CI/CD with Kubernetes

이 정 구
itgenius1004@gmail.com

목차

쿠버네티스와 GitOps

- 쿠버네티스 소개
- 선언적 vs 명령형 개체 관리
- 컨트롤러 아키텍처
- CI/CD 기본 개념 및 GitOps 도입
- 기본 GitOps 연산자
- 지속적인 통합 파이프 라인

쿠버네티스 소개

- 왜 쿠버네티스인가?
  - 현대의 소프트웨어는 다음과 같은 요구사항
    - 빠르고 빈번한 배포
    - 서비스 무중단 운영
    - 수평 확장
    - 자동 복구
  - 이러한 요구를 만족시키기 위해, 컨테이너 (ex. Docker)를 사용하여 애플리케이션을 생성하고, 쿠버네티스가 이 컨테이너들의 생명주기를 관리

쿠버네티스 소개

- 쿠버네티스의 주요 기능

기능	설명
자동 배포 및 롤백	애플리케이션을 자동으로 배포하고 문제가 생기면 이전 버전으로 롤백
서비스 디스커버리 및 로드 밸런싱	컨테이너 간 통신을 위한 네트워크와 부하 분산
자동 스케일링	CPU/메모리 사용량 또는 커스텀 지표 기반으로 Pod 수 조절
자가 치유(Self-healing)	실패한 컨테이너를 자동 재시작하거나 교체
구성 및 보안 관리	ConfigMap, Secret으로 설정과 민감 정보를 분리 관리

쿠버네티스 소개

- 쿠버네티스의 핵심 구성 요소

리소스	설명
Pod	쿠버네티스에서 실행되는 기본 단위 (1개 이상의 컨테이너 포함 가능)
Deployment	Pod의 선언적 관리, 롤링 업데이트 지원
Service	외부 또는 내부에서 Pod에 접근 가능하게 해주는 엔드포인트
ConfigMap / Secret	애플리케이션의 설정 정보 및 비밀값 분리 관리
Node	컨테이너가 실제로 실행되는 서버 (VM 혹은 물리 서버)
Cluster	Node들의 집합으로 쿠버네티스 전체를 의미

쿠버네티스 소개

- 전체 동작 흐름 예시

  개발자 Git에 코드 push

  ↓

  CI 도구 (예: Jenkins, GitHub Actions) 빌드 및 컨테이너 이미지 생성

  ↓

  CD 도구 (예: Argo CD) → 쿠버네티스에 배포

  ↓

  쿠버네티스가 Pod 생성 및 운영

쿠버네티스 소개

- 쿠버네티스의 장점 요약
  - 인프라와 애플리케이션 분리
  - 멀티 클라우드, 하이브리드 환경 지원
  - 자동화로 운영 부담 감소
  - 클라우드 네이티브 애플리케이션에 최적

선언적 vs 명령형 개체 관리

- 선언형(Declarative) 관리 방식
  - 최종 상태를 선언하고, 쿠버네티스가 자동으로 그 상태로 맞춤
  - 원리: 원하는 리소스의 상태를 YAML 파일에 정의
  - 실행: kubectl apply –f deployment.yaml
  - 특징
    - GitOps에 적합 (버전 관리 가능)
    - 인프라의 상태를 코드로 관리 (IaC)
    - 쿠버네티스가 현재 상태와 비교 후, 필요한 변경만 적용

선언적 vs 명령형 개체 관리

- 장점
  - 자동화, 추적성, 협업에 유리
  - 인프라 변경 이력 관리 가능

선언적 vs 명령형 개체 관리

- 명령형(Imperative) 관리 방식
  - 직접 지시해서 리소스를 생성하거나 수정
  - 원리: 명령어를 통해 리소스를 즉시 생성, 수정
  - 실행: kubectl run, kubectl create, kubectl delete, kubectl scale 등
  - 특징
    - 빠르고 직관적
    - 실험, 임시 리소스 테스트에 적합

선언적 vs 명령형 개체 관리

- 단점
  - 히스토리 관리 불가 (코드화 어려움)
  - 협업이나 자동화엔 부적합

선언적 vs 명령형 개체 관리

- 비교 요약

항목	선언형(Declarative)	명령형(Imperative)
방식	원하는 최종 상태를 기술	명령으로 직접 제어
도구	kubectl apply, kustomize, Argo CD	kubectl run/create/delete/scale
자동화	매우 용이	제한적
협업/버전관리	YAML을 Git에 저장	불가
학습 곡선	다소 있음	비교적 쉬움
사용 사례	실서비스 배포, GitOps	개발 초기 테스트, 실험

선언적 vs 명령형 개체 관리

- 실서비스 운영 환경: 선언형이 권장
- 개발/실험 용도: 빠르게 적용 가능한 명령형도 유용

컨트롤러 아키텍처

- 컨트롤러 아키텍처란?
    - 쿠버네티스의 Control Loop 개념을 기반으로 동작하는, 리소스의 실제 상태를 원하는 상태(Desired State)로 유지하도록 조정하는 구성 요소

컨트롤러 아키텍처

- 핵심 개념: Control Loop (제어 루프)
  - 쿠버네티스는 다음과 같은 루프 구조로 작동

    [사용자 또는 선언형 파일] → 원하는 상태 선언

    ↓

    [컨트롤러] ←→ [현재 상태 관찰]

    ↓

    [차이 발견 시] → 조정 작업 수행

  - 즉, 쿠버네티스는 현재 상태(Observed State)와 원하는 상태(Desired State)를 비교해서, 불일치할 경우 자동으로 조치

컨트롤러 아키텍처

- 주요 컨트롤러 예시

컨트롤러	설명
Deployment Controller	Deployment 리소스를 감시하고, 원하는 수의 Pod를 유지
ReplicaSet Controller	정확한 수의 복제본을 유지
Node Controller	클러스터의 노드 상태 감시 및 반응 (예: 다운된 노드 처리)
Job Controller	일회성 작업(Pod)이 성공할 때까지 관리
StatefulSet Controller	순서와 안정된 네트워크 ID가 필요한 상태 저장 애플리케이션 관리
DaemonSet Controller	모든 노드에 하나씩 Pod를 유지

컨트롤러 아키텍처

- 내부 동작 흐름
    1. 사용자가 kubectl apply로 리소스를 생성하거나 수정
    2. etcd에 원하는 상태 저장
    3. 해당 리소스를 감시하는 컨트롤러가 현재 상태를 감시
    4. 상태 불일치가 감지되면 API 서버를 통해 변경 요청
    5. 스케줄러 및 kubelet이 실제 작업(Pod 생성 등) 수행

컨트롤러 아키텍처

- 구조도

  사용자

  ↓

  API Server ←→ etcd (Desired State 저장)

  ↓

  컨트롤러 ←→ 현재 상태 감시 (리소스 Watch)

  ↓

  필요 시 작업 실행 (ex. Pod 재시작, 생성, 삭제 등)

컨트롤러 아키텍처

- 컨트롤러 아키텍처의 장점
  - 자가 치유: Pod가 죽으면 자동 복구
  - 자동화: 선언만 하면 컨트롤러가 알아서 유지
  - 확장성: 컨트롤러는 비동기적으로 병렬 수행 가능
  - 플러그인 구조: Custom Controller (Operator)로 기능 확장 가능

컨트롤러 아키텍처

- Custom Controller & Operator
  - 개발자가 직접 컨트롤러를 만들어 도메인 특화 리소스(Custom Resource)를 자동화할 수 있음
  - 이를 통해 쿠버네티스를 "플랫폼 오퍼레이터"처럼 활용 가능

CI/CD 기본 개념 및 GitOps 도입

- CI/CD 기본 개념
  - CI (Continuous Integration, 지속적 통합)
    - 개념: 개발자가 코드 변경을 자주(하루에도 여러 번) 중앙 저장소에 통합
    - 목적: 자동 빌드 & 테스트 → 빠른 피드백 & 문제 조기 발견
    - 예시 도구: GitHub Actions, Jenkins, GitLab CI, CircleCI 등

CI/CD 기본 개념 및 GitOps 도입

- CD (Continuous Delivery / Continuous Deployment)

Continuous Delivery	Continuous Deployment
자동 배포 준비까지 수행	자동으로 실제 환경까지 배포
운영자가 승인 후 배포	승인 없이 자동 배포
staging까지 자동화	production까지 자동화

  - 공통점: 배포 자동화로 인한 안정성 및 속도 향상
  - 차이점: 최종 배포의 자동 여부

CI/CD 기본 개념 및 GitOps 도입

- 기존 CI/CD 파이프라인 흐름 (쿠버네티스 없이)

  개발자 → Git Push

  ↓

  CI 서버: 빌드 + 테스트

  ↓

  CD 서버: Docker 이미지 생성 & 레지스트리에 푸시

  ↓

  kubectl 또는 Helm 등으로 수동 배포

CI/CD 기본 개념 및 GitOps 도입

- 문제점
  - 운영환경 접근 권한이 CI/CD 서버에 있음 → 보안 이슈
  - 쿠버네티스 YAML이 흩어져 있음 → 관리 어려움
  - 배포 과정이 선언적이지 않음 → 상태 불일치 발생

CI/CD 기본 개념 및 GitOps 도입

- GitOps란?
  - Git을 단일 진실의 출처(Single Source of Truth)로 삼아 쿠버네티스 리소스 상태를 Git과 자동으로 동기화하는 방식의 CD 운영 철학
  - 핵심 아이디어
    - 모든 인프라/애플리케이션 구성은 Git에 선언형으로 저장
    - Git 변경이 곧 배포를 의미
    - 쿠버네티스 클러스터 내부에서 변경 사항을 Pull해서 적용 (Push 아님)

CI/CD 기본 개념 및 GitOps 도입

- GitOps 구조

  Git 저장소 ← (개발자가 YAML 수정/Push)

  ↑

  [GitOps CD 툴 (ex. Argo CD, Flux)]

  ↓

  쿠버네티스 클러스터 상태를 Git과 동기화

CI/CD 기본 개념 및 GitOps 도입

- GitOps 도입의 장점

장점	설명
보안 강화	배포 권한이 Git 저장소에만 있음 (클러스터 내부에서 동작)
가시성 향상	운영 상태를 Git에서 직접 확인 가능
버전관리 가능	리소스 변경 이력 관리 및 롤백 용이
일관성 유지	쿠버네티스 상태 = Git 상태

CI/CD 기본 개념 및 GitOps 도입

- GitOps 도구

도구	특징
Argo CD	쿠버네티스 전용 GitOps 도구, 시각화 UI 제공
Flux CD	CNCF 공식 프로젝트, GitOps 자동화에 강점
Jenkins X	CI/CD + GitOps 통합형 도구

CI/CD 기본 개념 및 GitOps 도입

- GitOps 도입 단계 요약
  - 애플리케이션 선언 파일(YAML)을 Git에 버전관리
  - GitOps 도구(Argo CD 등) 설치 및 연결
  - Git 저장소 → 쿠버네티스 클러스터 자동 동기화 구성
  - 코드 변경 후 Git Push → 자동 배포 수행

CI/CD 기본 개념 및 GitOps 도입

- CI는 빌드 및 테스트 자동화
- CD는 배포 자동화
- GitOps는 선언형 Git 기반 CD의 구현 전략

기본 GitOps 연산자

- 기본 GitOps 연산자란?
  - “Git 저장소 ←→ 쿠버네티스 클러스터”의 자동 동기화 중심에 있는 핵심 컴포넌트
  - GitOps 연산자는 보통 다음 역할을 수행
    1. Git 저장소 감시 (폴링 or 웹훅)
    2. 변경 감지 시 선언형 YAML 파일 파싱
    3. 쿠버네티스 API 서버를 호출하여 리소스 생성/수정
    4. 현재 상태와 원하는 상태가 다를 경우 조정
    5. 상태를 UI 또는 이벤트로 보고

기본 GitOps 연산자

- 대표적인 GitOps 연산자
  - Argo CD
    - CNCF 인큐베이티드 GitOps 프로젝트
    - UI, CLI, API 지원
    - Git 저장소의 선언형 리소스를 클러스터에 자동 적용
    - 기반 구조:
      - argocd-repo-server: Git에서 YAML 파싱
      - argocd-application-controller: 상태 동기화 수행
      - argocd-server: UI 제공

기본 GitOps 연산자

- Flux CD
  - CNCF 공식 프로젝트로, GitOps 철학의 초기 주창자
  - 모듈 기반 구조
    - Source Controller: Git/Helm source 감시
    - Kustomize/Helm Controller: 리소스 적용
    - Notification Controller: 슬랙/웹훅 연동

기본 GitOps 연산자

- GitOps 연산자 주요 기능 비교

기능	Argo CD	Flux
UI	있음 (웹 대시보드)	없음 (외부 툴 필요)
멀티앱 관리	매우 용이	모듈 조합 필요
선언형 관리	Application CRD	GitRepository, Kustomization 등 CRD
멀티 클러스터	쉬움 (클러스터 등록)	가능 (Flux Multi-tenancy 구성 필요)
커스터마이징	Helm/Kustomize/Plain YAML 모두 지원	동일
사용성	입문자 친화적	구성 유연성 높음

기본 GitOps 연산자

- 동작 방식

  [Git 저장소]

  ↓ (변경 감지)

  [GitOps 연산자]

  ↓

  [쿠버네티스 API 호출]

  ↓

  [리소스 상태 자동 적용]

  - 연산자는 Pull 방식을 사용해 보안을 강화
  - 즉, CI 시스템이 직접 클러스터를 제어하지 않고, 연산자가 Git을 바라보고 직접 변경을 수행

기본 GitOps 연산자

- 요약

항목	설명
정의	Git 상태를 기준으로 쿠버네티스를 자동 조정하는 컨트롤러
대표 도구	Argo CD, Flux CD
핵심 역할	Git 감시 → 리소스 적용 → 상태 일치 유지
연산자 장점	자동화, 보안, 추적성, 롤백 가능성
GitOps 구현 필수 요소	Git 저장소 + 선언형 리소스 + GitOps 연산자

지속적인 통합 파이프 라인

- 지속적인 통합(CI, Continuous Integration) 파이프라인의 핵심 목적
  - "코드 변경 사항을 빠르고 안전하게 검증하여 메인 브랜치에 자주 통합할 수 있도록 한다."

지속적인 통합 파이프 라인

- CI 파이프라인 기본 구조
  - CI 파이프라인은 일반적으로 다음 단계들로 구성

    [1] 코드 푸시 (Push to Git)

    ↓

    [2] 의존성 설치 (Install dependencies)

    ↓

    [3] 정적 분석 (Lint, Code Quality 체크)

    ↓

지속적인 통합 파이프 라인

[4] 빌드 (Build)

↓

[5] 테스트 (단위 테스트, 통합 테스트 등)

↓

[6] 테스트 결과 보고 / 알림

지속적인 통합 파이프 라인

- 단계별 설명

단계	설명
1. 소스 코드 변경 감지	Git push, PR 생성 등 트리거로 자동 시작
2. 의존성 설치	예: npm install, pip install, mvn install 등
3. 정적 분석	코드 스타일, 보안 스캔 도구 (예: ESLint, SonarQube)
4. 빌드	실행 가능한 산출물 생성 (Docker image, JAR 등)
5. 테스트 실행	단위 테스트, 통합 테스트, 테스트 커버리지 측정
6. 결과 리포트 및 알림	Slack, Email, GitHub PR 상태 체크 등

지속적인 통합 파이프 라인

- CI 도구 종류

도구	특징
GitHub Actions	GitHub에 통합, 설정 간편
Jenkins	오픈소스, 유연성 최고 (플러그인 기반)
GitLab CI/CD	GitLab과 통합된 CI/CD 플랫폼
CircleCI	클라우드 기반, 빠른 빌드 속도
Travis CI	GitHub 연동 중심, 설정 간단

지속적인 통합 파이프 라인

- CI 파이프라인 구성 시 유의 사항
  - 작고 빠르게 유지: 전체 테스트가 5~10분 내 완료되도록 구성
  - Fail Fast: 문제 발생 시 즉시 실패 처리
  - 자동화된 피드백: 결과를 팀에게 즉시 알림
  - 분기마다 병합 전 실행: Pull Request에도 자동 실행

지속적인 통합 파이프 라인

- CI → CD 확장 흐름

  [CI 완료]

  ↓

  [Docker 이미지 생성 및 푸시]

  ↓

  [CD 파이프라인 (예: GitOps 연산자)]

  ↓

  [쿠버네티스에 자동 배포]

지속적인 통합 파이프 라인

CI 파이프라인은 다음을 가능하게 함

- 빠른 코드 통합
- 자동화된 빌드 & 테스트
- 코드 품질 유지
- 배포 전 안정성 확보

개발 환경 설정 및 구성

- 기술 스택 및 툴 소개
- 개발 환경 구성

기술 스택 및 툴 소개

- CI/CD 및 GitOps 기반 쿠버네티스 운영을 위한 기술 스택과 도구들을 아래와 같이 카테고리별로 정리하면 다음과 같음
- DevOps 파이프라인에서 일반적으로 사용되는 도구들임

기술 스택 및 툴 소개

- 전체 아키텍처 구성 개요

  [개발자]

  ↓ (코드 Push)

  [Git 저장소] ←→ [CI 도구]

  ↓ (빌드/테스트)

  [Docker 이미지 생성 → Registry 푸시]

  ↓

  [CD 도구 (GitOps 연산자)]

  ↓

  [Kubernetes 클러스터에 배포/운영]

기술 스택 및 툴 소개

- 소스 코드 및 버전 관리

도구	설명
Git	분산 버전 관리 시스템
GitHub / GitLab / Bitbucket	Git 기반 코드 저장소 및 협업 도구
Branch 전략	Git Flow, Trunk-Based 등

기술 스택 및 툴 소개

- CI (지속적 통합) 도구

도구	특징
GitHub Actions	GitHub 내장, 워크플로우 기반
GitLab CI	GitLab과 통합된 강력한 CI/CD
Jenkins	오픈소스 CI/CD의 대표주자
CircleCI / TravisCI	클라우드 기반, 빠른 빌드 속도

기술 스택 및 툴 소개

- 빌드 및 패키징

도구	설명
Docker	컨테이너 이미지 생성 및 실행
BuildKit / Kaniko	클러스터 내에서 Dockerless 빌드
Jib	Java용 Dockerfile 없이 이미지 생성 (Spring Boot 등에 적합)

기술 스택 및 툴 소개

- 이미지 저장소 (Container Registry)

도구	설명
Docker Hub	퍼블릭/프라이빗 레지스트리
GitHub Container Registry (GHCR)	GitHub 연동
Harbor	자체 구축 가능한 레지스트리
Google Artifact Registry / Amazon ECR	클라우드 연동

기술 스택 및 툴 소개

- CD (지속적 배포) 및 GitOps 연산자

도구	특징
Argo CD	쿠버네티스 네이티브 GitOps 도구 (UI 지원)
Flux CD	경량, GitOps 자동화 특화
Helm	쿠버네티스 패키지 관리자, 복잡한 배포 구성
Kustomize	YAML 오버레이 관리용 툴

기술 스택 및 툴 소개

- 운영 환경: Kubernetes 클러스터

구성 요소	설명
Kubernetes	컨테이너 오케스트레이션 플랫폼
Kubelet / API Server / Scheduler	클러스터 핵심 컴포넌트
Namespace / Pod / Deployment / Service	기본 리소스 객체
Ingress / LoadBalancer	외부 트래픽 라우팅

  - 실습용: minikube, kind, k3s
  - 실서비스용: GKE (Google), EKS (AWS), AKS (Azure)

기술 스택 및 툴 소개

- 모니터링 및 로깅

도구	기능
Prometheus + Grafana	메트릭 수집 및 대시보드 시각화
Loki / Fluent Bit / Elasticsearch	로그 수집 및 검색
Kiali + Istio	서비스 메시 모니터링

기술 스택 및 툴 소개

- 보안 및 관리 도구

도구	설명
Vault / Sealed Secrets	민감 정보 관리
RBAC	쿠버네티스 역할 기반 접근 제어
OPA / Kyverno	정책 기반 리소스 제어

기술 스택 및 툴 소개

- 개발 편의 도구

도구	용도
VSCode + Kubernetes Plugin	코드 & 클러스터 리소스 확인
kubectl / k9s	쿠버네티스 CLI 및 대화형 UI
Skaffold / Tilt	로컬 개발에서 클러스터 연동 자동화

기술 스택 및 툴 소개

- 추천 기술 스택

구성 요소	추천 도구
Git 저장소	GitHub
CI	GitHub Actions
빌드	Docker
이미지 저장소	Docker Hub
CD	Argo CD
클러스터	minikube (로컬), AKS or EKS or GKE (클라우드)
모니터링	Prometheus + Grafana

개발 환경 구성

- 기본 개발 장비 및 툴

항목	도구 / 설명
운영체제	Windows, macOS, Linux (개발자 OS)
IDE/에디터	VS Code, IntelliJ, PyCharm 등
터미널 툴	iTerm2 (mac), Windows Terminal, Zsh, Oh My Zsh
Git 클라이언트	Git CLI, GitHub Desktop, GitKraken 등
컨테이너 엔진	Docker Desktop 또는 Podman
패키지 매니저	Homebrew (mac), Chocolatey (Windows)

개발 환경 구성

- 로컬 쿠버네티스 환경 구성

도구	설명
Minikube	VM 또는 Docker 기반의 로컬 K8s 클러스터
Kind (Kubernetes in Docker)	Docker 기반의 lightweight 로컬 K8s
k3d	빠르고 가벼운 K3s 기반 클러스터, CI 테스트에 적합
kubectl	쿠버네티스 리소스 CLI (필수)
k9s	CLI 기반 쿠버네티스 리소스 브라우저 (권장)

  - Docker Desktop은 내장된 K8s 기능도 제공함

개발 환경 구성

- Git 저장소 구성
    - GitHub, GitLab, Bitbucket 등 선택

개발 환경 구성

- GitOps를 위한 구조

```
├── apps/
│   └── my-app/
│       ├── base/
│       │   └── deployment.yaml
│       └── overlays/
│           ├── dev/
│           └── prod/
└── manifests/
    ├── ingress.yaml
    └── namespace.yaml
```

개발 환경 구성

- Git 저장소를 코드 저장용 레포 + 배포 선언용 레포로 분리하는 것도 좋은 전략

개발 환경 구성

- CI/CD 환경 구성 (로컬 or 클라우드 기반)

항목	도구 / 설명
CI	GitHub Actions, GitLab CI, Jenkins 등
이미지 빌드	Docker, Kaniko, BuildKit 등
이미지 저장소	Docker Hub, GHCR, Harbor, ECR 등
CD / GitOps	Argo CD or Flux 설치 (쿠버네티스 내부에 배포)
Helm / Kustomize	리소스 템플릿 관리 도구

개발 환경 구성

- GitOps 연산자 설치 (ex. Argo CD)

  kubectl create namespace argocd

  kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

  - 설치 후 UI 접속: kubectl port-forward svc/argocd-server -n argocd 8080:443
  - 초기 로그인 후 Git 저장소 연결

개발 환경 구성

- 테스트 및 배포 환경

구성 요소	예시 도구 / 플랫폼
Dev 클러스터	로컬 (minikube, kind)
Staging / Prod 클러스터	EKS, GKE, AKS, 또는 Kubeadm 설치형
네임스페이스 기반 환경 분리	dev, staging, prod
도메인/인그레스	Ingress Controller + local DNS (nip.io, sslip.io)

개발 환경 구성

- 모니터링/디버깅 도구(옵션)

도구	설명
Prometheus + Grafana	메트릭 수집 및 시각화
Loki	로그 수집 및 탐색
Skaffold	로컬 개발 → 쿠버네티스에 빠르게 반복 배포
Tilt	개발자 친화적인 쿠버네티스 배포 자동화 도구

개발 환경 구성

- 환경 구성 요약

  [VS Code / IDE]

  ↓

  [Git 저장소 ←→ CI 파이프라인]

  ↓

  [Docker 이미지 빌드 & Push]

  ↓

  [CD 도구 (Argo CD, Flux)]

  ↓

개발 환경 구성

[로컬 K8s or 클라우드 K8s 클러스터]

↓

[모니터링 / 로깅 / 디버깅 도구]

개발 환경 구성

- 체크리스트
  - Git 저장소 구성 (YAML 관리 포함)
  - 컨테이너 이미지 빌드 및 레지스트리 연동
  - 로컬 쿠버네티스 환경 구성 및 테스트 배포
  - CI 자동화 (PR 빌드 + 테스트)
  - CD 자동화 (GitOps 기반 배포)
  - Argo CD or Flux 설치 및 Git 연동
  - 간단한 애플리케이션 배포 실습

환경 관리

- Git 전략 및 브랜치 운영
- 설정 관리

Git 전략 및 브랜치 운영

- Git 브랜치 전략이 왜 중요한가?
  - 코드 충돌 최소화
  - 배포 시점 관리 용이
  - 협업 구조의 명확화
  - 자동화(CI/CD) 연계 기반

Git 전략 및 브랜치 운영

- 주요 Git 브랜치 전략

전략명	설명	특징
Git Flow	기능 개발/릴리즈/핫픽스를 명확히 분리	브랜치 다양, 구조 명확함
GitHub Flow	간단한 기능 브랜치 + PR + 배포	단순하고 빠름
GitLab Flow	Git Flow + 환경 기반 브랜치	배포 환경 분리 가능
Trunk-Based Development	main/master 한 줄 개발 + Feature flag	린/CI/CD 지향

Git 전략 및 브랜치 운영

- 주요 Git 브랜치 전략

전략명	설명	특징
Git Flow	기능 개발/릴리즈/핫픽스를 명확히 분리	브랜치 다양, 구조 명확함
GitHub Flow	간단한 기능 브랜치 + PR + 배포	단순하고 빠름
GitLab Flow	Git Flow + 환경 기반 브랜치	배포 환경 분리 가능
Trunk-Based Development	main/master 한 줄 개발 + Feature flag	린/CI/CD 지향

Git 전략 및 브랜치 운영

- Git Flow 전략 (전통적, 팀 개발 지향)

```
main ← stable 배포 코드
  |
  ├── develop ← 개발 브랜치
  |     ├── feature/로그인
  |     └── feature/장바구니
  |
  ├── release/1.0 ← 배포 준비
  |
  └── hotfix/1.0.1 ← 긴급 수정
```

Git 전략 및 브랜치 운영

브랜치	용도
main	실제 운영 배포용 코드
develop	기능 개발 통합 브랜치
feature/*	단일 기능 개발 브랜치
release/*	릴리즈 준비 브랜치
hotfix/*	운영 이슈 수정 브랜치

- 단점: 브랜치가 많고 복잡, 소규모 팀에는 과함

Git 전략 및 브랜치 운영

- GitHub Flow 전략 (스타트업/애자일에 적합)

  main ← 운영 브랜치

  │

  └─── feature/댓글작성

  ↑

  PR & 리뷰 → Merge & 배포

  - main 브랜치에서만 배포
  - 모든 변경은 feature → PR → Merge
  - 배포 자동화와 잘 맞음 (CI/CD와 연계)

Git 전략 및 브랜치 운영

- GitLab Flow 전략 (환경 기반 + GitOps 연계에 유리)

  main ← 운영

  staging ← 테스트 배포

  dev ← 개발 통합

  └── feature/*

  - Git Flow와 유사하나, 환경별 브랜치(dev, staging, prod) 운영
  - GitOps 방식의 자동 배포 연계에 유리
    - → dev → 자동 배포 to dev 환경
    - → staging → QA 테스트 후 운영 승격

Git 전략 및 브랜치 운영

- Trunk-Based Development (CD 특화)
  - 하나의 main 브랜치에서 모든 개발
  - Feature flag로 실서비스 제어
  - short-lived branch (1~2일 내 Merge)
  - 빠른 CI/CD 자동화와 잘 맞음
  - GitOps + TDD + Microservice 환경에서 많이 채택됨

Git 전략 및 브랜치 운영

- 브랜치 운영 정책 예

정책 항목	예시
브랜치 이름 규칙	feature/, bugfix/, hotfix/, release/
Merge 방식	Pull Request 후 Code Review & Squash Merge
CI 트리거	PR 생성/수정 시 자동 빌드 & 테스트
보호 브랜치 설정	main, release/* 브랜치는 강제 PR 병합만 허용
태그 정책	v1.0.0, v1.0.1-hotfix, v1.1.0-rc1

Git 전략 및 브랜치 운영

- 적합한 조합 예

조직/상황	추천 전략
소규모 스타트업, 빠른 배포	GitHub Flow + GitOps
전통 대기업, QA 팀 존재	Git Flow or GitLab Flow
마이크로서비스 환경	Trunk-Based + GitOps + Feature Flags
교육/실습 환경	GitHub Flow (간단하고 직관적)

Git 전략 및 브랜치 운영

- GitOps와 연계하는 브랜치 전략
  - main → 운영 환경(prod) 자동 배포
  - staging → 테스트 환경 자동 배포
  - dev → 개발 클러스터 자동 배포

  - 각 브랜치에 대응되는 Kubernetes 리소스가 Git에 저장되어 있어야 함
  - Argo CD나 Flux는 해당 브랜치를 감시하여 자동 동기화

Git 전략 및 브랜치 운영

항목	정리
Git 전략	GitHub Flow, Git Flow, GitLab Flow, Trunk-Based
운영 규칙	브랜치 규칙 + Merge 룰 + CI/CD 트리거
GitOps 연계	브랜치 = 배포 환경, 선언형 YAML 관리

Git 전략 및 브랜치 운영

- 설정 관리란?
    - 애플리케이션 코드와 설정값(환경 변수, DB 주소, API 키 등)을 분리하고, 이를 안전하고 선언적으로 관리하는 일련의 전략

Git 전략 및 브랜치 운영

- 설정이 왜 분리되어야 하는가?

이유	설명
환경별 설정 분리	dev/staging/prod 환경에 따라 설정값이 다름
보안 강화	민감 정보(API Key 등)를 코드에 포함시키면 위험
배포 유연성	코드 수정 없이 설정만 바꿔 다양한 환경 적용 가능
GitOps 호환	설정도 Git에서 버전 관리 및 추적 가능해야 함

Git 전략 및 브랜치 운영

- 쿠버네티스 설정 관리 리소스

리소스	설명
ConfigMap	설정값(환경 변수, 설정파일 등) 저장용
Secret	암호화된 민감정보 저장 (Base64 인코딩 사용)
EnvFrom / VolumeMount	Pod에 ConfigMap/Secret을 주입하는 방법

Git 전략 및 브랜치 운영

- GitOps와 설정 관리
  - ConfigMap/Secret도 Git에 선언형 YAML로 관리
  - GitOps 도구(Argo CD 등)가 이를 자동으로 반영
  - 민감 정보는 Git에 직접 저장하면 안 됨

    → Sealed Secrets 또는 External Secret 사용 권장

Git 전략 및 브랜치 운영

- 설정 구성 전략 (환경 분리)
  - 디렉토리 구조 예시 (Kustomize 활용)

```
configs/
├── base/
│   ├── configmap.yaml
│   └── secret.yaml
├── overlays/
│   ├── dev/
```

Git 전략 및 브랜치 운영

```
│   │   └── kustomization.yaml
│   ├── staging/
│   └── prod/
```

- Kustomize로 환경별 설정값을 오버레이 가능
- GitOps 도구가 각 환경 브랜치나 디렉토리를 감시하도록 구성

Git 전략 및 브랜치 운영

- 설정 관리 베스트 프랙티스

항목	권장 방식
환경별 설정 구분	dev, staging, prod 별 ConfigMap/Secret 분리
민감 정보 관리	Sealed Secrets, External Secrets 사용
Git 저장소 구조	설정도 코드처럼 버전 관리 (YAML)
환경 변수/볼륨 주입	Pod 정의에서 명확히 관리
선언형 관리	kubectl apply or GitOps 방식으로 유지

Git 전략 및 브랜치 운영

구분	설명
일반 설정	ConfigMap 사용 (환경 변수, 파일 등)
민감 정보	Secret, Sealed Secret, External Secret 사용
환경 분리	Kustomize 또는 디렉토리/브랜치 구조로 분리
GitOps 연계	설정도 Git 기반 선언형으로 관리해야 일관성 확보

파이프라인

- CI/CD 파이프라인 단계
- GitOps 기반 CI/CD 통합
- Driving Promotions
- Code vs Manifest vs App Config
- 다른 파이프라인 확인
- 롤백 및 규정 준수 파이프라인
- 배포 기본 사항 및 ReplicaSet 확인

CI/CD 파이프라인 단계

- CI/CD 파이프라인 전체 흐름

  [1] 코드 변경 (Git Push)

  ↓

  [2] CI: 빌드 & 테스트 자동화

  ↓

  [3] Docker 이미지 생성 & Registry에 저장

  ↓

  [4] CD: 쿠버네티스 배포 자동화 (GitOps or 직접 배포)

  ↓

  [5] 운영 환경 반영 & 모니터링

CI/CD 파이프라인 단계

단계	설명
1. 코드 커밋 & 푸시	개발자가 Git 저장소에 코드 변경 푸시
2. CI 트리거	GitHub Actions, Jenkins 등 CI 도구가 푸시 이벤트 감지
3. 의존성 설치 & 정적 분석	npm install, mvn install, lint, SonarQube 등
4. 유닛/통합 테스트	자동 테스트 수행, 실패 시 파이프라인 중단
5. 빌드 & Docker 이미지 생성	애플리케이션 빌드 후 Docker 이미지 생성
6. 이미지 레지스트리 푸시	Docker Hub, GitHub Container Registry 등
7. 매니페스트 준비 (YAML)	deployment.yaml, service.yaml 등 작성 또는 템플릿화
8. CD 단계 실행	GitOps 방식 (Argo CD, Flux) 또는 kubectl, Helm 등 직접 배포
9. 클러스터 반영 확인	Pod 상태 확인, 헬스체크, 롤백 가능성 확보
10. 모니터링 & 알림	Prometheus, Slack, Grafana, Sentry 연동 가능

CI/CD 파이프라인 단계

- CI/CD 도구 연계 흐름

구분	도구
CI	GitHub Actions, GitLab CI, Jenkins
이미지 빌드	Docker, BuildKit, Kaniko
이미지 저장소	Docker Hub, GHCR, ECR
CD	Argo CD (GitOps), Helm, kubectl
테스트	Jest, JUnit, Cypress, Pytest 등
모니터링	Prometheus, Grafana, Slack 알림

CI/CD 파이프라인 단계

- GitOps 기반 CD 구성 예시

  [CI] 코드 빌드 및 Docker 이미지 푸시

  ↓

  [Git 저장소에 YAML 수정 (버전 올림)]

  ↓

  [Argo CD가 Git 저장소 감시]

  ↓

  [쿠버네티스에 변경사항 자동 반영]

CI/CD 파이프라인 단계

- 추천 CI/CD 파이프라인
  - feature/* 브랜치 → CI 빌드 & 테스트 자동 수행
  - main 브랜치 머지 → Docker 이미지 빌드 & 레지스트리에 Push
  - manifests/main 브랜치 YAML 업데이트 → GitOps로 자동 배포
  - Argo CD가 클러스터 상태와 Git 상태를 동기화

CI/CD 파이프라인 단계

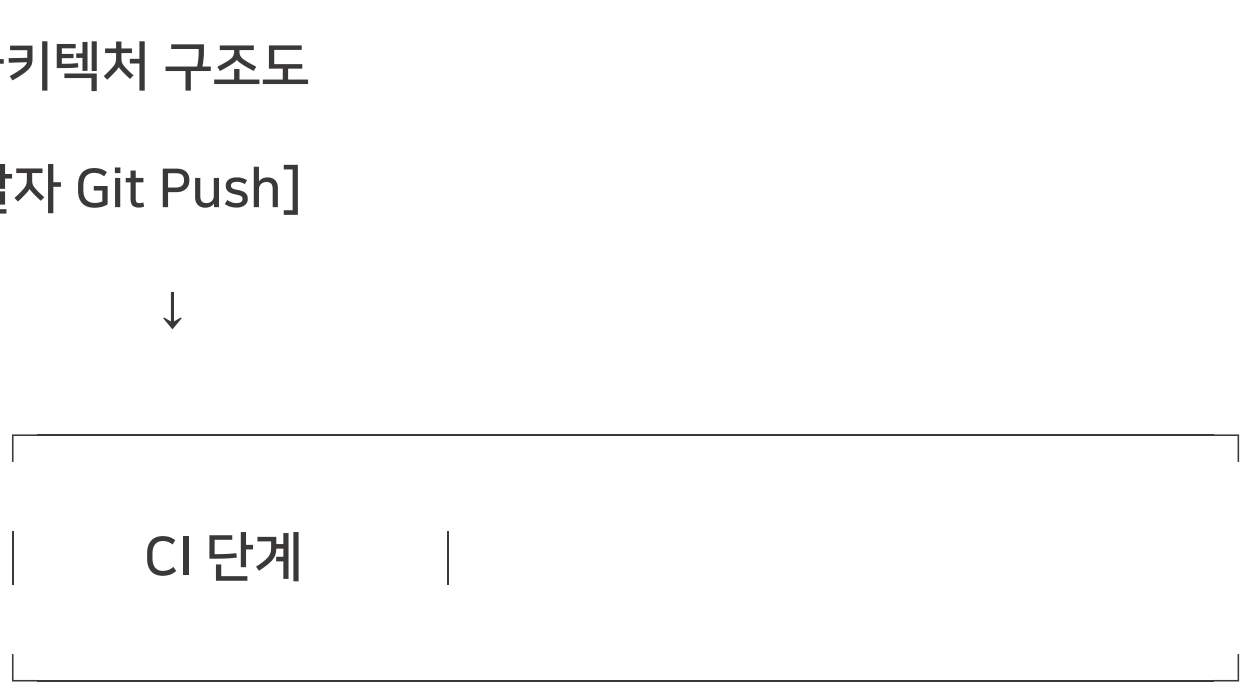
단계	키워드
CI	자동 테스트, 코드 품질 보증
이미지 빌드	컨테이너화, Registry 저장
CD	배포 자동화, GitOps 연동
운영	상태 확인, 알림, 롤백

GitOps 기반 CI/CD 통합

- GitOps 기반 CI/CD 통합 개요
  - CI = 코드 빌드 & 이미지 생성
  - CD = Git의 선언형 매니페스트를 기준으로 쿠버네티스 자동 배포
- 핵심 철학
  - Git에 기록된 YAML이 곧 운영 상태
  - 배포는 Push 방식이 아닌 Pull 방식 (보안 및 안정성 강화)

GitOps 기반 CI/CD 통합

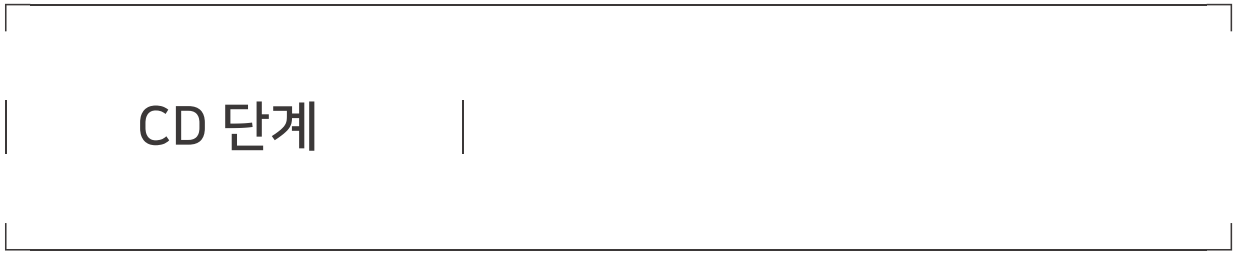
- 전체 아키텍처 구조도



GitOps 기반 CI/CD 통합

▶ Kubernetes YAML 수정 (tag 반영)

↓



▶ GitOps 연산자(Argo CD 등)가 Git 감시

▶ 클러스터에 변경 자동 적용

GitOps 기반 CI/CD 통합

- 단계별 흐름 상세
  - 1단계: CI (Continuous Integration)

작업	설명
Git Push	개발자가 코드 커밋 & PR 또는 main에 merge
테스트 & 빌드	GitHub Actions, Jenkins 등에서 자동 수행
Docker Build	애플리케이션을 컨테이너 이미지로 빌드
Registry Push	Docker Hub, GHCR, ECR 등으로 이미지 저장
매니페스트 업데이트	deployment.yaml에 새 이미지 태그 반영 & Git Commit

GitOps 기반 CI/CD 통합

- 2단계: CD (Continuous Deployment)

작업	설명
GitOps 도구 감시	Argo CD, Flux CD 등이 Git 저장소 변경 감지
상태 동기화	쿠버네티스 리소스를 선언형으로 업데이트
배포 완료 확인	UI / CLI로 상태 확인 및 자동 롤백 가능

GitOps 기반 CI/CD 통합

- 도구별 역할

역할	도구 예시
Git 저장소	GitHub / GitLab / Bitbucket
CI	GitHub Actions, GitLab CI, Jenkins
이미지 빌드	Docker, BuildKit, Kaniko
이미지 저장소	Docker Hub, GHCR, Harbor, ECR
GitOps 연산자 (CD)	Argo CD, Flux
배포 정의	Helm, Kustomize, YAML

GitOps 기반 CI/CD 통합

- 디렉토리 구조 예시

```
infra-gitops-repo/
├── apps/
│   └── my-app/
│       ├── base/
│       │   └── deployment.yaml     # 이미지 태그가 여기에 반영됨
│       └── overlays/
│           ├── dev/
│           └── prod/
├── argocd-app.yaml             # Argo CD Application 정의
```

GitOps 기반 CI/CD 통합

- GitOps 기반 통합의 장점

항목	설명
보안성 강화	클러스터 외부에서 직접 배포 명령이 없음 (Pull 방식)
버전관리 가능	배포 이력 모두 Git에 남음 (rollback 용이)
자동화	코드 Push → 배포까지 완전 자동화 가능
감사 및 추적성	변경 이유, 작성자, 시간 모두 Git 기록에 남음
테스트 환경 분리 용이	브랜치 또는 디렉토리 오버레이로 환경별 구성 가능

GitOps 기반 CI/CD 통합

구성 요소	핵심 역할
CI	코드 테스트, 이미지 빌드, YAML 반영
CD (GitOps)	Git 상태를 기준으로 클러스터 배포
연동 방식	GitHub Actions에서 GitOps용 리포 수정 → Argo CD가 자동 반영
보안 & 일관성	모든 배포 히스토리가 Git에 남고 클러스터가 그에 맞게 동작

Driving Promotions

- CI/CD 파이프라인에서의 Driving Promotions
  - 애플리케이션 아티팩트(코드, 이미지 등)를 개발 → 테스트 → 운영 환경으로 승격(Promote)하는 자동화된 흐름을 주도하는 것
- "Promotion"의 의미
  - 개발 환경에서 성공적으로 테스트된 빌드/이미지를
  - 점진적으로 스테이징 → 운영 환경으로 올리는 것

Driving Promotions

- Driving Promotions란?
  - 승격 조건을 만족한 버전을 정책 기반 자동화 또는 수동 승인 기반 절차를 통해 상위 환경으로 주도적으로 배포하는 행위
- GitOps 기반 Driving Promotions 파이프라인. 예)

  [1] feature 브랜치 → 개발 환경에 자동 배포

  [2] 테스트 통과 시 → staging 브랜치에 PR

  [3] 승인 후 → staging 환경으로 promotion

  [4] QA 통과 시 → main/master로 promotion

  [5] Argo CD가 운영 환경 자동 반영

Driving Promotions

- CI 도구 (GitHub Actions, Jenkins 등)는 테스트 수행
- GitOps 도구 (Argo CD, Flux)는 Git 변경 감지 → 배포 자동화
- Promotion은 Pull Request, 태그 또는 브랜치 머지로 구분

Driving Promotions

- 구성 요소

단계	설명
Artifact 생성	CI에서 Docker 이미지 빌드 후 레지스트리 저장
Promotion 트리거	테스트 통과 후 자동 PR or 수동 승인
Environment 매핑	브랜치별 환경: dev, staging, prod
GitOps 연동	각 환경 브랜치를 Argo CD가 감시
승격 기준 정의	테스트 결과, 커버리지, 승인자 등 기준을 기반으로 자동 승격 가능

Code vs Manifest vs App Config

- "Code vs Manifest vs App Config"는 현대 DevOps 및 GitOps 환경에서 자주 등장하는 3대 구성 요소

구분	설명
Code	애플리케이션의 **비즈니스 로직 및 기능**을 구현한 소스 코드
Manifest	쿠버네티스 등의 인프라 환경에 애플리케이션을 배포하기 위한 **배포 정의 파일 (YAML 등)**
App Config	애플리케이션 동작에 영향을 주는 **설정 값** (환경 변수, DB 주소, 외부 API Key 등)

Code vs Manifest vs App Config

- 비교 예

항목	예시
Code	main.py, AppController.java, routes.js
Manifest	deployment.yaml, service.yaml, ingress.yaml
App Config	.env, configmap.yaml, application.yml, secrets.yaml

- 책임 구분

역할	담당자
Code	개발자 (기능 구현)
Manifest	DevOps 엔지니어 (배포 자동화, GitOps 관리)
App Config	개발자 + DevOps (환경 변수, Secret 분리)

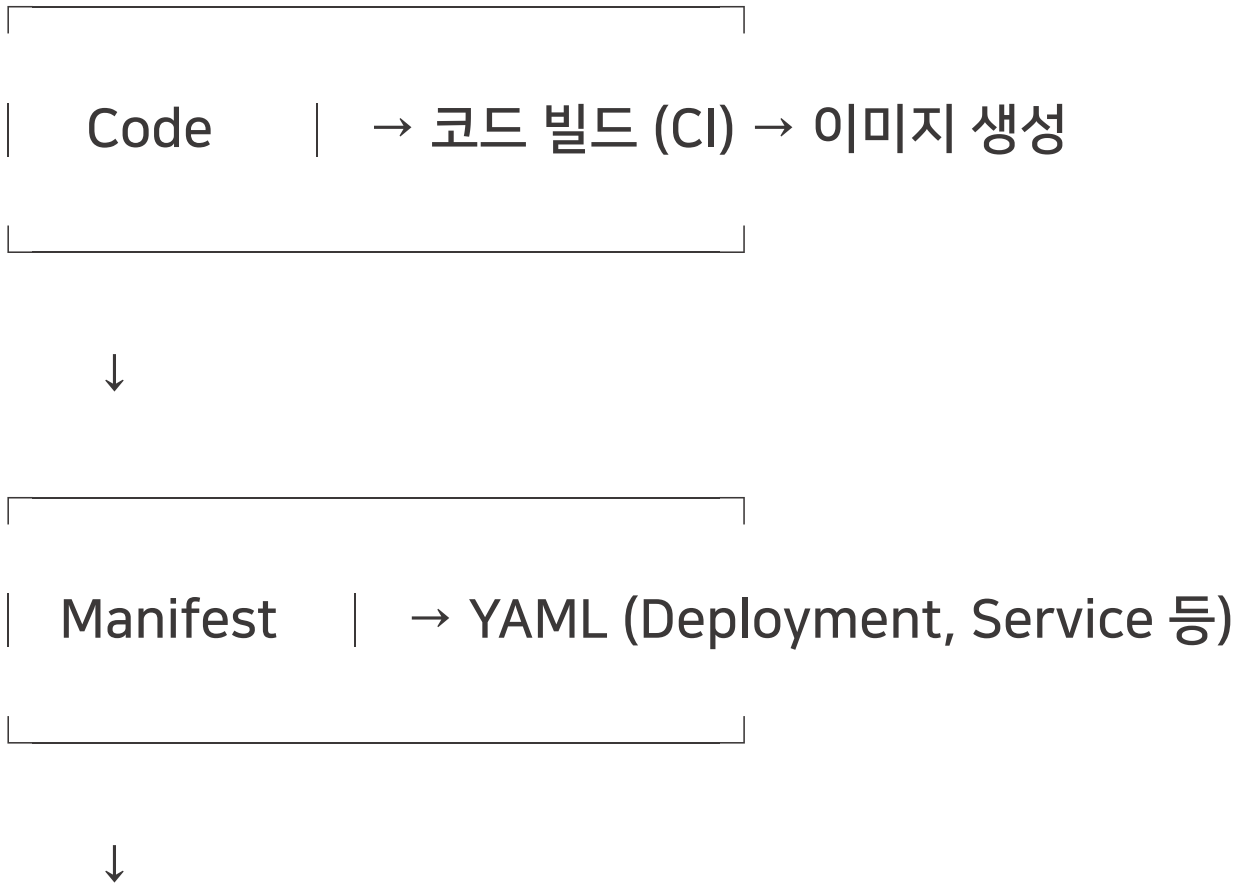
Code vs Manifest vs App Config

- GitOps 환경에서의 관리 방식

항목	저장 위치	관리 방식
Code	/src, /app	일반 코드 저장소 (main, feature/* 브랜치)
Manifest	/manifests, /overlays, /charts	GitOps 전용 리포지터리에서 선언형 관리
App Config	/config, ConfigMap, Secret	환경별 오버레이 또는 외부 Vault 연동

Code vs Manifest vs App Config

- 연관 구조도 (GitOps 기준)



Code vs Manifest vs App Config

Code vs Manifest vs App Config

- 비교

항목	Code	Manifest	App Config
정의	애플리케이션 로직	배포 정의 (인프라 리소스)	실행 환경 설정
포맷	.js, .py, .java 등	.yaml (K8s)	.yaml, .env, K8s ConfigMap/Secret
버전 관리	Git	Git (GitOps)	Git 또는 Vault
예시 파일	index.js	deployment.yaml	configmap.yaml, .env
자동화 대상	빌드 (CI)	배포 (CD)	실행 시 로드
관련 도구	GitHub, GitLab, VSCode	Helm, Kustomize, Argo CD	Vault, Sealed Secrets, Kustomize

Code	Manifest	App Config
무엇을 실행할 것인가?	**어떻게 배포할 것인가?**	**어떻게 동작할 것인가?**

다른 파이프라인 확인

- CI/CD 파이프라인 비교
  - 다른 프로젝트나 조직에서 사용 중인 CI/CD 파이프라인 구조를 확인하고 싶으신 경우
  - 예시: 3가지 대표 파이프라인 구조 비교

유형	구성	특징
전통형 파이프라인	Jenkins → Shell Script → 배포	유연성 높음, 관리 복잡
클라우드 네이티브형	GitHub Actions + Docker + Argo CD	GitOps 중심, 자동화 강점
GitLab 전용형	GitLab CI + GitLab Runner + Helm	All-in-one 구성, GitLab에 종속

다른 파이프라인 확인

- 환경별 파이프라인 (Dev/Staging/Prod)

  [Dev 브랜치]

  ↓

  CI: 테스트 & 빌드

  ↓

  자동 배포 (Dev 환경)

다른 파이프라인 확인

[Staging 브랜치]

↓

QA 승인 → 수동 승격

↓

자동 배포 (Staging 환경)

다른 파이프라인 확인

[Main 브랜치]

↓

CD: ArgoCD가 감시

↓

Prod 클러스터 자동 반영

다른 파이프라인 확인

- 데이터 파이프라인 vs 앱 배포 파이프라인
  - "파이프라인"이 ML/Data 흐름인지, 애플리케이션 배포 흐름인지 확인 필요

파이프라인 종류	주요 도구	사용 목적
애플리케이션 CI/CD	GitHub Actions, Jenkins, Argo CD	코드 → 배포 자동화
데이터 파이프라인	Airflow, Prefect, Dagster	ETL, 데이터 워크플로우
ML 파이프라인	Kubeflow, MLflow	모델 학습, 평가, 배포 자동화

다른 파이프라인 확인

- 멀티 서비스 or 마이크로서비스 파이프라인 비교
  - 여러 서비스가 각각 파이프라인을 가지는 구조 확인

```
repos/
├── service-a/
│   └── .github/workflows/ci-cd.yaml
├── service-b/
│   └── .github/workflows/ci-cd.yaml
└── manifests/
    ├── service-a/
    └── service-b/
```

롤백 및 규정 준수 파이프라인

- "롤백 및 규정 준수 파이프라인"은 DevOps와 GitOps 환경에서 안정성, 보안, 감사 가능성을 보장하는 핵심 구성 요소

용어	정의
롤백(Rollback)	잘못된 배포나 장애 발생 시 **이전 안정된 상태로 되돌리는 작업**
규정 준수(Compliance)	조직의 정책, 보안, 법률 규정을 만족시키는 배포 및 운영 체계

롤백 및 규정 준수 파이프라인

- 전체 파이프라인 흐름

  [Git Push]

  ↓

  [CI]

  - 테스트 & 보안 스캔

  - 정책 검사 (예: 리소스 제한, 네이밍 등)

  ↓

롤백 및 규정 준수 파이프라인

[CD (GitOps)]

- Argo CD / Flux로 배포
- 상태 감시 & 드리프트 복구
- 실패 시 자동 롤백

↓

[모니터링 & 감사 로그]

- Prometheus, Loki, Audit log, OPA 연동

롤백 및 규정 준수 파이프라인

- 롤백 전략 (Rollback Strategy)

방식	설명
GitOps 기반 롤백	Git의 이전 커밋으로 되돌리고 Argo CD가 자동 적용
Argo CD Auto-Rollback	Sync 실패/헬스체크 실패 시 자동 롤백
Helm Rollback	helm rollback 명령으로 이전 릴리스 복원
Canary 배포 + Failover	문제가 감지되면 자동 트래픽 전환 또는 복구
애플리케이션 레벨 Feature Flag	기능 단위로 롤백 (코드 롤백 없이 동작 변경)

롤백 및 규정 준수 파이프라인

- Argo CD에서 수동 롤백하는 방법
  - argocd app history my-app
  - argocd app rollback my-app <revision>

롤백 및 규정 준수 파이프라인

- 규정 준수 파이프라인 구성 요소

요소	설명
코드 정책 검사	CI에서 linters, OPA Gatekeeper, Kyverno 사용
이미지 보안 스캔	Trivy, Snyk, Clair로 취약점 감지
RBAC 검사	과도한 권한 설정 방지 (PodSecurity, PSP 등)
리소스 제한	resources.requests/limits 설정 확인
배포 검증	배포 전 정책 위반 여부 검사 후 승인 흐름 구성
감사 로그 보존	Argo CD Audit log 또는 Kubernetes Audit log 수집
Git 로그 + PR 기록	모든 배포 변경사항을 Git에서 추적 가능

배포 기본 사항 및 ReplicaSet 확인

- 배포(Deployment)의 기본 개념
  - 정의
    - Deployment는 쿠버네티스에서 애플리케이션을 선언적 방식으로 배포(Deploy), 업데이트(Update), 복구(Rollback) 할 수 있게 해주는 리소스
  - 주요 역할
    - 지정한 수만큼 Pod 복제본 유지
    - 새로운 이미지로 무중단 롤링 업데이트
    - 장애 시 자가 치유(Pod 재생성)
    - 이전 버전으로 롤백 가능

배포 기본 사항 및 ReplicaSet 확인

- Deployment 리소스 예

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
```

배포 기본 사항 및 ReplicaSet 확인

```
template:
  metadata:
    labels:
      app: my-app
  spec:
    containers:
      - name: web
        image: nginx:1.25
        ports:
          - containerPort: 80
```

배포 기본 사항 및 ReplicaSet 확인

- replicas: 3 → Pod 3개 유지
- template → Pod 정의 템플릿
- selector → 어떤 ReplicaSet과 Pod를 추적할지 지정

배포 기본 사항 및 ReplicaSet 확인

- ReplicaSet이란?
  - 정의
    - ReplicaSet은 Pod의 복제본 수를 관리하는 컨트롤러로, Deployment 내부에서 자동 생성되어 사용
    - ReplicaSet 단독 생성도 가능하나, 실무에서는 Deployment에 의해 자동 생성됨
    - Pod 수를 보장하고 재시작 등 자가 복구 기능을 담당
    - Pod 템플릿이 변경되면 새로운 ReplicaSet 생성 + 롤링 업데이트 진행

배포 기본 사항 및 ReplicaSet 확인

- Deployment ↔ ReplicaSet 관계

```
Deployment (my-app)
    ├── ReplicaSet (my-app-xxx123)
    │       ├── Pod (my-app-1)
    │       ├── Pod (my-app-2)
    │       └── Pod (my-app-3)
```

배포 기본 사항 및 ReplicaSet 확인

- 새로운 이미지 배포 시:
  - 새 ReplicaSet 생성
  - 새로운 Pod 생성 → 기존 ReplicaSet의 Pod 점진적 삭제
  - → 무중단 롤링 업데이트

배포 기본 사항 및 ReplicaSet 확인

- kubectl 명령어
  - Deployment 확인

    kubectl get deployments

    kubectl describe deployment my-app
  - ReplicaSet 확인

    kubectl get rs

    kubectl describe rs <replica-set-name>

배포 기본 사항 및 ReplicaSet 확인

- 연결 확인
  - ReplicaSet 이름은 Deployment명-랜덤해시 형식
  - kubectl get rs -l app=my-app 등으로 필터링 가능

배포 기본 사항 및 ReplicaSet 확인

- 롤백

  kubectl rollout history deployment my-app # 배포 이력 확인

  kubectl rollout undo deployment my-app # 이전 상태로 롤백

  - Deployment는 변경 추적 + 자동 관리 덕분에 운영 중에도 안전하게 버전 관리를 할 수 있음

배포 기본 사항 및 ReplicaSet 확인

- Deployment와 ReplicaSet의 차이

항목	Deployment	ReplicaSet
역할	배포 전략 관리, 업데이트, 롤백	지정된 수의 Pod 유지
생성 방식	사용자 생성	Deployment가 내부적으로 생성
업데이트	새로운 ReplicaSet 생성하여 롤링 업데이트	단독 사용 시 수동 관리 필요
롤백 지원	있음 (rollout undo)	없음 (직접 삭제/수정 필요)

- Deployment는 쿠버네티스 배포 자동화를 위한 핵심 리소스
- ReplicaSet은 실제 Pod 수를 유지하는 자동화 컨트롤러
- 항상 Deployment를 통해 Pod 배포, 업데이트, 롤백을 관리하는 것이 권장

배포 전략

- 배포 전략 소개
- Rollouts 소개 및 특징
- Blue-Green 배포 실습
- Canary 배포 실습
- 점진적인 Delivery 구현

배포 전략 소개

- 쿠버네티스 배포 전략

전략명	특징	장점	단점
Recreate	기존 Pod 모두 종료 후 새 Pod 생성	간단함	**서비스 중단 발생**
RollingUpdate	기존 Pod를 점진적으로 교체	무중단, 기본값	롤백 속도 느림
Blue/Green	새 버전 Pod를 별도 배포 후 트래픽 전환	빠른 롤백, 병렬 환경	리소스 추가 필요
Canary	일부 사용자만 새 버전 사용 후 점진 확대	안전한 실사용 테스트	설정 복잡
A/B Testing	조건 기반 트래픽 분기 (예: 지역, 브라우저)	정밀 제어 가능	트래픽 관리 도구 필요

배포 전략 소개

- 쿠버네티스 기본 전략: RollingUpdate

```
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1        # 추가로 생성할 수 있는 Pod 수
      maxUnavailable: 1  # 동시에 중단 가능한 Pod 수
```

배포 전략 소개

- maxSurge: 새 버전을 얼마나 더 띄울지 (기본 25%)
- maxUnavailable: 몇 개까지 중단 가능할지 (기본 25%)
- 무중단 업데이트가 기본 제공되며, 프로덕션 환경에서 가장 많이 사용

배포 전략 소개

- Recreate 전략

```
spec:
  strategy:
    type: Recreate
```

  - 모든 기존 Pod를 종료한 뒤 새 버전을 생성
  - 중단 시간 발생 가능 → 실무에서는 거의 사용하지 않음

배포 전략 소개

- Blue/Green 배포
  - blue: 현재 운영 중인 버전
  - green: 새 버전 → 검증 후 트래픽 전환
  - 구현 방식
    - 별도 Deployment 두 개 운영 (my-app-blue, my-app-green)
    - Service 리소스를 통해 어느 쪽에 트래픽 보낼지 결정
    - 문제 발생 시 빠르게 이전 서비스로 전환 가능

배포 전략 소개

- 장점
  - 빠른 롤백
  - 운영 중 환경 격리 가능

배포 전략 소개

- Canary 배포
  - 전체 사용자 중 일부만 먼저 새 버전을 사용 → 문제 없으면 점진적 확장
    - 보통 Istio, NGINX Ingress, Flagger 등과 함께 사용
    - 예: 10% 트래픽만 신규 버전으로 전송 후 점차 100% 확대
  - 적합한 상황:
    - 사용자가 많고 신중한 변경이 필요한 경우
    - 실제 트래픽에서 검증하고 싶은 경우

배포 전략 소개

- 전략 선택 기준

조건	추천 전략
단순하고 소규모 서비스	RollingUpdate, Recreate
대규모 트래픽, 빠른 롤백 필요	Blue/Green
점진적 검증 필요	Canary
조건별 사용자 실험	A/B Testing

배포 전략 소개

- 고급: GitOps + 배포 전략
  - GitOps 도구(Argo CD, Flagger 등)와 연동 시:
    - Git에 전략을 선언
    - 배포 상태 및 트래픽 분배 실시간 확인
    - 실패 시 자동 중단 또는 롤백
  - Argo Rollouts + Istio/Ingress 조합으로 Canary & Blue/Green 구현 가능

배포 전략 소개

전략	요약
RollingUpdate	기본값, 무중단, 점진 교체
Recreate	서비스 중단 있음, 단순
Blue/Green	두 환경 병렬 운영, 빠른 전환
Canary	일부 사용자에게 먼저 배포
A/B Testing	트래픽 조건별 분기 실험

Rollouts 소개 및 특징

- Argo Rollouts란?
  - Argo Rollouts는 Canary 배포, Blue/Green 배포, Progressive Delivery를 쿠버네티스에서 쉽게 구현할 수 있도록 해주는 오픈소스 컨트롤러(Operator)임
    - 쿠버네티스의 Deployment를 대체하는 Rollout 리소스를 제공
    - Argo CD와 연동 시 UI에서 실시간 트래픽 전환 & 상태 관찰 가능
    - Istio, NGINX Ingress, Service Mesh 등과 통합 가능

Rollouts 소개 및 특징

- 주요 배포 전략 지원

전략	설명
Canary	점진적 트래픽 전환: 5% → 25% → 100% 식의 배포
Blue/Green	새 버전과 기존 버전 병행 운영 후 트래픽 스위칭
Manual or Automated Promotion	사람이 수동 승인하거나 자동 조건 만족 시 다음 단계로 전환

Rollouts 소개 및 특징

- 기본 구조 예시 (Canary)

```
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: my-app
spec:
  replicas: 4
  strategy:
    canary:
```

Rollouts 소개 및 특징

```
steps:
    - setWeight: 10
    - pause: { duration: 60s }
    - setWeight: 50
    - pause: {}
 selector:
   matchLabels:
     app: my-app
```

Rollouts 소개 및 특징

```
template:
  metadata:
    labels:
      app: my-app
  spec:
    containers:
      - name: my-app
        image: my-app:v2
```

- 트래픽을 10%로 시작 → 60초 후 50% → 수동 승인 대기

Rollouts 소개 및 특징

- 특징

항목	설명
Progressive Delivery	Canary & Blue/Green 같은 점진적 배포를 선언형으로 지원
실시간 상태 추적	배포 진행 상태, 버전 별 트래픽 분포 등 시각화 (UI 제공)
자동/수동 승인	단계별 자동 전환 or 사람 승인 가능
서비스 메시/Ingress 통합	Istio, NGINX, SMI 등과 통합하여 트래픽 제어
재시도 & 롤백	실패 시 자동 중단, 이전 상태로 롤백 가능
Metric 기반 자동화	Prometheus 등과 연동하여 지표 기반 배포 판단 가능 (예: 오류율 증가 시 중단)

Rollouts 소개 및 특징

- Rollout vs Deployment

항목	Deployment	Rollout
배포 전략	RollingUpdate 위주	Canary, Blue/Green 등 고급 전략 지원
UI 연동	기본 없음	Argo CD 연동 시 시각화
조건부 배포	미지원	Pause, Approval, Metric 기반 등 유연한 조건 지원
실험적 적용	어렵다	일부 트래픽만 새 버전에 전달 가능

Rollouts 소개 및 특징

- 활용

환경	기능
QA 테스트	Canary 10% 배포 후 테스트 통과 시 전면 전환
프로덕션 운영	Blue/Green으로 운영 중 트래픽 스위칭
자동화	Prometheus의 오류율 지표로 실패 감지 후 자동 중단 및 롤백

Rollouts 소개 및 특징

- Argo Rollouts 연동 구성 요소

구성 요소	역할
Rollout Controller	Rollout 리소스를 감시하고 관리
Argo CD	UI 및 GitOps 관리
트래픽 라우터	Istio / NGINX / SMI 등 (Canary 분배에 필요)
Metric Provider	Prometheus, Kayenta 등 (지표 기반 자동화 시)

Rollouts 소개 및 특징

항목	설명
정체	쿠버네티스의 배포 전략 고도화 도구
주요 전략	Canary, Blue/Green
장점	선언형 구성, 점진적 배포, 실시간 관찰, 자동 롤백
연동 가능	GitOps(Argo CD), Ingress, Service Mesh, Prometheus 등

Blue-Green 배포 실습

- Blue-Green 배포 개요
  - 두 가지 버전(Blue = 현재 운영 중, Green = 새 버전)을 동시에 존재시킨 후, 트래픽을 Green으로 전환하여 배포하는 전략
    - 빠른 롤백 가능
    - 다운타임 없음
    - QA 확인 후 트래픽 전환

Blue-Green 배포 실습

- 실습 구조 개요

  [Deployment-blue] --> app:v1

  [Deployment-green] --> app:v2 (대기 중)

  ↑

  [Service] → 선택된 버전으로 트래픽 전달

Blue-Green 배포 실습

- 실습 순서

단계	설명
①	Deployment-blue 생성 (v1)
②	Service 생성 → blue를 선택
③	Deployment-green 생성 (v2)
④	검증 (green은 트래픽 없음)
⑤	Service를 green으로 변경
⑥	전환 완료, 필요 시 롤백

실습 리소스 YAML

① deployment-blue.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-blue
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
      version: blue
  template:
    metadata:
      labels:
        app: my-app
        version: blue
    spec:
      containers:
        - name: nginx
          image: nginx:1.21
          ports:
            - containerPort: 80
```

실습 리소스 YAML

② deployment-green.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-green
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
      version: green
  template:
    metadata:
      labels:
        app: my-app
        version: green
    spec:
      containers:
        - name: nginx
          image: nginx:1.25
          ports:
            - containerPort: 80
```

실습 리소스 YAML

③ service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
    version: blue     # 처음에는 blue 선택
  ports:
    - port: 80
      targetPort: 80
  type: ClusterIP
```

Blue-Green 배포 실습

- 실습 명령어

```
# 배포

kubectl apply -f deployment-blue.yaml

kubectl apply -f service.yaml

# green 버전은 아직 트래픽 없음

kubectl apply -f deployment-green.yaml

# 서비스 확인

kubectl get svc my-app-service

# 현재 연결된 Pod 확인

kubectl get endpoints my-app-service
```

Blue-Green 배포 실습

- 트래픽 전환

  # Service를 green으로 업데이트

  kubectl patch service my-app-service -p '{"spec": {"selector": {"app": "my-app", "version": "green"}}}'

  - 이제 모든 트래픽은 green으로 전환됨.

Blue-Green 배포 실습

- 롤백 (blue로 되돌리기)

  ```
  kubectl patch service my-app-service -p '{"spec": {"selector": {"app": "my-app", "version": "blue"}}}'
  ```

Blue-Green 배포 실습

검증 항목	명령어
현재 서비스 대상 Pod	kubectl get endpoints my-app-service
배포 상태 확인	kubectl get deployment
파드 로그 확인	kubectl logs -l version=green
트래픽 테스트	kubectl port-forward svc/my-app-service 8080:80 → 브라우저로 localhost:8080 접속

Blue-Green 배포 실습

- 확장: Argo CD + Blue/Green
  - Rollout 리소스를 활용해 Argo Rollouts와 연계 가능
  - 또는 Argo CD에서 두 개의 Deployment를 각각 관리하고, Service 변경 커밋을 통해 GitOps 방식으로 트래픽 전환

Blue-Green 배포 실습

항목	내용
목적	두 버전 공존 → 안전 전환
전환 방식	Service.selector 변경
장점	빠른 롤백, 무중단 배포
확장	Argo CD, Istio와 통합 가능

Canary 배포 실습

- Canary 배포 실습 개요
  - 신규 버전(v2)을 일부 트래픽만 받도록 먼저 배포 후, 문제가 없으면 점진적으로 확대
    - Replica 수와 Service Selector 조합으로 트래픽 분할
    - 쿠버네티스 기본 기능만으로 가능 (서비스 메시 없이도 가능)

Canary 배포 실습

- 실습 구조 개요

  [Deployment-v1 (Stable)] ← 90% 트래픽

  [Deployment-v2 (Canary)] ← 10% 트래픽

  ↑

  [Service] → app=my-app 선택

초기 Stable 버전 배포

① deployment-v1.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-v1
spec:
  replicas: 9
  selector:
    matchLabels:
      app: my-app
      version: v1
  template:
    metadata:
      labels:
        app: my-app
        version: v1
    spec:
      containers:
        - name: app
          image: nginx:1.21
          ports:
            - containerPort: 80
```

Service 생성

② service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
    - port: 80
      targetPort: 80
```

Canary 버전 배포 (v2)
③ deployment-v2.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
      version: v2
  template:
    metadata:
      labels:
        app: my-app
        version: v2
    spec:
      containers:
        - name: app
          image: nginx:1.25
          ports:
            - containerPort: 80
```

Canary 배포 실습

- 배포 실행

  kubectl apply -f deployment-v1.yaml

  kubectl apply -f deployment-v2.yaml

  kubectl apply -f service.yaml

Canary 배포 실습

- Canary 트래픽 테스트

  # 포트포워딩으로 트래픽 테스트

  kubectl port-forward svc/my-app-service 8080:80

  # 트래픽 분포 보기 (버전별 Pod 구분)

  watch kubectl get pods -l app=my-app -o wide

  - 여러 번 접속하거나 curl localhost:8080을 반복 실행
  - v1과 v2가 번갈아 응답하는지 확인 (v2가 10%)

Canary 배포 실습

- 트래픽 점진 확대 or 롤백

작업	명령어
Canary 확장	kubectl scale deployment my-app-v2 --replicas=5
Stable 축소	kubectl scale deployment my-app-v1 --replicas=5
롤백	kubectl delete deployment my-app-v2

Canary 배포 실습

단계	설명
v1 안정 버전 먼저 배포	9개의 replica
v2를 1개만 배포	Canary 역할
Service는 v1+v2 모두 선택	트래픽 분산
이상 없음 확인 후 점진 확대	수동 조정 or Rollouts 자동화
이상 발견 시 즉시 롤백	v2만 삭제하면 됨

점진적인 Delivery 구현

- 점진적 Delivery란?
  - “애플리케이션의 새로운 버전을 점진적으로 배포하여, 문제가 없는 경우에만 전체 배포로 확대하는 전략”

- 구현 목적

목적	설명
문제 조기 탐지	전체 트래픽 이전에 일부 사용자로 문제 발생 여부 파악
리스크 완화	오류가 발생하면 빠르게 중단/롤백 가능
실사용 기반 검증	실제 트래픽 조건에서 안정성 확인

점진적인 Delivery 구현

- 주요 구현 방식

방식	설명	트래픽 제어
Canary 배포	점진적으로 트래픽 전환 (예: 10% → 50% → 100%)	서비스 메시 or Ingress
Blue/Green 배포	두 환경을 병렬로 운영 후 트래픽 스위칭	Service or Gateway 전환
A/B 테스트	사용자 조건 기반 버전 분기 (예: 지역, 브라우저)	고급 트래픽 라우팅 필요

점진적인 Delivery 구현

- 핵심 구성 요소

구성 요소	설명
Rollout Controller	배포 진행 상태를 제어하는 컨트롤러
트래픽 라우터	Istio, NGINX Ingress, SMI 등
모니터링/지표	Prometheus, Kayenta, Datadog 등
자동화 도구	Argo Rollouts, Flagger 등
승인 메커니즘	자동/수동 승인, 헬스체크 기반 전환