2025 - 2 오픈소스소프트웨어 프로젝트

DevOps
CI/CD 파이프라인 자동화 구축

2조

20211241 권형미 20221154 나영은 20221188 김가윤 20230029 이은우 20231134 김지희

CONTENTS

01.

E-ON
서비스 소개

E-ON 서비스 소개

E-ON

초중등 학교 밖 청소년을 위한
학업 지속 및 사회성 발달 지원 서비스

E-ON 서비스 소개

주요 기능

학사 일정

학교별 학사 일정
평균 학사 일정

챌린지

비교과 활동
개설, 신청, 출석, 후기

활동 추천

개인별 맞춤 추천
시기별 맞춤 추천

커뮤니티

주제별 게시판

02.

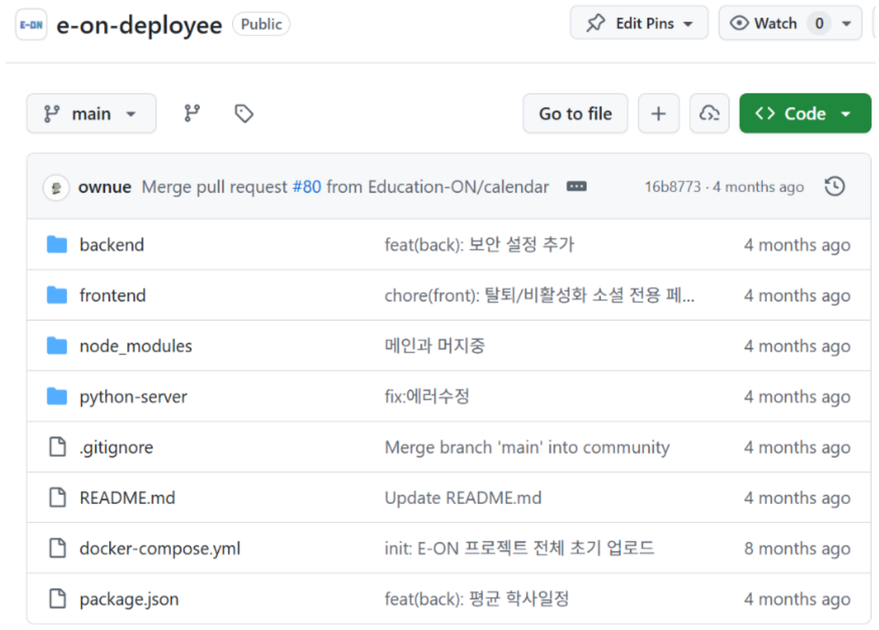
E-ON
개발 및 협업 방식

개발 및 협업 방식

깃허브 활용

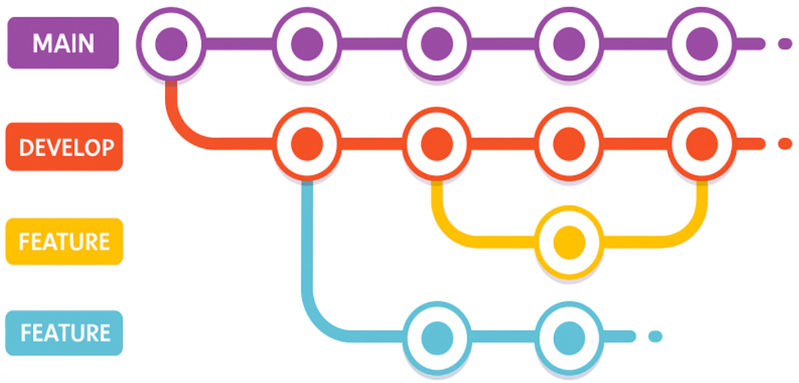
GitHub

GITHUB ORGANIZATION



개발 및 협업 방식

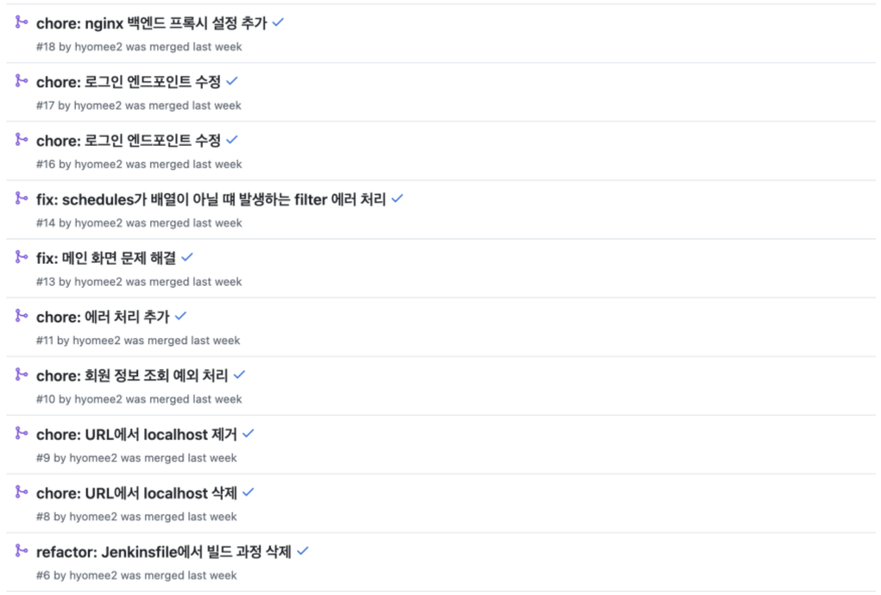
브랜치 활용



- MAIN 브랜치: 배포 가능한 안정 버전
- DEVELOP 브랜치: 기능 통합, 테스트 진행
- FEATURE 브랜치: 기능 단위 개발
- Merge Flow: Feature → Develop → Main

개발 및 협업 방식

이슈



- ISSUE로 작업 관리
- 담당자 지정 & 브랜치 연결
- 이슈에 라벨 부여 (feat, fix 등)

개발 및 협업 방식

Good Commit

feat: GKE Ingress 설정 파일 추가

fix: Jenkins 보안 경고 해결 및 파싱 문제 해결

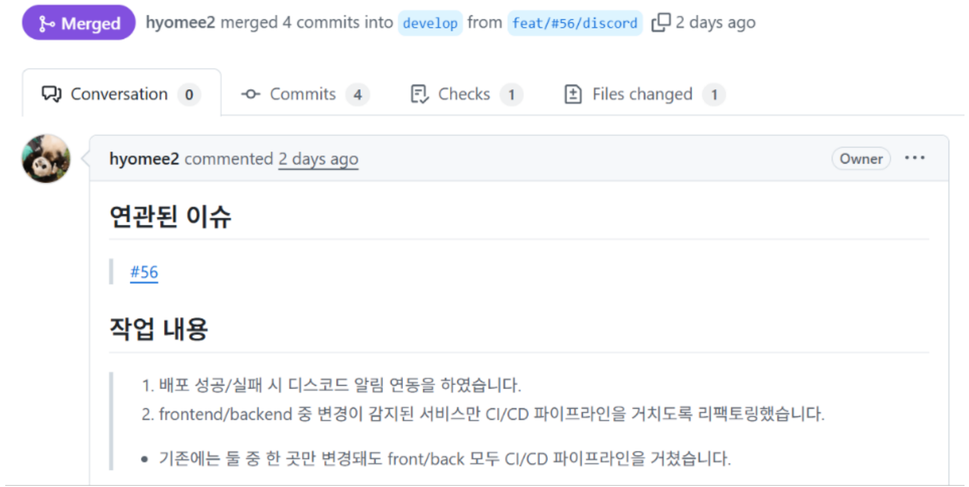
chore: 불필요한 줄바꿈 제거

feat	새로운 기능 추가
fix	버그 수정
chore	단순 코드 수정
docs	문서 수정
style	스타일 관련 수정
refactor	코드 리팩토링

- 커밋 메세지 컨벤션
- 태그 붙이기 -> 의도 파악 가능
- 변경 내용 기재

개발 및 협업 방식

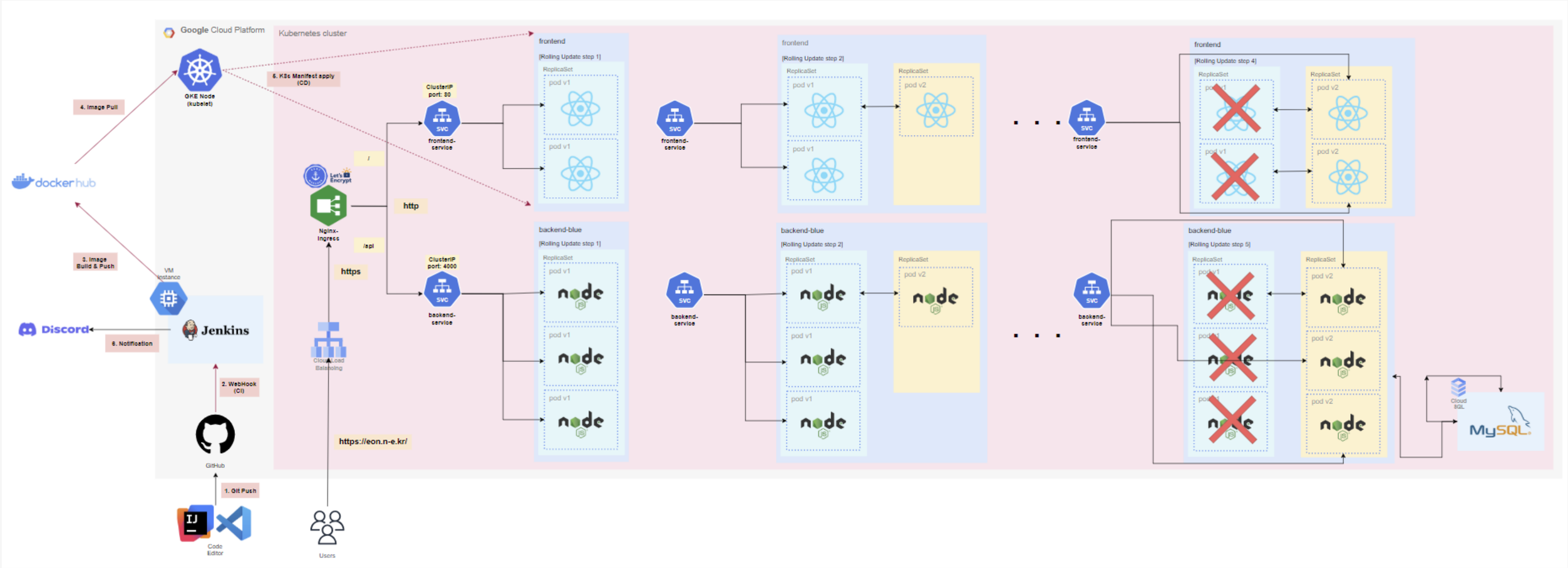
PR



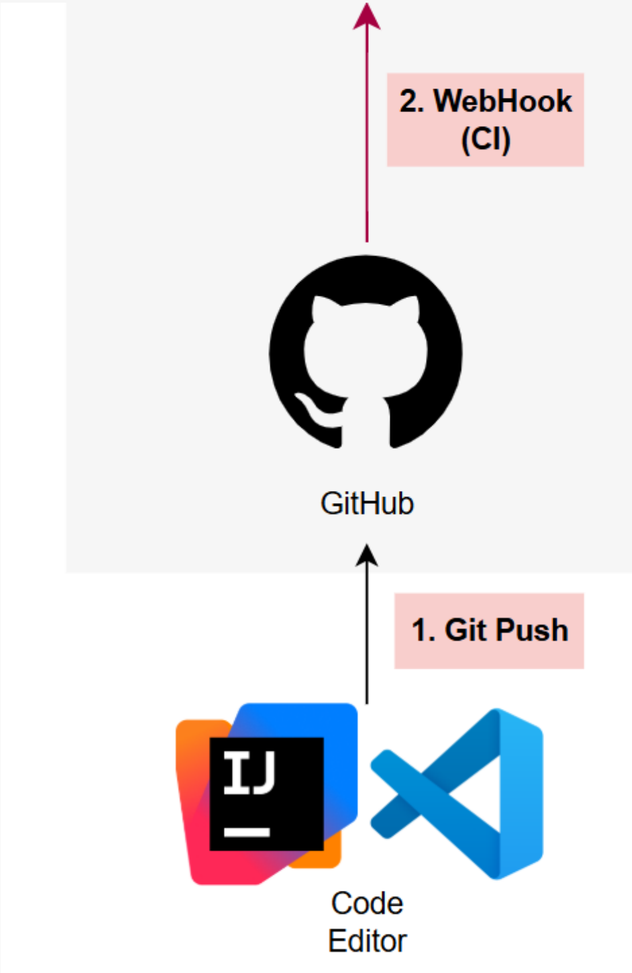
- 기능 완료 후 PR 생성
- 연결 이슈, 구현 내용 명시
- merge 진행

03.

ROLLING-UPDATE 기반
CI/CD 파이프라인 구축

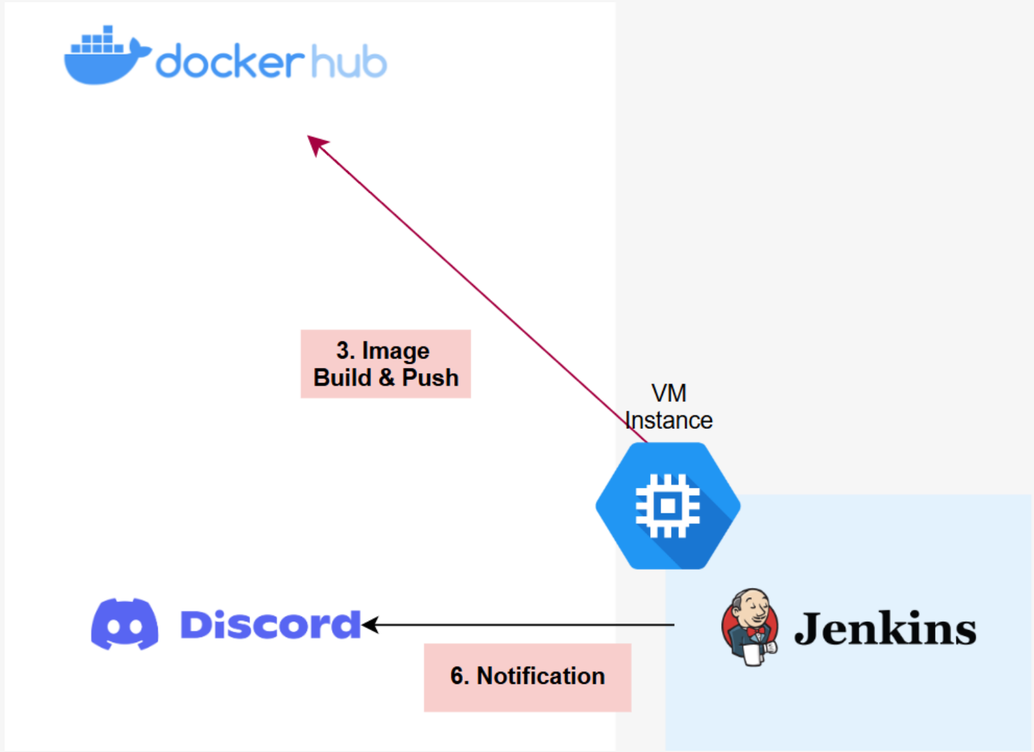

Google Cloud Platform
Kubernetes cluster
GKE Node (kubelet)
5. K8s Manifest apply (CD)
4. Image Pull
dockerhub
3. Image Build & Push
VM Instance
Jenkins
Discord
6. Notification
2. WebHook (CI)
GitHub
1. Git Push
Code Editor
Users
Let's Encrypt
Nginx-Ingress
Cloud Load Balancing
/
/api
http
https
https://eon.n-e.kr/
ClusterIP port: 80
frontend-service
ClusterIP port: 4000
backend-service
frontend
[Rolling Update step 1]
[Rolling Update step 2]
[Rolling Update step 4]
backend-blue
[Rolling Update step 5]
ReplicaSet
pod v1
pod v2
node
Cloud SQL
MySQL

CI 파이프라인 구축



파일을 수정 후 깃허브에 푸시하면
GitHub의 Webhook이 Jenkins에게 전달

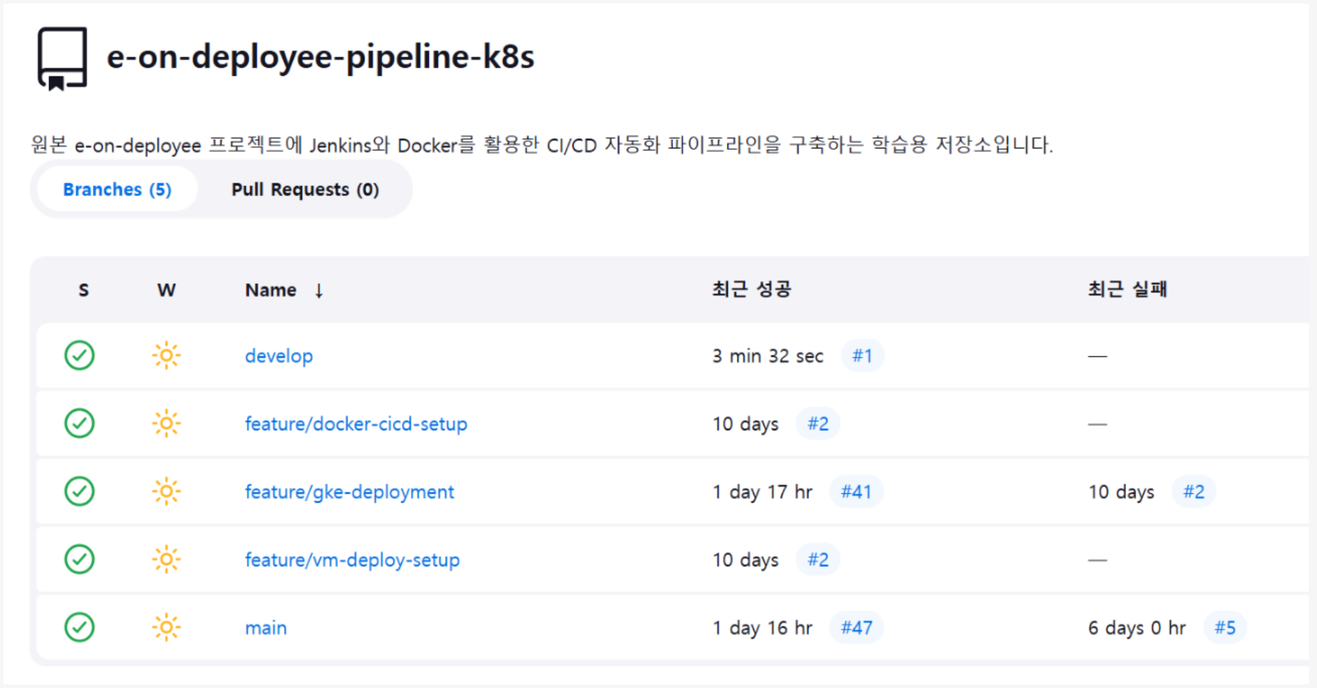
CI 파이프라인 구축



Jenkins는 Multibranch Pipeline 방식으로
브랜치에 맞는 Jenkinsfile을 읽고 빌드를 시작

CI 파이프라인 구축

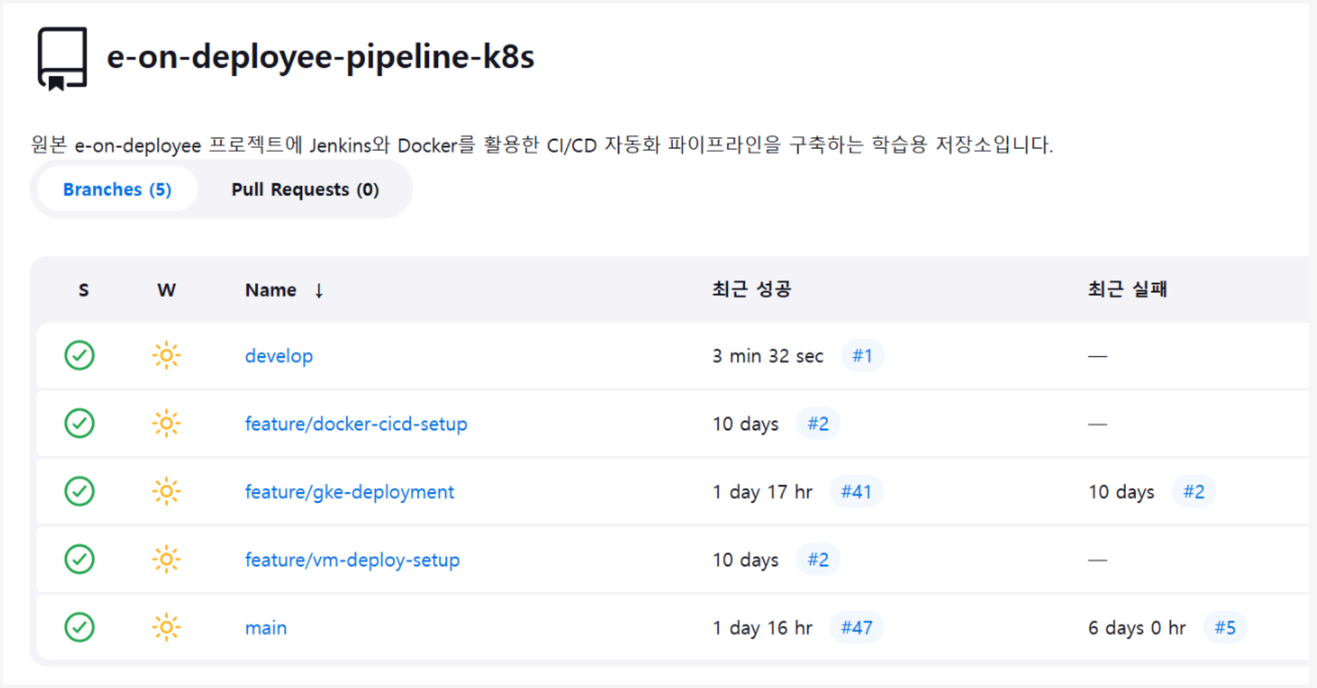
```groovy
stage('Deploy to GKE') {
    when {
        branch 'main'   // main 에서만 배포
    }
    steps {
        echo "=== Deploy to GKE (Rolling Update) ==="

        step([
            $class: 'KubernetesEngineBuilder',
            projectId:         env.PROJECT_ID,
            clusterName:       env.CLUSTER_NAME,
            location:          env.LOCATION,
            manifestPattern:   'k8s/*.yaml',   // 배포 yaml 경로
            credentialsId:     env.CREDENTIALS_ID,
            verifyDeployments: true
        ])
    }
}
```



Multibranch Pipeline 방식 → **브랜치마다 자동으로 파이프라인 생성 및 각 브랜치의 Jenkinsfile을 기준 독립적인 CI 수행**

CI 파이프라인 구축

```
stage('Deploy to GKE') {
    when {
        branch 'main'    // main 에서만 배포
    }
    steps {
        echo "=== Deploy to GKE (Rolling Update) ==="

        step([
            $class: 'KubernetesEngineBuilder',
            projectId:         env.PROJECT_ID,
            clusterName:       env.CLUSTER_NAME,
            location:          env.LOCATION,
            manifestPattern:   'k8s/*.yaml',   // 배포 yaml 경로
            credentialsId:     env.CREDENTIALS_ID,
            verifyDeployments: true
        ])
    }
}
```



Multibranch Pipeline 방식 → **develop 브랜치에서는 빌드만 하고,**
main 브랜치에서는 CI → CD까지 이어지도록 분리

CI 파이프라인 구축 - Jenkinsfile

기존 방식

```
stages {
    stage('Checkout') {
        steps {
            checkout scm
        }
    }

    stage('Build Backend') {
        steps {
            sh """
              echo "=== Build backend image ==="
              docker build -t ${BE_IMAGE_NAME}:latest -f backend/Dockerfile ./backend
            """
        }
    }

    stage('Build Frontend') {
        steps {
            sh """
              echo "=== Build frontend image ==="
              docker build --build-arg VITE_API_URL=${VITE_API_URL} \
                -t ${FE_IMAGE_NAME}:latest -f frontend/Dockerfile ./frontend
            """
        }
    }
```

프론트와 백엔드 이미지를 매번 동시에 빌드

CI 파이프라인 구축 - Jenkinsfile

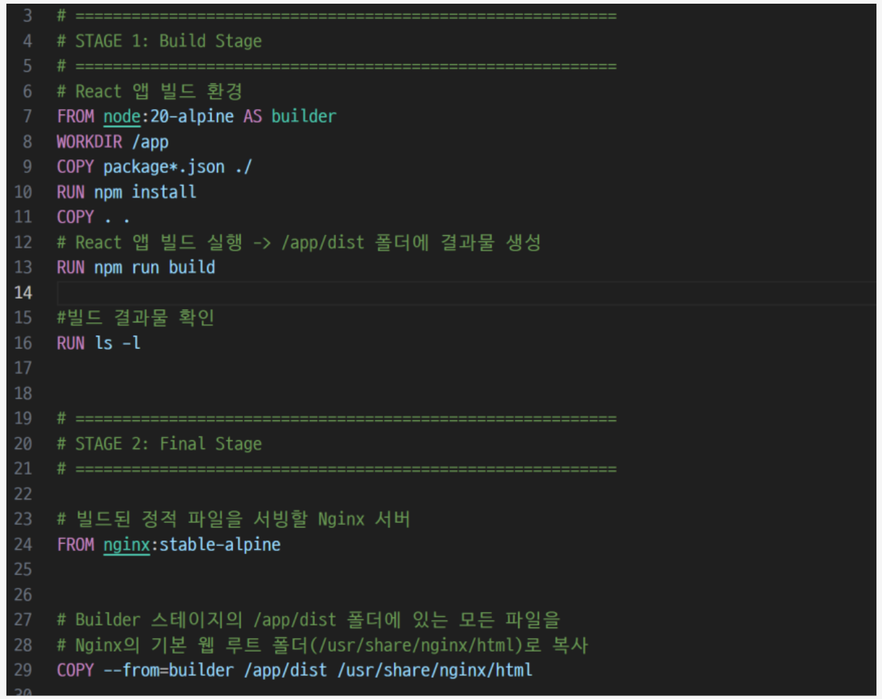
변경된 방식

```
stages {
    /* 1. 코드 체크아웃 */
    stage('Checkout') {
        steps {
            checkout scm
            echo "BRANCH_NAME = ${env.BRANCH_NAME}"

            script {
                // 변경된 파일 목록 가져오기 (프론트/백 중 변경된 서비스만 빌드/배포)
                def changed = sh(
                    script: "git diff --name-only HEAD~1 HEAD || true",
                    returnStdout: true
                ).trim()

                env.FRONT_CHANGED = changed.contains("frontend/") ? "true" : "false"
                env.BACK_CHANGED  = changed.contains("backend/") ? "true" : "false"
            }
        }
    }
```

Git diff 기반으로 변경된 서비스만
선택적으로 빌드하고 push

CI 파이프라인 구축 - Frontend

frontend/Dockerfile

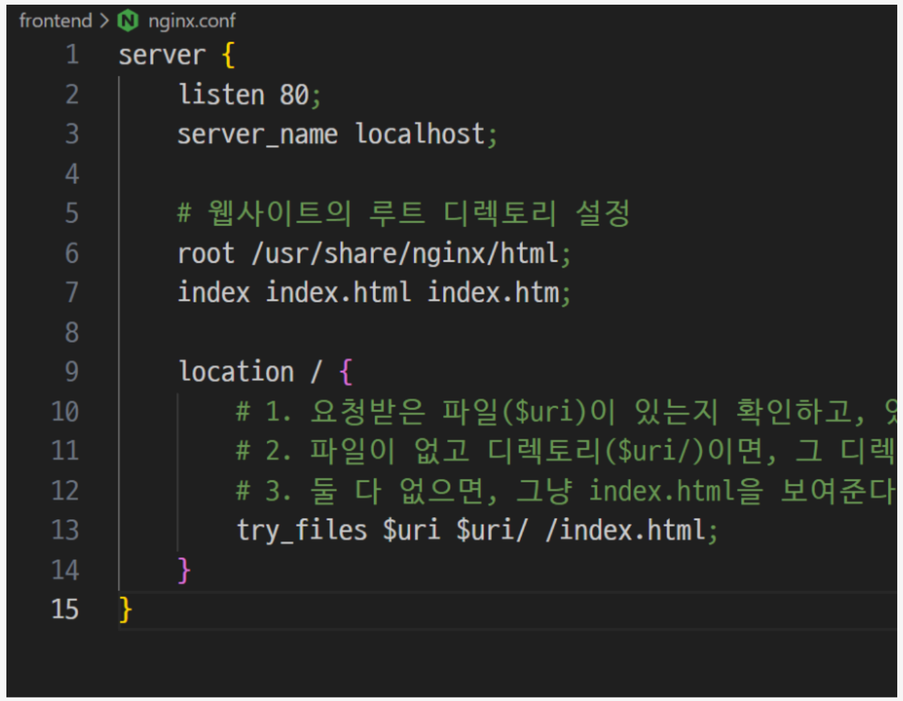


```
# ============================================================
# STAGE 1: Build Stage
# ============================================================
# React 앱 빌드 환경
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
# React 앱 빌드 실행 -> /app/dist 폴더에 결과물 생성
RUN npm run build

#빌드 결과물 확인
RUN ls -l


# ============================================================
# STAGE 2: Final Stage
# ============================================================

# 빌드된 정적 파일을 서빙할 Nginx 서버
FROM nginx:stable-alpine


# Builder 스테이지의 /app/dist 폴더에 있는 모든 파일을
# Nginx의 기본 웹 루트 폴더(/usr/share/nginx/html)로 복사
COPY --from=builder /app/dist /usr/share/nginx/html
```

React 앱을 정적으로 빌드 + Nginx로 서빙

CI 파이프라인 구축 - Frontend

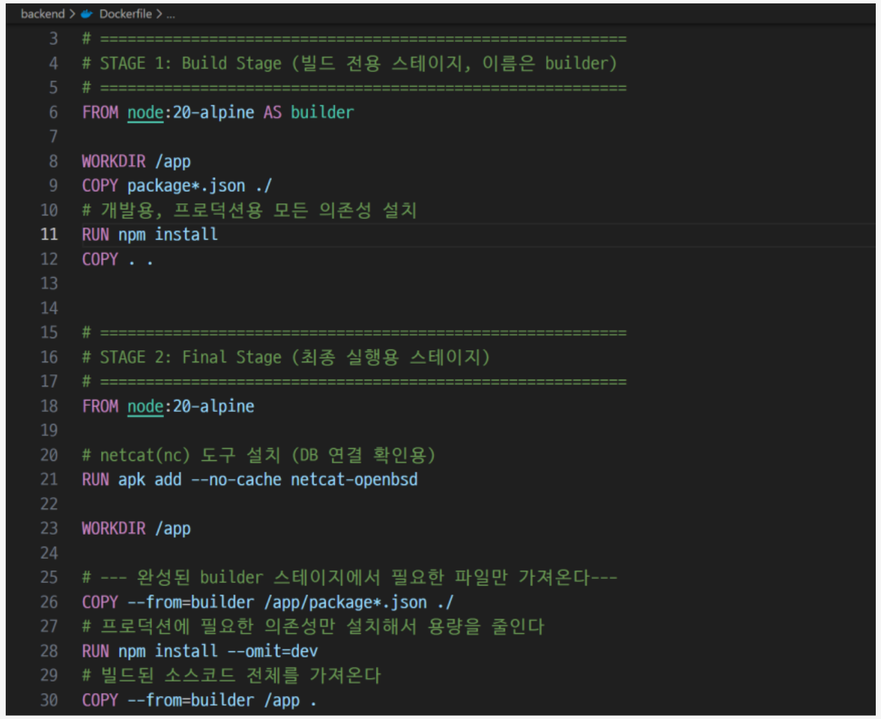
frontend/nginx.conf



try_files 설정을 통해 SPA 라우팅 안정적으로 동작

CI 파이프라인 구축 - Backend

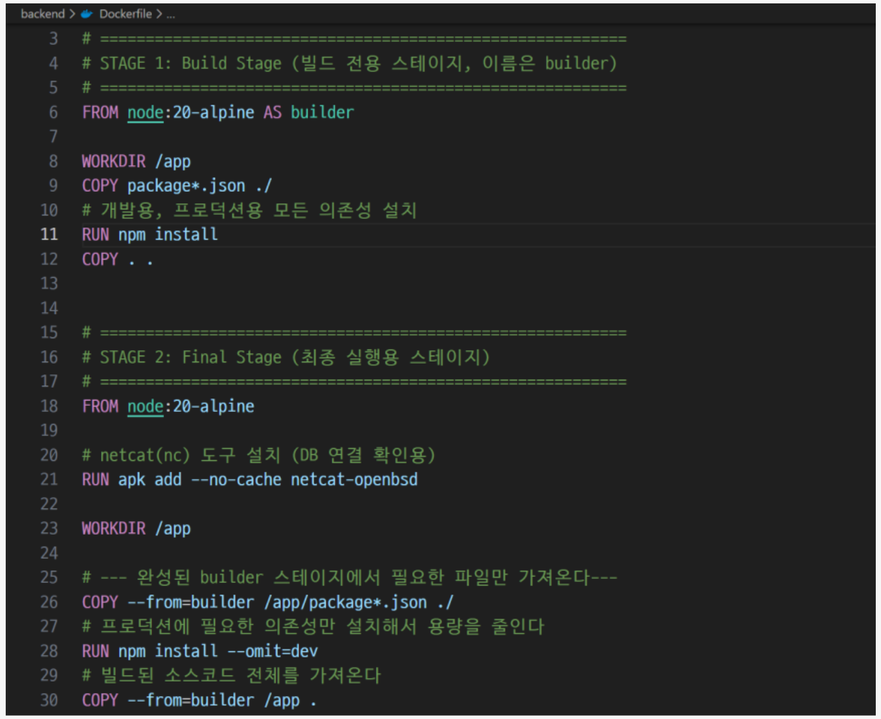
backend/Dockerfile



Node 실행 환경으로 이미지 생성 및
entrypoint 스크립트 가장 먼저 실행

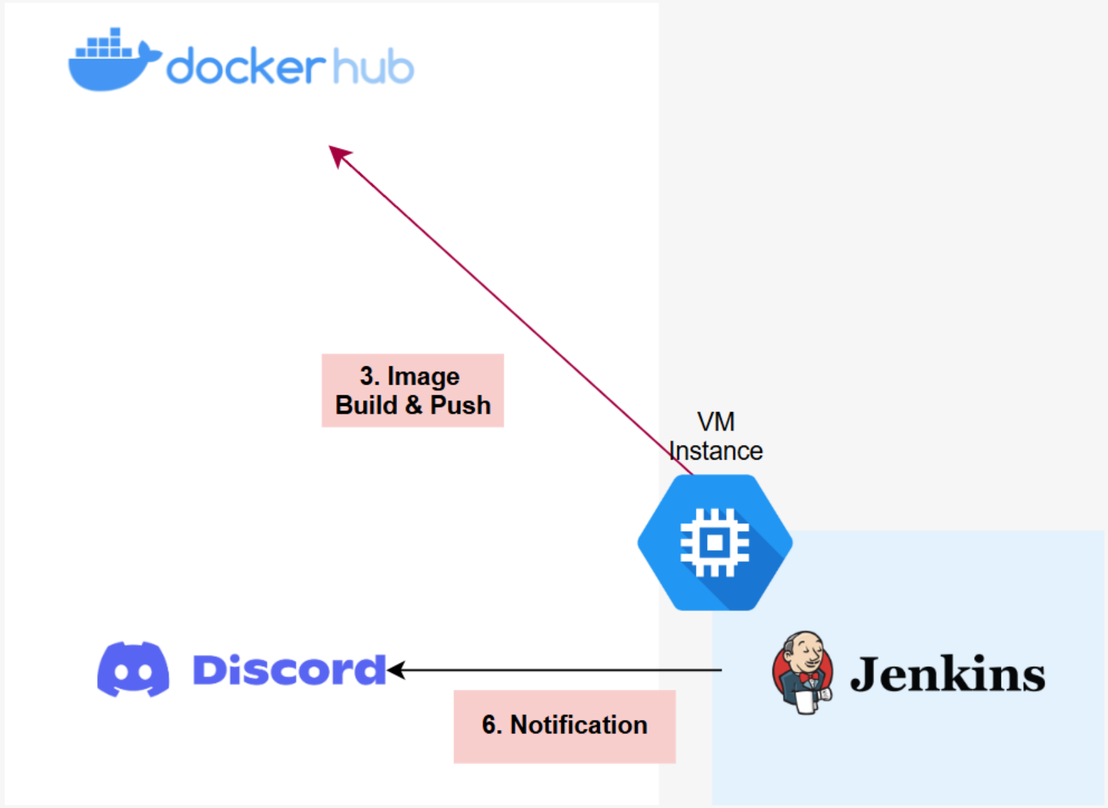
CI 파이프라인 구축 - Backend

backend/entrypoint.sh



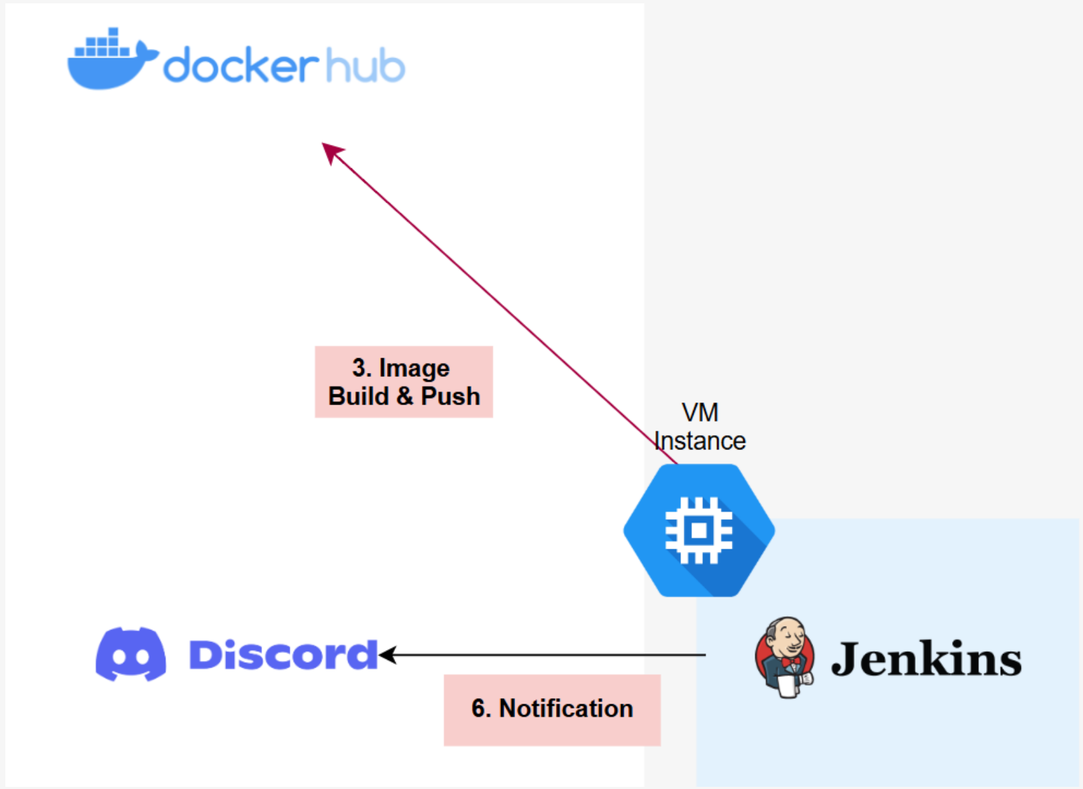
DB가 준비될 때까지 대기 및
Sequelize 마이그레이션과 Seeder를 자동으로 적용한 후 서버 시작

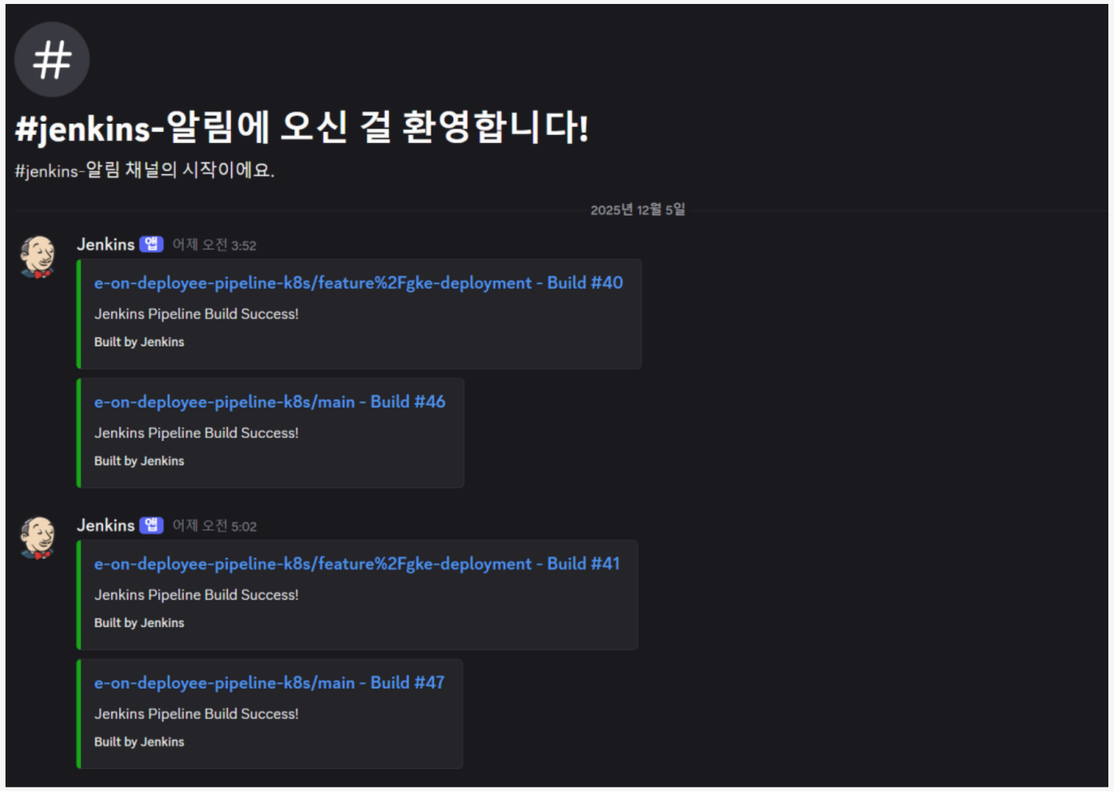
CI 파이프라인 구축



이미지 빌드가 완료되면
Docker Hub로 푸시

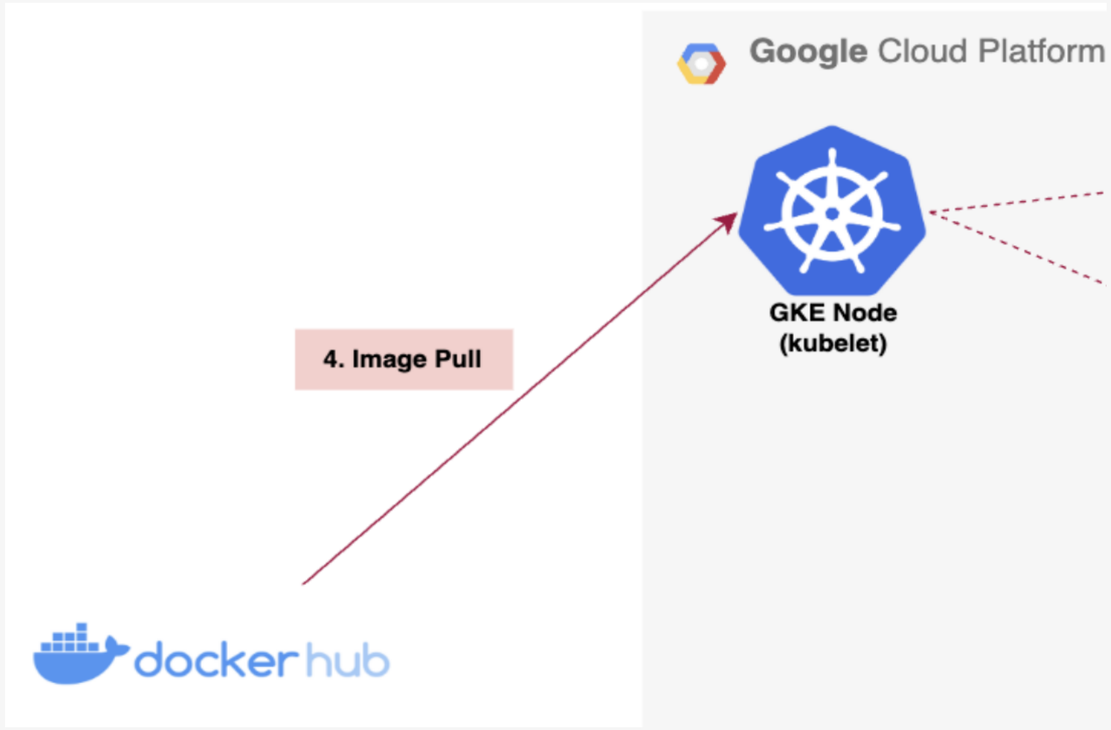
CI 파이프라인 구축 - Discord Webhook





빌드 혹은 배포 시에 Discord Webhook을 이용해 알림이 옴

CD 파이프라인

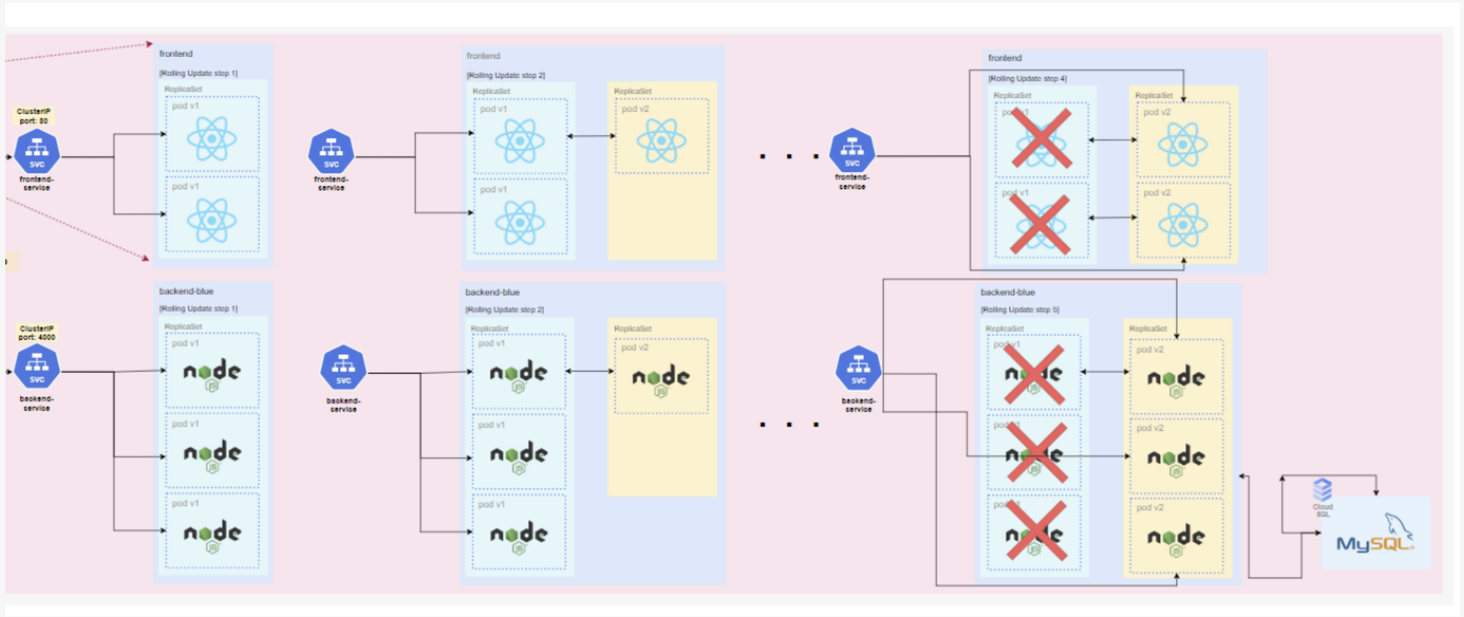


CI 과정에서 만들어진 이미지를 활용해
쿠버네티스에서 롤링 업데이트 방식의 CD 진행

CD 파이프라인

RollingUpdate란?

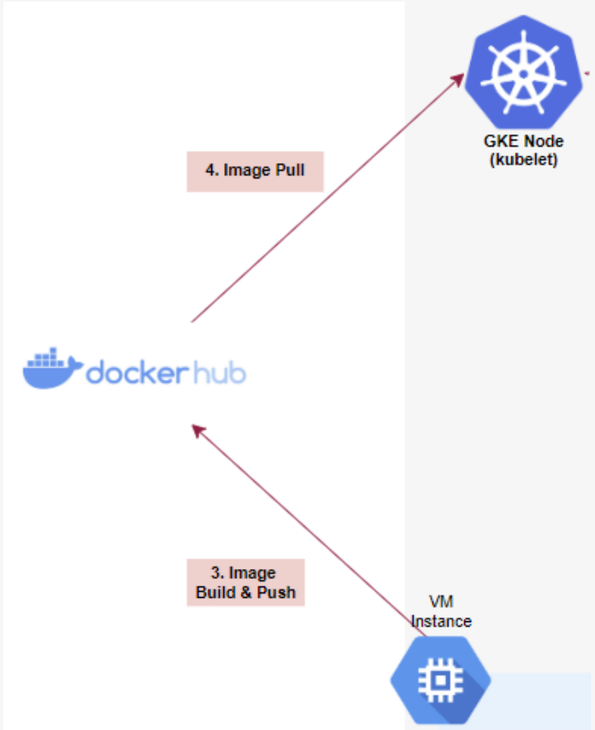
- Kubernetes의 기본 배포 전략
- Pod를 순차적으로 교체하여 서비스 중단 없이 업데이트
- 점진적 배포로 안정성 확보

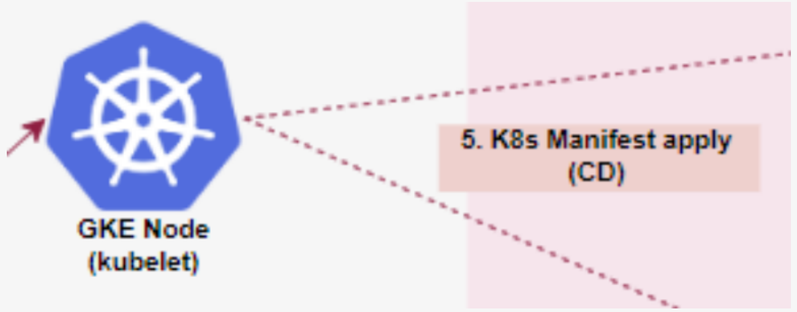


CD 파이프라인

배포 과정

1. Jenkins CI 완료 (이미지 빌드 & Push)
2. kubectl apply -f k8s/ (CD 시작)
3. Kubernetes 가 새 버전 배포
4. 사용자 서비스 중단 없이 업데이트 완료





CD 파이프라인 - Kubernetes 배포 방식

기본값의 한계

기본값(25%)으로는 배포 중 가용 Pod 수 예측 한계

```
kimjh0937@cloudshell:~ (open-source
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
```

개선 시도 - 명시적으로 설정

- maxSurge: 1 → 업데이트 중 최대 4개 Pod까지 동시 실행 가능
- maxUnavailable: 1 → 최소 2개 Pod는 항상 서비스 가능 상태 유지
- 배포 과정에서 일정한 가용 Pod 수 보장

```yaml
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
```

CD 파이프라인 - Jenkins CD 스테이지 실행 과정

Jenkins CD 스테이지 실행 과정

Jenkins CD 스테이지가 실행되면 새로운 이미지 태그로 Deployment 리소스 업데이트

readinessProbe를 설정하여 새로운 파드가 트래픽을 받을 준비가 완벽히 되었을 때만 기존 파드를 종료하도록 구성

```yaml
readinessProbe:
  httpGet:
    path: /health
    port: 4000
  initialDelaySeconds: 3
  periodSeconds: 5
  timeoutSeconds: 2
  failureThreshold: 3
```

CD 파이프라인 - Jenkins CD 스테이지 실행 과정

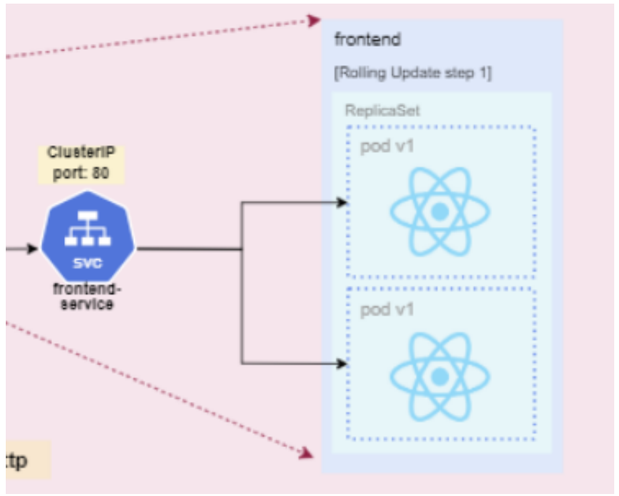
새 버전의 파드가 Running 상태가 된 후에야 기존 파드가 Terminating 되는 것을 확인할 수 있습니다.

```
kimjh0937@cloudshell:~ (open-source-gcp)$ kubectl get pods -w
NAME                         READY   STATUS    RESTARTS   AGE
backend-587599c77f-8c47q     1/1     Running   0          30h
backend-587599c77f-95c4v     1/1     Running   0          30h
backend-587599c77f-m4ckz     1/1     Running   0          30h
frontend-6f98884df7-87xsp    1/1     Running   0          30h
frontend-6f98884df7-gk2pv    1/1     Running   0          30h
mysql-997c5499d-9xqz8        1/1     Running   0          30h
^[[Dbackend-57bb7864b8-btnvr     0/1     Pending   0          0s
backend-57bb7864b8-btnvr     0/1     Pending   0          0s
backend-57bb7864b8-btnvr     0/1     ContainerCreating   0          0s
backend-57bb7864b8-btnvr     1/1     Running             0          23s
backend-587599c77f-m4ckz     1/1     Terminating         0          30h
backend-57bb7864b8-khljl     0/1     Pending             0          0s
backend-57bb7864b8-khljl     0/1     Pending             0          0s
backend-57bb7864b8-khljl     0/1     ContainerCreating   0          0s
backend-57bb7864b8-khljl     1/1     Running             0          10s
backend-587599c77f-95c4v     1/1     Terminating         0          30h
backend-57bb7864b8-zxz5j     0/1     Pending             0          1s
backend-57bb7864b8-zxz5j     0/1     Pending             0          1s
backend-57bb7864b8-zxz5j     0/1     ContainerCreating   0          1s
backend-57bb7864b8-zxz5j     1/1     Running             0          12s
backend-587599c77f-8c47q     1/1     Terminating         0          30h
```

CD 파이프라인 - 배포 단계별 동작

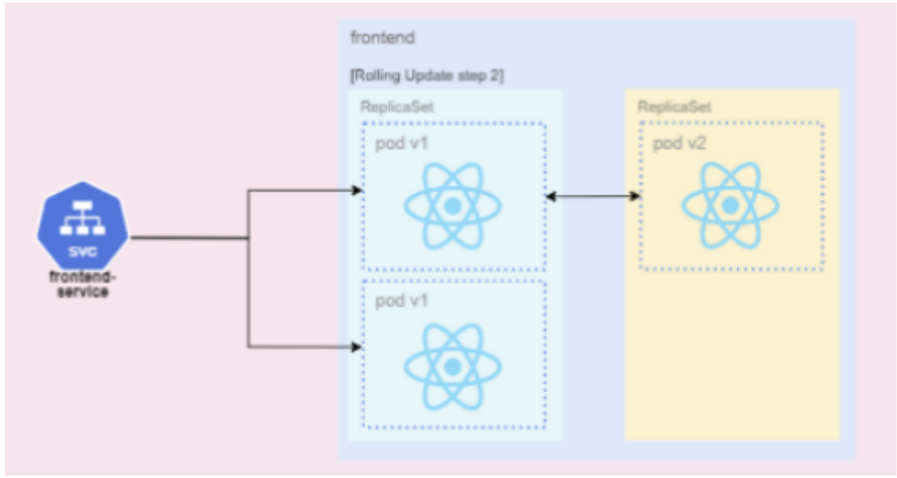
1 초기 상태

- 2개의 v1 Pod 실행 중



2 새 Pod 생성

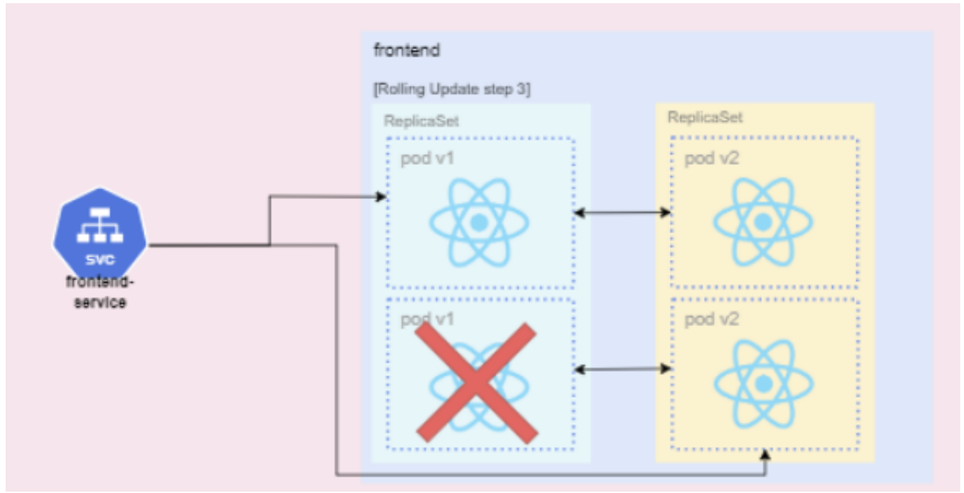
- maxSurge: 1 → 총 4개 Pod 실행



CD 파이프라인 - 배포 단계별 동작

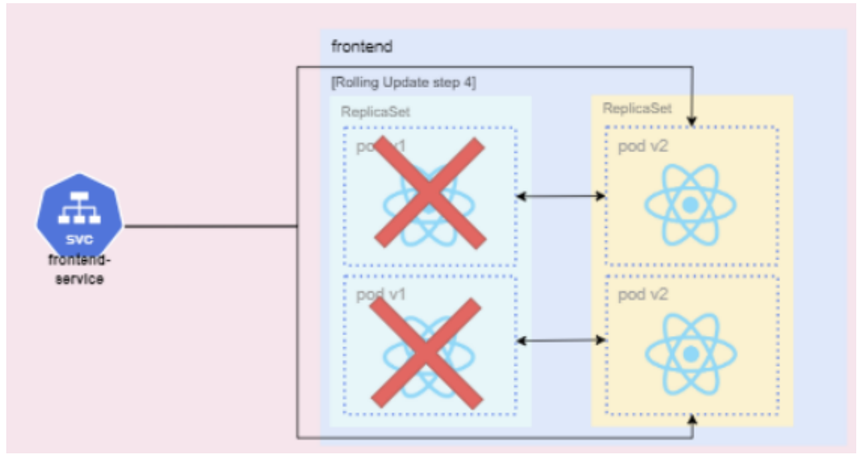
3 기존 Pod 제거

- **maxUnavailable: 1 → 기존 Pod 종료**



4 완료

- **모든 Pod가 v2 버전으로 업데이트**



Rolling Update 방식의 장점

안정성
순차적 교체로 리스크 최소화

무중단 배포
Service가 Ready Pod에만
트래픽 전달

자동 Rollback
잘못된 배포 시 기존
Pod 유지

간편한 관리
별도 인프라 불필요,
YAML 파일로 선언적 관리

04.

BLUE-GREEN 기반
CI/CD 파이프라인 구축

Rolling Update 방식의 문제점

Deployment를 이전 이미지로 되돌린 후
Pod가 순차적으로 재생성

기존 pod와 새 pod가 섞여 있어
충돌 발생 가능

새 버전이 실제 운영 환경에서
정상 동작하는지 미리 확인 불가

따라서, 이러한 문제들을 해결하기 위해
Blue-Green Deployment을 구축했습니다.

Blue-Green 배포

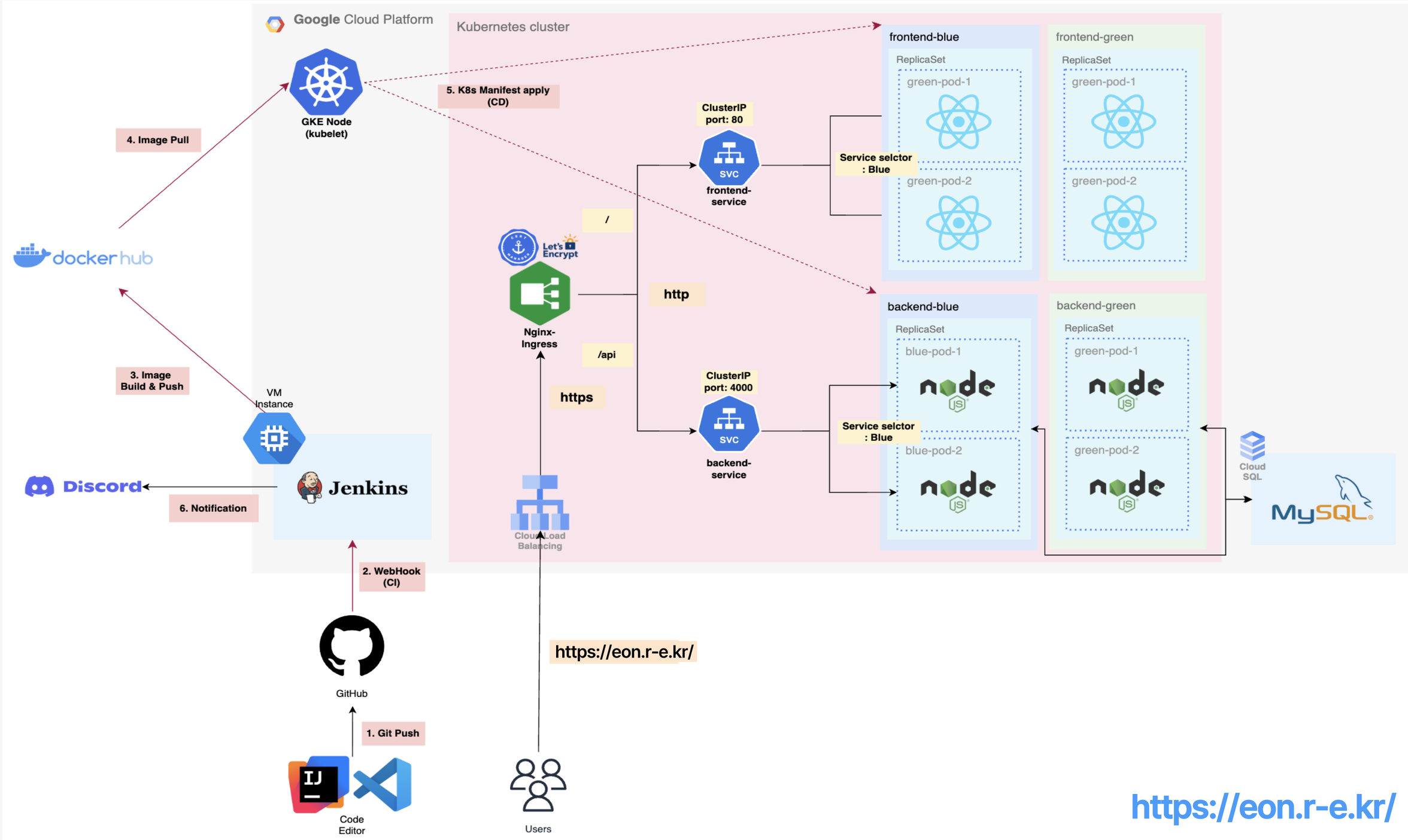
운영 중인 버전(Blue)과 새 버전(Green)을 **완전히 분리된 환경**으로 두고,
검증을 마친 뒤 **트래픽 라우팅만 전환**하는 방식

다운타임이 적은
무중단 배포

운영 환경에서
새 버전
사전 검증 가능

즉각적이고 안전한
Rollback

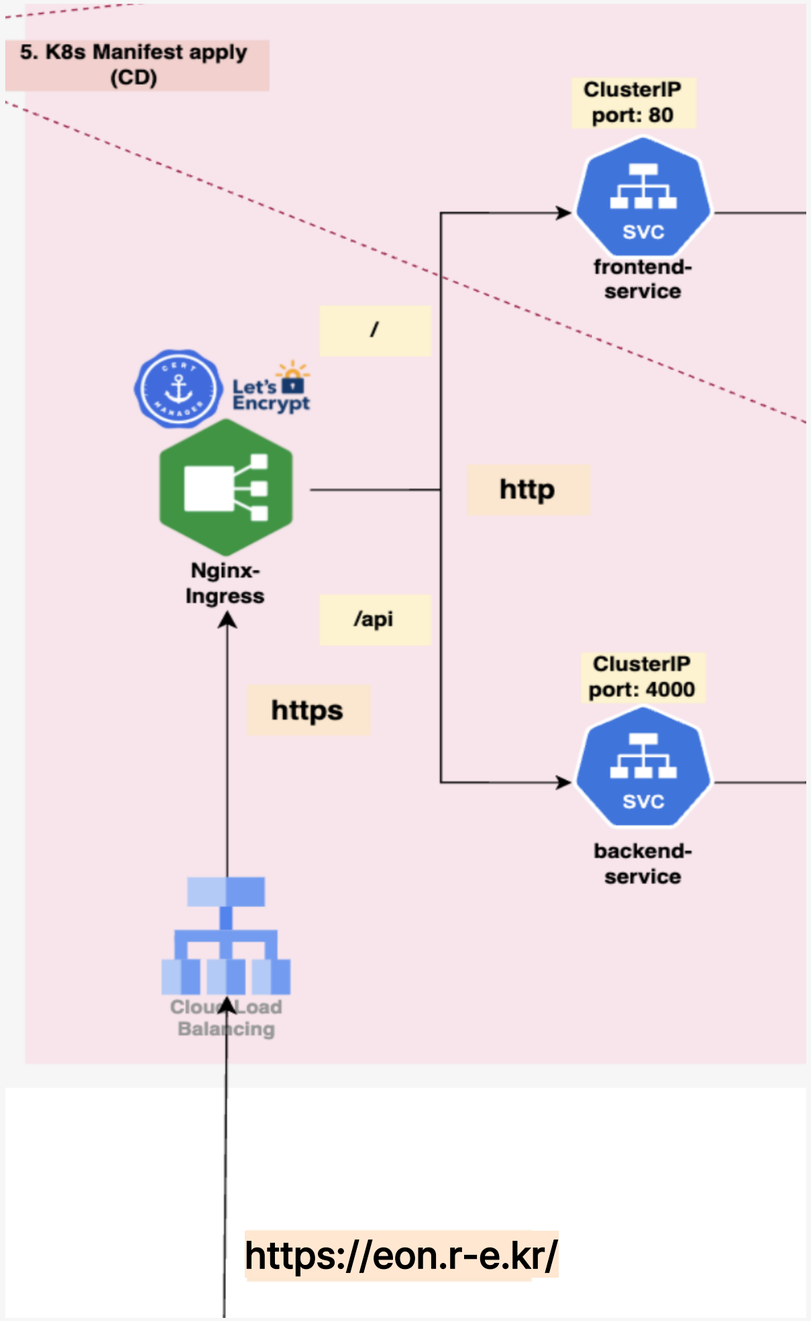
하나의 버전만 서비스하여
서비스의 안전성과
일관성 확보


Google Cloud Platform
Kubernetes cluster
GKE Node
(kubelet)
4. Image Pull
5. K8s Manifest apply
(CD)
frontend-blue
ReplicaSet
green-pod-1
green-pod-2
frontend-green
ReplicaSet
green-pod-1
green-pod-2
ClusterIP
port: 80
SVC
frontend-
service
Service selctor
: Blue
/
http
Let's Encrypt
Nginx-
Ingress
/api
https
ClusterIP
port: 4000
SVC
backend-
service
Service selctor
: Blue
backend-blue
ReplicaSet
blue-pod-1
blue-pod-2
backend-green
ReplicaSet
green-pod-1
green-pod-2
Cloud
SQL
MySQL
dockerhub
3. Image
Build & Push
VM
Instance
Discord
6. Notification
Jenkins
2. WebHook
(CI)
GitHub
1. Git Push
Code
Editor
Cloud Load
Balancing
https://eon.r-e.kr/
Users
https://eon.r-e.kr/

CD 파이프라인

2)
Nginx Ingress
TLS 종료 및
백엔드/프론트엔드 분기

1)
Cloud Load Balancer
외부에서 GKE 클러스터 접근



CD 파이프라인

2)

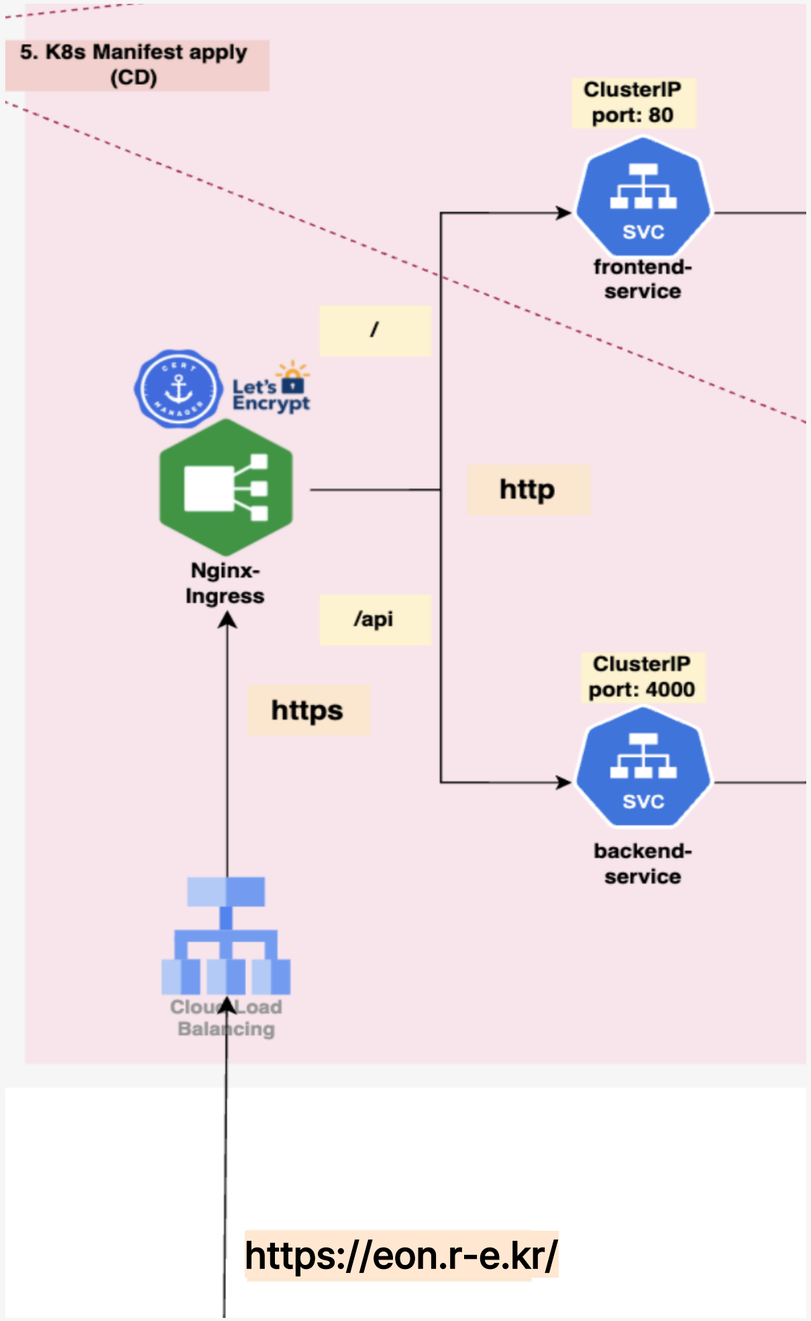
Nginx Ingress

TLS 종료 및
백엔드/프론트엔드 분기

1)

Cloud Load Balancer

외부에서 GKE 클러스터 접근



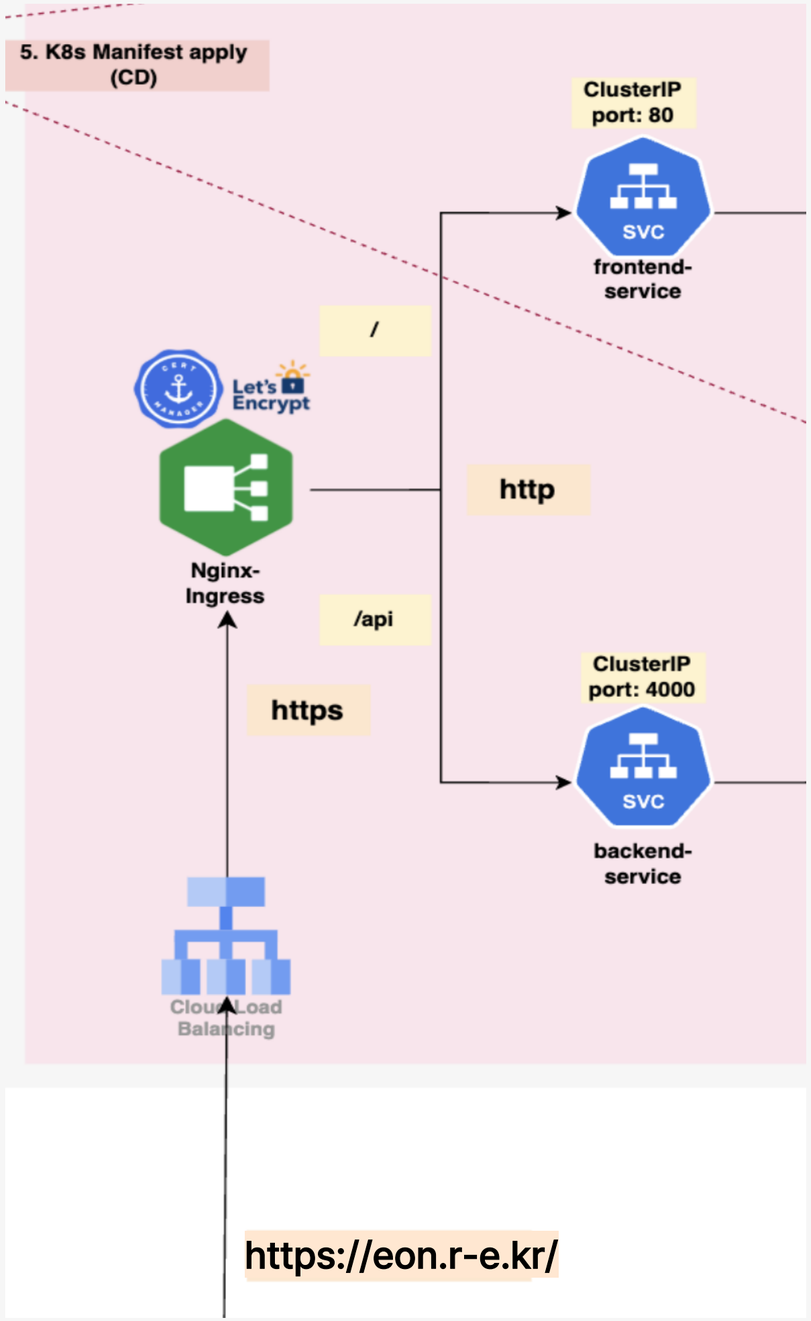
ingress.yml

```yaml
spec:
  ingressClassName: nginx

  tls:
    - hosts:
        - eon.r-e.kr
      secretName: eon-tls-cert

  rules:
    - host: eon.r-e.kr
      http:
        paths:
          # Backend API
          - path: /api
            pathType: Prefix
            backend:
              service:
                name: backend-service
                port:
                  number: 4000

          # Frontend
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend-service
                port:
                  number: 80
```

CD 파이프라인

2)
Nginx Ingress
TLS 종료 및
백엔드/프론트엔드 분기

1)
Cloud Load Balancer
외부에서 GKE 클러스터 접근



frontend-service.yml

```yaml
spec:
  type: ClusterIP
  selector:
    app: frontend
    version: blue
  ports:
    - port: 80
      targetPort: 80
```

backend-service.yml

```yaml
spec:
  type: ClusterIP
  selector:
    app: backend
    version: blue
  ports:
    - port: 4000
      targetPort: 4000
```

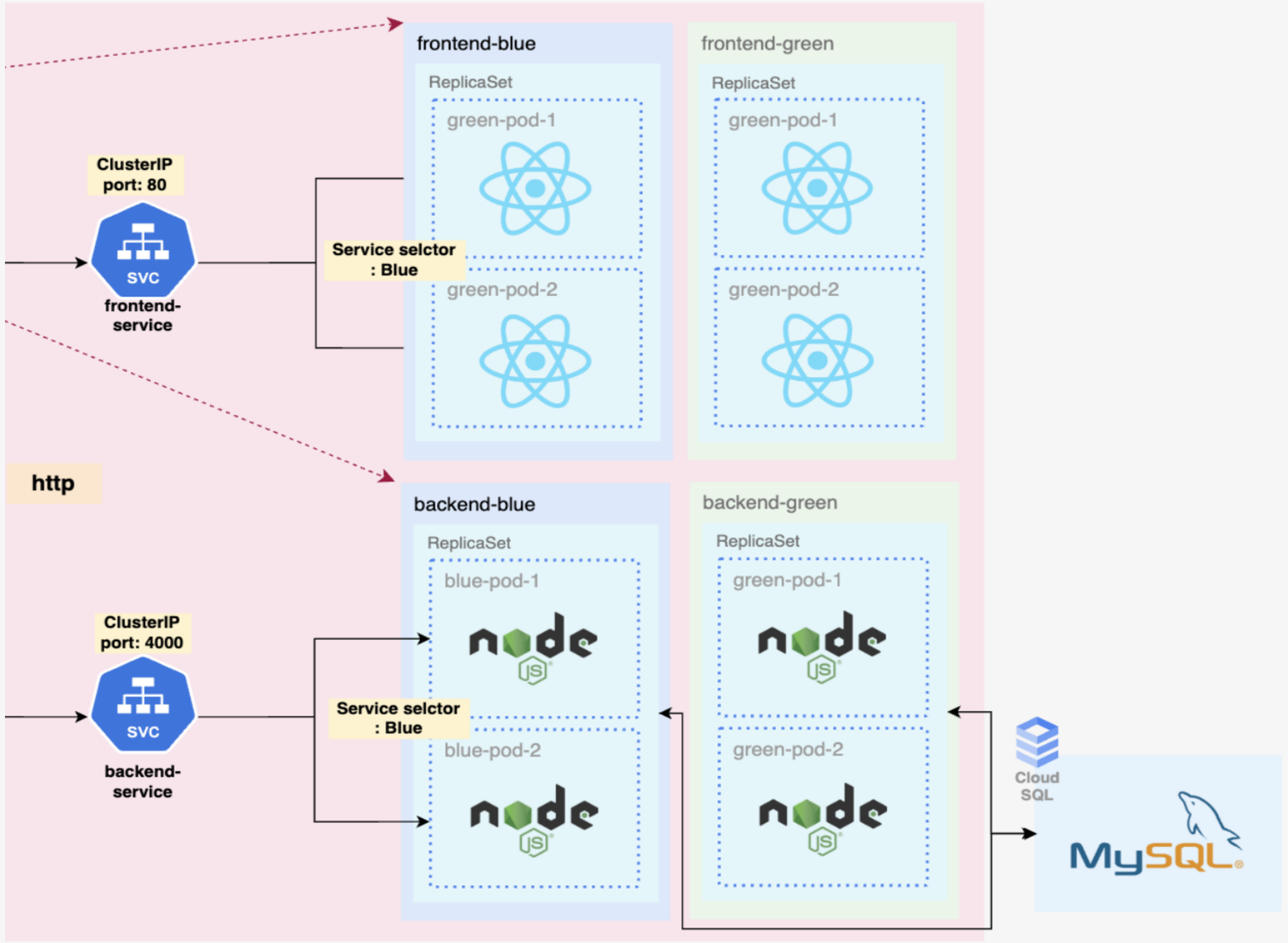
CD 파이프라인

3)

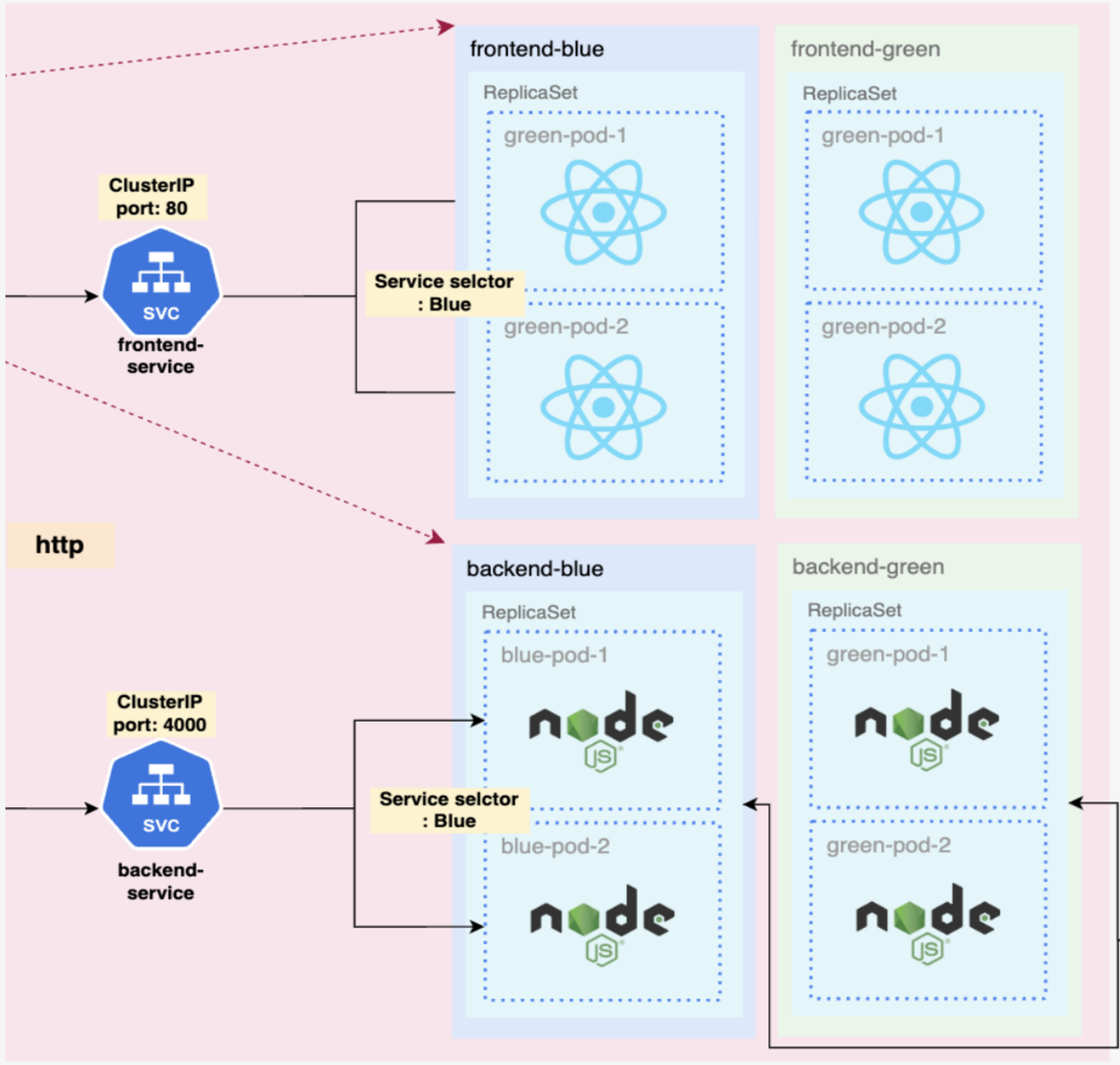
Service

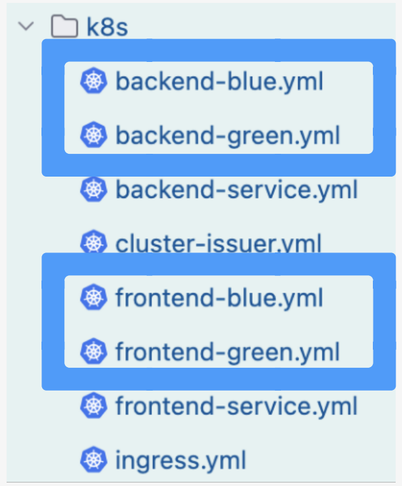
selector을 통해

적절한 파드를 찾아 트래픽 처리



CD 파이프라인





frontend-blue.yml

```yaml
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
      version: blue
```

frontend-green.yml

```yaml
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
      version: green
```

```
sh """
    kubectl set image deployment/${frontTargetDeploy} frontend=${FRONT_IMAGE}:${FRONT_TAG} -n ${NAMESPACE}
    kubectl rollout status deployment/${frontTargetDeploy} -n ${NAMESPACE}
    kubectl patch service ${FRONT_SERVICE} -n ${NAMESPACE} -p '{"spec": {"selector": {"app": "frontend", "version": "${frontTargetVersion}"}}}'
"""
```

Blue-Green 배포의 단점

높은 인프라 비용
운영 환경과 동일한
두 배의 컴퓨팅 리소스
일시적 유지 필요

데이터베이스 관리 복잡성
두 버전이 공유하는
데이터베이스 스키마 변경이
있을 경우 전략 필요

따라서, 여러 배포 방식의 Trade-off를 고려하여
서비스에 가장 적합한 배포 전략을 수립해야 합니다.

05.

트러블 슈팅

Jenkins OOM(Out Of Memory) 해결

문제 Frontend와 Backend 동시 빌드 시 Jenkins 컨테이너 강제 종료 및 워크스페이스 유실.

원인 GCP 인스턴스의 제한된 메모리 환경에서 병렬(Parallel) 빌드 수행으로 인한 리소스 고갈.

해결 parallel 스테이지를 제거하고 순차 실행으로 구조 변경 → 메모리 피크 사용량 감소.

```
[Pipeline] End of Pipeline
ERROR: missing workspace /var/jenkins_home/workspace/e-on-deployee-pipeline-k8s_main on

GitHub has been notified of this commit’s build result

Finished: FAILURE
```

```
#3 [internal] load metadata for docker.io/library/node:20-alpine
#3 ...

#17 exporting to image
#17 exporting layers
Resuming build at Sun Nov 30 11:40:06 UTC 2025 after Jenkins restart
Ready to run at Sun Nov 30 11:44:29 UTC 2025
[Pipeline] }
[Pipeline] }
[Pipeline] // stage
[Pipeline] // stage
[Pipeline] }
Failed in branch Build Backend
[Pipeline] }
Failed in branch Build Frontend
[Pipeline] // parallel
```

```
//병렬 제거하고 순차적으로 실행
stage('Build Backend') {
    steps {
        // 백엔드 Dockerfile로 이미지 빌드
        sh "docker build -t ${BE_IMAGE_NAME}:latest -f backend/Dockerfile ./backend"
    }
}
stage('Build Frontend') {
    steps {
        // 프론트엔드 Dockerfile로 이미지 빌드 (build-arg로 API 주소 주입)
        sh "docker build --no-cache --build-arg VITE_API_URL=${VITE_API_URL} -t ${DOCKERHUB_ID_TEXT}/e-on-frontend:${IMAGE_TAG} -f frontend/Dockerfile ./frontend"
}
```

Kubernetes 배포가 반영되지 않는 '이미지 캐싱' 문제

문제 Jenkins 빌드 성공 후 재배포했으나, GKE 상의 서비스는 구버전 그대로 유지됨.

원인 K8s의 이미지 캐싱 정책: 동일한 태그 사용 시 노드에 저장된 기존 이미지를 재사용. 정적 태그 사용으로 인한 버전 관리 불가.

해결 동적 태그 적용: 빌드 번호(${BUILD_NUMBER})를 이미지 태그로 사용.
Manifest 자동화: 배포 단계에서 sed 명령어로 Deployment 파일의 태그를 실시간 치환.

```
IMAGE_TAG = "${env.BUILD_NUMBER}"
```

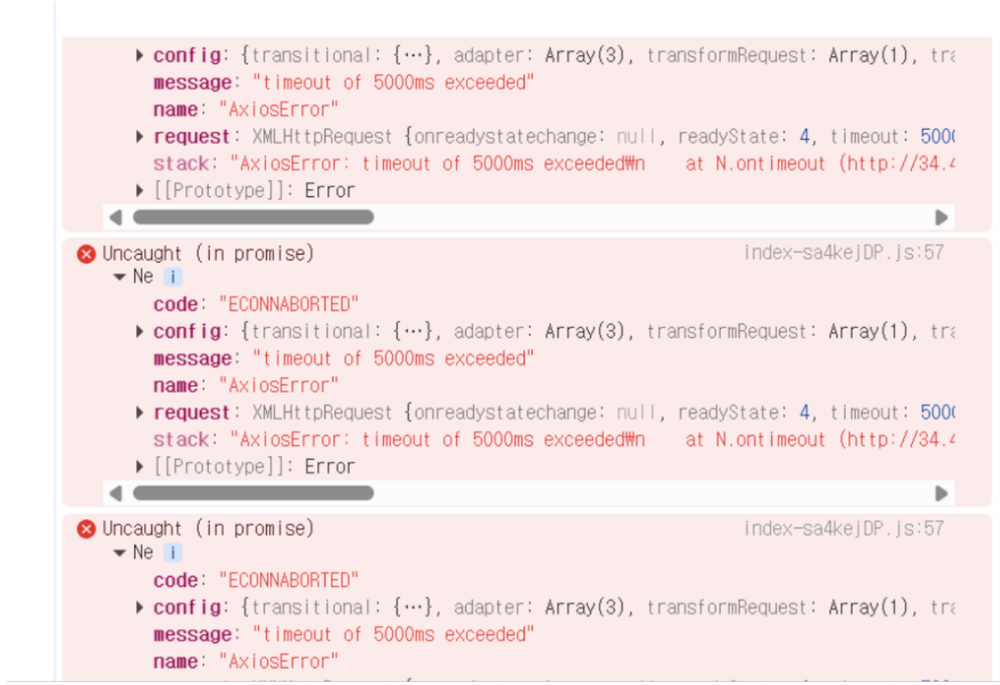
```
stage('Build Frontend') {
    steps {
        // 프론트엔드 Dockerfile로 이미지 빌드 (build-arg로 API 주소 주입)
        sh "docker build --no-cache --build-arg VITE_API_URL=${VITE_API_URL} -t ${DOCKERHUB_ID_TEXT}/e-on-front nd:${IMAGE_TAG} f frontend/Dockerfile ./frontend"
}
```

```
echo ">> Changing image tag t ${IMAGE_TAG} in deployment.yaml..."

# 'frontend:빌드번호'로 변경
sed -i 's|nye0817/e-on-frontend:.*|nye0817/e-on-fronten :${IMAGE_TAG}  k8s/frontend-deployment.yaml
```

서비스 노출 방식의 진화 (ClusterIP → Ingress + NodePort)

문제 사용자 브라우저에서 **백엔드 API 연결 실패** (Axios Timeout)

원인 백엔드 서비스가 **클러스터 내부 IP(ClusterIP)**로만 설정되어 있어, 외부망인 **사용자 브라우저에서 접근 불가.**



서비스 노출 방식의 진화 (ClusterIP → Ingress + NodePort)

해결

e-on-deployee > k8s > ! backend-deployment.yaml

```yaml
---
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  type: LoadBalancer
  selector:
    app: backend
  ports:
    - port: 4000
      targetPort: 4000
```

! ingress.yaml

e-on-deployee > k8s > ! ingress.yaml

You, 5 hours ago | 1 author (You)

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: eon-ingress
  annotations:
    kubernetes.io/ingress.class: "gce"
spec:
  defaultBackend:
    service:
      name: frontend
      port:
        number: 80
  rules:
  - http:
```

```yaml
kind: Service
metadata:
  name: frontend
spec:
  type: NodePort
```

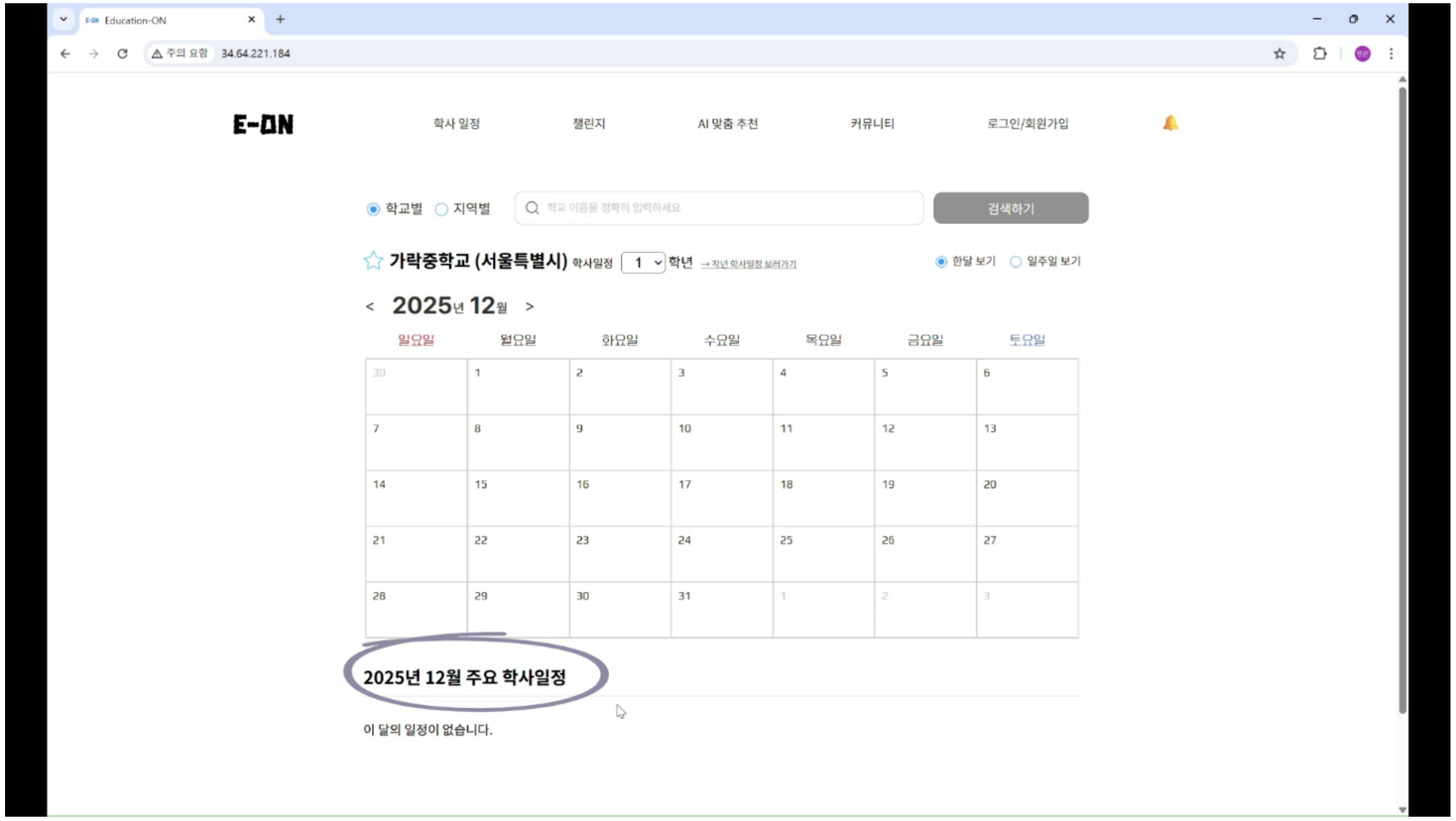
```yaml
kind: Service
metadata:
  name: backend
spec:
  type: NodePort
```

처음: LoadBalancer 타입 변경 (공인 IP 할당, 보안 취약).

이후: Ingress 도입. 단일 진입점에서 라우팅 처리 및 보안 강화.

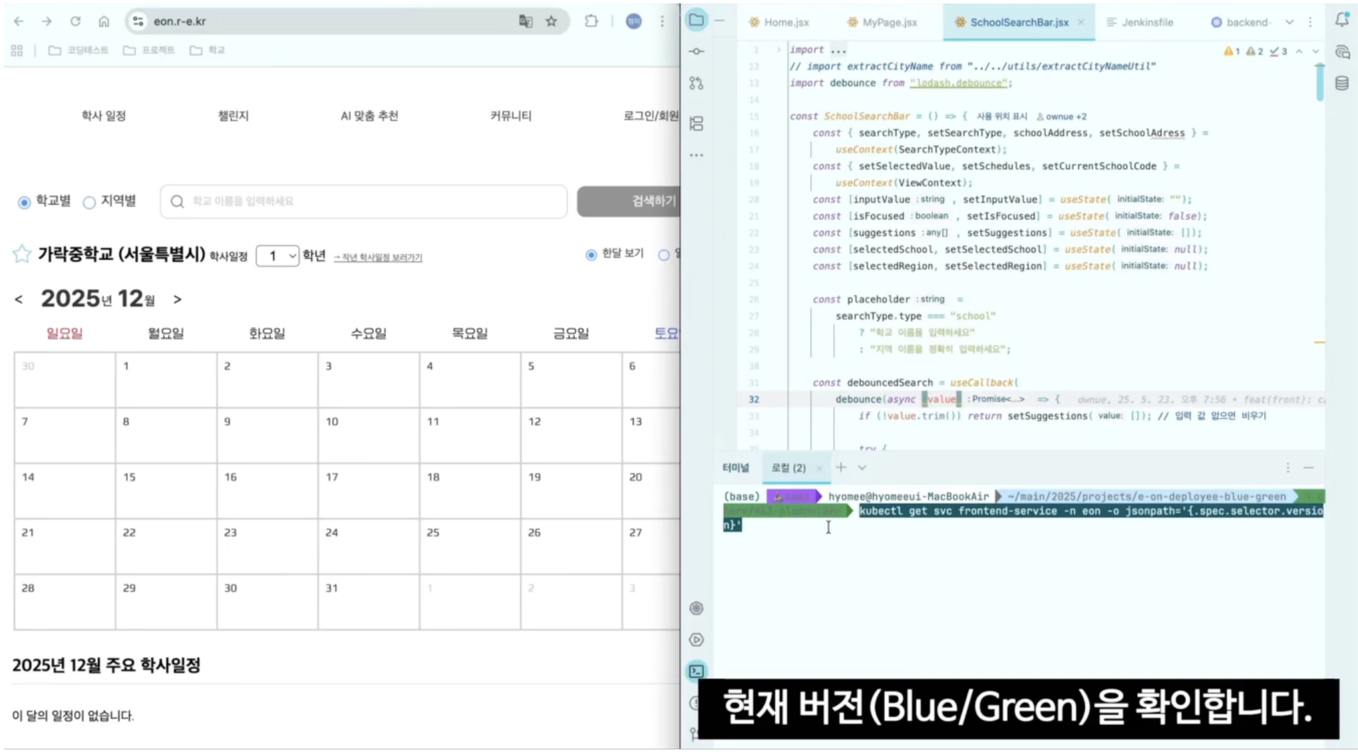
시연 영상

https://www.youtube.com/watch?si=RAF-czTh8lPkL9y9&v=QKsKL9WsBvA&feature=youtu.be



시연 영상

https://youtu.be/Jp0wrMOG1No?si=dz8O0K57S6DgkZWS



Lessons Learned

VM 기반의 배포에서 쿠버네티스 환경으로 확장해보며 컨테이너 오케스트레이션의 실효성을 체감할 수 있었다. CI 과정에서는 Webhook과 Jenkins를 연동해 변경된 서비스만 감지하여 빌드하는 방식을 도입하고, 런타임 환경까지 고려한 Docker 이미지를 구성하며 리소스 최적화의 중요성을 깊이 이해할 수 있었다.

특히 프로젝트 진행 중 흥미로웠던 점은 '구현의 다양성'이었다. 같은 CI/CD 기능을 구축하더라도 팀원마다 스크립트를 짜는 방식과 구조가 조금씩 다른 것을 공유하며 서로의 접근 방식에서 많은 것을 배울 수 있었다. 이러한 고도화 작업을 통해, 잘 구축된 데브옵스 환경이 실제 개발과 운영 과정을 얼마나 편하고 효율적으로 만들어주는지 몸소 경험했다.

나아가 K8s와 Ingress를 활용해 Blue-Green 배포를 직접 구현하고 트래픽을 제어해보며 실무적인 감각을 익힐 수 있었다. 이 과정에서 실제 발생하는 GCP 클라우드 비용을 분석해보며, 기술 도입 시에는 단순히 기능 구현뿐만 아니라 비용과 서비스 특성을 고려한 전략적 판단이 필수적임을 깨달을 수 있었다.

06.

질의응답

감사합니다.