

CO-OP 클라우드트랙

중간고사대체 보고서

컴퓨터학부

20211407

김지호

목차

I. 클라우드 이론 학습

1. Cloud
2. Cloud Model
3. Cloud Native Application
4. Container
5. Container Platform

II. NHN Cloud 실습

1. NHN Cloud 실습환경
2. 인프라구축
3. 컨테이너 플랫폼 구축

III. AWS 실습

1. AWS 소개
2. Amazon EKS로 WEB APP 구축
3. GitLab CI/CD 활용한 DevOps 실습

I - 1. Cloud

1. Cloud 개요

1) 정의

클라우드 컴퓨팅은 인터넷(클라우드)을 통해 서버, 스토리지, 데이터베이스, 네트워킹, 소프트웨어, 분석 기능 등 컴퓨팅 서비스를 제공하는 것을 의미한다. 사용자는 자원을 직접 소유하거나 관리하지 않고, 필요에 따라 유연하게 자원을 활용하고 사용한 만큼 비용을 지불한다.

2) 특징

: 클라우드 컴퓨팅은 기존의 IT 환경과 구별되는 다섯 가지 핵심 특성을 가지고 있으며, 이는 일반적으로 NIST의 정의를 따른다. 이 특성들은 클라우드 서비스의 효율성과 유연성을 극대화하는 기반이 된다.

(1) On-demand self-service : 주문형 셀프서비스

- (i) 사용자의 직접적인 제어와 즉각적인 자원 확보를 의미한다.
- (ii) 사용자는 서비스 제공자와의 별도 상호작용 없이, 필요할 때마다 컴퓨팅 자원을 자동으로 프로비저닝할 수 있다.
- (iii) IT 인프라 확보에 걸리는 시간을 획기적으로 단축, 비즈니스 민첩성 확보한다.

(2) Broad Network Access : 광범위한 네트워크 접근성

- (i) 접속 환경의 제약 없이 클라우드 서비스에 접근 가능하다.
- (ii) 클라우드 서비스는 표준화된 네트워크 메커니즘을 통해 접근 가능하며, 다양한 클라우드 플랫폼에서 이용 가능하다.
- (iii) 어디서든 업무를 수행할 수 있게 하여 이동성, 협업 환경을 보장한다.

(3) Resource Pooling : 리소스 공유

- (i) Multi-tenancy : 하나의 물리적 자원을 다수의 사용자에게 공유하는 방식
- (ii) 서비스 제공자는 다수의 고객을 위해 컴퓨팅 자원을 통합 관리하며, 필요에 따라 이 자원들을 할당한다.
- (iii) 고객은 자원의 정확한 물리적 위치 알 필요 없으며, 논리적 할당량만을 인지한다.
- (iv) 자원 활용률을 극대화하여 경제적 효율성을 높인다.

(4) Rapid Elasticity : 신속한 확장성

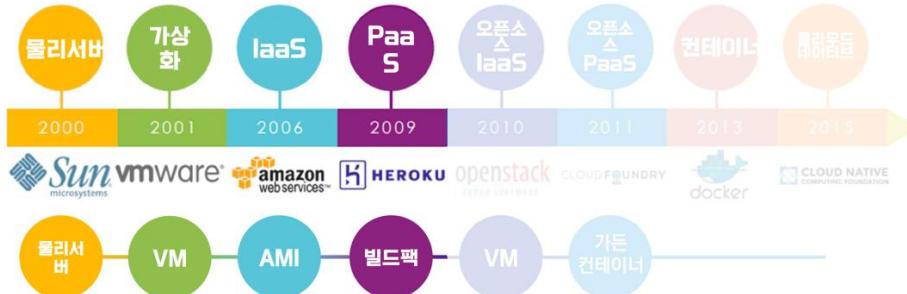
- (i) 부하에 따라 자원을 빠르고 유연하게 확장, 축소한다.
- (ii) 컴퓨팅 자원은 수요에 따라 신속하게(자동으로) 확장되어 배포하고, 축소되어 회수된다.
(사용자는 가능한 자원이 무한하게 보인다.)
- (iii) 트래픽 폭증 등 예측 불가능한 비즈니스 환경에 유연하게 대응할 수 있다.

(5) Measured Service : 측정가능한 서비스

- (i) 사용된 자원량에 비례하여 비용을 지불하는 방식 및 이를 위한 모니터링 체계
- (ii) 사용량은 유형별로 투명하게 측정되어 제공자, 사용자에게 보고된다.
- (iii) Pay-as-you-go(사용한 만큼 지불)하는 종량제 모델 가능하게 함.
- (iv) 사용자에게 비용 투명성을 제공하여 자원 최적화 유도한다.

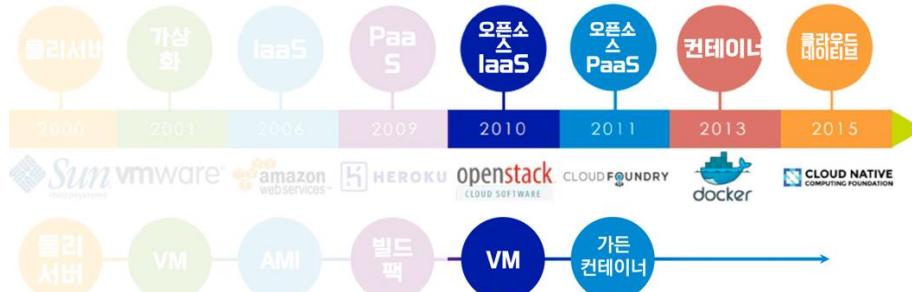
2. Cloud 발전과정

1) 1세대 : 클라우드 시장 최초 생성



가상화의 도입과 함께 IaaS와 초기 PaaS가 태동한 시기이다. 물리 서버 중심 운영에서 2001년경 VMware로 대표되는 하이퍼바이저 가상화가 보편화되며, 2006년 AWS가 대중적 IaaS를 제시했다. 이 과정에서 배포 단위는 물리 장비에서 가상머신으로 이동했고, 운영팀은 AMI(머신 이미지) 같은 템플릿을 활용해 환경을 빠르게 복제, 확장할 수 있게 되었다. 2009년 Heroku 등 초기 PaaS가 빌드팩을 통해 애플리케이션 배포를 단순화했지만, 여전히 게스트 OS를 포함한 VM 단위 운영이 기본이어서 환경이 무겁고 확장, 부팅에 시간이 걸린다는 제약이 있었다.

2) 2세대 : 클라우드 시장 본격적 성장



개방형 생태계의 확산과 컨테이너의 대중화가 나타난 시기이다. 2010년 OpenStack(오픈 IaaS), 2011년 Cloud Foundry(오픈 PaaS)가 등장하면서 특정 사업자에 종속되지 않는 스택에 대한 관심이 커졌고, 2013년 Docker가 표준화된 이미지 포맷, 레지스트리, 레이어링을 제공하며 배포 단위가 VM에서 가벼운 컨테이너로 급격히 이동했다. 2015년 CNCF가 출범하면서 '클라우드 네이티브'라는 용어가 산업 전반의 공감대를 얻었고, 이식성, 속도, 효율이라는 가치가 구체적 도구 체계로 조직화되기 시작했다.

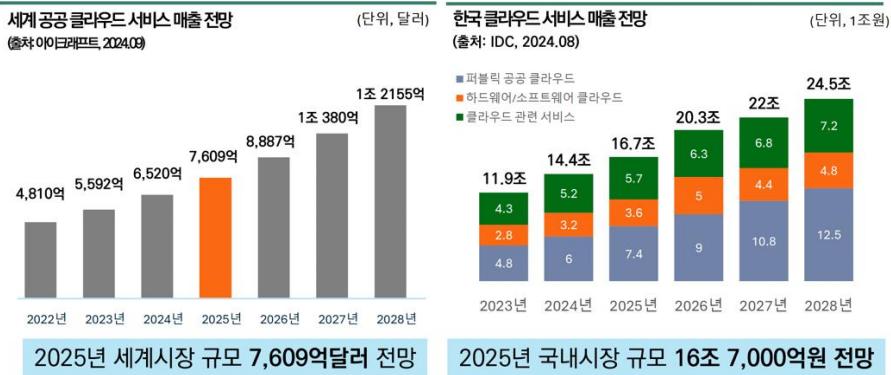
3) 3세대



플랫폼 중심 운영과 멀티 및 하이브리드 엣지로의 확장, 그리고 AI/ML의 본격 편입이 진행된다. 컨테이너 오케스트레이션은 Kubernetes를 표준으로 받아들이며 애플리케이션 관리는 선언적이고 자동화된 방식으로 재구성되었다. 생성형 AI와 대규모 학습, 추론이 확산되며 GPU 자원, 고속 네트워킹, 데이터 파이프라인의 최적화가 인프라 설계의 핵심 과제로 부상했다.

3. Cloud 산업동향

1) 세계시장



슬라이드에 따르면 전 세계 공공 클라우드 서비스 매출은 2025년 7,609억 달러, 2026년 8,887억 달러, 2028년에는 1조 2,155억 달러까지 확대가 전망된다. 흐름상 2022년(4,810억 달러) 이후 연속적 고성장이 이어지며, 슬라이드 상단의 코멘트대로 2025년 전 세계 IT 지출 9.3% 증가 전망이 배경으로 제시되는 것으로 확인된다. 성장 동력은 다음과 같다.

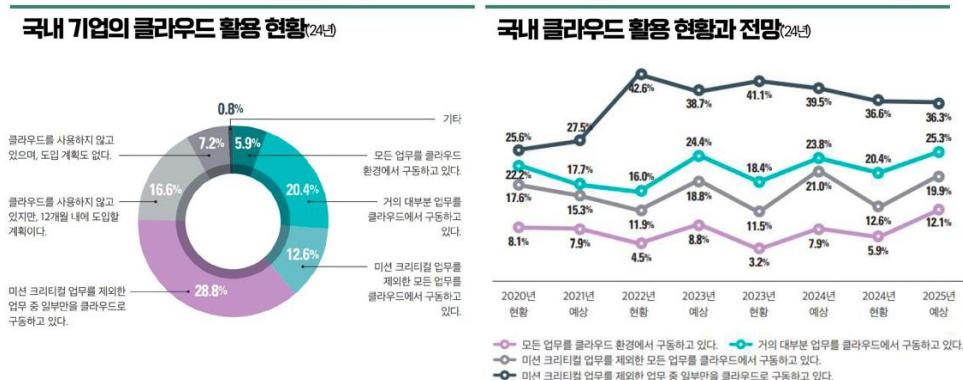
(1) 컨테이너·쿠버네티스 확산에 따른 플랫폼화

(2) AI/ML 워크로드의 본격 편입

(3) 산업별 특화 SaaS의 확대

동시에 비용, 보안, 데이터 주권 이슈가 커지면서 FinOps 및 거버넌스 수요가 함께 증가하는 점도 시사된다.

2) 국내시장



국내의 경우 2025년 16.7조 원, 2026년 20.3조, 2028년 24.5조 원으로 성장 전망이다. 2025년 구성을 보면 슬라이드의 스택 막대 기준으로 퍼블릭 공공 클라우드 약 7.4조, 하드웨어/소프트웨어형 클라우드 약 3.6조, 클라우드 관련 서비스 약 5.7조로 제시되어 서비스 부문 비중이 빠르게 커지는 구조를 보여줍니다. 별도 도넛 차트의 '24년 단면 자료에서는 "12개월 내 도입 계획"(16.6%) 응답이 크며, 완전 전환보다는 업무군을 나눠 단계적으로 이전하는 경향이 뚜렷하다. 우측의 추세 그래프는 2020~2025년 사이 클라우드 활용 전반이 우상향임을 시사한다고 판단된다.

3) 주요기업

(1) 글로벌 IT 기업 : AWS, Aesore, GCP가 전체 인프라의 67% 차지

이 Big3 구조는 기술, 생태계, 가격정책 등에서 규모의 경제가 강하게 작동하고 있음을 의미한다.

(2) 오픈소스 기반 PaaS 각광받음

"치열한 경쟁 속 종속성 문제 대두", "PaaS에 대한 SaaS 종속성 증가"를 지적하여 오픈소스의 급부상을 대안으로 제시하는 것을 볼 수 있다.

4. Cloud 트랜드 : Gartner, 2025 기반

1) 클라우드 불만족 증가

클라우드 도입에 대한 기대와 현실 간 격차로 인해, '클라우드 피로' 현상이 발생하고 있다.

Gartner는 이 현상과 관련하여 2027년까지 전체 조직의 25%가 클라우드 도입에 상당한 불만족을 경험할 것으로 예측했다. 주요 요인으로 다은 3가지가 있다.

(1) TCO 증가 : 예상보다 높은 총소유비용

(2) 설계 복잡성 : 복잡한 아키텍처 설계로 인한 운영 어려움

(3) 벤더 종속성 : 공급업체에 대한 종속성 심화와 이로 인한 운영의 어려움

2) 생성형 AI, ML 도입 가속화

생성형 AI, LLM(대규모 언어 모델), AI Copilot 등 첨단 AI 워크로드 수요가 클라우드 기반으로 빠르게 전환되고 있다. 이러한 AI 중심의 애플리케이션 확산을 위해 기업들은 클라우드 인프라에 대한 의존도를 증가시키고 있다.

3) 멀티클라우드 전략 대중화

기업들이 단일 CSP에 대한 종속성에서 탈피하려는 움직임과 워크로드 분산 관리 수요가 급증하게 되며, 멀티클라우드, 하이브리드 클라우드 전략이 대세로 자리잡고 있다고 한다. 2025년 까지 기업의 65% 이상이 멀티클라우드 전략을 채택할 것으로 전망된다고 한다.

4) 산업별 특화 클라우드 수요 증가

산업별 특성 및 규제 환경을 반영한 맞춤형 클라우드 플랫폼 채택이 증가하고 있다고 한다. 특히, 2028년까지 50%이상의 기업이 산업 맞춤형 클라우드 플랫폼을 채택할 것으로 전망된다.

5) 디지털 주권 전략 강화

Data Localization 요구와 개인정보 규제 강화 추세에 따라, 디지털 주권 확보가 기업의 핵심 전략 요소로 부상하고 있다. 이 디지털 주권 전략 강화의 핵심요소는 다음과 같다

(1) 데이터 로컬라이제이션

(2) 규제 준수

(3) 보안 강화

6) Sustainability : 지속가능성

환경, 사회, ESG 요소가 클라우드 도입의 핵심 고려사항으로 부상했다. 탄소 중립, 에너지 효율 등 ESG 요소가 중요하게 작용하고 있으며, Gartner는 지속가능한 클라우드 전략은 곧 기업 생존전략이라고 강조했다. 이에 대응하여 클라우드 제공자들은 그린 데이터센터 구축 및 재생에너지 기반 인프라 투자를 확대하고 있다고 한다.

5. 공공 Cloud 추진정책

1) 2022년 : 「센터 이전, 통합」 로드맵 가동

정부는 2020년에 수립한 「클라우드 센터 이전·통합 계획」을 바탕으로, 2022년부터 향후 5년간 단계적 전환을 본격 추진하다. 이전 통합의 기본 원칙은 소규모 전산실에서 자체 운영 중인 정보시스템을 공공 또는 민간 클라우드 센터로 이전하는 것이며, 업무·보안 성격에 따라 공공 클라우드 센터와 민간 클라우드 센터로 이원화했다. 공공 부문은 국가안보, 수사, 재판, 내부업무 등 중요 정보를 처리하는 시스템을 공공 클라우드 센터로 우선 이전하고, 대국민 공개용 홈페이지 등 효율성이 우선되는 시스템은 민간 클라우드 센터로 이전하는 구조이다.

슬라이드는 연도별 이전, 통합 대상 물량을 제시하여 물량 중심의 전환이 계획되었음을 보여준다. 또한 보안성 확보를 위해 공공 클라우드 내 보안 영역, 민관협력 영역, 민간 클라우드 내 보안인증 영역, Public 영역을 구분하고 연계하도록 한 점이 특징입니다.

2) 2023년 : 「클라우드 네이티브 전환」으로 정책 축 이동

2023년에는 공공 클라우드 정책이 '클라우드 네이티브 전환'으로 정책 축을 전환한다. 배경으로는 정부 재정 투자 방향 조정, 보안인증제 개편으로 내부행정 시스템의 민간 클라우드 이용 허용, 클라우드 네이티브 기술 일반화 및 SaaS 생태계 성장, 민간 클라우드 이용 확대 등 제도 변화가 제시되었다. 이에 따라 2023년 시범사업을 실시하고, 이를 토대로 2024년부터 '클라우드 네이티브 사업'을 본격화하는 단계 전환을 보여준다. 같은 해 활용모델 사업은 세 갈래로 제시됩니다.

(1) SaaS 모델은 CSAP 등록 SaaS의 기능 적합성을 검토해 업무 적용 대상을 선정한다.

(2) 네이티브 1단계는 컨테이너 관리 플랫폼을 활용하여 현행 응용 재배치하고 필요한 범위에서 일부 코드 수정을 수반한다.

(3) 네이티브 2단계는 안정성, 업무 변화, 개발 효율 등 MSA 도입 평가 기준을 통해 적합 대상을 선별한 뒤 응용 전면 개편까지 포함합니다.

즉, 2023년은 전환의 목표를 "네이티브 방식"으로 재규정하고, 단계적 도입 모델을 정립하였다.

3) 2024년

구분		'24	'25	'26	'27	'28	'29	'30
현행 시스템	전체 전환율 (연도별 목표치)	10% (10%)	30% (20%)	45% (15%)	60% (15%)	75% (15%)	90% (15%)	100% (10%)
	SaaS 적용률	10%	20%	40%	60%	70%	70%	70%
	네이티브 적용률*	10%	30%	70%	75%	80%	85%	90%
신규시스템 (전면 재구축 포함)	네이티브 적용률*	10%	30%	70%	75%	80%	85%	90%

* '24~'26 네이티브 적용률은 디지털플랫폼정부 실행 계획에 따른 목표치

2024년에는 클라우드 네이티브 전환 로드맵이 제시된다. 표에는 기관별로 연도별 전체 전환율 목표가 단계적으로 상향되는 구조가 제시되어 있으며, SaaS 적용률, 네이티브 적용률 등 지표 기반 관리가 도입된다. 이는 단순 이전이 아니라 서비스화(SaaS)와 아키텍처 현대화를 동시에 끌어올리는 이중 트랙을 의미한다. 사업은 두 축으로 구성된다.

(1) 클라우드 네이티브 상세설계

(2) 클라우드 네이티브 전환 구축

즉, 2024년부터는 정량 목표, 예산, 과제 선정, 실행 절차가 체계화되어, 기관별 로드맵 준수와 성과 관리가 본격화된 것으로 요약됩니다.

6. GCP : Google Cloud Platform

1) GCP 소개

GCP는 구글의 방대한 글로벌 인프라를 활용하여 구축된 클라우드 플랫폼이며, 여러 고유한 강점을 바탕으로 서비스를 제공한다.

2) 특징

(1) 글로벌 네트워크 인프라

GCP는 구글의 방대한 글로벌 네트워크 인프라를 기반으로 서비스를 제공한다.

(i) 초고속 네트워크 : 데이터 센터 간 초고속 네트워크 제공

(ii) Private Global Fiber Network : 타 클라우드 대비 낮은 Latency (지연 시간)와 높은 대역폭

(2) AI/ML 최적화 플랫폼

GCP는 AI/ML 워크로드에 **최적화된 플랫폼**을 제공한다.

(i) 통합 머신러닝 플랫폼 : Vertex AI는 모델 학습부터 배포까지 통합적으로 제공하는 End-to-End 머신러닝 플랫폼

(ii) 전용 칩셋 가속 : Google이 설계한 TPU 기반의 AI 전용 칩셋 가속 사용 가능.

(iii) SQL 기반 ML : BigQuery를 통해 SQL 기반 머신러닝을 지원하여, ML 전문가가 아닌 비전문가도 쉽게 사용할 수 있도록 도움

(3) 오픈소스 친화성

GCP는 오픈소스 기반 기술을 활용하고자 하는 기업, 개발자에게 매력적인 환경을 제공한다.

(i) Kubernetes의 초기 단계 개발을 구글이 주도함

(ii) 관리형 Kubernetes (GKE) : GKE(Google Kubernetes Engine)를 통해 성숙한 매니지드 Kubernetes 서비스를 제공

(iii) 오픈소스 프로젝트 주도 : TensorFlow, Apache Beam, Istio 등 다양한 주요 오픈소스를 구글이 주도함

(4) 비용 효율성 및 데이터 분석 강점

(i) Sustained Use Discount : 오래 쓸수록 자동으로 할인되는 지속 사용 할인 제도를 제공

(ii) 저렴한 VM : 스팟 인스턴스처럼 일시적으로 빌려쓰는 VM도 제공함. 이는 최대 80%까지 저렴

(iii) BigQuery : 페타바이트급 데이터 분석을 수초 만에 처리 가능

(iv) 최적화 기능 : 실시간 데이터 스트리밍 분석에 최적화되어 있으며, 데이터 시각화 기능도 강화되어 있음

(5) 하이브리드 및 멀티클라우드 지원

(i) Anthos : Anthos를 활용하여 온프레미스 환경뿐만 아니라 다른 클라우드(AWS, Azure)까지 Kubernetes를 통합 관리할 수 있도록 지원

3) AWS vs. GCP

(1) VPC

VPC는 자체 데이터 센터에서 운영하는 기존 네트워크와 아주 유사한 가상 네트워크를 제공한다.

항목	AWS VPC	GCP VPC
VPC 범위	리전(Region) 단위	글로벌(Global) 단위
서브넷	리전 내	리전 내
리전 간 통신	VPC Peering 필요	기본 제공 (내부 IP 사용)
라우트 테이블	서브넷 별 지정	VPC 전역
방화벽	서브넷 단위 Security Group	VPC 단위 Firewall Rules
Private Access	PrivateLink	Private Google Access
특징	리전 독립성, 세부 컨트롤	글로벌 네트워크, 간편한 확장성

AWS와 GCP는 VPC의 기본 구성 범위에서 가장 큰 차이를 보인다.

AWS는 VPC 자체가 리전 단위인 반면, GCP는 VPC가 글로벌이기 때문에 여러 리전에 걸쳐 하나의 VPC를 사용할 수 있습니다 (Global VPC > Region > Subnet > VM 구조). 이러한 글로벌 구조 덕분에 GCP는 리전 간 통신에 별도의 Peering 없이 내부 IP를 사용하여 기본적으로 통신할 수 있다는 장점이 있다.

(2) 컴퓨팅 서비스

GCP의 Compute Engine은 AWS의 EC2와 거의 동일한 가상 머신 서비스이다. 기능 자체는 유사 하지만, GCP는 더 높은 유연성과 가용성을 제공하는 특이점을 가집니다.

- (i) 유연성 및 가용성 강화 : Compute Engine은 AWS 대비 커스텀 머신 타입 기능을 지원하여 유연성을 높임
- (ii) Live Migration (높은 가용성) : GCP의 특이 기능인 Live Migration은 VM을 중단시키지 않고 실행 중에도 유지보수나 업그레이드가 가능하게 하여 서비스의 중단 없는 높은 가용성을 제공

(3) 스토리지 서비스

GCP의 GCS(Cloud Storage)와 AWS의 S3는 모두 Bucket에 실제 파일과 메타데이터인 Object를 저장하는 구조를 사용한다. 두 서비스 모두 파일 변경 시 이전 버전 저장을 지원하는 객체 단위 버전 관리(Versioning) 기능을 제공한다.

항목	멀티리전 (Multi-Region)	듀얼리전 (Dual-Region)	싱글리전 (Single-Region)
데이터 위치	대륙 단위 복제 (Google 관리)	사용자 지정 2개 리전 복제	하나의 리전에 저장
복제 방식	비동기 다중 리전 복제	동기식 2리전 복제	복제 없음
접근성	글로벌 접근 최적화	뒤 리전에 빠른 접근 최적화	지정 리전 내 빠른 접근
복구 속도	빠름	매우 빠름 (Turbo Replication 가능)	느릴 수 있음 (재해 복구 별도 구성 필요)
가용성	매우 높음	매우 높음	리전 장애 시 위험
주요 사용 사례	전 세계 대상 앱, 콘텐츠 전송	특정 지역 대상 고가용성, 재해 복구	지역 서비스, 비용 최적화 필요시
비용	중간	약간 비쌈	가장 저렴
AWS에 있는가?	O (S3 Multi-Region Access Point)	X	O(기본 S3 리전 저장)

- (i) 멀티리전은 구글이 대륙 단위로 비동기 다중 리전 복제를 수행하여 전 세계 접근이 최적화되고 복구가 빠워 글로벌 서비스, 콘텐츠 전송에 적합하다.
- (ii) 듀얼리전은 사용자가 지정한 두 리전 간 동기식 복제(Turbo Replication)로 RPO/RTO 요구가 높은 금융, 민감 데이터의 고가용성/재해복구에 유리하나 비용이 약간 더 높다.
- (iii) 싱글리전은 한 리전에만 저장해 비용이 가장 저렴하고 지역 규제 준수에 유리하지만 DR은 별도 설계가 필요하므로, 워크로드 특성에 따라 세 옵션을 혼합 설계하시는 것이 합리적이다.

(4) 컨테이너 서비스

- (i) GCP는 AWS의 ECS(Elastic Container Service)와 같은 독자 플랫폼을 만들지 않고, 오픈소스 표준인 Kubernetes와 완전 서버리스(Cloud Run) 두 축으로 컨테이너 전략을 진행한다.
- (ii) GKE는 GCP의 Kubernetes 관리형 클러스터 서비스로, 구글이 Kubernetes의 초기 단계 개발을 주도했다.
 - 관리형 범위 : GKE는 Control Plane과 Worker 노드까지 모두 관리형으로 제공
 - GKE Standard 모드 : Node Pool 단위로 관리 용이, 노드 자동 업그레이드, 자동 수리, 자동 스케일링을 제공
 - GKE Autopilot 모드 : 완전 서버리스 Kubernetes 클러스터입니다. GCP가 워커 노드조차 모두 관리해주기 때문에, 사용자는 VM 탑재, 수, 크기를 신경 쓸 필요가 없어짐
 - 사용 적합 사례 : GKE는 대규모 복잡한 쿠버네티스 워크로드에 최적화되어 있습니다.
 - 복잡한 마이크로서비스 아키텍처를 구축할 때.
 - Stateful App (DB 연동, 캐시 연동) 운영이 필요할 때.
 - 네트워크, 보안, IAM 설정을 아주 세밀하게 컨트롤해야 할 때.
 - GPU 노드 기반 대규모 AI 학습이 필요할 때 (Vertex AI로 갈 수도 있음).
 - 하이브리드 클라우드, 멀티클라우드 배포가 필요할 때 (Anthos 활용).
- (iii) Cloud Run은 AWS의 Fargate와 같은 서비스 컨테이너 실행 플랫폼이다.
 - 컨테이너 이미지만 제공하면 바로 실행 가능하며 완전히 관리형이다.
 - 스케일링 : HTTP 요청 기반으로 자동 스케일링되며, 트래픽에 따라 0개부터 수천 개의 컨테이너까지 확장된다
 - 요금은 사용한 만큼만 초 단위로 과금된다.
 - 사용 적합 사례 : Cloud Run은 서비스 컨테이너 서비스에 최적화되어 있다/
 - 빠르게 컨테이너 기반 API 서비스를 배포하고 싶을 때.
 - ML 모델 서빙(Serving)만 깔끔하게 하고 싶을 때.
 - 비용 최적화를 원할 때 (요청 없으면 0원).
 - 서버 인프라 신경 쓰기 싫을 때
- (iv) Artifact Registry는 AWS의 ECR(Elastic Container Registry)에 대응하는 서비스로, 컨테이너 이미지를 저장, 버전 관리, 배포하는 역할을 한다.
 - 멀티 포맷 지원 : Docker 이미지뿐만 아니라 Maven, npm, Python 패키지 저장도 지원
 - 발전 과정 : 과거에는 Container Registry (GCR)였으나, Artifact Registry로 오면서 리전 단위 저장을 지원하고, 포맷 다양화, 보안 강화, 스캔 기능 강화가 이루어짐
 - 보안 및 통합
 - IAM 기반 세부 권한 제어가 가능하며, VPC Service Controls로 민감 데이터 보호
 - Container Analysis를 통해 저장된 이미지에 대해 취약점 스캔을 할 수 있습니다.
 - GKE, Cloud Run, Vertex AI 등 GCP 전용 서비스와 통합됩니다

(5) AI/ML 플랫폼

(i) AI/ML 플랫폼 주요 영역 비교

영역	AWS SageMaker	GCP Vertex AI
모델 개발/배포	SageMaker Studio, Pipelines	Vertex AI Workbench, Pipelines
자동화 학습(AutoML)	SageMaker Autopilot	Vertex AutoML
하드웨어	GPU 인스턴스	TPU + GPU 인스턴스
LLM 지원	Bedrock	Vertex AI Model Garden (PaLM2, Gemini 등)
데이터 라벨링	Ground Truth	Vertex AI Data Labeling

- 하드웨어 특화 : Vertex AI는 AWS가 제공하는 GPU 인스턴스 외에 Google이 직접 설계한 TPU 기반의 AI 전용 칩셋 가속을 사용할 수 있다는 강점을 가짐
- AutoML 강점 : GCP는 Vertex AutoML을 통해 코드 거의 없이 클릭 몇 번으로 ML 모델 생성이 가능한 기능을 제공. 이는 SageMaker Autopilot과 유사하지만, GCP AutoML은 특히 비전, NLP, 테이블 데이터에 대한 강점
- 비전문가 지원 : Vertex AI 플랫폼과는 별도로, GCP는 BigQuery를 통한 SQL 기반 머신러닝을 지원하여 ML 비전문가도 쉽게 머신러닝을 사용할 수 있도록 함

(ii) LLM 및 Generative AI 지원 비교

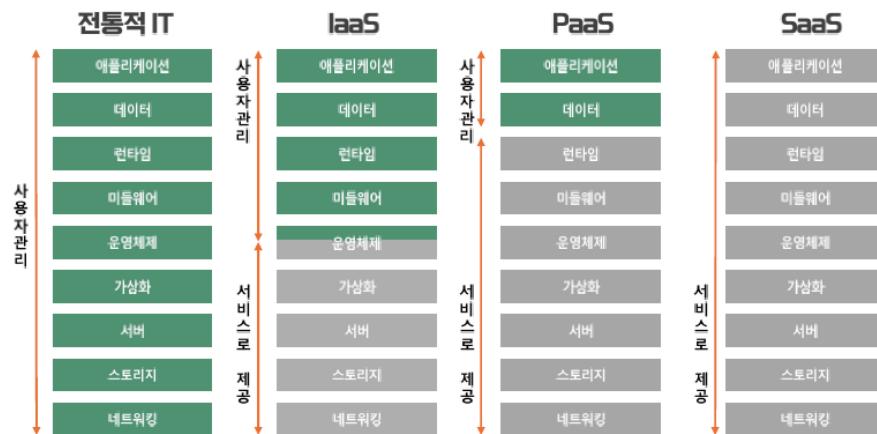
항목	AWS (Bedrock)	GCP (Vertex AI Model Garden)
자체 개발 LLM	거의 없음 (Titan 시리즈 일부)	PaLM 2, Gemini 1.5 (구글 자체 개발 SOTA 모델)
파트너 모델	Anthropic (Claude), Meta (Llama), Mistral, Cohere 등	Anthropic (Claude), Meta (Llama), Mistral, Hugging Face, Cohere 등 + 구글 자체 모델
멀티모달 모델 지원	Claude 3, 일부 Llama3 기반	Gemini 1.5 Pro/Flash (텍스트+이미지+코드 통합 모델)
Fine-tuning 지원	제한적 (Fine-tune Studio 출시)	Fine-tuning API 제공, TPU 최적화 지원
하드웨어	NVIDIA GPU (A100, H100)	TPU + GPU (TPU v4, v5e, H100 지원)
API 사용 편의성	Bedrock API	Vertex AI API (초거대모델 통합 관리)
가격 정책	약간 비싼 편 (특히 Claude, Llama)	상대적으로 저렴 (특히 PaLM2/3 기반 API)
툴 생태계	Bedrock Studio, SageMaker Jump Start	Vertex AI Studio, Colab Enterprise

대규모 언어 모델(LLM)과 생성형 AI 영역에서 AWS는 Bedrock을, GCP는 Vertex AI Model Garden을 통해 지원하며, 특히 GCP는 자체 개발한 SOTA(State-of-the-Art) 모델을 통합 제공하는 데 강점을 가짐

- SOTA 모델 접근성 : Vertex AI Model Garden은 PaLM2, Gemini, Imagen과 같은 Google 자체 LLM을 지원
- 멀티모달 기능 : Vertex AI는 Gemini 1.5 Pro/Flash와 같은 텍스트, 이미지, 코드를 통합 지원하는 멀티모달 모델을 바로 API로 사용할 수 있도록 제공
- 가격 효율성 : Vertex AI API는 상대적으로 저렴한 가격 정책을 제공

I - 2. Cloud Model

1. 서비스 모델 구분



- 전통적 IT는 네트워킹부터 애플리케이션까지 모든 계층을 조직이 직접 구축, 운영한다. 유연성과 통제는 높지만, 초기 투자(CAPEX)와 운영 부담, 확장 리드타임이 큰 편이다.
- IaaS는 공급자가 네트워킹부터 가상화까지 제공하고, 사용자가 OS 부터 앱을 관리한다. 유연성과 커스터마이징이 크지만 운영 책임도 크며, AWS/Azure/Google Cloud 등이 있다.
- PaaS는 인프라에 더해 OS부터 런타임까지 플랫폼이 관리하므로 사용자는 코드와 데이터에 집중합니다. 배포 자동화, 스케일링이 쉬워 개발 생산성이 높고, App Engine, Azure, K-PaaS 같은 서비스가 있다.
- SaaS는 애플리케이션까지 완제품으로 제공되어 사용자는 설정과 이용만 하면 된다. 도입이 가장 빠르고 운영 부담이 최소인 대신 커스터마이, 통제권은 제한적이며, Google Docs 같은 서비스가 있다.

2. 배포모델 구분

1) Public Cloud

서비스 제공자가 구축·운영하는 인프라를 다수 고객이 공유해 사용하며, 사용한 만큼 지불, 유지보수는 공급자가 담당하므로 비용 절감과 탄력 확장이 쉽다. 광대한 리전, 네트워크로 안정성이 높지만, 데이터 주권, 규제 대응은 설계와 거버넌스로 보완해야 한다.

2) Private Cloud

단일 조직 전용(싱글 테넌시) 클라우드로 온프레미스, 호스티드 형태 모두 포함되며, 환경, 보안, 정책을 조직이 원하는 수준으로 세밀하게 제어할 수 있다. 맞춤형 요구에는 유리하나 구축, 운영 책임과 비용 부담이 크므로 표준화, 자동화 체계가 전제되어야 한다.

3) Hybrid Cloud

민감 데이터, 핵심 업무는 프라이빗에 두고, 변동 부하, 대민 서비스는 퍼블릭으로 분산하는 등 둘 이상의 클라우드를 오케스트레이션해 하나의 논리 환경처럼 활용한다. 점진적 마이그레이션과 버스트 처리로 비용 효율, 유연성을 얻되, 네트워크, 보안, ID 일원화가 핵심 과제이다.

4) Multi Cloud

AWS,Azure,GCP 등 서로 다른 CSP를 동시에 활용해 벤더 종속을 완화하고 서비스별 강점을 조합한다. 최신 기술 채택과 가격 경쟁력 확보에 유리하지만, 운영 표준화, 관측, 비용 관리 복잡도가 높아 플랫폼, 거버넌스 체계가 필수이다.

3. IaaS : Infrastructure as a Service

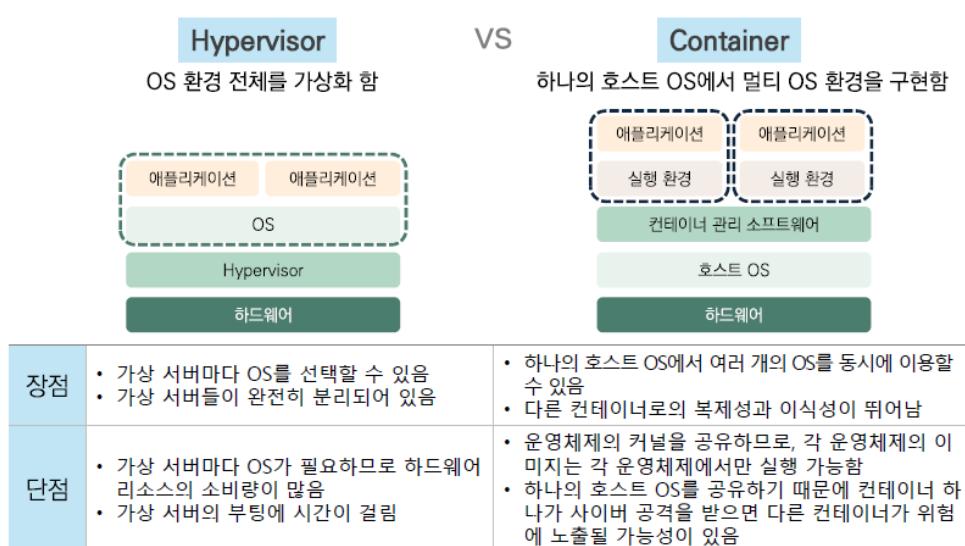
1) 개념

클라우드 서비스 제공자가 infrastructure 자원을 가상화된 서비스로 사용자에게 제공하는 서비스

2) 특징

- (1) 서버 가상화에 해당
- (2) 물리 서버 자원(CPU, 메모리 등)을 분할하여 논리적인 가상 서버를 생성
- (3) IaaS 환경에서는 복수의 OS 환경 사용이 가능 및 가상 서버마다 OS를 선택할 수 있음
- (4) PaaS의 기반이 되는 계층으로, PaaS는 IaaS와 SaaS의 중간 계층의 서비스에 해당

3) 가상화 유형



- 하이퍼바이저는 하드웨어 위에 하이퍼바이저를 두고 게스트 OS를 포함한 전체 OS 스택을 가상화하므로, VM마다 서로 다른 OS 선택과 강한 격리가 가능하지만 OS까지 품어 무겁고 부팅이 느린 특성이 있다.
- 컨테이너는 호스트 OS 커널을 공유하고 컨테이너 런타임이 실행 환경만 분리해 가볍고 빠르며, 하나의 호스트에서 다수 컨테이너를 고밀도로 운용하기에 적합하다. 다만 커널을 공유하는 특성상 커널 레벨 취약점이 전파될 리스크가 있고, 강한 격리가 필요한 워크로드에는 VM이 유리하다.

4. PaaS

1) 개념

IaaS와 SaaS 사이의 중간 계층 서비스로, 애플리케이션을 개발, 실행, 관리할 수 있는 표준화된 플랫폼을 서비스 형태로 제공한다. 즉, IaaS가 서버, 스토리지, 네트워크 등 표준화된 HW의 자동 설치, 구성 등을 제공한다면, PaaS는 여기에 더해 표준화된 SW의 자동 설치, 구성까지 제공하여 개발자가 플랫폼 구축 없이 코드를 바로 올려 사용하게 한다.

2) 특징

- (1) 신속한 개발, 테스트가 가능 : 개발, 테스트, 운영 환경이 미리 표준화되어 컨테이너/VM 위에 즉시 프로비저닝되므로 초기 설정, 패치, 미들웨어 조립 부담이 크게 줄어든다.
- (2) 개발·집중·운영 부담 경감 효과가 크다. 플랫폼이 OS/미들웨어/런타임을 관리하므로 개발자는 코드와 데이터에 집중하고, 배포 파이프라인을 통해 개발 후 배포 라이프사이클이 짧아 DevOps 적용이 용이하다.
- (3) 플랫폼 추상화의 이점과 제약이 공존한다. 표준화, 자동화 덕분에 생산성과 일관성이 높지만, 플랫폼이 제공하는 런타임, 프레임워크 범위를 벗어난 커스터마이징에는 제약이 생길 수 있어 서비스 선택 시 요구사항 적합성 검토가 필요해진다.

5. SaaS

1) 개념

SaaS는 IaaS, PaaS 위에서 소프트웨어 기능 자체를 서비스로 제공하며, SW와 관련 데이터를 중앙에서 관리하고 사용자는 인터넷으로 접속해 기능을 사용한다. 또한 소스코드 수정 없이 설정 중심으로 요구사항을 수용하는 모델로 설계된다.

2) 특징

- (1) 확장성, 가용성을 위해 다수 인스턴스 생성과 분산 데이터 관리, 가상화, 분산 병렬처리를 활용한다.
- (2) 멀티테넌시를 지원해 하나의 애플리케이션을 여러 테넌트가 공유한다. (ASP는 싱글 테넌트)
- (3) 데이터, 버전 관리를 공급자가 수행하고, 사용자는 인터넷 접속만으로 이용하며 사용량 기반 비용 지불이 가능하다.
- (4) 성숙도 측면에서 단일 고객 인스턴스 -> 메타데이터 기반 멀티테넌시 -> 부하분산으로 모든 고객을 지원하는 단계로 발전한다.

I -3. Cloud Native Application

1. Cloud Native

1) 정의

클라우드 네이티브는 어디서든 클라우드의 이점을 최대화하도록 애플리케이션을 설계, 구축, 운영하는 접근이다. 전형적으로 컨테이너, 서비스 메시, 마이크로서비스, 불변 인프라, 선언적 API를 사용하며, 자동화를 통해 적은 노력으로 예측 가능한 변경을 가능케 한다.

2) 발전과정

클라우드 네이티브의 도입은 공공 부문에서 정책적 변화를 통해 가속화되고 있다.

2023년에는 공공 클라우드 전환 계획이 클라우드 네이티브 전환 계획으로 변화되었으며, 행정안전부/NIA는 2023년 클라우드 컴퓨팅 서비스 활용 모델 사업을 추진했다.

2024년에는 클라우드 네이티브 전환 로드맵이 2030년까지 7개년으로 수립되었고, 행정, 공공기관 클라우드 네이티브 전환 사업이 예산 550억으로 본격 추진될 예정이다.

3) 특징 : 전통적인 app과 비교

- (1) 클라우드 네이티브는 애자일, DevOps 문화와 짧고 지속적인 배포(CI/CD)를 전제로, 적은 수고로 예측 가능한 변경을 가능하게 한다.
- (2) 아키텍처 : 모놀리식 -> 마이크로서비스로 전환되어 API 중심의 느슨한 결합을 이루고, 컨테이너로 실행 환경의 일관성과 이식성을 확보한다.
- (3) 인프라 : 온프레미스와 클라우드를 모두 전제로 설계하며, OS 종속성을 줄이고 수평 확장·자동 용량 조정을 기본 가정으로 한다.
- (4) 확장 방식 : 전통처럼 서버 전체를 늘리는 대신, 트래픽이 늘어난 특정 서비스만 컨테이너 단위로 유연하게 확장/축소한다.

결과로, 시장 대응 속도, 안정성, 협업 및 운영, 비용 효율이 동시 향상됩니다.

4) 효과

(1) On Demand Delivery

수요에 따른 용량 자동 조정이 가능하여 자원을 효율적으로 사용한다. 릴리스의 빈도와 속도를 개선하여 제품을 빠르게 혁신하고 개선하는 효과를 가져온다.

(2) Consistency & Continuous

짧고 지속적인 배포(CI/CD)를 사용하여 일관성을 유지한다. 견고한 자동화와 함께 최소한의 수고로 예측 가능한 변경을 수행할 수 있다.

(3) Rolling Update

운영 환경에 최종 결과물을 배포할 때 롤링 업데이트/배포 등을 사용한다. 이는 품질 보증과 안정적인 배포를 가능하게 한다.

(4) Self Recovery

회복성이 있고 가시성을 갖는 느슨하게 결합된 시스템을 사용할 수 있다. 마이크로서비스를 통해 서비스 안정성이 개선되는 효과를 얻는다.

(5) Application Scaling

수평 확장(스케일 아웃) 방식을 사용하여 단일 서비스 증설이 가능하다. 마이크로서비스를 통해 스케일링 용이성이 개선된다.

(6) Portable

컨테이너를 통해 IT 이식성과 유연성을 확보할 수 있다. OS 종속성이 제거되어 온 프레미스 및 클라우드 환경 기준 설계가 가능하다

5) 핵심요소

(1) Microservices

업무상의 기능 또는 역할을 하나의 기능 묶음으로 개발된 컴포넌트이다. 이는 회복성이 있고 느슨하게 결합된 시스템을 구축하는 데 사용되는 전형적인 접근 방식이다

(2) Containers

Cloud Native 소프트웨어의 가장 작은 단위이다. 가상화된 운영체제 위에서 애플리케이션의 독립적인 실행에 필요한 파일들을 모은 패키지이다

(3) DevOps

개발과 운영 프로세스의 통합 및 자동화를 통해 신속한 서비스를 지원하는 통합 체계이다. 개발자와 운영자가 협업하여 짧은 주기 내 신뢰성 있는 SW를 헐리즈할 수 있는 문화와 환경을 의미한다

(4) CI/CD

지속적인 통합, 서비스 제공, 배포의 자동화를 통해 고객에게 짧은 주기로 App 서비스를 제공하는 방식이다. 이는 DevOps를 실현하기 위해 필수적인 방법 요소이며, DevOps 라이프사이클을 실현하는 파이프라인이다

2. MSA : Micro Service Architecture

1) 개념

マイクロ서비스는 업무상의 기능 또는 역할을 하나의 기능 묶음으로 개발된 컴포넌트이다. MSA 아키텍처는 전통적인 모놀리식 아키텍처와 달리 느슨한 결합을 특징으로 하며, Microservice / API 기반 통신을 사용한다. 마이크로서비스는 REST API 등을 통하여 서비스들의 기능을 호출한다

2) 장점

- (1) 느슨한 결합 : 강한 결합을 특징으로 하는 전통적인 단일(Monolithic) 아키텍처에 비해 느슨한 결합이 가능하다.
- (2) 안정성 및 확장성 개선 : 마이크로서비스를 통해 서비스 안정성이 개선되며, 스케일링 용이성이 증가한다.
- (3) 독립적 확장 : 특정 업무 트래픽 증가 시, 전체 서버를 증설할 필요 없이 문제가 되는 단일 서비스만 증설할 수 있다. 이는 수평 확장(스케일 아웃)을 용이하게 한다.
- (4) 신기술 적용 용이성 : 서비스별로 신기술 적용이 용이하다 (예: C/JAVA, DB 등).

3) 단점

- (1) 개발·운영 복잡성 증가
서비스 수, 배포 단위, 네트워크 경로가 늘어나 전반적 관리 난도 상승했다. 서비스 디스커버리, 설정관리, 관측성 표준화가 필수가 된다.
- (2) 통합 테스트의 어려움
다수 서비스, 데이터 저장소가 연쇄적으로 얹혀 E2E 시나리오 검증이 어렵다. 계약기반 테스트, 테스트 데이터 관리 체계가 필요해진다.
- (3) 배포·운영 자동화 필요
수백 컴포넌트를 수동 배포할 수 없어 CI/CD·IaC·카나리/블루그린이 전제이다. 자동화 미흡 시 장애, 운영비 증가로 직결된다
- (4) 중복성(Duplication)
인증, 캐시 등 공통 기능이 여러 서비스에 중복 구현될 위험이 있다. 플랫폼/공통 라이브러리로 단일화하지 않으면 유지보수 비용이 커진다
- (5) 트랜잭션 처리 어려움
분산 환경에서 ACID 일관성 유지가 어렵고 SAGA·벤트 기반 보상 트랜잭션 설계가 필요하다. 재시도, 자연 허용치를 함께 설계해야 한다.
- (6) 디버깅 어려움
호출이 서비스 경계를 넘나들어 원인 추적이 힘들어진다. 분산 트레이싱/집중 로그/메트릭 기반 관측성을 선행 구축해야 한다.

4) 도입 시 검토사항

- (1) 조직·문화/DevOps 준비도
자동화 파이프라인·플랫폼 엔지니어링·온콜 체계를 운영할 역량이 있는가가 핵심이다. 준비가 부족하면 복잡성만 늘고 효과가 반감된다.
- (2) 도메인 경계·데이터 분리 전략
Bounded Context·서비스별 DB·트랜잭션 전략 등 데이터 관리 대안이 마련돼 있는가가 관건이다. 미비하면 일관성·성능 이슈가 상시화된다.

5) The Art of Scalability

(1) 개요

マイ클 T. 피셔가 제시한 3 차원(X·Y·Z) 확장성 모델로, 서비스 확장을 “동일 인스턴스 복제(X)·기능 분해(Y)·데이터 분할(Z)”의 세 축으로 체계화한 프레임워크이다. 각 축을 단독/조합해 쓰며 시스템 성장 단계와 병목 위치에 따라 선택한다.

(2) X 축

- 수평 확장 : 동일한 애플리케이션 인스턴스를 여러 대로 늘리고 로드밸런싱으로 트래픽을 분산한다.
- 장점 : 적용이 빠르고 초기 효과가 크다.
- 한계 : 상태 저장 워크로드/단일 DB 병목을 스스로 해결하지 못한다.
- 예시 : 쿠버네티스 Replica 증가, ASG 확장

(3) Y 축

- 서비스 분할(기능 분해) : 시스템을 도메인별 독립 서비스로 쪼개 각 서비스를 독립 배포/독립 확장한다.
- 장점 : 특정 기능만 선택적으로 확장하고 폴리글롯 스택을 허용해 민첩성을 높인다.
- 단점 : 분산 시스템 복잡성(통신/일관성 관리)이 커진다.

(4) Z 축

- 데이터 분할(샤딩/파티셔닝) : 사용자 ID/지역 등 규칙으로 데이터를 여러 파티션에 나눠 저장/처리한다.
- 장점 : 대용량 데이터에서 처리량과 장애 격리성을 높인다.
- 단점 : 분할 키 설계/재샤딩/일관성 관리가 어렵다.
- 예시 : MongoDB/Elasticsearch 샤딩, 지역별 데이터 센터 분산

(5) 세 축의 상호작용

- X 축으로 급한 처리량을 해결 → 트래픽/기능 특성이 갈라지면 Y 축으로 분해 → 데이터가 임계에 도달하면 Z 축으로 분할

(6) 조합 패턴

- X+Y : 마이크로서비스로 분리한 뒤 각 서비스를 수평 확장. (팀별/기능별 최적 확장)
- Y+Z : 서비스별 데이터 독립과 샤딩을 결합. (도메인 경계+데이터 경계 정렬)
- X+Z : 동일 앱 복제와 데이터 샤딩을 함께 적용. (API 서버 다중화 + DB 샤딩)

3. Container Orchestration

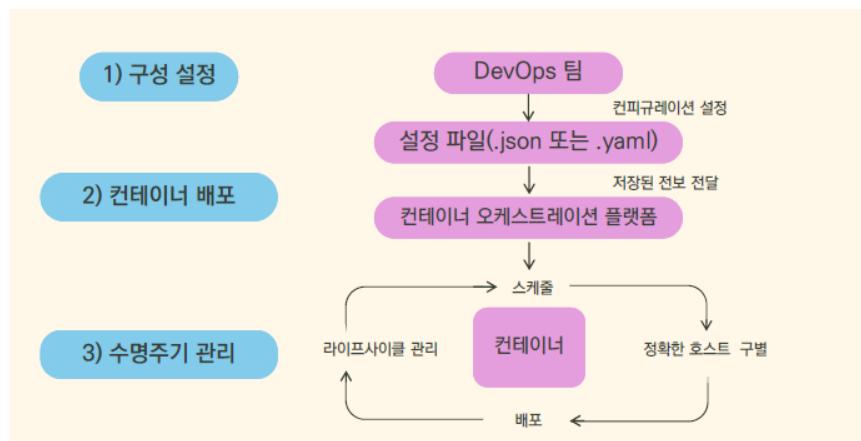
1) Container 특징

컨테이너는 애플리케이션과 의존 라이브러리(런타임, 바이너리 등)를 하나로 묶어 호스트 OS 커널을 공유한 채 독립 실행하는 가장 작은 배포 단위이다. 서버 가상화(Guest OS 포함)보다 가볍고 기동이 빨라 고밀도 배치, 신속 확장에 유리하며, VM 위에서도 동일하게 배포할 수 있어 환경 일관성과 이식성이 높다. 베어메탈→서버 가상화(Hypervisor)→컨테이너 가상화(OS 레벨)로 진화하면서, 하드웨어 자원 분할이 논리 컨테이너 단위까지 내려왔고, CaaS/PaaS/FaaS 같은 상위 서비스가 이를 토대로 구현된다.

2) 개념

컨테이너 오케스트레이션은 다수 컨테이너의 프로비저닝·배포·네트워킹·확장·가용성·수명주기를 선언적 방식으로 자동화하는 제어 계층이다. 소수 컨테이너는 수동으로 관리 가능하지만, 대규모 애플리케이션은 자동화 없이는 불가능하므로 Kubernetes가 사실상 표준 플랫폼으로 활용된다. 오케스트레이터는 “원하는 상태(Desired State)”를 받아 배치, 자가치유 영구 스토리지 연동 등을 포괄적으로 담당한다.

3) 동작원리



- (1) DevOps 팀이 YAML/JSON 매니페스트로 배포 단위(예: Deployment/Pod), 서비스 노출, 리소스 요구, 헬스체크, 구성/시크릿 등을 기술한다. 이 정의가 오케스트레이션 플랫폼으로 전달되면 API 서버/컨트롤러가 “원하는 상태”를 저장한다.
- (2) 스케줄러가 사용 가능한 노드의 자원, 친화도, 테인트 등을 고려해 파드를 배치하고, 각 노드의 에이전트(kubelet)가 이미지 풀 -> 컨테이너 실행을 수행한다. 이후 서비스 네트워킹 / 로드밸런싱으로 통신이 연결된다.
- (3) 운영 중에는 라이프사이클 관리가 자동화됩니다. Liveness/Readiness로 헬스체크를 하고, 장애 시 재시작, 재스케줄링하는 자가치유, 부하에 따른 오토스케일링(HPA), 롤링 업데이트/즉시 롤백으로 무중단 배포를 보장한다.

4. DevOps

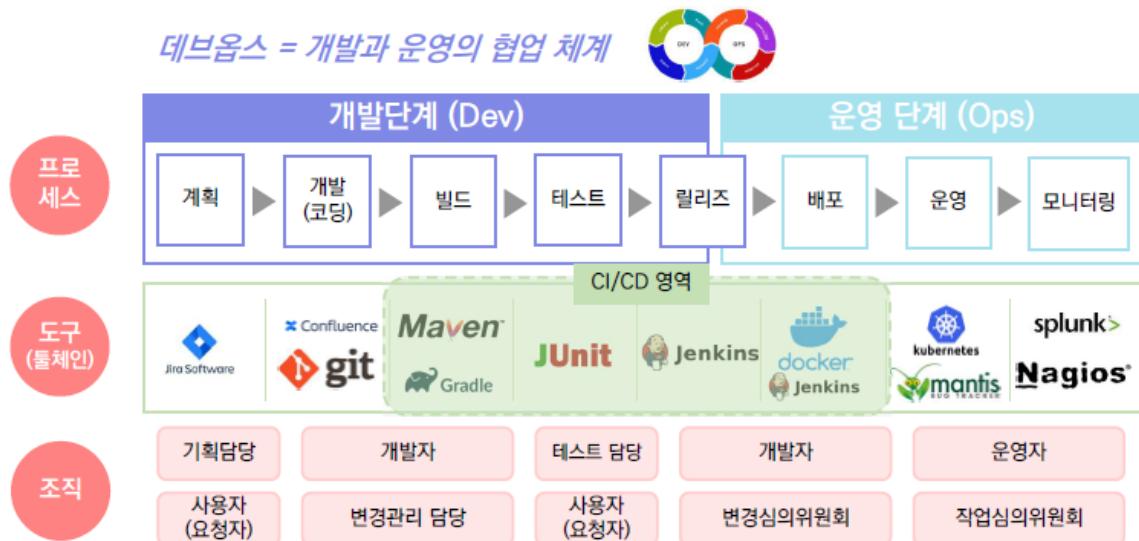
1) 정의

DevOps는 개발(Dev)과 운영(Ops)의 프로세스, 도구, 조직문화를 통합하여 자동화를 기반으로 변경을 더 빠르고 지속적으로 배포하게 만드는 협업 체계이다.

2) 등장배경

폭포수 방식은 단계 간 대기, 재작업이 커 릴리스 주기가 길고 시장 대응이 느린 한계가 있었다. 애자일 전환과 클라우드/マイ크로서비스 확산으로 짧은 주기의 안정적 배포와 자동화, 협업이 필수 과제가 되며 DevOps가 대두되었다.

3) 개념



DevOps의 개념은 프로세스, 도구, 조직을 하나의 흐름으로 통합하는 데 있다. 프로세스 측면에서 계획→개발(코딩)→빌드→테스트→릴리스→배포→운영→모니터링이 CI/CD 파이프라인으로 연속 연결되어, 변경이 자동으로 검증/전달된다. 도구 측면에서는 형상관리(Git), 빌드/테스트(Maven/Gradle, JUnit), CI/Jenkins, 컨테이너(Docker), 오케스트레이션(Kubernetes), 관측/모니터링(Splunk, Nagios 등)을 체인으로 연계해 표준화와 자동화를 구현한다. 조직 측면에서는 개발/운영/품질/보안 등이 공동책임 구조로 협업하며, 역할 간 커뮤니케이션을 통해 동일한 파이프라인과 데이터를 공유한다.

4) 효과

- (1) 속도/민첩성 : 릴리스 빈도와 속도가 향상되어 시장 변화에 더 빨리 적응하고 비즈니스 가치를 신속히 제공한다.
- (2) 안전성 : 지속적 통합과 점진 배포, 표준화된 테스트가 결합되어 변경 품질이 보장되고 롤백/복구가 용이하다.
- (3) 확장/보안 : 규정 준수, 구성관리, 접근제어가 자동화되어 규모가 커져도 안정적으로 운영하며 보안을 지속 유지할 수 있다.
- (4) 협업 강화 : 개발과 운영이 절차와 정보를 공유하여 팀 간 경계가 낮아지고, 공동 책임 기반의 효율적인 조직운영이 가능해진다.

5. CI/CD

1) 정의

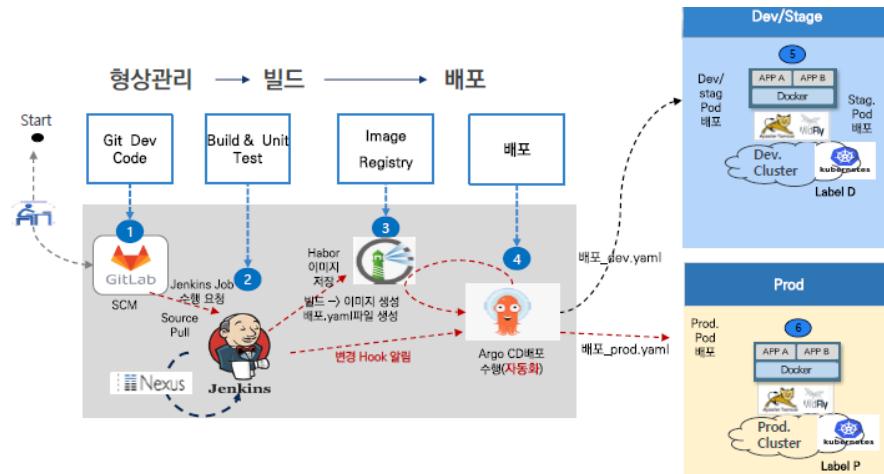
CI/CD는 “개발과 운영 프로세스의 통합·자동화를 통해 신속하게 서비스를 제공”하기 위한 체계로, 도구와 절차, 조직문화까지 포괄한다. 여기서 CI(지속적 통합)는 코드 변경을 빌드·테스트 후 공용 리포지토리에 자주 병합하는 활동이고, CD는 두 갈래로 나뉘어 지속적 제공은 테스트를 거친 산출물을 리포지토리에 자동 업로드하는 단계, 지속적 배포는 그 변경을 운영(프로덕션) 환경까지 자동 릴리스하는 단계이다.

- (1) CI(지속적 통합) : 새 코드 변경을 정기적으로 빌드·테스트하고 공용 저장소에 병합한다. 이렇게 해야 이후 단계의 자동화를 안정적으로 연결할 수 있다.
- (2) CD(지속적 제공, Delivery) : 통합된 변경이 테스트를 통과하면 산출물이 저장소에 자동 업로드된다. CD(지속적 배포, Deployment)는 여기서 더 나아가 프로덕션까지 자동 릴리스가 이뤄지는 상태를 말한다.

2) 파이프라인

전형적인 파이프라인은 코드 커밋 -> 빌드 -> 테스트(단위/통합) -> 릴리즈 패키징 -> 배포의 일련 과정을 자동화한다. 실무에서는 운영과 유사한 스테이징 단계를 거쳐 품질을 검증하고, 최종 프로덕션 배포 시 블루/그린 롤링 업데이트 같은 전략을 사용한다. 또한 각 단계는 설정에 따라 자동/수동 게이트를 둘 수 있고, 빠른 피드백을 위해 빌드·테스트를 여러 단계로 분리해 병렬화한다.

3) PaaS 기반 CI/CD 자동화



- (1) 형상관리(GitLab)에 코드가 푸시되면 웹훅(Hook)으로 Jenkins CI 잡이 트리거되고, 빌드/유닛테스트를 수행한다. 성공 시 컨테이너 이미지를 생성해 이미지 레지스트리(Harbor)에 푸시하고, 환경별 배포 매니페스트(yaml)를 생성한다.
- (2) 변경 알림을 받은 Argo CD가 매니페스트를 감시/동기화하여 Dev/Stage 클러스터에 먼저 자동 배포하고, 검증 후 Prod 클러스터로 승격 배포한다.

이 과정은 PaaS 가 표준화된 런타임·레지스트리·배포 도구를 제공하므로 쉽게 자동화됩니다.

4) DevOps 와의 관계

- (1) DevOps는 조직/프로세스/도구 전반의 협업 문화이고, CI/CD는 DevOps를 실현하는 필수 자동화 방법이다. 파이프라인은 DevOps 라이프사이클과 직접 연결되어 빈번한 소규모 릴리스를 가능케 한다.
- (2) 실무적으로 DevOps는 애자일과 CI/CD를 축으로 짧은 주기로 고객에게 서비스를 제공하며, 품질 향상과 배포 시간 절감을 달성한다.

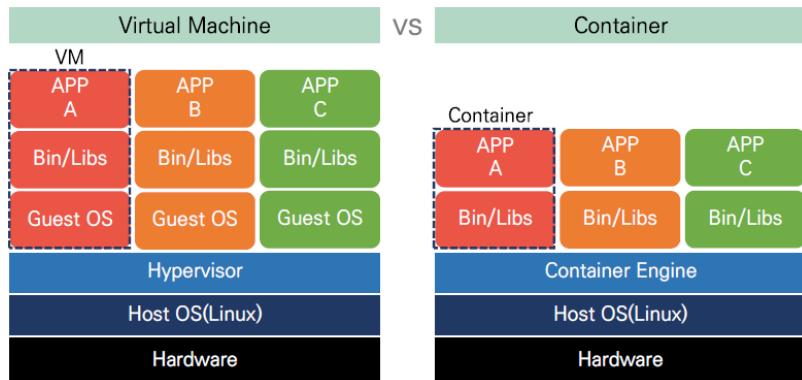
I -4. Container 기술 / 도구

1. Container

1) 정의

컨테이너는 가상화된 운영체제 위에서 애플리케이션 실행에 필요한 소스/바이너리, 라이브러리, 설정을 하나의 패키지(이미지)로 묶어, 어디서나 동일하게 실행되도록 만든 배포 단위이다. 개발/테스트/운영 환경이 달라도 이미지로부터 컨테이너를 생성하면 동일한 결과를 재현할 수 있어, “앱 코드와 종속성을 함께 번들로 제공”할 수 있다.

2) Hypervisor vs. Container



하이퍼바이저 기반 가상머신(VM)은 각 인스턴스마다 게스트 OS를 포함해 하드웨어를 가상화하므로 격리는 강하지만 무겁고 기동이 느리다. 반면 컨테이너는 호스트 OS 커널을 공유하는 OS 레벨 가상화로, 게스트 OS가 없어 훨씬 가볍고 빠르게 많은 인스턴스를 고밀도로 배치할 수 있다.

3) 특징

- (1) 빠른 생성/배포 : 컨테이너 이미지를 손쉽게 만들고 재사용하여 애플리케이션을 기민하게 배포할 수 있다.
- (2) 지속적 개발/배포 적합: 이미지를 기준으로 빌드/릴리스를 표준화하므로 CI/CD 파이프라인에 자연스럽게 연결된다.
- (3) 관심사 분리 : 빌드 시 앱과 종속성을 이미지에 고정하기 때문에 개발과 운영의 경계가 명확해진다.
- (4) 일관성/이식성: 노트북/온프레姆/퍼블릭 클라우드 어디서나 동일하게 구동되어 환경 간 차이를 최소화한다.
- (5) 가시성/격리: 헬스체크, 로그/메트릭 수집이 용이하고, 네임스페이스/cgroups 격리로 예측 가능한 성능을 제공한다.

4) Container Engine

컨테이너 엔진은 컨테이너를 관리하기 위한 API/CLI를 제공하는 소프트웨어로, 사용자의 명령을 받아 이미지를 Pull하고 실행 메타데이터를 작성한 뒤 런타임에 전달하여 생성/시작/중지 등을 orchestrate 한다.

5) Container Runtime

컨테이너 런타임은 엔진으로부터 전달받은 루트 파일시스템과 스펙(spec) 정보를 사용해 실제로 격리된 프로세스를 생성/실행/종료하는 저수준 실행기이다. 표준 OCI를 따르는 runC가 가장 일반적이며, 상위 런타임/데몬인 containerd 나 CRI-O 가 내부적으로 runC를 호출해 컨테이너를 구동한다.

2. Docker

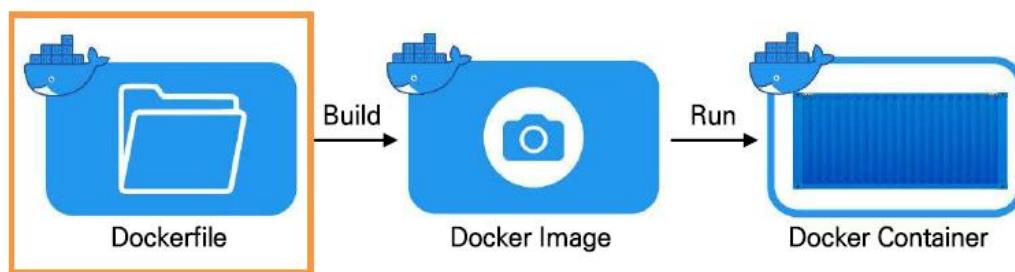
1) 정의

Docker는 애플리케이션과 실행환경을 컨테이너 형태로 패키징/배포/실행할 수 있게 해주는 오픈소스 컨테이너 플랫폼이다. 개발자는 동일한 이미지로 어디서나 같은 방식으로 실행되도록 보장받고, 운영자는 표준화된 도구로 배포 및 수명주기를 관리한다.

2) 특징

- (1) 빠른 기동과 낮은 오버헤드 : 게스트 OS가 없는 OS-레벨 가상화라 컨테이너 생성 및 시작이 빠르고 자원 소모가 적다.
- (2) 손쉬운 배포와 이미지 공유 : 이미지를 한 번 빌드하면 레지스트리에 푸시/풀하여 팀 간 재사용이 쉽다.
- (3) 가벼운 용량/독립성 : 앱과 종속성을 이미지에 고정해 환경 차이를 제거하고, 컨테이너 간 격리로 충돌을 줄인다.
- (4) 보안 통제 강화 : 네임스페이스/cgroups 등 커널 격리와 최소 실행 단위(컨테이너)로 표면적을 줄여 운영 리스크를 낮춘다.
- (5) CI/CD 친화성 : 코드 변경 -> 빌드 -> 테스트 -> 배포의 표준 파이프라인에 자연스럽게 결합되어 릴리스 빈도와 품질을 동시에 끌어올린다.

3) 구성



(1) Dockerfile

컨테이너 이미지를 생성하기 위한 선언적 스크립트이다. 베이스 이미지, 패키지 설치, 파일 복사, 환경변수, 실행 명령(CMD/ENTRYPOINT) 등을 정의하며, docker build 시 이 지시문을 순차 실행해 레이어를 만든다. 결과 이미지는 재현 가능하고 캐시를 활용해 빠르게 재빌드된다.

(2) Docker Image

Dockerfile을 빌드해 얻는 불변(immutable) 실행 패키지입니다. 애플리케이션과 라이브러리 설정이 층(layer)으로 쌓여 있으며, 레지스트리(예: Docker Hub/Harbor)에 태그(tag)로 저장 및 공유된다. 동일 이미지로부터 생성한 컨테이너는 어디서나 같은 결과를 보장한다.

(3) Docker Container

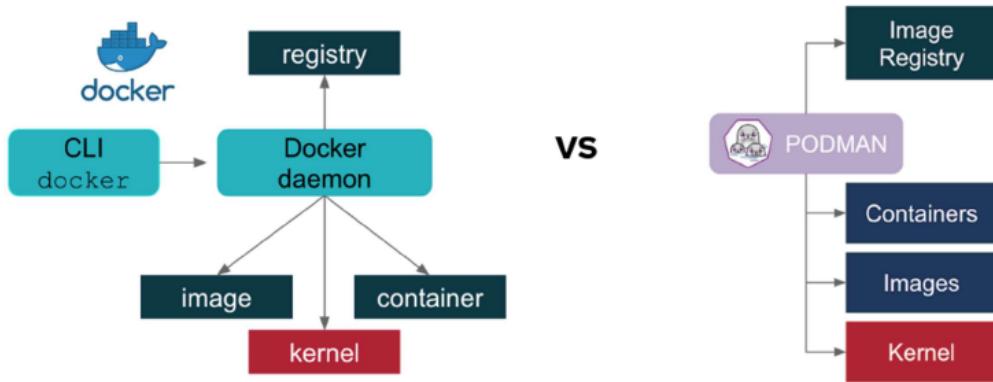
이미지를 실행(run) 해서 만들어지는 격리된 프로세스 인스턴스이다. 수명주기(생성/시작/중지/삭제)와 네트워크, 볼륨을 엔진이 관리하며, 다수의 컨테이너를 병렬로 띄워 확장하거나 롤링 업데이트로 무중단 배포를 구현할 수 있다.

4) Podman

(1) 정의

Podman은 리눅스에서 컨테이너와 컨테이너 이미지를 실행/관리/배포하도록 설계된 데몬리스(daemonless) 컨테이너 엔진이다. 별도의 상주 데몬 없이 명령이 들어올 때마다 프로세스를 생성해 동작하며, 루트리스(rootless) 실행과 Pod 단위 관리를 지원해 접근성과 보안성을 높인다. 또한 OCI(Open Container Initiative) 규격을 따르고 Docker와 거의 동일한 CLI 호환성을 제공하며, 관리 자동화를 위한 RESTful API도 제공한다.

(2) Docker vs. Podman



- 아키텍쳐 차이
Docker는 사용자의 CLI가 dockerd(데몬)에 요청을 보내고, 데몬이 이미지/컨테이너를 통합 관리하는 클라이언트-서버 구조이다. 반면 Podman은 데몬 없이 CLI가 직접 컨테이너, 이미지 레이어와 커널 기능을 호출해 작업을 수행하므로 단일 데몬에 대한 의존과 단일 장애 지점을 줄인다.
- 보안과 운영 모델 차이
Podman은 기본적으로 루트리스 실행을 지원해 다른 사용자/시스템 리소스와의 격리를 강화하고, 컨테이너를 사용자 프로세스 트리에서 관리하기 쉬운 구조이다. Docker도 루트리스 모드를 제공하지만, 교안의 관점에서는 데몬 중심 운영이 기본 흐름이다.
- 호환성과 생태계 측면에서는 두 엔진 모두 OCI 이미지/레지스트리를 사용하므로 동일한 레지스트리의 이미지를 pull/run 할 수 있고, build/pull/run 등 주요 CLI가 유사하다. 즉, 워크플로는 비슷하지만 Docker는 데몬을 통해, Podman은 데몬 없이 각각 컨테이너와 이미지를 관리한다는 점이 핵심 차이이다.

5) Docker CLI

(1) 환경/인증

\$ docker version

명령 의미

Docker 버전 정보를 표시

기본 형태

docker version [options]

\$ docker login(또는 logout)

명령 의미

컨테이너 레지스트리에 로그인/로그아웃

기본 형태

docker login [options] [registry]

(2) 이미지 생성/가공

\$ docker build

명령 의미

컨테이너 파일을 사용하여 컨테이너 이미지 빌드

기본 형태

docker build [options] [context]

\$ docker tag

명령 의미

로컬 이미지에 새로운 이름 추가

기본 형태

docker tag *image[:tag]* [*target-name[:tag]*]...
[options]

(3) 레지스트리 전송

\$ docker pull

명령 의미

레지스트리에서 이미지 가져오기

기본 형태

docker pull [options] source [source...]

\$ docker push

명령 의미

이미지, 매니페스트 목록 또는 이미지 인덱스를 local 스토리지에서 다른 곳으로 보내기

기본 형태

docker push [options] image [destination]

(4) 컨테이너 실행/접속

\$ docker run

명령 의미

컨테이너 이미지로 새 컨테이너 실행

기본 형태

docker run [options] image [command[arg ...]]

\$ docker exec

명령 의미

실행 중인 컨테이너에서 명령 실행

기본 형태

docker exec [options] container [command[arg ...]]

(5) 컨테이너 중지/정리

\$ docker stop

명령 의미

실행 중인 컨테이너 중지

기본 형태

docker stop [options] container ...

\$ docker kill

명령 의미

실행 중인 컨테이너 즉시 종료

\$ docker rm

명령 의미

컨테이너 삭제

기본 형태

docker kill [options] [container ...]

docker rm [options] container

(6) 상태 조회

\$ docker ps

명령 의미

컨테이너에 대한 정보 출력

기본 형태

docker ps [options]

3. Kubernetes

1) 정의

쿠버네티스(Kubernetes)는 컨테이너화된 애플리케이션을 자동으로 배포/스케일링/복구하고 서비스 단위로 관리해 주는 오픈소스 오케스트레이션 플랫폼이다. 컨테이너 워크로드와 서비스를 이식성 있게 운영하도록 추상화/자동화된 제어면을 제공한다.

2) 특징

(1) 서비스 디스커버리와 로드밸런싱

클러스터 내부/외부에서 서비스 이름으로 접근하고, 트래픽을 파드에 고르게 분산한다.

(2) 스토리지 오케스트레이션

다양한 스토리지(Cloud/온프레미스)를 선언적으로 마운트/관리한다.

(3) 자동 롤아웃/롤백

배포 전략(rolling 등)을 통해 무중단 업데이트와 이전 버전 복구를 자동화한다.

(4) 자동화된 bin packing

노드 자원을 고려해 파드를 효율적으로 스케줄링해 자원 활용을 극대화한다.

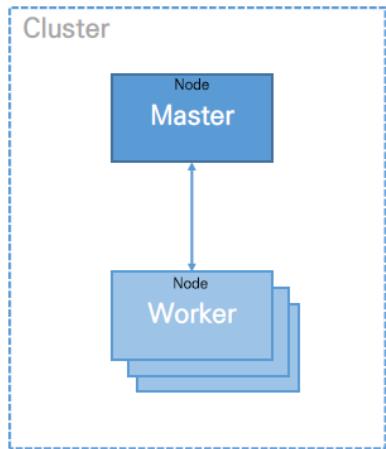
(5) 셀프 힐링(self-healing)

장애 파드를 재시작·재스케줄링하고 헬스체크 실패 시 교체한다.

(6) 시크릿/설정 관리

시크릿과 ConfigMap 을 분리 보관/주입해 보안성과 이식성을 높인다.

3) 구성요소



(1) 클러스터(Cluster)

컨테이너화된 애플리케이션을 실행하는 노드들의 집합으로, 하나의 논리적 시스템으로 동작한다.

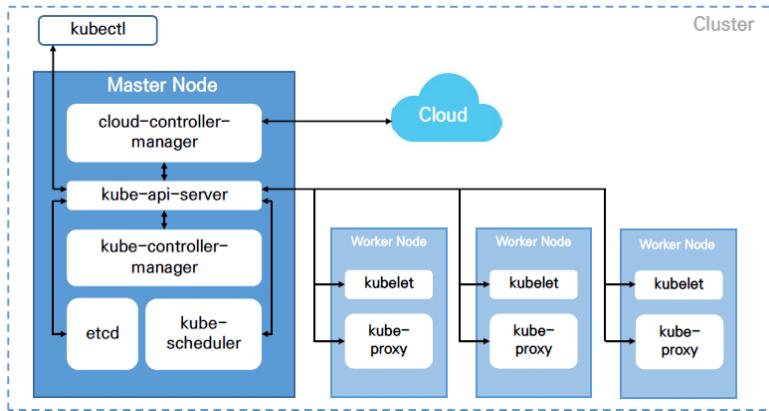
(2) 마스터/컨트롤 플레인(Master/Control Plane)

클러스터 전반을 제어하는 두뇌로서 API 서버/스케줄러/컨트롤러 등이 배포/스케일링/상태 수렴을 관리한다.

(3) 워커 노드(Worker)

실제 컨테이너가 실행되는 머신으로, kubelet/컨테이너 런타임이 파드를 실행하고 kube-proxy 가 서비스 네트워킹을 담당한다.

4) Kubernetes Cluster 구성



(1) kube-api-server

클러스터의 모든 요청이 통과하는 중앙 REST API 엔드포인트로, 인증/인가/어드미션을 수행하고 최종 상태는 etcd에 반영한다. 컨트롤러/스케줄러/kubectl/kubelet 등 모든 컴포넌트가 이 서버와만 통신한다.

(2) kube-controller-manager

노드/파드/레플리카/엔드포인트/잡 등 “컨트롤러” 프로세스 묶음을 실행하며, 실제 상태를 감시해 선언된 원하는 상태로 수렴시킨다. 예를 들어 파드가 사라지면 지정된 레플리카 수를 맞추도록 새로 만든다.

(3) kube-scheduler

새로 생성된 파드에 대해 자원 여유, 어피니티/안티어피니티, 테인트/툴러레이션, 토플로지 등을 평가해 최적 노드를 할당한다. 스케줄만 담당하며 실제 실행은 kubelet이 수행한다.

(4) etcd

클러스터의 모든 메타데이터와 오브젝트 상태를 저장하는 분산 키-값 저장소이다. 일관성/가용성을 제공하며, api-server만이 etcd를 읽고 쓸 수 있다.

(5) cloud-controller-manager

클라우드 제공자 연동을 담당하여 로드밸런서, 노드 생명주기, 라우트, 볼륨 등 클라우드 리소스를 쿠버네티스 오브젝트와 동기화한다. 온프레미스가 아닌 환경에서 인프라 자원을 자동 관리 한다.

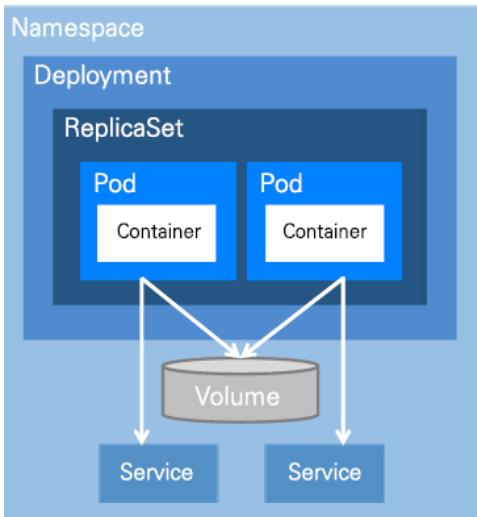
(6) kubelet

각 워커 노드에서 동작하는 에이전트로, api-server가 전달한 파드 스펙을 받아 컨테이너 런타임을 통해 실제 파드를 생성/감시한다. 주기적 헬스체크로 상태를 보고하고, 필요 시 재시작한다.

(7) kube-proxy

서비스(ClusterIP) 가상 IP와 실제 파드 엔드포인트 간의 네트워킹을 위해 iptables/IPVS 규칙을 관리하고 L4 로드밸런싱을 제공한다. 노드별로 동작하며 클러스터 내부 트래픽 라우팅을 담당한다.

5) Resource 포함관계



(1) Namespace

클러스터 하나를 논리적으로 나눠 팀/서비스 단위 경계 공간을 제공한다. 리소스 이름은 네임스페이스 안에서만 고유하며, Service/Deployment 등도 여기에 속한다.

(2) Deployment

원하는 Pod 상태를 선언하면 이를 유지하도록 ReplicaSet을 자동 관리하는 상위 컨트롤러다. 롤링 업데이트/롤백의 단위이며 실제 Pod를 직접 만들지 않고 사양만 기술한다

(3) ReplicaSet

항상 N 개 Pod 개수 보장을 담당하는 컨트롤러이다. Deployment 가 새 버전을 배포하면 새 ReplicaSet 을 만들고, 이전 ReplicaSet 의 Pod 수를 줄여가며 전환한다.

(4) Pod

컨테이너가 실행되는 최소 배포 단위로, 하나 이상의 컨테이너와 그들이 함께 쓰는 네트워킹, 스토리지를 묶는다. 스케줄러가 노드에 배치하면 kubelet 이 생성,헬스체크를 수행한다.

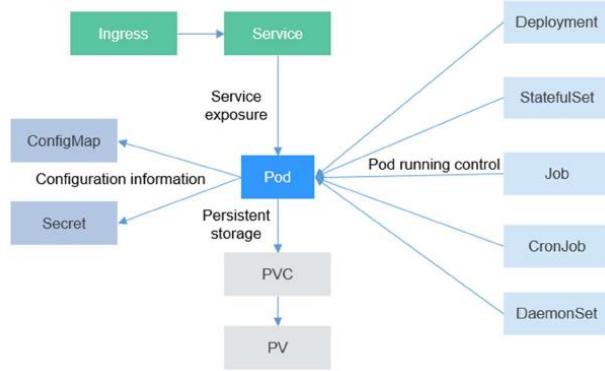
(5) Service

클러스터 내부/외부에서 Pod 집합에 안정적인 IP 와 로드밸런싱을 제공한다. 라벨 셀렉터로 대상 Pod를 찾으며, Pod 교체가 일어나도 접근점은 변하지 않는다.

(6) Volume

Pod 내부에서 컨테이너 간 공유되는 데이터 영역이다. 컨테이너 재시작에도 유지되며, 타입에 따라 임시(emptyDir)부터 외부 스토리지(예: PVC 연동)까지 사용할 수 있다.

6) Resource



(1) 워크로드

- Deployment : 선언형으로 무상태(Stateless) 앱의 배포 단위를 관리-자동화하는 컨트롤러이다. ReplicaSet 을 생성·교체하며 롤링업데이트/롤백을 제공해 짧고 안전한 배포를 가능하게 한다
- StatefulSet : 상태를 가지는 워크로드(DB, 메시지 브로커)를 순서와 고유성을 보장하며 배포, 스케일, 업데이트하기 위한 쿠버네티스 컨트롤러이다.
- DaemonSet : 모든(또는 라벨로 선택한) 노드마다 하나씩 Pod 를 보장 배치한다. 노드 에이전트, 로그/모니터링, CNI 같은 노드-레벨 데몬 배포에 사용된다.
- Job : 종료가 있는 배치 작업을 보장 실행하는 컨트롤러이다. 목표 완료 수(Completions)까지 Pod 를 재시도·대체해 성공 완료를 보장한다.
- CronJob : 스케줄 기반으로 Job 을 주기 실행한다. 크론 표현식으로 시간대를 정의하며 운영 작업, 리포트, 정기 백업 등에 적합하다

(2) 네트워킹

- Service : Pod 집합 앞단의 안정 IP/이름과 로드밸런싱을 제공하는 가상 엔드포인트이다. Pod 교체/스케일에도 주소가 변하지 않아 발견성을 보장하며, Ingress 의 백엔드가 된다.
- Ingress : L7(HTTP/HTTPS) 라우팅을 제공해 클러스터 외부 트래픽을 내부 Service 로 보낸다. 고정 URL, TLS/SSL, 도메인 기반 가상호스팅, 로드밸런싱을 지원(컨트롤러 필요)한다.

(3) 설정

- ConfigMap : 민감하지 않은 설정을 키-값으로 분리해 이미지와 독립 배포한다. 환경변수, 파일로 주입되어 환경 간 일관성과 재배포 최소화를 돋는다.
- Secret : 토큰·자격증명 등 민감 데이터 전달 전용 리소스이다. Base64 인코딩으로 저장되어 Pod 에 환경변수/파일로 안전 주입되며, ConfigMap 과 달리 접근·감사 통제가 전제된다.

(4) 저장소

- PV (PersistentVolume) : 클러스터 관점의 실제 스토리지 리소스이다. 관리자가 미리(Static) 또는 StorageClass 로 동적(Dynamic) 프로비저닝하며, 바인딩 후 PVC 가 사용한다
- PVC (PersistentVolumeClaim) : 워크로드 관점의 스토리지 청구서이다. 용량, 접근모드, StorageClass 를 요청하면 적합한 PV 와 바인딩되어 Pod 에 마운트된다
- 동적 프로비저닝 vs. 정적 프로비저닝 : 정적 프로비저닝은 관리자가 미리 PV 를 만들어 두고 PVC 가 그 PV 에 바인딩하는 방식이며, 동적 프로비저닝은 PVC+StorageClass 로 요청 시 PV 가 자동 생성, 바인딩된다. 동적은 셀프서비스/자동화로 신속하고 운영 부담이 적지만, 설정 미흡 시 과다/과소 할당·비용 스파이크 위험이 있다. 정적은 거버넌스/예측 가능성성이 높고 특수 스토리지 요구를 염격히 통제하기 좋습니다

7) Kubectl

(1) 개요

kubectl 은 Kubernetes API 서버와 통신하여 클러스터의 리소스를 생성, 조회, 수정, 삭제하는 표준 CLI 이다. 선언형(YAML apply)과 명령형(create, run, edit 등) 모두 지원하며, kubeconfig 의 컨텍스트/자격증명을 사용해 여러 클러스터를 전환할 수 있다.

(2) 명령어

- 생성

(생성) apply

명령 의미

파일이나 표준입력(stdin)으로부터 리소스에 구성 변경 사항을 적용

기본 형태

```
$ kubectl apply -f FILENAME [flags]
```

(생성) create

명령 의미

파일이나 표준입력에서 하나 이상의 리소스를 생성

기본 형태

```
$ kubectl create -f FILENAME [flags]
```

- 조회

(조회) get

명령 의미

하나 이상의 리소스를 조회

기본 형태

```
$ kubectl get [TYPE] [flags]  
$ kubectl get [TYPE] [NAME]  
$ kubectl get [TYPE] [NAME] [flags]
```

(로그조회) logs

명령 의미

특정 이름을 가진 리소스의 로그 정보 출력

기본 형태

```
$ kubectl logs [flags] [NAME]
```

(상세조회) describe

명령 의미

하나 이상의 리소스의 상세한 상태 조회

기본 형태

```
$ kubectl describe -f FILENAME
```

- 수정

(Editor 수정) edit

명령 의미

특정 이름을 가진 리소스 정보 수정

기본 형태

```
$ kubectl edit [TYPE] [NAME]
```

8) Resource 관리: yaml 작성

(1) deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: edu-msa-board
  labels:
    app: board-msa
spec:
  replicas: 1
  selector:
    matchLabels:
      app: board-msa
  template:
    metadata:
      labels:
        app: board-msa
```

```
spec:
  containers:
    - name: board-msa
      image: paastakr/edu-msa-board:latest
      imagePullPolicy: Always
      ports:
        - containerPort: 28082
      imagePullSecrets:
        - name: edu-msa-secret
      nodeSelector:
        kubernetes.io/hostname: paas-ta-worker-1
```

- apiVersion : 리소스를 처리할 Kubernetes API 그룹/버전을 지정한다(예: apps/v1). 서버는 이 값으로 스키마를 선택하며 틀리면 객체가 거부된다.
- kind : 생성·관리할 리소스의 유형을 지정한다(예: Deployment, Service). API 서버가 어떤 컨트롤러/스키마를 적용할지 결정한다.
- metadata : 리소스 식별·분류용 메타데이터 컨테이너이다.
 - metadata.name : 네임스페이스 내에서 유일해야 하는 리소스 이름이다. 생성 후 변경이 제한되며 kubectl/디스커버리에서 기본 식별자로 쓰인다.
 - metadata.labels : key: value 라벨 집합으로 리소스를 묶거나 선택할 때 사용한다. 셀렉터, 정책 타깃팅, 모니터링 분류 등에 활용된다.
 - metadata.labels.app : 관례적으로 애플리케이션 묶음을 나타내는 라벨 키이다.(예: app: board-msa). selectorService 매칭과 대시보드 분류에 자주 사용된다.
- spec : 컨트롤러가 수렴시켜야 할 원하는 상태(Desired state)를 기술한다. 현재 상태는 컨트롤러에 의해 이 값에 맞게 변경된다.
 - spec.replicas : 유지할 파드 개수이다. 값 변경 시 ReplicaSet을 통해 안전하게 롤링된다.
 - spec.selector : 이 리소스가 관리할 파드를 고르는 규칙이다.
 - spec.selector.matchLabels : 특정 라벨 키/값이 정확히 일치하는 파드만 선택한다.
- spec.template : 생성될 파드의 메타데이터와 실행 사양을 함께 담는다.
 - spec.template.metadata : 생성될 파드에 부여할 라벨/주석 등 메타데이터이다. selector.matchLabels 와 동일 라벨을 설정해야 관리 대상이 된다.
 - spec.template.spec : 파드의 실행 사양을 정의한다. 실제 런타임 동작에 영향을 미친다.
 - spec.template.spec.containers[] : 파드 내 컨테이너 목록이다. 각 항목이 하나의 컨테이너 스펙을 나타낸다.
 - spec.template.spec.containers[].name : 파드 안에서 유일한 컨테이너 식별자이다.
 - spec.template.spec.containers[].image : 사용할 컨테이너 이미지이다. 변경 시 롤링 업데이트가 트리거된다.

- spec.template.spec.containers[].imagePullPolicy : 이미지 다운로드 정책이다. :latest 태그는 기본적으로 Always 가 적용된다.
- spec.template.spec.containers[].ports[] : 컨테이너가 리스닝하는 포트를 문서화한다. Service 선택, 프로브/네트워크 설정과 주로 연동된다.
- spec.template.spec.imagePullSecret : 프라이빗 레지스트리 인증을 위한 시크릿 참조이다. 하나 이상 사용할 수 있으며 시크릿은 같은 네임스페이스에 있어야 한다.
- spec.template.spec.nodeSelector : 지정 라벨을 가진 노드에만 파드를 스케줄하도록 제한한다.

(2) service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: edu-msa-board
  labels:
    app: board-msa
spec:
  ports:
    - nodePort: ${EDU_MSA_BOARD}
      port: 28082
      protocol: TCP
      targetPort: 28082
  selector:
    app: board-msa
  type: NodePort
```

- spec.ports[].nodePort : type: NodePort(또는 LoadBalancer 에서 내부적으로 NodePort 사용)일 때, 각 노드에서 외부로 개방되는 포트 번호이다. 기본 자동 할당 범위는 30000~32767이며, 직접 지정도 가능하다.
- spec.ports[].port : **Service(ClusterIP)**의 공개 포트로, 클러스터 내부 클라이언트가 ClusterIP:port 로 접속할 때 사용한다. NodePort/LoadBalancer 인 경우에도 트래픽 경로는 nodePort → port → targetPort 로 이어진다.
- spec.ports[].protocol : 서비스가 사용하는 전송 프로토콜입니다. 기본값은 TCP이며 UDP, SCTP 도 지원한다.
- spec.ports[].targetport : **백엔드 파드(컨테이너)**가 실제로 리스닝 중인 포트이다. 숫자 또는 컨테이너 포트의 이름을 참조할 수 있다.
- spec.type : 서비스 노출 방식이다. 기본 ClusterIP(내부 전용), NodePort(각 노드 포트 개방), LoadBalancer(클라우드 LB 연동), ExternalName(DNS 별칭) 중 선택한다. 스크린샷은 NodePort 라 외부에서 NODE_IP:nodePort 로 접근한다.

I -5. Container Platform

1. Container Platform 정의

컨테이너 플랫폼은 쿠버네티스를 기반으로 컨테이너화된 애플리케이션의 배포/스케일링/운영을 자동화하고, 스토리지·레지스트리·보안·관측 가능성(Observability) 등 운영 필수 기능을 통합 제공하는 오픈소스 PaaS 환경이다.

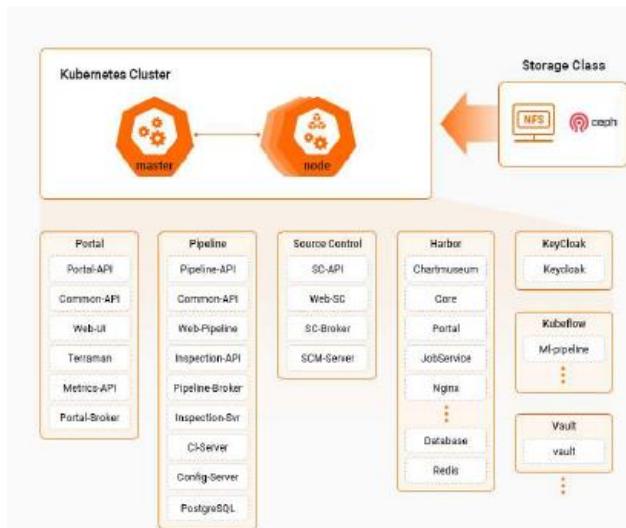
운영 관점에서 워크로드/서비스/클러스터 자원을 일관되게 관리하며, 단독 배포 또는 엣지(Edge) 환경 배포 등 다양한 배포 모델을 지원한다.

2. K-PaaS 표준모델

1) 정의

K-PaaS는 “클라우드 인프라 위에서 SW/서비스의 개발, 실행, 운영, 관리를 가능하게 하는 기반 SW 환경”을 표준화한 모델로, 다양한 상용 PaaS 간 호환성과 공통 기능을 오픈소스로 제공/개방하는 것을 목표로 한다.

2) 구성



Kubernetes Cluster + Storage Class를 중심으로, Portal, 소스/파이프라인, Harbor(이미지 레지스트리), Keycloak(인증/SSO), Ingress Controller, Database, Terraform, Istio, Kubeflow, Vault 등 운영 패키지를 함께 제공한다.

3) 단독형 배포

데이터센터 등 중앙 환경에 K-PaaS를 단독으로 설치하여 하나의 독립된 쿠버네티스 클러스터와 플랫폼 패키지(Portal/Harbor/Keycloak/Storage 등)를 통합 제공하는 방식이다.

- (1) 단일 클러스터 기준으로 표준화된 배포/운영 절차와 일원화된 관리 포털을 제공
- (2) 내부 레지스트리(Harbor), 인증(Keycloak), 보안/Vault, 네트워킹(Ingress), IaC(Terraform) 등을 플러그형 컴포넌트로 즉시 사용 가능

4) Edge 배포

엣지 컴퓨팅 시나리오에서 KubeEdge(CloudCore/EdgeCore)를 활용해 중앙의 쿠버네티스와 분산 엣지 노드를 연동, 엣지에 컨테이너 플랫폼 패키지를 독립적으로 배포하는 방식이다.

- (1) 현장(엣지)에서 저지연·오프라인 내성을 확보하면서 중앙과 정책, 이미지, 인증을 동기화.
- (2) 지점/공장/IoT 등 분산 사이트 다수를 중앙에서 일관 관리하고, 로컬 트래픽은 엣지에서 처리해 네트워크 비용과 지연을 절감

3. 주요 소프트웨어

1) 클러스터 구성 소프트웨어

클러스터 구성 소프트웨어	Kubespray	Apache License 2.0
	Kubernetes	Apache License 2.0
	CRI-O	Apache License 2.0
	Calico	Apache License 2.0
	MetalLB	Apache License 2.0
	Ingress Nginx Controller	Apache License 2.0
	Helm	Apache License 2.0
	Istio	Apache License 2.0
	Podman	Apache License 2.0
	OpenTofu	Mozilla Public License 2.0
	Kubeflow	Apache License 2.0
	NFS	Apache License 2.0
	Rook Ceph	Apache License 2.0

- (1) Kubespray : Ansible 플레이북과 인벤토리를 이용해 Public/Private Cloud 어디서나 쿠버네티스 클러스터를 자동으로 구축하는 오픈소스 배포 도구이다.
- (2) Kubernetes : 대규모 컨테이너 워크로드의 배포/스케줄링/확장/상태 관리를 오케스트레이션으로 제공하며, 다수의 서버에 흩어진 컨테이너를 일관되게 운영하게 해준다. 이러한 오케스트레이션을 통해 서비스 가용성과 보안 수준을 높일 수 있다.
- (3) CRI-O : Kubernetes의 CRI 표준을 최소한의 런타임으로 구현한 OCI 호환 컨테이너 런타임이다. 실행에 집중하여 도커식 빌드/이미지 관리 기능은 제공하지 않으며, Podman 등과 역할분담한다.
- (4) Calico : 대표적 3rd-party CNI로, L3 기반의 가상 네트워킹과 네트워크 정책을 제공하여 Pod 간 통신과 보안을 세밀하게 제어한다. 오버레이 없이 고성능/대규모 네트워킹과 분산 방화벽 수준의 정책 적용이 가능하다.
- (5) MetalLB : 온프레미스 클러스터에서 서비스 타입 LoadBalancer를 가능하게 해 주는 컴포넌트이다. L2(ARP/NDP) 또는 L3(BGP) 방식을 이용해 외부로 노출되는 LB IP를 제공하므로 운영환경 연동이 수월하다.
- (6) NGINX Ingress Controller : Ingress 리소스의 규칙을 실제 L7 프록시로 적용/운영하는 컨트롤러로, HTTP 등 외부 트래픽의 클러스터 진입을 관리한다. 가장 널리 사용되는 컨트롤러라고 한다.
- (7) Helm : 애플리케이션을 '차트'로 패키징하여 배포/업그레이드/롤백을 자동화하는 쿠버네티스 패키지 관리자다. 변수 재정의를 통해 동일 프레임워크로 여러 인스턴스를 일관되게 구성 가능하다.
- (8) Istio : 플랫폼 독립적인 서비스 메시로, 마이크로서비스 간 통신을 위한 트래픽 관리, 인증/인가, 암호화(mTLS), 관측성을 표준화한다. 대규모 분산 애플리케이션 운영의 네트워크 복잡도를 낮춘다.
- (9) Podman : 데몬 없이 컨테이너 생명주기를 관리하는 아키텍처로, 보안성과 유연성이 특징이다. 컨테이너 생성/이미지 전송 등은 Buildah/Skopeo와 조합해 사용하며 Docker와의 차별점이 된다.
- (10) OpenTofu : Terraform의 대안 오픈소스 IaC로, 사람 읽기 쉬운 구성 파일로 클라우드/온프레미스 리소스를 선언하고 수명주기 전반을 일관된 워크플로로 관리한다. 컴퓨팅, 스토리지 같은 하위 리소스부터 DNS/SaaS 같은 상위 구성요소까지 다룬다.
- (11) NFS : 네트워크를 통해 원격 호스트의 파일을 공유하는 고전적인 분산 파일 시스템으로 다양한 OS에서 폭넓게 사용된다. 단일 서버로 간편하게 제공할 수 있으나, 클러스터형 스토리지에 비해 안정성이 낮을 수 있다.
- (12) Rook Ceph : Ceph(파일/블록/오브젝트 스토리지)를 쿠버네티스에 배포/구성/확장/업그레이드/모니터링까지 자동화하여 '자가 관리/확장/치유' 스토리지를 제공한다. 대규모 프로덕션 배포에 적합한 CNCF Graduated 프로젝트이다.

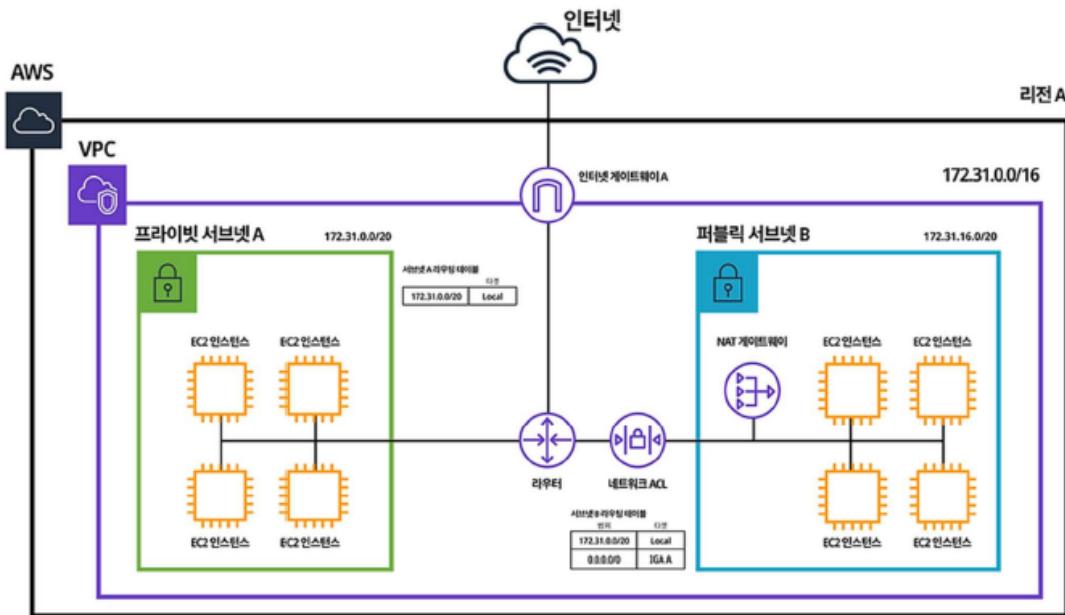
2) 포털 구성 소프트웨어

포털 구성 소프트웨어	Vault	Business Source License 1.1
	Harbor	Apache License 2.0
	MariaDB	General Public License 2.0
	Keycloak	Apache License 2.0

- (1) Vault : 컨테이너 플랫폼 포털에서 인증·자격증명 데이터를 중앙에서 관리하는 비밀관리 솔루션이다. 민감한 시크릿을 UI/CLI/API로 안전하게 저장·제공하고, 접근 전 유효성 검증과 승인을 수행한다.
- (2) Harbor : 기업 내부에 설치해 쓰는 프라이빗 컨테이너 이미지 레지스트리이다. Docker Hub와 유사한 웹 UI를 제공하면서도 폐쇄망/사내용 보안 요구를 충족하도록 설계되었다.
- (3) MariaDB : 포털 메타데이터를 저장하는 관계형 DBMS이다. MySQL과 호환되며 확장성과 쿼리 성능이 우수하고(시퀀스 엔진, 가상 열 등) 대용량 데이터에 유리하다.
- (4) Keycloak : 포털 사용자의 인증·인가를 담당하는 SSO/IDP 솔루션이다. OIDC·SAML·OAuth2 기반으로 중앙집중식 인증/권한부여와 SSO를 제공한다.

II-1. NHN Cloud 실습환경

1. 아키텍처



1) 전체 설계

전 A 내부에 172.31.0.0/16 대역의 VPC를 만들고, 이를 퍼블릭 서브넷 B(172.31.16.0/20)와 프라이빗 서브넷 A(172.31.0.0/20)로 분리한 2-티어 네트워크 아키텍처이다. VPC에는 인터넷 게이트웨이(IGW)가 연결되어 있고, 각 서브넷은 자체 라우팅 테이블과 네트워크 ACL을 통해 트래픽 경로와 서브넷 경계 보안을 분리하여 제어한다는 점이 핵심이다.

2) 퍼블릭 서브넷 B

0.0.0.0/0 기본 경로를 IGW로 내보내도록 설정되어 외부와 직접 통신이 가능하며, 공인 IP를 가진 인스턴스나 로드밸런서/배스천 호스트를 두는 구간이다. 동일 서브넷에 NAT 게이트웨이가 배치되어 프라이빗 서브넷의 아웃바운드 인터넷 접근을 대행하는 역할을 하며, 외부에서 이 NAT로 들어오는 신규 연결은 허용되지 않는 SNAT 단방향 구조이다.

3) 프라이빗 서브넷 A

외부에서 직접 접근이 차단되도록 IGW 경로가 없고, 기본 경로를 퍼블릭 서브넷의 NAT 게이트웨이로 지정해 아웃바운드만 허용하는 구성이다. 애플리케이션/데이터베이스 같은 내부 워크로드를 두어 공격면을 최소화하며, 필요 시 퍼블릭 영역의 배스천을 통해서만 관리 접속을 허용한다. 인스턴스 단 보안은 상태저장형 Security Group으로, 서브넷 경계 보안은 무상태 Network ACL로 계층화하여 운영한다

4) 트래픽 흐름

외부 사용자가 퍼블릭 서브넷의 서비스에 접근하면 인터넷→IGW→퍼블릭 서브넷 인스턴스 경로로 유입된다. 프라이빗 인스턴스가 패키지 업데이트 등으로 인터넷에 나갈 때는 프라이빗 서브넷→라우팅 테이블→NAT 게이트웨이→IGW 경로를 사용하고, 응답은 NAT가 세션을 유지해 내부로 되돌린다. 고가용성을 위해 NAT 게이트웨이와 퍼블릭 서브넷을 다중 AZ로 분산하고, 라우팅 테이블을 서브넷 단위로 명시적으로 연결해야한다고 한다.

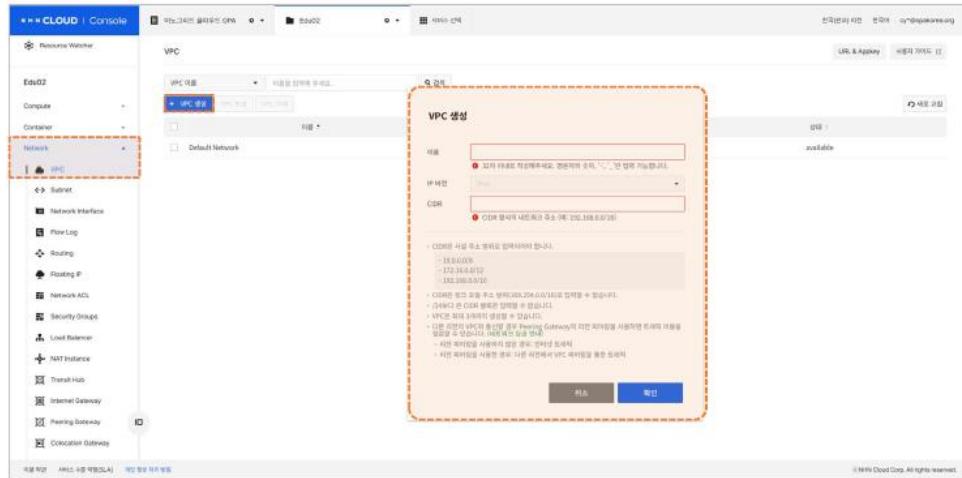
2. Network 메뉴

1) VPC

(1) 이해

VPC는 사용자가 소유한 격리된 가상 네트워크로, 사설 CIDR/서브넷/라우팅/IGW NAT/보안규칙을 포함하는 네트워크 경계이다. 애플리케이션 도메인의 IP 체계와 통신 정책의 최상위 소유 단위가 된다.

(2) 생성



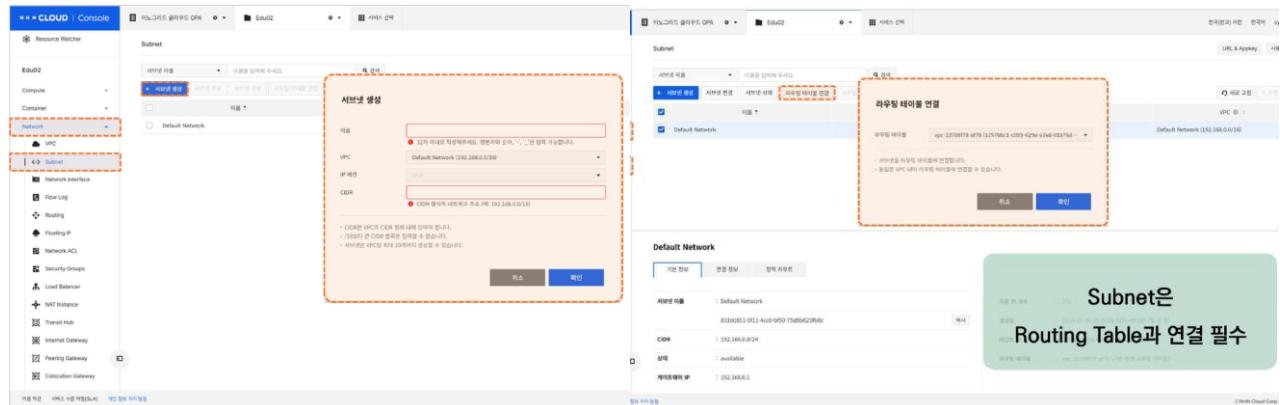
- VPC 생성 → CIDR(예: 172.31.0.0/16) 지정 → DNS 옵션/태그 지정한다.
 - 결과로 독립 네트워크 그릇이 생기며, 이후 모든 리소스(Subnet/Router/IGW)가 이 VPC 범위 안에서만 만들어진다.

2) Subnet

(1) 이해

서브넷은 VPC 주소공간을 AZ/용도별로 나눈 L3 브로드캐스트 도메인이다. 퍼블릭(IGW 경로 보유)·프라이빗(NAT 경로)로 역할을 분리한다.

(2) 생성 및 라우팅 테이블 연결



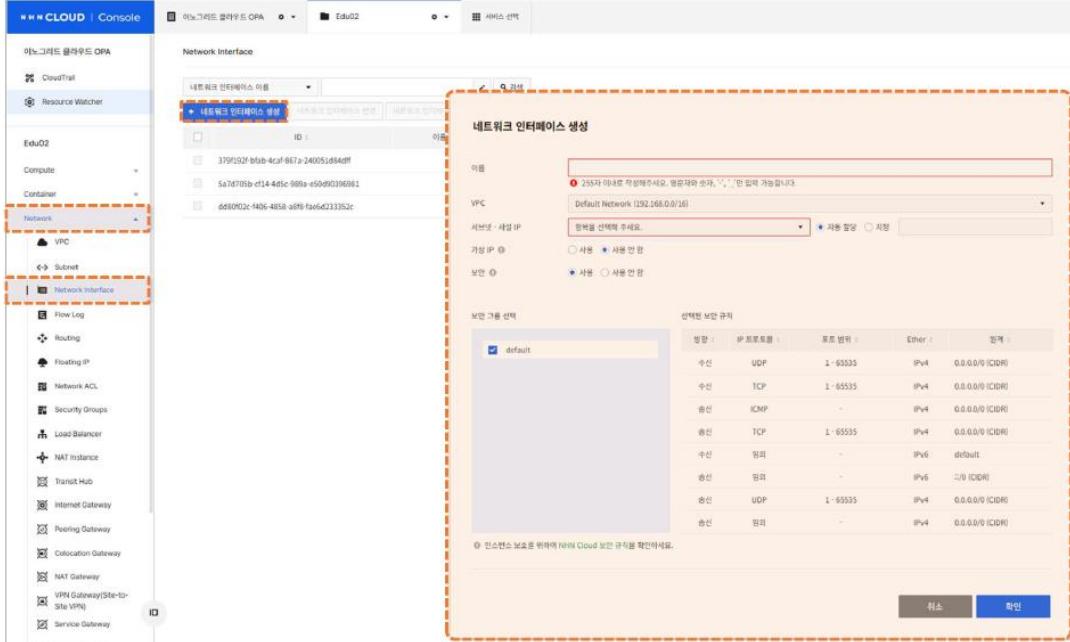
- 서브넷 생성 → 각 서브넷에 라우팅 테이블 연결.
 - 결과로 2-티어 분리가 완성되어 외부 노출과 내부 워크로드가 격리된다.

3) Network Interface

(1) 이해

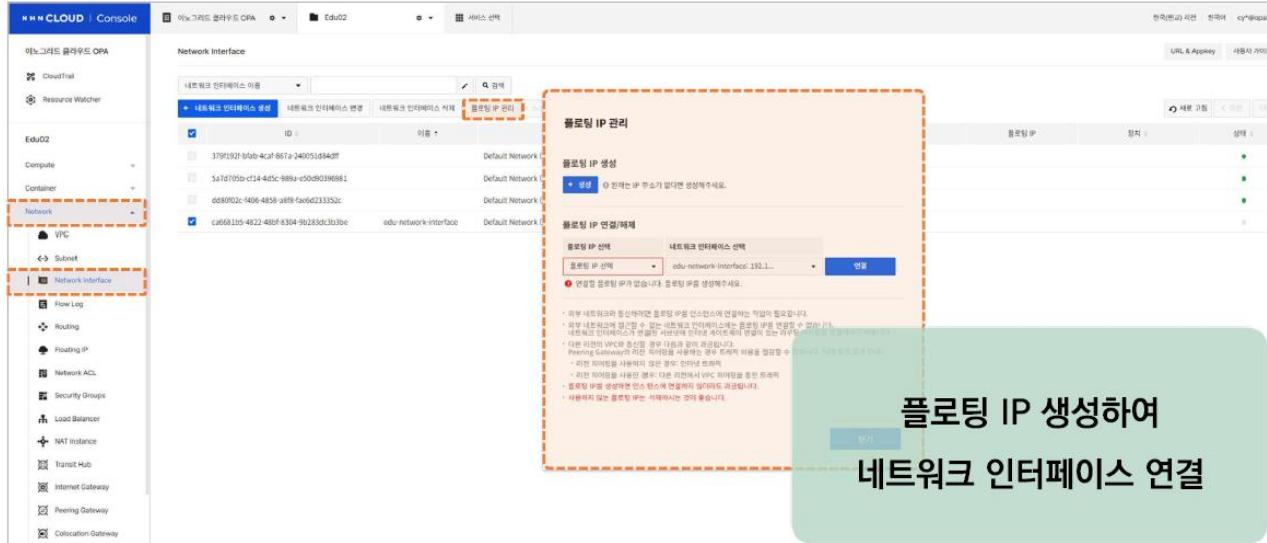
네트워크 인터페이스(ENI)는 VPC 안의 서브넷·보안그룹·사설 IP를 담는 가상 NIC로, 인스턴스에 붙여 트래픽을 송수신하는 단위이다. 하나의 인터페이스에 1개(또는 여러 개)의 사설 IP가 할당되며, 필요 시 플로팅 IP를 매핑해 외부와 통신한다

(2) 생성



- Network → Network Interface → '네트워크 인터페이스 생성'. 이름, VPC/서브넷, IP 할당(자동/수동), 보안그룹을 지정해 생성한다.
- 생성 결과, 선택한 서브넷 대역의 사설 IP를 가진 NIC가 사용 가능 상태로 목록에 나타난다

(3) 플로팅 IP 연결



- Network → Floating IP → 플로팅 IP 생성으로 공인 IP를 먼저 확보한다.
- Network Interface 목록에서 대상 NIC 선택 → 플로팅 IP 연결을 눌러 앞서 만든 공인 IP를 매핑한다.
- NIC의 사설 IP ↔ 플로팅 IP가 1:1로 매핑되어 인터넷에서 접근 가능한 엔드포인트가 된다.

(4) Instance 연결

The screenshot shows the Ncloud Cloud Console interface. On the left, there's a sidebar with various service icons like Compute, Network, Storage, Database, etc. The main area shows an 'Instance' list with one item: 'edu-master-cluster'. Below the list, a modal window titled '네트워크 인터페이스 연결 추가' (Add Network Interface Connection) is open. This modal has several tabs: '네트워크 인터페이스 생성' (Create Network Interface), '기존 네트워크 인터페이스 사용' (Use Existing Network Interface), and '내부 네트워크 인터페이스' (Internal Network Interface). The third tab is selected. It shows a table with one row for 'edu-network-interface'. The table columns are: 이름 (Name), VPC, 서브넷 (Subnet), 사설 IP (Private IP), 풀링 IP (Pooling IP), and 보안 그룹 (Security Group). The '사설 IP' column shows '192.168.0.100'. At the bottom right of the modal are '취소' (Cancel) and '확인' (Confirm) buttons. To the right of the modal, there's a green callout box with the text '네트워크 인터페이스 연결' (Network Interface Connection).

- Compute → Instance에서 대상 인스턴스 선택 → '네트워크 인터페이스 연결 추가' → 리스트에서 생성한 NIC 선택 후 확인. 연결되면 인스턴스의 추가 NIC로 반영된다
- 스턴스가 추가 사설 IP/보안그룹을 가진 인터페이스를 얻어 트래픽 분리(외부/내부), 멀티 네트워크 경로, DMZ 구성 등을 구현할 수 있다.

4) Routing

(1) 이해

라우팅은 CIDR 표기로 목적지 네트워크(예: 0.0.0.0/0 등)과 다음 흡(타깃)을 매핑한 라우팅 테이블로 패킷 전달 경로를 결정하는 기능이다. 서브넷은 하나의 라우팅 테이블과 연결되며, 해당 서브넷의 인스턴스는 그 테이블의 규칙에 따라 통신한다.

(2) 생성

The screenshot shows the Ncloud Console Routing section for the 'Edu02' project. On the left sidebar, under the 'Network' tab, the 'Routing' option is selected. In the main area, there is a 'Routing 테이블 생성' (Create Routing Table) button. A modal window titled '라우팅 테이블 생성' (Create Routing Table) is open, prompting for a table name ('이름') and VPC ('VPC'). The table name is set to 'vpc-22709f78-af78'. The VPC dropdown is set to 'Default Network (192.168.0.0/16)'. The '라우팅 방식' (Routing Method) dropdown is set to '분산형 라우팅(DVR)'. At the bottom of the modal are '취소' (Cancel) and '확인' (Confirm) buttons. The background shows a list of existing routing tables, including 'vpc-22709f78-af78' which is currently selected.

- Network → Routing [라우팅 테이블 생성] 클릭 → 이름, 대상 VPC, (제공 시) 라우팅 방식 (DVR 등)을 지정하고 확인을 눌러 생성
- 생성된 테이블 상세의 [연결 정보] → [연결 대상 추가]에서 이 테이블을 적용할 서브넷을 선택해 연결
- 선택한 서브넷의 인스턴스는 이제 이 테이블의 경로 규칙을 따른다

(3) 설정

The screenshot shows the Ncloud Console Routing section for the 'Edu02' project. The 'Routing' tab is selected. A modal window titled '인터넷 게이트웨이 연결' (Connect Internet Gateway) is open, showing the connection between a routing table and an Internet Gateway. The routing table 'vpc-22709f78-af78' is selected, and the Internet Gateway 'ig-125768c1-c093' is connected. A green callout box highlights the text 'Routing은 Internet Gateway와 연결 필수' (Routing must be connected to Internet Gateway). The background shows the list of routing tables, with 'vpc-22709f78-af78' selected.

- 라우팅 테이블 화면에서 [인터넷 게이트웨이에 연결]을 눌러 해당 VPC의 Internet Gateway를 연결
- 퍼블릭 서브넷은 인터넷 접근이 가능해지고, 프라이빗 서브넷은 NAT를 통한 아웃바운드만 허용되는 등 서브넷별 통신 정책이 구현

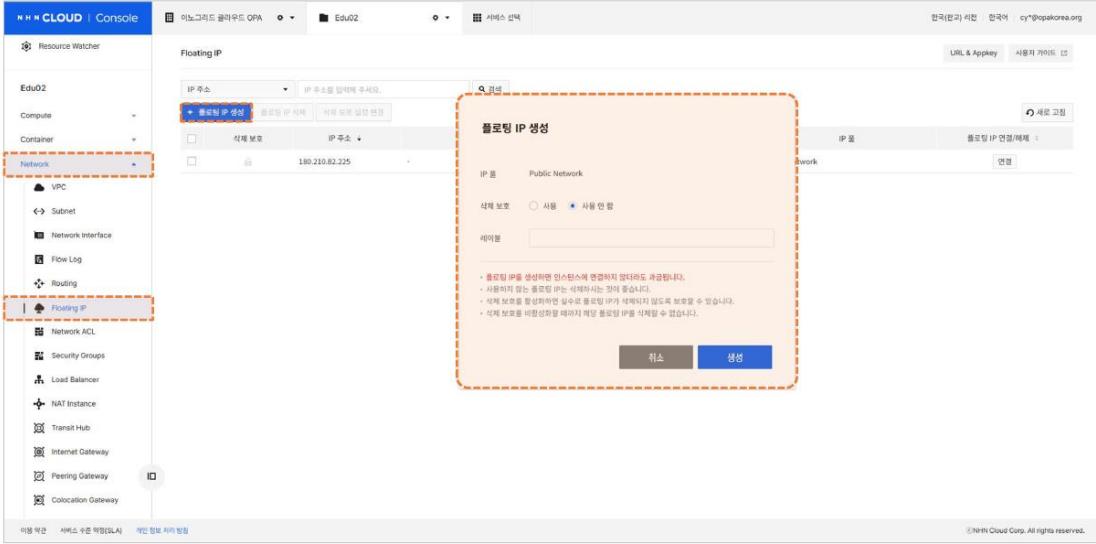
5) Floating IP

(1) 이해

플로팅 IP는 퍼블릭 네트워크의 공인 IP를 사설망(서브넷) 내 네트워크 인터페이스(NIC)에 매핑해 외부(인터넷)에서 인스턴스로 직접 접근할 수 있게 하는 기능이다.

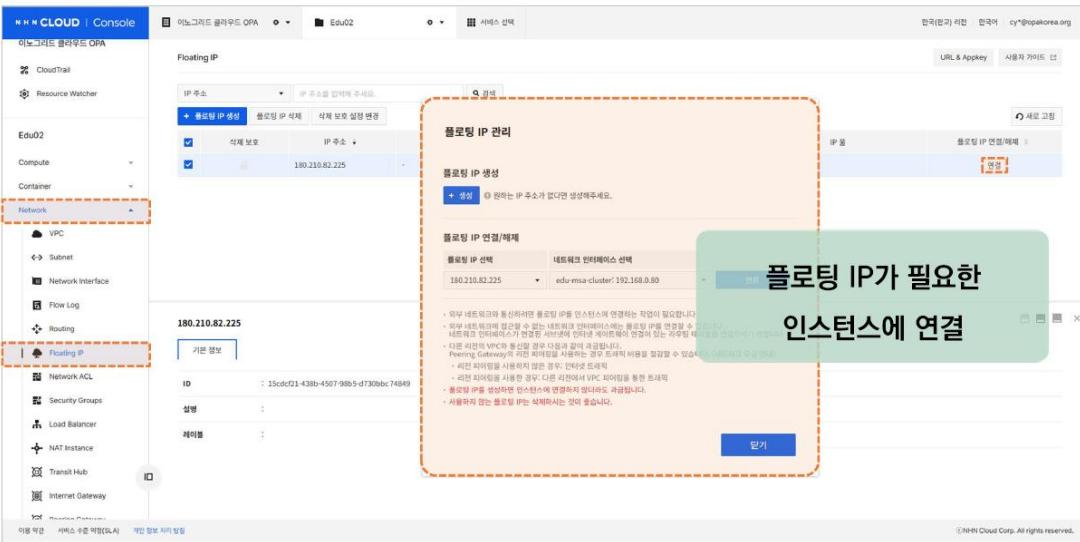
- 라우팅/인터넷 게이트웨이, 보안그룹이 열려 있어야 실제 통신이 된다.

(2) 생성



- Network -> Floating IP -> '플로팅 IP 생성' 클릭 -> IP 풀 Public Network 선택 -> 레이블(선택) 입력 -> 생성
- 새 공인 IP가 Available(미연결) 상태로 목록에 할당된다. 아직 어떤 NIC/인스턴스와도 연결되지 않은 상태이다.

(3) 설정



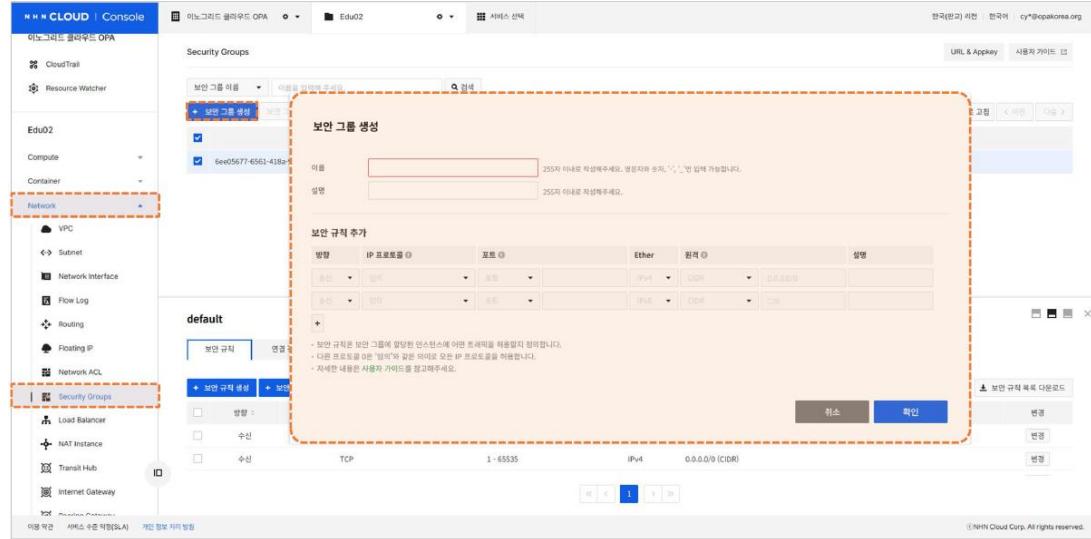
- Floating IP 목록에서 방금 만든 IP 선택 -> '플로팅 IP 연결/해제' -> 연결할 네트워크 인터페이스 선택(예: edu-msa-cluster 192.168.0.X) -> 연결.
- 선택한 NIC에 공인 IP가 부여되어 외부에서 SSH/HTTP 등으로 접속 가능해진다. 연결 해제 시 공인 접근은 즉시 중단된다.

6) Security Groups

(1) 이해

보안 그룹은 인스턴스/네트워크 인터페이스(NI)에 적용되는 상태저장형(Stateful) 가상 방화벽이다. 인바운드/아웃바운드 규칙으로 프로토콜, 포트 범위, 출발지/목적지 CIDR을 지정해 트래픽을 허용/차단한다.

(2) 생성



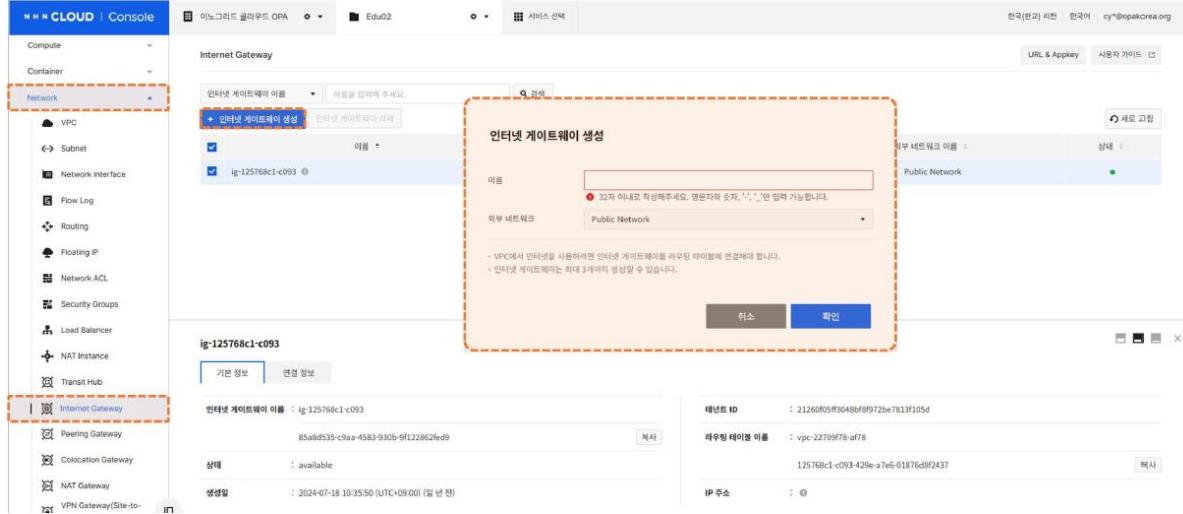
- Network → Security Groups → 보안 그룹 생성 → 이름/설명 입력 → 보안 규칙 추가에서 방향(인/아웃), IP 프로토콜(TCP/UDP/ICMP 등), 포트 범위(예: 22, 80, 443, 30000–32767), Ether(IPv4/IPv6), 원격(CIDR) 지정 → 확인
- 새 보안 그룹이 목록에 생성되고 규칙 테이블이 초기화된다

7) Internet Gateway

(1) 이해

Internet Gateway(IGW)는 VPC 내부 리소스를 공인 인터넷과 연결하는 관문이다. VPC에 단순히 IGW를 생성만 해서는 통신이 되지 않고, 라우팅 테이블에 0.0.0.0/0목적지의 다음 흡으로 IGW를 연결해야 퍼블릭 서브넷의 인스턴스가 외부와 송수신할 수 있다.

(2) 생성



- Network - Internet Gateway - 인터넷 게이트웨이 생성 - 이름 입력 - 외부 네트워크를 Public Network로 선택으로 생성한다.
- 생성 후 Network - Routing으로 이동하여 사용 중인 라우팅 테이블을 선택하고 인터넷 게이트웨이 연결 버튼으로 방금 만든 IGW를 연결한다.
- 결과로 IGW 상태가 available로 표시되고 라우팅 테이블에 0.0.0.0/0 → IGW 경로가 생긴다. 이로써 퍼블릭 서브넷의 인스턴스는 외부 인터넷과 통신할 수 있고, Floating IP를 할당한 NIC/인스턴스는 인터넷에서 직접 접근 가능해진다

3. Compute 메뉴

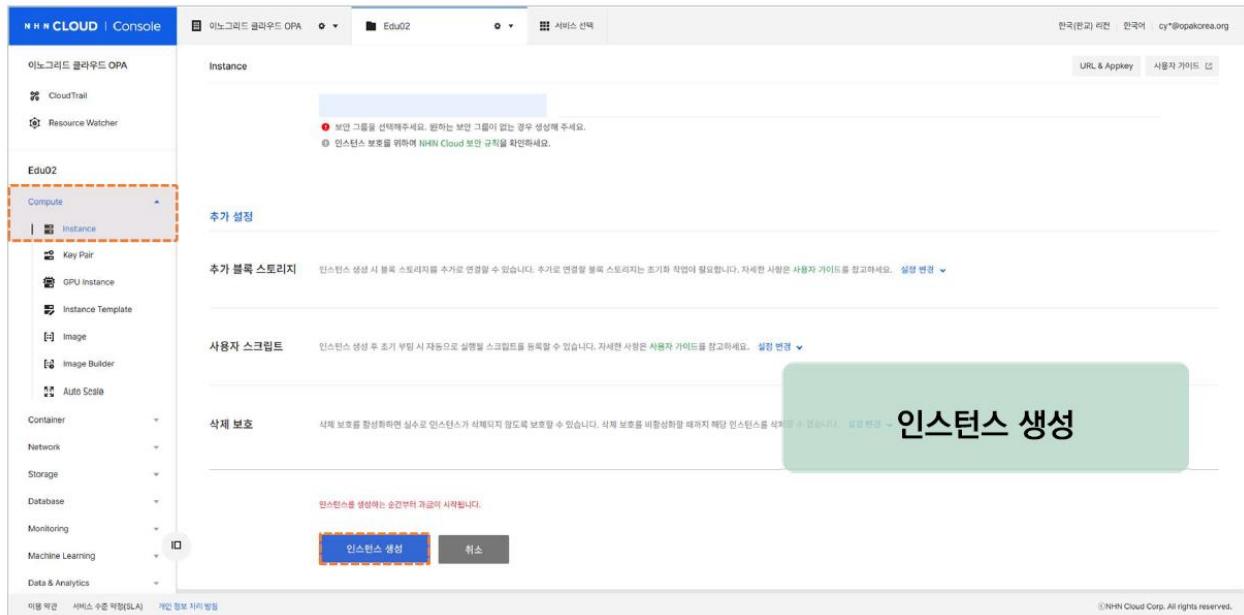
1) Instance

(1) 이해

인스턴스는 하이퍼바이저 위에서 동작하는 가상 서버로 vCPU/메모리/NIC/블록 스토리지로 구성되어 애플리케이션을 실행하는 기본 단위이다.

기본은 VPC/서브넷의 사설 IP로 통신하며, 외부 접속에는 Internet Gateway + 라우팅 (+ 필요 시 플로팅 IP) 구성이 필요하다.

(2) 생성



- 진입 : Compute → Instance로 이동 후 [인스턴스 생성]
- 이미지(OS) 선택 : 공용 이미지/개인 이미지 중에서 배포할 OS를 고른다.
- 루트 블록 스토리지 설정 : 타입(HDD/SSD), 크기(GB), 삭제 정책(인스턴스 삭제 시 유지/삭제)를 지정한다. 이는 부팅 디스크 성능, 비용에 직접 영향을 줍니다.
- 기본 정보 입력 : 가용영역, 배치정책(필요 시), 인스턴스 이름/설명, 인스턴스 수를 설정한다.
- 인스턴스 타입 선택 : 팝업에서 vCPU/메모리 조합이 정해진 타입을 골라 성능, 비용을 결정한다.
- 키 페어 선택/생성 : SSH 또는 RDP 접속용 키 페어를 선택하거나 새로 생성한다. 키를 분실하면 접속이 불가하므로 안전하게 보관해야한다.
- 네트워크 설정(인터페이스/서브넷) : 네트워크 인터페이스 신규 생성 또는 기존 인터페이스 지정을 선택하고, 연결할 서브넷을 고른다.
- 플로팅 IP(공인 IP) 선택 : 외부에서 직접 접속해야 하면 플로팅 IP 사용을 선택하고 IP 풀/연결 서브넷을 지정한다. 필요 없으면 미사용으로 두고 나중에 연결할 수도 있다.
- 보안 그룹 선택 : 기본(default) 또는 직접 만든 보안 그룹을 선택한다.
- 검토 및 생성 : 요약을 확인하고 생성을 누른다. 상태가 Provisioning → Running 으로 바뀌면 준비 완료이다.

2) Key Pair

(1) 이해

- 키페어는 공개키/개인키로 구성된 인증 수단이며, 인스턴스 SSH 접속 시 비밀번호 없이 안전하게 인증하기 위해 사용한다.
- 클라우드 콘솔에는 공개키만 저장되고, 개인키(.pem)는 사용자가 다운로드해 안전하게 보관해야 한다.

(2) 생성

Key Pair

Key Pair Generation

Key Pair Name: edu-key

Message: 32자 이내로 작성해주세요. 영문자와 숫자, '.', '-'만 입력 가능합니다.

Buttons: 취소, 생성

키페어는 생성 후 다운로드 받은 개인키는 분실 주의

- Compute → Key Pair → 키페어 생성 → 이름 입력 → 생성 클릭 → 개인키(.pem)

(3) 등록

Key Pair

Key Pair Import

Public Key File Selection: 파일 선택

Key Pair Name: edu-key

Message: 32자 이내로 작성해주세요. 영문자와 숫자, '.', '-'만 입력 가능합니다.

Buttons: 파일 선택, 등록

키페어 쌍을 가지고 있다면 등록하여 사용 가능

설명

1. ssh-keygen을 이용하여 공개/개인 키를 생성할 수 있습니다.
ssh-keygen -t rsa -f cloud.key

2. 생성한 공개 키 파일을 선택하여 추가하면 키ペ어 이름[파일 이름]과 공개 키가 입력됩니다.
파일을 선택하여 추가하거나 직접 입력할 수도 있습니다.

3. 디렉토리에 있는 키 파일을 [키페어 이름] 입력 상자에 입력해 주세요.

4. 인스턴스가 생성된 이후, 가지고 있는 개인 키를 이용하여 아래와 같은 형태로 로그인할 수 있습니다.
(User 이름은 미리지정해 놓으셨거나, 인스턴스에 로그인해 주었습니다.)

```
ssh -i cloud.key ubuntu@<instance_ip>
```

- 이미 가진 키 쌍이 있다면 Compute → Key Pair → 키페어 가져오기에서 공개키를 붙여넣고 이름을 지정해 등록
- 업로드한 공개키가 키페어 목록에 추가되고, 새 인스턴스 생성 시 해당 키페어 선택 → 로컬의 기준 개인키로 접속한다.
- 생성된 인스턴스에 사후 변경은 제한적이므로 필요 시 OS의 `~/.ssh/authorized_keys`에 직접 공개키를 추가해야 한다.

II-2. 인프라 구축

1. 인프라 구축 : IaaS 설정

1) Subnet 생성

(1) 조별로 알맞은 IP의 subnet 생성

- 서브넷 생성

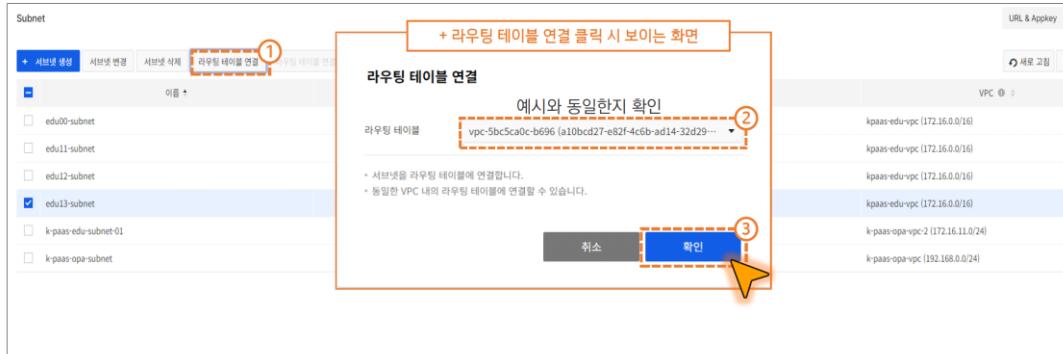
왼쪽 메뉴의 [Network] → [Subnet] → [서브넷 생성] 클릭



(2) 라우팅테이블 생성

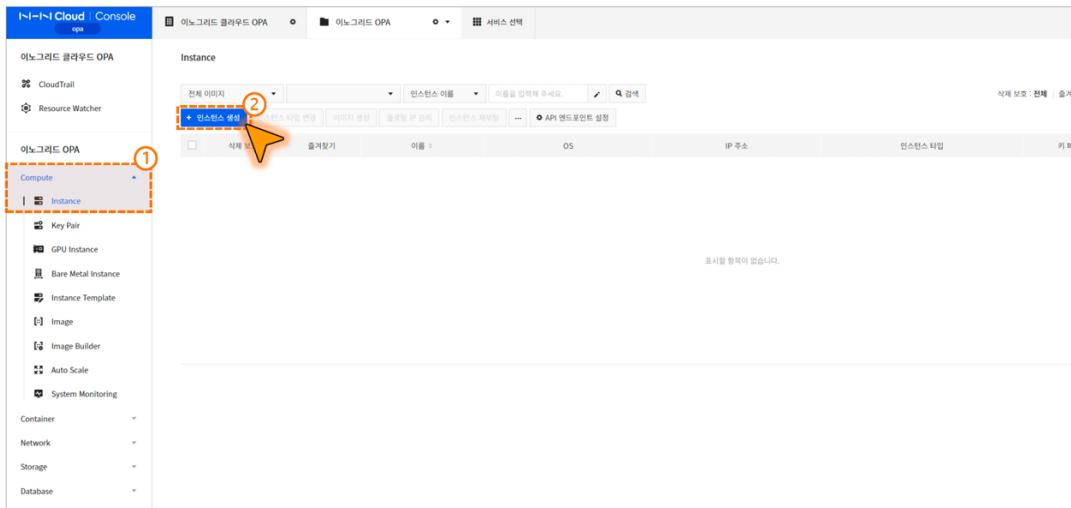
- 서브넷 라우팅 테이블 연결

왼쪽 메뉴의 [Network] → [Subnet] → [라우팅 테이블 연결] 클릭



2) Instance 생성

왼쪽 메뉴의 [Compute] → [Instance] → [인스턴스 생성] 클릭



(1) 이미지 : ubuntu 22.04

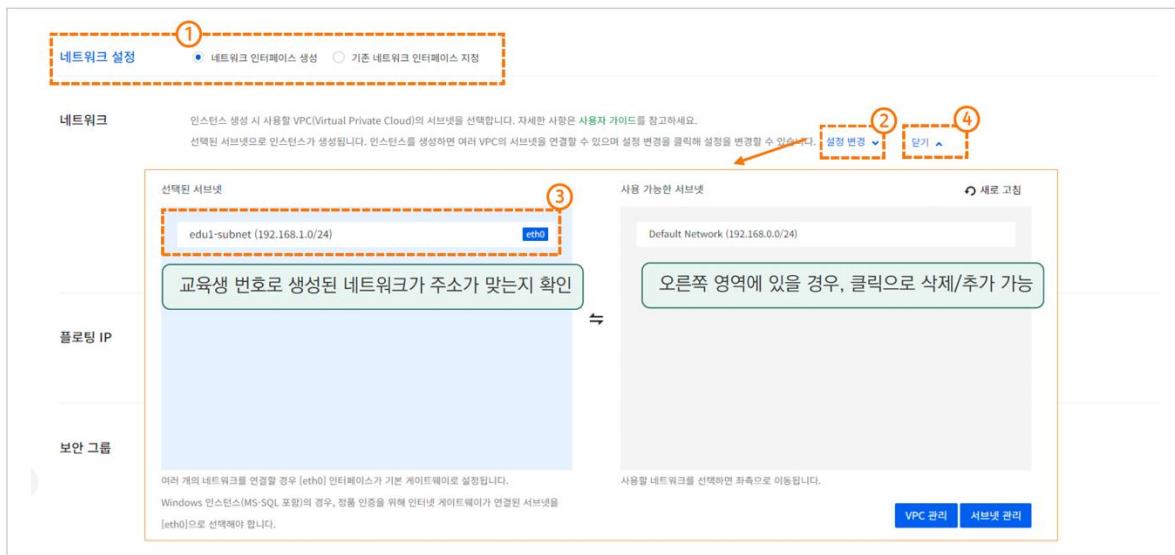
이미지	설명	언어	최소 블록 스토리지(GB)	비트
Ubuntu Server 20.04 LTS	Ubuntu Server 20.04.6 LTS with Apache Kafka 3.3.1 (2023.03.21)		20	64 Bit
Ubuntu Server 20.04 LTS	Ubuntu Server 20.04.6 LTS with CUBRID 10.2.10 (2023.03.21)		20	64 Bit
Ubuntu Server 20.04 LTS	Ubuntu Server 20.04.6 LTS with MariaDB 10.6.11 (2023.03.21)		20	64 Bit
Ubuntu Server 20.04 LTS	Ubuntu Server 20.04.6 LTS with MySQL 8.0.27 (2023.03.21)		20	64 Bit
Ubuntu Server 20.04 LTS	Ubuntu Server 20.04.6 LTS with Redis 7.0.5 (2023.03.21)		20	64 Bit
Ubuntu Server 22.04 LTS	Ubuntu Server 22.04.3 LTS (2024.02.20)		20	64 Bit

(2) 인스턴스 정보 입력

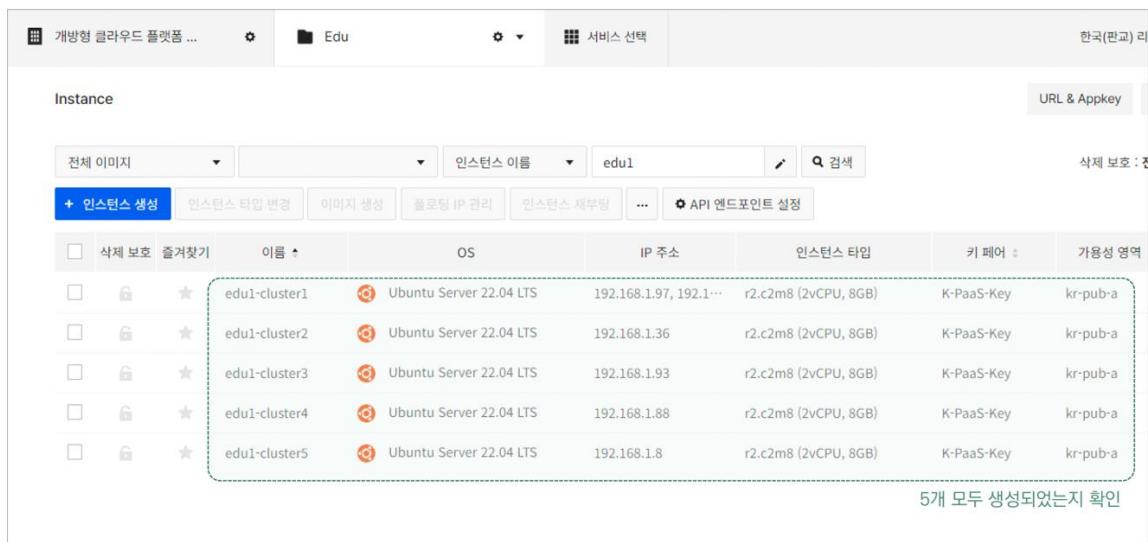
타입	vCPU	RAM
r2.12.2m8	2	8GB
r2.12.4m16	4	16GB
r2.12.8m32	8	32GB
r2.12.16m64	8	64GB

- 인스턴스 수 5개로 생성 : control plane 1개, worker 3개, nfs 1개
- 키페이는 주어진 것 사용

(3) 네트워크 인터페이스 설정



(4) 생성 완료



3) Floating IP 설정

(1) 플로팅IP 생성하여 연결



- 로컬 환경에서 SSH 접속을 위해, Control Plane에 플로팅 IP 연결

4) Network Interface 설정

(1) 생성



- 네트워크 인터페이스(포트)를 별도로 생성·연결하여 외부 노출(플로팅 IP/Ingress)과 내부 클러스터 트래픽을 분리

(2) 연결

인스턴스	작제 보호	줄거리기	이름	인스턴스 시작	OS	IP 주소	인스턴스 타입	키 페어	기용상
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	edu1-cluster5	인스턴스 중지	tu Server 22.04 LTS	192.168.1.8	r2.c2m8 (2vCPU, 8GB)	K-PaaS-key	kr-pub-a
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	edu1-cluster2	인스턴스 종료	tu Server 22.04 LTS	192.168.1.16	r2.c2m8 (2vCPU, 8GB)	K-PaaS-key	kr-pub-a
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	edu1-cluster3	인스턴스 삭제	tu Server 22.04 LTS	192.168.1.61	r2.c2m8 (2vCPU, 8GB)	K-PaaS-key	kr-pub-a
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	edu1-cluster4	보안 그룹 변경	tu Server 22.04 LTS	192.168.1.75	r2.c2m8 (2vCPU, 8GB)	K-PaaS-key	kr-pub-a
<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	edu1-cluster1	작제 보호 설정 변경	tu Server 22.04 LTS	192.168.1.14,	r2.c2m8 (2vCPU, 8GB)	K-PaaS-key	kr-pub-a

- 인스턴스 중지 후 연결

5) Instance 접속

4. Worker 노드 접속

```
ubuntu@edu3-cluster1:~$ ssh -i K-PaaS-Key3.pem ubuntu@192.168.3.29
```

Pem key를 통해 worker 노드에 접속

\$ ssh -i [pem키 파일명] ubuntu@[worker노드IP주소]

* Pem key가 정상 동작하지 않을 경우 접속 불가

- ssh를 통해 instance에 접속 가능

II-3. 컨테이너 플랫폼 구축

1. 클러스터 설치

1) NFS 스토리지 설치

(1) 설치목적

NFS 스토리지는 k8s의 PV 백엔드로서, 파드와 노드 수명주기와 무관하게 데이터를 중앙에서 보관하고, 필요 시 여러 파드가 동시에 접근할 수 있게 하는 공유 파일 스토리지다

(2) APT 업데이트 및 APT 패키지 설치

```
$ sudo apt-get update  
  
$ sudo apt-get install -y nfs-common nfs-kernel-server portmap
```

(3) NFS Server에서 사용할 디렉토리 생성 및 권한 부여

```
$ sudo mkdir -p /home/share/nfs  
$ sudo chmod 777 /home/share/nfs
```

(4) 공유 디렉토리 설정

```
$ sudo vi /etc/exports  
  
/home/share/nfs {{MASTER_NODE_PRIVATE_IP}}(rw,no_root_squash,async)  
/home/share/nfs {{WORKER1_NODE_PRIVATE_IP}}(rw,no_root_squash,async)  
/home/share/nfs {{WORKER2_NODE_PRIVATE_IP}}(rw,no_root_squash,async)
```

- IP자리에 이전에 설정한 인스턴스들의 IP를 넣는다

(5) 재시작

```
$ sudo /etc/init.d/nfs-kernel-server restart  
$ sudo systemctl restart portmap
```

2) SSH Key 생성 및 배포

(1) SSH Key?

(2) 키 생성

```
$ ssh-keygen -t rsa -m PEM -N '' -f $HOME/.ssh/id_rsa
Generating public/private rsa key pair.
Your identification has been saved in /home/ubuntu/.ssh/id_rsa
Your public key has been saved in /home/ubuntu/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:odWdv3PDIEpkPuoS53yM0hrsEQZL4mHvM0KwLK2uC57 ubuntu@cp-master
The key's randomart image is:
+---[RSA 3072]---+
|                               |
|                               |
|                               . . .
| .+ o     = . o
|+= o * . . .
|oo+o .. S . . .
|.+.o o o . + .
|o +o O. .      *
|=o.o=o*          o
|E++o ++
+---[SHA256]---+
```

(3) 공개키 복사

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB5QrbqzV6g4iT4iR1u+EKKVQGqBy4DbGqH7/PVfmA
```

(4) 인스턴스에 공개키 추가

```
$ vi .ssh/authorized_keys

ex)
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQDRueywSiuwyfCSechU7iwyi3xYS1xigAnhR/RMg/lw
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB5QrbqzV6g4iT4iR1u+EKKVQGqBy4DbGqH7/PVfmA
```

- 각 인스턴스에 접근해서 공개키를 authorized_keys 파일 마지막에 복사한다.

3) Deployment 설치

(1) 목적

NFS를 백엔드로 한 PVC→PV→NFS 경로를 Deployment에 선언·마운트함으로써, 애플리케이션의 영속 데이터 보존, 무중단 갱신, 스케일링, 공유 접근(RWX) 을 표준 방식으로 실현할 수 있다.

(2) Deployment 다운로드

```
$ git clone https://github.com/K-PaaS/cp-deployment.git -b branch_v1.6.x
```

(3) 환경변수 수정

```
$ cd ~/cp-deployment/single
```

```
$ vi cp-cluster-vars.sh
```

```
export MASTER1_NODE_HOSTNAME=kjh-cluster1
export MASTER1_NODE_PUBLIC_IP=133.186.229.182
export MASTER1_NODE_PRIVATE_IP=192.168.7.86

export KUBE_WORKER_HOSTS=3

export WORKER1_NODE_HOSTNAME=kjh-cluster2
export WORKER1_NODE_PRIVATE_IP=192.168.7.77
export WORKER2_NODE_HOSTNAME=kjh-cluster3
export WORKER2_NODE_PRIVATE_IP=192.168.7.70
export WORKER3_NODE_HOSTNAME=kjh-cluster4
export WORKER3_NODE_PRIVATE_IP=192.168.7.85

export STORAGE_TYPE=nfs

export NFS_SERVER_PRIVATE_IP=192.168.7.11

export METALLB_IP_RANGE=192.168.7.150-192.168.7.160

export INGRESS_NGINX_IP=192.168.7.

export INSTALL_KYVERNO=Y
```

위와 같이 내 클러스터 정보로 수정한다.

4) Deployment.yaml 설정

(1) yaml 파일 생성 및 수정

```
vi deployment.yaml
```

- vim 으로 생성

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kjh-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

- nginx 1.14.2 이미지를 replicas=3으로 기동하고, app: nginx 라벨을 부여하도록 선언

(2) 리소스를 처음 생성

```
kubectl create -f deployment.yaml
```

(3) 선언형 갱신

```
kubectl apply -f deployment.yaml
```

- 파일에 적힌 원하는 상태를 클러스터의 현재 상태와 병합하여 필요한 부분만 변경

(4) 현재 배포 상태 확인

```
kubectl get deployment
```

- 현재 네임스페이스에서 Deployment들의 요약 상태 조회

5) Service.yaml 설정

(1) 목적

Deployment가 띄운 app=nginx 파드 집합에 안정적 엔드포인트를 부여하고, NodePort(30087)로 외부에서 접근 가능하게 한다.

(2) yaml 파일 생성

```
vi service.yaml
```

(3) yaml 파일 수정

```
apiVersion: v1
kind: Service
metadata:
  name: kjh-service
spec:
  type: NodePort
  selector:
    app : nginx
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30087
```

- 외부 진입포트 : 30087로 설정
- selector는 deployment 와 일치시킴

(4) 확인

```
kubectl get service
```

- ClusterIP/NodePort 및 ENDPOINTS(파드 연결) 확인

6) 배포 및 확인

133.186.229.182:30087

- http://133.186.229.182:30087 접속 시 NGINX 기본 페이지가 출력
- Control plane의 floating IP와 NodePort로 적은 포트번호 조합으로 배포한 샘플앱에 웹으로 접근 가능

2. 포털 배포

1) 포털 사용

- 포털은 프로젝트/네임스페이스/권한을 웹 UI에서 일괄 관리하고, Keycloak(인증)/Harbor(레지스터리) 와 연동된 플랫폼 진입점이다.
- kubectl 명령 없이도 이미지/배포/환경변수/시크릿을 품 기반으로 설정/추적 가능하다

2) 포털 배포 방법

(1) 컨테이너 플랫폼 포털 Deployment 파일 다운로드

```
# Deployment 파일 다운로드 경로 생성  
$ mkdir -p ~/workspace/container-platform  
$ cd ~/workspace/container-platform  
  
# Deployment 파일 다운로드 및 파일 경로 확인  
$ wget --content-disposition https://nextcloud.k-paas.org/index.php/s/x7ccTRQYrBHsTD4/download  
  
$ ls ~/workspace/container-platform  
cp-portal-deployment-v1.6.2.tar.gz  
  
# Deployment 파일 압축 해제  
$ tar -xvf cp-portal-deployment-v1.6.2.tar.gz
```

(2) 포털 변수 정의

```
$ cd ~/workspace/container-platform/cp-portal-deployment/script  
$ vi cp-portal-vars.sh
```

- 파일 접근

```
K8S_MASTER_NODE_IP="133.186.229.182"  
K8S_CLUSTER_APISERVER="https://${K8S_MASTER_NODE_IP}:6443"  
K8S_STORAGECLASS="cp-storageclass"  
HOST_CLUSTER_IAAS_TYPE="4"  
HOST_DOMAIN="133.186.241.55.nip.io"
```

- 컨트롤플레인의 플로팅IP 설정, Ingress 외부 IP 기반 도메인 등을 설정

(3) 배포 스크립트 실행

```
$ chmod +x deploy-cp-portal.sh  
$ ./deploy-cp-portal.sh
```

(4) 포털 pod 조회 예시

```
$ kubectl get pods -n cp-portal
```

NAME	READY	STATUS	RESTARTS	AGE
cp-portal-api-deployment-8c4d87657-58rr9	1/1	Running	0	3m19s
cp-portal-catalog-api-deployment-54f56948bb-m17gz	1/1	Running	0	3m19s
cp-portal-chaos-api-deployment-7d9959c57c-42hpq	1/1	Running	0	3m19s
cp-portal-chaos-collector-deployment-6ff45d89d4-2v57b	1/1	Running	0	3m19s
cp-portal-common-api-deployment-65b87c5ddb-h8b42	1/1	Running	0	3m19s
cp-portal-metric-api-deployment-69ccb775-6gm26	1/1	Running	0	3m19s
cp-portal-terraman-deployment-599795db7d-8m97k	1/1	Running	0	3m19s
cp-portal-ui-deployment-5b5f465f7b-nnhpg	1/1	Running	0	3m19s

- 모든 파드가 Running/Ready 인지 확인

4) 포털 사용

(1) 접속/ 로그인

kubectl get ingress -A

- ingress-nginx 또는 cp-portal 관련 Ingress의 HOST/ADDRESS를 확인
- Keycloak 로그인으로 포털 진입

(2) namespace 생성

List		생성	
Name	Labels	Status	Created Time
kjh	kubernetes.io/metadata.name : kjh	Active	2025-10-21 15:46:57

- 같은 클러스터 안에서 프로젝트별로 오브젝트, 권한, 리소스쿼터를 논리적으로 분리해 충돌과 오남용을 방지

(3) deployment 확인

List		생성	
Name	Namespaces	Pods	Images
kjh-deployment	kjh	5 / 5	nginx

- 사용 이미지(nginx)가 기대와 일치하는지 확인

(4) service 확인

List		생성	
Name	Namespaces	Type	Clusters IP
kjh-service	kjh	NodePort	10.233.32.14

- Type=NodePort, ClusterIP, Port=80:30087/TCP 를 확인

3. 파이프라인 사용

1) 파이프라인 포털 사용 목적

소스→빌드→테스트→배포 과정을 한 화면에서 단계(집)로 정의하고, 실행 이력,로그를 추적

2) 파이프라인 배포 실습

(1) 작업 생성

The screenshot shows the 'Pipeline Configuration' screen for a project named 'spring-music'. In the 'Build' step, the 'Job Type' is set to 'Gradle'. The 'Docker File' field contains a Gradle build command. Below it, the 'Deployment Strategy' section is set to 'Gradle'.

- 잡 이름/유형/트리거 를 지정하고, 저장소 연결(브랜치/토큰/웹훅) 및 도커파일 경로를 설정
- 빌드 옵션(Gradle/Maven 명령, 환경변수/시크릿)과 이미지 태그 규칙을 정의해 재현성을 확보
- 배포 대상을 선택하여 빌드 완료 후 자동 배포되도록 연결

(2) 작업 실행

The screenshot shows the 'Pipeline Execution' screen with four stages: 1. Build job, 2. Analysis job, 3. Test (JUnit-Test), and 4. Deployment job. Each stage has a status icon and a green progress bar indicating the pipeline is running.

- 실행 중 실시간 로그(빌드 로그, 이미지 푸시)와 상태 아이콘(대기/진행/성공/실패) 으로 진행 상황을 확인

(3) 실행완료

The screenshot shows the 'Pipeline Execution' screen after completion. All four stages (Build, Analysis, Test, Deployment) are now marked as successful with green checkmarks and green progress bars.

4. Openbao Unseal 방법

1) 필요성

OpenBao(Vault 계열 비밀관리)는 기동 시 'Sealed(암호화된 상태)'로 올라옵니다. 이때 저장소의 마스터키가 분할되어 있어 퀘럼 수만큼 Unseal Key를 주입해야만 복호화가 시작된다

2) unseal 방법

(1) 배포된 vault의 unseal key 확인

```
$ cat ~/workspace/container-platform/cp-portal-deployment/secmg/unseal-key
```

(2) openbao-0 pod 접속

```
| $ kubectl exec -it openbao-0 -n openbao -- sh
```

(3) vault unseal 상태로 변경

```
| / $ vault operator unseal
```

- 두번 실행하여 첫번째, 두번째 키 입력

```
/ $ exit
```

- 셸 종료

3) 정상 작동 확인

```
$ kubectl get pod -n openbao
```

NAME	READY	STATUS	RESTARTS	AGE
openbao-0	1/1	Running	5	6d18h
openbao-agent-injector-6567764cc9-7zx8l	1/1	Running	3	4d20h

- 모두 Running 인 것 확인

4) session 제한 해제

(1) 필요성

동일 사용자가 여러 기기/브라우저에서 동시에 로그인해야 하는 실습상황에서, 기본 동시 세션 수 제한이 걸려 있으면 새 로그인이 차단된다. 따라서, 동시 세션 제한을 완화해 개발을 원활히 진행한다.

(2) keycloak 주소 확인

```
$ kubectl get ing -n keycloak
NAME      CLASS      HOSTS           ADDRESS      PORTS      AGE
keycloak   nginx     keycloak.133.186.215.162.nip.io  192.168.9.8  80, 443   6d18h
```

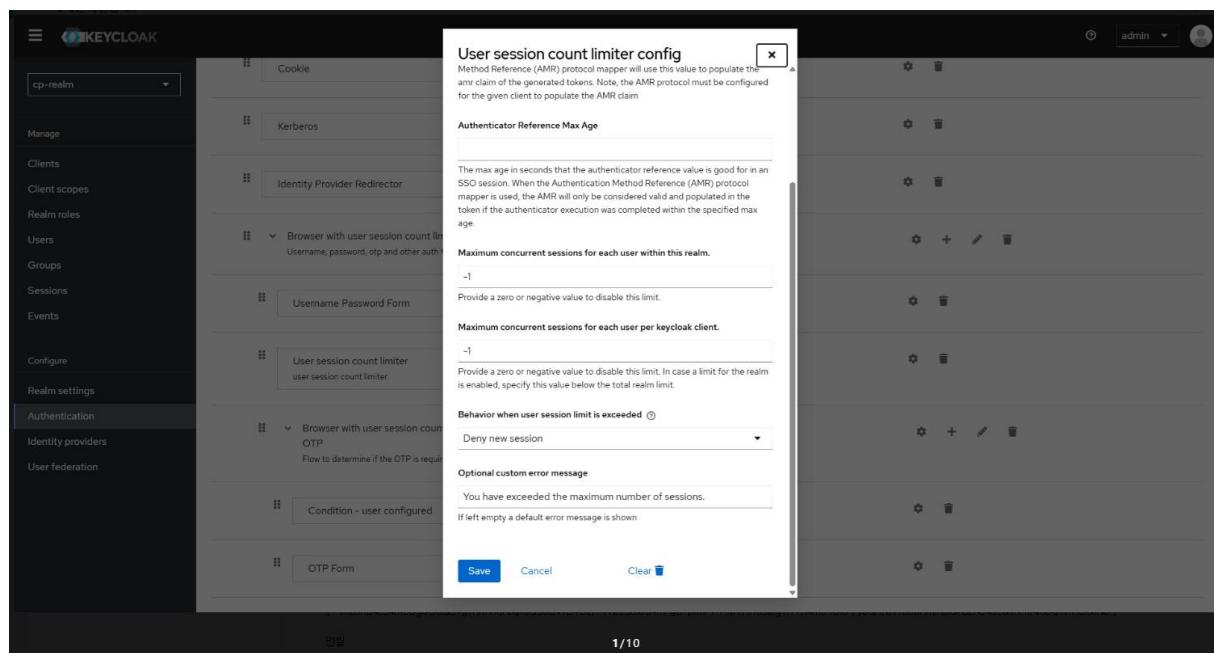
(3) HOST 접속

ID: admin

PWD: admin

- 기본 id 및 pw

(4) 설정



Authentication ->

Browser with user session count limiter ->

User session count limiter의 설정 아이콘 클릭 ->

Maximum concurrent sessions for each user per keycloak client.를 '-1'로 변경

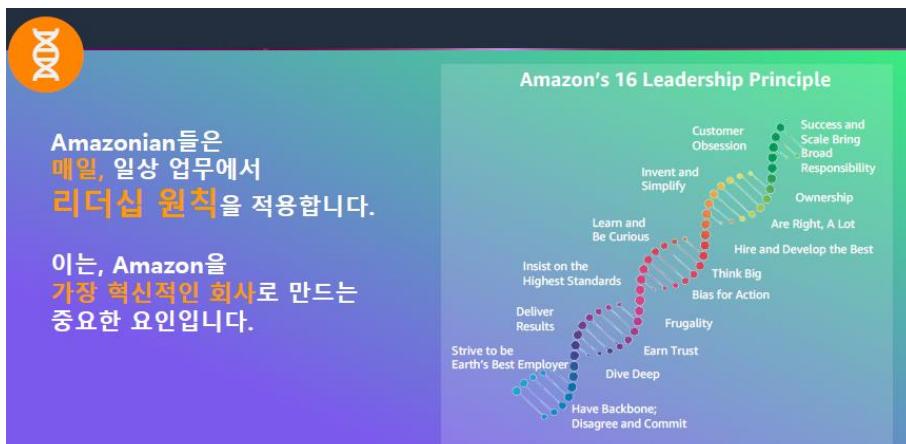
III-1. AWS 소개

1. Amazon Vision

우리의 비전은 지구상에서
가장 고객 중심적인 회사가 되는 것이며,
사람들이 온라인에서
사고 싶은 모든 것을 찾고 발견할 수
있는 장소를 만드는 것입니다.

아마존의 비전은 가장 고객 중심적인 회사가 되어, 사람들이 온라인에서 원하는 것을 찾고, 발견 및 구매하게 돋는 것이다. 이 철학은 제품·조직·채용 전반을 관통하며, 고객 여정에서 문제를 역산 (Working Backwards) 하여 해결한다. AWS 역시 같은 원칙으로 가용성/보안/성능/비용을 고객 경험 지표로 관리한다.

2. AWS 리더쉽 원칙



- 16개 원칙은 일상 의사결정의 규칙으로, 고객 집착/소유권/발명과 단순화·Dive Deep·높은 기준이 기술 조직의 기본 태도이다.
- 실무에선 Bias for Action으로 빠르게 실험하고, 관측 지표로 검증한 뒤 안정 구간을 Savings Plans/RI로 전환해 책임 있게 최적화한다.
- 결과적으로 Think Big을 유지하되 품질 기준을 높게 잡아, 대규모 확장과 보안을 전제로 한 클라우드 아키텍처를 설계한다.

3. IT B2B 생태계 용어 정리

1) CSP(Cloud Service Provider)

: 클라우드 인프라와 관리형 서비스를 제공(AWS, Azure, GCP / 국내: NHN 등).

2) MSP(Managed Service Provider)

: 설계·마이그레이션·운영/모니터링·비용최적화까지 대행/지원(예: 메가존, 베스핀 등).

3) ISV(Independent Software Vendor)

: SaaS/소프트웨어를 만들어 공급(Sendbird 등).

실무에선 CSP(기반) ↔ ISV(솔루션)를 MSP가 설계, 운영으로 연결하는 구조

4. AWS 선택 가이드

1) EC2 인스턴스 선택

(1) 워크로드 특성 파악

: CPU 중심/메모리 중심/가속 필요/HPC/저지연 I/O 여부를 구분

(2) 크기/영역 결정

: 피크 대비 60~70% 목표로 사이징, 다중 AZ 배치로 가용성 확보.

(3) 스토리지 결합

: EBS 탑재/IOPS/처리량과 용량 계획을 동시에 결정

(4) 검증/튜닝

: CloudWatch 지표로 CPU/메모리/디스크/네트워크를 관찰하여 패밀리 사이즈를 재선정

2) 비용모델 선택

(1) On-Demand

: 약정 없음, 실험·변동 부하에 적합(비용은 가장 높음).

(2) Savings Plans

: 지출 약정(시간당 \$)으로 폭넓은 할인. Compute SP는 인스턴스 리전 변경에도 적용 유연.

(3) Reserved Instances(RI)

: 용량 약정. Standard는 할인 최대(유연성 낮음), Convertible은 변경 가능(할인 다소 낮음)

(4) Spot

: 미사용 용량 경매형, 중단 허용 워크로드에 저가.

(5) 선택상황

- 상시/예측 가능 : SP(Compute) 또는 Standard RI.

- 변동/파일럿 : On-Demand 시작 → 지표 축적 후 SP 전환.

- 배치/HPC, 내결함성 : Spot 혼용으로 TCO 절감.

3) 스토리지 선택

(1) EBS(블록)

: EC2 개별 인스턴스에 붙는 디스크. DB/트랜잭션 워크로드, IOPS/처리량 튜닝. 스냅샷/암호화.

(2) S3(오브젝트)

: 데이터 레이크/백업/정적자산. 버전관리, 수명주기(Lifecycle)로 클래스 전환.

(3) EFS(네트워크 파일/NAS)

: 여러 EC2에서 동시 마운트(RWX). 웹 앱 공유 디렉터리, 홈디렉터리 등에 적합.

(4) FSx for Lustre(병렬 파일)

: HPC/ML에 고처리량·저지연, S3와 고속 연동으로 대규모 데이터 전처리 가능.

(5) 선택 상황

- S3: 접근 유형에 따라 클래스 선택 + 버전관리/수명주기로 비용/보존 최적화.

- EFS: NAS처럼 사용 → 멀티 AZ, 자동 확장, 권장: 보안그룹/암호화/KMS.

- EBS: 성능은 IOPS/처리량으로 설계 → gp3 기본, 고성능은 io1/io2.

III-2. Amazon EKS로 WEB APP 구축

1. 전체흐름

이 워크숍은 로컬에서 애플리케이션 이미지를 빌드하고 보안 스캔, 최적화를 거쳐 Amazon ECR에 푸시한 뒤, EKS 클러스터를 생성하고 노드 그룹을 구성하여 Kubernetes 매니페스트(Deployment/Service/Ingress)로 애플리케이션을 배포하는 순서로 진행된다. 외부 노출은 ALB Ingress Controller를 사용해 도메인, TLS를 포함한 L7 진입점을 마련하고, Horizontal Pod Autoscaler(HPA) 와 Cluster Autoscaler 설정으로 트래픽 변화에 자동 대응하도록 만든다. 배포 후에는 CloudWatch/Container Insights(혹은 Prometheus/Grafana)로 로그를 수집해 상태를 관찰하고, 필요 시 CodeBuild/CodePipeline 등으로 CI/CD를 연계해 소스 변경이 자동으로 빌드→이미지 푸시→쿠버네티스 배포까지 이어지도록 구성한다. 또한, 비용과 보안을 위해 ECR 라이프사이클, Pod 보안 컨텍스트, IAM Roles for Service Accounts(IRSA), 리소스 요청/제한을 점검하며, IaC를 통해 동일 구성을 반복 재현할 수 있음을 확인합니다.

이 과정을 통해 “컨테이너 이미지 → 레지스트리 → 쿠버네티스 배포 → 네트워크 노출 → 관찰성 → 자동화”로 이어지는 클라우드 네이티브 서비스 공급망을 끝단까지 한 번에 이해할 수 있었습니다. 특히 EKS에서는 애플리케이션 확장(HPA)과 인프라 확장(Cluster Autoscaler)이 분리되어 조합된다는 점, Ingress(앱 레벨 L7)와 LoadBalancer(서비스 레벨 L4/7)의 역할 차이가 설계 결정에 직접적이라는 점, 그리고 보안,비용,운영 품질은 배포 후가 아니라 이미지,매니페스트,파이프라인 단계에서 표준화함으로써 담보된다는 점을 알 수 있었다. 나아가 EC2,ECR,ALB,IAM,CloudWatch 등 AWS 기본 구성요소가 EKS와 어떻게 맞물리는지가 명확해져, 이후 현업 설계 시에도 동일한 패턴(이미지 빌드 표준화, 레지스트리 정책, Ingress 전략, 관찰성·자동화 가드레일)을 재사용할 수 있다고 생각했다.

2. 실습환경 구축

1) kubectl 설치

```
sudo curl -o /usr/local/bin/kubectl \
https://s3.us-west-2.amazonaws.com/amazon-eks/1.33.0/2025-05-01/bin/linux/amd64/kubectl
```

```
sudo chmod +x /usr/local/bin/kubectl
```

- 설치

```
kubectl version --client=true
```

- 설치 확인

2) eksctl 설치

```
curl --silent --location "https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp
```

- eksctl 바이너리 다운

```
sudo mv -v /tmp/eksctl /usr/local/bin
```

- 위치 옮김

```
eksctl version
```

- 설치 확인

3. 도커 컨테이너 이미지 제작

1) Dockerfile 생성 : 컨테이너 이미지 만들기 위한 설정파일

```
cd ~/environment/  
  
cat << EOF > Dockerfile  
FROM nginx:latest  
RUN echo '<h1> test nginx web page </h1>' >> index.html  
RUN cp /index.html /usr/share/nginx/html  
EOF
```

- root 폴더에서 생성

2) docker build : 이미지 생성

```
docker build -t test-image .
```

- 빌드 명령어

```
docker images
```

- 생성된 이미지 확인 가능

3) docker run : 이미지 컨테이너 실행

```
docker run -p 8080:80 --name test-nginx test-image
```

4) docker ps : 실행중인 컨테이너 확인

```
docker ps
```

5) docker logs : 컨테이너의 로그 출력

```
docker logs -f test-nginx
```

6) docker exec : 컨테이너 내부 쉘 환경으로 접근

```
docker exec -it test-nginx /bin/bash
```

7) docker stop : 실행중인 컨테이너 중지

```
docker stop test-nginx
```

8) docker rm : 컨테이너 삭제

```
docker rm test-nginx
```

9) docker rmi : 컨테이너 이미지 삭제

```
docker rmi test-image
```

4. ECR에 이미지 올리기 : 생성한 컨테이너 이미지를 레포지토리에 저장

1) 소스코드 다운

```
git clone https://github.com/joozero/amazon-eks-flask.git
```

2) 이미지 레포지토리 생성 -> Amazon ECR 콘솔에서도 확인 가능

```
aws ecr create-repository \
--repository-name demo-flask-backend \
--image-scanning-configuration scanOnPush=true \
--region ${AWS_REGION}
```

3) 인증 토큰 get 및 docker login

```
aws ecr get-login-password --region ${AWS_REGION} |
```

```
docker login --username AWS --password-stdin $ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com
```

4) docker build : 소스로 도커 이미지 빌드

```
cd ~/environment/amazon-eks-flask
```

```
docker build -t demo-flask-backend .
```

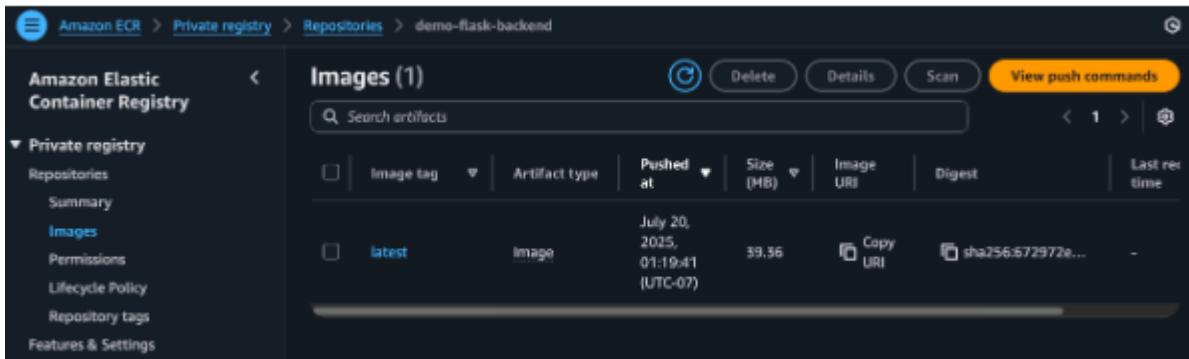
5) docker tag : 이미지가 레포지토리에 푸쉬되도록 준비

```
docker tag demo-flask-backend:latest $ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/demo-flask-backend:latest
```

6) docker push : 이미지 레포지토리에 푸쉬

```
docker push $ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/demo-flask-backend:latest
```

7) 콘솔창에서 확인



5. eksctl로 배포 클러스터 생성

1) eksctl : EKS 의 공식 CLI

(1) eksctl은 EKS 전용 라이프사이클 도구로, 하나의 명령 또는 ClusterConfig YAML로 VPC·보안그룹·컨트롤 플레인, 노드그룹, 애드온을 IaC 방식으로 일괄 생성·삭제하는 도구이다.

(2) CloudFormation을 백엔드로 사용하며 생성 완료 시 kubeconfig 컨텍스트를 자동 갱신해 즉시 kubectl 조회가 가능해지는 구조이다.

2) Config 파일 생성

```
cd ~/environment

cat << EOF > eks-demo-cluster.yaml
---
apiVersion: eksctl.io/v1alpha5
kind: ClusterConfig

metadata:
  name: eks-demo
  region: ${AWS_REGION}
  version: "1.33"

vpc:
  cidr: "10.0.0.0/16"
  nat:
    gateway: HighlyAvailable

managedNodeGroups:
  - name: node-group
    instanceType: m5.large
    desiredCapacity: 3
    volumeSize: 20
    privateNetworking: true
    iam:
      withAddonPolicies:
        imageBuilder: true
        awsLoadBalancerController: true
        cloudWatch: true
        autoScaler: true
        ebs: true

    cloudWatch:
      clusterLogging:
        enableTypes: ["*"]
EOF
```

- ClusterConfig YAML에는 클러스터 이름/리전/버전, VPC CIDR·NAT, 노드그룹 타입/용량/프라이빗 네트워킹, CloudWatch 로깅과 IRSA/애드온 권한을 선언

3) 클러스터 배포

```
eksctl create cluster -f eks-demo-cluster.yaml
```

- 실행 전 IAM 권한, 서비스 한도, 서브넷/퍼블릭 IP 옵션을 점검

4) 노드가 배포되었는지 확인

```
kubectl get nodes
```

- kubectl get nodes에서 워커가 Ready로 보이면 노드그룹 조인이 정상이라는 뜻이다

6. Ingress Controller 생성 - AWS Load Balancer Controller 생성

1) manifest 폴더 생성

```
cd ~/environment  
  
mkdir -p manifests/alb-ingress-controller && cd manifests/alb-ingress-controller
```

- ALB 컨트롤러 관련 매니페스트/Helm values 파일을 한곳에 모아 형상(버전)과 환경값을 분리하기 위한 작업이다. 디렉터리를 분리해 두면 이후 값 변경을 파일 단위로 재적용/롤백하기에 용이하다.

2) IAM OIDC identity Provider 생성

```
eksctl utils associate-iam-oidc-provider \  
--region ${AWS_REGION} \  
--cluster eks-demo \  
--approve
```

- EKS 클러스터의 OIDC Issuer를 IAM에 등록해 IRSA를 가능하게 한다. 이를 통해 ALB 컨트롤러 파드가 노드 역할이 아닌 전용 역할로 최소 권한을 받아 AWS API를 호출할 수 있다.

3) IAM Policy 생성

```
curl -O https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balancer-controller/v2.13.3/docs/install/iam_policy.json
```

```
aws iam create-policy \  
--policy-name AWSLoadBalancerControllerIAMPolicy \  
--policy-document file://iam_policy.json
```

- 공식 iam_policy.json을 내려 받아 ALB 리소스 조작에 필요한 최소 권한을 갖는 정책을 만든다.

4) Service Account 생성

```
eksctl create iamserviceaccount \  
--cluster eks-demo \  
--namespace kube-system \  
--name aws-load-balancer-controller \  
--attach-policy-arn arn:aws:iam::$ACCOUNT_ID:policy/AWSLoadBalancerControllerIAMPolicy \  
--override-existing-serviceaccounts \  
--region ${AWS_REGION} \  
--approve
```

- kube-system 네임스페이스의 SA 와 연결된 IAM Role 을 동시 생성 및 연결한다. 이후 컨트롤러 파드가 이 SA로 실행되어 OIDC 토큰으로 역할을 Assume 하고, 앞서 만든 정책 권한으로 ALB를 자동 생성, 동기화할 수 있다.

7. Ingress Controller - Cluseter에 Controller 추가

1) cert-manager 설치

```
kubectl apply --validate=false -f https://github.com/cert-manager/cert-manager/releases/download/v1.12.3/cert-manager.yaml
```

- Ingress의 TLS 인증서 발급/갱신 자동화를 위해 설치하는 구성요소이다.
- 인증서 수명주기를 쿠버네티스에서 선언형으로 관리할 수 있게 한다.

2) Load balancer controller yaml 파일 다운

```
curl -Lo v2_13_3_full.yaml https://github.com/kubernetes-sigs/aws-load-balancer-controller/releases/download/v2.13.3/v2_13_3_full.yaml
```

- 고정된 매니페스트를 받아 Deployment, Webhook 등 컨트롤러 구성 전부를 일괄 적용한다.

3) Service Account 섹션 제거

```
sed -i.bak -e '730,738d' ./v2_13_3_full.yaml
```

- eksctl create iamserviceaccount 로 IRSA가 연결된 SA를 이미 만들었으므로, 매니페스트의 SA 정의를 제거해 중복 생성/충돌을 방지

4) Deployment 섹션의 클러스터 이름 변경

```
sed -i.bak -e 's/your-cluster-name/eks-demo/' ./v2_13_3_full.yaml
```

- 컨트롤러 컨테이너의 --cluster-name 인자를 실제 클러스터명으로 바꿔 감시 대상을 지정

5) AWS Load Balancer controller 파일 배포

```
kubectl apply -f v2_13_3_full.yaml
```

- Ingress 리소스 ↔ ALB 리소스 간 상태를 동기화하는 컨트롤 루프를 활성화

6) IngressClass, IngressClassParams 매니페스트 다운로드 및 적용

```
curl -Lo v2_13_3_ingclass.yaml https://github.com/kubernetes-sigs/aws-load-balancer-controller/releases/download/v2.13.3/v2_13_3_ingclass.yaml
```

```
kubectl apply -f v2_13_3_ingclass.yaml
```

- Kubernetes IngressClass 를 정의해 이 컨트롤러가 처리할 기본 클래스와 동작 파라미터 설정

7) 배포 확인

```
kubectl get deployment -n kube-system aws-load-balancer-controller
```

```
kubectl get sa aws-load-balancer-controller -n kube-system -o yaml
```

- 레플리카, Ready 확인

7. 서비스 배포 - flask 백엔드 배포

1) manifests 폴더 위치로 이동

```
cd ~/environment/manifests/
```

- 배포에 필요한 Deployment, Service, Ingress 파일을 한 디렉터리에 모아 형상관리와 재적용을 쉽게 하기 위해 이 폴더에 생성한다.

2) deploy manifest 생성

```
cat <<EOF> flask-deployment.yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-flask-backend
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: demo-flask-backend
  template:
    metadata:
      labels:
        app: demo-flask-backend
    spec:
      containers:
        - name: demo-flask-backend
          image: $ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/demo-flask-backend:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
EOF
```

- ECR 에 푸시한 이미지를 참조하여 replicas, 라벨, 컨테이너 포트(8080)를 선언하고 원하는 상태(파드 수.이미지)를 정의한다

3) service manifest 생성

```
cat <<EOF> flask-service.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: demo-flask-backend
  annotations:
    alb.ingress.kubernetes.io/healthcheck-path: "/contents/aws"
spec:
  selector:
    app: demo-flask-backend
  type: NodePort
  ports:
    - port: 8080 # 서비스가 생성할 포트
      targetPort: 8080 # 서비스가 접근할 pod의 포트
      protocol: TCP
EOF
```

- type: NodePort 로 클러스터 내부 파드 집합에 안정적 엔드포인트를 부여하고, ALB 가 붙은 타깃 포트(8080)를 노출한다.

4) ingress manifest 생성

```
cat <<EOF> flask-ingress.yaml
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: "flask-backend-ingress"
  namespace: default
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/group.name: eks-demo-group
    alb.ingress.kubernetes.io/group.order: '1'
spec:
  ingressClassName: alb
  rules:
  - http:
    paths:
    - path: /contents
      pathType: Prefix
      backend:
        service:
          name: "demo-flask-backend"
          port:
            number: 8080
EOF
```

- AWS Load Balancer Controller 가 ALB/리스너/타깃그룹을 자동 프로비저닝하도록 지시

5) 생성한 manifest 배포 -> AWS ALB 프로비저닝

```
kubectl apply -f flask-deployment.yaml
kubectl apply -f flask-service.yaml
kubectl apply -f flask-ingress.yaml
```

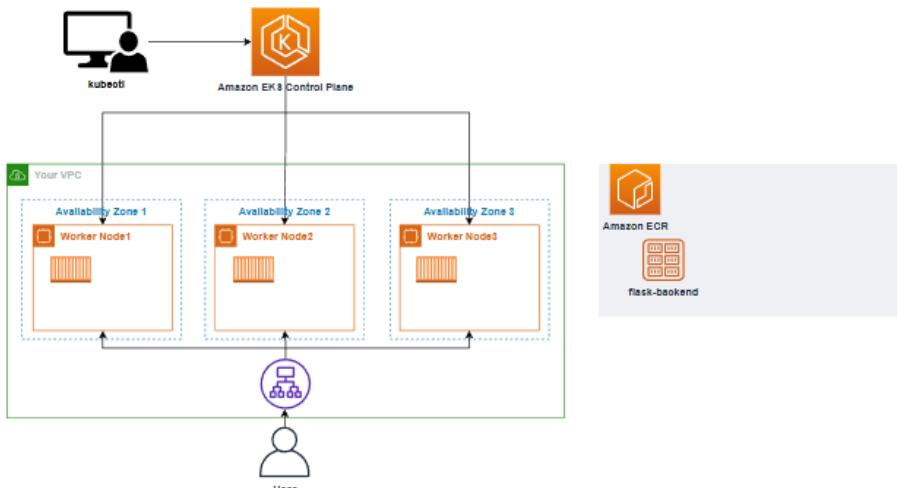
- 세 파일을 적용하면 컨트롤러가 이벤트를 감지해 ALB 리소스를 생성한다.

6) 확인

```
echo http://$(kubectl get ingress/flask-backend-ingress -o jsonpath='[.status.loadBalancer.ingress[*].hostname]')/contents/aws
```

- HTTP 응답을 확인

7) 현재 아키텍쳐



- 사용자는 ALB DNS 로 접속하고, ALB 가 Ingress 규칙에 따라 Service(NodePort)→Pod(Flask) 로 트래픽을 전달한다.
- 이미지는 ECR 에서 Pull 되며, 필요 시 HPA/Cluster Autoscaler 로 앱, 인프라 확장을 분리해 운영한다.

8. 서비스 배포 - express 백엔드 배포

1) manifests 폴더로 이동

```
cd ~/environment/manifests/
```

2) deploy manifest 생성

```
cat <<EOF> nodejs-deployment.yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-nodejs-backend
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: demo-nodejs-backend
  template:
    metadata:
      labels:
        app: demo-nodejs-backend
    spec:
      containers:
        - name: demo-nodejs-backend
          image: public.ecr.aws/y7c9e1d2/joozero-repo:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 3000
EOF
```

- ECR 이미지를 참조하여 replicas=3, 라벨, 컨테이너 포트(3000)를 선언한다.

3) service manifest 생성

```
cat <<EOF> nodejs-service.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: demo-nodejs-backend
  annotations:
    alb.ingress.kubernetes.io/healthcheck-path: "/services/all"
spec:
  selector:
    app: demo-nodejs-backend
  type: NodePort
  ports:
    - port: 8080
      targetPort: 3000
      protocol: TCP
EOF
```

- type: NodePort, targetPort: 3000 → port: 8080 으로 패드 집합 앞단에 안정적인 엔드포인트를 제공한다.

4) ingress manifest 생성

```
cat <<EOF nodejs-ingress.yaml
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: "nodejs-backend-ingress"
  namespace: default
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/group.name: eks-demo-group
    alb.ingress.kubernetes.io/group.order: '2'
spec:
  ingressClassName: alb
  rules:
  - http:
      paths:
      - path: /services
        pathType: Prefix
        backend:
          service:
            name: "demo-nodejs-backend"
            port:
              number: 8080
EOF
```

- AWS Load Balancer Controller가 ALB/리스너/타깃그룹을 자동 프로비저닝하도록 한다.

5) manifest 배포

```
kubectl apply -f nodejs-deployment.yaml
kubectl apply -f nodejs-service.yaml
kubectl apply -f nodejs-ingress.yaml
```

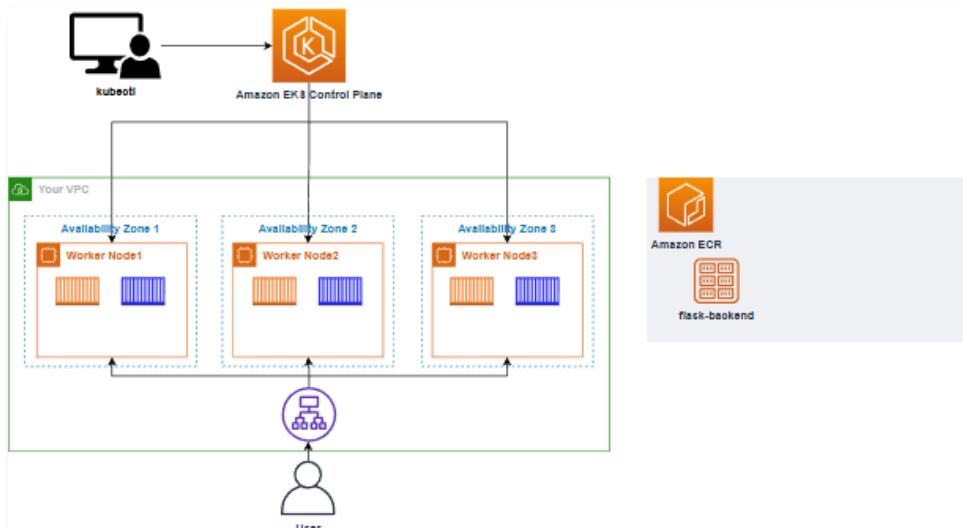
- 컨트롤러 이벤트를 트리거한다.

6) 수행결과 확인

```
echo http://$(kubectl get ingress/nodejs-backend-ingress -o jsonpath='{{.status.loadBalancer.ingress[*].hostname}}')/services/all
```

- 획득한 ALB 호스트네임에 /services/all 을 붙여 HTTP 응답을 확인한다.

7) 현재 아키텍쳐



- 사용자는 ALB DNS로 접속하고, ALB가 Ingress 규칙에 따라 NodePort Service를 통해 Express 파드(3000)로 트래픽을 전달한다.

9. 서비스 배포 - react 프론트엔드 배포

1) 컨테이너라이징할 소스코드 다운

```
cd /home/ec2-user/environment  
git clone https://github.com/joozero/amazon-eks-frontend.git
```

- Git 저장소에서 프론트엔드 소스를 받아 로컬 빌드→이미지화 준비

2) 이미지 레포지토리 생성

```
aws ecr create-repository \  
--repository-name demo-frontend \  
--image-scanning-configuration scanOnPush=true \  
--region ${AWS_REGION}
```

- ECR에 demo-frontend 리포지토리를 만들고 푸시 시 보안 스캔을 활성화한다.

3) url 변경

- 두 개의 백엔드 결과 값을 화면에 뿌리기 위해, 일부 소스 코드를 수정한다.
- 프론트엔드 소스 코드가 담긴 폴더(예: /home/ec2-user/environment/amazon-eks-frontend/src)에서 App.js 파일과 page/UpperPage.js 파일에 있는 url 값을 앞서 배포한 인그레스 주소로 변경한다.

4) npm 설치

```
cd /home/ec2-user/environment/amazon-eks-frontend  
npm install  
npm run build
```

- 정적 산출물(React 빌드)을 만든다.

5) 이미지 레포지토리 생성 및 이미지 푸시

```
docker build -t demo-frontend .  
  
docker tag demo-frontend:latest $ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/demo-frontend:latest
```

6) manifests에 yaml 생성

```
cd /home/ec2-user/environment/manifests

cat <<EOF> frontend-deployment.yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-frontend
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: demo-frontend
  template:
    metadata:
      labels:
        app: demo-frontend
    spec:
      containers:
        - name: demo-frontend
          image: $ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/ /demo-frontend:latest
          imagePullPolicy: Always
          ports:
            - containerPort: 80
EOF
```

```
cat <<EOF> frontend-service.yaml
---
apiVersion: v1
kind: Service
metadata:
  name: demo-frontend
  annotations:
    alb.ingress.kubernetes.io/healthcheck-path: "/"
spec:
  selector:
    app: demo-frontend
  type: NodePort
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
EOF
```

```
cat <<EOF> frontend-ingress.yaml
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: "frontend-ingress"
  namespace: default
  annotations:
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/group.name: eks-demo-group
    alb.ingress.kubernetes.io/group.order: '3'
spec:
  ingressClassName: alb
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: "demo-frontend"
                port:
                  number: 80
EOF
```

7) manifest 배포

```
kubectl apply -f frontend-deployment.yaml  
kubectl apply -f frontend-service.yaml  
kubectl apply -f frontend-ingress.yaml
```

8) 정상 작동 확인

```
echo http://$(kubectl get ingress/frontend-ingress -o jsonpath='{.status.loadBalancer.ingress[*].hostname}')
```

The screenshot shows the EKS DEMO Blog homepage. At the top, there's a navigation bar with a cloud icon, the title "EKS DEMO Blog", and the time "9:03:51 AM". Below the navigation, there's a section titled "Based on Nodejs api" featuring icons for AWS services: AWS (aws logo), Amazon ECR (orange hexagon), Amazon EKS (orange hexagon with a 'K'), Amazon ECS (orange hexagon), AWS Fargate (orange hexagon with a cube), and Amazon Cloud9 (blue cloud with a '9'). Each service has a "SEE MORE" link below its icon. Below this section is a search bar with the placeholder "Enter your keyword to search" and a "CLICK" button. A green speech bubble icon with a checkmark and the text "Based on Flask api" is positioned above the search bar. The main content area contains a list of search results related to AWS services, each preceded by a small green checkmark icon.

- [AWS services explained in one line each](#)
- [AWS forked my project and launched it as its own service](#)
- [Amazon threatens to suspend Signal's AWS account over censorship circumvention](#)
- [Amazon LightSail: Simple Virtual Private Servers on AWS](#)
- [I want to have an AWS region where everything breaks with high frequency](#)
- [Earth on AWS – Open geospatial data](#)
- [Why I Turned Down an AWS Job Offer](#)
- [AWS Lambda pricing now per ms](#)
- [Google Cloud is 50% cheaper than AWS](#)
- [AWS Ground Station – Ingest and Process Data from Orbiting Satellites](#)
- [AWS Icon quiz](#)
- [AWS Tips I Wish I'd Known Before I Started](#)
- [CodeStar – Quickly Develop, Build, and Deploy Applications on AWS](#)
- [A Comprehensive Guide to Building a Scalable Web App on AWS](#)
- [AWS mistakes to avoid](#)
- [AWS Fargate – Run Containers Without Managing Infrastructure](#)
- [Networking on AWS \(2018\)](#)
- [AWS Lambda](#)
- [AWS, MongoDB, and the Economic Realities of Open Source](#)
- [Dropbox saved almost \\$75M over two years by moving out of AWS](#)

- ALB DNS에 접속해 화면 상단(노드/익스프레스), 하단(플라스크) 영역이 각 백엔드 결과로 렌더링되는지 확인한다

10. Container Insights 사용

1) manifest 관리용 폴더 생성

```
cd ~/environment  
mkdir -p manifests/cloudwatch-insight && cd manifests/cloudwatch-insight
```

- 관측(모니터링) 관련 매니페스트를 별도 디렉터리에 분리하여 형상과 환경값을 일관되게 관리할 수 있게 한다.

2) namespace 생성

```
kubectl create ns amazon-cloudwatch
```

- 네임스페이스에 CloudWatch Agent/Fluent Bit 리소스를 격리하여 권한/라벨/리소스쿼터를 독립적으로 적용한다.

3) Cloudwatch 에이전트, Fluent Bit 설치

```
ClusterName=eks-demo  
RegionName=$AWS_REGION  
FluentBitHttpPort='2020'  
FluentBitReadFromHead='Off'  
[[ ${FluentBitReadFromHead} = 'On' ]] && FluentBitReadFromTail ='Off'||| FluentBitReadFromTail='On'  
[[ -z ${FluentBitHttpPort} ]] && FluentBitHttpServer='Off' ||| FluentBitHttpServer='On'
```

- 클러스터명/리전/수집 모드(FromTail/FromHead)/Fluent Bit HTTP 서버 포트 등을 변수로 선언해 배포 시 자동 치환한다.

4) yaml 파일 다운로드

```
wget https://raw.githubusercontent.com/aws-samples/amazon-cloudwatch-container-insights/latest/k8s-deployment-manifest-templates/deployment-mode/daemonset/container-insights-monitoring/quickstart/cwagent-fluent-bit-quickstart.yaml
```

- AWS 샘플 리포지터리에서 DaemonSet 기반 QuickStart 매니페스트를 내려받아 에이전트와 Fluent Bit를 노드마다 실행한다.

5) 환경변수 적용

```
sed -i  
's/{{cluster_name}}/'${ClusterName}'/;s/{{region_name}}/'${RegionName}'/;s/{{http_server_toggle}}/'${FluentBitHttpServer}'"/;s/{{http_server_port}}/'${FluentBitHttpPort}'"/;s/{{read_from_head}}/'${FluentBitReadFromHead}'"/;s/{{read_from_tail}}/'${FluentBitReadFromTail}'"/' cwagent-fluent-bit-quickstart.yaml
```

- 템플릿의 플레이스홀더를 실제 값(클러스터명, 리전, 토큰, 포트 등)으로 치환하여 현 환경에 맞는 단일 매니페스트로 만든다.

6) deploy

```
kubectl apply -f cwagent-fluent-bit-quickstart.yaml
```

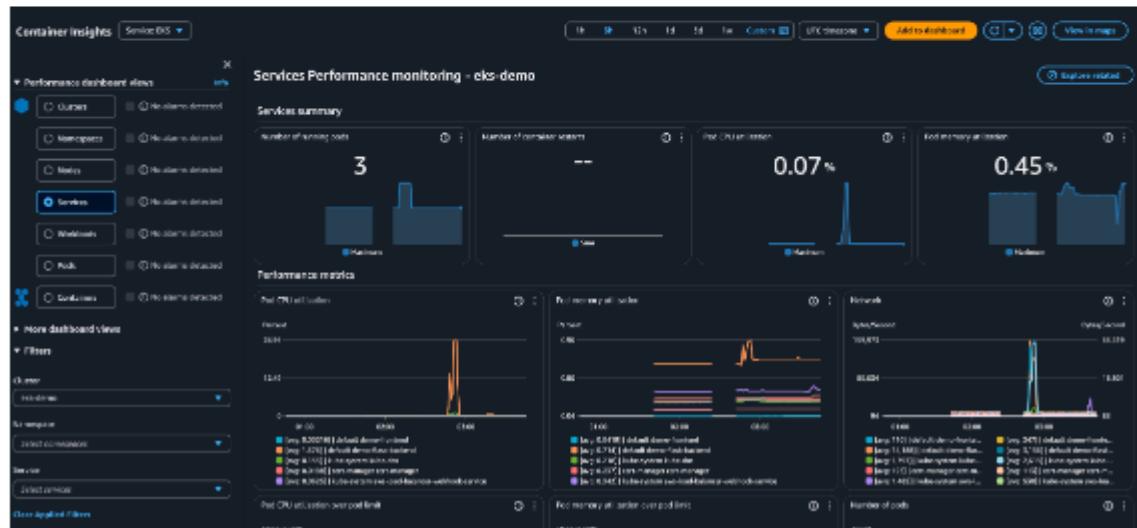
- DaemonSet/ConfigMap/ServiceAccount 등을 전개한다.

7) 정상 설치 확인

```
kubectl get po -n amazon-cloudwatch
```

- 노드당 에이전트/Fluent Bit 파드가 Running 인지 확인한다.

8) 콘솔에서 보기



- CloudWatch → Container Insights 대시보드에서 클러스터/노드/네임스페이스/서비스/파드 단위의 CPU/메모리/네트워크 지표와 에러율/재시작 횟수를 확인한다.
 - 문제 탐지 후 Logs Insights로 쿼리해 근본 원인을 추적할 수 있다.

11. Autoscaling Pod & Cluster – HPA 적용

1) Metrics Server 확인

```
kubectl get deployment metrics-server -n kube-system
```

- HPA는 CPU/메모리 등 리소스 메트릭을 Metrics API로 조회해 스케일 결정을 내리므로, kube-system에 Metrics Server가 배포·정상 동작 중이어야 한다. 이를 점검한다.

2) flask deployment yaml 수정

```
cd /home/ec2-user/environment/manifests
```

```
cat <<EOF> flask-deployment.yaml
```

```
---
```

```
apiVersion: apps/v1
```

```
kind: Deployment
```

```
metadata:
```

```
  name: demo-flask-backend
```

```
  namespace: default
```

```
spec:
```

```
  replicas: 1
```

```
  selector:
```

```
    matchLabels:
```

```
      app: demo-flask-backend
```

```
  template:
```

```
    metadata:
```

```
      labels:
```

```
        app: demo-flask-backend
```

```
    spec:
```

```
      containers:
```

```
        - name: demo-flask-backend
```

```
          image: $ACCOUNT_ID.dkr.ecr.$AWS_REGION.amazonaws.com/demo-flask-backend:latest
```

```
          imagePullPolicy: Always
```

```
        ports:
```

```
          - containerPort: 8080
```

```
      resources:
```

```
        requests:
```

```
          cpu: 250m
```

```
        limits:
```

```
          cpu: 500m
```

```
EOF
```

- resources.requests/limits 를 정의해 목표치(%)의 분모를 명확히 하고, 초기 replicas: 1 로 시작해 확장 효과를 관찰 가능하게 구성한다

3) yaml 적용

```
kubectl apply -f flask-deployment.yaml
```

- 수정된 요청/제한과 파드 수가 반영된다.

4) hpa 용 yaml 생성

```
cat <<EOF> flask-hpa.yaml
---
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: demo-flask-backend-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: demo-flask-backend
  minReplicas: 1
  maxReplicas: 5
  targetCPUUtilizationPercentage: 30
EOF
```

- scaleTargetRef 로 대상 Deployment 를 지정하고, minReplicas/maxReplicas 로 확장 범위를, targetCPUUtilizationPercentage: 30 으로 목표 평균 CPU 를 설정한다.

5) yaml 적용

```
kubectl apply -f flask-hpa.yaml
```

- HPA 객체가 생성된다. 즉시 스케일되지는 않으며 관측 주기 후 판단된다.

6) HPA 상태 확인

```
kubectl get hpa
```

- CURRENT/TARGET 지표와 MIN/MAX 범위를 확인하고, 세부 원인은 kubectl describe hpa/... 로 이벤트, 최근 스케일 이력을 본다

7) 부하 테스트

```
kubectl get hpa -w
```

```
curl -LO https://hey-release.s3.us-east-2.amazonaws.com/hey_linux_amd64
chmod +x hey_linux_amd64
sudo mv hey_linux_amd64 /usr/local/bin/hey
```

```
export flask_api=$(kubectl get ingress/flask-backend-ingress -o
jsonpath='{.status.loadBalancer.ingress[*].hostname}')/contents/aws
```

```
hey -n 20000 -c 1000 http://$flask_api
```

- hey 로 ALB Ingress 주소에 다중 동시 요청을 보내 CPU 사용률을 인위적으로 상승시켜 HPA 트리거를 검증한다.

12. Autoscaling Pod & Cluster – Cluster Autoscaler 적용

1) 클러스터 워커노드의 정보를 환경변수에 저장

```
export ASG_NAME=$(aws autoscaling describe-auto-scaling-groups \
    --query "AutoScalingGroups[?Tags[?Key=='eks:cluster-name' && Value=='eks-
    demo']].AutoScalingGroupName | [0]" \
    --output text)
```

```
echo ${ASG_NAME}
```

- eks:cluster-name 태그로 해당 EKS 노드그룹의 ASG 이름을 추출하여 ASG_NAME 변수에 담는다.

2) ASG의 값 확인

```
aws autoscaling describe-auto-scaling-groups \
    --auto-scaling-group-names "$ASG_NAME" \
    --query "AutoScalingGroups[].{AutoScalingGroupName, MinSize, MaxSize, DesiredCapacity}" \
    --output table
```

- 선택된 ASG의 Min/Max/Desired 용량을 조회해 현재 스케일 한계와 상태를 파악한다.

3) 오토스케일링 그룹 최대값을 5로 변경

```
aws autoscaling update-auto-scaling-group \
    --auto-scaling-group-name $ASG_NAME \
    --max-size 5
```

- 상한을 늘려 노드 증설 여지를 확보합니다. Max가 낮으면 Cluster Autoscaler가 확장을 요청해도 ASG가 거부하여 노드가 늘지 않는다.

4) Cluster Autoscaler 예제 다운

```
cd /home/ec2-user/environment/manifests
```

```
wget https://raw.githubusercontent.com/kubernetes/autoscaler/master/cluster-
autoscaler/cloudprovider/aws/examples/cluster-autoscaler-autodiscover.yaml
```

- 공식 매니페스트(autodiscover 버전)를 받아 Deployment/RBAC/파라미터를 기본값으로 구성한다.

5) 클러스터 이름 변경, 배포

```
sed -i 's|<YOUR CLUSTER NAME>|eks-demo|g' cluster-autoscaler-autodiscover.yaml
```

```
kubectl apply -f cluster-autoscaler-autodiscover.yaml
```

- 매니페스트의 --cluster-name 인자를 실제 EKS 이름(예: eks-demo)으로 치환한 뒤 적용하여 컨트롤러를 띄운다.

6) 부하테스트

```
kubectl get nodes -w
```

```
kubectl create deployment autoscaler-demo --image=nginx  
kubectl scale deployment autoscaler-demo --replicas=100
```

```
kubectl get deployment autoscaler-demo --watch
```

- nginx 100레플리카를 생성해 Pending 파드를 일부러 만들면 Autoscaler가 이를 감지하고 ASG Desired 용량을 늘린다.
- kubectl get nodes -w 와 kubectl get deployment autoscaler-demo --watch 로 노드 증설과 레플리카 가동을 실시간 확인한다.

7) 생성 파드 삭제하여 결과 확인

```
kubectl delete deployment autoscaler-demo
```

- 데모 Deployment를 삭제하면 사용량이 줄고 Autoscaler가 점차 Desired 용량을 축소하여 노드를 드레이닝 후 제거한다.
- 축소는 즉시가 아니라 안전 쿨다운을 거쳐 진행된다.

III-3. GitLab CI/CD 활용한 DevOps 실습

1. GitLab 설치

1) 사용자 변경

```
1 sudo su
```

- GitLab Omnibus 설치 스크립트는 패키지 추가/포트 바인딩/서비스 등록을 수행하므로 루트 권한이 필요하다.

2) 쉘 스크립트 다운로드

```
wget https://ws-assets-prod-iad-r-pdx-f3b3f9f1a7d6a3d0.s3.us-west-2.amazonaws.com/bc2ef89d-92fa-46f3-9d84-2f5b1547f122/install_gitlab.sh
```

- 제공된 install_gitlab.sh 는 의존 패키지 설치, 리포지토리 추가, GitLab Omnibus 설치를 일괄 자동화하는 스크립트이다.

3) 스크립트 실행

```
1 sh install_gitlab.sh
```

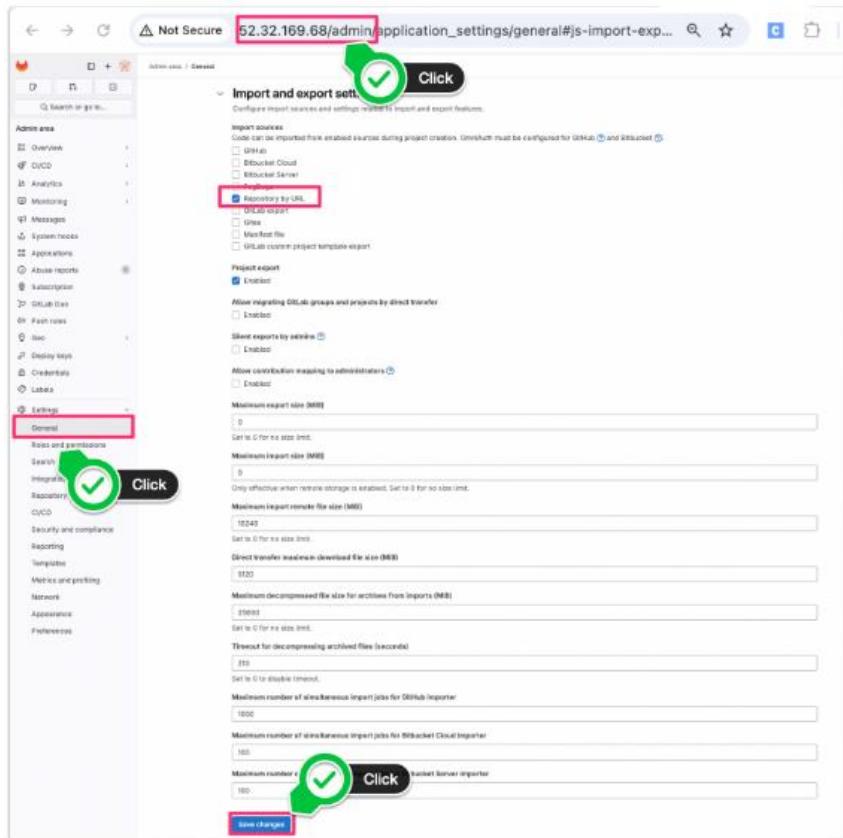
- external_url 설정, 방화벽/보안그룹의 80/443/22 개방, 서비스 구동(gitlab-ctl reconfigure)이 순차 진행된다.

4) GitLab 접속 및 관리자 계정 로그인

- 브라우저에서 https://<external_url> 로 접속하고, 초기 관리자 비밀번호는 /etc/gitlab/initial_root_password 에 생성되어 있다

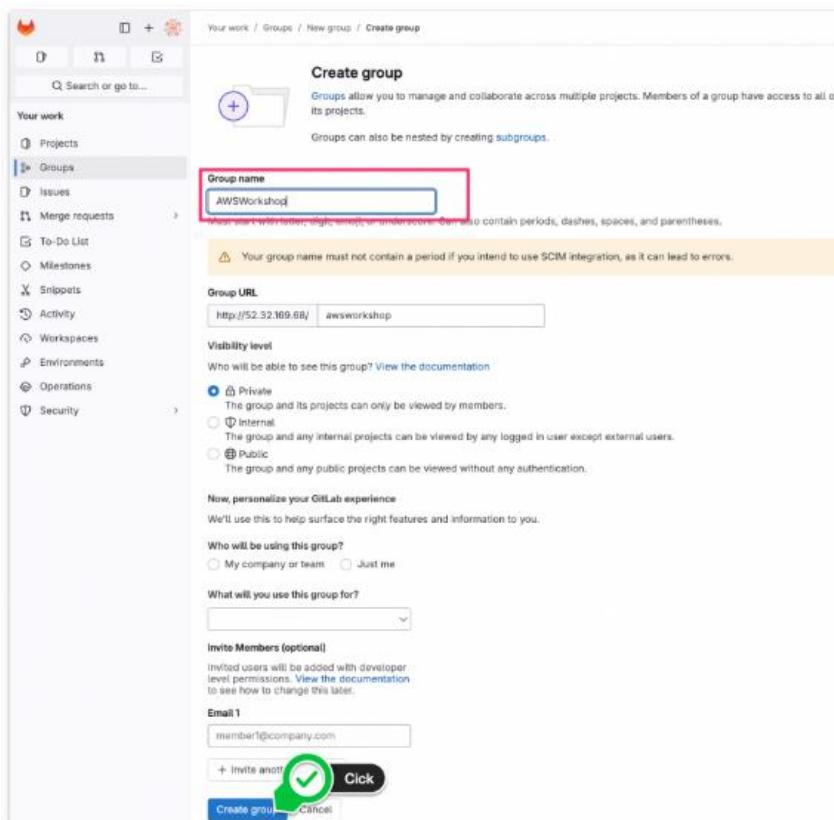
2. GitLab 환경설정 – Import Repository

1) Repository by URL 활성화



- Admin Area -> Settings -> General에서 설정

2) New Group 생성



- Groups -> New Group 이동
- Group name 지정후 Create group

3) Project 생성

Git repository URL

Username (optional)

Password (optional)

Project name

Must start with a lowercase or uppercase letter, digit, emoji, or underscore. Can also contain dots, pluses, dashes, or spaces.

Project URL

/

Project description (optional)

Description format

Visibility Level ?

Private
Project access must be granted explicitly to each user. If this project is part of a group, access is granted to members of the group.

Internal
The project can be accessed by any logged in user except external users.

Public
The project Click without any authentication.

Create project**Cancel**

- Create a project -> Import project -> Repository by URL 에 다음 URL 입력
(<https://github.com/gatsbyjs/gatsby-starter-default.git>)

3. GitLab 환경설정 – Runner 설치 및 등록

1) EC2 인스턴스 접속

- CI 작업을 실제로 수행할 실행 호스트(러너)를 준비하기 위해 SSH로 EC2에 접속한다.

2) Runner 다운로드, 설치

```
curl -L "https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh" | sudo bash
```

```
sudo apt-get install -y gitlab-runner
```

- 공식 스크립트 저장소를 등록 후 gitlab-runner 패키지를 설치한다.

3) Runner 서비스 시작 및 부팅시 자동 시작되도록 설정

```
1 sudo systemctl enable gitlab-runner  
2 sudo systemctl start gitlab-runner
```

- 서비스 상시 실행 상태를 보장한다.

4) Runner 등록

New project runner

Create a project runner to generate a command that registers the runner with all its configurations.

Tags

Tags

Add tags to specify jobs that the runner can run. [Learn more](#).

Separate multiple tags with a comma. For example, `macos, shared`.

Run untagged jobs

Use the runner for jobs without tags in addition to tagged jobs.

Configuration (optional)

Runner description

Paused

Stop the runner from accepting new jobs.

Protected

Use the runner on pipelines for protected branches only.

Lock to current projects

Use the runner for the currently assigned projects only. Only administrators can change the assigned projects.

Maximum job timeout

Maximum amount of time the runner can run before it terminates. If a project has a shorter job timeout period, the job timeout period of the instance runner is used instead.

Enter the job timeout in seconds. Must be a minimum of 600 seconds.

Create runner

- Settings -> CI/CD -> New project runner 이동 후 Create runner

```

root@ip-10-0-1-23:/var/snap/amazon-ssm-agent/9881# gitlab-runner register --url http://10.0.0.10 --token 3153ccc6
Runtime platform
arch=amd64 os=linux pid=191941 revision=17.7.0
Running in system-mode.

Enter the GitLab instance URL (e.g. https://gitlab.com/):
[http://10.0.0.10]
Verifying runner...
Enter a name for your runner (used only in the local config.toml file):
[ip-10-0-1-23]
Enter an executor: docker-autoscaler, instance, ssh, parallels, docker, docker-windows, kubernetes, custom, shell, virtualbox, docker+machine, shell
Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!
Configured authentication token) was saved in "/etc/gitlab-runner/config.toml"
root@ip-10-0-1-23:/var/snap/amazon-ssm-agent/9881#

```

- 프롬프트에서 Executor는 'shell'(실습 단순화), Run untagged jobs 체크로 태그 없는 잡도 처리하도록 설정한다.

5) Runner 정보 확인

AWS Workshop / Gatsby Starter Default / CI/CD Settings

Auto DevOps

Automate building, testing, and deploying your applications based on your continuous integration and delivery configuration. [How do I get started?](#)

Runners

Runners are processes that pick up and execute CI/CD jobs for GitLab. [What is GitLab Runner?](#)

Register as many runners as you want. You can register runners as separate users, on separate servers, and on your local machine.

How do runners pick up jobs?

Runners are either:

- **active** - Available to run jobs.
- **paused** - Not available to run jobs.

Tags control which type of jobs a runner can handle. By tagging a runner, you make sure runners only handle the jobs they are equipped to run. [Learn more.](#)

Project runners

These runners are assigned to this project.

[New project runner](#) [⋮](#)

Assigned project runners

#1 (-bt1sk1S)	Edit	Remove runner
---------------	----------------------	-------------------------------

Instance runners

These runners are available to all groups and projects.

[Enable instance runners for this project](#)

This GitLab instance does not provide any instance runners yet. Administrators can register instance runners in the admin area.

Group runners

These runners are shared across projects in this group.

Group runners can be managed with the [Runner API](#).

[Disable group runners](#) for this project

This group does not have any group runners yet. To register them, go to the [group's Runners page](#).

- 프로젝트 CI/CD → Runners 화면에서 상태가 online 으로 표시되는지 확인한다. 필요 시 러너에 태그를 부여해 특정 잡만 선택적으로 실행하게 하거나, 보호 브랜치 전용 실행은 Protected 옵션으로 제한한다.

4. 배포할 Application 준비

1) S3 버킷 생성

1. AWS 관리 콘솔에서 S3 서비스로 이동합니다.
2. **Create bucket** 버튼을 클릭합니다.
3. 버킷 이름을 입력합니다. 이름은 전역적으로 고유해야 합니다. 예: my-static-website-[AccountID]
4. 리전을(예: us-west-2) 선택합니다.
5. "Block all public access" 섹션에서 모든 옵션의 체크를 해제합니다. 정적 웹 사이트 확인을 하지 않고, 배포여부만 확인이 필요한 경우 이 단계는 생략 하실 수 있습니다. "Turning off block all public access might result in this bucket and the objects within becoming public" 옵션을 체크합니다.
6. 다른 모든 설정은 기본값으로 유지합니다.
7. **Create bucket** 버튼을 클릭합니다.

2) 정적 웹사이트 호스팅 활성화

1. 생성한 버킷을 클릭합니다.
2. **Properties** 탭으로 이동 후, 아래로 스크롤하여 "Static website hosting" 섹션을 찾습니다.
3. **Edit**버튼을 클릭합니다.
4. Static website hosting **Enable**를 선택합니다.
5. Hosting type 으로 **Host a static website**을 선택합니다.
6. 인덱스 문서로 index.html을 입력합니다.
7. **save changes** 버튼을 클릭합니다.

3) 버킷 정책 설정

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Sid": "PublicReadGetObject",  
            "Effect": "Allow",  
            "Principal": "*",  
            "Action": "s3:GetObject",  
            "Resource": "arn:aws:s3:::YOUR-BUCKET-NAME/*"  
        }  
    ]  
}
```

- Permissions → Bucket policy → Edit 에서 아래 정책의 YOUR-BUCKET-NAME 을 실제 버킷명으로 바꿔 저장한다.
- 저장 후 정책 시뮬레이터 경고가 없고, Objects can be public 표시가 나오면 정상이다.

4) IAM Policy 설정

The screenshot shows the 'Specify permissions' step of the IAM policy creation wizard. The JSON editor contains the following policy document:

```

1 * {  
2     "Version": "2012-10-17",  
3     "Statement": [  
4         {  
5             "Effect": "Allow",  
6             "Action": [  
7                 "s3:PutObject",  
8                 "s3:GetObject",  
9                 "s3>ListBucket",  
10                "s3>DeleteObject"  
11            ],  
12            "Resource": [  
13                "arn:aws:s3:::my-static-website/*",  
14                "arn:aws:s3:::my-static-website/*/*"  
15            ]  
16        }  
17    ]  
18 }

```

The 'Actions' tab is selected in the top right. A green checkmark icon is overlaid on the editor area. On the right side, there's a sidebar with the message 'Select a statement' and a button '+ Add new statement'. At the bottom, there are status indicators: Security: 0, Errors: 0, Warnings: 0, Suggestions: 0, and character count: 5905 of 6144 characters remaining.

- AWS 관리 콘솔 -> IAM 서비스 -> Policies -> Create Policy
- 정책이름 입력 후 create policy

5) EC2 인스턴스에 할당된 IAM Role 확인

The screenshot shows the EC2 Instances page with one instance listed:

Name	Instance ID	Instance State	Type	Status Checks	Launch Type
GitLabServer-cfn	i-0d1b3e9117ac55911	Running	m5.large	3/3 checks passed	On-Demand

Details view for the instance:

- Instance summary**:
 - Instance ID: i-0d1b3e9117ac55911
 - Public IPv4 address: [REDACTED] | open address
 - Private IP4 addresses: [REDACTED]
 - Public IPv4 DNS: [REDACTED].us-west-2.compute.amazonaws.com | open address
- IAM Role**: cfn-GitLabEC2InstanceRole (highlighted with a red arrow)

- EC2 콘솔에서 GitLab Runner 가 실행 중인 인스턴스에서 IAM Role 이 할당되었는지 확인

6) EC2 인스턴스에서 S3 접근권한 확인

- Self Managed GitLab Runner EC2 인스턴스에 접속

```
sudo apt update && sudo apt install -y curl unzip && \
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip" && \
unzip awscliv2.zip && \
sudo ./aws/install && \
aws --version
```

- AWS CLI 설치

```
1 aws s3 ls s3://[YOUR-BUCKET-NAME]
```

- S3 버킷 목록 확인
- 정상적으로 실행된다면, IAM 역할이 제대로 연결되어 있다는 뜻.

5. CI/CD 파이프라인 구성 - 빌드 환경 구성

1) Gatsby 빌드환경 Node.js 환경설정

(1) GitLab 설치된 EC2 인스턴스에 접속

(2) GitLab runner 사용자로 변경

```
1 sudo su
2 su - gitlab-runner
```

- 러너 홈(~gitlab-runner)에 도구를 설치한다.

(3) nvm으로 Node.js 설치 및 설정 추가

```
curl https://raw.githubusercontent.com/creationix/nvm/master/install.sh | bash
```

```
echo 'export NVM_DIR="$HOME/.nvm"' >> ~/.bashrc
echo '[ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh"' >> ~/.bashrc
echo '[ -s "$NVM_DIR/bash_completion" ] && . "$NVM_DIR/bash_completion"' >> ~/.bashrc
source ~/.bashrc
```

- nvm은 프로젝트별로 Node 버전 고정(LTS 권장)과 손쉬운 전환을 지원한다.
- .bashrc에 NVM 환경변수와 초기화 스크립트를 추가하면 러너가 로그인/접 시작 시 자동으로 nvm/node를 로드한다.

(4) Node.js 버전 설치

```
1 nvm install 22.12.0
2 nvm alias default 22.12.0
```

- nvm 기본 버전 고정

(5) 실행결과

```
$ sudo su
root@ip-172-31-21-31:/var/snap/amazon-ssm-agent/9881# su - gitlab-runner
gitlab-runner@ip-172-31-21-31:~$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh | bash
  % Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload Total   Spent    Left  Speed
100 14984  100 14984    0      0  241k   0 --:--:-- --:--:-- 243k
=> nvm is already installed in /home/gitlab-runner/.nvm, trying to update using git
=> => Compressing and cleaning up git repository

=> nvm source string already in /home/gitlab-runner/.bashrc
=> bash_completion source string already in /home/gitlab-runner/.bashrc
=> Close and reopen your terminal to start using nvm or run the following to use it now:

export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads nvm
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion" # This loads nvm bash_completion
gitlab-runner@ip-172-31-21-31:~$ echo 'export NVM_DIR="$HOME/.nvm"' >> ~/.bashrc
echo '[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"' >> ~/.bashrc
echo '[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion"' >> ~/.bashrc
gitlab-runner@ip-172-31-21-31:~$ source ~/.bashrc
gitlab-runner@ip-172-31-21-31:~$ nvm install 22.12.0
Downloading and installing node v22.12.0...
Downloading https://nodejs.org/dist/v22.12.0/node-v22.12.0-linux-x64.tar.xz...
#####
Computing checksum with sha256sum
Checksums matched!
Now using node v22.12.0 (npm v10.9.0)
Creating default alias: default -> 22.12.0 (> v22.12.0)
gitlab-runner@ip-172-31-21-31:~$ nvm alias default 22.12.0
default -> 22.12.0 (> v22.12.0)
```

- 설치 로그에서 Now using node v22.12.0 및 default -> 22.12.0이 보이면 정상이다.

2) 개별 JOB Artifacts 사이즈 증가

The screenshot shows the GitLab interface for 'My Static Website' under 'CI/CD Settings'. The left sidebar is collapsed. The main area displays the 'General pipelines' configuration. A red box highlights the 'Maximum artifacts size' input field, which is set to '500'. Other visible settings include 'Public pipelines' (checked), 'Auto-cancel redundant pipelines' (checked), 'Prevent outdated deployment jobs' (checked), 'Use separate caches for protected branches' (checked), and 'Minimum role required to cancel a pipeline or job' set to 'Developer'. The 'gitlab-ci.yml' file path is listed as '.gitlab-ci.yml'. The 'Git strategy' section shows 'git fetch' selected. The 'Git shallow clone' field is set to '20'. The 'Timeout' field contains '1h'. The 'Save changes' button is at the bottom.

- Runner 등록할 프로젝트 -> Settings -> CI/CD -> General pipelines -> Maximum artifacts size에서 프로젝트 단위로 산출물 한도를 상향한다.
- 한도 상향은 저장소 용량과 네트워크 비용에 영향을 주므로, 필요 파일만 산출물로 지정하여 보존 기간과 범위를 최소화한다.

6. CI/CD 파이프라인 구성 – GitLab 구성

1) Self Managed GitLab ruuner 동작하는지 확인

The screenshot shows the GitLab Pipeline editor interface. On the left, there's a sidebar with various project management sections like Issues, Merge requests, Manage, Plan, Code, Build, Pipelines, Jobs, Pipeline editor (which is highlighted with a red box), Pipeline schedules, and Artifacts. In the center, there's a large circular icon with a checkmark and a play button, followed by the text "Optimize your workflow with CI/CD Pipelines". Below that, it says "Create a new .gitlab-ci.yml file at the root of the repository to get started." At the bottom right, there's a prominent blue button labeled "Configure pipeline".

2) 파이프라인 정의

```
1  image: node:latest
2
3  before_script:
4    - source ~/.nvm/nvm.sh
5    - nvm use node
6
7  stages:
8    - build
9    - deploy
10
11 cache:
12   paths:
13     - node_modules/
14
15 install_dependencies:
16   stage: build
17   script:
18     - npm ci
19   artifacts:
20     paths:
21       - node_modules/
22
23
24 deploy-job:
25   stage: deploy
26   environment: production
27   script:
28     - npm install -g gatsby-cli
29     - gatsby build
30     - aws s3 sync public s3://[YOUR-BUCKET-NAME] --delete
```

- before_script에서 nvm 초기화 후 고정 Node 버전을 사용하고, stages: [build, deploy]로 단계 분리하여 실패 지점을 명확히한다. cache: node_modules/를 선언해 재빌드 시간을 단축한다.
- deploy 단계는 Gatsby 빌드 후 aws s3 sync로 정적 산출물을 버킷에 반영한다.

3) Pipeline Status 확인

The screenshot shows the AWS Lambda CI/CD Pipelines interface. On the left, there's a sidebar with 'Project' and 'Build' sections, where 'Pipelines' is selected. The main area displays a table of pipelines. One pipeline is highlighted with a red border: 'Update .gitlab-ci.yml file' (Pipeline #11), which passed 9 hours ago with a duration of 00:02:50. The pipeline has two stages, both of which are green.

- Build → Pipelines에서 각 파이프라인의 단계별 성공/실패 아이콘과 실행 시간을 확인한다.
- 실패 시 해당 Jobs로 들어가 로그 하이라이트에서 오류 원인을 즉시 파악한다.

4) 배포결과 확인

The screenshot shows a Gatsby website deployed to AWS CloudFront. The URL is my-static-website-website-us-west-2.amazonaws.com. The page content includes a Gatsby logo, a 'Welcome to Gatsby!' message, and a 'Welcome to Gatsby!' icon. Below the header, there's a 'Welcome to Gatsby!' section with a sub-section titled 'Welcome to Gatsby!'. The page also lists 'Example pages: Page 2 · TypeScript · Server Side Rendering · Deferred Static Generation' and provides instructions to edit 'src/pages/index.js' to update the page. There are also links for 'Tutorial', 'Examples', 'Plugin Library', and 'Build and Host'.

- S3 정적 웹사이트 엔드포인트(또는 CloudFront 도메인)로 접속해 Gatsby 기본 페이지가 노출 되는지 확인한다.
- 첫 로드가 성공해도 403/404가 뜬다면 버킷 퍼블릭 액세스/정책/인덱스 문서 설정을 재검토 해야한다.

7. CI/CD 활용 – Execution Order, DAG

1) Commit Changes

```
1 test:
2   stage: test
3   image: gliderlabs/herokuish:latest
4   script:
5     - cp -R . /tmp/app
6     - ./bin/herokuish buildpack test
7   after_script:
8     - echo "Our race track has been tested!"
9
10 super_fast_test:
11   stage: test
12   script:
13     - echo "If you're not first you're last"
14   needs: []
15
```

Commit message: Update .gitlab-ci.yml file

Branch: master

Commit changes

- Pipelines에서 job이 parallel하게 실행되는 것 확인

2) Directed Acyclic Graph

```

stages:
  - build
  - test
  - race

build_car_a:
  stage: build
  script:
    - echo "build_car_a"

build_car_b:
  stage: build
  script:
    - echo "build_car_b"

build_car_c:
  stage: build
  script:
    - echo "build_car_c"

build_car_d:
  stage: build
  script:
    - echo "build_car_d"

build_car_e:
  stage: build
  script:
    - echo "build_car_e"

build_car_f:
  stage: build
  script:
    - echo "build_car_f"

test_car_a:
  stage: test
  needs: [build_car_a]
  script:
    - echo "test_car_a"

test_car_b:
  stage: test
  needs: [build_car_b]
  script:
    - echo "test_car_b"

test_car_c:
  stage: test
  needs: [build_car_c]
  script:
    - echo "test_car_c"

test_car_d:
  stage: test
  needs: [build_car_d]
  script:
    - echo "test_car_d"

test_car_e:
  stage: test
  needs: [build_car_e]
  script:
    - echo "test_car_e"

test_car_f:
  stage: test
  needs: [build_car_f]
  script:
    - echo "test_car_f"

race_car_a:
  stage: race
  needs: [test_car_a]
  script:
    - echo "race_car_a"

race_car_b:
  stage: race
  needs: [test_car_b]
  script:
    - echo "race_car_b"

race_car_c:
  stage: race
  needs: [test_car_c]
  script:
    - echo "race_car_c"

race_car_d:
  stage: race
  needs: [test_car_d]
  script:
    - echo "race_car_d"

race_car_e:
  stage: race
  needs: [test_car_e]
  script:
    - echo "race_car_e"

race_car_f:
  stage: race
  needs: [test_car_f]
  script:
    - echo "race_car_f"

```

- GitLab의 needs: 키를 사용하면 stage 순서에 얹매이지 않고 잡 간 직접 의존성을 선언해 DAG를 구성할 수 있다.
- 예시에서 test_car_* 는 각 build_car_* 를, race_car_* 는 각 test_car_* 를 실행 조건으로 지정 한다.

3) pipeline 통해 conditional하게 job 실행되는 것 확인

The screenshot shows the GitLab pipeline interface for a project named 'Gatsby Starter Default'. The pipeline has three stages: 'build', 'test', and 'race'. Each stage contains six jobs corresponding to the stages in the .gitlab-ci.yml file. The jobs are color-coded: green for success and orange for pending or failed. The 'Job dependencies' tab is visible at the bottom right of the pipeline view.

- 파이프라인 상세 화면의 각 잡 카드 색상(녹색=성공, 회색=스킵)과 “Job dependencies” 탭으로 실행/스킵/의존성을 함께 검증한다.