CLOUD-NATIVE PLATFORM · DEVOPS PORTFOLIO

Notiflex Platform

B2B 알림 SaaS의 쿠버네티스 운영 환경을 빈 클러스터부터 구축했습니다. GitOps 배포 파이프라인 · 3축 관측 가능성 · 무중단 배포 전략 · 이벤트 기반 아키텍처를 하나의 플랫폼으로 통합한 **엔드투엔드 인프라 프로젝트**입니다.

ROLE	PERIOD	PLATFORM	REPOSITORY
DevOps Engineer · 1인	2026.07 · 약 3주	GKE Standard · GCP	github.com/easydong02/notiflex-platform

8 구축 단계 (기반 → 고도화)	**15+** 통합 오픈소스 도구	**3** 배포 전략 진화 단계
16 아키텍처 결정 기록(ADR)	**84** Git 커밋 (전 이력 추적)	**4** GitOps 앱 전부 Synced

CORE STACK

`GKE Standard` `ArgoCD` `Argo Rollouts` `Prometheus` `Grafana` `Loki` `Tempo` `Kafka` `Valkey` `Gateway API` `GitHub Actions` `OpenTelemetry` `Strimzi` `Workload Identity` `Secrets Store CSI` `Fluent Bit` `Alertmanager` `Go (stdlib)`

CONTENTS

02 — OVERVIEW

프로젝트 개요

"코드는 돌아가지만 운영은 없는" 상태에서 출발해, 프로덕션급 운영 환경으로 끌어올린 전 과정입니다.

Notiflex는 B2B 알림 SaaS 플랫폼입니다. 애플리케이션은 외부 프레임워크 없는 **Go 표준 라이브러리**로 작성해 `scratch` 컨테이너로 빌드했고, 인프라는 비용 효율을 위해 **Spot VM** 위에서 운영했습니다. 모든 변경은 **Git을 단일 진실 소스**로 삼아 선언적으로 관리되며, 클러스터 상태는 ArgoCD가 자동으로 Git과 일치시킵니다. 이 극단적 리소스 제약이 곧 리소스 최적화와 장애 대응 역량을 검증하는 무대가 되었습니다.

BEFORE · 시작 상태

- 배포가 수동 `kubectl apply` — 무엇이 왜 배포됐는지 추적 불가
- 장애를 사용자 제보로 인지 — 메트릭·로그·알림 전무
- 배포 시 다운타임 발생, 롤백은 재빌드 후 재배포
- 시크릿이 매니페스트 평문 / SA 키 파일에 노출
- 동기 처리 병목 — 요청이 몰리면 응답 지연
- 단일 노드풀에 모든 워크로드 혼재

→

AFTER · 완성 상태

- push 한 번으로 빌드 → 배포까지 무인 진행, selfHeal이 수동 변경 자동 복구
- 메트릭·로그·트레이스 3축을 Grafana 단일 창구로 통합, 알림 자동 발송
- Canary 점진 배포(20→50→80→100%), 롤백은 Git revert 하나
- Secret Manager + Workload Identity로 **키 파일 없는** 인증
- Kafka 비동기 처리로 요청 경로에서 분리
- 역할별 노드풀 분리 + Namespace 멀티테넌시

운영 환경 구성

영역	구성	비고
클러스터	GKE Standard (Zonal) · `asia-northeast3-a`	서울 리전 · 단일 존
노드풀	`default-pool` e2-medium×2 · `api-pool` e2-medium×1 · `worker-pool` e2-standard-2×1	전부 Spot VM (비용 최적화)
네임스페이스	notiflex · enterprise · kafka · argocd · argo-rollouts · monitoring	테넌트/기능별 격리
GKE 기능	Gateway API · Workload Identity · Secret Manager CSI	클러스터 생성 시 사전 활성화

담당 범위 — 1인 프로젝트로 설계부터 운영까지 전 영역 수행

인프라 프로비저닝

GKE 클러스터·노드풀 설계 및 생성, Artifact Registry 구성, 리전/존·머신 타입·Spot 여부까지 비용을 고려해 결정.

CI/CD 파이프라인 구축

GitHub Actions 워크플로 작성, ArgoCD 설치·저장소 연동, App of Apps 구조 설계 및 sync-wave 기동 순서 제어.

애플리케이션 개발

Go 표준 라이브러리로 API 작성, Valkey·Kafka 클라이언트 연동, OpenTelemetry 계측 미들웨어 구현, scratch 이미지 빌드.

관측 체계 설계

메트릭·로그·트레이스 수집 경로 설계, Grafana 대시보드 및 데이터소스 구성, 알림 규칙 작성과 실제 발생 검증.

보안 · 권한 설계

Workload Identity 매핑, GCP IAM 역할 부여, Secret Manager 연동과 CSI 마운트 구성으로 키 파일 없는 인증 체계 수립.

운영 · 장애 대응

리소스 예산 관리, 장애 원인 분석과 복구, 결정 근거(ADR)·진행 이력(JOURNEY)·아키텍처 스냅샷 문서화.

03 — TECH STACK

기술 스택

각 도구는 "탐색 → 비교 → 결정" 3단계로 대안과 견주어 선택했고, 그 근거를 16개 ADR로 저장소에 남겼습니다.

CLOUD / INFRASTRUCTURE

GKE Standard (Zonal) · Spot VM · Artifact Registry · Cloud Build · 멀티 노드풀

GITOPS / CI · CD

ArgoCD · Argo Rollouts · GitHub Actions · App of Apps · selfHeal

OBSERVABILITY

Prometheus · Grafana · Loki · Fluent Bit · Alertmanager · Tempo · OpenTelemetry

DATA / MESSAGING

Valkey · Apache Kafka · Strimzi Operator · KRaft

SECURITY

GCP Secret Manager · Workload Identity · Secrets Store CSI · Keyless 인증

TRAFFIC / APPLICATION

Gateway API · HTTPRoute · Go (stdlib) · scratch image

관측 가능성 3축 — 구성 요소와 역할

축	도구	역할	수집 방식
메트릭	Prometheus + Grafana	애플리케이션 `/metrics` 및 클러스터 지표 수집·시각화	ServiceMonitor 기반 scrape
로그	Loki + Fluent Bit	전체 Pod 로그를 라벨 기반으로 인덱싱해 단일 창구에서 조회	노드별 DaemonSet 수집
트레이스	Tempo + OpenTelemetry	요청 하나의 전체 여정을 span 단위로 추적 (Valkey·Kafka 자식 span)	OTLP gRPC :4317
알림	Alertmanager	PrometheusRule 임계치 위반 시 자동 라우팅 (firing → resolved 검증 완료)	PrometheusRule CRD

애플리케이션 — GO 표준 라이브러리 기반

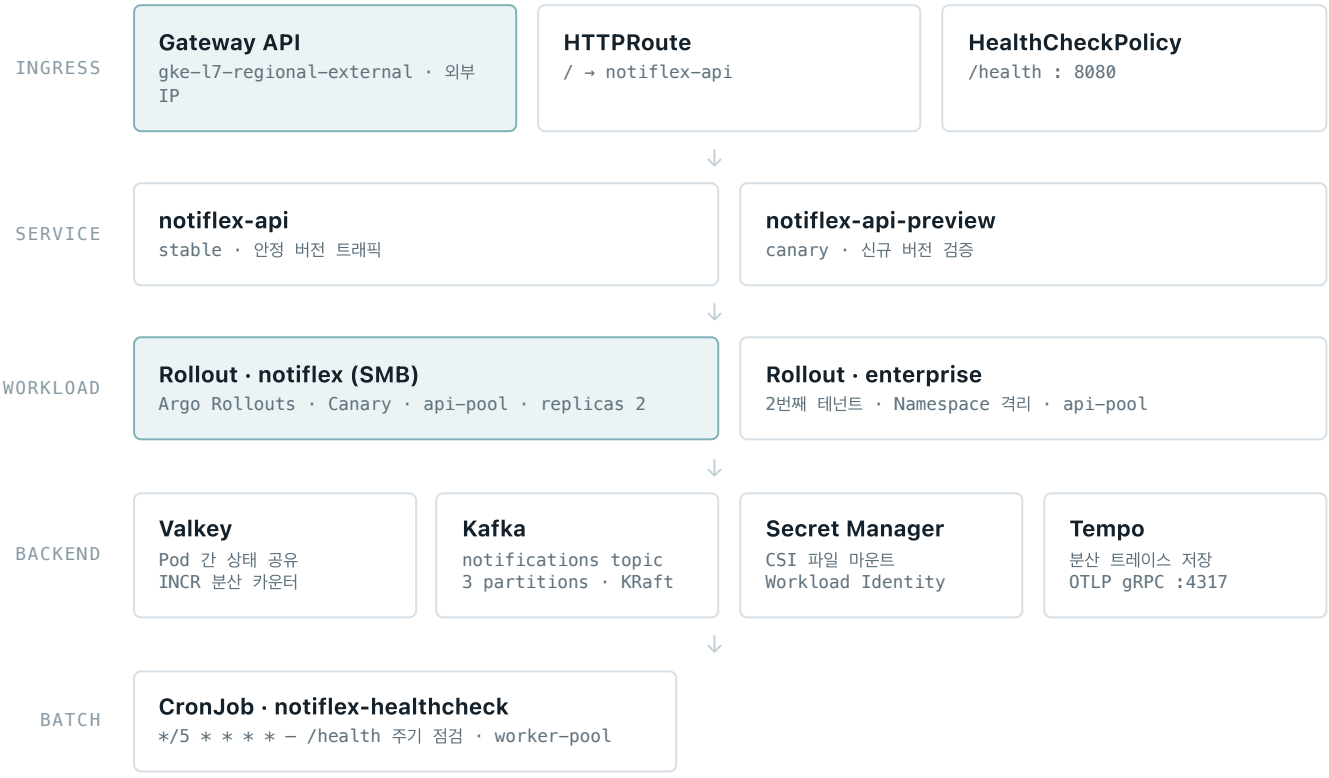
엔드포인트	역할	연동
`/health`	헬스체크 — Gateway HealthCheckPolicy · CronJob 주기 점검 대상	Gateway API · CronJob
`/metrics`	Prometheus 노출 지표	ServiceMonitor
`/version`	배포된 이미지 버전 확인 — 롤아웃 진행 상황 실시간 관측	Canary 검증에 활용
`/id`	분산 ID 발급 — 캐시 카운터 증가 후 알림 메시지 발행	Valkey INCR · Kafka Producer · OTel span

03 — ARCHITECTURE

시스템 아키텍처

외부 요청은 Gateway API로 진입해 Canary Rollout으로 라우팅되고, 캐시·메시징·시크릿·트레이싱 백엔드와 연결됩니다.

런타임 요청 경로



GITOPS 배포 파이프라인 — 코드 PUSH부터 프로덕션 반영까지 무인 진행

ARGOCD APP OF APPS — 선언 하나로 전체 애플리케이션 트리 관리

APPLICATION	감시 경로	NAMESPACE	SYNC-WAVE	역할
root-app	argocd/apps/	argocd	—	하위 Application 3개를 선언적으로 관리하는 루트
kafka	k8s/kafka/	kafka	1	메시지 브로커 — 앱보다 먼저 기동되어야 함
notiflex-smb	k8s/smb/	notiflex	2	기본 테넌트 API Rollout · Service · Gateway
notiflex-enterprise	k8s/enterprise/	enterprise	2	2번째 테넌트 — 동일 이미지 재사용, 환경만 분리

sync-wave로 기동 순서를 제어해, 의존 컴포넌트(Kafka)가 준비되기 전에 애플리케이션이 올라가 실패하는 상황을 방지했습니다.

04 — DEPLOYMENT

배포 전략의 진화

단순 롤링 업데이트에서 시작해, 안전성과 리소스 효율을 저울질하며 두 번의 전략 전환을 거쳤습니다. 모든 전환은 Git 커밋만으로 롤백 가능합니다.

PHASE · 초기

Rolling Update

v0.1.x

- ArgoCD 기본 배포 방식
- Pod을 하나씩 순차 교체
- 추가 리소스 불필요
- 문제 발견이 배포 **이후**

PHASE · 무중단

Blue / Green

v0.2.0

- Argo Rollouts 도입
- active / preview 완전 분리
- 30초 검증 후 auto-promote
- 리소스 **2배** 필요

PHASE · 점진 (현재)

Canary

v0.4.1 ~ v0.6.0

- 트래픽 20→50→80→100%
- 단계별 30초 pause
- 리소스 2배 불필요
- 일부 사용자로 **조기 발견**

전략별 트레이드오프 — 무엇을 얻고 무엇을 내주는가

비교 항목	ROLLING UPDATE	BLUE / GREEN	CANARY (채택)
다운타임	거의 없음 (순차 교체)	**없음** (전환 시점 스위치)	**없음** (비율 전환)
추가 리소스	**불필요**	**2배** (양쪽 동시 기동)	**최소** (일부 Pod만 추가)
문제 조기 발견	**전량 배포 후**	preview 검증 가능	**실사용 트래픽 20%부터**
롤백 속도	재배포 필요 (느림)	**즉시** (트래픽 되돌림)	**즉시** (abort)
영향 범위	점진적이나 통제 불가	전환 시 **100% 즉시**	**단계별 통제**
적합한 상황	내부 도구 · 개발 환경	리소스 여유 + 사전 검증 중시	리소스 제약 + 실사용 검증 필요

Canary를 선택한 이유: Spot VM 소규모 노드 환경에서 Blue/Green의 리소스 2배 요구는 곧 스케줄 실패 위험이었습니다. Canary는 일부 Pod만 추가하면서도 **실사용 트래픽으로** 신규 버전을 검증할 수 있어, 제약 조건과 안전성을 동시에 만족하는 선택이었습니다. 이 판단 근거는 ADR-010에 기록했습니다.

롤백 · 자가 치유 메커니즘

01 Git revert 기반 롤백

배포 문제 발생 시 이미지를 재빌드하지 않습니다. 매니페스트 커밋을 되돌리면 ArgoCD가 이전 이미지 태그를 감지해 자동 재배포합니다.

복구 시간	커밋 1개 revert + auto-sync
감사 추적	롤백 사실도 Git 이력에 남음

02 selfHeal 자가 치유

누군가 클러스터를 직접 수정해도(`kubectl edit` 등) ArgoCD가 Git 상태와의 차이를 감지해 자동으로 되돌립니다. 클러스터 실제 상태와 Git이 항상 일치합니다.

감시 주기	지속적 diff 비교
효과	구성 드리프트 원천 차단

05 — JOURNEY

구축 여정

빈 클러스터에서 이벤트 기반 플랫폼까지, 각 단계가 다음 단계의 전제가 되도록 7개 국면으로 점진적으로 쌓아 올렸습니다.

PHASE 1 · 기반

클러스터 · 컨테이너 · 첫 배포

GKE Standard(Spot) 클러스터를 생성하고 Go API를 `scratch` 이미지로 빌드해 첫 배포를 수행. Gateway API를 사전 활성화해 이후 단계의 재작업을 방지하고, kubectl 컨텍스트 안전 규칙을 확립해 잘못된 클러스터 조작 가능성을 차단.

GKE Standard | Artifact Registry | Go / scratch | Cloud Build

PHASE 2 · GITOPS CI/CD

배포를 Git으로 옮기다

ArgoCD로 GitOps를 구축하고 GitHub Actions CI를 연결. 코드 push → 이미지 빌드 → 매니페스트 자동 갱신 → ArgoCD 자동 배포로 이어지는 엔드투엔드 파이프라인을 완성하고, 실제 롤백까지 수행해 검증.

ArgoCD | GitHub Actions | auto-sync | selfHeal

PHASE 3 · 관측 가능성

메트릭 · 로그 · 알림

kube-prometheus-stack과 Loki + Fluent Bit를 설치하고 Notiflex 전용 Grafana 대시보드를 구성. PrometheusRule 알림은 문서상 확인에 그치지 않고 실제 Pod 재시작을 유발해 **firing → resolved** 전 과정을 검증.

Prometheus | Grafana | Loki | Fluent Bit | Alertmanager

PHASE 4 · 무중단 배포

외부 트래픽 · Rollouts

Gateway API로 외부 진입점을 열고, Deployment를 Argo Rollouts로 전환해 Blue/Green 무중단 배포를 도입. 이 시점부터 아키텍처 결정을 ADR로 기록하기 시작해 "왜 그렇게 했는가"를 저장소에 누적.

Gateway API | HTTPRoute | Argo Rollouts | ADR

PHASE 5 · 엔터프라이즈 기반

상태 공유 · 시크릿 · Canary

Valkey로 Pod 간 상태를 공유해 다중 인스턴스 정합성을 확보하고, 시크릿을 **Secret Manager + Workload Identity + CSI** 조합으로 전환해 SA 키 파일을 완전히 제거. 배포 전략을 Canary로 진화.

Valkey | Secret Manager | Workload Identity | Secrets Store CSI | Canary

PHASE 6 · 규모 확장

노드 분리 · App of Apps · 멀티테넌시

역할별 노드풀로 워크로드를 격리해 API와 배치 부하가 서로 간섭하지 않도록 분리하고, App of Apps로 여러 Application을 하나의 루트 선언으로 통합. Namespace 기반 멀티테넌시로 2번째 테넌트를 추가.

멀티 노드풀 | nodeSelector | App of Apps | 멀티테넌시

PHASE 7 · 고도화

이벤트 기반 · 분산 트레이싱 · 배치

Kafka(Strimzi · KRaft)로 비동기 처리를 도입해 요청 경로에서 알림 발송을 분리하고, OpenTelemetry + Tempo로 요청의 전체 여정을 추적. CronJob으로 헬스체크를 자동화하며 관측 3축을 완성.

Kafka | Strimzi | Tempo | OpenTelemetry | CronJob

06 — IMPLEMENTATION

핵심 구현

플랫폼의 성격을 결정한 네 가지 축입니다. 각각은 특정 운영 문제에 대한 직접적인 답으로 도입했습니다.

01 GitOps 배포 파이프라인

Git을 유일한 진실 소스로 삼아, 코드 push 한 번으로 빌드부터 Canary 배포까지 사람 개입 없이 진행됩니다. selfHeal이 클러스터의 수동 변경을 자동으로 되돌려 구성 드리프트를 차단합니다.

트리거	app/ 변경 시 GitHub Actions 자동 실행
이미지	git SHA 기반 태그 — 배포본↔커밋 1:1 추적
동기화	ArgoCD auto-sync + selfHeal
해결한 문제	수동 배포의 추적 불가·재현 불가

02 관측 가능성 3축 통합

메트릭·로그·트레이스를 Grafana 하나로 통합했습니다. `/id` 요청 하나가 Valkey·Kafka 자식 span으로 분해되어, 어느 구간에서 시간이 소모되는지 눈으로 확인할 수 있습니다.

메트릭	Prometheus · ServiceMonitor scrape
로그	Loki · Fluent Bit DaemonSet
트레이스	Tempo · OTLP gRPC · 자식 span 분해
해결한 문제	장애를 사용자 제보로 인지하던 상황

03 Keyless 시크릿 관리

비밀번호를 Git이나 K8s Secret 평문이 아니라 GCP Secret Manager에 두고, Workload Identity로 **SA 키 파일 없이** Pod에 파일로 마운트합니다. 두 테넌트가 동일 GSA를 안전하게 재사용합니다.

저장소	GCP Secret Manager (버전 관리·감사 로그)
인증	Workload Identity — 키 파일 없음
전달	Secrets Store CSI 파일 마운트
해결한 문제	유출·만료 관리가 필요한 SA 키 파일 제거

04 이벤트 기반 비동기 처리

동기 처리로 몰리면 느려지던 요청을 Kafka로 분리했습니다. API가 Producer로 메시지를 발행하고 Consumer가 소비 — Consumer가 죽어도 디스크에 남은 메시지는 유실되지 않습니다.

브로커	Kafka · KRaft 모드 (ZooKeeper 불필요)
운영	Strimzi Operator · CRD 선언 관리
토픽	notifications · 3 partitions
해결한 문제	요청 경로의 동기 처리 병목

확장성 설계 — 노드풀 분리와 멀티테넌시

노드풀	머신 타입	배치 워크로드	분리 목적
`default-pool`	`e2-medium × 2 (Spot)`	ArgoCD · 관측 스택(Prometheus/Grafana/Loki) · Valkey	플랫폼 공통 기반
`api-pool`	`e2-medium × 1 (Spot)`	notiflex API Rollout · enterprise API Rollout	사용자 트래픽 전용 — 배치 부하와 격리
`worker-pool`	`e2-standard-2 × 1 (Spot)`	Kafka(Strimzi) · Tempo · healthcheck CronJob	메모리 집약 백엔드 · 배치

멀티테넌시: `notiflex`(SMB)와 `enterprise` 두 Namespace로 고객 환경을 분리했습니다. 동일 이미지를 재사용하되 replicas·시크릿·서비스를 테넌트별로 독립 선언하고, 각각 별도 KSA에 동일 GSA를 매핑해 시크릿 접근 권한을 재사용하면서도 워크로드 경계는 유지했습니다.

07 — TROUBLESHOOTING

문제 해결 경험

가장 많이 배운 지점들입니다. 특히 **"에러 없이 조용히 실패하는"** 유형의 문제를 종료 코드·재시작 카운트·스키마 검증 같은 객관적 근거로 추적해 해결했습니다.

리소스 고갈로 인한 연쇄 장애

CRITICAL · 연쇄

상황	소규모 노드 환경에서 공유 캐시 Valkey가 **14시간 넘게 Pending** 상태로 방치. 그 여파로 Valkey에 의존하는 API Pod이 **275회 CrashLoopBackOff**를 기록.
원인	Canary 전환과 노드풀 갱신으로 노드 여유 CPU가 줄어든 상태에서 Valkey(50m 요청)가 스케줄에 실패 → 이를 의존하는 애플리케이션까지 도미노로 붕괴. 단일 컴포넌트의 스케줄 실패가 도메인 전체 장애로 확대된 사례.
해결	CPU 요청을 `10m`으로 축소해 스케줄 가능하게 만들고, 이 값을 helm-values 파일로 저장소에 남겨 재현성을 확보. 이후 StatefulSet이 `helm upgrade` 후에도 이전 리비전을 계속 재사용하는 문제를 발견해 `scale 0 → 1`로 강제 재기동해 새 스펙을 적용.

LESSON

공유 백엔드의 스케줄 실패는 단일 앱이 아니라 그 백엔드에 의존하는 전체 도메인을 무너뜨린다. 리소스 예산은 평상시가 아니라 **"최악의 동시 배포 시점"**을 기준으로 잡아야 하며, 장애 대응 후에는 반드시 선언 파일에 값을 남겨 다음 배포에서 되돌아가지 않게 한다.

조용한 실패 — Tempo OOMKilled

WARNING · 은폐된 장애

상황	분산 트레이싱은 "설치 완료" 상태였고 대시보드도 떠 있었지만, Tempo가 **약 30분 주기로 OOMKilled** 되고 있었음. 애플리케이션은 죽지 않아 겉보기엔 아무 문제가 없었음.
원인	메모리 한도 256Mi가 트레이스 블록 flush 시점의 순간 사용량을 감당하지 못함. 애플리케이션의 트레이스 export는 `connection refused`로 조용히 실패했지만 사용자 요청 경로에는 전혀 노출되지 않아 인지가 늦어짐.
해결	메모리 한도를 `512Mi`로 상향. 종료 코드 `137`(OOM)과 재시작 카운트 증가를 근거로 원인을 확정했고, 수정 후 신규 트레이스가 실제로 저장·조회되는지까지 확인해 종결.

LESSON

"설치됨 ≠ 동작함". 관측 스택 자체도 관측 대상이다 — 배포 직후 상태가 `Running`인 것만 보지 말고, 재시작 카운트와 exit code를 반드시 확인해야 은폐된 주기적 장애를 잡아낼 수 있다.

조용한 실패 — Kafka 파티션 누락으로 인한 메시지 유실

WARNING · 데이터 손실

상황	Producer는 정상 동작하고 메시지도 발행되는데, Consumer 로그에는 메시지가 하나도 찍히지 않음. **에러 로그조차 없었음.**
원인	Consumer가 파티션 0만 구독하도록 작성되어 있었음. Producer는 메시지 Key 없이 발행했고, 기본 파티셔너가 3개 파티션에 무작위로 분산 → 약 2/3의 메시지가 구독되지 않는 파티션으로 흘러가 조용히 사라짐.
해결	`consumer.Partitions()`로 토픽의 전체 파티션 목록을 조회해 각 파티션마다 구독을 생성하도록 수정. 이후 발행한 모든 메시지가 누락 없이 소비되는 것을 확인.

LESSON

분산 시스템의 기본 동작 원리(파티셔닝·키 기반 라우팅)를 이해하지 못하면 **"에러 없는 데이터 손실"**을 만난다. 에러가 없다는 것이 정상 동작의 증거가 될 수 없으며, 발행 건수와 소비 건수를 대조하는 검증이 필요하다.

CRD 스키마의 함정 — Strimzi nodeSelector

INFO · 스키마 검증

상황	Kafka Pod을 특정 노드풀에 배치하려고 `nodeSelector`를 지정했는데 전혀 적용되지 않고, ArgoCD는 계속 `OutOfSync` 상태를 반복.
원인	Strimzi CRD의 `template.pod` 구조적 스키마에는 `nodeSelector` 필드 자체가 존재하지 않음 → API 서버가 **에러 없이 조용히 제거**. 그래서 Git의 매니페스트와 클러스터 실제 상태가 영원히 어긋나며 OutOfSync가 해소되지 않았음. 이 Operator는 노드 배치를 `affinity`로만 지원.
해결	`kubectl explain`으로 CRD 스키마를 직접 확인한 뒤 `affinity.nodeAffinity`로 교체. 적용 후 Kafka Pod이 의도한 노드풀에 배치되고 Sync 상태도 정상화.

LESSON

Operator가 정의한 CRD는 쿠버네티스 표준 리소스와 필드 구성이 다르다. "표준에 있으니 당연히 여기도 있겠지"를 가정하지 말고 스키마로 검증하는 습관이 필요하다. 또한 **지속되는 OutOfSync는 그 자체로 진단 신호**다 — 매니페스트가 클러스터에 온전히 반영되지 않았다는 뜻이다.

제약 속 최적화 — 소규모 노드에 전체 스택 수용

WIN · 사전 대응

상황	e2-medium 소규모 노드 위에 ArgoCD · 관측 스택 · Valkey · CSI 드라이버를 모두 올려 CPU 사용률이 **95% 이상**에 도달. 여기에 신규 컴포넌트를 추가하면 Pending이 확실시되는 상황.
대응	Secret Manager CSI(노드당 약 120m 소요) 활성화 **전에** 선제적으로 리소스를 회수 — Prometheus/Grafana/Alertmanager 요청을 `100m → 5m`, Loki/Fluent Bit를 `10m → 1m`로 축소하고, 모든 변경을 helm-values 파일로 저장소에 반영.
결과	CSI 드라이버 전환과 Workload Identity 활성화가 Pending·재시도 **없이 한 번에** 성공. 앞선 Valkey 연쇄 장애를 겪은 뒤 얻은 교훈을 실제로 적용해 동일 유형의 장애를 예방한 사례.

LESSON

리소스 예산을 사후 대응이 아니라 **사전 확보**의 문제로 다루면 연쇄 장애를 예방할 수 있다. 장애 대응 경험이 다음 작업의 체크리스트로 전환될 때 비로소 학습이 완료된다.

문제 해결에서 얻은 운영 원칙

원칙	내용	적용 사례
Running ≠ Healthy	Pod 상태가 Running이어도 주기적으로 죽고 있을 수 있다. 재시작 카운트와 exit code를 함께 본다.	Tempo OOMKilled
에러 없음 ≠ 정상	가장 위험한 실패는 로그를 남기지 않는다. 입력과 출력의 건수를 대조해 검증한다.	Kafka 파티션 누락
OutOfSync는 진단 신호	해소되지 않는 Sync 불일치는 매니페스트가 클러스터에 온전히 반영되지 않았다는 증거다.	Strimzi CRD 스키마
임시 조치는 선언으로	장애 대응으로 바꾼 값은 반드시 helm-values·매니페스트에 반영해 다음 배포에서 되돌아가지 않게 한다.	Valkey CPU 축소
예산은 최악 기준으로	리소스는 평상시가 아니라 동시 배포가 겹치는 최악의 순간을 기준으로 계획한다.	CSI 전환 사전 확보

08 — ADR

아키텍처 의사결정 기록

"무엇을 왜 선택했고 무엇을 쓰지 않았는가"를 16개의 ADR로 저장소에 누적했습니다. 다른 사람도, 미래의 나도 같은 근거를 추적할 수 있습니다.

ID	결정	대신 쓰지 않은 것	영역
ADR-001	GitOps 도구로 **ArgoCD** 채택	Flux · Jenkins X · Spinnaker	GitOps
ADR-002	CI로 **GitHub Actions** 채택	Cloud Build · GitLab CI · Jenkins	CI
ADR-003	메트릭에 **Prometheus + Grafana**	Datadog · Cloud Monitoring	관측
ADR-004	로그에 **Loki + Fluent Bit**	ELK Stack · Cloud Logging	관측
ADR-005	알림에 **PrometheusRule + Alertmanager**	Grafana Alerting	관측
ADR-006	외부 트래픽에 **Gateway API**	Ingress NGINX · Istio	트래픽
ADR-007	무중단 배포에 **Argo Rollouts** (Blue/Green)	Flagger · Istio 카나리	배포
ADR-008	캐시에 **Valkey**	Redis(SSPL 라이선스) · Memcached	데이터
ADR-009	시크릿에 **Secret Manager CSI + Workload Identity**	Sealed Secrets · SA 키 파일	보안
ADR-010	배포 전략을 **Canary**로 전환	Blue/Green 유지	배포
ADR-011	워크로드 배치에 **nodeSelector + 전용 노드풀**	taint/toleration · nodeAffinity 단독	확장
ADR-012	다중 앱 관리에 **App of Apps**	ApplicationSet · 개별 수동 관리	확장
ADR-013	멀티테넌시에 **Namespace 분리**	vCluster · 클러스터 완전 분리	확장
ADR-014	메시징에 **Kafka (Strimzi · KRaft)**	RabbitMQ · NATS · Pub/Sub	고도화
ADR-015	트레이싱에 **Grafana Tempo**	Jaeger · Zipkin	고도화
ADR-016	배치 작업에 **Kubernetes CronJob**	외부 cron · Argo Workflows	고도화

대표 결정의 판단 근거

ADR-008 Valkey

Redis가 SSPL 라이선스로 전환되면서 상업적 사용에 제약이 생겼습니다. Valkey는 Redis 포크로 API가 완전히 호환되면서 **BSD 라이선스**를 유지해, 코드 변경 없이 라이선스 리스크를 제거할 수 있었습니다.

ADR-009 Workload Identity

SA 키 파일은 유출 시 무기한 유효하고 만료·회전 관리 부담이 큽니다. Workload Identity는 KSA↔GSA 매핑으로 **키 파일 자체를 제거**하며, CSI로 시크릿을 파일 마운트해 애플리케이션 코드도 단순해졌습니다.

09 — METHOD

프로젝트 방법론 · GitAIOps

전 과정을 **Git 중심 + AI 페어링 + 운영 자동화**가 결합된 GitAIOps 워크플로로 수행했습니다. 모든 도구 선택은 "탐색 → 비교 → 결정" 3단계를 거치고, 위험 작업은 가드레일로 통제하며, 결정은 ADR로, 진행 상황과 트러블슈팅은 JOURNEY 기록으로 남겨 **재현 가능하고 감사 가능한** 방식으로 진행했습니다.

Git

단일 진실 소스

모든 변경은 커밋으로만 이뤄집니다. 클러스터는 Git을 자동 추종하고, 롤백은 revert 하나로 끝납니다.

AI

가드레일 페어링

반복·위험 작업을 AI와 페어링하되, 실행 전 상태 확인과 승인 규칙으로 통제해 사고를 예방합니다.

Ops

3층 문서 체계

규칙(CLAUDE.md) · 현재 그림(claude-context) · 결정 누적(ADR)으로 컨텍스트를 분리해 관리합니다.

10 — COMPETENCY

검증된 역량

GitOps · CD 파이프라인 설계

ArgoCD App of Apps와 Argo Rollouts로 선언적·자가치유 배포를 엔드투엔드로 구축하고 롤백까지 검증.

관측 가능성 엔지니어링

메트릭·로그·트레이스 3축을 Grafana로 통합하고, 알림을 실제 장애 상황으로 유발해 검증.

장애 대응 · 근본 원인 분석

연쇄 Pending, OOMKilled, 무손실로 보이던 데이터 유실을 exit code·스키마·리소스 근거로 추적·해결.

클라우드 보안 · IAM

Workload Identity 기반 keyless 인증과 Secret Manager 연동으로 SA 키 파일을 완전히 제거.

리소스 최적화 · 비용 효율

Spot VM 소규모 노드 제약 위에서 관측·데이터·메시징 전체 스택을 사전 예산 관리로 운영.

확장성 · 멀티테넌시

역할별 노드풀 분리와 Namespace 테넌시로 워크로드를 격리하고 2번째 테넌트를 안전하게 추가.

알려진 한계와 다음 단계 — 완결이 아니라 진행 중인 시스템으로서

현재 한계

- 단일 존(Zonal) 클러스터 — 존 장애 시 전체 중단, 프로덕션은 Regional 필요
- Kafka 단일 브로커 · replicas 1 — 복제 없음
- 테넌트 간 캐시 키가 공유되어 데이터 수준 격리는 미적용
- 관측 스택 일부가 아직 GitOps 밖(Helm CLI 관리)

다음 단계

- Regional 클러스터 + Kafka 다중 브로커로 가용성 확보
- 테넌트별 캐시 키 네임스페이싱으로 데이터 격리 완성
- Helm 관리 컴포넌트를 App of Apps로 편입해 GitOps 일원화
- 메트릭 기반 자동 Canary 분석(AnalysisTemplate)으로 승격 판단 자동화

Notiflex Platform

GKE 위 클라우드 네이티브 운영 환경 구축 · 빈 클러스터에서 이벤트 기반 플랫폼까지의 엔드투엔드 기록. 전체 매니페스트 · ADR · 트러블슈팅 이력은 저장소에서 확인하실 수 있습니다.

github.com/easydong02/notiflex-platform
sdh-portpolio.com