유희준

DevOps · Infrastructure Engineer

운영 중 생기는 문제를 끝까지 따라가 원인을 찾고 다시 생기지 않게 구조로 막는 인프라 엔지니어입니다. EKS 기반 SaaS 전환 실무와 AWS 인프라를 혼자 설계하고 운영한 프로젝트에서 Kubernetes 운영, Terraform과 GitOps 자동화, 서비스별 권한 분리, 모니터링 구축을 경험했습니다.

Email yhjyhj3855@gmail.com
GitHub UHeeJoon
Web uheejoon.github.io/devops

목차

(주)옥타코 멀티테넌트 SSO 인증 서비스

실무 · 2025.04 ~ 2026.02

01	서비스별 AWS 권한 분리로 탈취 피해를 한 서비스 안으로 제한
02	Probe와 CPU limit 조정으로 수백 번 반복되던 Pod 재시작을 0번으로 해소
03	Admission Webhook 개발로 배포 경로와 관계없이 같은 배포 기준 적용
04	GitOps 전환으로 배포 덮어쓰기를 없애고 Argo Rollouts로 무중단 배포 구성
05	장애 주입 시험으로 미리 찾은 Redis SPOF를 Circuit Breaker와 DB 우회로 해결

사전예약 서비스 NOVA

팀 프로젝트 · 2026.09 ~ 2026.10

01	YAML 설정 하나로 AWS 인프라가 만들어지는 Terraform 자동화
02	Atlantis를 활용한 AWS 인프라 변경 관리
03	ALB 앞에 CloudFront를 붙여 엣지에서 거르는 WAF 구성과 경로별 규칙 조정
04	원인 분석을 수십 분에서 수 분으로 줄인 OpenTelemetry와 Grafana 통합 모니터링
05	운영 경로 그대로 판매 오픈을 재현하는 ECS 기반 k6 부하 시험 플랫폼
06	목표 구조는 코드에 남기고 월 운영비를 73% 줄인 Terraform 변수 기반 비용 절감

홈랩 Kubernetes 플랫폼

개인 프로젝트 · 2026.05 ~ 현재

01	Hubble 통신 기록을 활용한 CNI 전환 중 DNS 장애 분석
02	디스크가 남았는데도 볼륨이 만들어지지 않던 Longhorn 스토리지 장애 해결

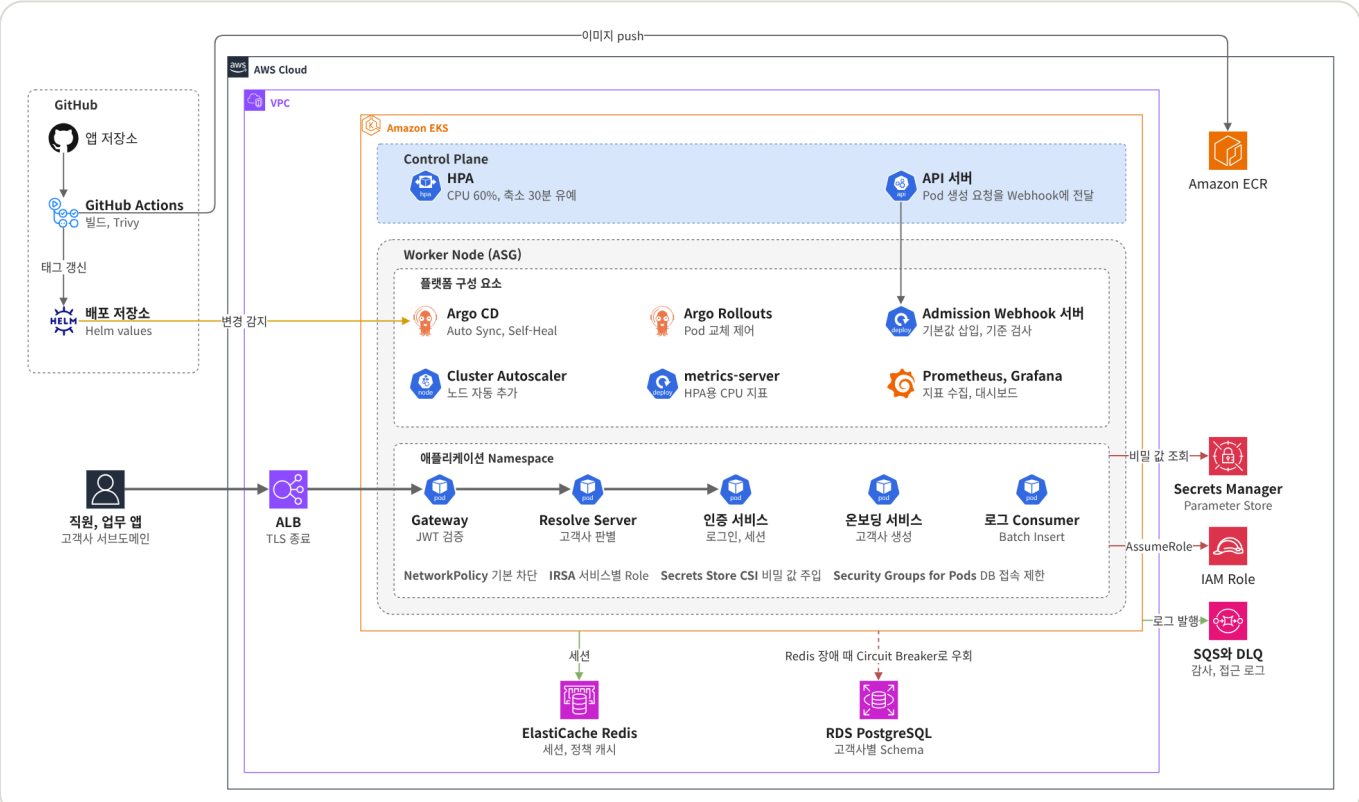
(주)옥타코 멀티테넌트 SSO 인증 서비스

온프레미스 SSO를 AWS EKS 기반 멀티테넌트 SaaS로 전환하는 프로젝트

기간	2025.04 ~ 2026.02, 11개월, 정규직
소속과 역할	백엔드 3명으로 구성된 초기 TF, DevOps와 Backend 엔지니어
담당	AWS EKS 기반 인프라 구축 Java/Spring 기반 사용자와 세션 인증 도메인 개발
기술	`AWS EKS` `Helm` `Argo CD` `Argo Rollouts` `GitHub Actions` `Amazon ECR` `IRSA` `NetworkPolicy` `Admission Webhook` `Secrets Manager` `Resilience4j` `Chaos Monkey` `k6`

서비스 개요

여러 고객사가 함께 쓰는 SSO 인증 서비스입니다. 고객사 직원은 회사 계정으로 한 번 로그인하면 연결된 업무 앱을 다시 로그인하지 않고 씁니다. 온프레미스로 운영하던 SSO를 AWS EKS 위의 멀티테넌트 구조로 새로 만드는 프로젝트였고 출시 전 단계의 초기 TF에 합류했습니다.



▲ 업무를 바탕으로 재구성한 옥타코의 요청 경로와 배포, 플랫폼 구성 요소, 데이터 구성

서비스별 AWS 권한 분리로 탈취 피해를 한 서비스 안으로 제한

서비스마다 IAM Role 분리 | Pod 통신과 DB 접속 제한 | 비밀 값은 중앙 저장소에서 주입

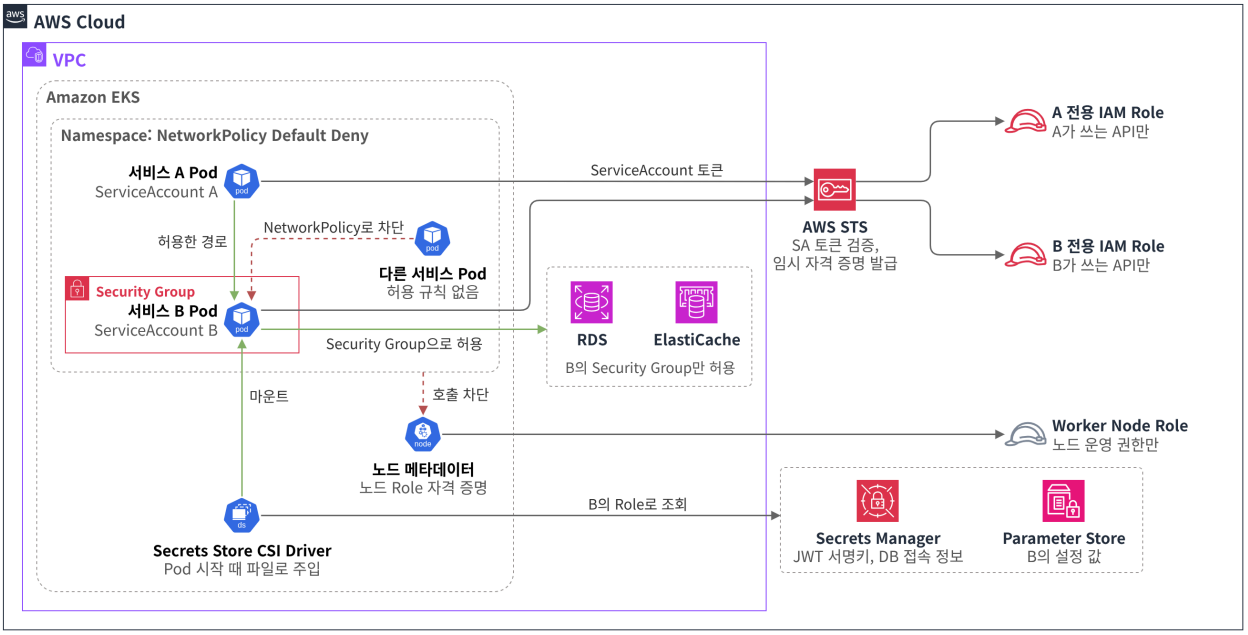
문제 발견

클러스터를 점검해 보니 서비스 하나가 탈취됐을 때 피해가 그 서비스에서 멈추지 않는 경로가 세 군데 있었습니다.

- 모든 Pod가 노드의 IAM Role을 공유해 한 서비스가 탈취되면 **다른 서비스의 AWS 권한과 비밀 값까지 쓸 수 있었습니다**.
- Pod 사이 통신이 Kubernetes 기본값대로 모두 열려 있어 한 Pod에서 다른 서비스를 바로 호출할 수 있었습니다.
- RDS와 ElastiCache 접속 허용 규칙이 CIDR 기준이라 같은 대역의 어떤 리소스든 DB와 캐시에 접근할 수 있었습니다.

비밀 값도 각자 로컬에서 ConfigMap과 Secret으로 직접 만들어 **한곳에서 관리되지 않았습니다**.

해결 과정



▲ 서비스마다 IAM Role을 받는 IRSA 흐름과 Pod 통신, 노드 메타데이터, DB와 캐시, 비밀 값 접근 제한

① 서비스마다 IAM Role 분리

- **IRSA로 서비스마다 Role을 나누고** 자기 비밀 값 조회처럼 그 서비스가 실제로 쓰는 AWS API만 허용했습니다.
- Worker Node Role에는 노드 운영에 필요한 권한만 남겼습니다.

② Pod 통신과 DB 접속 제한

- VPC CNI NetworkPolicy로 Pod의 **Ingress와 Egress를 Default Deny**로 두고 필요한 경로만 열었습니다.
- Egress가 기본 차단이 되면서 Pod가 노드 메타데이터에 접근해 Worker Node Role 권한을 받아 가는 **우회 경로도 함께 막혔습니다**.

- Security Groups for Pods로 서비스마다 Security Group을 붙이고 RDS와 ElastiCache는 **그 그룹을 가진 Pod에만 접속을 허용**했습니다.

③ 비밀 값은 중앙 저장소에서 주입

- JWT 서명키, DB 접속 정보, 설정 값은 Secrets Manager와 Parameter Store에 두고 **Secrets Store CSI Driver로 Pod에 주입**했습니다.

검증과 결과

Pod에 접속해 직접 요청을 보내 차단되는지 확인했습니다.

시도한 요청	결과
다른 서비스의 비밀 값 조회	AccessDenied로 거절
노드 메타데이터 호출	차단
허용하지 않은 Pod로의 요청	차단
Security Group이 없는 Pod의 DB 접속	차단

비밀 값은 Secrets Manager와 Parameter Store에서만 관리해 **로컬에서 Secret을 만들지 않고 배포**합니다.

Probe와 CPU limit 조정으로 수백 번 반복되던 Pod 재시작을 0번으로 해소

Pod 재시작 수백 번에서 0번으로 | 20번 이상 재생성 테스트에서 Probe 실패 0건 | 공통 기본값과 배포 기준으로 재발 방지

문제 발견

주말 동안 **Pod가 수백 번 재시작**한 것을 발견했습니다.

원인 분석

- 로그를 확인하니 애플리케이션 **기동이 끝나기 전에 liveness Probe가 실패**해 기동 중인 정상 Pod가 재시작되고 있었습니다.
- Pod spec을 보니 기동이 끝날 때까지 liveness 검사를 미루는 **startupProbe가 없었습니다**.

1차 조치

- liveness Probe는 그대로 두고 로컬에서 잰 기동 시간을 기준으로 **startupProbe를 설정**했습니다.

설정	정한 값	근거
startupProbe `initialDelaySeconds`	가장 짧은 기동 시간	로컬에서 20번가량 잰 기동 시간 중 최솟값

설정	정한 값	근거
`periodSeconds` 와 `failureThreshold`	5초 주기로 평균 기동 시간의 2배까지 대기	같은 실측의 평균 기동 시간

재발과 원인 분석

적용 뒤에도 CPU 부하가 높을 때 startupProbe가 실패해 재시작되는 현상을 발견했습니다. 앱 로그에는 로컬에서 잰 것보다 훨씬 느린 기동 시간이 찍혀 있었습니다.

Grafana에서 그 시점의 CPU 지표를 보니 기동 중 CPU 사용량이 **limit 1 core에 닿으면서 throttling이 급증**하고 있었습니다.

2차 조치

CPU limit을 1 core에서 2 core로 올려 기동 중 throttling이 생기지 않게 했습니다.

설정	정한 값	근거
CPU limit	2 core	기동 중 throttling 해소, limit은 없애지 않아 과도한 CPU 사용 방지
CPU requests	평상시 사용량 그대로	노드에 예약하는 자원은 늘리지 않음

검증과 결과

- 조치 뒤 근무하는 동안 평시와 CPU 작업이 몰릴 때 모두 **Probe 실패와 Pod 재시작이 발생하지 않았습니다**.
- Pod를 20번 이상 다시 만드는 테스트에서도 **Probe 실패는 0건**이었습니다.

후속 조치

- 검증한 Probe와 리소스 설정을 **Helm 공통 values 기본값**으로 넣어 새 서비스도 같은 기준으로 배포되게 했습니다.
- requests와 limits가 빠진 배포는 **Admission Webhook에서 거절**하도록 해 같은 문제가 다른 서비스에서 반복되지 않게 막았습니다. 이 Webhook은 03 사례에서 다룹니다.

Admission Webhook 개발로 배포 경로와 관계없이 같은 배포 기준 적용

빠진 환경 변수와 보안 설정 자동 삽입 · requests와 limits 누락 배포 거절 · default Namespace 배포 거절

설계 배경

- 시간대, JVM 메모리, 보안 설정 같은 기본값이 Helm 공통 Template에만 들어 있어 차트를 거치지 않고 kubectl로 직접 적용한 배포에는 **기준이 빠질 수 있는 구조**였습니다.
- 02 사례의 CPU limit 문제처럼 리소스 기준이 빠지면 운영 장애로 이어질 수 있었습니다.

- **배포 경로와 상관없이 클러스터 입구에서 리소스와 보안 기준을 지킬 방법**이 필요했습니다.

설계 판단

검토한 방식	장점	단점
Kyverno, OPA Gatekeeper	정책 표현이 풍부하고 예제가 많음	정책 문법을 새로 익히고 컨트롤러와 CRD를 추가로 운영
Admission Webhook 직접 개발	삽입과 검사를 코드로 자유롭게 작성	Webhook 서버를 직접 개발하고 운영

> **필요한 규칙이 몇 개뿐이라 참고할 예제 코드를 바탕으로 Admission Webhook을 직접 개발하는 쪽을 택했습니다.**

구현

기준은 **Pod Security Standards** ↗를 참고해 보안 항목을 고르고 시간대와 JVM 메모리처럼 회사에서 필요한 항목을 더해 정했습니다.

Webhook 서버는 Pod 여러 개로 띄워 Pod 하나에 장애가 나도 요청 처리가 이어지게 했습니다.

Mutating Webhook으로 빠진 기본값만 삽입

- Pod가 만들어질 때 `TZ=Asia/Seoul`, JVM 메모리 옵션, 환경 이름과 서비스 이름 환경 변수가 **Pod spec에 없으면 넣도록** 했습니다.
- `runAsNonRoot`, `allowPrivilegeEscalation: false` 같은 보안 설정도 지정하지 않은 경우에만 넣고 서비스가 직접 지정한 값은 덮어쓰지 않게 했습니다.
- Webhook이 멈춰도 기본값이 빠지지 않도록 Helm Chart에도 같은 기본값을 그대로 뒀습니다.

Validating Webhook으로 기준이 빠진 배포 거절

- requests와 limits가 없는 Pod는 CPU limit 때문에 재시작한 적이 있어 **자동으로 넣지 않고 거절해 직접 지정하게** 했습니다.
- default Namespace 배포는 Namespace를 빠뜨린 배포를 막고 **서비스와 환경을 Namespace로 구분**하려고 거절하게 했습니다.
- 거절할 때는 배포한 사람이 바로 고칠 수 있도록 아래처럼 빠진 항목을 메시지로 알려 주게 했습니다.

```
# requests와 limits 없이 default Namespace에 배포한 경우
$ kubectl apply -f pod.yaml
Error from server: admission webhook "..." denied the request:
  default Namespace에는 배포할 수 없습니다
  resources.requests와 resources.limits를 지정해야 합니다
```

▲ 거절될 때 kubectl에 나오는 모양. 메시지 문구는 예시입니다.

검증과 결과

항목	이전	이후
기본 환경 변수, 보안 설정	Helm 기본값이라 차트 밖 배포에는 빠질 수 있음	빠져 있으면 Pod 생성 때 자동 삽입
리소스, Namespace	차트 밖 배포는 기준이 없어도 그대로 생성	requests와 limits 누락, default Namespace 배포 거절

- 환경 변수와 보안 설정을 뺀 Pod를 적용해 생성된 Pod spec에 **빠진 값이 들어간 것을 확인**했습니다.
- requests와 limits가 없는 Pod를 kubectl로 직접 적용하고 default Namespace에 배포를 시도해 **두 경우 모두 거절되는 것을 확인**했습니다.

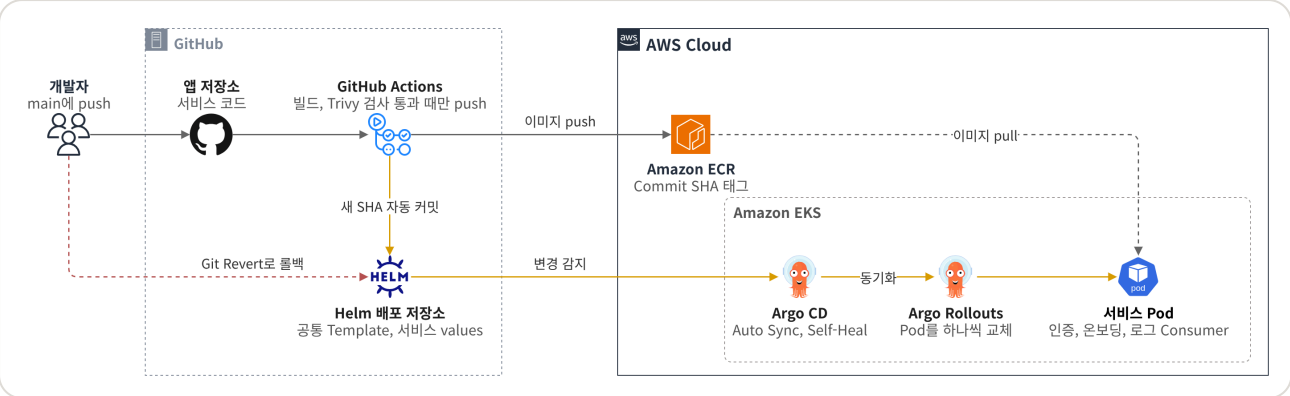
GitOps 전환으로 배포 덮어쓰기를 없애고 Argo Rollouts로 무중단 배포 구성

배포 경로를 Git 하나로 통일 | Argo Rollouts로 무중단 배포

문제 발견

- 개발자마다 자기 PC에서 직접 배포하고 있어 올라가는 코드가 PC마다 달랐습니다.
- 여러 명이 배포하면 **서로의 배포를 덮어써** 서버에 어떤 버전이 올라가 있는지 알 수 없었습니다.

해결 과정



▲ push부터 GitHub Actions, ECR, Helm 배포 저장소, Argo CD 동기화와 Argo Rollouts의 Pod 교체, Git Revert 롤백까지의 배포 흐름

① 배포 경로를 Git 하나로 통일

PC에서 직접 배포하던 것을 **main에 push하면 아래 순서로 배포되도록** 바꿨습니다.

단계	구성 요소	하는 일
1. 빌드와 검사	GitHub Actions, Trivy	이미지를 빌드하고 HIGH 이상 취약점이 나오면 배포 중단
2. 이미지 등록	ECR	Commit SHA 태그로 올려 이미지가 어느 커밋에서 나왔는지 추적
3. 설정 갱신	Helm 배포 저장소	이미지 태그를 Commit SHA로 자동 커밋해 배포 이력을 Git 커밋으로 관리
4. 동기화	Argo CD	Auto Sync로 반영하고 클러스터를 직접 수정하면 Self-Heal이 Git 상태로 복구
5. 롤백	Git Revert	배포 저장소 커밋을 되돌리면 Argo CD가 이전 이미지 태그로 복구

- Helm 배포 저장소의 공통 Template 하나로 Rollout, Service, HPA를 만들고 서비스별 설정 값은 values 파일로 override했습니다.
- 새 서비스는 **Helm values 파일과 Argo CD Application 등록만으로 배포**되도록 구성했습니다.

② Argo Rollouts로 배포 중 끊김 없이 Pod 교체

배포 중에도 서비스가 끊기지 않도록 세 가지 배포 방식을 비교했습니다.

배포 방식	동작	판단
Blue/Green	새 버전을 통째로 띄운 뒤 전환	교체 중 Pod가 두 배가 되어 노드가 늘어날 수 있음
Canary	일부 요청부터 단계별로 확대	단계별 값을 테스트로 실측해 정해야 해 준비에 시간이 오래 걸림
Rolling Update	Pod를 하나씩 새 버전으로 교체	따로 정할 값이 적어 바로 적용 가능

> **출시 전에는 빨리 적용하는 것이 중요해 가장 단순한 Rolling Update를 우선 선택했습니다.**

Canary 전략에 steps를 두지 않아 지금은 Rolling Update로 동작하고 **출시 뒤 steps만 더하면 Canary로 바뀌도록** 설정했습니다.

```
# Rollout
spec:
  ...
  strategy:
    canary: {}    # 지금: steps가 없어 Rolling Update로 동작
    # 출시 뒤: steps만 더하면 Canary
    # canary:
    #   steps:
    #     - setWeight: 20
    #     - pause: { duration: 5m }
  ...
```

▲ Rollout의 배포 전략 부분. 주석의 20%와 5분은 예시 값입니다.

검증과 결과

- 배포 경로가 하나로 모여 **서로 덮어쓰는 일이 없어졌고** 올라간 버전은 Git 이력과 이미지 태그로 확인합니다.
- 배포 중 k6로 부하를 걸어 **실패한 요청 없이 Pod가 교체되는 것을 확인**했습니다.

후속 조치

- 배포와 롤백 절차를 문서로 남겨 다른 개발자도 혼자 배포하고 되돌릴 수 있게 했습니다.

장애 주입 시험으로 미리 찾은 Redis SPOF를 Circuit Breaker와 DB 우회로 해결

Redis 장애 중에도 로그인과 인증 처리 | 복제가 못 막는 지연과 연결 끊김 대응 | 감지부터 복귀까지 자동 전환

문제 발견

- 출시 전 k6 부하 중 Chaos Monkey for Spring Boot로 Redis 호출에 timeout을 주입하자 **모든 고객사의 세션 인증이 실패**했습니다.
- 세션을 Redis에만 두고 있어 Redis 하나가 모든 고객사 인증의 **SPOF가 되는 구조**였습니다.

설계 판단

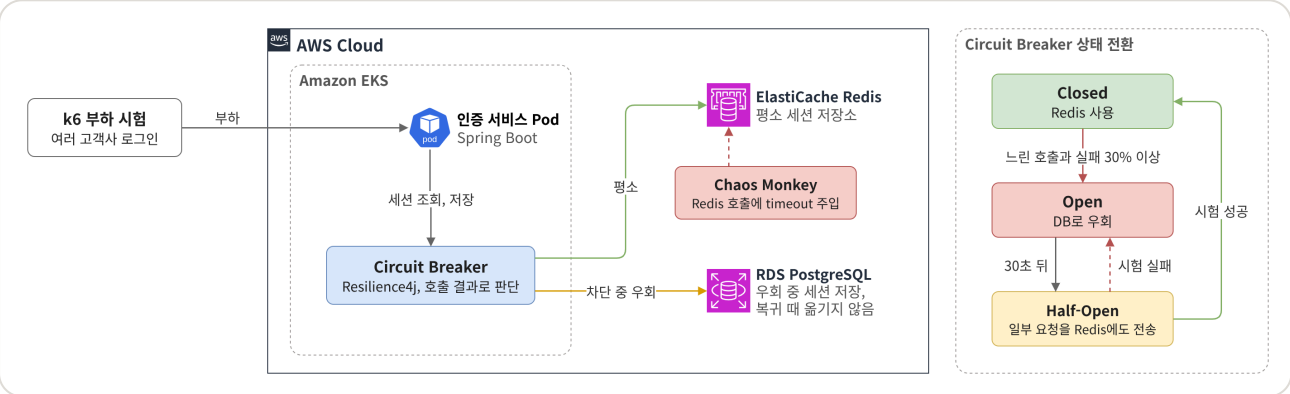
장애 유형별로 나눠 보니 ElastiCache 복제는 노드 장애만 막고 서버와 Redis 사이 **지연이나 연결 끊김에는 대응하지 못했습니다**.

장애 유형	ElastiCache 복제	Circuit Breaker와 DB 우회
Redis 노드 장애	복제본으로 전환	DB로 우회
서버와 Redis 사이 지연	대응 불가	느린 호출로 집계해 DB로 우회
서버와 Redis 사이 연결 끊김	대응 불가	실패한 호출로 집계해 DB로 우회

> **장애의 원인과 상관없이 호출 결과로 판단하는 Circuit Breaker를 애플리케이션에 두기로 했습니다.**

우회할 저장소는 서비스가 이미 쓰고 있고 여러 Pod가 함께 조회할 수 있는 PostgreSQL로 정했습니다.

○ 구현



▲ 장애 주입 시험과 Redis 장애 때 Circuit Breaker가 DB로 우회하는 흐름과 상태 전환

차단 기준과 복귀 방식

Resilience4j Circuit Breaker로 Redis 호출을 감싸고 상태마다 세션을 처리하는 저장소와 전환 기준을 아래처럼 설정했습니다.

상태	세션 처리	다음 상태로 넘어가는 기준
Closed	Redis	최근 100건 중 300ms를 넘긴 느린 호출과 실패가 30% 이상이면 Open
Open	PostgreSQL	30초 뒤 Half-Open
Half-Open	PostgreSQL로 응답하고 요청 10건은 Redis에도 전송	실패율이 30% 미만이면 Closed, 이상이면 다시 Open

- Redis 호출 timeout은 500ms로 두고 그보다 짧은 300ms를 느린 호출 기준으로 잡아 timeout 전의 지연도 집계하게 했습니다.
- Redis에 보낸 요청이 실패해도 **응답은 PostgreSQL이 맡아** 복귀를 확인하는 동안 사용자 요청에는 영향이 없도록 했습니다.

> **세션은 옮기지 않고 재로그인으로 처리**
>
> Redis와 DB 사이에 세션을 옮기다 다시 장애가 나면 처리가 복잡해지고 두 저장소의 세션이 섞일 수 있어 **세션은 옮기지 않고** 기존 사용자가 장애 시작과 복구 때 한 번씩 다시 로그인하게 했습니다.

검증과 결과

- k6 부하 중 Redis timeout 주입과 해제를 반복해 차단, DB 우회, 복귀를 상태 전환 기록으로 확인했고 k6 요청은 차단 전 잠깐만 실패한 뒤 정상 처리됐습니다.
- Redis 장애 중에도 **로그인과 인증이 계속 처리**되고 감지부터 복귀까지 사람이 개입하지 않습니다.

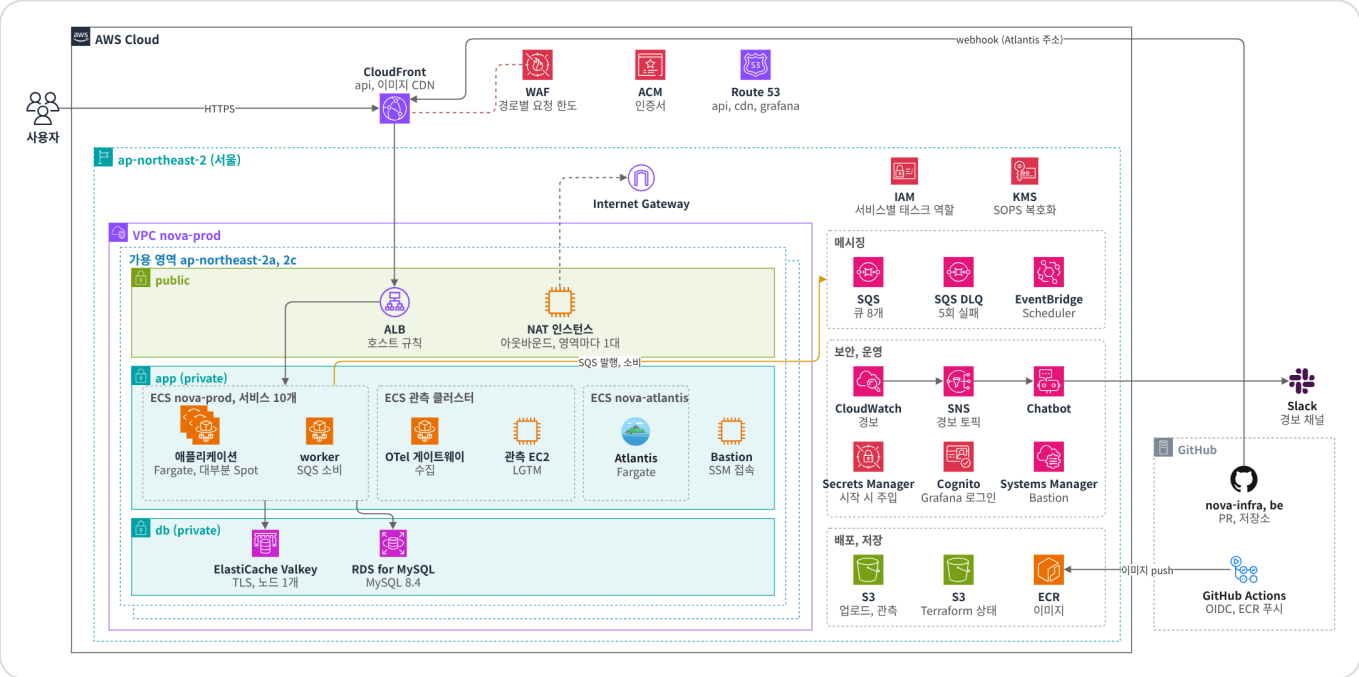
사전예약 서비스 NOVA

판매가 열리면 10초 사이에 예약 5,000건이 몰리는 사전예약 서비스를 AWS에 구축한 프로젝트

기간	2026.09 ~ 2026.10, LG U+ 유레카 부트캠프
소속과 역할	백엔드 5명과 프론트엔드 2명으로 구성된 7인 팀, 인프라 단독 담당과 백엔드 개발
담당	AWS 인프라 설계와 구축, 운영 사전예약 접수 서버와 대기열, 외부 연동 worker 개발
저장소	Nova-Grandis/nova-infra, Nova-Grandis/atlantis-deploy
기술	`ECS Fargate` `SQS` `CloudFront` `WAF` `Terraform` `Atlantis` `IAM` `OIDC` `Secrets Manager` `SOPS` `KMS` `OpenTelemetry` `Grafana` `Mimir` `Loki` `Tempo` `k6`

서비스 개요

회차마다 정해진 시각에 판매가 열리는 사전예약 서비스입니다. 판매가 열리면 10초 사이에 예약 5,000건이 몰리기 때문에 그 순간의 요청을 받아 내는 구조를 AWS에 구축하는 프로젝트였습니다. 7인 팀에서 인프라를 혼자 맡아 설계부터 구축, 운영까지 담당했습니다.



▲ CloudFront와 ALB를 거쳐 ECS Fargate로 들어오는 요청 경로와 데이터, 배포, 모니터링 구성

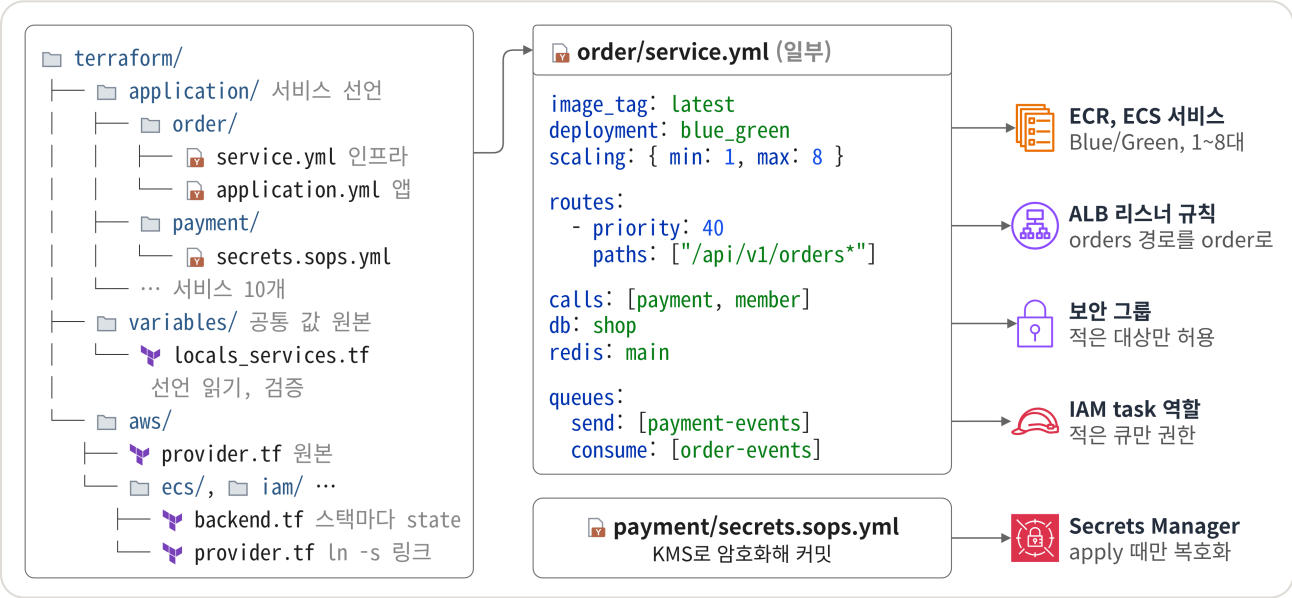
YAML 설정 하나로 AWS 인프라가 만들어지는 Terraform 자동화

YAML 설정 하나로 인프라 선언 | 잘못된 값은 plan에서 차단 | 로컬 에뮬레이터로 먼저 검증

설계 배경

- 서비스의 경로, 보안 그룹, 확장 설정을 바꿀 때마다 **인프라 담당 한 명에게 요청해야 하는 구조**였습니다.
- 백엔드 팀원은 Terraform과 AWS를 몰라 직접 고치기 어려웠고 고치더라도 잘못된 값이 그대로 적용될 위험이 있었습니다.
- 팀원이 이미 아는 방식으로 입력하고 잘못된 값은 적용 전에 막는 구조가 필요했습니다.

해결 과정



▲ 저장소 폴더 구조와 service.yml 한 파일에서 AWS 자원이 만들어지는 흐름

① service.yml 선언 하나로 서비스 인프라 생성

- 팀원이 `application.yml`로 매일 쓰던 YAML 형식으로 서비스마다 `service.yml` **하나만 작성**하게 했습니다.
- 항목 이름은 보안 그룹, ALB 규칙 같은 AWS 용어 대신 `routes`, `calls`, `queues`, `scaling`처럼 서비스가 하는 일로 정의했습니다.
- 팀원이 묻지 않고도 작성할 수 있게 README에 필수 항목, 기본값, 항목마다 만들어지는 자원을 예시와 함께 정리했습니다.

선언한 내용은 모든 service.yml을 한곳에서 읽고 각 스택이 그 파일을 심볼릭 링크로 가져와 **서비스마다 자원을 만들도록** 구성했습니다.

```
# terraform/variables/locals_services.tf
# 모든 service.yml을 읽어 서비스 목록 생성
locals {
  _service_raw = {
    for f in fileset(
      local.application_dir, "*/service.yml"
    ) :
    dirname(f) => yamldecode(
      file("${local.application_dir}/${f}")
    )
    ...
  }
  ...
}
```

```
# terraform/aws/ecs/nova_apnortheast2/services.tf
# 왼쪽 파일을 심볼릭 링크로 가져와 서비스마다 모듈 실행
module "service" {
  # 직접 만든 모듈. ECS 서비스, 태스크 정의,
  # ALB 규칙, 오토스케일링 생성
  source   = "../_modules/service"
  # service.yml에서 읽은 서비스 목록
  for_each = local.deployed_services

  name   = "${local.name_prefix}-${each.value.name}"
  cpu    = each.value.cpu
  # service.yml의 routes가 ALB 리스너 규칙이 됨
  routes = each.value.routes
  ...
}
```

▲ 왼쪽은 service.yml을 읽는 부분, 오른쪽은 읽은 값을 서비스마다 모듈에 넘겨 자원을 만드는 부분

② 잘못 적은 값을 plan에서 차단

- apply 전 plan 단계에서 service.yml 값을 검사해 틀리면 아래처럼 **어느 파일의 무엇이 틀렸는지 오류 메시지로 알려 주게** 했습니다.
- 모르는 필드나 오타, 없는 호출 대상, 다른 규칙과 겹치는 ALB 우선순위, RDS 연결 한도를 넘는 최대 대수를 검사하게 했습니다.

```
# order의 service.yml에 scaling을 scalling으로 잘못 적은 경우
$ terraform plan
Error: ...
"application/order/service.yml — 모르는 필드 'scalling'"
```

▲ 오타를 넣은 service.yml로 plan을 실행한 출력에서 핵심 줄만 남겼습니다.

로컬 검증

Terraform 코드를 실제 AWS에서 바로 시험하면 요금이 들고 실수가 운영 환경에 그대로 반영되기 때문에 **AWS 에뮬레이터 Floci를 활용해 로컬에서 먼저 apply** 해 검증했습니다.

- 가짜 자격 증명으로만 연결해 실제 AWS 계정에는 적용되지 않게 했습니다.
- 스택마다 state를 로컬 파일로 두고 앞 스택의 출력을 이어 받아 VPC, 보안 그룹, ECR, IAM, ECS의 실제 순서대로 적용했습니다.
- `terraform validate`는 통과하지만 **실제로 apply하면 AWS가 거절하는 설정을 미리 찾아 고쳤고** `calls`와 `routes`대로 보안 그룹과 ALB 규칙이 생기는지 확인했습니다.

결과

팀원이 인프라 담당자를 거치지 않고 ALB 경로, 서비스 간 호출 허용, 큐 권한, 최소 대수를 **service.yml과 PR로 직접 변경**했습니다.

Atlantis를 활용한 AWS 인프라 변경 관리

AWS 쓰기 권한은 Atlantis에만 | 팀원은 읽기 전용으로 plan까지 | apply 성공 뒤에만 병합

설계 배경

- 7명이 AWS 계정 하나를 함께 쓰는데 콘솔이나 로컬에서 바꾸면 코드에 남지 않아 **누가 무엇을 바꿨는지 알 수 없고** 지우지 않은 자원에 요금이 계속 나가는 구조였습니다.
- 인프라를 바꾸는 경로를 하나만 두고 모든 변경이 PR에 남는 절차가 필요했습니다.

설계 판단

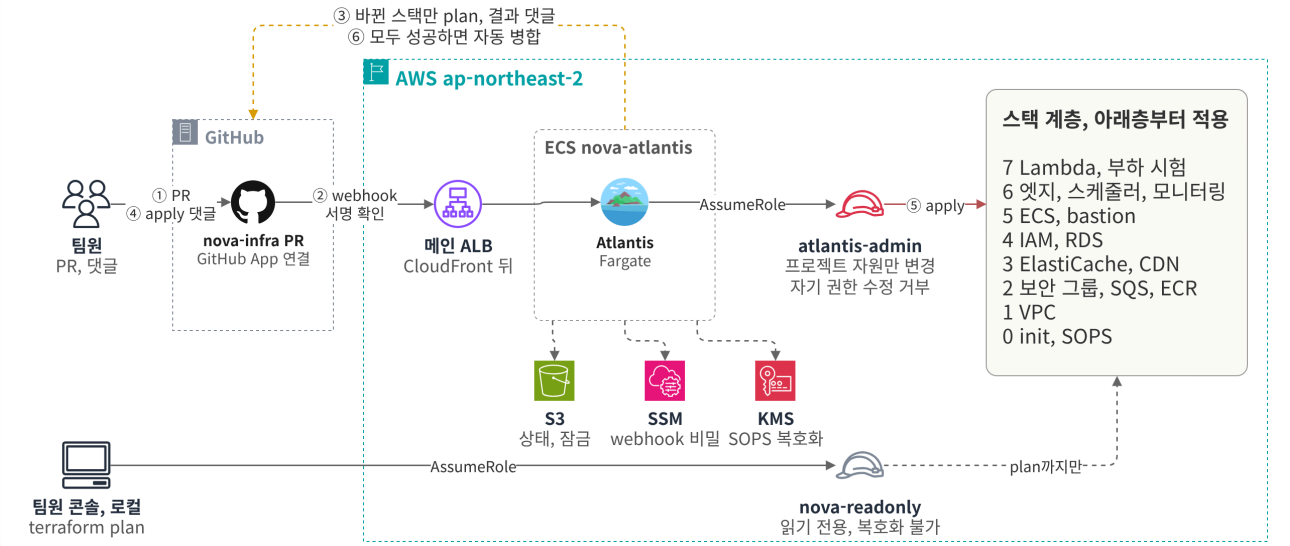
PR에서 plan과 apply를 실행하는 방법으로 GitHub Actions와 Atlantis를 비교했습니다.

필요한 기능	GitHub Actions	Atlantis
PR에 plan 결과 표시	워크플로 직접 작성	PR을 열면 자동 plan 후 댓글
PR에서 apply 실행	직접 구현	apply 댓글로 실행

필요한 기능	GitHub Actions	Atlantis
스택 잠금	직접 구현	plan한 PR이 병합될 때까지 잠금

> **GitHub Actions로는 apply와 스택 잠금을 따로 만들어야 해서 둘을 기본 제공하는 Atlantis를 택했습니다.**

○ 운영 방식



▲ PR 하나로 plan, apply, 자동 병합이 이어지는 경로와 팀원의 읽기 전용 경로

① AWS 쓰기 권한을 Atlantis에만 부여

팀원에게는 콘솔과 로컬 모두 plan까지만 되는 읽기 전용 역할을 주고 **쓰기 역할은 Atlantis에만 부여**했습니다.

② 바뀌는 스택만 plan하고 잠금

- PR이 올라오면 **Atlantis가 GitHub webhook을 받아 바뀐 스택만 plan**하고 결과를 PR 댓글로 남기게 했습니다.
- 공용 변수와 모듈이 바뀌면 그 파일을 읽는 스택까지 plan해 빠지는 스택이 없게 했습니다.
- plan한 스택은 **PR이 병합될 때까지 잠가** 다른 PR의 apply 때문에 앞선 plan 결과가 달라지지 않게 했습니다.

③ plan을 통과한 PR만 순서대로 apply하고 병합

- 팀원이 PR에 apply 댓글을 달면 Atlantis가 적용하고 plan이 실패했거나 main과 충돌한 PR은 거절하게 했습니다.
- ECS 서비스는 VPC와 보안 그룹이 먼저 있어야 하듯 스택 사이에 순서가 있어 Atlantis 설정 파일 `atlantis.yaml`에 **스택마다 적용 순서를 적고** 앞 스택이 실패하면 뒤 스택은 멈추게 했습니다.
- 병합을 먼저 하면 apply가 실패해도 main에 적용되지 않은 변경이 남기 때문에 **apply가 모두 성공한 뒤에만 자동 병합**되게 하고 브랜치 보호 규칙으로 직접 병합도 막았습니다.

결과

- 모든 인프라 변경이 Atlantis에서만 PR을 거쳐 적용되어 **누가 무엇을 바꿨는지 plan과 apply 결과와 함께 PR에 남았습니다**.
- 콘솔에서 따로 만든 자원이 없어 코드에 없는 자원이 요금으로 남지 않았습니다.
- AWS 쓰기 권한 없이도 팀원이 PR 안에서 plan 확인부터 apply와 병합까지 직접 처리했습니다.

ALB 앞에 CloudFront를 붙여 엣지에서 거르는 WAF 구성과 경로별 규칙 조정

WAF 검사를 ALB 앞 엣지로 이동 | ALB는 CloudFront를 거친 요청만 허용 | 정상 요청을 막던 규칙은 경로별로 조정

설계 배경

- 처음에는 ALB가 인터넷에서 API를 직접 받아 **연결과 공격 요청을 모두 서울 리전의 ALB가 받아 낸 뒤에야** ALB에 붙인 WAF로 걸러지는 구조였습니다.
- 판매가 열리면 10초 사이에 예약 5,000건이 몰리기 때문에 이 순간의 연결과 과한 요청을 ALB보다 앞에서 받아 낼 곳이 필요했습니다.

설계 판단

처음 구성과 ALB 앞에 CloudFront를 붙인 구성을 비교했습니다.

항목	처음 구성	바꾼 구성
API 경로	사용자 → ALB(+WAF) → ECS	사용자 → CloudFront + WAF → ALB → ECS
ALB가 받는 연결	사용자마다 새 TLS 연결	TLS는 엣지에서 끝내고 ALB 구간은 연결 재사용
막힌 요청이 닿는 곳	VPC 안 ALB까지 들어온 뒤 차단	VPC 밖 엣지에서 차단, ALB에 닿지 않음
새로 생기는 위험	없음	ALB를 직접 부르면 WAF를 우회, ①에서 차단

> **API는 캐시하지 않지만 엣지에서 먼저 거르려고 WAF를 CloudFront로 옮겼고, 이때 생기는 ALB 직접 호출 경로는 CloudFront를 거친 요청만 받게 해 막았습니다.**

구현

① CloudFront로 옮기며 생긴 우회 경로 차단

- ALB 보안 그룹은 **CloudFront 관리형 prefix list만 허용**해 다른 IP에서는 연결부터 맺어지지 않게 했습니다.
- prefix list는 모든 CloudFront 배포의 대역이라 다른 사람의 배포를 거친 요청은 NOVA 배포가 붙인 무작위 40자 **비밀 헤더가 없으면 ALB가 404**로 돌려보내게 했습니다.

- 공개 PR의 plan 출력에 비밀 헤더 값이 찍힌 것을 발견해 당일 값을 바꾸고 `sensitive`로 지정해 이후 plan에 나오지 않게 했습니다.

② 같은 IP 뒤 대기자를 막지 않는 요청 수 제한

- `/api/`는 IP당 5분 3,000건으로 제한했는데, 통신사 CGNAT나 학교, 회사처럼 **공인 IP 하나 뒤에 대기자가 몰리면 정상 사용자가 한꺼번에 막힐 수 있었습니다**.
- 대기열 경로는 이 제한에서 빼고 IP당 5분 300,000건으로 따로 제한했습니다.
- 이 값은 대기자 한 명이 2~4초마다 순번을 조회해 5분에 약 100건을 보내는 것을 기준으로 **IP 하나 뒤 약 3,000명**을 받도록 계산했습니다.

③ 오탐 경로는 통째로 열지 않고 크기 상한으로 대신 검사

CloudFront의 WAF에는 AWS 관리형 규칙 3종(Common, Known Bad Inputs, IP Reputation)을 적용했는데, 실제로 쓰는 중에 아래 세 경로가 막혔습니다.

막힌 경로	걸린 규칙
카카오 로그인 콜백	본문의 `redirectUri`를 SSRF로 판단
상품 사진 업로드	XSS, LFI 규칙과 본문 8KB 상한
관리자 상품 등록	본문 8KB 상한

- 관리형 규칙은 **Count 모드로 라벨만 붙이고** 다음 규칙에서 경로와 라벨을 함께 보고 차단할지 정하게 해 예외를 경로 단위로 줄 수 있게 했습니다.
- 예외 경로가 아니면 라벨이 붙은 요청을 차단하고 예외로 둔 경로는 검사 없이 열리지 않도록 업로드 10MiB, 관리자 API 512KB처럼 **경로마다 정한 본문 크기 상한으로 대신 검사**했습니다.

결과

- ALB로 가는 요청이 모두 **CloudFront와 WAF를 거치게** 되어 막힌 요청은 VPC 안의 ALB까지 오지 않았습니다.
- 카카오 로그인과 사진 업로드, 관리자 상품 등록이 더 이상 막히지 않았고 업로드와 관리자 API에는 본문 크기 검사가 남았습니다.
- 실제 사용자와 같은 CloudFront, WAF, ALB 경로로 보낸 판매 오픈 부하 시험에서 첫 1분 동안 **5xx 없이 초당 약 2,090건**을 처리했습니다. (05 사례)

개선할 점

- ALB가 아직 public subnet에 있어 CloudFront VPC origin으로 **ALB를 private subnet에 옮기면** 비밀 헤더와 prefix list 없이도 인터넷에서 ALB로 가는 경로 자체를 없앨 수 있습니다.
- WAF 로그를 수집하지 않아 오탐 원인을 샘플 요청으로만 확인했기 때문에 **WAF 로그를 S3로 모으면** 차단된 요청을 규칙과 경로별로 바로 확인할 수 있습니다.

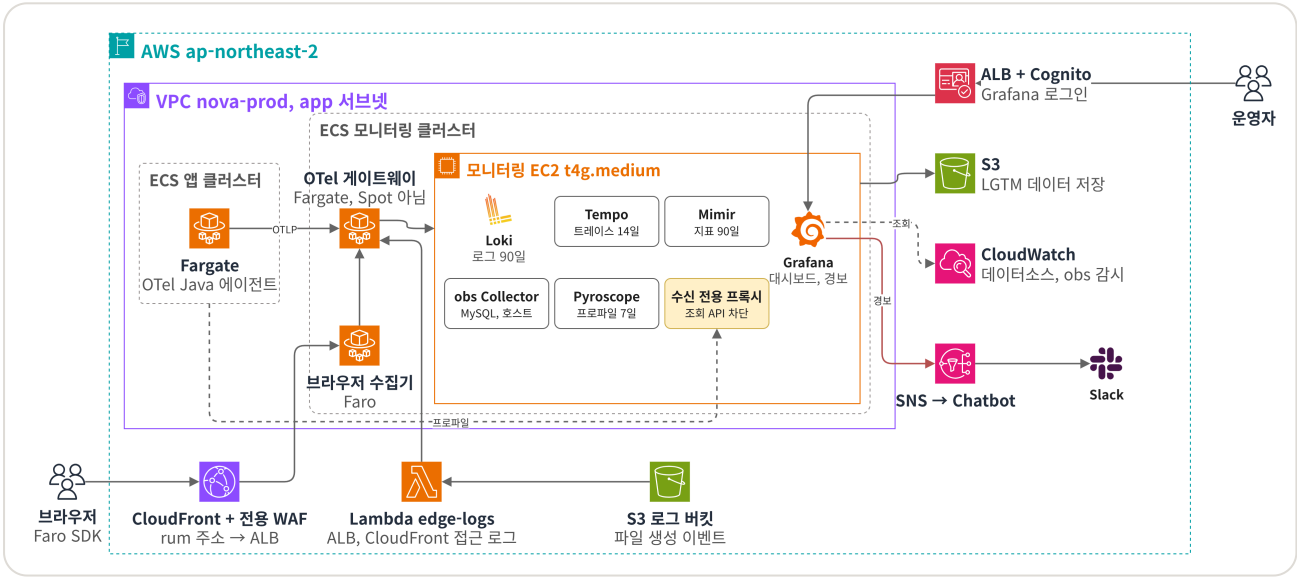
원인 분석을 수십 분에서 수 분으로 줄인 OpenTelemetry와 Grafana 통합 모니터링

지표 · 로그 · 트레이스를 한곳에 수집 | 요청 시작부터 DB까지 한 화면에서 추적 | 원인 분석 수십 분 → 수 분

설계 배경

- 지표는 CloudWatch에, 로그는 서비스마다 흩어져 있어 장애가 발생했을 때 둘을 오가며 **원인을 찾는 데 수십 분이 걸렸습니다**.
- 한 요청이 여러 서비스를 거치는데 서비스별 로그만으로는 요청의 흐름을 이어 볼 수 없었습니다.
- 사용자 영향이 생기면 알림을 받고 요청 시작부터 DB까지 한 화면에서 따라갈 수 있는 구조가 필요했습니다.

구축 방식



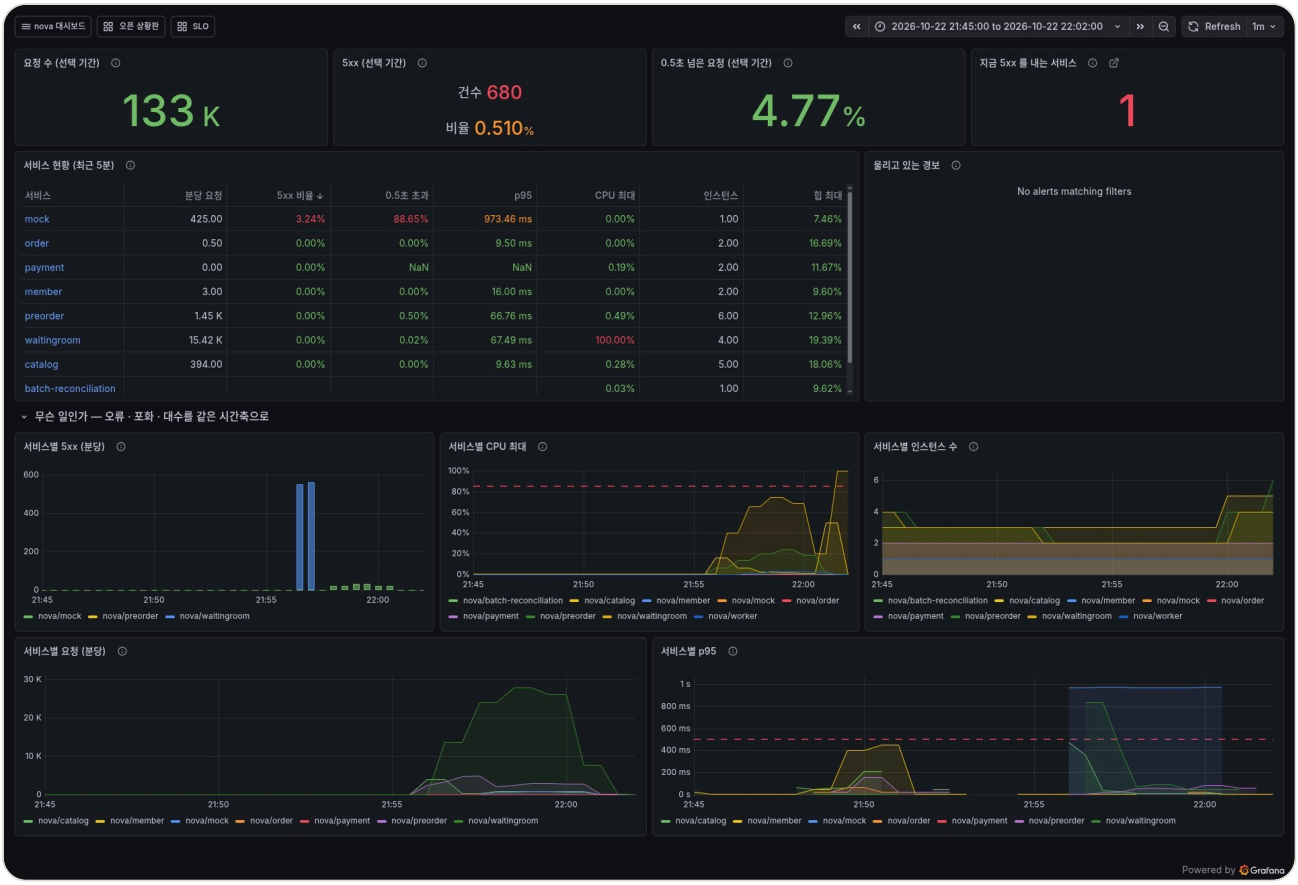
▲ 앱, 브라우저, 엣지 로그가 OTel 게이트웨이를 거쳐 각 저장소에 모이고 Grafana 경보가 Slack으로 가는 구성

① 지표 · 로그 · 트레이스 · 프로파일을 한곳에 수집

- OpenTelemetry Java Agent와 Pyroscope 확장으로 **앱 코드 수정 없이** 지표, 로그, 트레이스, 프로파일을 수집하고 브라우저 오류와 속도는 Faro RUM으로 모았습니다.
- CloudWatch 대신 EC2 한 대에 Mimir, Loki, Tempo, Pyroscope를 올리고 S3에 저장해 Grafana 한곳에서 조회하게 했습니다.
- 트레이스는 **오류와 느린 요청은 전부, 나머지는 일부만 저장**해 서버 부담을 줄이고 지표는 모든 요청으로 계산해 정확도를 유지했습니다.

② 사용자 영향에서 원인 순서로 대시보드 배치

개요 실제 화면



▲ 요청 수와 5xx, 0.5초 넘은 요청, 지금 5xx를 내는 서비스, 서비스별 5xx와 CPU, 인스턴스 수, p95

- 개요 화면은 **사용자 영향을 맨 위에** 두고 그 아래 서비스별 5xx, CPU, 인스턴스 수를 같은 시간축에 놓아 오류와 포화가 겹치는 시점을 비교할 수 있게 했습니다.
- 서비스 상세 화면은 위에서 아래로 원인을 좁혀 가는 순서로 두고 **trace ID로 로그와 트레이스를 오가게** 했습니다.

③ 사용자 영향이 있을 때만 critical 경보 발송

- SLA와 SLO를 가상으로 정해 **가용성 99.9%, 요청의 95%가 0.5초 안**을 기준으로 삼고 서비스별 5xx와 SLO 예산 소진처럼 사용자가 겪는 증상에만 critical 경보를 걸었습니다.
- JVM 힙, DB 연결 사용률 같은 **원인 지표는 warning으로 낮추고** 수신처를 나눠 critical이 묻히지 않게 했습니다.

결과

여러 도구를 오가며 수십 분 걸리던 원인 분석을 Grafana 한 화면에서 요청 시작부터 DB까지 따라가 **수 분으로 줄였습니다**.

부하 시험에서는 이 화면으로 아래 문제를 찾아 고쳤습니다.

찾은 문제	근거	조치
접수 응답 지연	경보에서 트레이스로 따라가 DB 행 잠금 대기 468ms 확인	잠금을 커밋 직전으로 옮겨 p50 521ms → 48ms

찾은 문제	근거	조치
급증 뒤에야 붙는 증설	CPU와 경보 시각을 같은 시간축에서 비교해 약 3분 지연 확인	오픈 전 사전 증설로 변경
대기열과 상품 조회 CPU 포화	서비스별 CPU 그래프에서 포화된 서비스 확인	태스크 사양과 최소 대수 상향
반복되는 배포 롤백	새 태스크가 CPU 부족으로 기동을 반복하다 롤백되는 것 확인	CPU 할당 조정

운영 경로 그대로 판매 오픈을 재현하는 ECS 기반 k6 부하 시험 플랫폼

7가지 시나리오로 판매 오픈 재현 · 운영 SLO와 같은 기준을 흐름별로 판정 · 여러 대의 결과는 Mimir 히스토그램으로 합산

설계 배경

- 판매 오픈 몰림에 운영 인프라가 어떻게 반응하는지는 **운영 경로로 실제 부하를 보내야** 알 수 있었습니다.
- 대기열에 선 회원은 입장할 때까지 k6 VU를 하나씩 잡고 있어 판매 오픈을 재현하려면 **VU를 최대 15,000개까지** 띄워야 했습니다.
- VU 하나가 1~5MB를 써서 부하를 만드는 쪽에만 메모리가 수십 GB 필요했습니다.

설계 판단

항목	결정	이유
실행 위치	평소 0대인 시험 전용 ECS 클러스터	시험마다 필요한 대수만 띄우고 끝나면 비용 없음
k6 태스크	4 vCPU 8GB 온디맨드 태스크를 부하에 맞춰 N대	VU 메모리는 대수로 늘리고 Spot 회수로 시험이 끊기지 않게 함
vCPU 할당량	Fargate vCPU 할당량 상향 요청	운영 서비스와 같은 할당량을 써서 k6를 늘리면 운영 증설 여유가 줄어듦
부하 경로	CloudFront → WAF → ALB → 서비스	실제 사용자와 같은 경로
k6 네트워크	public 서브넷의 공인 IP	NAT를 거치면 서비스보다 NAT가 먼저 막힘
실행 방법	GitHub Actions 수동 실행과 승인	상시 서버 없이 OIDC로 실행하고 실행 기록이 남음

> **부하를 만드는 쪽이 먼저 막히면 서비스의 한계를 잴 수 없어 NAT와 Spot 회수, vCPU 할당량처럼 시험 도구가 병목이 될 곳을 먼저 없앴습니다.**

◯ 구현

▲ 평소 0대인 시험 클러스터에서 k6를 띄워 운영과 같은 경로로 부하를 보내고 Mimir에서 판정하는 구성

① 실행 구조와 시험 회원

- GitHub Actions에서 시나리오를 실행하고 승인하면 VPC 안의 컨트롤러가 k6를 띄우고 Mimir 지표로 판정하게 만들었습니다.
- 예약은 회원 한 명당 1건이라 **시험 회원 10만 명**을 DB에 미리 넣어 두고 한 시험에서 회원 한 명은 한 번만 진입하게 했습니다.
- 카카오 로그인은 자동화할 수 없어 토큰 발급기가 AWS 안에서 시험마다 필요한 회원의 토큰을 k6 태스크별로 겹치지 않게 만들었습니다.

② 최대 15,000명의 동시 요청 만들기

- k6의 실행 구간 기능으로 부하와 VU를 N대에 똑같이 나눠 초당 500건이면 5대가 100건씩 보내게 했습니다.
- 태스크 대수는 초당 요청 수가 아니라 **동시에 기다리는 회원 수에 VU당 메모리를 곱해** 정하고 30% 여유를 두었습니다.
- 응답 본문은 대기열과 접수처럼 꼭 필요한 요청만 받아 VU 하나가 쓰는 메모리를 줄였습니다.
- Fargate 태스크는 뜨는 시간이 30~60초씩 달라 컨트롤러가 약 2분 뒤의 시작 시각을 정해 넘기고 k6 이미지에 넣은 Go 시작 프로그램이 **그 시각까지 기다렸다가 함께 시작**하게 했습니다.
- 1만 명 queue 시험에서 입장이 늦어지자 예상치만큼만 잡아 둔 VU가 모자라 **2,124명이 시작하지 못해** VU를 인원만큼 늘릴 수 있게 하고 태스크 대수도 그 VU에 맞춰 계산하도록 고쳤습니다.

③ 시험 방법

평시부터 판매 오픈과 한계 상황까지 7가지 시나리오로 나눠 시험했습니다.

시나리오	목적	부하
smoke	적은 부하에서 모든 흐름이 오류 없이 끝나는지	평시 수준을 2분
load	평시의 2배를 오래 받아도 기준 안에서 처리하는지	평시의 2배를 30분
spike	판매 오픈 10초 몰림을 기준 안에서 처리하는지	10초 동안 진입 5,000건 뒤 10분 관찰
rush	몰린 회원이 진입부터 결제 시도까지 끝내는지	spike와 같은 몰림에서 절반이 결제 시도

시나리오	목적	부하
queue	입력한 인원이 모두 대기열을 지나 접수되는지	5,000명을 초당 500명씩 진입
stress	부하를 올릴 때 가장 먼저 막히는 곳	평시의 1·2·4·6·8배를 5분씩
soak	4시간 동안 커넥션과 메모리, 캐시 키가 새지 않는지	평시 수준을 4시간

- 모든 시나리오는 **평시 사용자 비율**(상품 조회 80 : 예약 조회 10 : 진입과 접수 5 : 주문과 결제 5 : 취소 1)로 요청을 섞고 흐름마다 프론트 화면과 같은 순서로 API를 불렀습니다.
- 부하는 동시 사용자 수가 아니라 **초당 들어오는 사용자 수**로 정해 서버가 느려져도 보내는 양이 줄지 않게 했습니다.

④ 통과 조건과 판정

- 합격 기준은 운영 SLO와 같은 **p95 0.5초 이하와 5xx 0.1% 이하**로 정했습니다.
- 요청의 80%가 가벼운 상품 조회라 전체 p95만 보면 접수가 느려도 합격으로 나올 수 있어 **접수나 결제 같은 흐름마다 따로** 기준을 적용했습니다.
- k6 여러 대가 각자 계산한 p95는 평균을 내도 전체 p95가 되지 않아 응답 시간을 히스토그램으로 Mimir에 모아 한 번에 계산했습니다.
- 히스토그램의 p95는 근사값이라 0.5초를 구간 경계로 두고 "0.5초 안에 끝난 요청이 95% 이상인가"로 판정해 오차를 없앴습니다.

결과가 서비스의 문제인지 부하 도구의 문제인지 구분할 수 있게 판정을 네 가지로 나눴습니다.

판정	조건
합격	모든 k6가 끝까지 돌고 전체와 흐름별 기준을 모두 지킴
불합격	전체나 흐름별 기준 중 하나라도 넘음
판정 불가	지표가 1분 이상 끊기거나 k6가 계획한 부하를 다 보내지 못함
한계 측정	stress에서 기준을 지킨 마지막 단계와 처음 막힌 곳을 기록

목표 구조는 코드에 남기고 월 운영비를 73% 줄인 Terraform 변수 기반 비용 절감

월 운영비 약 $890 → $240 | 변수 하나로 목표 구조 원복 | 6주 예산 40만 원 안에서 운영

문제 발견

- 처음 설계한 목표 구조는 Multi-AZ, NAT Gateway, VPC 엔드포인트를 포함해 **월 약 890달러**로 6주 예산 40만 원을 넘었습니다.
- 설계 자체를 낮추면 트래픽이 늘거나 예산이 생겼을 때 다시 만들어야 했습니다.

> **목표 구조를 버리지 않고 비용만 줄이는 방법이 필요했습니다.**

1차 개선

- NAT 방식, Multi-AZ, VPC 엔드포인트처럼 비용이 큰 항목을 `nat_mode`, `multi_az`, `enable_interface_endpoints` 같은 **Terraform 변수로 분리**하고 기본값만 비용이 낮은 쪽으로 두었습니다.
- 목표 구조 코드는 그대로 남아 있어 **변수 값만 바꾸면 목표 구조로 돌아가게** 했습니다.
- NAT 인스턴스를 교체하는 동안 그 AZ의 외부 통신이 끊기는 점처럼 항목마다 감수할 점과 목표 구조로 되돌리는 방법을 문서에 남겼습니다.
- 모니터링 스택은 CloudWatch 대신 EC2 한 대에 Mimir, Loki, Tempo, Pyroscope를 올리고 S3에 저장했습니다. (04 사례)

배포 후 확인

- 전체 서비스를 배포한 뒤 하루 비용이 **20달러 안팎**으로 예상보다 많이 나왔습니다.
- Cost Explorer로 항목별로 나눠 보니 Fargate가 가장 컸고 **Container Insights 지표 요금이 그다음**이었습니다.

2차 개선

- Container Insights는 그 지표를 쓰던 곳을 아래처럼 먼저 옮긴 뒤 꺼 **월 약 75달러**를 줄였습니다.

없어지는 지표	쓰던 곳	대체
실행 중 태스크 수	worker 확장 기준, 태스크 0개 경보	무료 AWS/ECS 지표 LiveTaskCount
모자란 태스크, 뜨는 중 태스크	Grafana 개요	ECS 이벤트 경보로 배포 실패, 배치 실패, 비정상 종료 알림
태스크 네트워크, 임시 스토리지	컴퓨팅 대시보드	판단에 쓰지 않던 지표라 대체하지 않음

- Fargate는 부하 시험으로 트래픽을 감당하는 최소 대수와 사양을 찾아 접수는 2 vCPU에서 1 vCPU로, 외부 시스템 목업은 2대에서 1대로 낮췄습니다.
- 부하 시험 리소스는 전용 ECS 클러스터에 모으고 평소에는 0대로 두어 시험할 때만 비용이 생기게 했습니다.

결과

월 약 890달러로 추정된 목표 구조를 **월 약 240달러**로 73% 줄여 예산 안에서 운영했습니다.

항목	목표 구조	현재 운영	월 절감(추정)
Fargate 용량	서비스마다 온디맨드 1대 + 증설분 Spot	전부 Spot (외부 시스템 목업만 온디맨드)	약 160달러

항목	목표 구조	현재 운영	월 절감(추정)
VPC 엔드포인트	ECR, Logs, Secrets Manager, SSM 인터페이스형	제거하고 NAT 경유 (S3는 무료 게이트웨이형 유지)	약 130달러
RDS	Multi-AZ (대기 인스턴스 포함)	단일 AZ (db.t4g.large)	약 120달러
Container Insights	켬 (태스크 단위 지표)	끔, 무료 AWS/ECS 지표와 이벤트 경보로 대체	약 75달러
NAT	AZ별 NAT Gateway	AZ별 NAT 인스턴스 (fck-nat t4g.micro)	약 70달러 + 처리 요금
Valkey	복제본 + 자동 장애 조치	노드 1개	약 20달러
기타	mock 2대 → 1대, preorder 2 vCPU → 1 vCPU, Atlantis 전용 ALB → 메인 ALB 공유, 엣지 로그 내부 NLB → DNS 레코드, Flow Logs 끔		약 75달러

월 절감액은 서울 리전 정가 기준 추정치입니다.

줄인 항목 대부분은 **변수 값만 바꾸면 목표 구조로 돌아가** 트래픽이 늘거나 예산이 생기면 절감액이 큰 항목부터 되돌릴 수 있습니다.

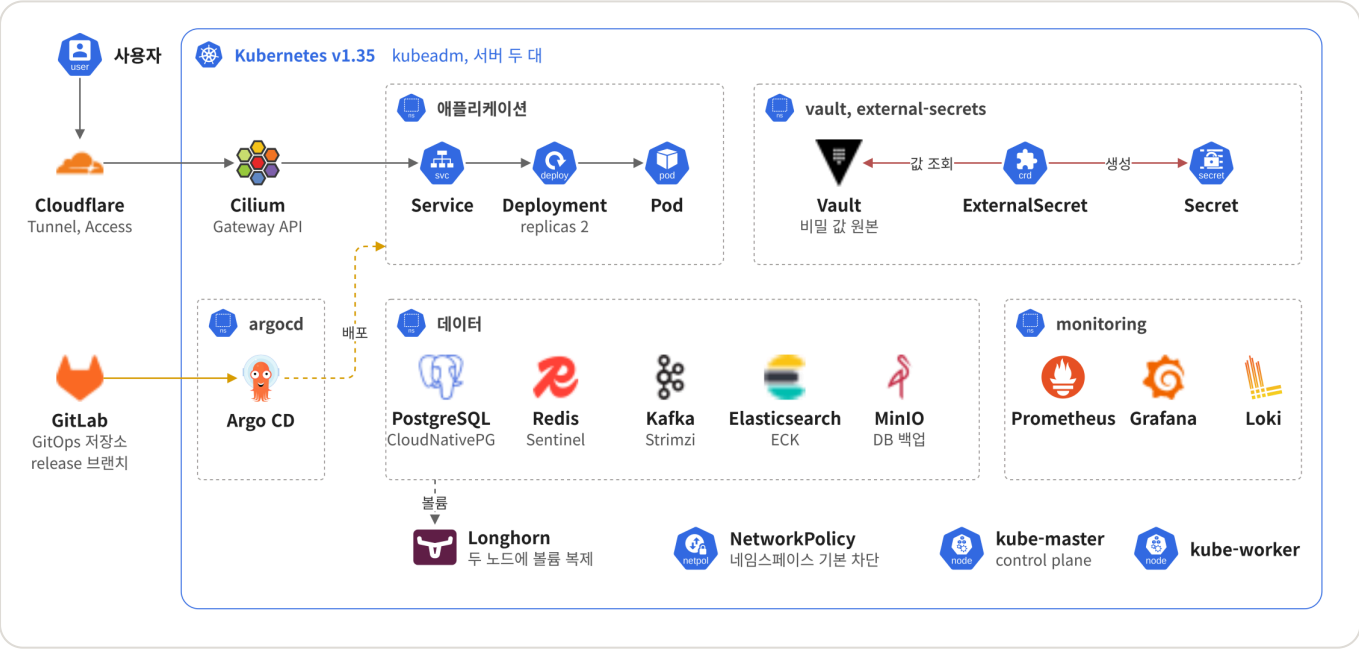
홈랩 Kubernetes 플랫폼

서버 두 대에 Kubernetes를 직접 설치해 운영하는 홈랩

기간	2026.05 ~ 현재, 1인
저장소	sago-panda/sapari-helm-manifest
기술	`kubeadm` `Cilium` `Hubble` `Gateway API` `Cloudflare Tunnel` `Longhorn` `GitLab CI` `cosign` `Argo CD` `Vault` `External Secrets` `Prometheus` `Grafana` `Loki`

프로젝트 개요

Rocky Linux 서버 두 대에 kubeadm으로 설치한 클러스터에 개인 프로젝트 애플리케이션과 Keycloak, 데이터베이스, 모니터링 스택을 올려 운영합니다. 네트워크, 스토리지, 배포, 비밀 값 관리를 직접 구성했고 클러스터 안 컴포넌트는 Argo CD가 Git에서 가져와 배포합니다. 외부 요청은 Cloudflare Tunnel을 거쳐 공개용과 관리용 Gateway로 나뉘어 들어오고, Argo CD와 Grafana, Vault 같은 관리 도구에만 Cloudflare Access 인증을 걸었습니다.



▲ Cloudflare Tunnel에서 Cilium Gateway로 들어오는 요청 경로와 네임스페이스별 구성

Hubble 통신 기록을 활용한 CNI 전환 중 DNS 장애 분석

Hubble에 기록이 없어 Cilium 밖의 문제로 판단 | 원인은 제거하지 않고 남은 calico-node | Calico 제거로 DNS 복구

문제 발견

- Calico 무료 버전은 도메인 단위 허용을 지원하지 않아 CNI를 Cilium으로 바꿨습니다.
- 전환한 날 Keycloak이 메일을 보내지 못했고 시험 Pod에서 메일 서버 도메인을 조회하면 **timeout** 됐습니다.
- Redis Pod의 DNS 조회만 정상이었습니다.

원인 분석

① 메일 서버 도메인 허용 정책

- 전환한 날 넣은 정책을 먼저 의심해 지우고 다시 조회했지만 timeout이었습니다.
- 나중에 보니 Argo CD가 지운 정책을 몇 초 만에 되살려 정책이 있는 채로 시험한 것이었습니다.

② kube-proxy 설정

- Pod 대역 설정 때문에 응답 경로가 어긋났는지 보려고 `clusterCIDR` 을 확인했지만 빈 값이라 원인이 아니었습니다.

③ Hubble의 DNS 통신 기록

- 정상인 Redis는 요청과 응답이 모두 기록됐는데 실패한 Pod의 조회는 **DROPPED 기록도 없었습니다**.
- Cilium이 막았다면 기록이 남으므로 패킷이 Cilium에 닿기 전에 막힌다고 보고 정책 대신 노드를 조사했습니다.

```
$ kubectl -n kube-system exec <cilium-pod> -- hubble observe --port 53 --last
storage/redis-node-* → coredns   to-endpoint FORWARDED (UDP)
```

▲ Hubble의 DNS 통신 기록 출력 일부

④ 노드의 Pod와 CNI 설정

- Cilium 밖에서 패킷을 막는 것을 찾으려고 IP가 Cilium 대역이 아닌 Pod를 추렸더니 **calico-node가 두 노드에서 실행 중** 이었습니다.
- Cilium도 다른 CNI와 함께 쓰는 전환 중 설정 그대로였습니다.

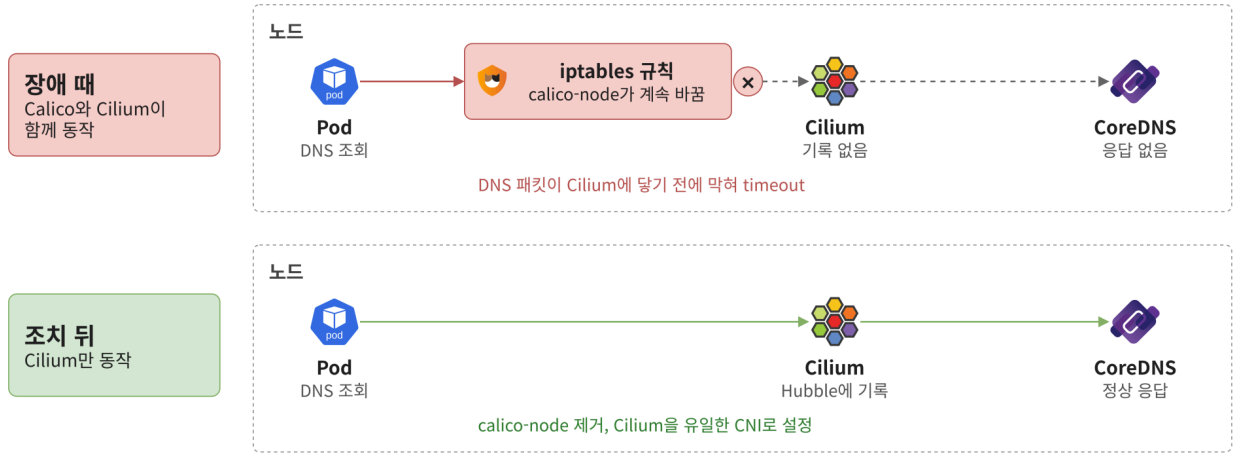
```
$ kubectl get pods -A -o wide | grep -v "10.244\."
calico-node-7h4jd   kube-worker   Running   15d

$ helm -n kube-system get values cilium | grep -A4 "cni:"
  customConf: true
  exclusive: false
```

▲ 노드에서 확인한 Pod와 Cilium 설정 출력 일부

원인

전환 절차의 마지막 단계인 Calico 제거가 남아 있어 calico-node가 **두 노드의 iptables 규칙을 계속 다시 썼고** DNS 패킷이 Cilium에 닿기 전에 막혔습니다.



▲ 남아 있던 calico-node 때문에 DNS 패킷이 Cilium에 닿지 못하던 장애 때와 Calico를 제거한 뒤

조치

- **Calico를 제거하고 Cilium을 유일한 CNI로** 설정했습니다.
- 명령으로 바꾼 값은 Git의 values 파일에도 넣어 다음 upgrade에서 되돌아가지 않게 했습니다.

```
$ kubectl delete -f calico.yaml
$ helm upgrade cilium cilium/cilium -n kube-system -f bootstrap/cilium-values.
    --set cni.customConf=false --set cni.exclusive=true
$ kubectl -n kube-system rollout restart ds/cilium
```

▲ Calico를 제거하고 Cilium을 유일한 CNI로 설정한 명령

결과

항목	이전	이후
DNS 조회	Redis를 뺀 Pod에서 timeout	도메인 허용 정책을 둔 채로 정상
외부 메일 서버 허용	사설망을 뺀 모든 주소	메일 서버 도메인 하나

배운 점

- CNI를 바꿀 때는 이전 CNI를 지우는 단계까지 끝낸 뒤에 동작을 확인합니다.
- Argo CD가 관리하는 리소스를 지워 시험할 때는 시험 Pod를 먼저 띄워 두고 지웁니다.
- Hubble에 차단 기록이 없으면 정책보다 노드를 먼저 봅니다.

디스크가 남았는데도 볼륨이 만들어지지 않던 Longhorn 스토리지 장애 해결

Pod 이벤트에서 볼륨 · 노드 · 디스크 순으로 추적 | 원인은 root 파티션만 등록된 Longhorn 디스크 | /home 파티션 추가로 볼륨과 Pod 복구

문제 발견

Prometheus, Alertmanager, Kafka Pod가 **Pending과 ContainerCreating에서 멈춰** 시간이 지나도 Running이 되지 않았습니다.

원인 분석

① Pod 이벤트

- Pod 이벤트에 볼륨을 붙이지 못했다는 경고가 반복돼 애플리케이션이 아니라 **볼륨 문제**로 봤습니다.

```
$ kubectl -n monitoring describe pod prometheus-...-0 | tail -20
Warning  FailedAttachVolume  volume pvc-... is not ready for workloads
```

▲ Pod 이벤트 출력 일부

② Longhorn 볼륨 상태

- 정상이 아닌 볼륨만 추려 보니 새 볼륨들이 **어느 노드에도 붙지 못한 faulted 상태**였습니다.

```
$ kubectl -n longhorn-system get volumes.longhorn.io -o custom-columns=\
'VOL:.metadata.name,STATE:.status.state,ROBUST:.status.robustness,NODE:.status
VOL          STATE      ROBUST     NODE          SIZE
pvc-87ab...  detached   faulted    <none>        5Gi
pvc-289b...  attached   degraded   kube-worker   10Gi
```

▲ Longhorn 볼륨 상태 출력 일부

③ 노드 상태

- Longhorn에서 본 두 노드는 모두 Ready였고 볼륨 배치도 가능해 노드 장애는 아니었습니다.

④ Longhorn이 쓰는 디스크와 실제 디스크

- Longhorn 데이터 경로는 **70GB root 파티션**에 있었고 같은 디스크의 **391GB /home 파티션은 비어 있었습니다**.

```
$ df -h /var/lib/longhorn
/dev/mapper/rl-root   70G   25G   46G   36%   /

$ lsblk
nvme0n1      476.9G
├── rl-root      70G   /
└── rl-home     391G   /home
```

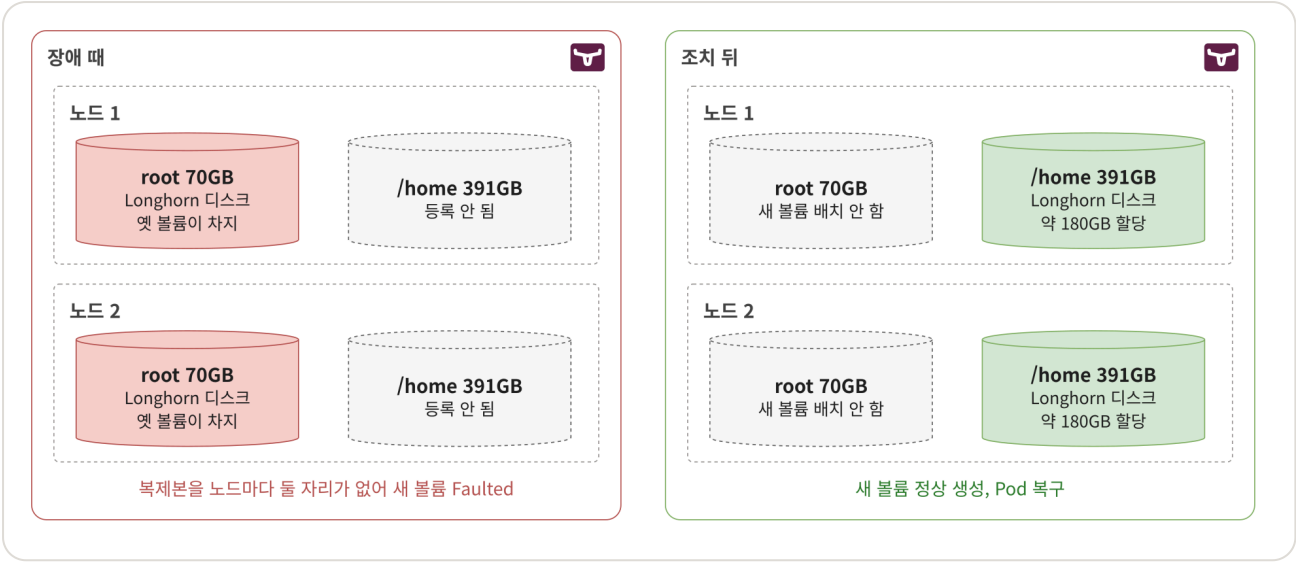
▲ Longhorn 데이터 경로의 파티션과 노드의 디스크 출력 일부

⑤ 지워지지 않고 남은 볼륨

- 재설치하며 지운 옛 Kafka와 Redis의 볼륨이 Released 상태로 남아 root 파티션을 차지하고 있었습니다.

◯ 원인

- Longhorn이 쓸 수 있는 공간이 **노드마다 root 파티션 70GB**뿐이었습니다.
- 복제본 2개를 서로 다른 노드에 둬야 하는데 옛 볼륨까지 남아 **두 노드 모두 자리가 없어** 새 볼륨이 faulted가 됐습니다.



▲ root 파티션만 쓰던 장애 때와 /home 파티션을 디스크로 추가한 조치 뒤의 Longhorn 디스크 구성

조치

- 두 노드의 **/home 파티션을 Longhorn 디스크로 추가**하고 root 파티션에는 새 볼륨이 배치되지 않게 했습니다.
- Released로 남은 옛 볼륨과 faulted 볼륨을 지우고 PVC를 다시 만들었습니다.
- 자체 복제를 하는 Kafka는 복제본 1개에 회수 정책이 Delete인 StorageClass를 따로 만들어 옮겼습니다.

결과

항목	이전	이후
Longhorn 디스크	노드마다 root 파티션 70GB	노드마다 /home 파티션에서 약 180GB
새 볼륨	faulted	정상이 아닌 볼륨 없음
Prometheus, Alertmanager, Kafka Pod	Pending, ContainerCreating	Running

배운 점

- Longhorn을 설치할 때는 쓸 디스크를 직접 지정합니다.
- 재설치가 잦은 볼륨은 회수 정책을 Retain 대신 Delete로 둡니다.