Module 5. CI/CD 설계

목차

5.1 배포 패턴

5.2 브랜치 전략

5.3 배포 기법

5.4 pipeline

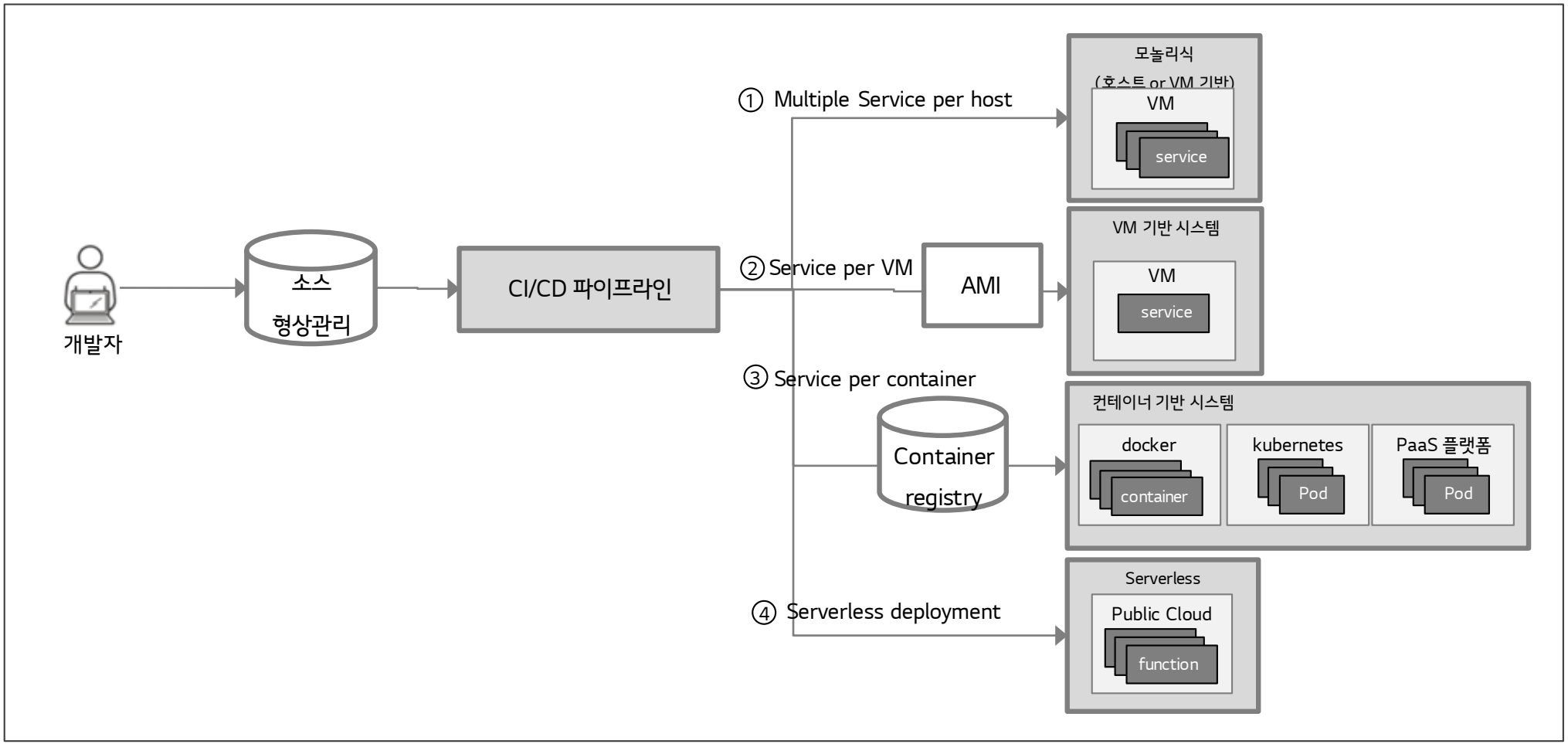
5.5 docker file

5.6 yaml file

Summary

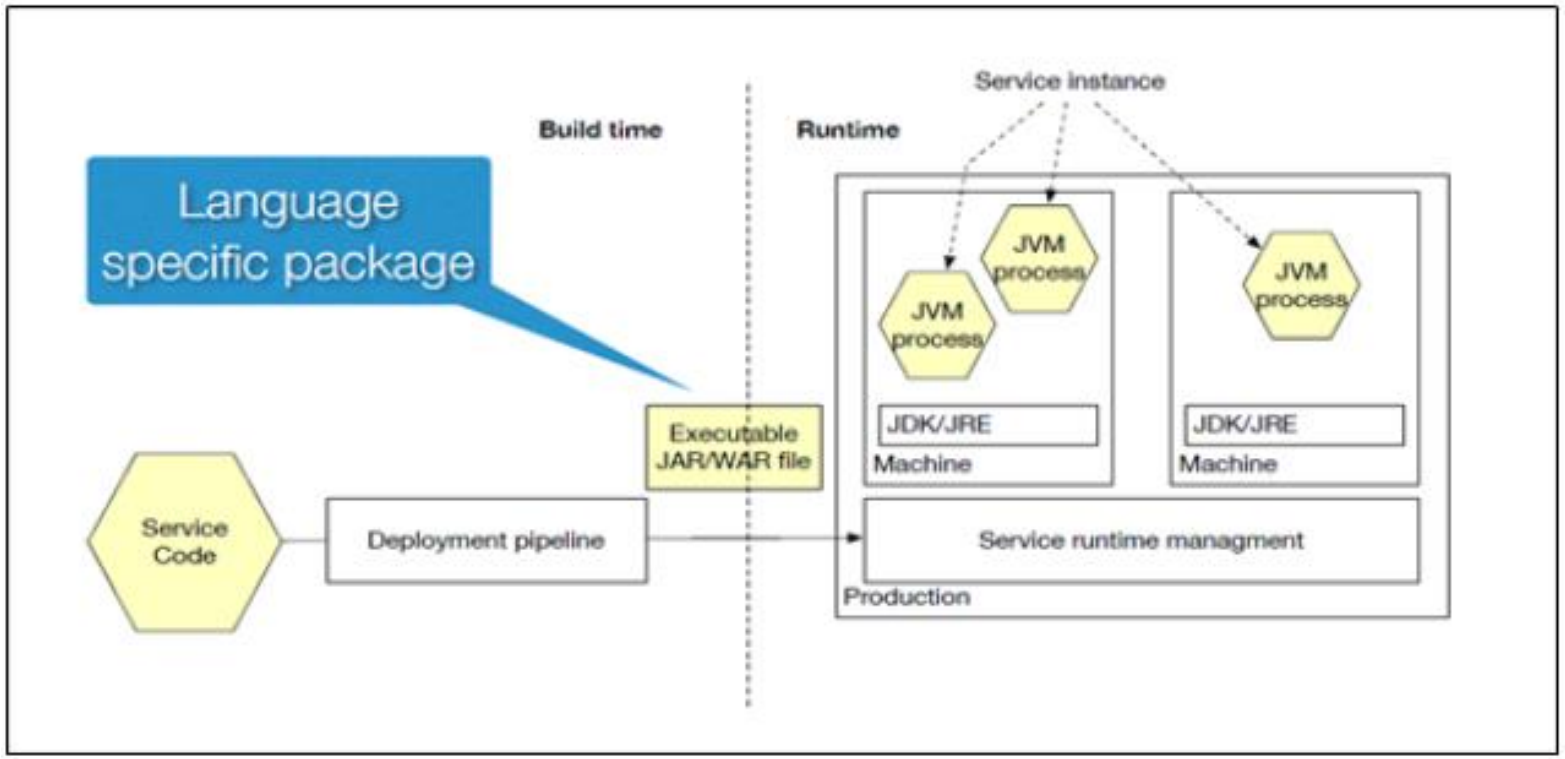
5.1 배포 패턴

- MSA의 배포 형태는 여러가지가 될 수 있다.
- 대표적으로 4가지 형태의 패턴으로 정의
- Service per container, Serverless deployment 패턴이 MSA에 적합한 패턴



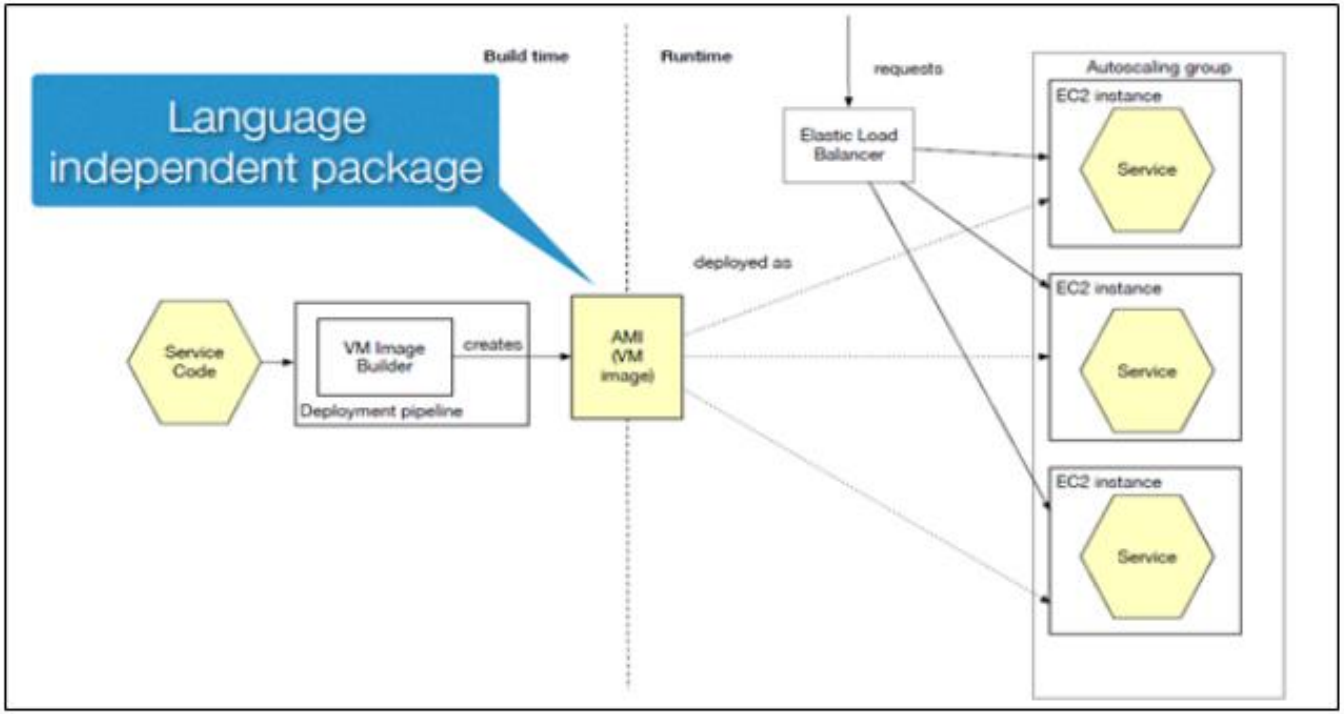
Multiple Service per host

- 호스트(물리적 또는 가상의) 하나에 서비스를 여러 개 배포하는 유형
- MSA 환경보다는 전통적인 온프레미스 또는 VM 기반 환경에서 멀티 서비스를 구성할 때 많이 사용하는 패턴
- 서비스별 자원 제한 불가
- 특정 서비스가 CPU, 메모리 등의 자원을 많이 사용한다면 타 서비스에 영향



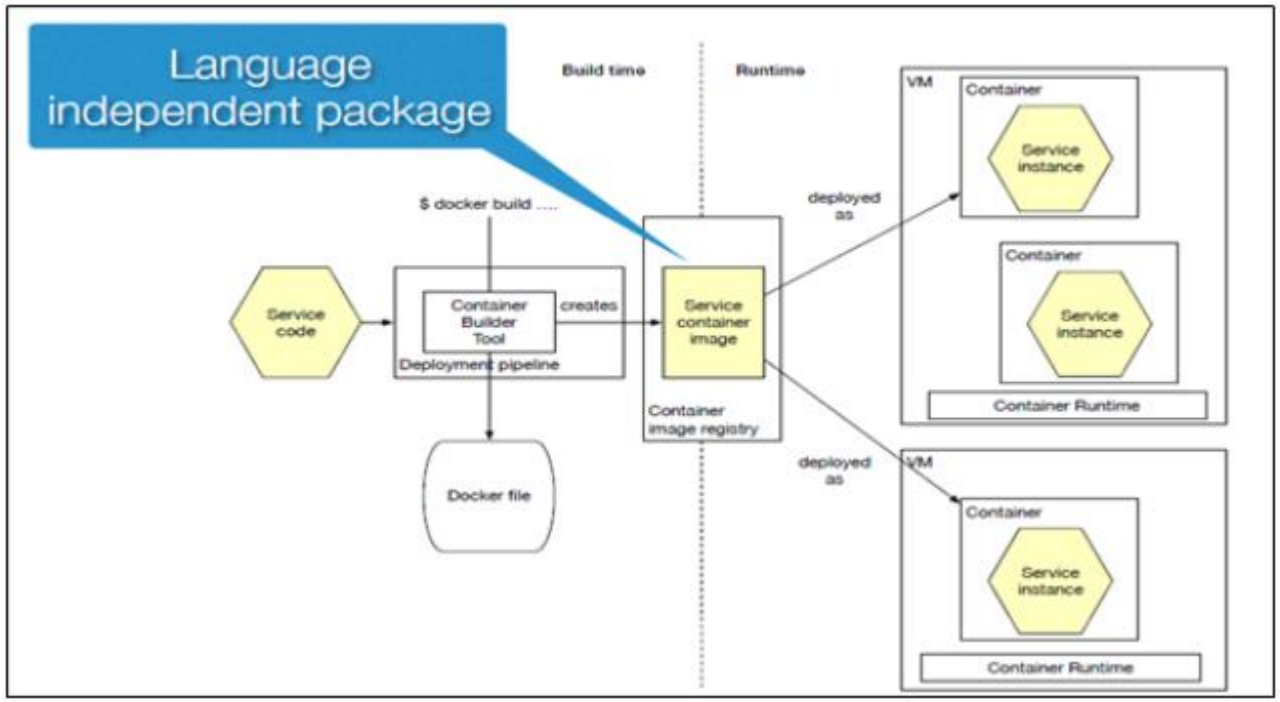
Service per VM

- 각 서비스는 VM 이미지로 패키징 되고 VM 이미지를 이용해서 VM 상에서 동작하는 서비스 인스턴스를 배포
- VM 1개당 1개의 서비스를 제공하는 방식으로 VM은 Cloud 환경에 구성할 수 있으며 VM 이미지를 배포한다.
- 호스트별로 캡슐화되어 각 서비스간 독립
- AutoScaling, 로드밸런싱과 같은 클라우드 제공 기능 사용 가능
- OS 포함으로 인해 이미지 사이즈 큼
- 느린 실행 시간으로 인해 빠른 배포의 어려움



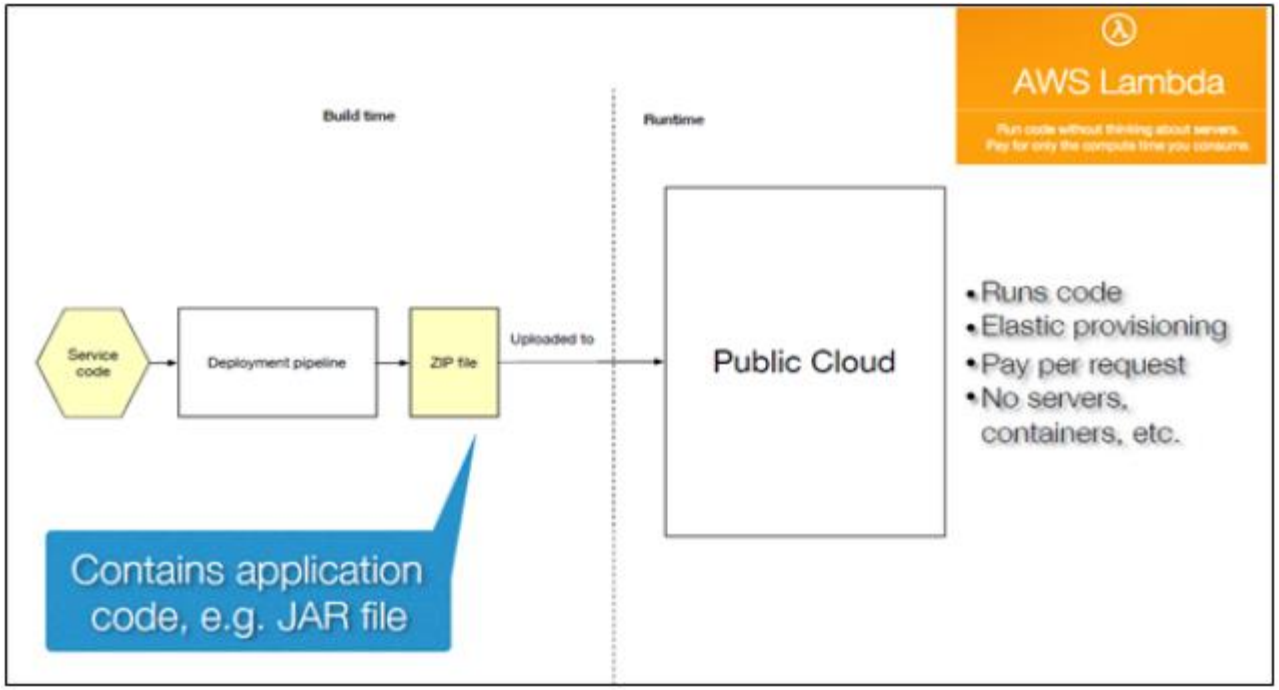
Service per Container

- 각 서비스 응용 프로그램을 컨테이너 이미지로 패키지화하고 이를 배포하는 방식
- 컨테이너 이미지는 코드, 런타임, 시스템 도구, 시스템 라이브러리, 설정을 실행하는 데 필요한 모든 것을 포함하는 소프트웨어 조각의 경량 독립 실행 형 패키지
- 컨테이너 하다 당 한 개의 서비스를 제공하는 방식
- 가볍고 빠른 배포가 가능한 컨테이너의 장점을 활용한 방식으로 컨테이너 이미지를 배포
- kubernetes 같은 오케스트레이션 도구를 통해 컨테이너 관리 가능
- OS 커널 공유로 인한 보안 문제 / 오케스트레이션 도구 사용 시 러닝커브

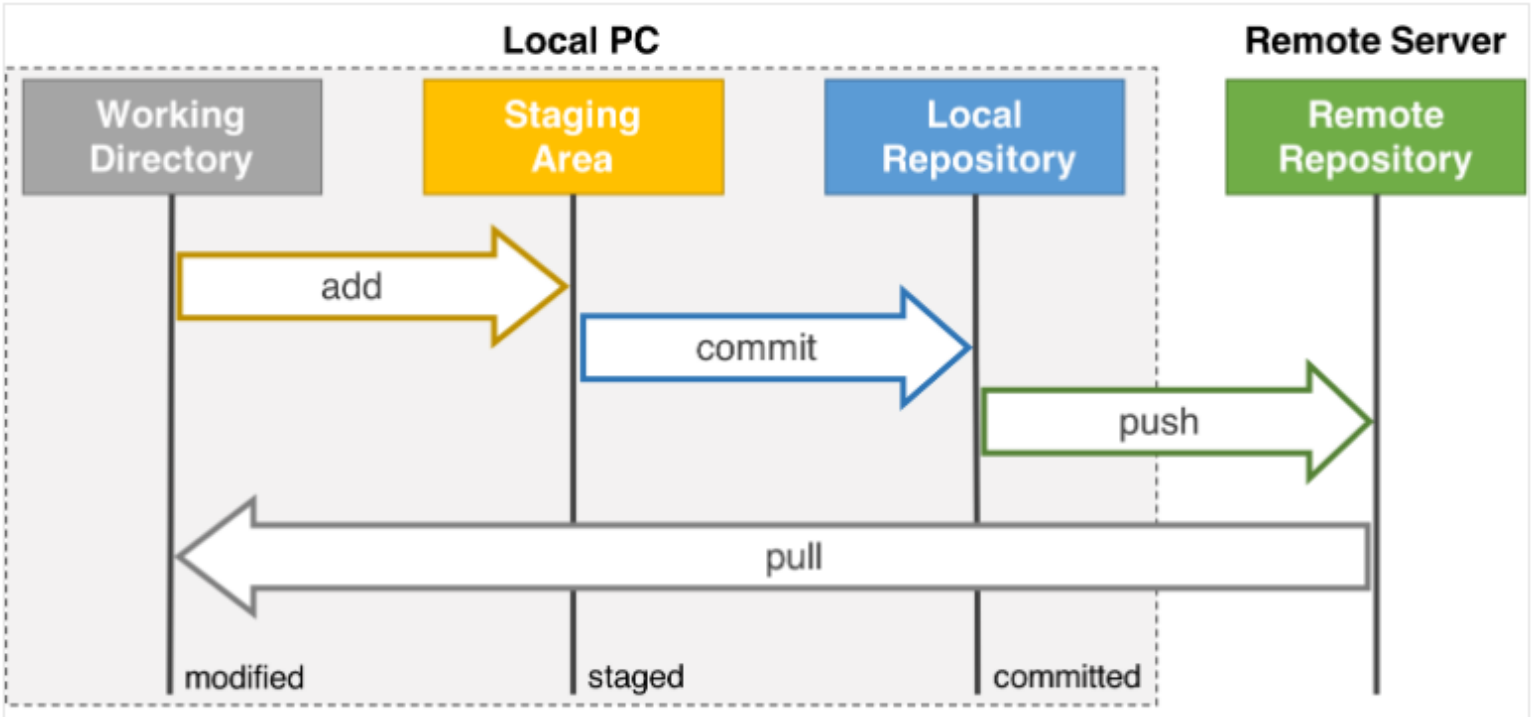


Serverless

- 서버리스(serverless)는 실제 서버가 없는 것은 아니지만 서버, VM, 컨테이너 등에 대해서 고민하지 않고 애플리케이션 개발에 집중할 수 있어서 서버가 없다(serverless)라고 표현
- 시간, 메모리, 네트워크 등 사용량에 따라 비용을 지불하는 구조로 서비스를 제공(네트워크, 메모리 등 서버 관리에 자유로움)
- 별도의 인프라 구성없이 Public Cloud에 실행시킬 소스만 업로드하여 Public Cloud 내부 런타임 환경을 통해 실행하는 방식으로 소스코드만 배포
- 마이크로서비스 배포 편리
- Cloud 서버리스 서비스에서 지원하는 언어만 사용 가능

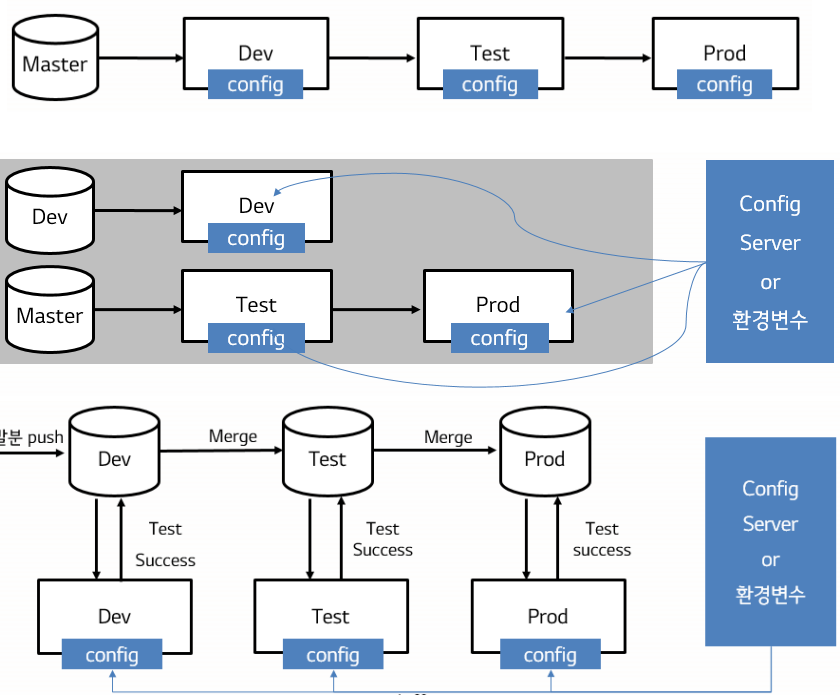


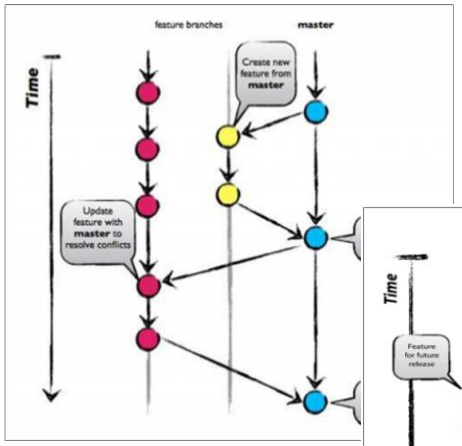
- Working Directory : 현재 작업 중인 Git 프로젝트 파일들이 있는 내 PC의 디렉토리
- Staging Area : 커밋할 변경 내역들의 대기 장소. 'git add' 명령어를 사용하면 Working Directory의 변경내역을 Staging Area에 add
- Local Repository : 커밋들이 스냅샷으로 기록된 곳. 'git commit' 명령어를 사용하면 Staging Area에 있는 변경내역들을 실제 스냅샷으로 묶어 Local Repository에 올라감. 즉, 하나의 commit이 하나의 버전의 구분점이 되며 이전 버전에서 해당 버전으로의 변경내역이 담겨 있음
- Remote Repository : 로컬 PC를 넘어 원격 서버에서 관리되는 저장소. 이 곳에 올라온 커밋은 다른 사람들도 참조 가능.

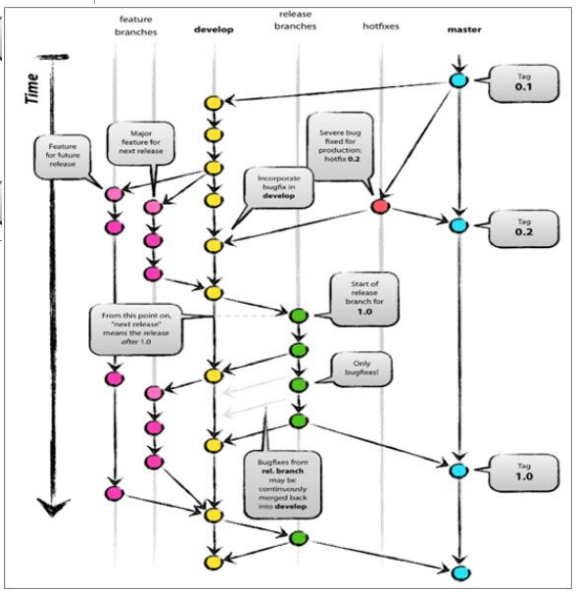


5.2 브랜치 전략

- 소스형상관리(Git) 브랜치 전략은 브랜치를 몇 개로 가져가는지 용도가 무엇인지에 따라서 다르다.
- 상황에 따라 1개의 브랜치에서 전체 환경에 배포하는 전략, 2개의 브랜치, 각 환경별로 브랜치를 구성하는 전략, sub 브랜치를 구성하는 전략 등 다양한 형태로 설계
- 검증된 git-flow, github-flow와 같은 브랜치 전략도 존재



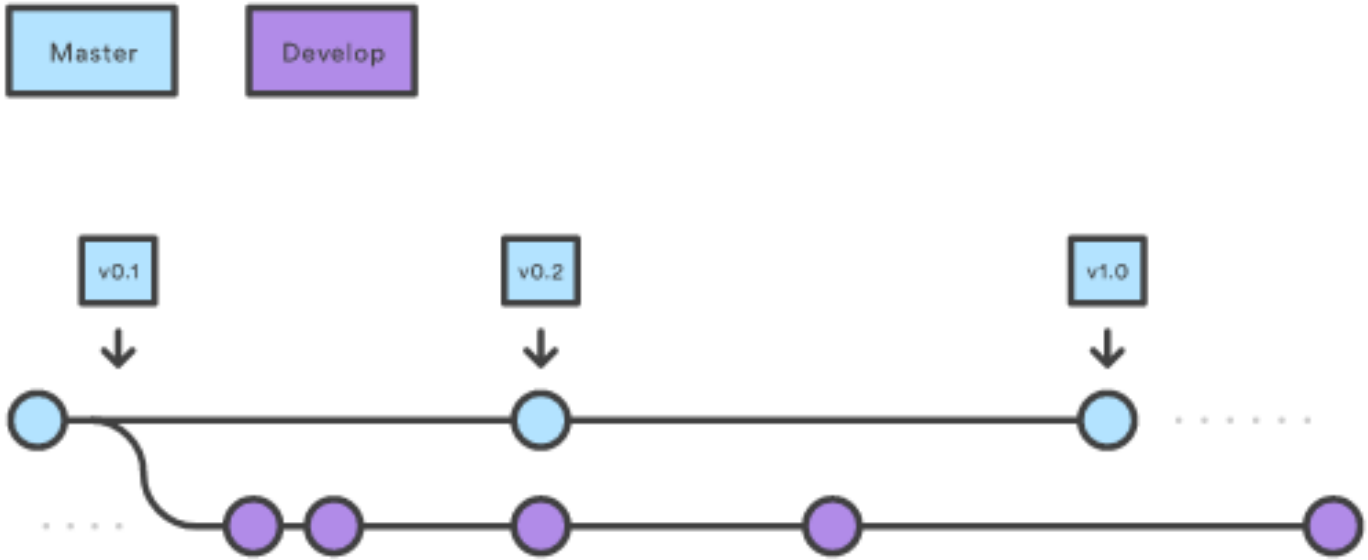




브랜치 종류

Master 브랜치

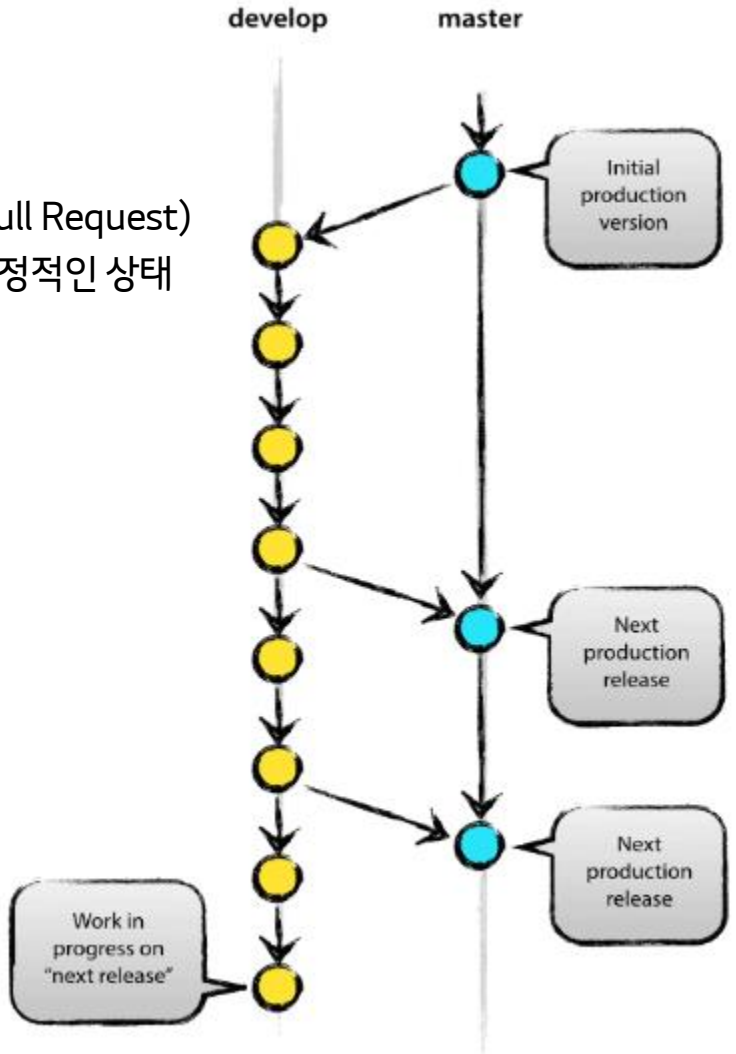
- Repository 생성 시, 기본 제공 브랜치
- 제품으로 출시될 수 있는 브랜치
- 배포 이력을 관리하기 위해 사용. 즉 배포 가능한 상태만을 관리한다.



브랜치 종류

Develop 브랜치

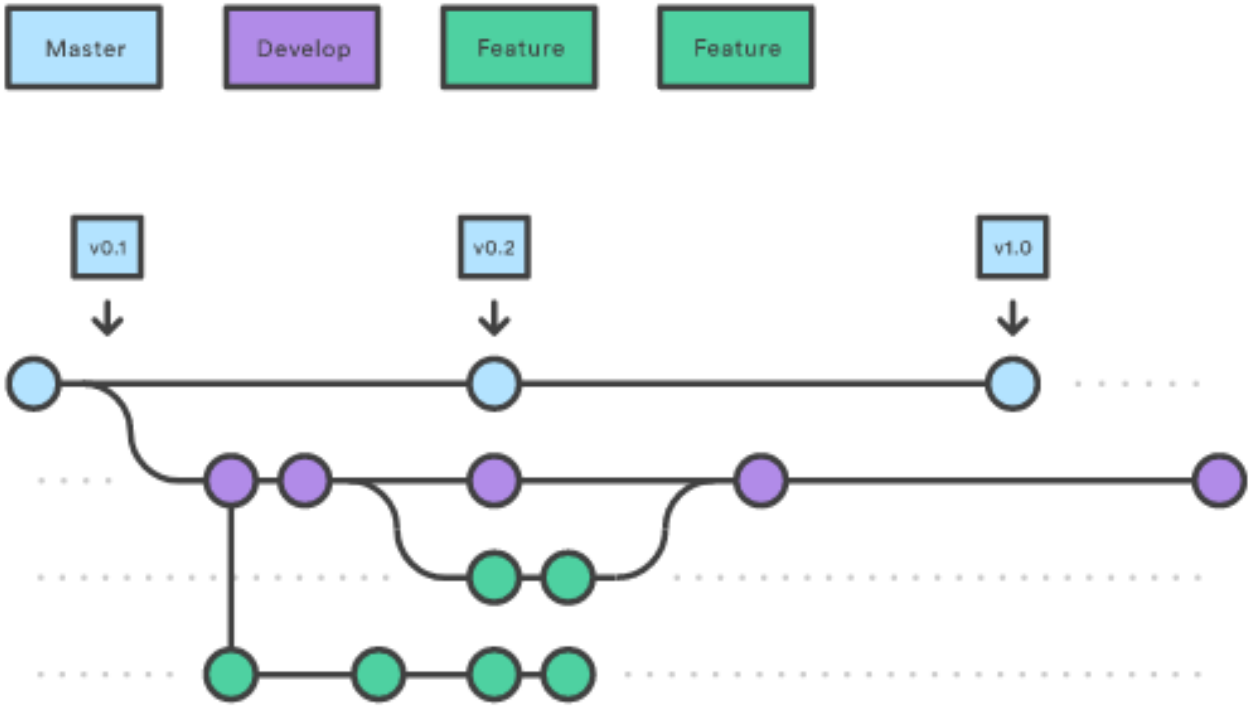
- 다음 출시 버전을 개발하는 브랜치
- 기능 개발을 위한 브랜치들을 merge하기 위해 사용(Pull Request)
- 모든 기능이 추가되고 버그가 수정되어 배포 가능한 안정적인 상태라면 develop 브랜치를 master 브랜치에 merge
- 평소에는 이 브랜치를 기반으로 개발 진행



브랜치 종류

Feature 브랜치

- 기능을 개발하는 브랜치
- 새로운 기능 개발 및 버그 수정이 필요할 때마다 develop 브랜치로부터 분기
- feature 브랜치에서의 작업은 기본적으로 공유할 필요가 없기 때문에 자신의 로컬 저장소에서 관리한다.
- 개발이 완료되면 develop 브랜치로 merge하여 다른 사람들과 공유



브랜치 종류

Feature 브랜치

1) develop 브랜치에서 새로운 기능에 대한 feature 브랜치를 분기한다.
2) 새로운 기능에 대한 작업을 수행한다.
3) 작업이 끝나면 develop 브랜치로 merge 한다.
4) 더 이상 필요하지 않은 feature 브랜치는 삭제한다.
5) 새로운 기능에 대한 feature 브랜치를 중앙 원격 저장소에 올린다.(push)

참고) --no-ff 옵션

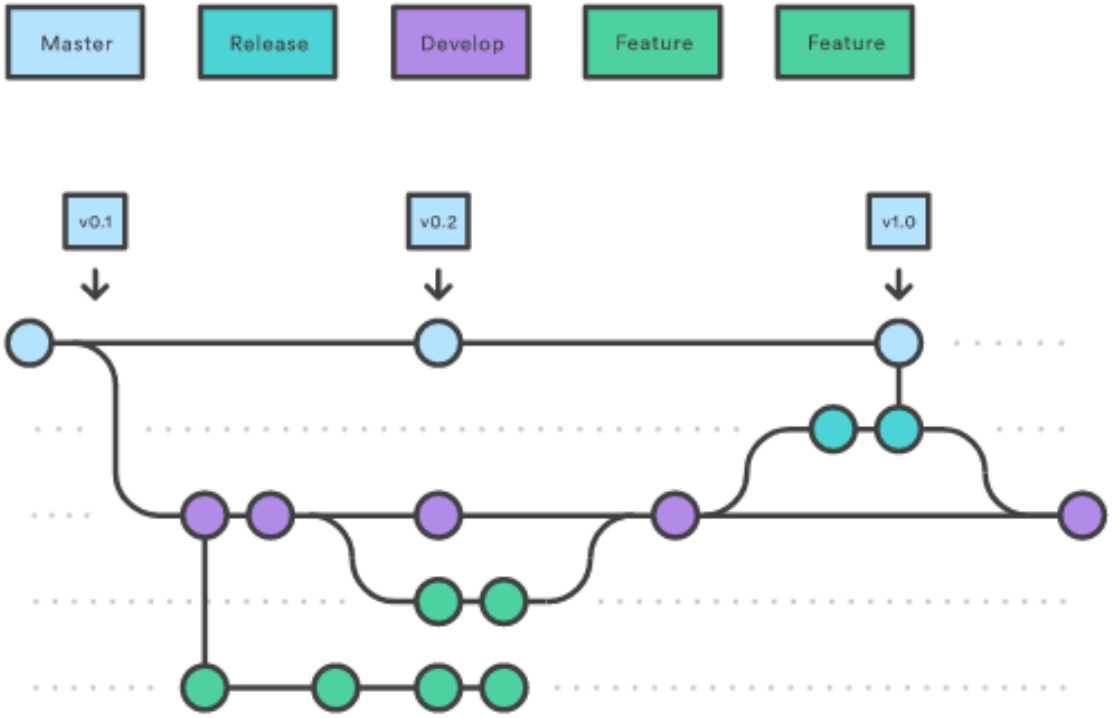
- 새로운 커밋 객체를 만들어 'develop' 브랜치에 merge
- 'feature' 브랜치에 존재하는 커밋 이력을 모두 합쳐서 하나의 새로운 커밋 객체를 만들어 'develop' 브랜치로 merge

```
// feature 브랜치(feature-login)를 'develop' 브랜치('master' 브랜치에서 따는 것이 아니다!)에서 분기
$ git checkout -b feature-login develop

/* ~ 새로운 기능에 대한 작업 수행 ~ */

/* feature 브랜치에서 모든 작업이 끝나면 */
// 'develop' 브랜치로 이동한다.
$ git checkout develop
// 'develop' 브랜치에 feature-login 브랜치 내용을 병합(merge)한다.
// --no-ff 옵션: 왼쪽 설명 참고
$ git merge --no-ff feature-login
// -d 옵션: feature-login에 해당하는 브랜치를 삭제한다.
$ git branch -d feature-login
// 'develop' 브랜치를 원격 중앙 저장소에 올린다.
$ git push origin develop
```

브랜치 종류

Release 브랜치

- 이번 출시 버전을 준비하는 브랜치
- 배포를 위한 전용 브랜치를 사용함으로써 한 팀이 해당 배포를 준비하는 동안 다른 팀은 다음 배포를 위한 기능 개발을 계속할 수 있다.
- 배포 마일스톤을 정하고 개발 및 의사소통하기 좋음



브랜치 종류

- **Release 브랜치**

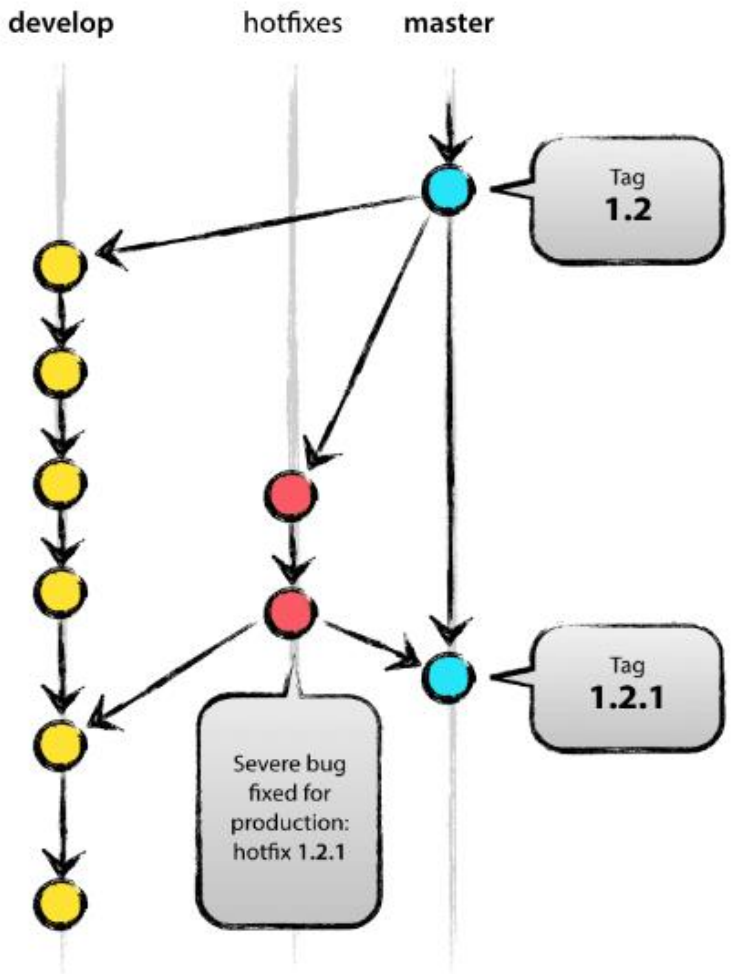
1) develop 브랜치에서 배포할 수 있는 수준의 기능이 모이거나 배포 일정이 정해지면 release 브랜치를 분기한다.
2) 배포를 위한 최종적인 버그 수정, 문서 추가 등 릴리즈와 직접적으로 관련된 작업을 수행한다.
3) 직접적으로 관련된 작업들 외에는 release 브랜치에 새로운 기능을 추가로 merge 하지 않는다.(develop 브랜치까지만 merge)
4) release 브랜치에서 배포 가능한 상태가 되면 master 브랜치에 merge 한다.(merge 시 release 버전 태깅)
5) release 브랜치가 변경되었을 수 있으므로 develop 브랜치에도 merge

```
// release 브랜치(release-1.2)를 'develop' 브랜치('master' 브랜치에서 따는 것이 아니다!)에서 분기
$ git checkout -b release-1.2 develop

/* ~ 새로운 기능에 대한 작업 수행 ~ */

/* release 브랜치에서 배포 가능한 상태가 되면 */
// 'master' 브랜치로 이동한다.
$ git checkout master
// master' 브랜치에 release-1.2 브랜치 내용을 병합(merge)한다.
$ git merge --no-ff release-1.2
// 병합한 커밋에 Relase 버전 태그를 부여한다.
$ git tag –a 1.2

/* 'release' 브랜치의 변경 사항을 'develop' 브랜치에도 적용 */
// 'develop' 브랜치로 이동한다.
$ git checkout develop
// 'develop' 브랜치에 release-1.2 브랜치 내용을 병합(merge)한다.
$ git merge --no-ff release-1.2
// -d 옵션: release-1.2에 해당하는 브랜치를 삭제한다.
$ git branch -d release-1.2
```

브랜치 종류

Hotfix 브랜치

- 출시 버전에서 발생한 버그를 수정하는 브랜치
- master 브랜치에서 분기하는 브랜치로 develop 브랜치에서 문제가 되는 부분을 수정하여 배포 가능한 버전을 만들기에는 많은 시간 소요
- 바로 배포가 가능한 master 브랜치에 merge하여 배포
- 버그 수정만을 위한 브랜치를 따로 만들었기 때문에 다음 배포를 위해 개발하던 작업 내용에 전혀 영향을 주지 않음
- master 브랜치를 부모로 하는 임시 브랜치



브랜치 종류

- **Hotfix 브랜치**

1) 배포한 버전에 긴급하게 수정을 해야할 필요가 있을 경우 master 브랜치에서 hotfix 브랜치를 분기한다.
2) 문제가 되는 부분을 수정한다.
3) master 브랜치에 merge 하여 다시 배포한다.
4) 새로운 버전 이름으로 태깅한다.
5) hotfix 브랜치에서의 변경사항은 develop 브랜치에도 merge 한다.

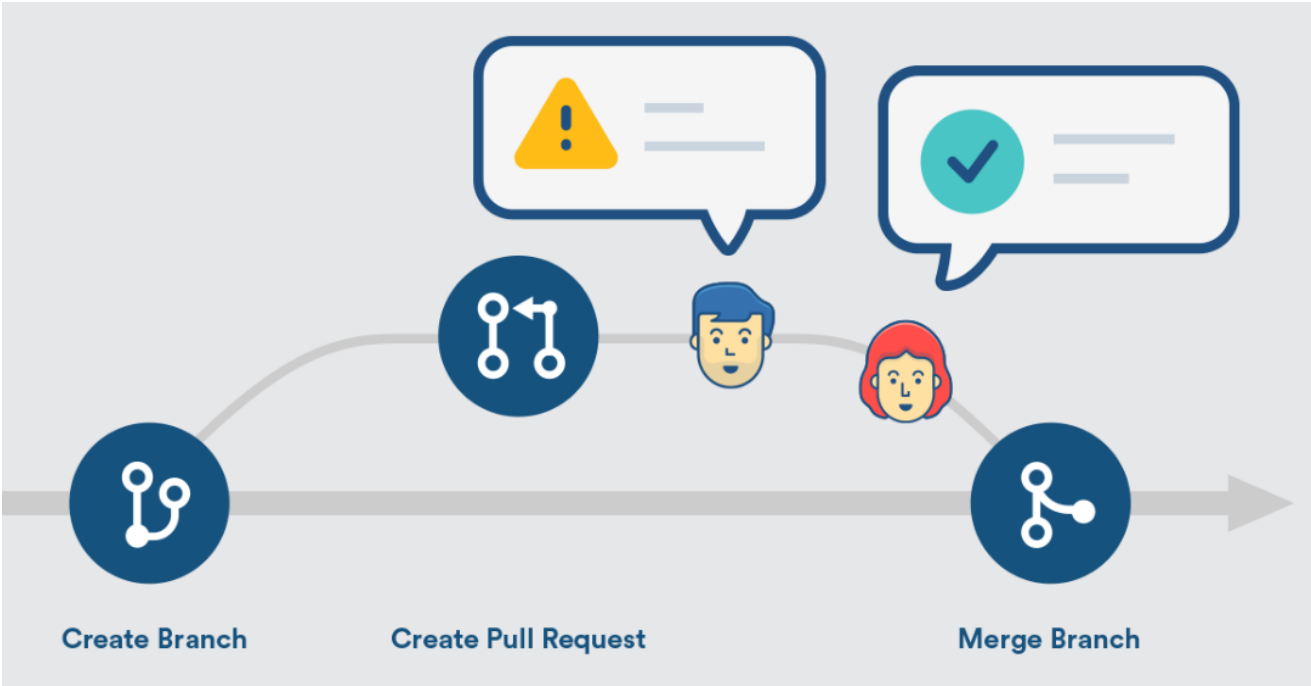
```
// release 브랜치(hotfix-1.2.1)를 'master' 브랜치(유일!)에서 분기
$ git checkout -b hotfix-1.2.1 master

/* ~ 문제가 되는 부분 수정  ~ */

/* 수정 완료되면 */
//  ' master '  브랜치로 이동한다.
$ git checkout master
// 'master '  브랜치에 hotfix-1.2.1 브랜치 내용을 병합(merge)한다.
$ git merge –no-ff hostfix-1.2.1
// 병합한 커밋에 버전 태그를 부여한다.
$ git tag –a 1.2.1

/* 'hotfix' 브랜치의 변경 사항을 'develop' 브랜치에도 적용 */
// 'develop' 브랜치로 이동한다.
$ git checkout develop
// 'develop' 브랜치에 release-1.2 브랜치 내용을 병합(merge)한다.
$ git merge --no-ff hostfix-1.2.1
```

Pull Request(Merge Request)

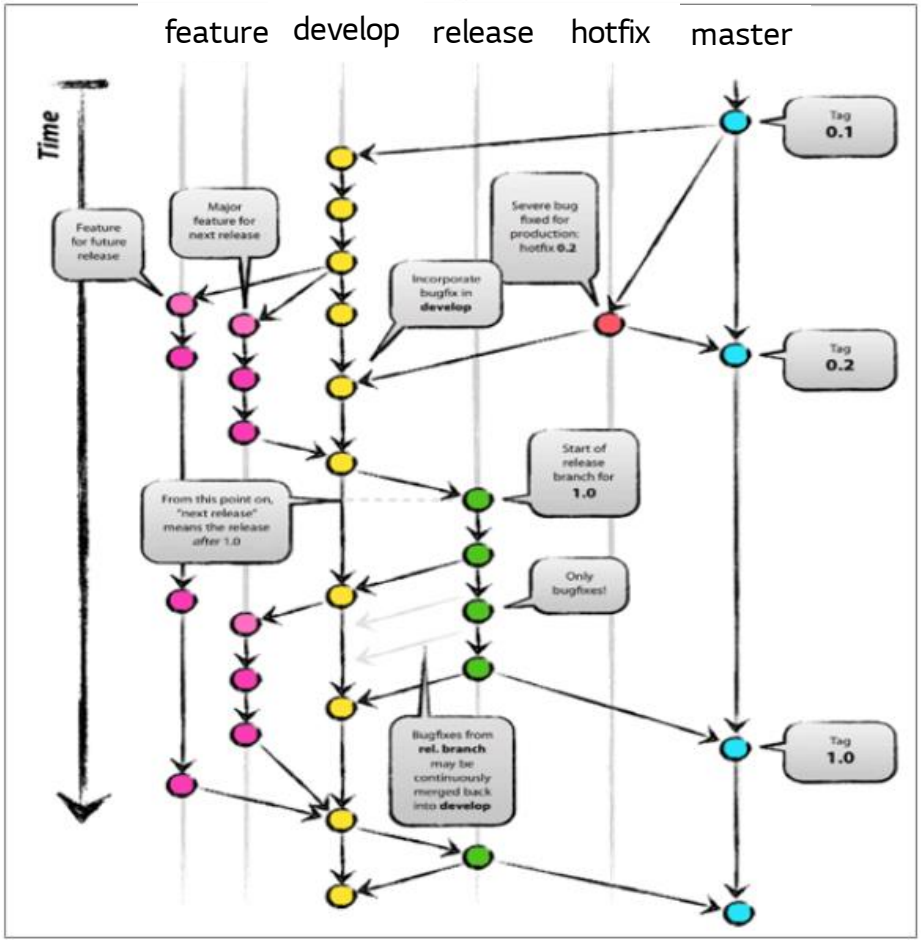
- Git의 협업 기능
- 소스형상관리의 브랜치 간 merge를 위한 요청 (내가 작업한 브랜치를 당겨 검토 후 merge 해주세요!)
- 자연스러운 코드리뷰 (공동 리뷰/학습)
- Pull Request에 대해 댓글로 토론 가능
- 코드 리뷰 완료 시 브랜치에 merge → 브랜치 품질 유지
- Push 권한이 없는 오픈 소스 프로젝트에서 많이 사용

Pull Request 순서

① 신규 branch 생성
② 수정 작업 후 add, commit, push
③ Pull Request 생성
④ 코드리뷰, Merge 승인
⑤ (권장)Merge 후 branch 삭제
- Pull request 옵션으로 삭제 가능



Git-flow 브랜치 전략

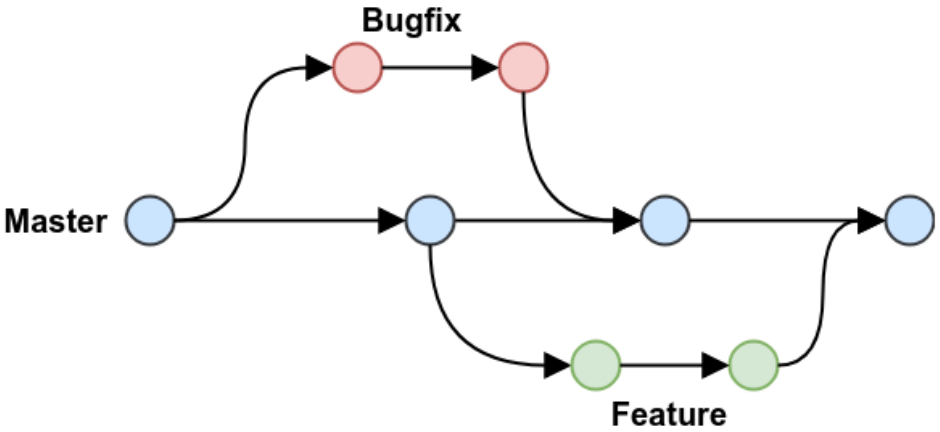


- 여러 개의 브랜치와 flow로 인해 메시지, 버전만으로 파악이 쉽지 않지만 flow를 보여주는 여러 도구로 인해 한눈에 이력/버전 파악이 쉬워짐
- 브랜치를 역할별로 좀 더 상세하게 나누어 5개의 브랜치를 구성
- 항상 유지되는 메인 브랜치(master, develop)과 일정 기간 동안만 유지되는 보조 브랜치(feature, release, hotfix)

브랜치	설명
master	제품으로 출시될 수 있는 브랜치
develop	다음 출시 버전을 개발하는 브랜치
feature	기능을 개발하는 브랜치
release	이번 출시 버전을 준비하는 브랜치
hotfix	출시 버전에서 발생한 버그를 수정하는 브랜치

- *SI와 같은 개발보다는 솔루션과 같이 버전/이력 관리가 좀 더 필요한 상황에 적합*
- *규모가 작은 프로젝트의 경우 오히려 복잡한 구성으로 인해 에포트가 더 소모될 수 있다.*

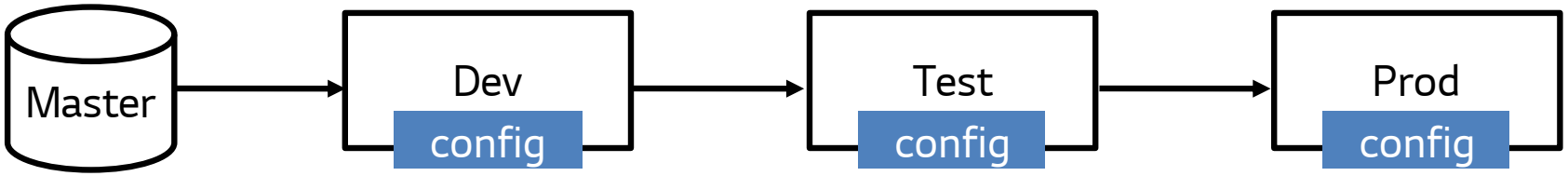
Github-flow 브랜치 전략



- 심플한 브랜치 구성
- 빠른 출시가 필요한 경우 적합
- 버그수정, 기능개발 모두 master 브랜치에서 분기
- 작업 완료 시 master 브랜치로 merge(충분한 검토/테스트 후)
- master 브랜치 기준으로 제품 출시
- *쉽고 빠른 배포가 가능하여 버전관리보다는 빠른 제품 출시가 필요한 프로젝트에 적합*

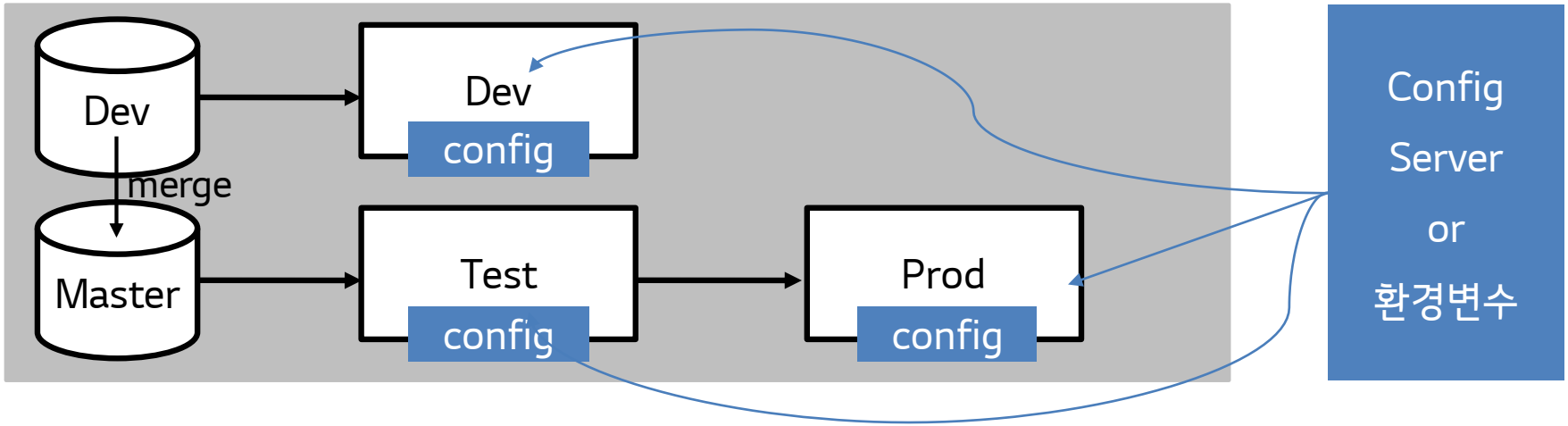
One 브랜치 전략(No-flow)

- 1개의 브랜치로 개발, 테스트, 운영환경까지 모두 배포하는 전략
- 타브랜치로의 merge 과정이 없어 운영환경 배포까지 가장 빠름
- 빠른 속도의 제품 출시에 적합(Prod 환경까지의 빠른 배포)
- 브랜치 품질 유지가 중요하므로 로컬에서 충분한 테스트와 품질 체크 후 push 필요
- 브랜치 품질이 깨진 경우 빠른 피드백 필요
- 각 환경에 배포시 Config설정 + 동일 이미지 배포



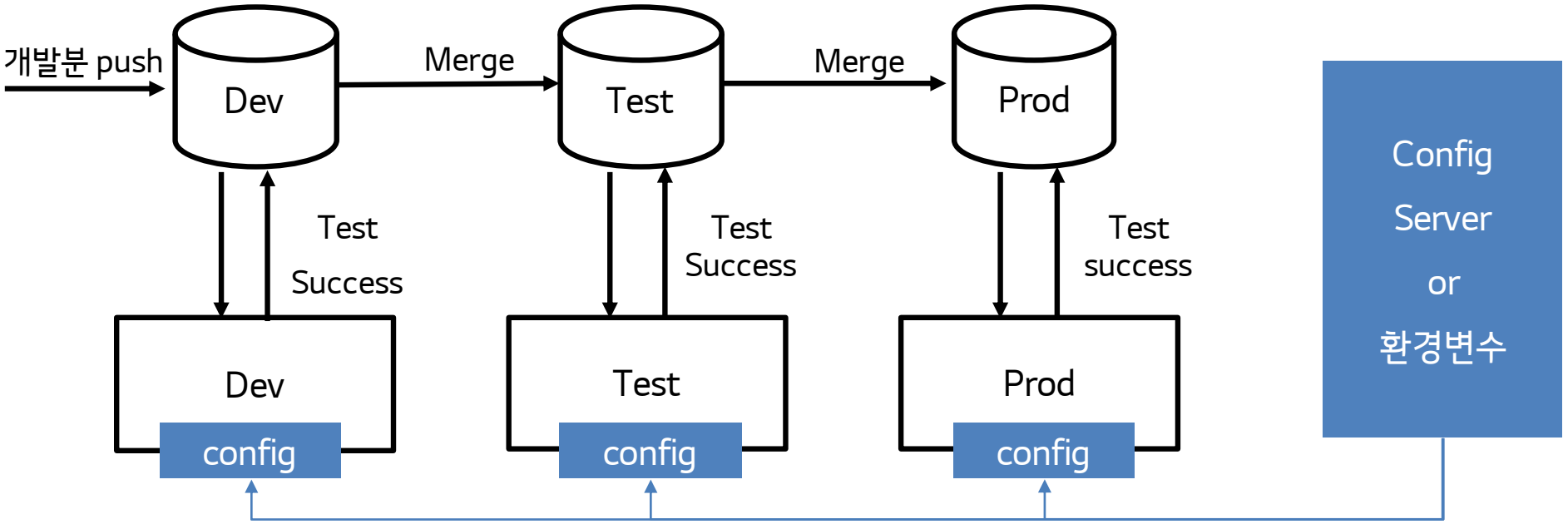
Two 브랜치 전략

- Dev 브랜치는 개발환경 배포 대상 브랜치
- Master 브랜치는 테스트환경, 운영환경 배포 대상 브랜치
- Dev → Master 브랜치로의 Merge 과정에 승인 적용가능(Pull Request)
- Master 브랜치의 버전 관리, 품질 유지
- 빠른 배포와 일정 수준 이상의 품질 유지하기에 적합



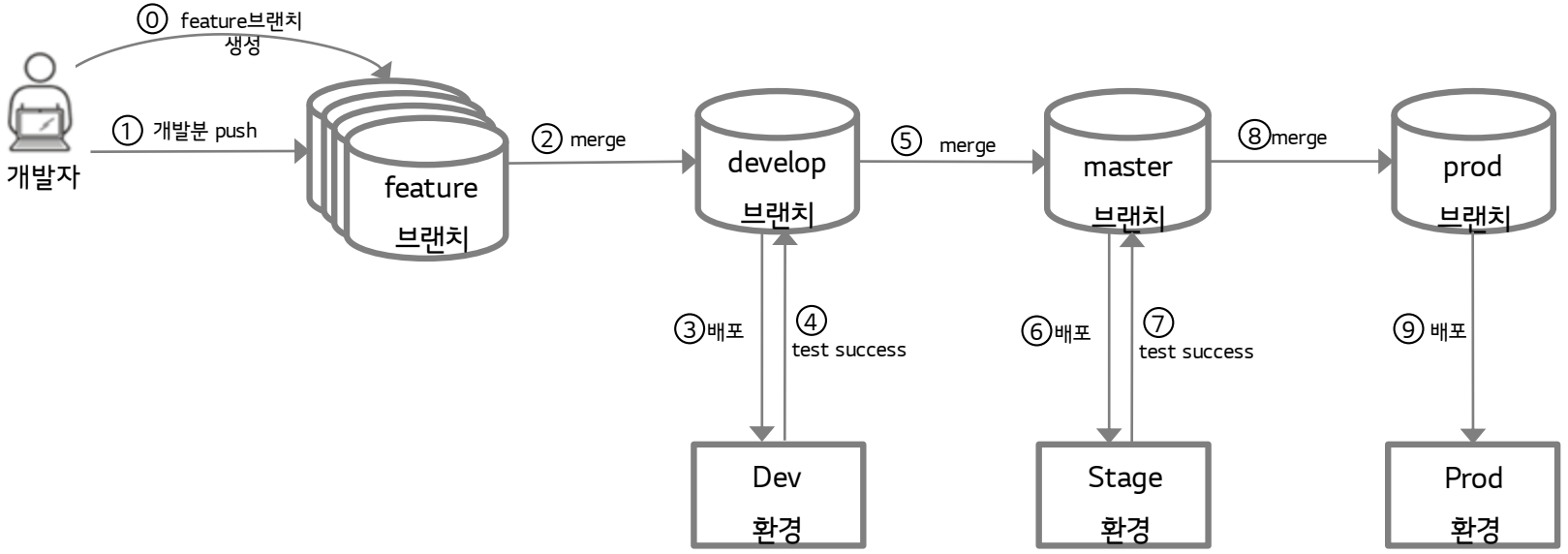
Three 브랜치 전략

- 환경별로 브랜치를 구성해 배포/테스트하는 전략
- 환경별 소스 분리되어 관리되며 환경별 소스 이력 확인 가능
- 각 환경별 브랜치를 두어 제어를 강화
- 브랜치 간 영향도 최소화
- Dev → Test → Prod 브랜치로 갈 수록 품질 증가하며 환경별 브랜치 품질 유지 쉬움
- 브랜치 merge 등의 작업이 필요하므로 Prod 배포까지의 주기가 긴 편 (속도 < 품질)
- 초급개발자부터 고급개발자까지 다양한 개발자풀을 가진 경우 품질 유지를 위해 적합



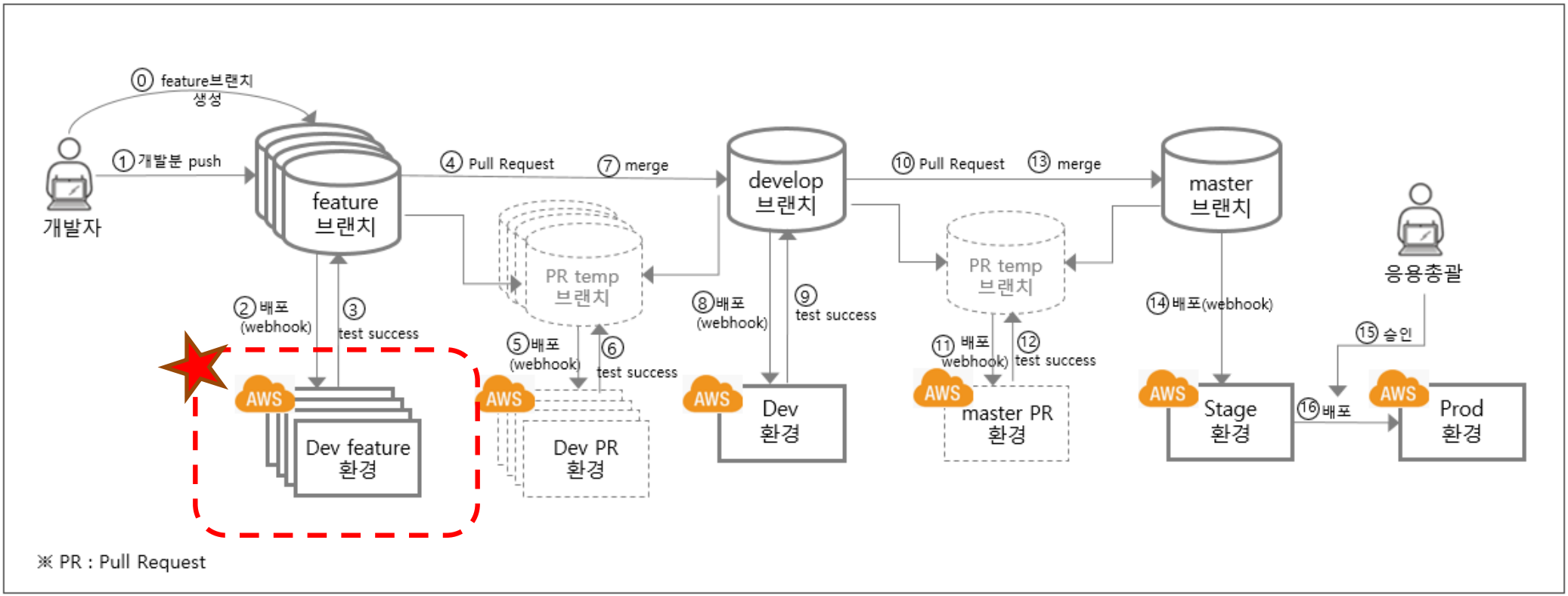
사례 1. feature 브랜치 활용

- Three 브랜치와 유사한 구조
- 개발자가 dev 브랜치에 직접 push하지 않고 feature 브랜치 생성 후, 작업
- feature 브랜치에서 작업 작업 완료 시, develop 브랜치로 merge 요청(Pull Request)
- Pull Request를 통해 코드 리뷰 진행/승인을 거쳐 develop 브랜치에 merge 됨
- feature 브랜치를 통해 기능별/이슈별 이력 관리 가능



사례 2. feature 브랜치 & IaC 활용

- 사례 1. feature 브랜치 활용 사례에서 각 feature에 대한 환경 제공 전략
- 빌드센터(Slalom) 사례
- feature 브랜치 파이프라인에서 IaC(terraform)을 통해 feature 환경 생성
- feature 환경 생성은 곧 비용이므로 비용과 품질 등을 고려하여 도입 결정 필요



5.3 배포 기법

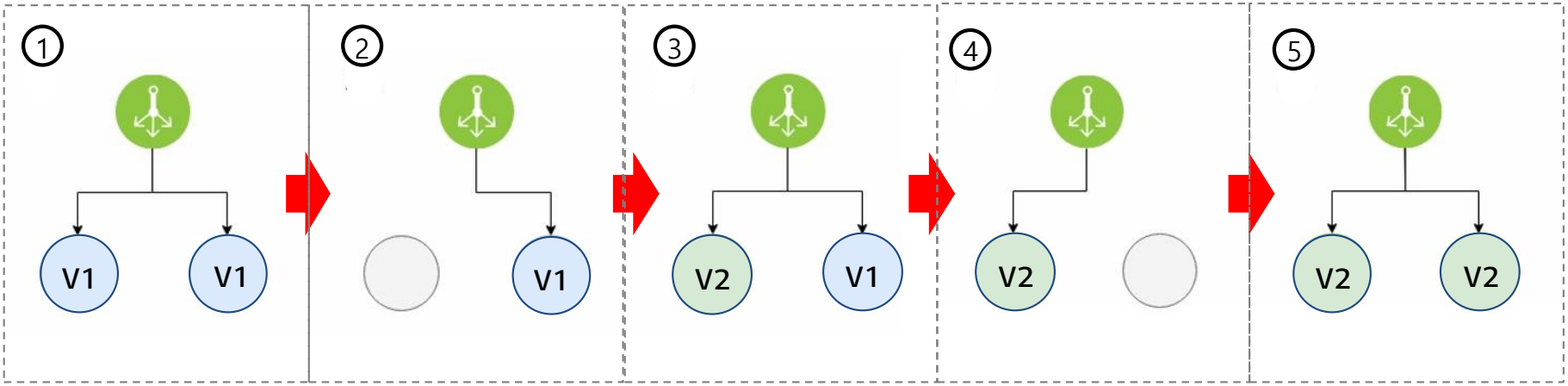
- MSA는 서비스를 더 작게 만들어 보다 많은 서비스가 존재하고 빠른 개발주기를 위해 더 자주 배포해야 한다. 이런 이유로 다양한 배포 유형이 발전되고 있다.

구현 난이도

롤링 업데이트	블루/그린	카나리
• **단순하게 한 대씩 재시작하는 배포 방법** • 구성이 간단함 • 서버의 제약이 있을때 사용가능 • **저 위험 프로젝트 또는 개발 환경에서 사용**	• 새로운 버전을 배포하고 새 버전으로 스위칭하는 배포 유형 • 코드 변경 따른 사이드 이펙트 없음 • **운영 환경 또는 무중단 필요시 사용**	• 1대 또는 특정 user에게만 미리 배포했다가 잘되면 전체 배포하는 유형 • 구현 난이도 높음 • **소스코드 변경이 많아 영향도 파악이 어려울 때 사용(무중단 가능)**

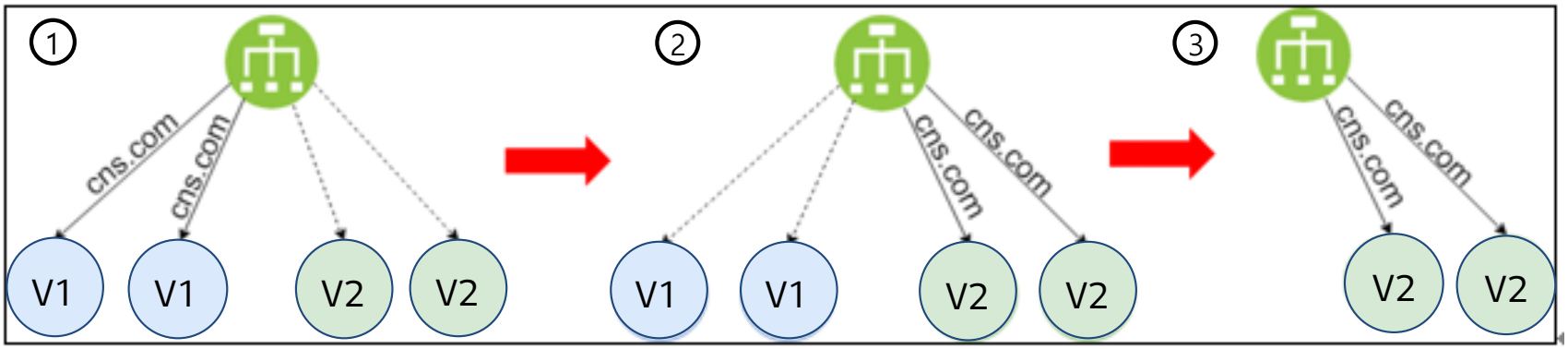
롤링 업데이트

- 롤링 업데이트 배포는 서버를 한 대씩 구 버전에서 새 버전으로 교체해가는 배포 유형
- 일반적으로 가장 많이 사용하는 방식
- 서버 수의 제약이 있을 경우 유용하나 배포 중 인스턴스의 수가 감소 되므로 서버 처리 용량을 미리 고려 필요



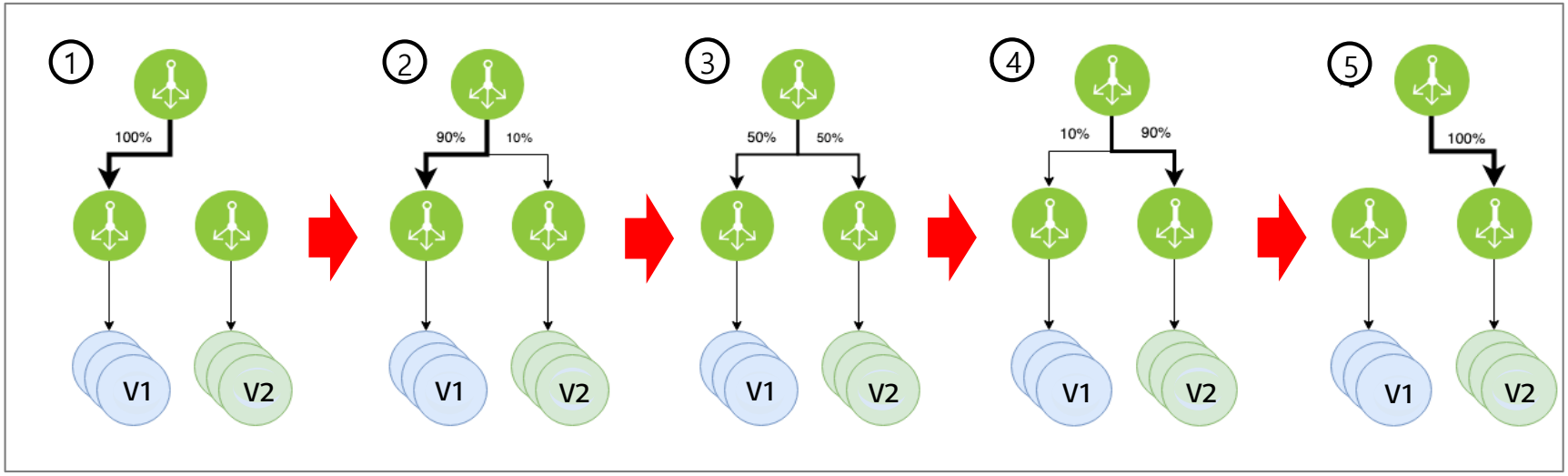
블루/그린

- Blue/Green 배포는 구 버전에서 새 버전으로 일제히 전환하는 배포 유형
- 구 버전의 서버와 새 버전의 서버들을 동시에 나란히 구성하고 배포 시점이 되면 트래픽을 일제히 전환
- 하나의 버전만 프로덕션 되므로 버전 관리 문제를 방지할 수 있음(V1과 V2 혼재하는 시간 없이 전환 가능)
- V1에 대한 시스템이 스탠바이 상태로 있으므로 빠른 롤백 가능
- 운영 환경에 영향을 주지 않고 실제 서비스 환경으로 새 버전 테스트가 가능

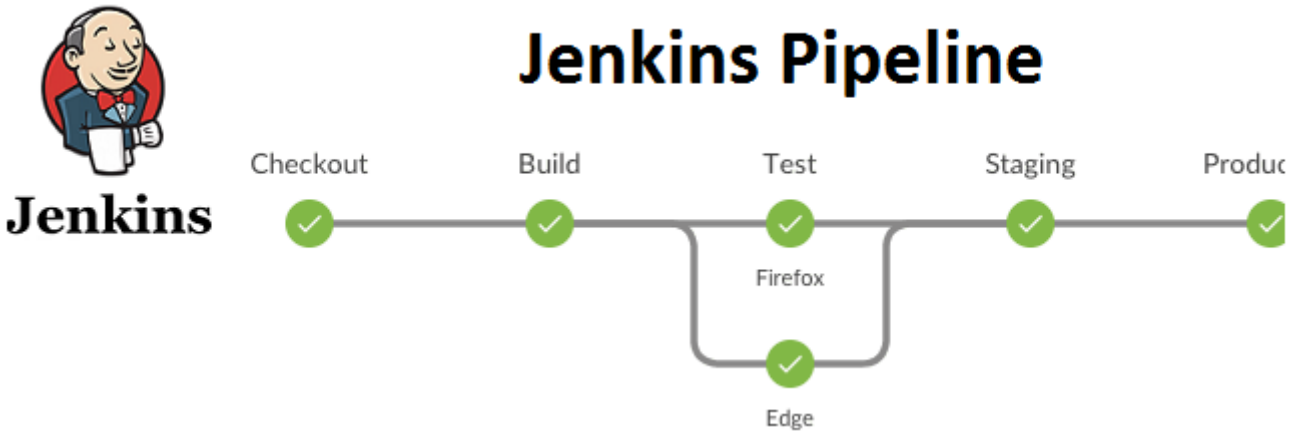


Canary

- 블루/그린 배포와 마찬가지로 구 버전의 서버와 새 버전의 서버들을 구성
- 신규버전으로 트래픽을 일제히 전환하는 블루/그린 방식과는 다르게 일부 트래픽을 새 버전으로 분산하여 오류를 판단하는 배포 기법
- 특정 user에게만 일부 트래픽을 열어주어 V2에 대한 테스트를 수행하는 방식으로 점차 V2에 대한 트래픽 비율을 늘리면서 버전을 전환
- 100% 배포까지의 시간이 소요되며 한번에 신규버전으로 전환하여 V1과 V2과 공존하는 시간이 없는 블루/그린 배포와 다르게 V1과 V2의 공존 시간이 긴 편
- 트래픽 비율 제어와 같은 정교한 트래픽 라우팅이 필요하여 블루/그린배포 보다 복잡한 설정을 해야 한다는 단점
- Spinnaker에서는 레드/블랙 용어 사용

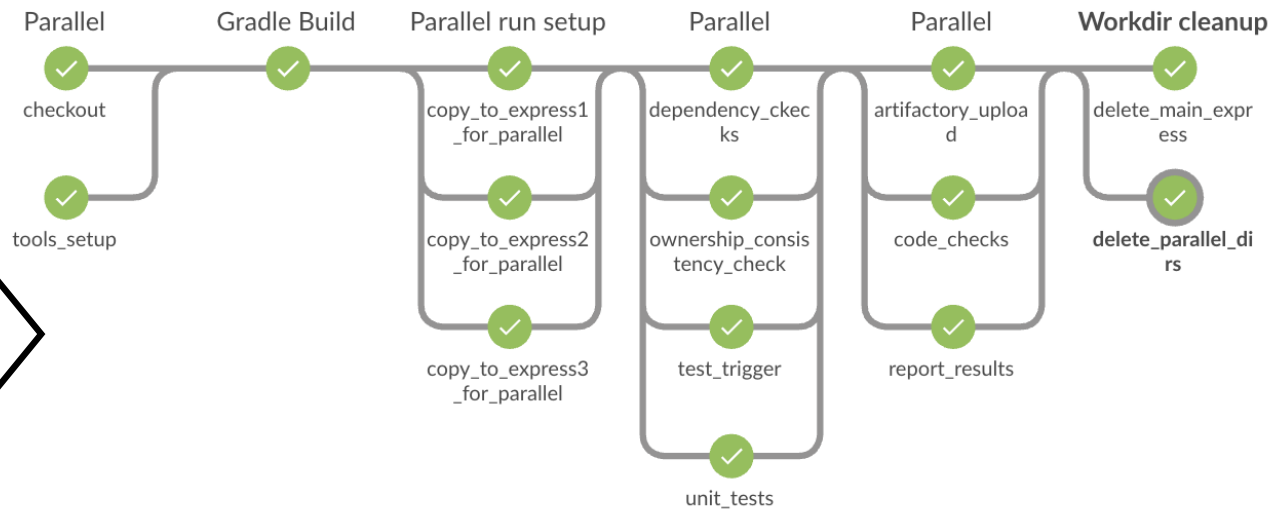


하나의 데이터 처리 단계의 출력이 다음 단계의 입력으로 이어지는 형태로 연결된 구조



고객의 복잡한 CI/CD 요구사항

소스체크아웃
도구 설치 (빌드를 위한 셋업)
소스 빌드
Express 1 설치
Express 2 설치
Express 3 설치
Dependency 검사
Consistency 검사
Test 수행
Unit Test 수행
Jar 업로드
코드 검사
Report 작성
Main Express 제거
디렉토리 제거



소스 저장소, 혹은 admin web에 작성된 스크립트를 통해 파이프라인의 흐름을 정의하는 기능

VS

```
Pipeline{
 agent none
 stages {
    stage('Build') {
      agent any
      steps {
        checkout scm
        sh 'make'
        stash includes: '**/target/*.jar', name: 'app'
       }
    }
    stage('Test on Linux') {
        agent {
            label 'linux'
        }
        steps {
            unstash 'app'
            sh 'make check'
        }
    }
```

Oops !

A problem occurred while processing the request.
Please check our bug tracker to see if a similar problem has already been reported.
If it is already reported, please vote and put a comment on it to let us gauge the impact of the problem.
If you think this is a new issue, please file a new issue.
When you file an issue, make sure to add the entire stack trace, along with the version of Jenkins and relevant plugins.
The users list might be also useful in understanding what has happened.

Stack trace

```
java.io.FileNotFoundException: /jenkins_data/jenkins/jobs/QA_SparkPerfTest/builds/2014-09-22_19-15-15/log (Permission denied)
        at java.io.RandomAccessFile.open(Native Method)
        at java.io.RandomAccessFile.<init>(RandomAccessFile.java:241)
        at org.kohsuke.stapler.framework.io.LargeText$FileSession.<init>(LargeText.java:397)
        at org.kohsuke.stapler.framework.io.LargeText$2.open(LargeText.java:120)
        at org.kohsuke.stapler.framework.io.LargeText.writeLogTo(LargeText.java:210)
        at hudson.console.AnnotatedLargeText.writeHtmlTo(AnnotatedLargeText.java:169)
        at hudson.console.AnnotatedLargeText.writeLogTo(AnnotatedLargeText.java:143)
        at org.kohsuke.stapler.framework.io.LargeText.doProgressText(LargeText.java:262)
        at hudson.console.AnnotatedLargeText.doProgressiveHtml(AnnotatedLargeText.java:91)
        at sun.reflect.GeneratedMethodAccessor482.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:606)
        at org.kohsuke.stapler.Function$InstanceFunction.invoke(Function.java:298)
        at org.kohsuke.stapler.Function.bindAndInvoke(Function.java:161)
        at org.kohsuke.stapler.Function.bindAndInvokeAndServeResponse(Function.java:96)
        at org.kohsuke.stapler.MetaClass$1.doDispatch(MetaClass.java:120)
        at org.kohsuke.stapler.NameBasedDispatcher.dispatch(NameBasedDispatcher.java:53)
        at org.kohsuke.stapler.Stapler.tryInvoke(Stapler.java:728)
```

VS

	build	test: integration-&-quality	test: functional	test: load-&-security	approval	deploy: prod
Average stage times: (Average full run time: ~5s)	836ms	20min 43s	9ms	7ms	89ms	5ms
#17 Sep 22 15:05 No Changes Retry Download	538ms	10s	10ms	8ms	72ms (paused for 7s)	4ms
#16 Sep 22 15:04 No Changes Retry Download	479ms	6s	9ms	9ms	74ms (paused for 6s)	5ms
#15 Sep 22 15:03 No Changes Retry Download	922ms	6s	10ms	9ms failed		

소스 저장소, 혹은 admin web에 작성된 스크립트를 통해 파이프라인의 흐름을 정의하는 기능

```
pipeline {
    agent any
    tools {
    }
    environment {
    }
    stages {
        stage ('Initialize') {
            steps { }
        stage ('Build') {
            steps {
                sh 'mvn -Dmaven.test.failure.ignore=true install'
            post {
                success {
                    junit 'target/surefire-reports/**/*.xml'
        stage ('Docker Build') {
            steps {
            }
        stage ('Kubernetes Deploy') {
            steps {
```

파이프라인 선언 및 도구, 파라미터 등 기술

파이프라인 시작

파이프라인 단계 구분

파이프라인 단계별 실행 스탭

소스 저장소, 혹은 admin web에 작성된 스크립트를 통해 파이프라인의 흐름을 정의하는 기능

```
pipeline {
    agent any
    tools {
        maven 'mvn2.6.0'  → maven build 버전
        jdk 'jdk1.8'→ jdk build 버전
    }
    environment {
        IMAGE_NAME = https://dpks.com/serviceabc:2  → 어플리케이션 이미지
        REPO_URL= https://harbor.io/jenkins  → 이미지 저장소
        JAR_NAME = "rest-service-0.1.0.jar"→  Application Jar 이름
    }
```

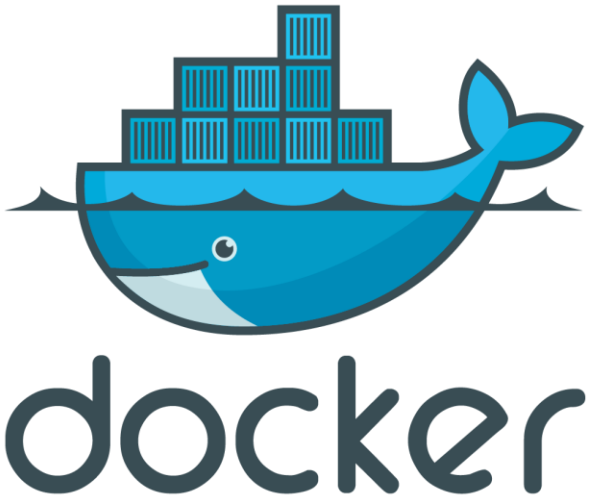
```
stages {
    stage ('Initialize') {
        steps {
            sh '''
                echo "PATH = ${PATH}"
                echo "M2_HOME = ${M2_HOME}"
            '''          }      }
    stage ('Build') {
        steps {
            sh 'mvn -Dmaven.test.failure.ignore=true install'          } -> Build
        post {
            success {
                junit 'target/surefire-reports/**/*.xml' -> Test Automation
            }          }      }
    stage ('Docker Build') {
        steps {
                sh '''
                    cp "${WORKSPACE}/target/${JAR_NAME}" "${WORKSPACE}/"
                    docker build -t $IMAGE_NAME . --build-arg JAR_FILE=$JAR_NAME
                    docker login
                    docker push $IMAGE_NAME
                '''                }          }
```

```
stage ('Kubernetes Deploy') {
        steps {
            withCredentials([certificate(aliasVariable:  '', credentialsId: 'cicd_user',
            keystoreVariable: 'CICD_USER_CERT',
            passwordVariable:  'CICD_USER_CERT_PASSWORD')]) {
                sh '''
                    kubectl apply –f xxx.yaml -> Deploy Automation
                '''
             }              }          }    }}
```

Docker Image를 만들기 위한 설정 파일

```
FROM ubuntu:14.04
RUN mkdir -p /app
WORKDIR /app
ADD     . /app
RUN apt-get update
RUN apt-get install apache2
RUN service apache2 start
VOLUME ["/data", "/var/log/httpd"]
EXPOSE 80
CMD ["/app/log.backup.sh"]
```

docker build –t 이미지명:태그



Docker Image를 만들기 위한 설정 파일

```
FROM ubuntu:14.04

RUN mkdir -p /app

WORKDIR /app
```

베이스 이미지 레이어
(<이미지 이름>:<태그>)

도커이미지가 생성되기 전에 수행할 쉘 명령어

CMD에서 설정한 실행 파일이 실행될 디렉토리

Docker Image를 만들기 위한 설정 파일

```
ADD     . /app                          # 파일을 이미지에 추가
RUN apt-get update
RUN apt-get install apache2             # 도커이미지가 생성되기 전에 수행할 쉘 명령어
RUN service apache2 start
```

Docker Image를 만들기 위한 설정 파일

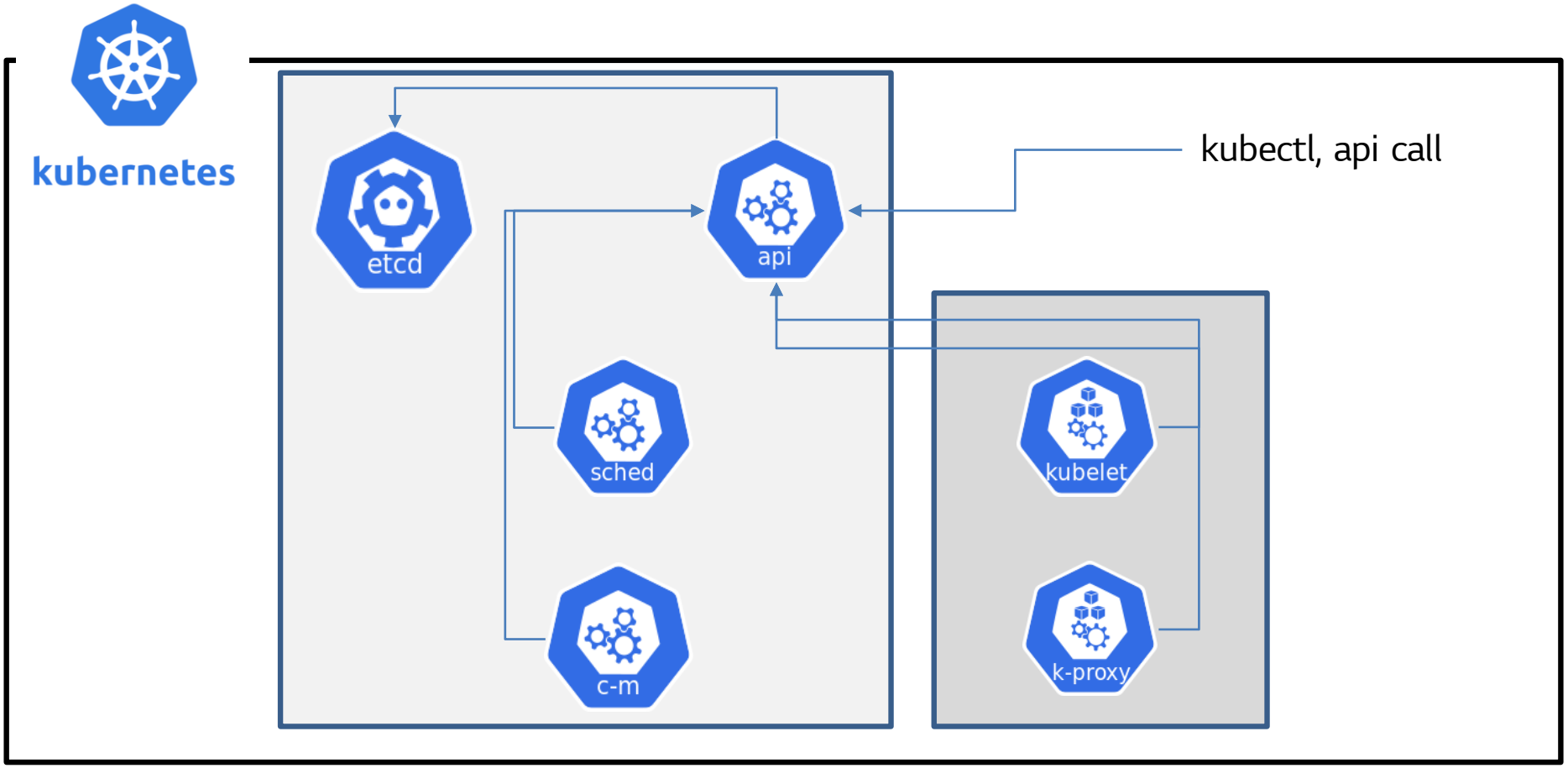
```
VOLUME ["/data", "/var/log/httpd"]
EXPOSE 80
CMD ["/app/log.backup.sh"]
```

디렉토리의 내용을 컨테이너에 저장하지 않고 호스트에 저장하도록 설정
(볼륨을 추가)

외부로 노출하는 포트 정보

컨테이너 시작 시 실행하는 파일 혹은 쉘 스크립트

컨테이너화된 애플리케이션을 자동으로 배포, 스케일링 및 관리해주는 오픈소스 시스템



쿠버네티스에서 오브젝트를 생성하기 위해 기본적인 정보와 상태를 기술한 오브젝트 spec을 기술한 파일

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

쿠버네티스에서 오브젝트를 생성하기 위해 기본적인 정보와 상태를 기술한 오브젝트 spec을 기술한 파일

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
```

쿠버네티스에서 적용되는 API 버전

어떤 종류의 오브젝트를 생성하는지 (Service Pod ConfigMap Ingress 등)

이름 문자열, UID, 네임스페이스를 포함하여 오브젝트를 구별하게 해주는 metadata

쿠버네티스에서 적용되는 오브젝트 내용

쿠버네티스에서 오브젝트를 생성하기 위해 기본적인 정보와 상태를 기술한 오브젝트 spec을 기술한 파일

```
spec:

  selector:
    matchLabels:
      app: nginx
  replicas: 2
```

쿠버네티스에서 관리하는 Selector의 내용과 일치하는 리소스

컨테이너 복제본의 수 (상시 유지)

쿠버네티스에서 오브젝트를 생성하기 위해 기본적인 정보와 상태를 기술한 오브젝트 spec을 기술한 파일

```
spec:

  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Pod 생성 정보 (도커 이미지, Port, 라벨 등)