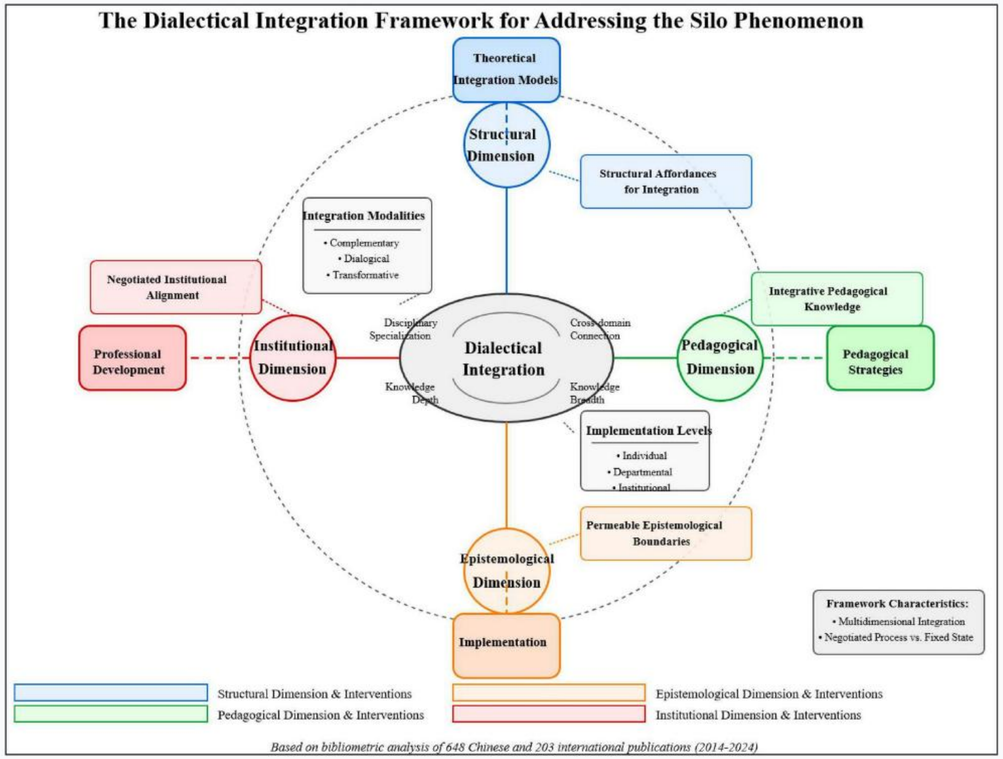
DevOps와 CI/CD

S개발자 프로그램

Section 1. DevOps의 개요

기존 IT 운영 환경의 한계

- ⚠ 개발의 민첩성과 인프라의 안정성이라는 **상충된 목표**가 고착화됨
- ⚠ 부서 간 장벽(Silo)으로 인해 코드 배포 일정이 지속적으로 지연됨
- ⚠ 수동 배포 작업에 의존하여 **인적 오류 (Human Error)** 가능성이 높음



전통적 사일로(Silo) 문제

- **개발팀(Dev):** 잦은 기능 변경과 신속한 코드 배포를 최우선 목표로 삼음
- **운영팀(Ops):** 시스템의 무결성과 무중단 등 안정적인 가용성 확보를 중시함
- 각 부서의 단절된 소통 체계와 상반된 평가지표는 **책임 전가의 원인**으로 작용함
- 결국, 고객에게 새로운 가치를 전달하는 전체 리드 타임이 심각하게 지연됨

DevOps의 탄생 배경

- **애자일(Agile)의 확산:** 소프트웨어 개발 속도는 극도로 빨라졌으나, 인프라 배포가 이를 따라가지 못해 병목 현상이 심화됨
- **클라우드 환경의 발전:** 하드웨어 기반이던 인프라를 클라우드 API를 통해 동적이고 유연하게 제어할 수 있게 됨
- 신속한 비즈니스 피드백 루프 구성을 위해 **개발과 운영의 유기적 결합**이 필수 요소로 대두됨

DevOps의 핵심 정의

단순 직무 결합 탈피

- 개발과 운영 부서의 물리적인 조직 통합만을 의미하지 않음
- 애플리케이션을 신속히 제공하기 위한 **문화적 철학, 방식 및 도구의 결합체**임

가치 전달의 극대화

- 전통적인 개발 프로세스 대비 제품의 **진화와 배포 속도를 비약적으로 단축**함
- 이를 통해 최종 고객의 만족도를 높이고 비즈니스 경쟁력을 강화함

DevOps 핵심 철학: CAMS 모델이란?

CAMS

DevOps 4대 철학

가장 널리 통용되는 기준

- DevOps를 조직에 정착시키기 위해 필요한 **4가지 필수 프레임워크**임
- Culture, Automation, Measurement, Sharing의 약자를 결합하여 명명됨

CAMS 모델 1: Culture (문화)

- DevOps는 도구 도입 이전에 **상호 협업의 문화**에서부터 출발함
- 부서 간의 장벽을 제거하고 프로젝트 전체를 관통하는 투명한 정보 공유를 지향함
- 장애가 발생했을 때 개인을 문책하지 않고 프로세스를 개선하는 **비난 없는 사후 분석 (Blameless Post-mortem)** 문화를 조성함

CAMS 모델 2: Automation (자동화)

- 인간의 반복적인 수동 작업은 배포 속도를 저하시키고 예측 불가능한 인적 오류를 유발함
- 소스 코드의 빌드, 단위 테스트, 패키징, 배포 등 **모든 프로세스를 코드로 작성하여 시스템이 수행하도록 자동화**함
- CI/CD 파이프라인의 성공적인 구축이 자동화의 핵심 구현체로 작용함

CAMS 모델 3: Measurement (측정)

- "측정할 수 없는 것은 관리할 수 없다"는 원칙을 시스템 운영에 철저히 적용함
- 서버 리소스, 애플리케이션 응답 속도, 사용자 트래픽 등 서비스의 상태를 **정량적인 지표(Metric)로 지속 수집**함
- 수집된 데이터를 시각화하여 모니터링하고 병목 구간을 탐지하여 시스템 성능을 개선함

CAMS 모델 4: Sharing (공유)

- 성공적인 자동화 스크립트 도구, 장애 해결 사례, 모범 사례를 조직 내에 적극적으로 전파함
- 특정 인원의 개인적 지식을 팀과 조직의 자산으로 승화시켜 **공동의 기술 역량 강화**를 도모함
- 사내 위키, 기술 세미나 등을 지속적으로 개최하여 지식의 선순환 구조를 확립함

기존 모델 vs DevOps 모델

구분	전통적 (Waterfall) 모델	DevOps 모델
조직 구조	개발팀, QA팀, 운영팀 분리	다양한 직무가 통합된 목적 조직
배포 주기	분기 및 월 단위 대규모 릴리스	일 단위 혹은 수시 소규모 릴리스
배포 주체	운영 담당자의 수동 개입 배포	파이프라인 통한 자동화 배포
장애 대응	원인 파악 지연 및 책임 전가	빠른 피드백 기반 즉각적 조치

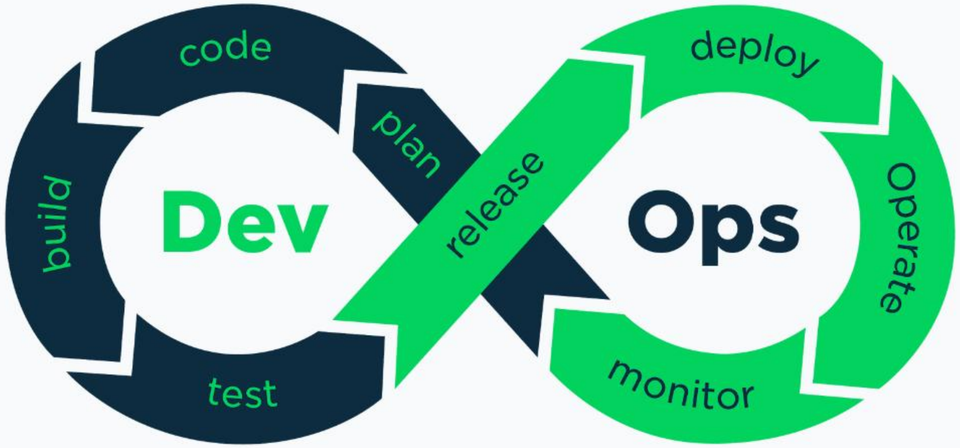
DevOps 도입의 비즈니스 가치

- **시장 출시 시간 단축:** 고객 요구사항을 신속하게 제품에 반영하여 비즈니스 경쟁력 우위 선점
- **제품 품질 및 안정성 제고:** 작은 단위의 배포와 철저한 자동화 테스트를 통해 장애 파급력을 최소화
- **개발자 생산성 증대:** 인프라 수동 관리에 낭비되는 리소스를 줄이고 핵심 비즈니스 로직 개발에 집중

Section 2. 7Cs of DevOps

7Cs of DevOps 개념 정의

- 소프트웨어의 기획부터 배포, 모니터링까지의 **전체 라이프사이클을 7개의 단계로 세분화**한 모델임
- 모든 단계 앞에는 **'Continuous (지속적)'**이라는 단어가 붙으며, 멈춤 없이 순환하는 루프를 의미함
- 단방향 흐름이 아닌, 마지막 단계의 결과가 다시 첫 단계로 피드백되는 무한한 개선 구조임



1. Continuous Development (지속적 개발)

- 비즈니스 요구사항을 수집하여 기획하고, 실제 소프트웨어 코드를 작성하는 전 단계를 포괄함
- 거대한 목표를 2~3주 단위의 애자일 스프린트 단위로 세분화하여 단기 로드맵을 수립함
- Git과 같은 형상 관리 도구를 적극 활용하여 분산 환경에서 효율적인 코드 버전 관리를 수행함

2. Continuous Integration (지속적 통합)

- 다수의 개발자가 각자 수정한 코드를 중앙의 메인 브랜치로 빈번하게 병합함
- 코드가 병합되는 즉시 소스 코드의 컴파일 및 패키징 등 빌드 프로세스가 자동으로 트리거됨
- 소스 코드 충돌 현상을 조기에 해결하고, 프로젝트를 항상 실행 가능한 상태로 통제함

3. Continuous Testing (지속적 테스트)

- 빌드된 코드를 대상으로 단위 테스트 및 통합 테스트 스크립트를 자동 실행함
- 새롭게 반영된 코드가 기존 기능에 악영향을 주지 않았는지 즉각적으로 판별함
- 수동으로 검증하던 품질 통제(QA) 과정을 자동화하여 검증의 정확도를 대폭 상향시킴

4. Continuous Deployment (지속적 배포)

- 모든 테스트를 무사히 통과한 빌드 산출물을 최종 목적지인 운영 환경에 릴리스함
- 자동화된 배포 스크립트를 사용하여 사람의 개입이나 서버 중단 없이 안전하게 시스템에 반영함
- 실제 사용자에게 새로운 비즈니스 가치가 물리적으로 전달되는 가장 핵심적이고 최종적인 단계임

5. Continuous Feedback (지속적 피드백)

- 실제 운영에 배포된 애플리케이션에 대한 내/외부 사용자의 반응과 지표 데이터를 체계적으로 수집함
- 사용자 설문, 이슈 트래커 등을 통해 제품 개선을 위한 정성적 및 정량적 인사이트를 확보함
- 수집된 피드백은 **다음 기획(Plan) 사이클의 방향성을 결정짓는 필수 지표**로 직접 환류됨

6. Continuous Monitoring (지속적 모니터링)

- 배포 완료된 시스템 인프라 및 애플리케이션의 런타임 상태를 24시간 상시 감시함
- 서버 리소스 부하, 네트워크 트래픽, 에러 로그 등 필수 기술적 지표를 대시보드에서 분석함
- 장애 혹은 병목 지점 발생 시 즉각적인 알림 메시지를 발생시켜 서비스 중단 타임을 최소화함

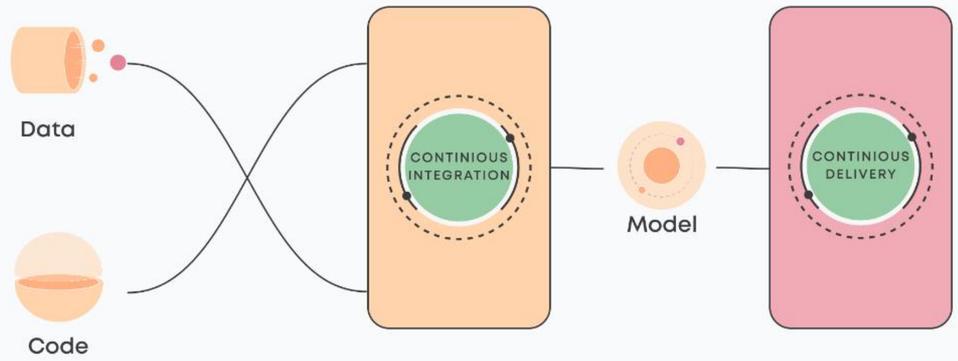
7. Continuous Operations (지속적 운영)

- 정기 점검 및 인프라 패치 작업 중에도 서비스 중단이 발생하지 않도록 고가용성 아키텍처를 유연하게 운영함
- 사용자 트래픽이 급증할 경우 클라우드 서버 인스턴스를 동적으로 확장하여 부하를 분산함
- 하드웨어 장애 상황에서도 시스템 스스로 복구 로직을 수행하도록 내결함성 환경을 유지함

Section 3. CI/CD 파이프라인의 이해

CI/CD의 개념적 통합

- DevOps의 철학적 사상을 기술적 시스템으로 구체화하는 **가장 핵심적인 자동화 프랙티스**임
- 소스 코드의 병합부터 운영 서버 배포까지의 전 단계를 하나의 **파이프라인 (Pipeline)**으로 결속함
- 수동 작업을 파이프라인에서 완벽히 제거하여 릴리스의 속도와 안정성을 극적으로 향상시킴



Continuous Integration (지속적 통합)

CI의 기본 개념

- 개발자가 작성한 로컬 코드를 원격 저장소에 **하루에도 수십 번씩 주기적으로 병합**함

CI의 핵심 동작

- 병합 이벤트 감지 시 CI 서버가 코드를 가져와 **빌드 및 사전에 정의된 테스트를 일괄 실행**함

CI 환경 구축의 핵심 이점

- **통합 지옥 예방:** 프로젝트 후반부에 코드를 억지로 합치며 발생하는 수많은 모듈 간 충돌 현상을 원천 방지함
- **초기 버그 식별:** 변경된 코드의 정상 작동 여부가 즉시 검증되므로, 결함을 코딩 단계에서 저비용으로 해결함
- **코드 베이스 무결성:** 테스트를 통과한 정상 코드만 메인 브랜치에 반영되므로 즉시 배포 가능한 고품질 상태를 유지함

Continuous Delivery (지속적 제공)

개념 및 특징

- CI를 통과한 빌드 패키지를 실제 서비스 환경에 배포할 수 있도록 **준비 상태로 자동 유지**함

수동 승인 절차

- 스테이징 환경까지는 자동 배포되나, 운영 환경 최종 적용은 **반드시 관리자의 수동 승인**을 거침

Continuous Deployment (지속적 배포)

궁극의 자동화 배포

- 파이프라인의 모든 검증 관문을 통과한 코드가 **운영 환경까지 사람의 개입 없이 100% 자동 릴리스**됨

높은 성숙도 요구

- 운영망으로 직행하는 구조상 **매우 정교한 자동화 테스트 커버리지** 구축이 필수적 선결 조건임

CD 환경 구축의 핵심 이점

- **리드 타임 최소화:** 코드 커밋부터 실제 고객에게 기능이 도달하기까지의 물리적 지연 시간을 크게 단축시킴
- **수동 배포 오류 근절:** 복잡한 환경 설정 명령을 기계가 정확하게 수행하여 담당자 실수로 인한 장애 가능성을 제거함
- **손쉬운 롤백 보장:** 배포 이력이 버전으로 관리되므로, 문제 발생 시 즉시 이전 버전 스크립트를 재실행하여 시스템 복구가 원활함

Delivery vs Deployment 핵심 차이

항목	Continuous Delivery (제공)	Continuous Deployment (배포)
운영 환경 배포 주체	사람 (수동 클릭 및 의사 결정)	시스템 (100% 자동화 파이프라인)
적용 목적	특정 시점에 비즈니스 통제 배포	가장 빠른 속도의 사용자 가치 전달
테스트 신뢰성 의존도	상대적으로 중간 (최종 수동 검수 존재)	매우 높음 (테스트가 유일한 방어선)

CI/CD 파이프라인 4단계 구성

1. Source

저장소에 코드 커밋 등 이벤트 발생 감지

2. Build

소스 코드 컴파일 진행 및 배포 패키지 생성

3. Test

사전 정의된 자동화 테스트 및 정상 작동 검증

4. Deploy

각종 운영 인프라 환경으로 산출물 최종 배포

Section 4. 인프라 자동화와 배포 전략

IaC (Infrastructure as Code) 개념

인프라의 코드화

- 클라우드 콘솔 수동 클릭 대신, **선언적 텍스트 파일(코드)**을 작성하여 인프라 자원을 배포함

효율성과 일관성

- 인프라 환경을 코드로 동일하게 재현하여 **휴먼 에러를 방지**하고 변경 이력 추적을 보장함

불변 인프라 (Immutable Infrastructure)

- 서버 인프라가 최초 생성된 이후에는 **절대 내부의 속성이나 환경 설정을 인위적으로 수정하지 않음**
- 패치나 업데이트가 필요할 경우, **새로운 이미지를 빌드하여 서버 인스턴스 자체를 통째로 교체**함
- 시간이 지남에 따라 서버마다 설정이 미세하게 달라지는 구성 표류(Configuration Drift) 문제를 원천 차단함

무중단 배포 (Zero Downtime)

- 과거에는 새로운 버전 배포 시 기존 프로세스를 종료하며 **필연적인 서비스 접속 중단 (Downtime)**이 발생함
- 글로벌 서비스가 기본이 된 환경에서는 사용자 체감 중단 시간이 0에 수렴하는 고급 배포 기법이 필수적임
- **로드 밸런서(Load Balancer)의 라우팅 기술**을 이용하여 트래픽을 정교하게 제어하며 서버를 무중단 교체함

배포 전략 1: 롤링 (Rolling Update)

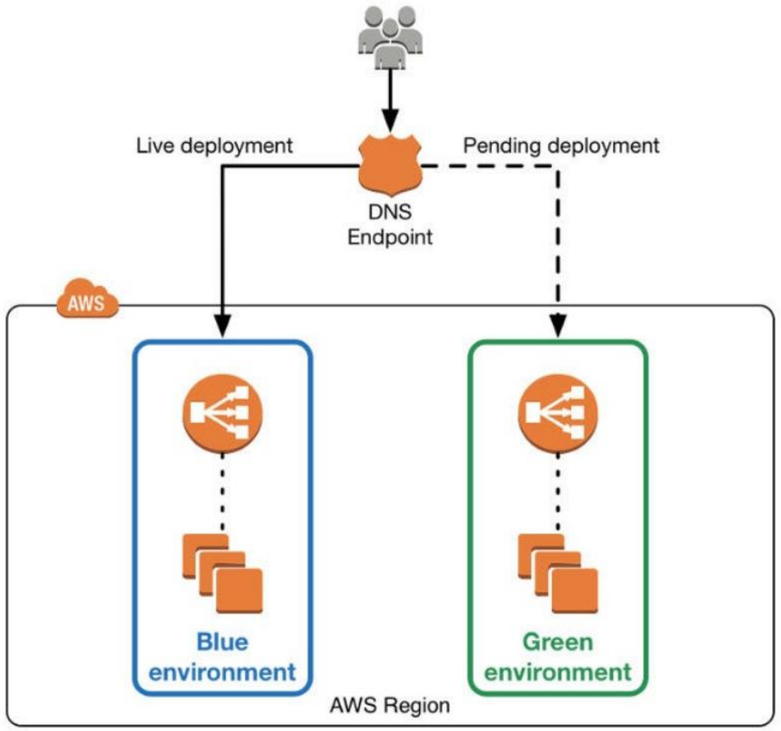
- 운영 서버 구동 시, 모든 서버를 한 번에 내리지 않고 **순차적으로 몇 대씩 인스턴스를 교체**함
- 사전에 구축된 대규모 추가 리소스가 불필요하여 초기 인프라 비용 측면에서 상대적으로 유리함
- 배포 중에는 신/구 버전 코드가 혼재되어 서비스되므로, API 및 DB 호환성에 대한 세심한 통제가 요구됨

배포 전략 2: 카나리 (Canary Release)

- 광산의 가스를 감지하기 위해 카나리아 새를 먼저 보냈던 일화에서 유래한 매우 안전한 배포 방식임
- 신버전 코드를 **전체 트래픽의 극히 일부 사용자에게만 우선적으로 라우팅**하여 배포함
- 해당 트래픽에서 심각한 오류가 없는지 모니터링 후, 점진적으로 신버전 비중을 늘려가는 보수적 기법임

배포 전략 3: 블루/그린 (Blue/Green)

- 기존 운영 환경(Blue)과 **동일한 환경 (Green)을 독립적으로 사전 구축**함
- 준비 완료 시, 로드 밸런서의 트래픽을 신규 서버로 **일시에 전환(Switch)**함
- 문제 발생 시 즉각 라우팅을 복구하여 **속도가 가장 뛰어남**



Section 5. 주요 CI/CD 플랫폼 분석

CI/CD 플랫폼 생태계 분류

플랫폼 분류 형태	대표 플랫폼	주요 특징 요약
설치형 / 오픈소스	Jenkins, GitLab CI	기업 인프라 내 직접 설치, 높은 커스터마이징 허용
관리형 SaaS (Cloud)	GitHub Actions, Circle CI	별도 서버 관리 불필요, 외부 소스 저장소 연동 용이
퍼블릭 클라우드 전용	AWS CodeBuild 등	특정 클라우드 내부 자원 및 IAM 보안 체계에 완벽 동화

Jenkins 개요

- Java 언어로 개발되었으며, **전 세계에서 가장 보편적으로 사용되는 오픈소스 자동화 서버**임
- 분산 빌드를 지원하는 Master-Agent 아키텍처를 채택하여 복잡하고 방대한 병렬 워크로드 처리에 능숙함
- Jenkinsfile을 기반으로 파이프라인 전체 단계를 코드로 명세화하는 기능을 충실히 지원함

Jenkins 장단점 분석

장점: 무한한 확장성

- 수천 개의 **플러그인 생태계**를 바탕으로 모든 외부 도구와 손쉽게 연동됨

단점: 운영 유지 오버헤드

- 플러그인 의존성이 매우 높아 **버전 충돌 및 유지보수 난이도가 크게 상승**함

GitHub Actions 개요

- GitHub 저장소 플랫폼 내부에 추가 설치 없이 **완벽하게 내장된 클라우드 기반 CI/CD 서비스**임
- Push, Pull Request 등 저장소에서 발생하는 **모든 이벤트를 파이프라인 트리거로 활용**할 수 있음
- 자체 인프라를 프로비저닝할 필요 없이, GitHub 호스팅 러너(Runner)를 통해 빌드를 즉시 구동함

GitHub Actions 핵심 구성요소

- **Workflow:** 하나 이상의 작업으로 구성된 전체 단위의 자동화 프로세스 최상위 집합임
- **Job:** 동일한 환경(Runner)에서 반드시 순차적으로 실행되도록 지정된 단계들의 묶음임
- **Step:** 실제 스크립트 명령어를 실행하거나 패키지화된 Action을 직접 호출하는 작업 단위임
- **Action:** 다른 개발자가 제작한 **재사용 가능한 코드 모듈**로 마켓플레이스에서 적용함

GitHub Actions 장단점 분석

장점: 관리 제로 및 높은 연동성

- 서버 인프라를 직접 유지보수할 필요가 없으며, 풍부한 **Marketplace 생태계**를 통해 신속하게 파이프라인을 구축함

단점: 내부망 연동의 복잡성

- 사내망(Private)과 통신하기 위해 자체 호스팅 러너(Self-hosted) 설정이 요구되며, 대규모 사용 시 과금 부담이 존재함

AWS 개발자 도구 (AWS CodeBuild)

완전 관리형 CI 서비스

- 소스 코드를 가져와 컴파일하고 테스트를 수행하여 실행 가능한 소프트웨어 산출물을 자동 생성함

탄력적인 자원 확장성

- 서버 자원의 한계가 없어 **빌드 대기열 정체 없이 다수의 요청을 동시에 병렬 처리**함

AWS CodeDeploy 및 CodePipeline

AWS CodeDeploy

- EC2, Lambda 등 타겟 환경으로 릴리스를 전담하며, **무중단 배포(블루/그린 등) 로직을 내장**함

AWS CodePipeline

- 빌드부터 배포까지의 **전체 파이프라인 흐름을 통합적으로 조율(Orchestration)하고 시각화**함

AWS 개발자 도구 장단점 분석

장점: 완벽한 네이티브 통합

- IAM 및 CloudWatch와 결합되어 강력한 보안 통제가 가능하며, **종량제(Serverless)** 방식의 경제성을 보장함

단점: 클라우드 벤더 종속성

- 타 클라우드 자원과의 혼합 연동이 다소 제한적이며, 결과적으로 **특정 벤더에 대한 강한 종속성(Lock-in)**을 유발함

Section 6. GitOps 패러다임

GitOps의 탄생 배경

- 현대 인프라가 마이크로서비스 아키텍처로 진화함에 따라 **관리 대상 리소스가 기하급수적으로 폭증**함
- 관리자들의 수동 콘솔 조작으로 인해 **클러스터 상태 추적이 불가해지고 인프라 구성의 일관성이 붕괴**됨
- 기존 CI 서버가 외부망에서 접속 권한을 독점하는 구조상, **해킹 시 인프라 전체가 장악되는 보안 위험**이 잔존함

GitOps의 명확한 정의

선언적 인프라 통제

- **Git 저장소를 인프라 정보의 유일한 중심**으로 삼아 배포 및 클러스터 구성을 선언적으로 관리하는 패러다임임

보안 강화 파이프라인

- 운영망 콘솔 접근을 원천 차단하고, 오로지 Git 저장소에 YAML 코드를 **커밋 및 검토하는 행위만으로 배포를 통제**함

DevOps, CI/CD 그리고 GitOps

- **DevOps:** 개발과 운영 부서 간의 긴밀한 소통과 투명한 협업 문화를 강조하는 광범위한 철학적 뼈대임
- **CI/CD:** DevOps 사상을 현실에서 구현하기 위해 빌드와 배포를 파이프라인으로 엮어내는 기술적 실천 도구임
- **GitOps:** CI/CD 체계를 K8s 환경에 맞추어 **보안성이 강화된 Pull 방식으로 고도화하고 인프라 영역까지 확장**한 모델임

GitOps의 핵심 원칙

단일 진실 공급원 확립

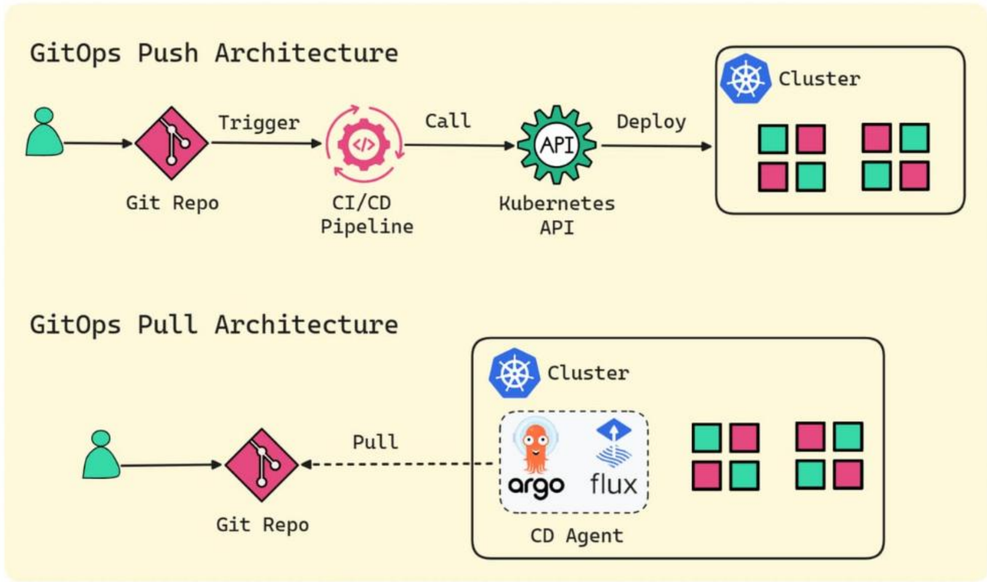
- 시스템 인프라의 기대 상태(명세서)는 **오직 Git 저장소 한 곳에만 선언**되며 엄격히 버전 관리됨

자동 상태 동기화

- 클러스터 실제 상태와 Git의 기대 상태를 에이전트가 상시 감시하고, 차이 발생 시 **강제로 동기화**함

Push 방식 vs Pull 방식 (GitOps)

- **기존 Push (CI/CD):** 외부 CI 플랫폼이 내부망 배포 권한을 직접 쥐고 명령을 주입하므로 공격 표면이 증가함
- **GitOps Pull:** 방화벽 내부에 위치한 에이전트가 외부 Git 저장소를 감시하다가 **변경 사항을 내부로 안전하게 당겨옴(Pull)**
- 서버 접근 크리덴셜을 외부에 일절 노출하지 않아 가장 견고한 보안 아키텍처를 완성함



Argo CD 주요 특징

- 쿠버네티스 생태계에 완벽히 동화된 대표적인 선언형 **GitOps 기반 지속적 배포 제어 플랫폼**임
- 대상 클러스터 내부에 컨트롤러 형태로 설치되어 백그라운드에서 매니페스트 파일 변경 이력을 감시함
- 직관적이고 시각적인 Web UI 대시보드를 제공하여 복잡한 컨테이너 리소스의 동기화 상태를 직관적으로 파악함

Argo CD 운영 강점

Self-Healing (자동 복구)

- 누군가 수동으로 서버 설정을 변경하더라도 즉각 비정상 상태로 진단하고 **Git 기준 코드로 강제 원상 복구**시킴

손쉬운 원클릭 롤백

- 치명적 장애 발생 시 복잡한 조치 없이 **Git 저장소의 커밋 이력을 한 단계 되돌리는 것**만으로 인프라가 즉시 복구됨

Section 7. 실전 배포 아키텍처

실전 배포 파이프라인 흐름

1. Code Push

로컬에서 코드 수정 후 Git 저장소 업로드

2. Webhook

이벤트 발생 감지 시 CI 플랫폼으로 신호 발송

3. Build

CI 서버가 코드 다운 및 컨테이너 이미지 빌드

4. Deploy

운영 타겟 서버에 이미지 배포 및 라우팅 갱신

[별첨] DevSecOps와 Shift Left

전통적 보안 검토 프로세스의 한계

- 기존 환경에서 보안 취약점 검사는 소프트웨어 개발 생명주기의 **가장 마지막 릴리스 직전 단계**에 치우쳐 있었음
- 개발이 모두 종료된 시점에 치명적인 아키텍처 결함이 적발될 경우 대규모 롤백으로 인해 **심각한 일정 병목 현상**이 초래됨
- 배포 속도를 추구하는 DevOps 패러다임과 전통적인 관문형(Gatekeeper) 보안 통제는 구조적 마찰을 빚을 수밖에 없음

DevSecOps 체계로의 전환

- 기획부터 배포에 이르는 파이프라인 전 과정에 **보안 절차를 유기적으로 결합 (내재화)**하는 강력한 문화적 방법론임
- 자동화된 스캔 코드를 파이프라인 로직에 직접 삽입하여 속도 저하를 최소화함
- "보안은 전담 부서만의 임무가 아니라, **개발팀을 포함한 프로젝트 인원 모두의 필수 책임**"으로 인식함

Shift Left 보안의 구체적 구현

시간 역순의 방어망 구축

- 개발 후반부에 국한되어 존재하던 보안 검증을 **극초반부(왼쪽, Shift Left)로 대폭 앞당겨 강력히 적용**함

개발 중 실시간 피드백

- IDE 플러그인이나 커밋 즉시 **취약점을 실시간 경고**하여 사소한 결함조차 메인 브랜치 유입을 차단함

Shift Left의 경제적 가치

100x

수정 비용 기하급수 폭증

선제적 조치의 압도적 효율성

- 설계 단계 결함 수정 비용 대비, 실제 운영망 오픈 후 적발된 취약점의 수습 비용은 **통계상 최대 100배 이상의 손실**을 유발함
- 비용적 측면에서 Shift Left는 조직을 지키는 가장 확실한 예방 투자임

파이프라인 보안 테스트 1: SCA

- **SCA (Software Composition Analysis / 소프트웨어 구성 요소 분석)**
- 애플리케이션 개발에 활용된 **모든 외부 오픈소스 라이브러리와 서드파티 패키지**를 정밀 스캔함
- 빌드 단계에서 종속성 파일들의 상업적 라이선스 위반 유무 및 기 공표된 취약점(CVE) 유무를 즉각 색인함

파이프라인 보안 테스트 2: SAST

- **SAST (Static Application Security Testing / 정적 코드 보안 분석)**
- 소프트웨어 구동 환경 없이 **소스 코드 원문(텍스트) 그 자체의 구조적 논리 결함**을 스캔하여 진단함
- 하드코딩된 패스워드나 인증키, SQL 인젝션 유발 구문 등의 치명적 코딩 실수를 초기에 적발함

파이프라인 보안 테스트 3: DAST

- **DAST (Dynamic Application Security Testing / 동적 애플리케이션 보안 분석)**
- 애플리케이션이 **실제로 실행 중인 런타임 상태**일 때 외부 해커의 블랙박스 시각으로 침투 모의 공격을 수행함
- 코드 분석만으로는 찾기 어려운 세션 가로채기(Hijacking)나 구동 서버 환경설정 오류 등을 심도 있게 검증함

성공적인 DevSecOps 정착 전략 (1/2)

- **과도한 알림 통제:** 도입 초기부터 모든 취약점 경고 시 파이프라인이 마비됨. 최상위 위험 수준 위주로 점진적인 차단 정책을 적용해야 함
- **CI 빌드 속도 보장:** 파이프라인 실행 시간이 비대해지면 안됨. 무거운 보안 스캔 도구는 별도의 야간 배치(Nightly Build)로 분리 운영함

성공적인 DevSecOps 정착 전략 (2/2)

- ✓ **사내 보안 챔피언 지정:** 각 개발 조직 내에 보안 마인드가 우수한 엔지니어를 선발하여 보안 전담 부서와의 협업 창구 역할을 부여함
- ✓ **지속적인 보안 내재화 교육:** 개발자가 취약점 스캐닝 결과를 주도적으로 해석하고 조치할 수 있도록 맞춤형 가이드를 상시 제공함

Q&A 및 마무리

인프라 자동화와 내재화된 보안은 서비스 혁신을 위한 필수 기술 역량입니다.

수고하셨습니다. 모든 발표를 마칩니다.

Image Sources

https://www.frontiersin.org/files/Articles/1670492/xml-images/feduc-10-1670492-g005.webp

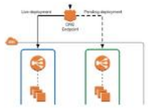
Source: www.frontiersin.org

https://instatus.com/_next/image?url=https%3A%2F%2Fres.cloudinary.com%2Fsup%2Fimage%2Fupload%2Fv1723598981%2Frfwsacyart66nfbawmaj.png&w=3840&q=75

Source: instatus.com

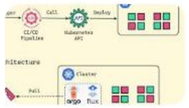
https://valohai.com/assets/img/cicd-abstract.png

Source: valohai.com



https://miro.medium.com/1*EzVYul3GfKXW330RpgBE9A.jpeg

Source: medium.com



https://media.beehiiv.com/cdn-cgi/image/fit=scale-down,quality=80,format=auto,onerror=redirect/uploads/asset/file/25983eb2-0bd3-4588-9723-6e46dc3ab8c7/gitops.png

Source: www.techopsexamples.com

https://image-optimizer.cyberriskalliance.com/unsafe/1920×0/https://files.cyberriskalliance.com/wp-content/uploads/2025/12/122925_ai_cybersecurity.jpg

Source: www.scworld.com