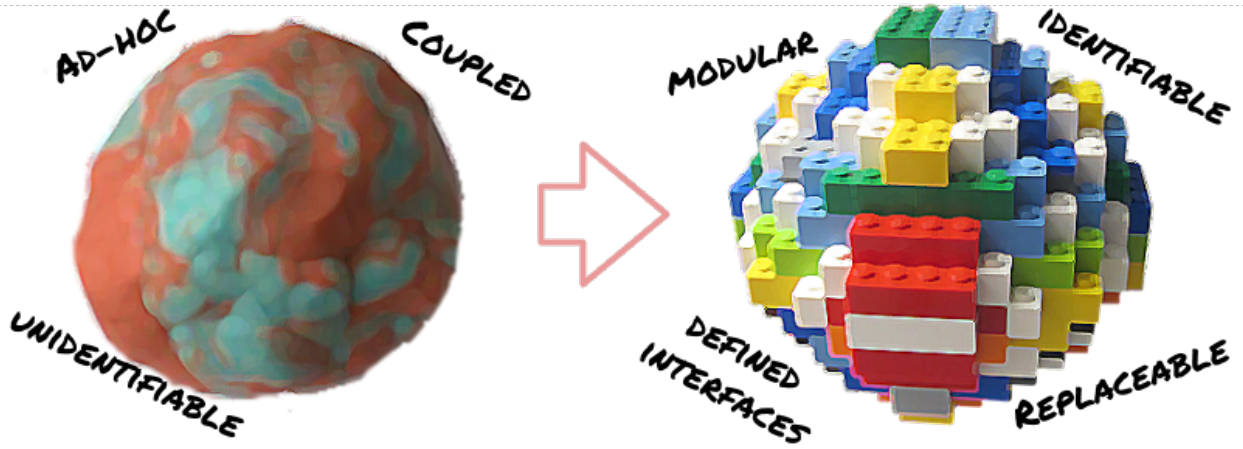
마이크로 서비스로 애플리케이션 배포

목차

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

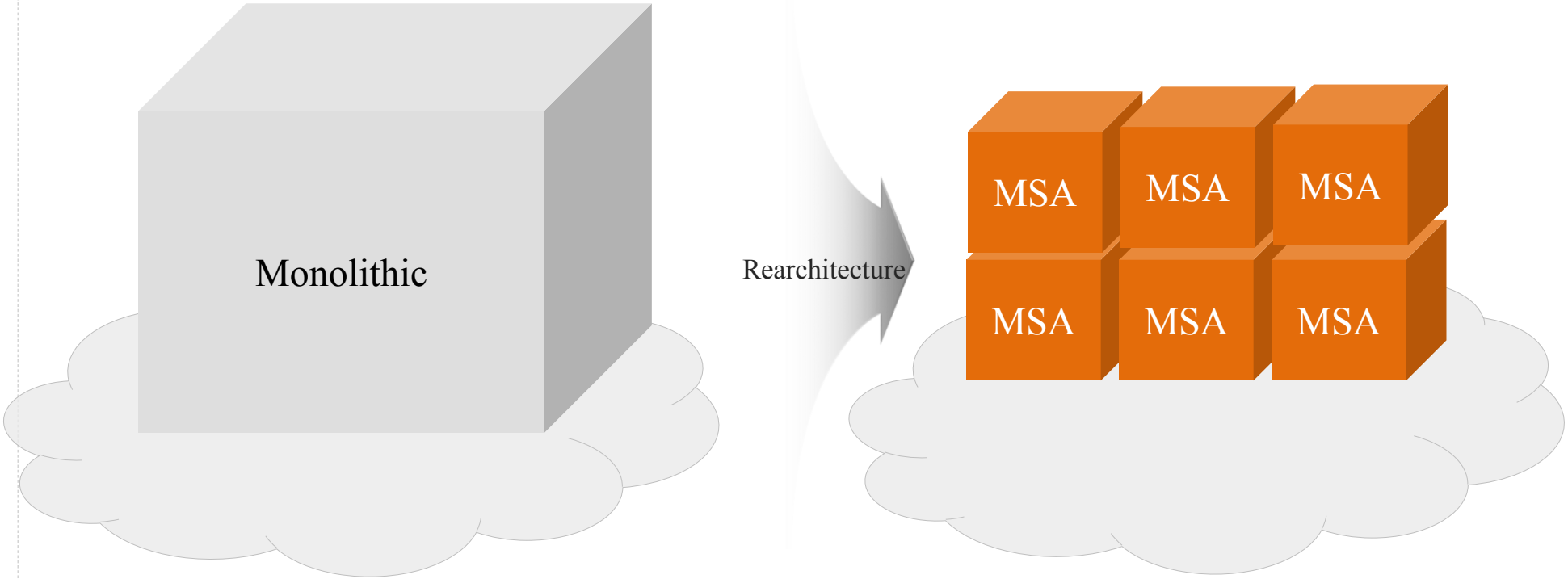
2.2.3. Cloud Native : Reachitecturing

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



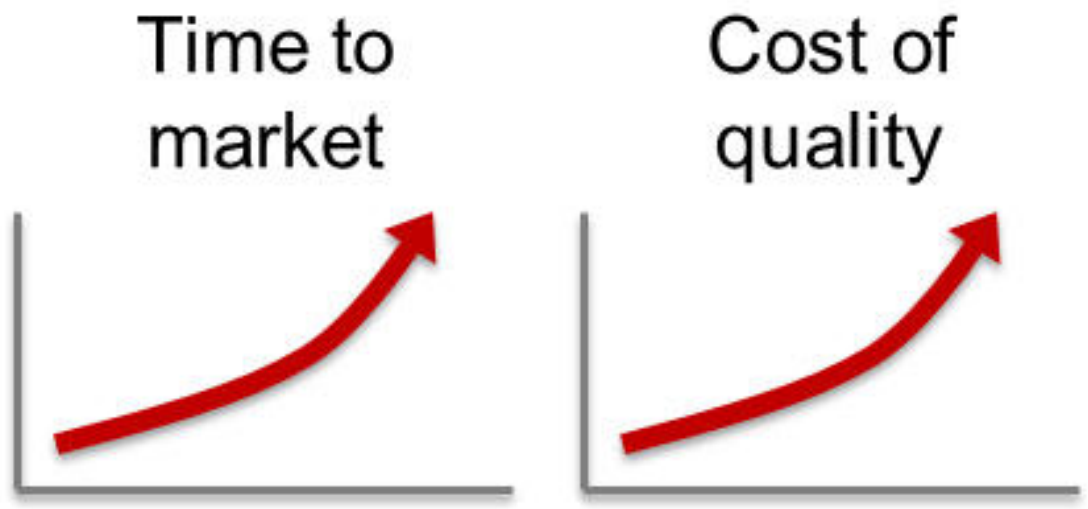
2.2.3. Cloud Native : Reachitecturing

기존의 모놀리식 애플리케이션을 작은 단위로 쪼개서 운영하는 방식입니다.



2.2.3. Cloud Native : Reachitecturing

마이크로서비스 아키텍처로 달성 해야 하는 비즈니스 요구사항이 명확히 정의되어야 함!



“작고, 독립적인”

2.2.3. Cloud Native : Reachitecturing

마이크로 서비스 적용 대상 애플리케이션

어떤 애플리케이션이 마이크로서비스로 전환되어야 할까?

Pain Point 도출해 보기

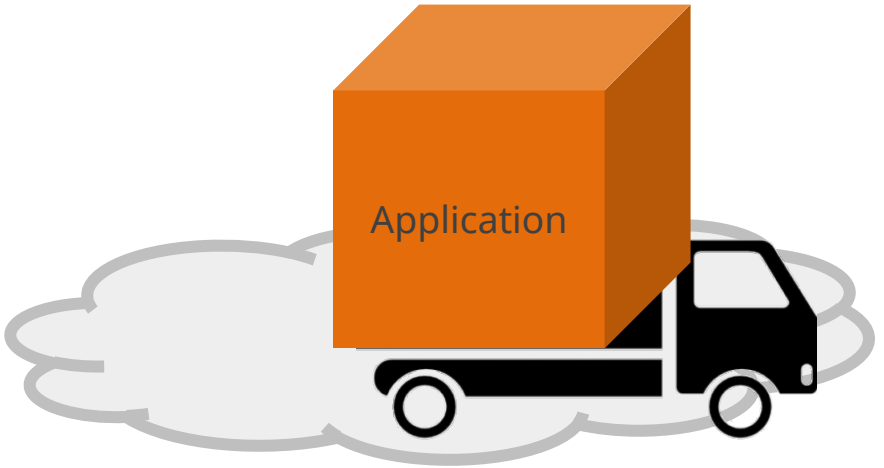
2.2.3. Cloud Native : Reachitecturing

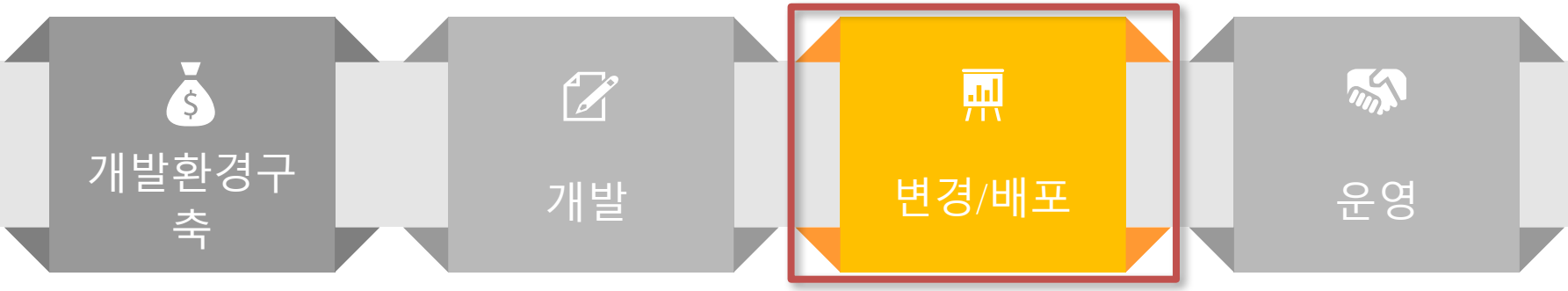
모놀리식 애플리케이션은 변경 주기가 상대적으로 큼 .

적시 배포(서비스 출시)

서비스 출시시기가 더 빨라질 수 없나?

변경 영향도를 최소화 할 수는 없나?





2.2.3. Cloud Native : Reachitecturing

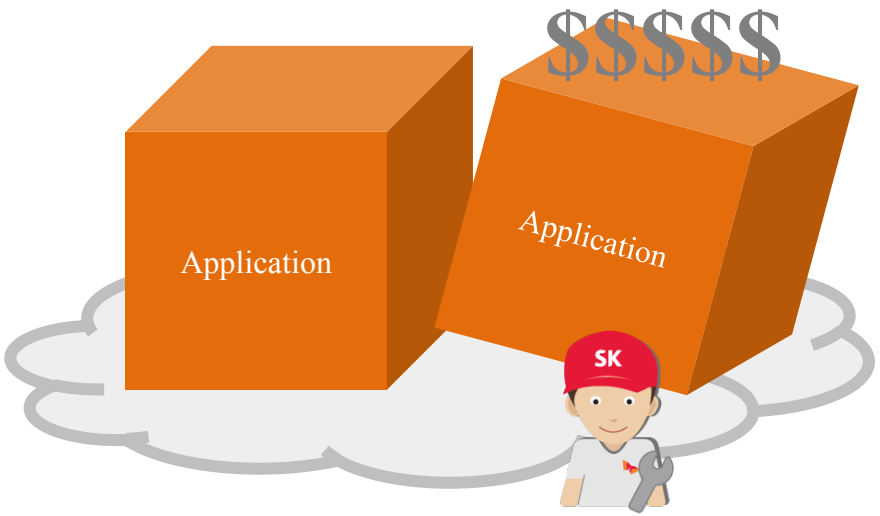
전체 VM을 스케일링하기보다는 작은 단위의 스케일링이 요구됩니다.

용이한 스케일링과 효율적인 이중화

Appl.이 원하는 자원이 즉시 제공 가능한가?

자원 추가 시 서비스 영향도는 없는가?

필요한 서비스만 이중화/삼중화 할 수 있는가?



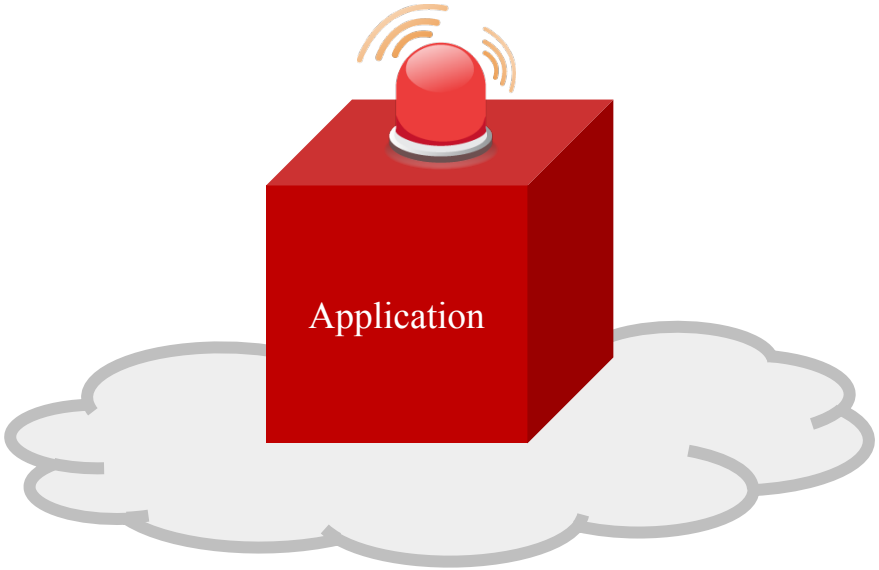
개발환경구축 → 개발 → 변경/배포 → 운영

2.2.3. Cloud Native : Reachitecturing

장애 분리

장애 영향도

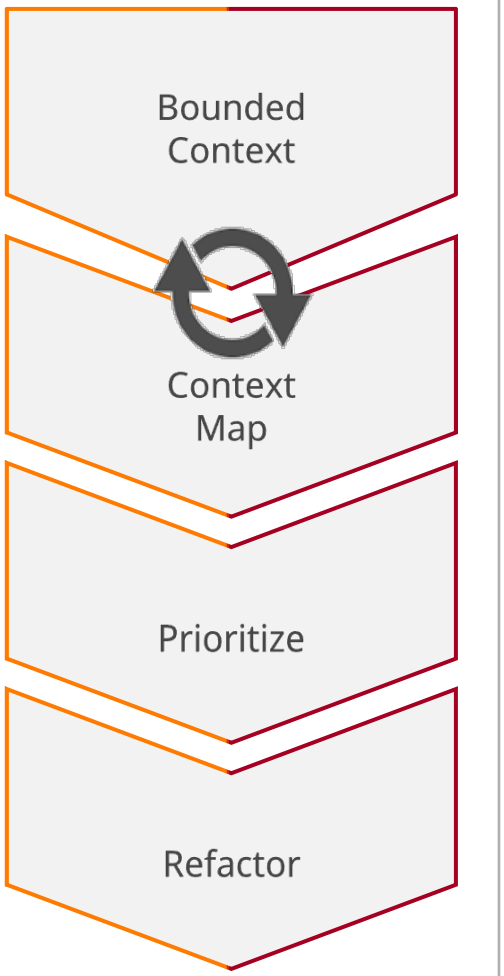
장애 발생시 영향도 범위를 줄일 수 있는가?



개발환경구축 → 개발 → 변경/배포 → 운영

2.2.3. Cloud Native : Reachitecturing

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, gain을 도출 해서 결정)

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

4. Refactoring

2.2.3. Cloud Native : Reachitecturing

업무 담당자의 Best Guesses 도출을 위한 Workshop

1. 현재 시스템의 요구사항 나열해보기

2. 현재 시스템의 비즈니스 목표 도출하기

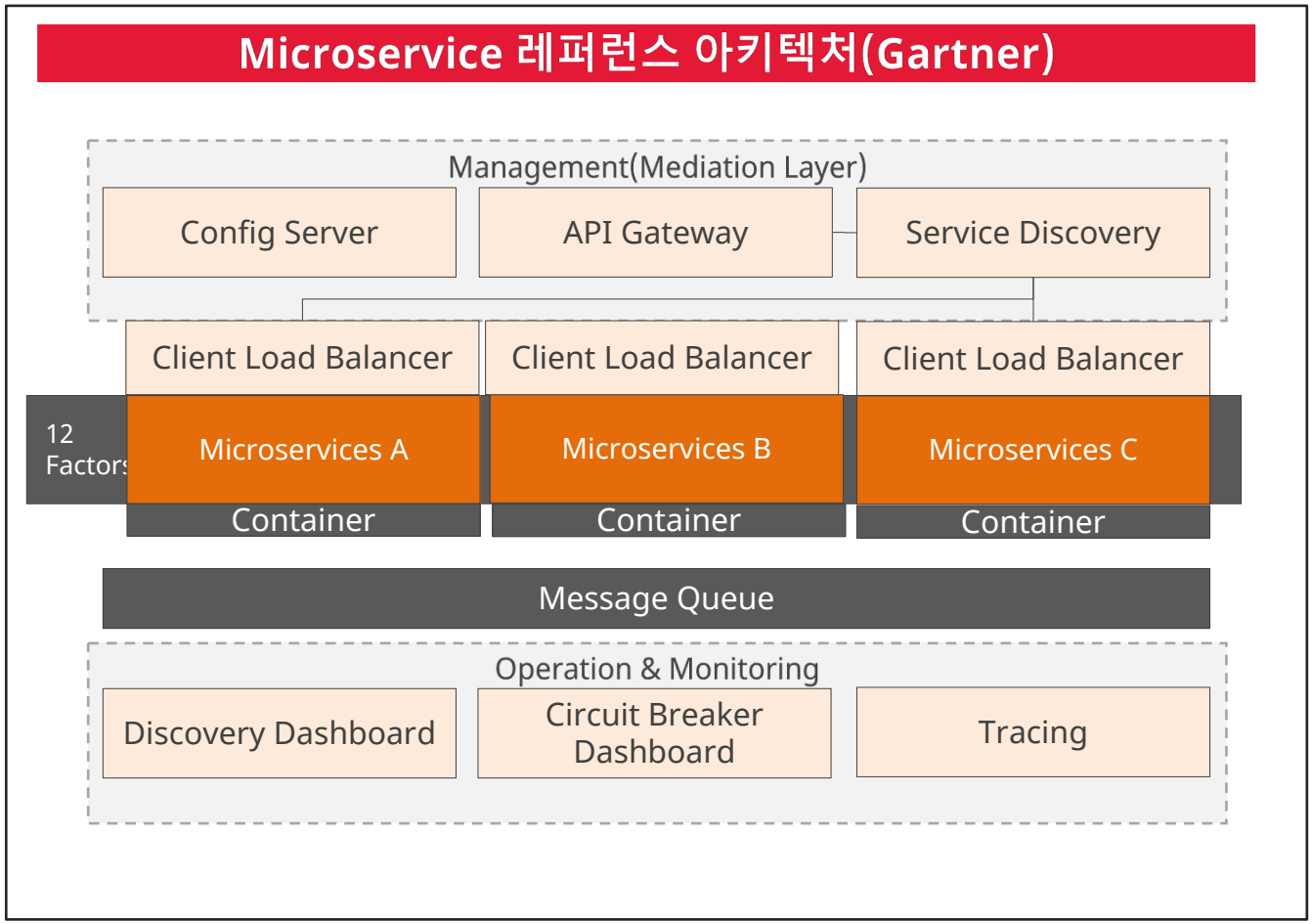
3. Bounded Context분리

- 비즈니스 목표를 달성 할 수 있는 단위로 Bounded Context 묶어보기
  - 돈과 관련된 장애에 민감한 업무
  - 외부 연동 인터페이스
  - 이벤트 성 업무 (특정기간 동안 존재)
  - 확장 가능성
  - 배치성 업무

4. Bounded Context 결정

2.2.4. 마이크로 서비스 Outer Architecture 설계

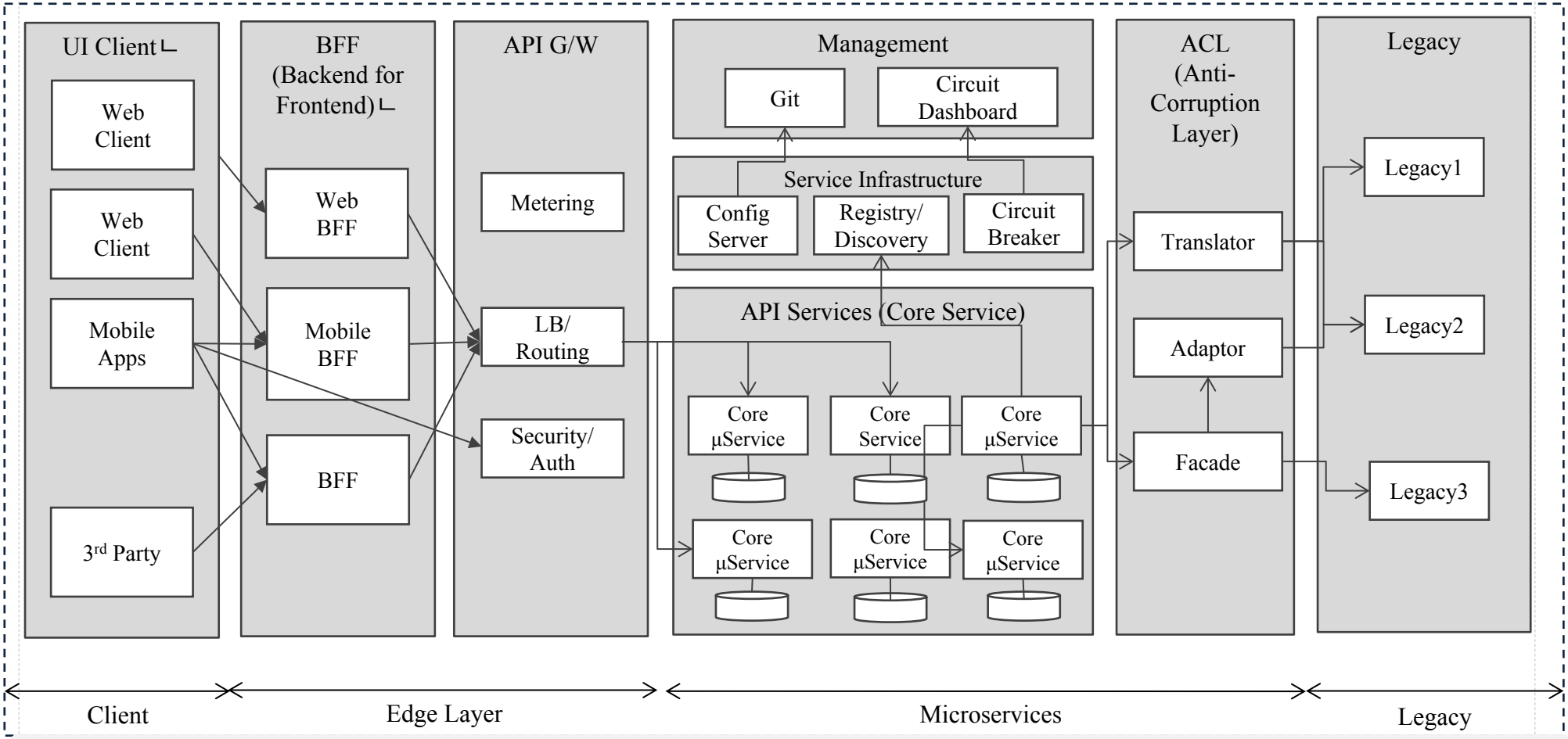
마이크로서비스의 효과적인 운영을 위한 전체 아키텍쳐(Outer Architecture) 를 수립합니다.



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