MSA 서비스 최적화 및 안정성 강화 프로젝트

SeSAC Market

Cloud2scape

변경 사항

버전	작성일	변경 내용	설명	작성자
0.0.1	2024년 11월 22일	최초 작성	스타일, 디자인 정형화	옥재욱
0.1.0	2024년 11월 23일	목차 골조 작성		옥재욱
0.2.0	2024년 11월 24일	테스트 양식 작성		옥재욱
0.3.0	2024년 11월 25일	테스트 내용 작성		김수민
0.3.1	2024년 11월 26일	테스트 양식 수정		옥재욱
0.4.1	2024년 11월 27일	인프라 관련 부문 작성		고나연
0.4.2	2024년 11월 28일	인프라 부문 작성		고나연
0.5.2	2024년 11월 29일	기술 스택 그림 추가		옥재욱
0.6.2	2024년 12월 1일	쿠버네티스 구성도 추가		옥재욱
0.7.2	2024년 12월 2일	모니터링 추가		박찬준
0.7.3	2024년 12월 3일	보고서 일부 수정 및 추가		고나연
0.7.4	2024년 12월 4일	검수	스타일, 오타	옥재욱
0.7.5	2024년 12월 4일	보고서 일부 추가	모니터링, 인프라 부문	고나연
1.0.0	2024년 12월 4일	초안 작성		박찬준
1.0.1	2024년 12월 5일	오탈자 수정		옥재욱
1.0.2	2024년 12월 6일	인프라 아키텍처 수정		옥재욱
1.0.3	2024년 12월 9일	인프라 아키텍처 수정		옥재욱

<제목 차례>

1. 프로젝트 개요

본 프로젝트는 시스템의 효율적인 운영과 안정적인 서비스 제공을 목표로, 마이크로서비스 아키텍처(MSA)를 도입하여 서비스 독립성, 확장성 및 개발 효율성을 강화했습니다. 다층적 모니터링 체계를 구축하여 인프라, 애플리케이션 성능, 비즈니스 메트릭 및 로그 분석을 통해 시스템의 실시간 상태를 추적하고 문제를 빠르게 해결할 수 있도록 했습니다. 서비스 디스커버리, 데이터 정합성 확보, 보안 체계 강화를 통해 MSA 환경에서의 시스템 안정성을 높였습니다.

1.1. 배경

사용자 수는 시스템의 규모와 복잡도에 큰 영향을 미치며, 서버 비용, 데이터베이스 용량, 네트워크 대역폭 등에 차이를 발생시킵니다. 사용자 수가 증가하면 시스템 통합 정도와 보안 요구사항이 높아져, 개발 및 유지보수 비용이 증가합니다. 이를 효율적으로 관리하고 클라우드에 대한 비용을 최적화하며 자원 사용에 대한 가시성을 확보할 필요가 있습니다. 사용량 증가에 따른 지출을 통제하고 효율적인 자원 할당을 위한 관측가능성을 구축할 수 있습니다.

1.2. 목표

본 프로젝트는 시스템의 성장과 변화에 유연하게 대응하고, 서비스 품질을 유지하기 위해 마이크로서비스 아키텍처(MSA)를 도입했습니다. MSA는 서비스를 독립적으로 개발, 배포하고 관리할 수 있도록 하여 시스템의 확장성과 유연성을 높였습니다. 또한, 다층적인 모니터링 체계를 구축하여 시스템의 실시간 상태를 파악하고 문제 발생 시 신속하게 대응할 수 있도록 했습니다. 특히, 사용자 증가에 따른 시스템 부하 증가에 대비하여, 클라우드 기반 자원을 효율적으로 활용하고 비용을 최적화하는 방안을 모색했습니다. 이를 통해 시스템의 안정성과 확장성을 확보하고, 지속적인 성장을 위한 기반을 마련하고자 하였습니다. 본 프로젝트의 궁극적인 목표는 사용자 수 증가에 따른 시스템 변화에 유연하게 대응하고, 최적의 성능과 안정성을 제공하는 시스템을 구축하는 것입니다.

시스템 안정성 향상: 99% 이상의 가용성 확보

- 목표치에 도달하여 최소 120분 이상 서비스 유지
- 서비스 장애에 대한 폴백 매커니즘 설계

성능 최적화: 응답시간

- Disk IO 시간 - 500ms 이내
- DB 쿼리시간 - 300ms 이내
- 애플리케이션 응답시간 - 2000ms 이내

확장성있는 아키텍처 구축

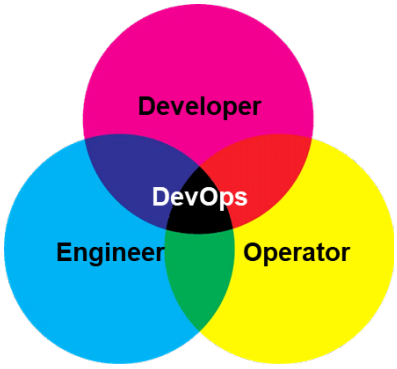
- 컨테이너 기반 오토스케일링
- 읽기/쓰기 분리를 통한 트래픽 분산
- 클라우드 네이티브
- 효율적인 캐시 전략

효율적인 모니터링 시스템 구현

- 메트릭 수집
- 로그 통합 관리
- 시각화

1.3. 구성원 역할

각 구성원의 역할이 DevOps에 도달할 수 있도록
연관 관계가 적은 업무부터 단계적으로 수행했습니다.
점진적으로 연관된 업무가 수행될 수 있도록 분담했습니다.
관계를 감산혼합(CYMK)으로 표현했습니다.



[그림 1] 역할 벤다이어그램

박찬준(Engineer, PM, Operator)
- 모니터링 환경 구축 및 데이터 시각화
- 데이터 가공
- 일정 관리 및 의견 조율
- 신뢰성 및 효율성 테스트 설계

김수민(Engineer, DBA)
- 데이터베이스 구축
- 부하 테스트 설계

고나연(Engineer, TA)
- 클라우드 인프라 아키텍처 설계
- CD(ArgoCD) 구축
- K8s 클러스터 환경 제공

옥재욱(Developer, QA)
- 개발 인프라 환경 구축
- 애플리케이션 개발
- CI(Github Actions) 구축
- 품질 테스트

Developer + Engineer
- IaC 코드 작성(Terraform, K8s)
- 보안 아키텍처 설계/구현

Engineer + Operator
- 인프라 모니터링 체계 구축
- 리소스 사용량 최적화

Operator + Developer
- 로깅 및 모니터링 지표 최적화
- 서비스 안정성 메트릭 개선

DevOps
- 서비스 품질 검증
- 워크플로우 개선

1.4. 프로젝트 일정

Role 별 색 표현, 겹치는 영역을 감산혼합(CMYK)으로 표현(개발 보라, 구축 시안, 운영 노랑)

기획 및 착수

- 서비스 최소 요구사항 설정
- 아키텍처 설계

수행 1 주차

- 애플리케이션 개발
- 개발(로컬)/운영(클라우드) 환경 인프라 구축

수행 2 주차

- 애플리케이션 배포
- 운영 모니터링 환경 구축

수행 3 주차

- 역할 별 기능/비기능 보완
- 테스트 제반 마련 및 시나리오 작성

수행 4 주차

- 테스트 수행 및 보고서 작성
- 시스템 최종 보완

종료

- 프로젝트 WIKI 작성, 산출물 정리

2. 시스템 아키텍처

2.1. 기술스택

App

Spring boot, Spring Cloud, Spring Batch, Spring Security, Resilience4j, Gradle, JPA Hibernate, Okta, OpenAPI, Thymeleaf

Database

Mysql, Redis, Kafka, H2

CI/CD

Actions, Argo CD

IaC

K8s, Helm, Terraform, Ansible, Vagrant, Shell, Docker

Monitoring

Grafana, Prometheus, Loki, Promtail, Open telemetry, Zipkin

Cloud

EKS, S3, Cloud Watch, Aurora, Elasti Cache, Route53, VPC, ELB, Shield, WAF

Tests

JUnit5, Mockito, Monkey Fixture, Test Container, Sonar qube, Jmeter, Lighthouse

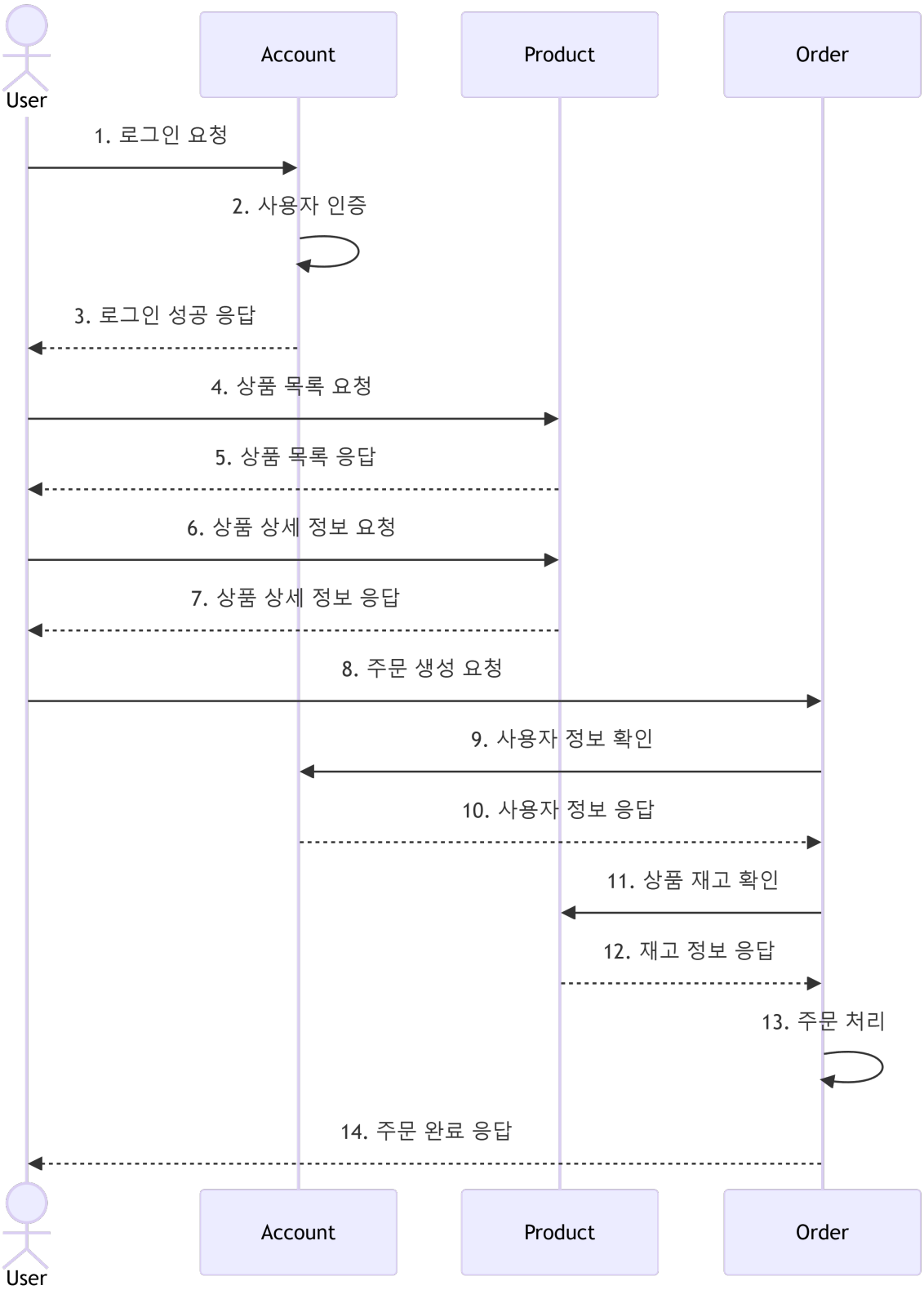
Other

Slack, Github, IDEA, VSCode, Bootstrap Studio

[그림 2] 기술스택

2.2. 유스케이스 및 시나리오

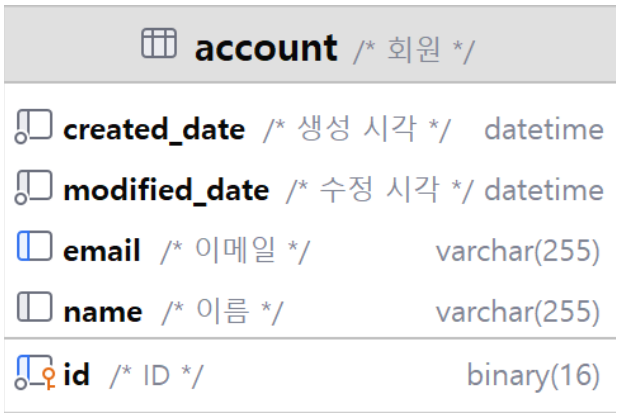
마켓 서비스의 유스케이스입니다.



[그림 3] 새싹 마켓의 유스케이스 다이어그램

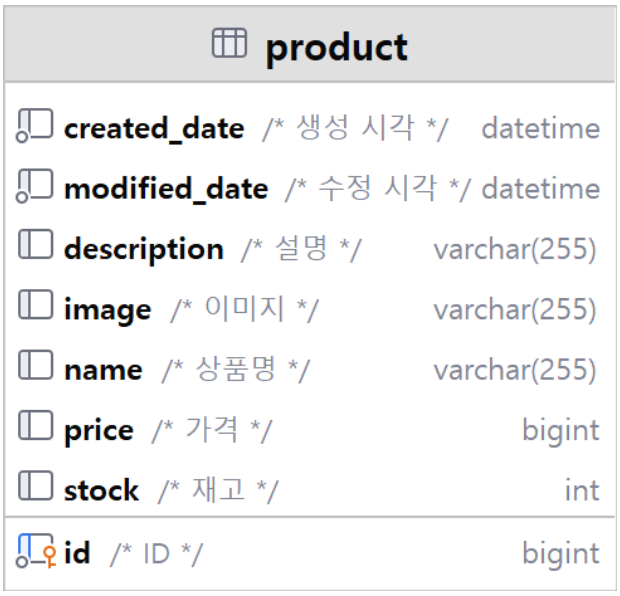
2.3. ERD(Entity Relationship Diagram)

테이블 구성



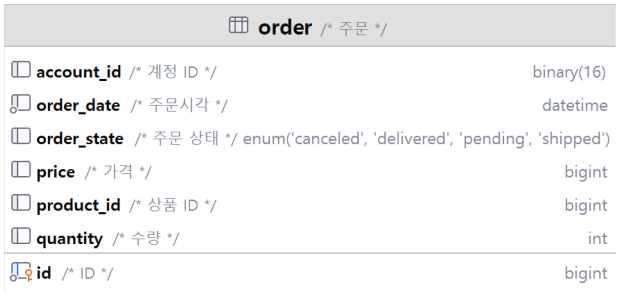
[그림 4] 회원(Account) 테이블

- 회원 ID는 UUID 로 생성했음
- 생성, 수정 시각은 JPA로 자동 생성됨
- 이메일은 유효성 검사



[그림 5] 상품(Product) 테이블

- 가격, 재고 유효성 검사(0 이상)
- 이미지 경로 유효성 검사(http)
- 생성, 수정 시각은 JPA로 자동 생성됨
- 상품 ID는 TSID 적용했음(분산 DB 고려)

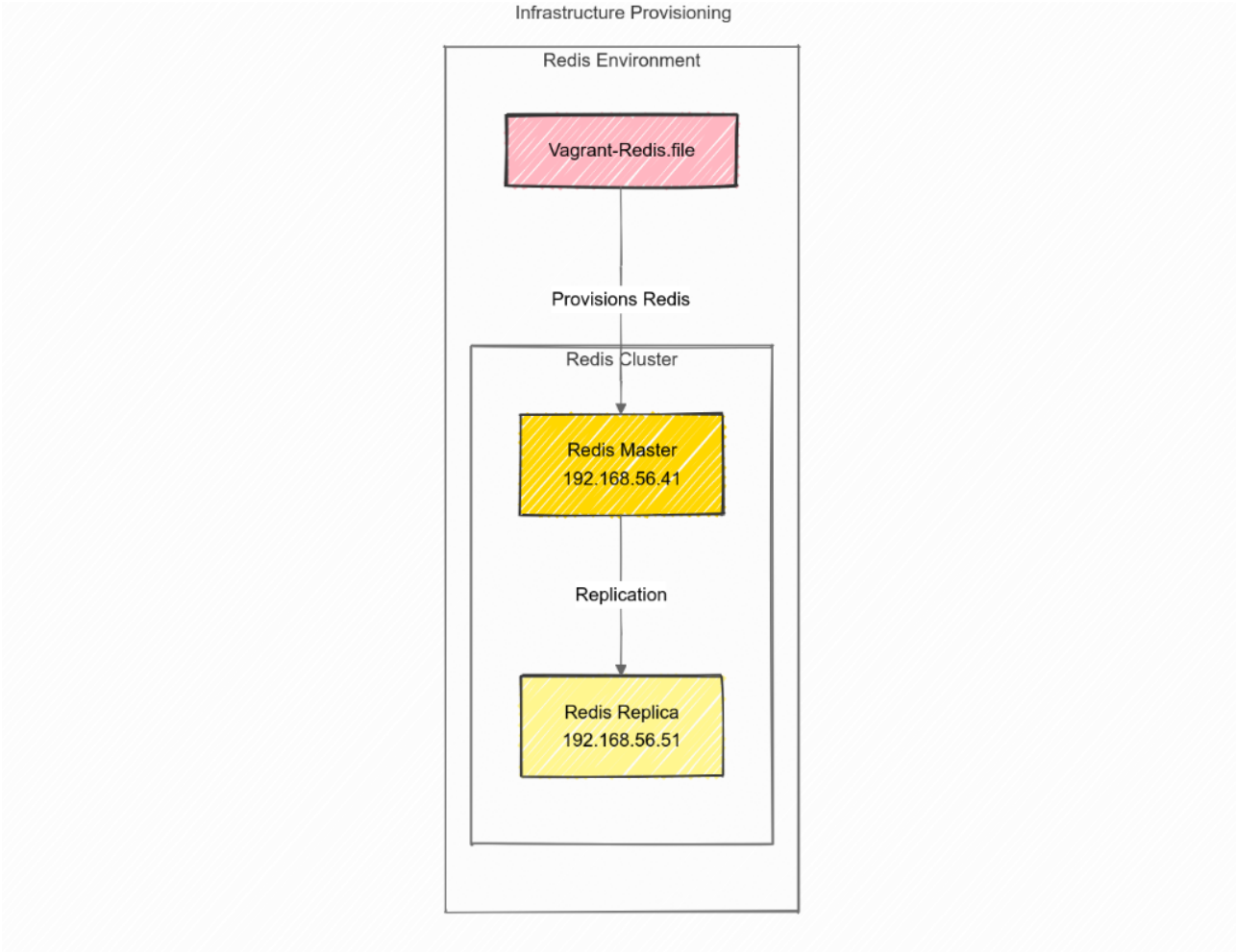


[그림 6] 주문(Order) 테이블

- 주문 ID는 TSID 적용했음(분산 DB 고려)
- 생성, 수정 시각은 JPA로 자동 생성됨
- 가격, 수량 유효성 검사(0 이상)

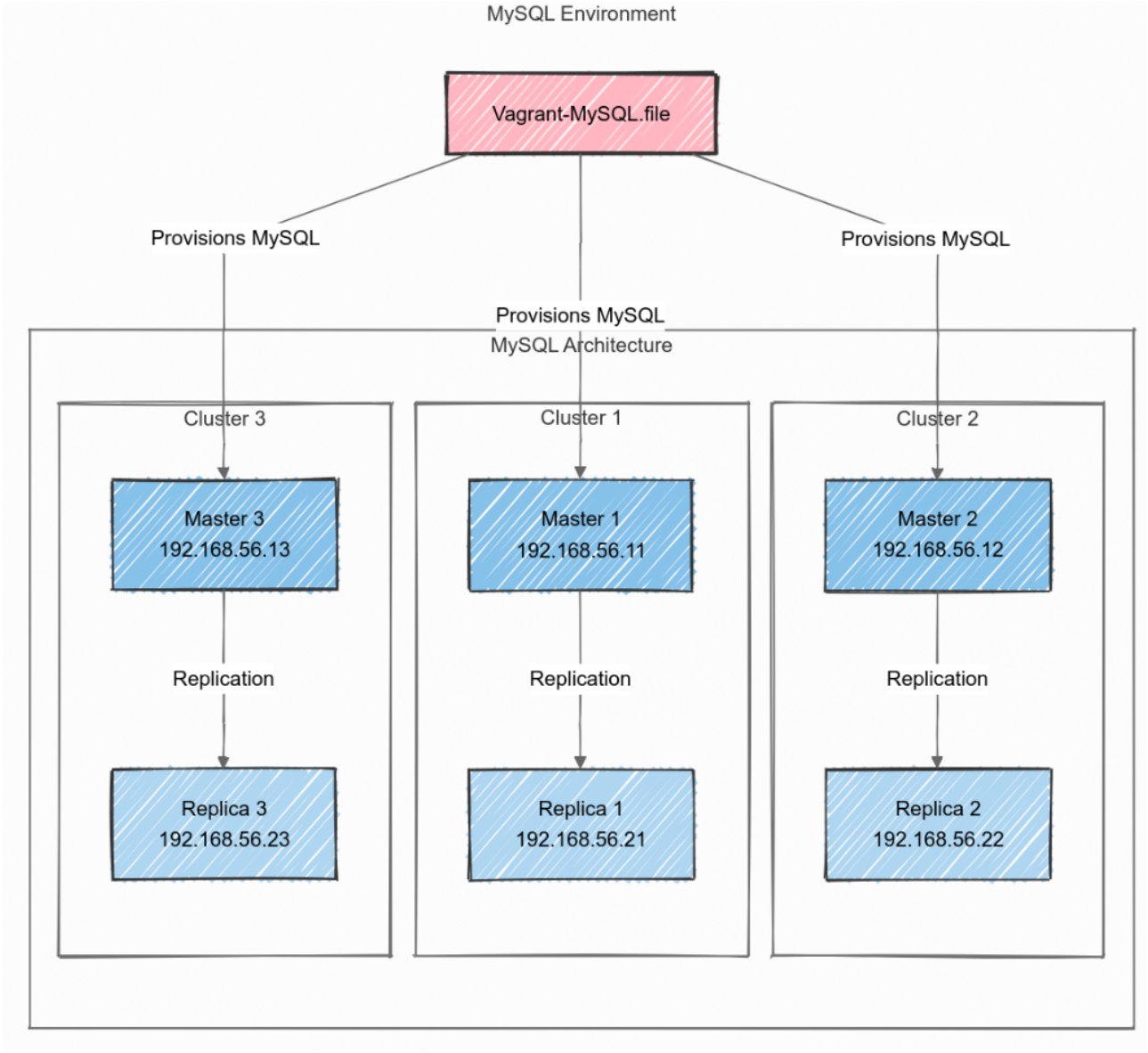
3. 인프라 구축

3.1. 개발환경(온프레미스) 설계



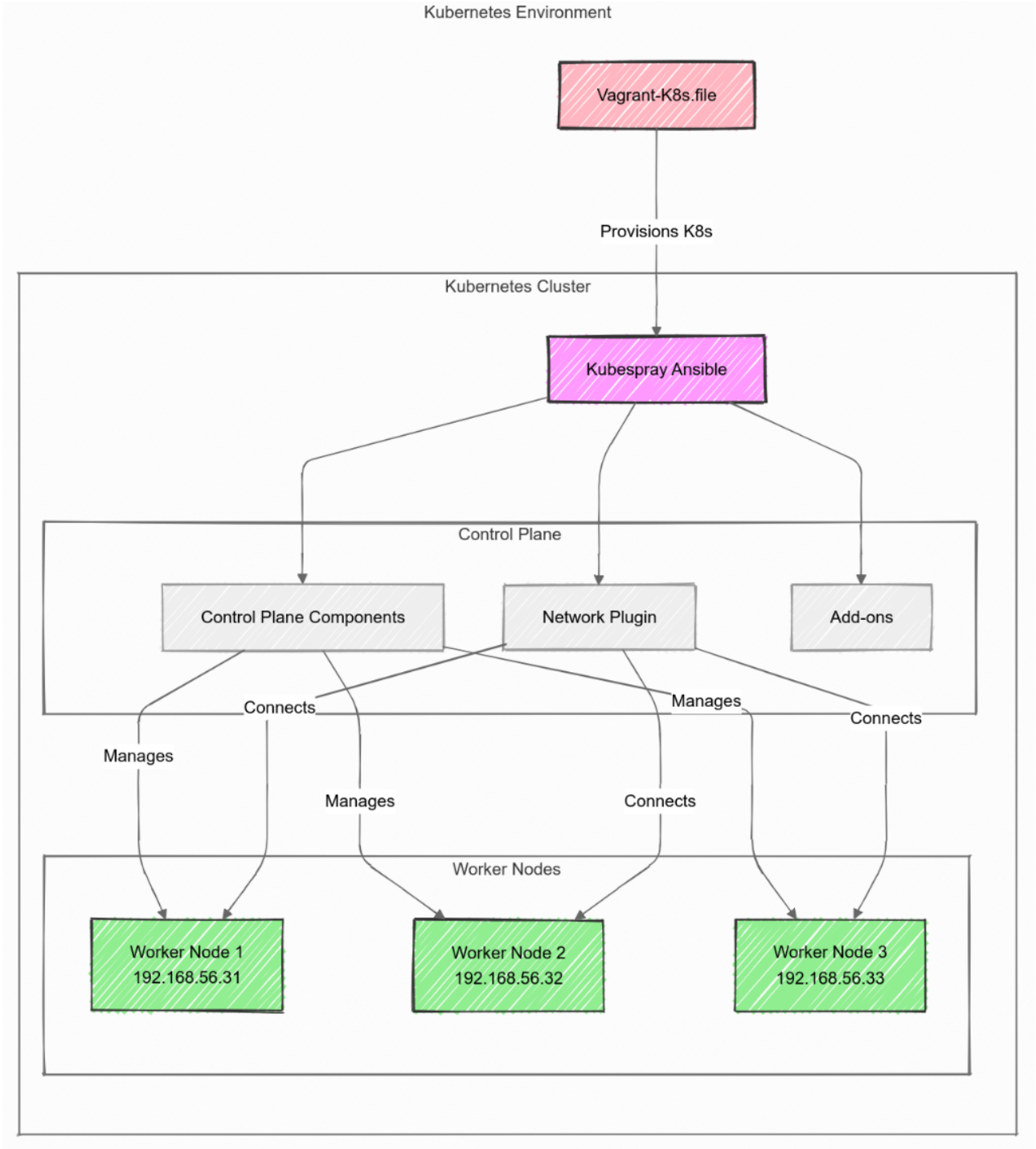
[그림 7] 개발 환경에 Vagrant로 제공 – Redis Cluster

마스터 노드(192.168.56.41)와 레플리카 노드(192.168.56.51) 한 쌍으로 이루어진 클러스터 형태를 제공하여 개발단이 최소한의 자원 사용으로 테스트할 수 있도록 서빙합니다. Vagrant로 작성함으로 언제든지 다시 복구하여 테스트할 수 있도록 자동화됩니다. VirtualBox, VMware 등 다양한 VM에 대응할 수 있도록 하여 로컬 환경의 변화에도 대응 가능합니다. Redis 클러스터의 마스터-레플리카 구성 및 샤딩, 메모리 설정이 코드로 명시되어 있으므로 개발자가 통제 가능한 환경 구축이 용이합니다. 코드 형태로 이루어져 있으므로 테스트 환경의 버전 관리가 가능합니다.



[그림 8] 개발 환경에 Vagrant로 제공 – Mysql Replication

실제 환경과 유사한 분산 DB 환경을 구축하므로 개발자들은 실제와 같은 데이터 복제, 장애 복구, 데이터 동기화 등의 시나리오를 테스트하고 검증할 수 있습니다. 격리된 환경이기 때문에 장애 발생 시 빠르게 복구할 수 있으며, 다른 리소스에 2차 피해를 주지 않습니다. 개발자가 바뀌더라도 일관된 환경을 제공할 수 있어 작업 효율성이 높아집니다. 클러스터 설정 과정을 자동화함으로써 설정 오류를 최소화할 수 있습니다. 또한, 신속한 환경 구성이 가능하여 개발자의 위치에 관계없이 필요한 환경을 제공할 수 있습니다.



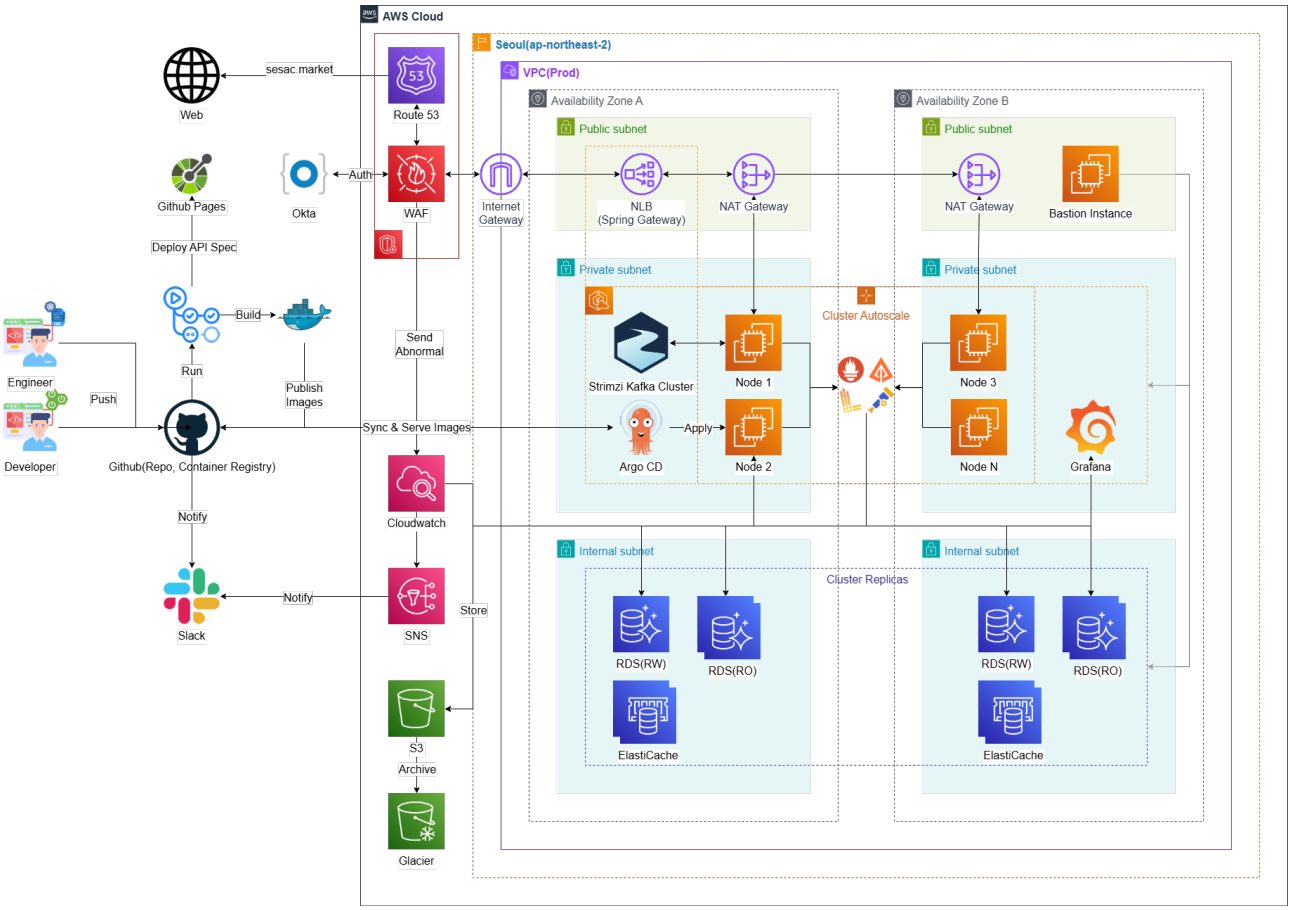
[그림 9] 개발 환경에 Vagrant로 제공 – Kubespray(K8s)

복잡한 쿠버네티스 설정을 자동화된 방식으로 구성할 수 있습니다. 컨트롤 플레인과 워커 노드, 네트워크 플러그인 및 보안 설정 등이 Vagrant 파일에 코드로 정의되어 있으므로 모든 개발자에게 동일한 환경을 보장할 수 있습니다.

3.2. 운영환경(클라우드 인프라) 설계

3.2.1. 클라우드 배포된 환경

본 아키텍처는 확장성, 보안성, 그리고 모니터링 효율성을 고려하여 설계되었습니다.



[그림 10] 클라우드 인프라 구조

데이터 부문 Redis, Aurora, Kafka는 클러스터로 구성했습니다.
애플리케이션은 CI/CD 자동화되어 개발자가 업데이트하면 Github를 거쳐 AWS까지 전달되어 오케스트레이션 될 수 있도록 설계했습니다. 애플리케이션의 Gateway를 ELB내 NLB로 연결시킨 후 Route53으로 DNS할당해 외부로 노출시켰습니다.
이를 로그, 트레이스, 트레이스 별 컬렉터를 연결해 Grafana로 전달하여 시각화했습니다.

3.2.2. Network Load Balancer (NLB)

EKS 클러스터에 배포된 애플리케이션이 외부 트래픽을 받을 수 있도록 안정적으로 라우팅하며, 클러스터의 오토스케일링에 영향을 받지 않습니다.
다이어그램에서 NLB는 VPC 내외부 트래픽을 연결하는 중추적인 역할을 하며, 외부 클라이언트와 내부 리소스를 안전하게 연결합니다.
NLB를 통해 CI/CD 파이프라인에서 애플리케이션 배포와 테스트 트래픽 처리를 효율적으로 수행할 수 있습니다.
NLB가 클라우드 환경에서 안정성과 확장성을 제공하며, 특히 대규모 트래픽 처리와 Kubernetes 환경에서 필수적인 로드 밸런싱을 제공하는 데 중요한 역할을 합니다.

3.2.3. Elastic Kubernetes Service (EKS)

EKS는 AWS에서 관리형 Kubernetes 서비스를 제공합니다. Kubernetes는 컨테이너화된 애플리케이션을 배포, 확장, 관리하는 오픈소스 플랫폼으로, EKS는 이를 AWS에서 쉽게 사용할 수 있도록 지원합니다. EKS는 Kubernetes 클러스터를 자동으로 관리하고 AWS의 다양한 기능과 통합하여 운영 및 확장성을 높입니다. 완전 관리형 Kubernetes인 만큼 컨트롤 플레인(Control Plane)의 관리 및 유지보수를 AWS가 자동으로 처리합니다.

이를 통해 사용자는 워커 노드와 애플리케이션 운영에만 집중할 수 있습니다.
통합된 AWS 서비스라 Auto Scaling, IAM, VPC, CloudWatch 등 AWS의 기본 서비스와 긴밀히 통합되어 Kubernetes 클러스터 관리가 용이합니다. 또한, AWS Outposts 또는 온프레미스에서 Kubernetes를 실행하도록 지원합니다.

IAM 인증 통합 및 Amazon VPC 네트워크 보안을 활용해 클러스터를 안전하게 보호합니다. OIDC를 통해 역할 기반 접근 제어(RBAC)를 더욱 세밀하게 설정 가능합니다. 또한, Kubernetes 버전 업그레이드 및 보안 패치를 AWS가 자동으로 처리합니다. 본 다이어그램에서는 프라이빗 서브넷에 배치된 애플리케이션 컨테이너를 실행합니다. 외부 접근 차단 및 Kubernetes 기반으로 애플리케이션 배포, 관리, 확장 가능합니다.

3.2.4. RDS (Relational Database Service)

완전 관리형 서비스로 데이터베이스 소프트웨어의 설치, 패치, 업그레이드, 백업 및 복구 등 반복적인 관리 작업을 AWS가 대신 처리합니다. 데이터베이스 관리 작업을 자동화하여 운영에 소요되는 시간을 절약할 수 있습니다. 사용한 만큼만 지불하는 요금제를 통해 초기 비용 부담 없이 유연한 데이터베이스 관리를 지원합니다. 또한, 다양한 엔진을 지원하기에 MySQL, PostgreSQL, MariaDB, Oracle, SQL Server 등 널리 사용되는 데이터베이스 엔진을 선택할 수 있습니다.
본 다이어그램에서는, RDS에 중요 정보가 포함되어 있는 만큼 따로 내부 서브넷을 만들어 배치하여 외부 접근을 차단합니다. 또한 다중 가용 영역(Multi-AZ) 배포 옵션을 제공하여 장애 발

생 시 자동으로 복구(지정된 기간동안 자동 백업을 유지하고 필요한 경우 특정 시점으로 복구(Point-in-Time Recovery) 기능 제공) 가 가능합니다.

본 인프라에서는 필요에 따라 데이터베이스의 스토리지 용량 및 읽기 성능을 확장(리드 레플리카)할 수 있습니다. 읽기 전용(Read Only)과 쓰기 전용(Write Only) RDS 인스턴스 분리를 통해 성능을 최적화합니다. 또한, Amazon VPC와 통합되어 데이터베이스를 격리된 네트워크 환경에서 실행할 수 있으며, IAM 및 KMS로 인증과 암호화를 제공합니다.

3.2.5. Redis

Redis는 Remote Dictionary Server의 약자로, 오픈 소스 기반의 인메모리 키-값 데이터 저장소입니다. 데이터베이스, 캐시, 메시지 브로커 등 다양한 용도로 사용되며, 특히 빠른 읽기/쓰기 속도로 잘 알려져 있습니다.
Redis는 데이터를 메모리에 저장하기 때문에 높은 성능을 제공하며, 리스트, 셋, 정렬된 셋, 해시 등 다양한 데이터 구조를 지원합니다. 이를 통해 실시간 애플리케이션에 적합한 솔루션을 제공합니다.
데이터를 메모리에 저장하여 매우 낮은 레이턴시로 데이터 처리가 가능하며, Redis 클러스터를 통해 데이터 분산 및 복제를 지원하며, 고가용성과 확장성을 제공합니다. 또한, 사용자 정의 스크립트를 실행하여 데이터 처리 로직을 효율적으로 수행합니다
본 다이어그램에서는 EKS와 통합되어 세션 데이터와 캐시를 처리하고, 내부 서브넷에 배치되어 보안을 강화합니다.

3.2.6. CI/CD Pipeline

Continuous Integration (CI) 단계에서는 코드 변경이 자동으로 빌드되고, 테스트 환경에서 테스트를 실행하여 통합되는 단계입니다. 팀이 동시에 작업해도 충돌을 줄이고, 코드의 안정성을 유지합니다.

Continuous Delivery/Deployment (CD) 단계에서는 CI 후, 준비된 애플리케이션이 자동으로 프로덕션 배포를 준비합니다. 운영 환경으로의 배포를 자동화하거나 수동 승인을 포함할 수 있습니다. 혹은, 준비된 코드를 프로덕션 환경에 자동으로 배포하는 단계입니다. 배포를 완료한 후에도 모니터링을 통해 지속적인 관리를 지원합니다

본 아키텍처에서는 GitHub Actions와 ArgoCD를 사용한 지속적 통합 및 배포합니다. GitHub Actions는 GitHub에서 제공하는 CI/CD 도구로, 코드를 자동으로 빌드, 테스트, 배포할 수 있는 워크플로우를 제공합니다. '.github/workflows'디렉터리에 YAML 형식으로 작성된 구성 파일을 기반으로 작동합니다. 이는 GitHub 리포지토리와 완벽히 통합되어, 코드 푸시, 풀 리퀘스트 등의 이벤트에 따라 트리거될 수 있습니다. 또한, GitHub Marketplace에서 워크플로우 확장을 위한 플러그인을 제공하며, 커스텀 액션도 작성할 수 있으며, 코드 푸시 시 빌드 및 테스트를 자

동화하고, 안정성을 보장합니다. 유연한 트리거 조건으로 특정 브랜치, 태그, 이벤트에 따라 작업을 실행할 수 있습니다.

ArgoCD는 Kubernetes 환경에서 GitOps를 구현하는 CD(Continuous Deployment) 도구입니다. 애플리케이션의 선언적 배포를 지원하며, Git 리포지토리를 소스로 사용하여 Kubernetes 클러스터에 애플리케이션을 배포합니다. 이는 Git 리포지토리를 단일 진실의 원천(Single Source of Truth)으로 사용하여 클러스터 상태를 관리하는 GitOps를 지원합니다. 또한, 선언적 구성 파일을 기반으로 Kubernetes 클러스터 상태와 Git 리포지토리 상태를 동기화합니다. 배포 이력을 기록하고 이전 상태로 롤백이 가능하며, 애플리케이션 배포 상태와 클러스터 상태를 시각적으로 확인할 수 있는 UI를 제공하여 보다 편리합니다.

GitHub Actions와 ArgoCD를 함께 사용함으로써 코드에서 Kubernetes 배포까지의 전체 파이프라인을 완전 자동화할 수 있습니다. GitHub Actions는 테스트와 빌드를 자동화하고, ArgoCD는 배포를 책임지므로 개발 및 운영 간 협업을 강화할 수 있으며, 코드 변경 후 GitHub Actions를 통해 컨테이너 이미지를 자동으로 생성하고, ArgoCD가 Kubernetes에 이를 배포하여 변경 사항을 빠르게 적용할 수 있습니다.

GitHub Actions에서 변경된 애플리케이션 정의를 Git 리포지토리에 저장하고, ArgoCD가 이를 기반으로 클러스터 상태를 관리함으로써 선언적 방식의 안정성을 보장합니다. Kubernetes 클러스터와 GitOps를 기반으로 다중 환경(예: 스테이징, 프로덕션)에서 손쉽게 확장 가능하며, 고가용성을 유지합니다.

3.2.7. Monitoring Tools

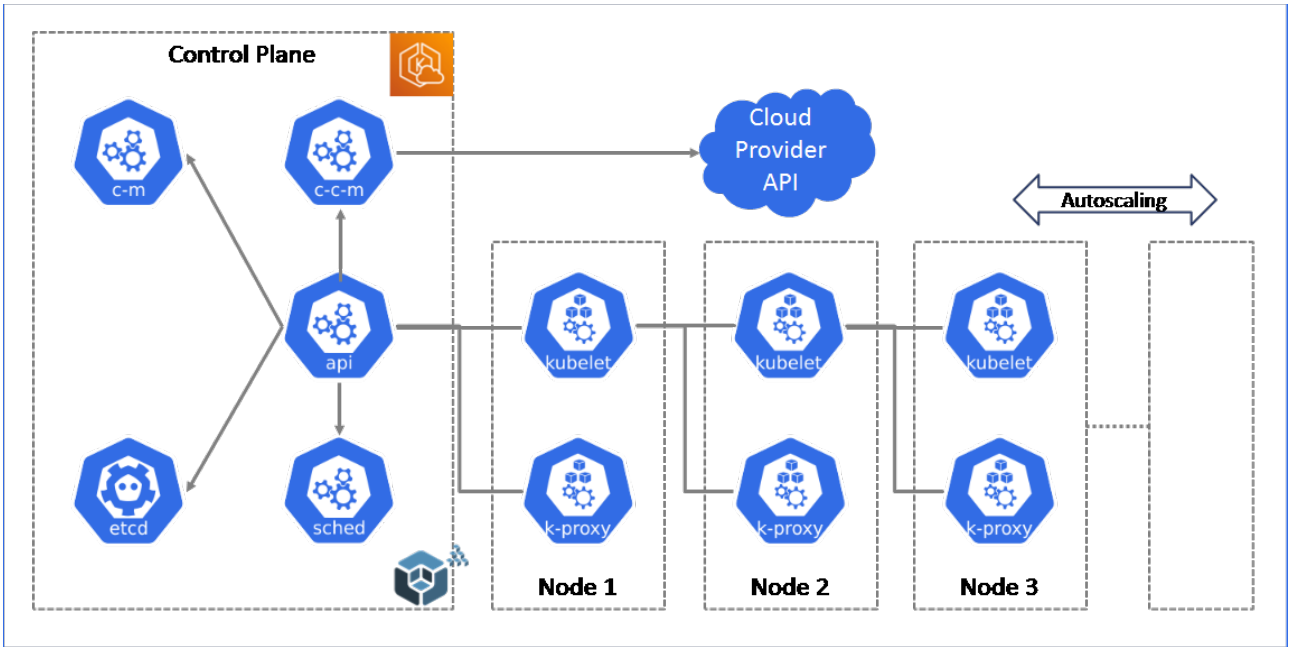
본 아키텍처에서는, Prometheus와 Grafana로 EKS, Redis, RDS 상태를 모니터링합니다. Zipkin과 OpenTelemetry로 분산 추적과 병목 구간을 분석합니다.

Prometheus와 Grafana는 시스템 성능을, Zipkin과 OpenTelemetry는 분산 서비스 호출 간 병목을 추적하여 엔드투엔드 모니터링을 제공합니다.

실시간 상태 모니터링과 추적 기능을 통해 문제 발생 시 신속한 대응과 해결이 가능하여 운영 효율성이 향상됩니다. 또한, 모든 도구가 확장 가능한 아키텍처를 기반으로 하여, 클러스터와 서비스의 확장에 쉽게 적응할 수 있으며 병목 및 장애를 줄이고 시스템의 안정성을 높이는 데 기여합니다. 마지막으로, 모든 레이어(EKS, Redis, RDS 등)에 대한 상태를 단일 대시보드와 추적 시스템에서 파악할 수 있어 운영의 투명성이 강화됩니다.

3.3. 컨테이너 오케스트레이션

개발/운영 환경에서의 클러스터 구성 컴포넌트는 다음과 같습니다.



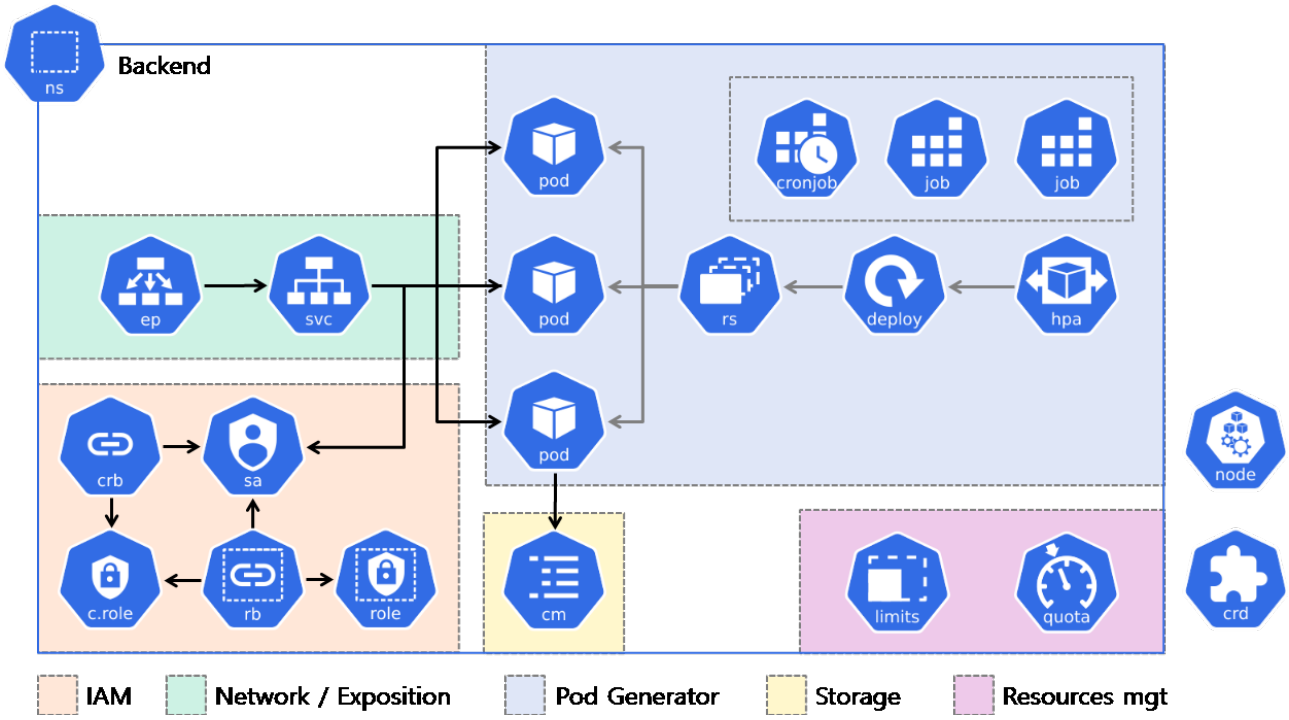
[그림 11] 개발/운영환경의 클러스터 구조

Control Plane

클러스터의 두뇌 역할을 하며 전체 쿠버네티스 시스템을 제어하고 관리합니다.
주요 구성 요소로는 API 서버, etcd, 스케줄러, 컨트롤러 매니저가 있으며, 이들이 협력하여 클러스터의 원하는 상태를 유지합니다. API 서버는 모든 클러스터 작업의 진입점 역할을 하고, etcd는 클러스터의 상태 정보를 저장하는 분산 데이터베이스로 활용됩니다.
개발/운영 모두 Control Plane 가 1대 이고, 개발환경에서는 Kubespray를, 운영환경에서는 공급사가 관리하는 EKS를 사용했습니다.

Node

AMD64 기반 프로세서로 동작하는 노드입니다.
각 노드는 컨테이너 런타임, kubelet, kube-proxy를 포함하며, 실제 워크로드를 실행하는 물리적/가상 머신입니다. Control Plane의 관리하에 Pod를 스케줄링하고 실행하며, 네트워크 프록시 기능을 제공합니다. 노드 리소스 모니터링과 상태 보고를 통해 클러스터의 안정적인 운영을 지원합니다. Desired는 3이며, 최대 5대까지 생성되도록 설계했습니다.



[그림 12] 애플리케이션 - Backend 네임스페이스 구성

IAM

클러스터 내 리소스에 대한 접근 제어와 보안 정책을 관리합니다.
파드에 적재된 Spring Cloud 제품군(Gateway, Service Discovery)과 호환되도록 구성했습니다.

Network / Exposition

네트워크 정책과 서비스 노출을 관리합니다.
Gateway가 담긴 파드의 서비스 타입을 로드밸런서로 설정하고,
ELB(운영), MetalLB(로컬)로 외부 IP주소를 생성했습니다.

Pod Generator

컨테이너화된 애플리케이션의 배포를 자동화하는 컴포넌트입니다.
Probe, Rolling Update, HPA, Resources limit 설정해 고가용성을 보장했습니다.

Storage

저장소와 데이터 관리를 담당하는 컴포넌트입니다.
애플리케이션의 환경변수를 제공하는 목적으로 사용했습니다.

Resources Management

클러스터 리소스의 할당과 관리를 담당하는 컴포넌트입니다.
다른 네임스페이스의 자원을 침범해 2차 피해가 가지 않도록 설계했습니다.

3.4. 프로비저닝

인프라를 IaC Terraform으로 프로비저닝했습니다.

```
tf/
├── addon
├── aurora
├── bastion
├── iamrole
├── kubeconfig
├── main
├── metrics-server
├── outputs
├── rds
├── redis
├── s3
├── providers
├── terraform.tfvars
└── variables
```

[그림 13] 민감 정보는 tfvars파일에 저장, variables파일에서 변수 관리

```hcl
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "~> 5.0"

  name = local.cluster_vpc_name

  cidr = var.network_cidr
  azs  = slice(data.aws_availability_zones.available.names, 0, 3)

  # Private과 Public 서브넷을 설정합니다.
  private_subnets = [cidrsubnet(var.network_cidr, 8, 1), cidrsubnet(var.network_cidr, 8, 2)]
  public_subnets  = [cidrsubnet(var.network_cidr, 8, 101), cidrsubnet(var.network_cidr, 8, 102)]

  enable_nat_gateway   = true
  single_nat_gateway   = true
  enable_dns_hostnames = true
}

# 별도의 internal_subnets을 정의하여 VPC에 연결
resource "aws_subnet" "internal_subnet" {
  count = length(slice(data.aws_availability_zones.available.names, 0, 3))

  vpc_id                  = module.vpc.vpc_id
  cidr_block              = cidrsubnet(var.network_cidr, 8, 3 + count.index)
  availability_zone       = data.aws_availability_zones.available.names[count.index]
  map_public_ip_on_launch = false

  tags = {
    Name = "${local.cluster_vpc_name}-internal-${count.index + 1}"
  }
}

# 내부 서브넷들을 리스트로 참조
locals {
  internal_subnets = aws_subnet.internal_subnet[*].id
}
```

[그림 14] Public, Private, Internal Subnet 생성 및 지정

```hcl
locals {
  # OIDC Provider의 ARN과 CA Thumbprint를 동적으로 생성
  oidc_provider_url = data.aws_eks_cluster.cluster.identity[0].oidc[0].issuer
  eks_ca_thumbprint = substr(trimspace(data.aws_eks_cluster.cluster.certificate_authority[0].data), 0, 40)
}

# 기존 OIDC Provider가 있는지 확인
data "aws_iam_openid_connect_provider" "existing_oidc" {
  url = local.oidc_provider_url
}

# 존재하지 않을 경우 OIDC Provider 생성
resource "aws_iam_openid_connect_provider" "oidc_provider" {
  count           = length(data.aws_iam_openid_connect_provider.existing_oidc.arn) == 0 ? 1 : 0
  client_id_list  = ["sts.amazonaws.com"]
  thumbprint_list = [local.eks_ca_thumbprint]
  url             = local.oidc_provider_url

  lifecycle {
    prevent_destroy = false
    ignore_changes  = [url, thumbprint_list]
  }
}
```

[그림 15] OIDC 관련 설정

OIDC(OpenID Connect)는 인증 프로토콜로, 사용자의 신원을 안전하게 확인하기 위해 설계된 표준 프로토콜입니다. OIDC는 사용자 인증 및 프로필 정보를 안전하게 애플리케이션에 제공할 수 있도록 설계되었습니다.

```hcl
# Bastion 인스턴스에 접근할 SSH 키 페어 이름
variable "key_pair_name" {
  description = "SSH key pair name for accessing the bastion instance"
  type        = string
  default     = "cloudscape-keypair"
}

# Key Pair Private Key 생성 및 저장
resource "tls_private_key" "bastion_private_key" {
  algorithm = "RSA"
  rsa_bits  = 2048
}

resource "aws_key_pair" "bastion_generated_key" {
  key_name   = var.key_pair_name
  public_key = tls_private_key.bastion_private_key.public_key_openssh
}
```

[그림 16] Bastion Host 생성 후 키페어 생성&출력&저장

```
resource "aws_rds_cluster" "aurora_cluster2" {
  cluster_identifier     = "aurora-cluster2"
  engine                 = "aurora-mysql"
  engine_version         = "8.0.mysql_aurora.3.04.1"
  database_name          = var.rds_db_name2
  master_username        = var.rds_username
  master_password        = var.rds_password
  db_subnet_group_name   = aws_db_subnet_group.rds_subnet_group.name
  skip_final_snapshot    = true
  port = 3316

  vpc_security_group_ids = [aws_security_group.aurora_sg.id]

  db_cluster_parameter_group_name = aws_rds_cluster_parameter_group.aurora_cluster_param_group.name

  backup_retention_period = 3
  preferred_backup_window = "03:00-04:00"

  tags = {
    Name = "aurora-cluster"
  }
}

# Primary Instance
resource "aws_rds_cluster_instance" "aurora_primary2" {
  identifier         = "aurora-primary2"
  cluster_identifier = aws_rds_cluster.aurora_cluster2.id
  instance_class     = "db.t3.medium"  # Modify according to your needs
  engine             = aws_rds_cluster.aurora_cluster2.engine
  engine_version     = aws_rds_cluster.aurora_cluster2.engine_version
  auto_minor_version_upgrade = false
  publicly_accessible = true


  tags = {
    Name = "aurora-primary"
  }
}

# Replica Instance
resource "aws_rds_cluster_instance" "aurora_replica2" {
  identifier         = "aurora-replica2"
  cluster_identifier = aws_rds_cluster.aurora_cluster2.id
  instance_class     = "db.t3.medium"  # Modify according to your needs
  engine             = aws_rds_cluster.aurora_cluster2.engine
  engine_version     = aws_rds_cluster.aurora_cluster2.engine_version
  auto_minor_version_upgrade = false
  publicly_accessible = true


  tags = {
    Name = "aurora-replica"
  }
}
```

[그림 17] DB 접속 정보

```
# EKS 클러스터와 노드 그룹의 생성 완료를 기다림

data "aws_iam_role" "node_group_role" {
  depends_on = [null_resource.wait_for_node_group]
  name = module.eks.eks_managed_node_groups["main_group"].iam_role_name
}

resource "aws_iam_policy_attachment" "autoscaler_policy_attach" {
  name       = "cluster-autoscaler-attach"
  roles        = [data.aws_iam_role.node_group_role.name]
  policy_arn = aws_iam_policy.cluster_autoscaler_policy.arn

  depends_on = [null_resource.wait_for_node_group]
} # 노드 그룹의 IAM 역할에 클러스터 오토스케일러 관련 정책을 연결
```

[그림 18] IAM 정책 생성 및 연결, 콘솔과 IaC와의 정책 연결

```
resource "helm_release" "cluster_autoscaler" {
  name          = "cluster-autoscaler"
  namespace     = "kube-system"
  chart         = "cluster-autoscaler"
  repository    = "https://kubernetes.github.io/autoscaler"
  version       = "9.29.0" # 최신 버전 확인 가능
}
```

[그림 19] Helm 과 Cluster Autoscaler 추가

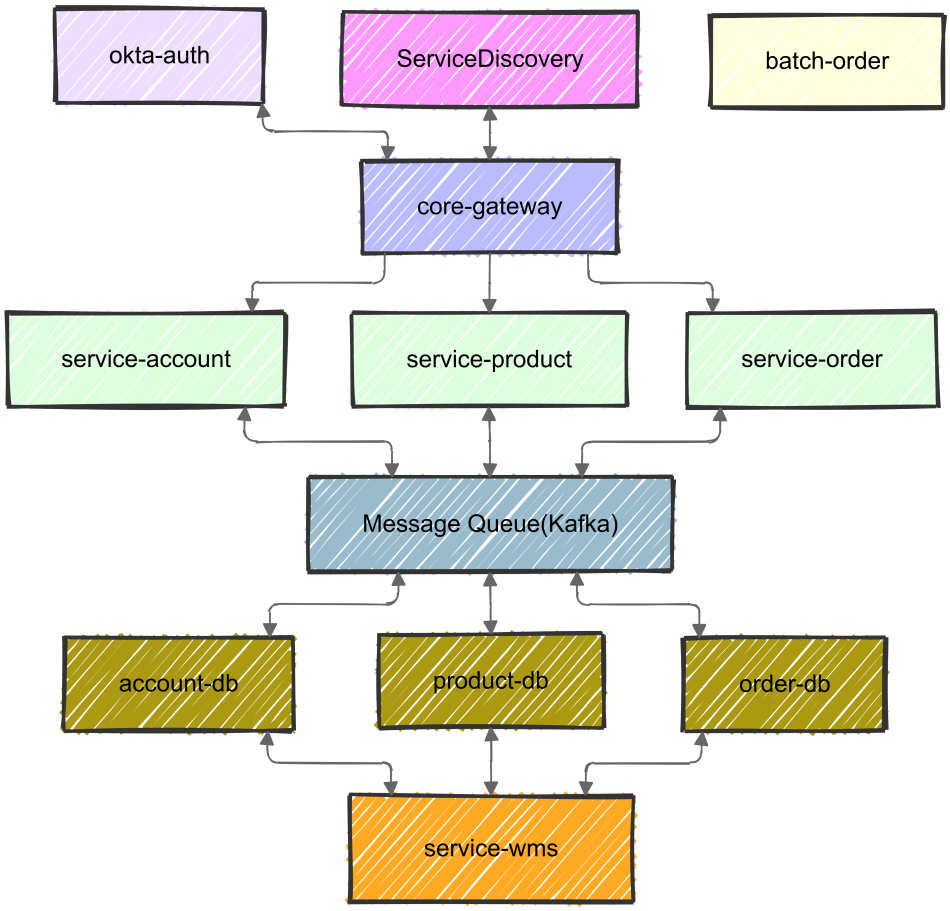
```
# ElastiCache Redis 레플리케이션 그룹
resource "aws_elasticache_replication_group" "redis" {
  description = "for cluster"
  replication_group_id       = "redis-cluster"
  node_type                  = "cache.t3.micro"
  port                       = var.redis_port1
  parameter_group_name       = "default.redis7.cluster.on"
  automatic_failover_enabled = true
  engine                     = "redis"
  engine_version             = "7.1"
  subnet_group_name          = aws_elasticache_subnet_group.redis-subnet-group.name
  security_group_ids         = [aws_security_group.redis.id]
  multi_az_enabled = false

  num_cache_clusters         = 6

  tags = {
    Name = "redis-cluster"
  }
}
```

[그림 20] Redis Cluster 구성

4. 애플리케이션 구축

4.1. 마이크로서비스 아키텍처 설계

4.1.1. Market 서비스 구성

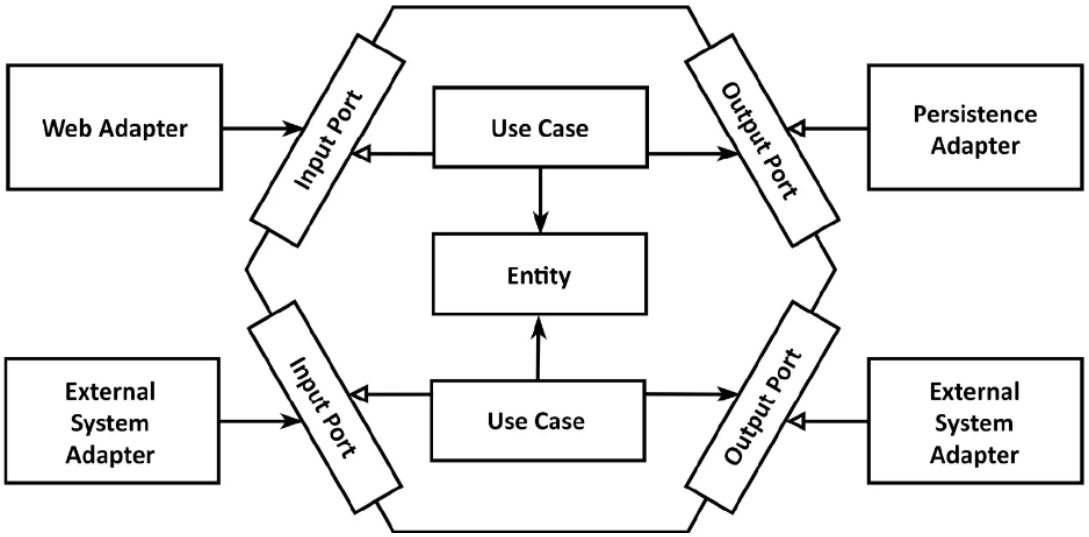


[그림 21] 애플리케이션 구성도

- core-gateway: 모든 클라이언트 요청의 진입점 역할을 하는 API Gateway 서버로, 라우팅, 로드밸런싱, 인증/인가를 담당
- service-account: 회원 관리, 인증 등 사용자 관련 기능을 처리하는 회원 서버
- service-product: 상품 정보, 재고 관리 등 상품 관련 기능을 처리하는 상품 서버
- service-order: 주문 처리, 배송 상태 관리 등 주문 관련 기능을 처리하는 주문 서버
- batch-order: 지연된 배송 처리, 최소된 주문 삭제 등 배치 작업을 수행하는 배치 서버
- Service Discovery: 서비스 등록, 검색, 상태 모니터링을 담당하는 서비스 디스커버리로 마이크로서비스 간 동적 서비스 검색을 지원
- Message Queue: 서비스 간 비동기 통신을 위한 메시지 브로커로, 이벤트 기반 아키텍처를 지원하고 서비스 간 느슨한 결합을 가능하게 함
- okta-auth: 외부 인증 제공자로서 사용자 인증(OIDC) 및 권한 관리를 담당

4.1.2. 헥사고날 프로젝트 구조

헥사고날 아키텍처를 적용해 MSA 환경에서 각 마이크로서비스의 독립성을 강화했습니다. 도메인 로직의 테스트 용이성과 유지보수성이 향상되며, 기술 스택 변경에도 유연합니다.



[그림 22] 헥사고날 아키텍처 기본 구조

Application 계층은 비즈니스 로직을, Domain 계층은 핵심 도메인 모델을, Infrastructure 계층은 외부 시스템과의 통합을 담당하여 각 계층의 책임을 분리했습니다.

```
market/                                  # 루트 패키지
├── batch-order                          # 배치 주문 서비스, 대량의 주문을 처리하는 서비스
├── core-discovery                       # 서비스 디스커버리, 마이크로서비스의 위치를 찾는 서비스
├── core-gateway                         # API 게이트웨이, 클라이언트 요청을 적절한 서비스로 라우팅하는 역할
├── service-account/                     # 계정 서비스, 사용자 계정 관련 기능을 제공
│   ├── application/                     # 애플리케이션 레이어, 비즈니스 로직을 포함
│   │   ├── dto/                         # 데이터 전송 객체 (DTO), 계층 간 데이터 교환을 위한 객체
│   │   │   ├── request                  # 요청 DTO, 클라이언트 요청 데이터를 담는 객체
│   │   │   └── response                 # 응답 DTO, 서버 응답 데이터를 담는 객체
│   │   ├── port/                        # 포트 레이어, 애플리케이션의 입출력 인터페이스를 정의
│   │   │   ├── input                    # 입력 포트, 애플리케이션으로 들어오는 인터페이스
│   │   │   └── output                   # 출력 포트, 애플리케이션에서 나가는 인터페이스
│   │   └── service                      # 서비스 레이어, 비즈니스 로직을 구현하는 서비스 클래스
│   ├── domain/                          # 도메인 레이어, 핵심 비즈니스 개념과 규칙을 포함
│   │   ├── event                        # 도메인 이벤트, 도메인 내에서 발생하는 중요한 사건
│   │   ├── exception                    # 도메인 예외, 도메인 내에서 발생하는 예외 상황
│   │   ├── model                        # 도메인 모델, 비즈니스 개념을 표현하는 객체
│   │   └── repository                   # 도메인 리포지토리, 도메인 객체의 영속성을 관리
│   └── infrastructure/                  # 인프라 레이어, 외부 시스템과의 통신을 담당
│       ├── adapter/                     # 어댑터, 외부 시스템과의 통신을 위한 어댑터
│       │   ├── input/                   # 입력 어댑터, 외부에서 들어오는 요청을 처리
│       │   │   ├── controller           # 컨트롤러, HTTP 요청을 처리하는 클래스
│       │   │   ├── converter            # 변환기, 데이터 형식을 변환하는 클래스
│       │   │   ├── messaging.consumer   # 메시징 컨슈머, 메시지 큐에서 메시지를 소비하는 클래스
│       │   │   └── validator            # 검증기, 입력 데이터를 검증하는 클래스
│       │   └── output/                  # 출력 어댑터, 외부로 나가는 요청을 처리
│       │       ├── feign                # Feign 클라이언트, HTTP 클라이언트를 생성하는 라이브러리
│       │       ├── messaging.producer   # 메시징 프로듀서, 메시지 큐에 메시지를 보내는 클래스
│       │       └── persistence          # 영속성 어댑터, 데이터베이스와의 통신을 담당
│       └── configuration                # 설정, 애플리케이션 설정 클래스
├── service-order                        # 주문 서비스, 주문 관련 기능을 제공
└── service-product                      # 제품 서비스, 제품 관련 기능을 제공
```

[그림 23] 헥사고날 아키텍처가 반영된 프로젝트 구조

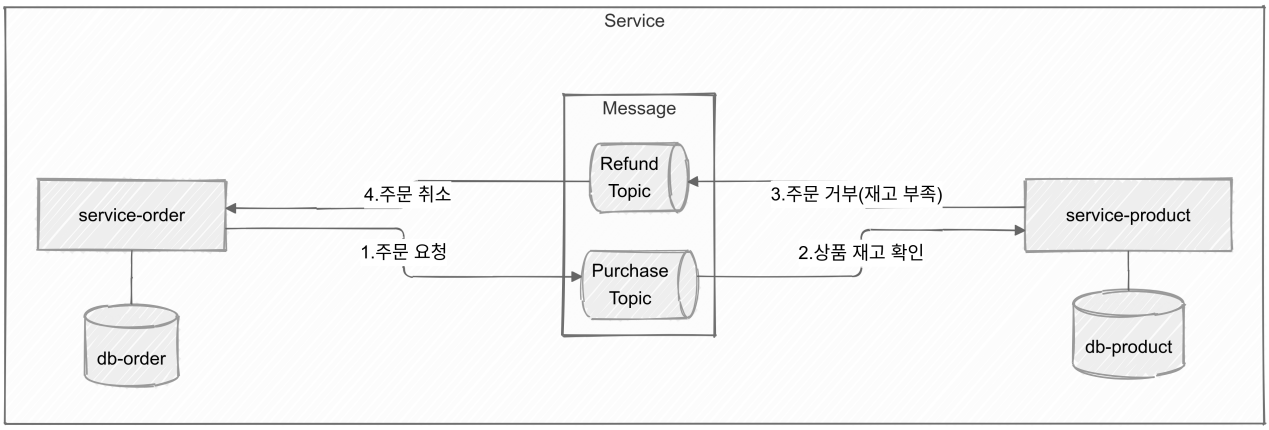
4.1.3. 로드밸런서 설정

```yaml
spring:
  cloud:
    discovery:
      enabled: true
    gateway:
      mvc:
        routes:
          - id: service-account
            uri: lb://service-account
            predicates:
              - Path=/account/**
              - Method=GET,POST,PUT,PATCH,DELETE
            filters:
              - RewritePath=/account/?(?<segment>.*), /$\{segment}
          - id: service-order
            uri: lb://service-order
            predicates:
              - Path=/order/**
              - Method=GET,POST,PUT,PATCH,DELETE
            filters:
              - RewritePath=/order/?(?<segment>.*), /$\{segment}
          - id: service-product
            uri: lb://service-product
            predicates:
              - Path=/product/**
              - Method=GET,POST,PUT,PATCH,DELETE
            filters:
              - RewritePath=/product/?(?<segment>.*), /$\{segment}
```

[그림 24] Spring Gateway 서버의 라우팅 설정

4.1.4. 메시지 기반 서비스 간 통신

이벤트 드리븐 아키텍처를 반영해 분산 시스템 간 느슨한 결합도를 제공하고, 시스템 간의 실시간 데이터 동기화와 비동기 처리를 가능하게 했습니다. 이를 통해 주문, 재고 처리와 같은 프로세스를 안정적으로 처리할 수 있습니다.



[그림 25] 메시지 기반 통신 방식. 주문 프로세스 구조

4.2. 성능 최적화

4.2.1. CQRS 패턴 적용

시스템의 명령(Command)와 조회(Query)작업을 분리하는 아키텍처 패턴을 적용했습니다.
읽기 작업이 빈번한 시스템을 고려해 AWS Aurora Cluster 내 Autoscaling을 이용하여 사용자 수가 증가함에 따라 DB가 증설되어 효과적으로 처리할 수 있게 되었습니다.

```yaml
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    hikari:
      source:
        url: jdbc:mysql://${PRODUCT_DB_HOST}:${PRODUCT_DB_PORT}/${PRODUCT_DB_DATABASE}
        username: ${PRODUCT_DB_USER}
        password: ${PRODUCT_DB_PASSWORD}
      replica:
        url: jdbc:mysql://${PRODUCT_DB_REPLICA_HOST}:${PRODUCT_DB_REPLICA_PORT}/${PRODUCT_DB_REPLICA_DATABASE}
        username: ${PRODUCT_DB_REPLICA_USER}
        password: ${PRODUCT_DB_REPLICA_PASSWORD}
      maximum-pool-size: ${DB_POOL_SIZE}
```

[그림 26] CQRS 적용 설정

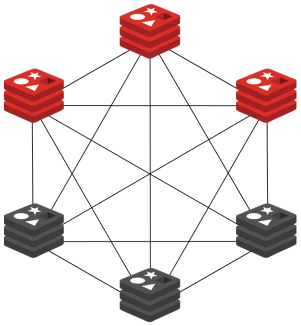
```java
public interface ProductCommand {
    Product create(CreateProductRequest request);

    Product update(UpdateProductRequest request);

    boolean delete(DeleteProductRequest request);

    Product updateStock(UpdateProductStockDecreaseRequest request);
}
```

```java
public interface ProductQuery {
    Product read(ReadProductRequest query);

    Page<Product> read(ReadProductsRequest query);
}
```

[그림 27] CQRS 사용 예시

4.2.2. 서비스 응답 시간 개선(캐시 도입)

Redis Cluster의 자동 샤딩 기능을 활용하여 데이터를 여러 노드에 분산 저장하고, 각 샤드별 마스터-슬레이브 복제를 구성하여 장애 상황에서도 서비스의 연속성을 보장했습니다.



[그림 28] Redis Cluster

종류	값
문자열	accounts:: Page request [number: 0, ...] orders:: Page request [number: 0, ...] product:: Page request [number: 0, ...] Dspring:session:expires:a1449ceb-xxxx-...
세트	spring:session:expirations:1732816... spring:session:index:org.springframework.xxx...
해시테이블	spring:session:sessions:a1449-xxxx-....

다른 구성 시스템과의 비교표입니다.

구분	Redis Cluster	Redis Sentinel	Redis Standalone
주요 목적	데이터 분산 및 수평적 확장	고가용성 및 자동 페일오버	단일 노드 운영
구성 방식	여러 마스터 노드에 데이터 분산	단일 마스터-슬레이브 구성	단일 노드
확장성	수평적 확장 용이(샤딩)	수직적 확장만 가능	수직적 확장만 가능
데이터 관리	자동 샤딩을 통한 분산 저장	마스터 노드에 전체 데이터 저장	단일 노드에 전체 데이터 저장
복제 방식	각 샤드별 마스터-슬레이브 복제	전체 데이터의 마스터-슬레이브 복제	복제 없음
사용 사례	대용량 데이터 처리가 필요한 경우	단순 고가용성이 필요한 경우	소규모 데이터, 개발 환경

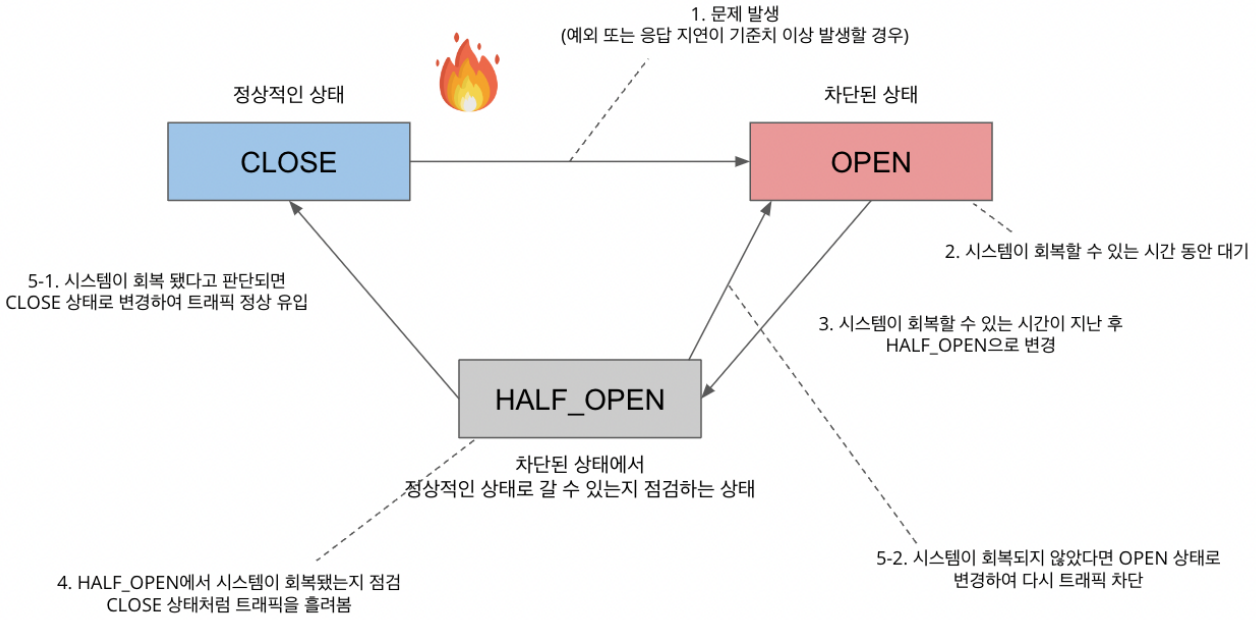
[그림 29] 스트레스테스트 P99에서의 캐시 적용 전(좌)과 후(우) 성능 차이

4.3. 장애 허용 시스템 구현

4.3.1. 서킷브레이커(CircuitBreaker) 패턴 구현

서킷 브레이커 패턴을 구현하여 마이크로서비스 간의 장애 전파를 방지하고 시스템의 복원력을 향상시켰습니다. Resilience4j의 CircuitBreaker 기능을 활용하여 서비스 호출의 실패율이 임계값을 초과할 경우 자동으로 회로를 차단하고, Fallback 메커니즘을 통해 대체 응답을 제공하도록 구성했습니다.

또한 슬라이딩 윈도우 방식으로 실패율을 모니터링하여 시스템의 안정성을 실시간으로 관리할 수 있도록 했습니다.



[그림 30] 서킷브레이커 기본 동작 구조(출처. 인프런)

- Gateway의 라우터에 서킷브레이커를 설정해 하위 서비스에 장애 발생시 Fallback 주소로 우회되도록 설계했습니다.

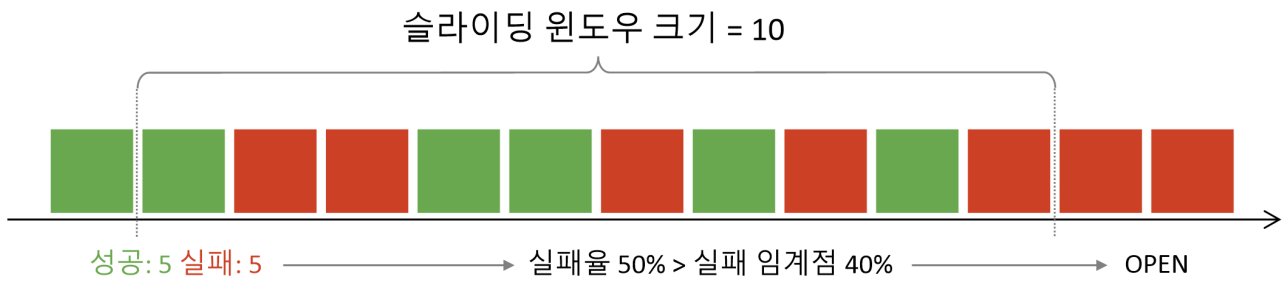
```java
@Configuration
public class RouterConfig {

    ...
    @Bean
    public RouterFunction<ServerResponse> productServiceRoute() {
        return RouterFunctions.route()
                .route(GatewayRequestPredicates.path("/product/**"), http())
                .filter(lb("service-product"))
                .filter(rewritePath("/product/?(?<segment>.*)", "/$\\{segment}"))
                .filter(circuitBreaker("productCircuitBreaker", URI.create("forward:/fallback/product")))
                .build();
    }

}
```

[그림 31] Gateway 내 Router에 서킷브레이커 설정하는 코드

설정값은 다음과 같습니다.

```yaml
resilience4j:
  circuitbreaker:
    configs:
      default: # 기본 구성 이름
        registerHealthIndicator: true  # 애플리케이션의 헬스 체크에 서킷 브레이커 상태를 추가하여 모니터링 가능
        slidingWindowType: COUNT_BASED  # 슬라이딩 윈도우의 타입을 호출 수 기반(COUNT_BASED)으로 설정
        slidingWindowSize: 10  # 슬라이딩 윈도우의 크기를 N번의 호출로 설정
        minimumNumberOfCalls: 5  # 서킷 브레이커가 동작하기 위해 필요한 최소한의 호출 수를 5로 설정
        slowCallRateThreshold: 60  # 느린 호출의 비율이 이 임계값을 초과하면 서킷 브레이커가 동작
        slowCallDurationThreshold: 3000  # 느린 호출의 기준 시간(밀리초)으로, N초 이상 걸리면 느린 호출로 간주
        failureRateThreshold: 40  # 실패율이 이 임계값을 초과하면 서킷 브레이커가 동작
        permittedNumberOfCallsInHalfOpenState: 5  # 서킷 브레이커가 Half-open 상태에서 허용하는 최대 호출 수
        waitDurationInOpenState: 20s  # Open 상태에서 Half-open 상태로 전환되기 전에 대기하는 시간
  timelimiter:
    configs:
      default:
        timeoutDuration: 5s
        cancelRunningFuture: false
```

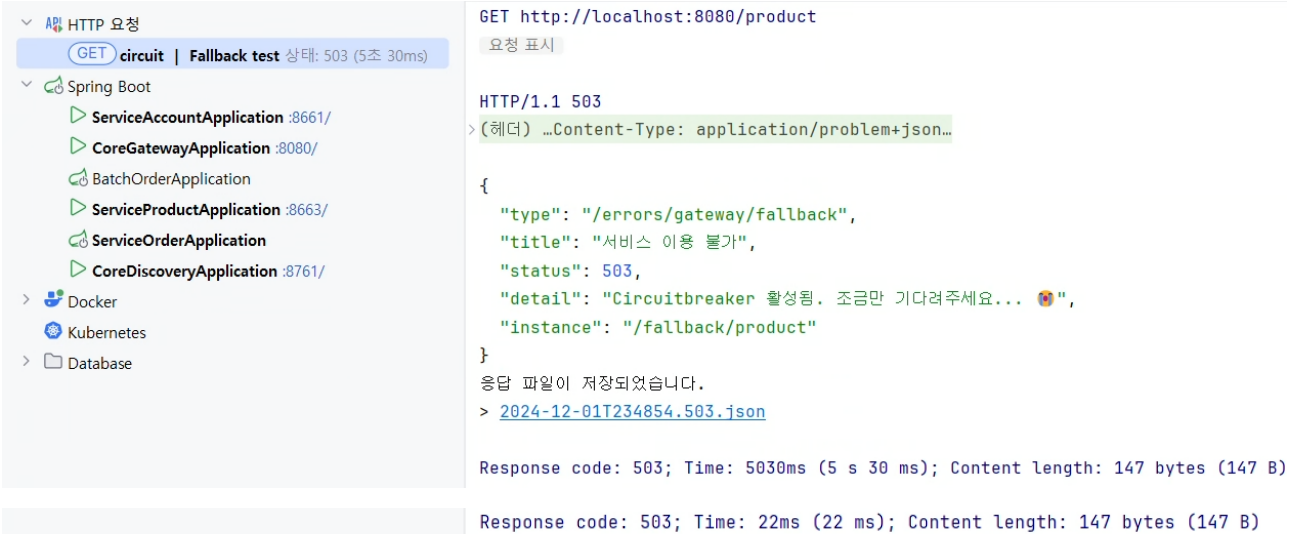
[그림 32] 서킷브레이커 설정값



[그림 33] 서킷브레이커 슬라이딩 윈도우 예시

10회 중 5번 실패한 경우 임계점을 넘어 서킷브레이커가 OPEN 됩니다.
20초 뒤 HALF_OPEN 상태로 전환되고 헬스체크에 정상적으로 응답할 경우 CLOSE 됩니다.

서버 내 Product DB 장애 발생시 Gateway에서 응답한 결과입니다.



[그림 34] 서킷브레이커 OPEN 전 timelimiter 5s로 폴백 되고, 호출 횟수 충족 이후 활성됨

4.3.2. 페일오버 반영(Retry)

DB에서 데이터를 가져오고 이를 캐시에 저장해 응답속도를 개선하는 기능이 있는데,
만약에 도중에 캐시서버가 장애가 나 사용 불가능한 상태가 된다면 DB는 정상 동작함에도 불구하고 정상적인 응답을 하지 못하게 됩니다.
이 경우에 Retry를 사용하면 캐시 서버에 장애가 발생했을 때 일정 횟수만큼 재시도를 수행하고, 여러 번의 시도 후에도 실패할 경우 DB에서 직접 데이터를 조회하는 대체 로직을 실행할 수 있습니다.

```
1    @Override
2    @Cacheable(value = "order", key = "#query.id()", cacheManager = "orderCacheManager")
3    @Retryable(retryFor = {RedisConnectionException.class}, maxAttempts = 1, backoff = @Backoff(delay = 100))
4    public Order read(ReadOrderRequest query) {
5        return getOrder(query);
6    }
7
8    private Order getOrder(ReadOrderRequest query) {
9        return port.get(query.id())
10               .orElseThrow(BizException.NoneExists::new);
11   }
12
13   @Recover
14   @SuppressWarnings("unused")
15   public Order readWithoutCache(ReadOrderRequest query) {
16       return getOrder(query)
17   }
```

[그림 35] 캐시 서버가 동작하지 않을 때 DB액세스를 시도하는 코드

이러한 재시도 메커니즘은 @Retryable 어노테이션을 통해 구현되며,
RedisConnectionException이 발생할 경우 최대 1회 재시도를 수행하고 100ms의 지연 시간을 두도록 설정되어 있습니다. 재시도 실패 시 @Recover 어노테이션이 지정된 메서드가 실행되어 캐시 없이 직접 DB에서 데이터를 조회하게 됩니다.

4.4. 모니터링 및 로깅

4.4.1. 종합 모니터링

시스템의 종합적인 정보 시각화를 위해 Grafana Dashboard를 이용했습니다.

4.4.2. 분산 추적 시스템 구현

분산 시스템에서의 요청 추적을 위해 Zipkin을 도입했습니다.
각 서비스 간 호출의 전체 흐름을 추적하고, 각 단계별 지연 시간과 병목 구간을 식별할 수 있도록 구현했습니다. 특히 TraceID와 SpanID를 활용하여 요청의 전체 여정을 시각화하고, 문제 발생 시 신속한 원인 분석이 가능하도록 했습니다.

4.4.3. 메트릭 수집

Prometheus, OpenTelemetry로 JVM 메트릭, 애플리케이션 성능 지표, 인프라 상태 등을 실시간으로 수집하고 시각화하여 시스템의 건강 상태를 지속적으로 관찰합니다. 특히 커스텀 메트릭을 정의하여 비즈니스 관련 지표도 함께 모니터링해 시스템과 비즈니스 성과를 통합적으로 분석할 수 있도록 구현했습니다.

4.4.4. 로그 집중화

각 서비스에서 생성되는 모든 로그는 중앙 집중식 로깅 시스템으로 수집되어 실시간으로 분석되고 저장됩니다. Promtail이 각 서비스(account, product, order)의 로그를 수집하고 Loki로 전송하며, Grafana를 통해 로그를 시각화하고 검색할 수 있습니다. 이를 통해 문제 발생 시 신속한 원인 분석과 대응이 가능하며, 로그 기반의 알림 시스템을 구축하여 잠재적인 문제를 사전에 감지할 수 있습니다.

4.5. 보안 구현

4.5.1. 인증/인가 시스템 구축

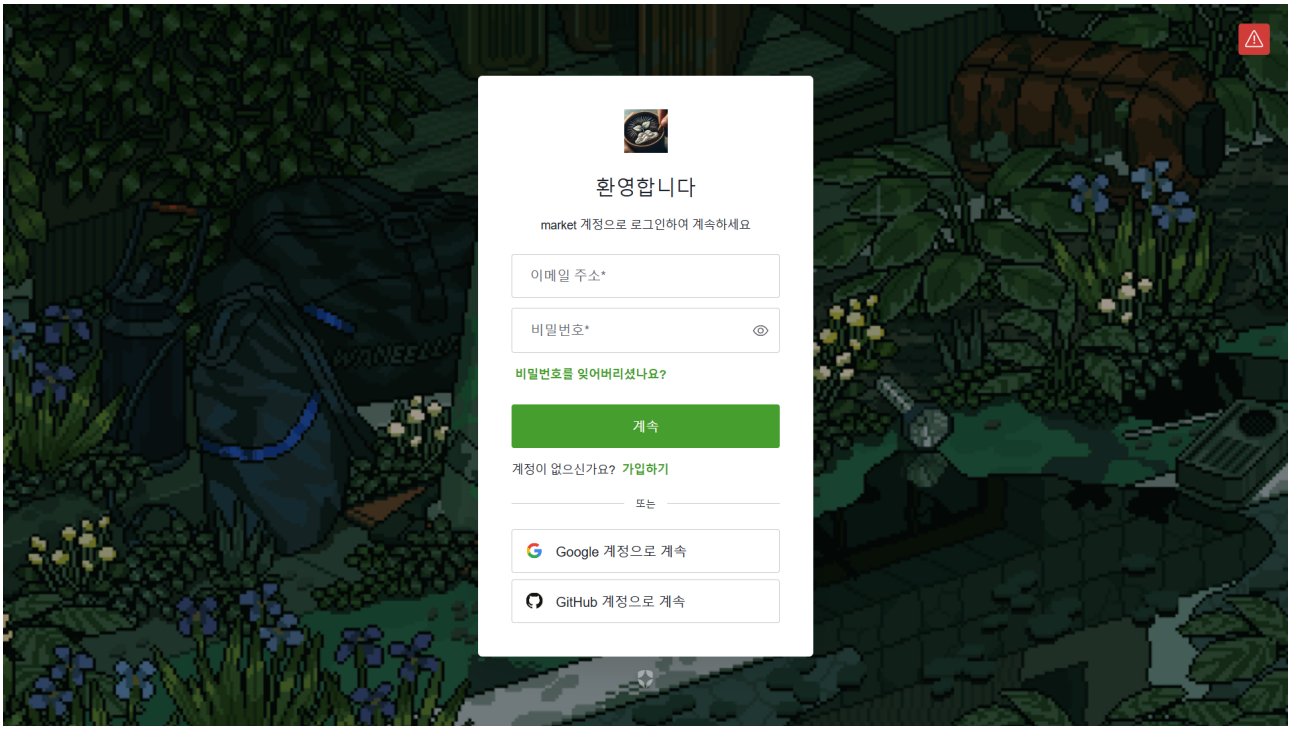
Okta OIDC를 이용해 통합 로그인(+ 소셜 로그인)을 제공합니다.

```java
@EnableWebSecurity
@RequiredArgsConstructor
@Configuration
public class SecurityConfig {

    private final CustomOidcUserService customOidcUserService;
    private final SessionRegistry sessionRegistry;

    @Value("${okta.oauth2.issuer}")
    private String issuer;

    @Value("${okta.oauth2.client-id}")
    private String clientId;

    @Bean
    public WebSecurityCustomizer webSecurityCustomizer() {
        return (web) -> web.ignoring().requestMatchers("/account/**", "/product/**", "order/**");
    }

    @Bean
    public SecurityFilterChain configure(HttpSecurity http) throws Exception {

        http.oauth2Login(oauth2Login -> oauth2Login
                .loginPage("/oauth2/authorization/okta")
                .userInfoEndpoint(userInfoEndpoint -> userInfoEndpoint.oidcUserService(customOidcUserService))
        );

        http.formLogin(AbstractHttpConfigurer::disable);
        http.httpBasic(AbstractHttpConfigurer::disable);

        http.logout(logout -> logout.logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
                .invalidateHttpSession(true)
                .deleteCookies("SESSION")
                .clearAuthentication(true)
                .addLogoutHandler(logoutHandler()));

        http.sessionManagement(session -> {
            session
                    .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
                    .maximumSessions(1)
                    .maxSessionsPreventsLogin(false)
                    .sessionRegistry(sessionRegistry);
        });

        return http.build();
    }

    private LogoutHandler logoutHandler() {
        return (request, response, authentication) -> {
            try {
                if (!response.isCommitted()) {
                    String baseUrl = ServletUriComponentsBuilder.fromCurrentContextPath()
                            .build()
                            .toUriString();
                    response.sendRedirect(issuer + "v2/logout?client_id=" + clientId + "&returnTo=" + baseUrl);
                }
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        };
    }
}
```

[그림 36] 세션 + OIDC 인증 설정

4.5.2. 세션 기반 인증

Redis Session을 사용해 분산 애플리케이션에서의 인증을 구현하였습니다.
Redis를 세션 저장소로 활용함으로써 여러 서버 간에 세션 정보를 공유하고, 확장성과 고가용성을 확보할 수 있었습니다.
또한 세션 만료 시간을 정해 용량 제약을 극복했습니다.
기 로그인한 사용자가 다른 클라이언트에서 로그인하면 이전 로그인 정보를 무효화 후 세션을 다시 등록하도록 설계했습니다.

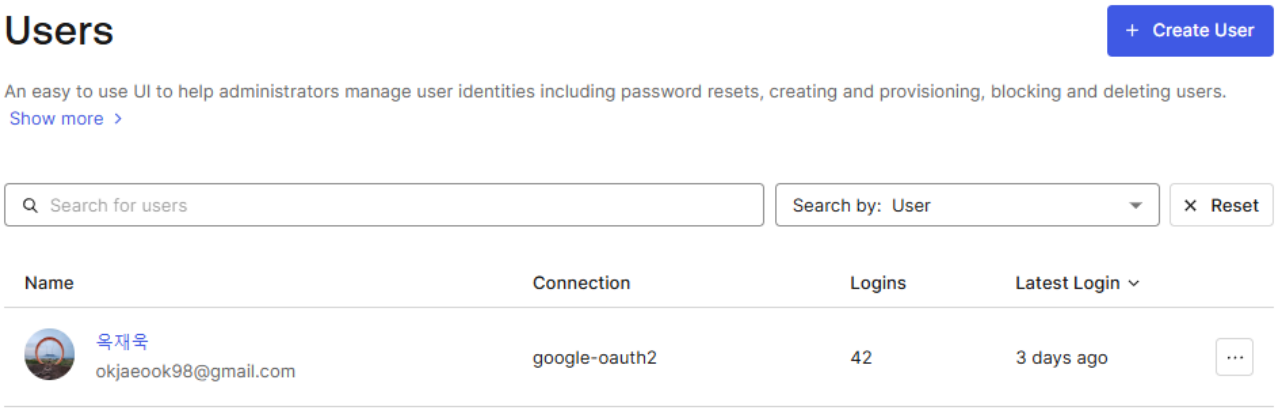


[그림 37] Okta 인증서버에서 제공하는 로그인 템플릿. 소셜 로그인 지원

[그림 38] 로그인 후 세션 정보가 Redis 서버에 저장됨

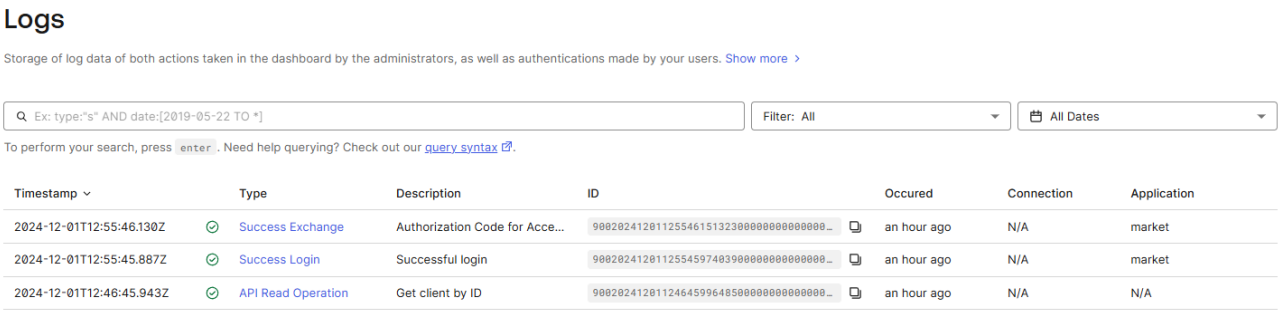
4.5.3. 인증 DB와 API Account DB와의 분리

민감 정보는 Okta에 저장되고, 그 외 정보는 Account DB에 제공됩니다.



[그림 39] Okta 내 DB에서 민감정보를 가지고 있음

인증 정보와 일반 사용자 정보를 분리함으로써 보안성을 강화하고, 각 데이터베이스의 역할을 명확히 구분하여 시스템의 유지보수성을 향상시켰습니다. Okta에서는 비밀번호, 인증 토큰 등의 민감한 정보를 안전하게 관리하고, Account DB에서는 사용자의 프로필, 설정 등 일반적인 정보를 효율적으로 관리할 수 있게 되었습니다.



[그림 40] Okta 인증 서버에 성공적으로 인증됨

4.6. 문서화

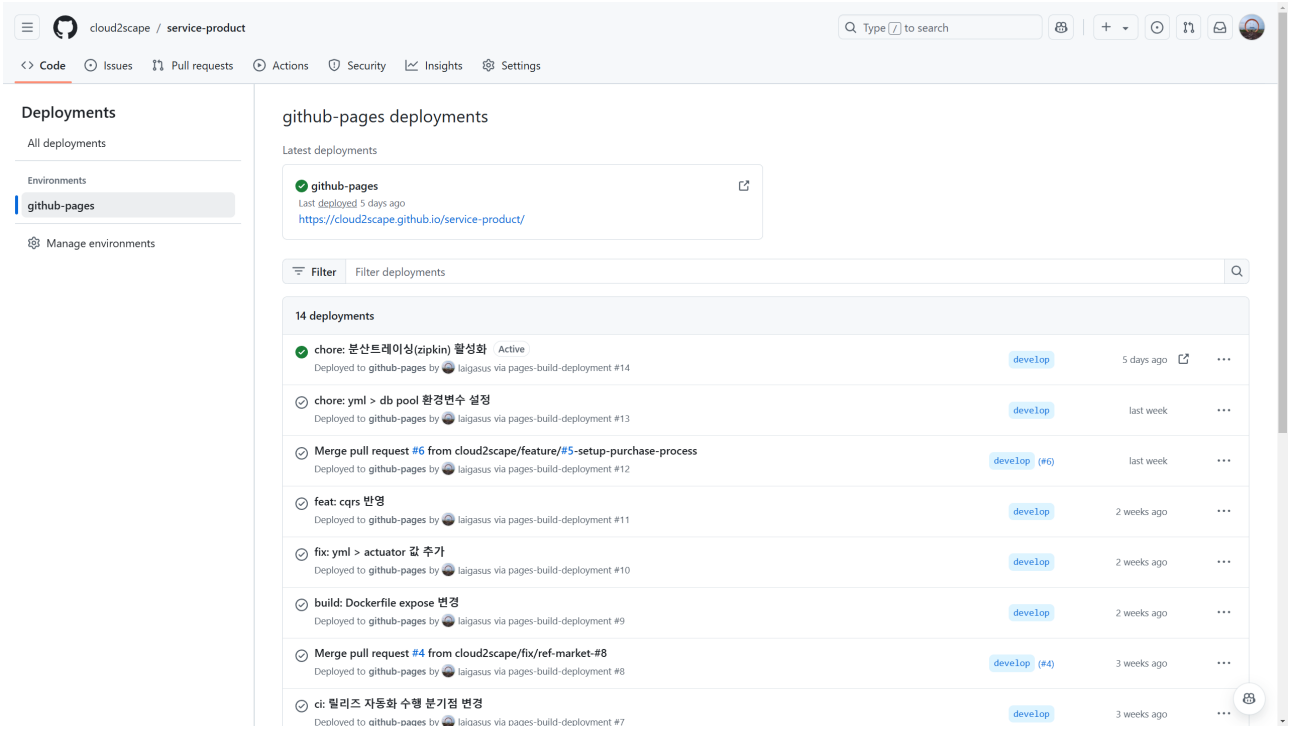
4.6.1. API 명세서 관리

개발 환경에서 애플리케이션을 실행하면 OpenAPI 정보를 갱신하도록 설정했습니다.

```
openApi {
    apiDocsUrl = "http://localhost:8663/v3/api-docs"
    outputDir = file("${project.projectDir}/docs")
    outputFileName = "openapi.yaml"
    waitTimeInSeconds = 30
    customBootRun {
        args.set(["--spring.profiles.active=dev", "--server.port=8663"])
    }
}
```

[그림 41] OpenAPI 자동 생성 코드

GitHub Actions을 설정해두어 커밋시 웹 페이지로 호스팅 되도록 했습니다.



[그림 42] 호스팅된 OpenAPI 기반 명세서

프로젝트의 사용 가이드나 정보는 Wiki, README를 작성했습니다.
이를 통해 프로젝트의 기술적 구현 사항과 운영 방법을 명확하게 문서화하여, 조직원들이 쉽게 이해하고 참조할 수 있도록 했습니다.

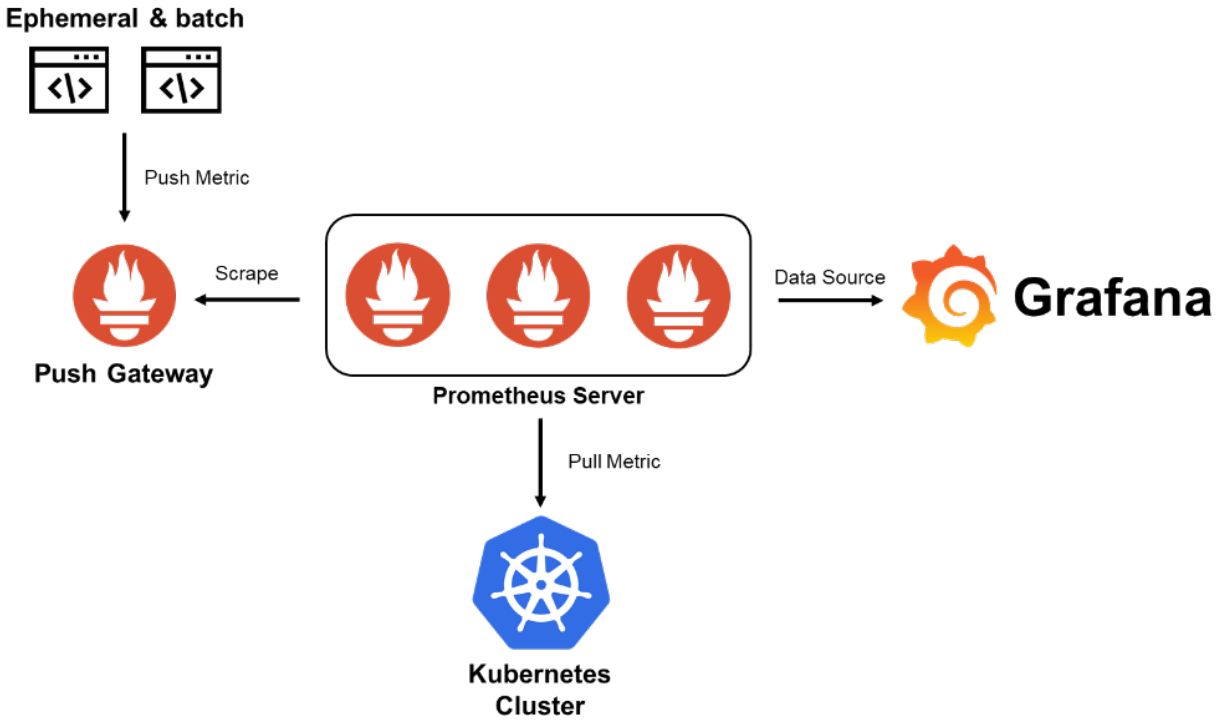
5. 모니터링

5.1. 도구

본 프로젝트의 서비스는 회원(Account), 주문(Order), 상품(Product) 등의 독립적인 마이크로서비스로 구성되어 있습니다. 이러한 분산 시스템을 안정적으로 운영하고 최적의 성능을 제공하기 위해서는 체계적인 모니터링 체계가 필수적입니다. 본 프로젝트에서는 AWS의 EKS(Elastic Kubernetes Service) 클러스터를 인프라로 활용하고 있으며, Grafana Cloud, k9s, Aptakube 등의 도구를 사용하여 서비스와 인프라에 대한 모니터링을 수행하였습니다.

모니터링은 시스템의 가용성, 성능, 보안 등을 지속적으로 추적하고 문제를 신속하게 탐지하여 대응할 수 있게 해줍니다. 특히 MSA 환경에서는 다수의 서비스와 인프라 구성 요소들이 상호작용하므로, 전체 시스템의 상태를 종합적으로 모니터링하는 것이 중요합니다. 이를 통해 병목 현상, 리소스 부족, 보안 위협 등의 문제를 사전에 방지하고 시스템의 안정성과 효율성을 높일 수 있습니다.

Grafana Cloud는 다양한 데이터 소스로부터 지표를 수집하고 시각화할 수 있는 강력한 모니터링 플랫폼입니다. 본 프로젝트에서는 Grafana Cloud를 통해 서비스 메트릭, 로그, 추적 데이터 등을 통합적으로 모니터링하고 대시보드를 구성하였습니다. k9s와 Aptakube는 각각 Kubernetes 클러스터와 워크로드를 효과적으로 관리하고 모니터링할 수 있는 도구입니다. 이들을 활용하여 인프라 리소스 사용량, 노드 상태, 네트워크 트래픽 등을 실시간으로 확인하고 최적화할 수 있었습니다.

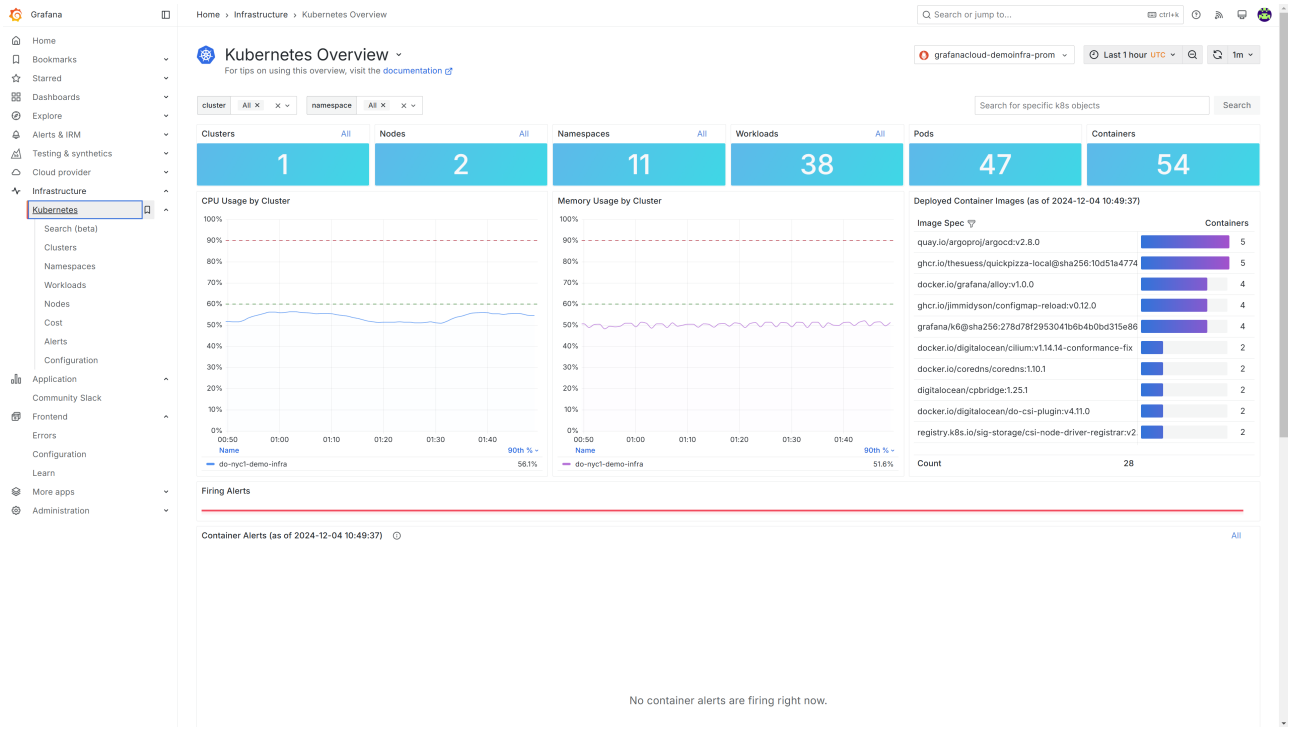


[그림 43] Prometheus 아키텍쳐

5.1.1.Grafana Cloud

Grafana Cloud는 데이터 시각화 및 모니터링 플랫폼으로, 다양한 데이터 소스로부터 메트릭을 수집하고 이를 시각화하는 데 특화되어 있습니다. Grafana는 오픈 소스 대시보드 도구로 시작했지만, Grafana Cloud는 클라우드 기반의 관리형 서비스로 제공됩니다. 주요 기능은 다음과 같습니다.

- 다양한 데이터 소스 지원: Prometheus, InfluxDB, Elasticsearch 등 다양한 데이터 소스를 연결하여 메트릭을 수집할 수 있습니다.
- 시각화 도구: 다양한 차트, 그래프, 대시보드를 통해 데이터를 시각적으로 표현할 수 있습니다. 사용자 맞춤형 대시보드를 구성할 수 있는 기능이 뛰어납니다.
- 알림 기능: 특정 조건이 충족되면 알림을 받을 수 있는 기능이 있어, 시스템의 이상 징후를 신속하게 감지하고 대응할 수 있습니다.
- 사용자 관리 및 보안: 여러 사용자가 함께 사용할 수 있도록 권한 관리 기능이 제공되며, 보안 설정을 통해 데이터 접근을 제어할 수 있습니다.

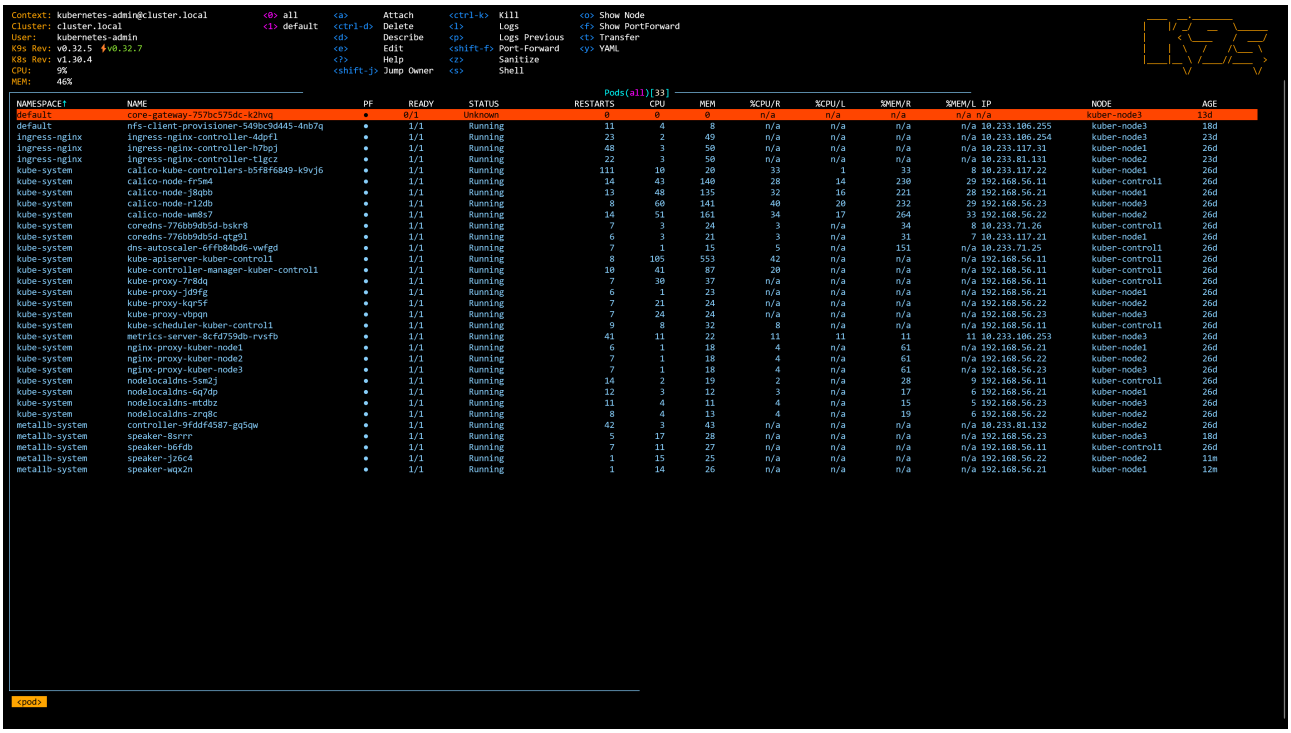


[그림 44] Grafana Cloud 화면

5.1.2. k9s

k9s는 Kubernetes 클러스터를 관리하기 위한 CLI 기반의 도구로, Kubernetes 리소스를 쉽게 탐색하고 관리할 수 있도록 돕습니다. 주요 특징은 다음과 같습니다.

- CLI 인터페이스: 터미널에서 Kubernetes 클러스터를 직관적으로 탐색할 수 있는 인터페이스를 제공합니다. 사용자는 명령어를 입력하는 대신 키보드 단축키를 사용하여 빠르게 이동할 수 있습니다.
- 리소스 모니터링: 클러스터 내의 파드, 서비스, 디플로이먼트 등 다양한 Kubernetes 리소스의 상태를 실시간으로 모니터링할 수 있습니다.
- 간편한 작업 수행: 파드 로그 보기, 포드 재시작, 리소스 삭제 등 다양한 작업을 간편하게 수행할 수 있습니다.
- 커스터마이징: 사용자가 직접 설정 파일을 수정하여 자신에게 맞는 환경으로 커스터마이징할 수 있습니다.



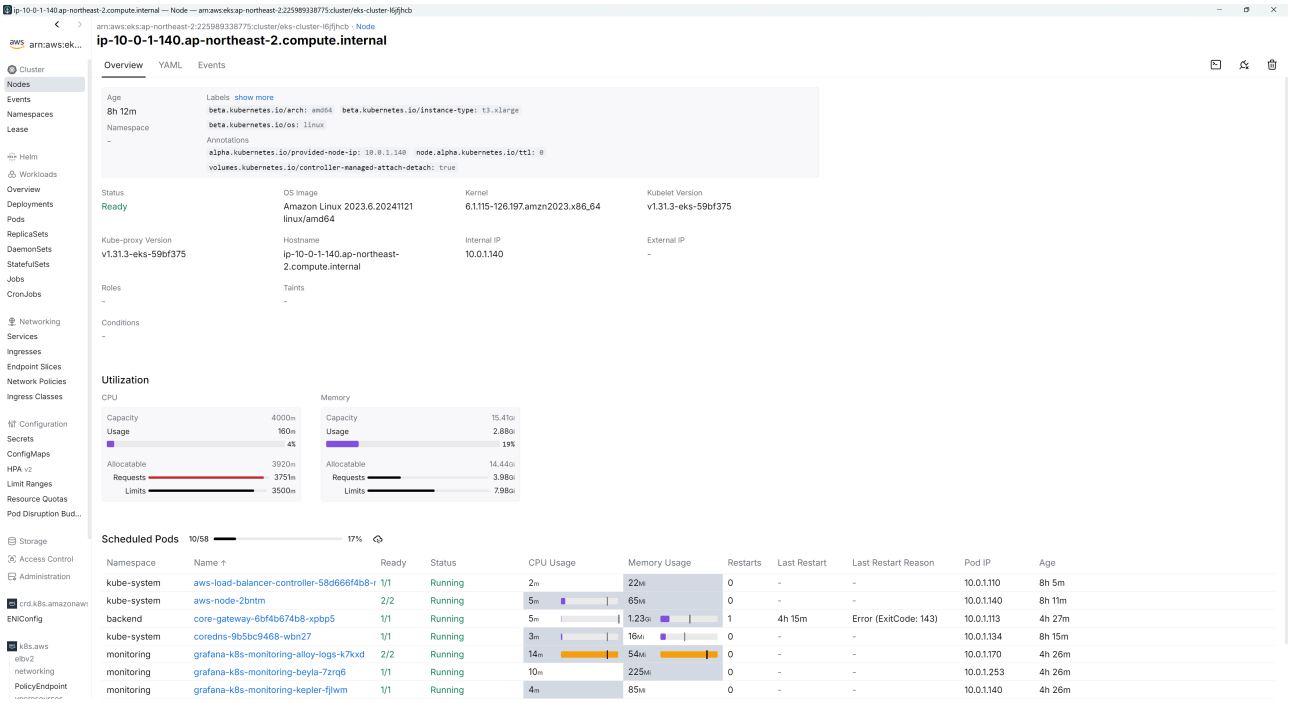
[그림 45] k9s 화면

5.1.3. Aptakube

Aptakube는 Kubernetes 클러스터의 상태를 모니터링하고 관리하기 위한 도구로, 특히 Kubernetes 리소스의 상태와 메트릭을 시각화하는 데 중점을 두고 있습니다. 주요 기능은 다음과 같습니다.

- 리소스 시각화: Kubernetes 클러스터의 리소스 상태를 실시간으로 시각화하여, 클러스터의 전반적인 건강 상태를 한눈에 파악할 수 있도록 돕습니다.
- 상태 점검: 각 리소스의 상태를 점검하고, 이상 징후를 탐지하여 경고를 발생시킬 수 있습니다.
- 간편한 대시보드: 사용자 친화적인 대시보드를 제공하여, 복잡한 Kubernetes 환경을 쉽게 이해하고 관리할 수 있습니다.
- 통합 기능: 다른 모니터링 툴과 통합하여, 데이터 수집 및 알림 기능을 강화할 수 있습니다.

이 세 가지 도구는 각각의 역할과 기능을 통해 MSA(Microservices Architecture) 환경에서 서비스와 인프라를 효과적으로 모니터링하고 관리하는 데 큰 도움을 줍니다.



[그림 46] Aptakube 화면

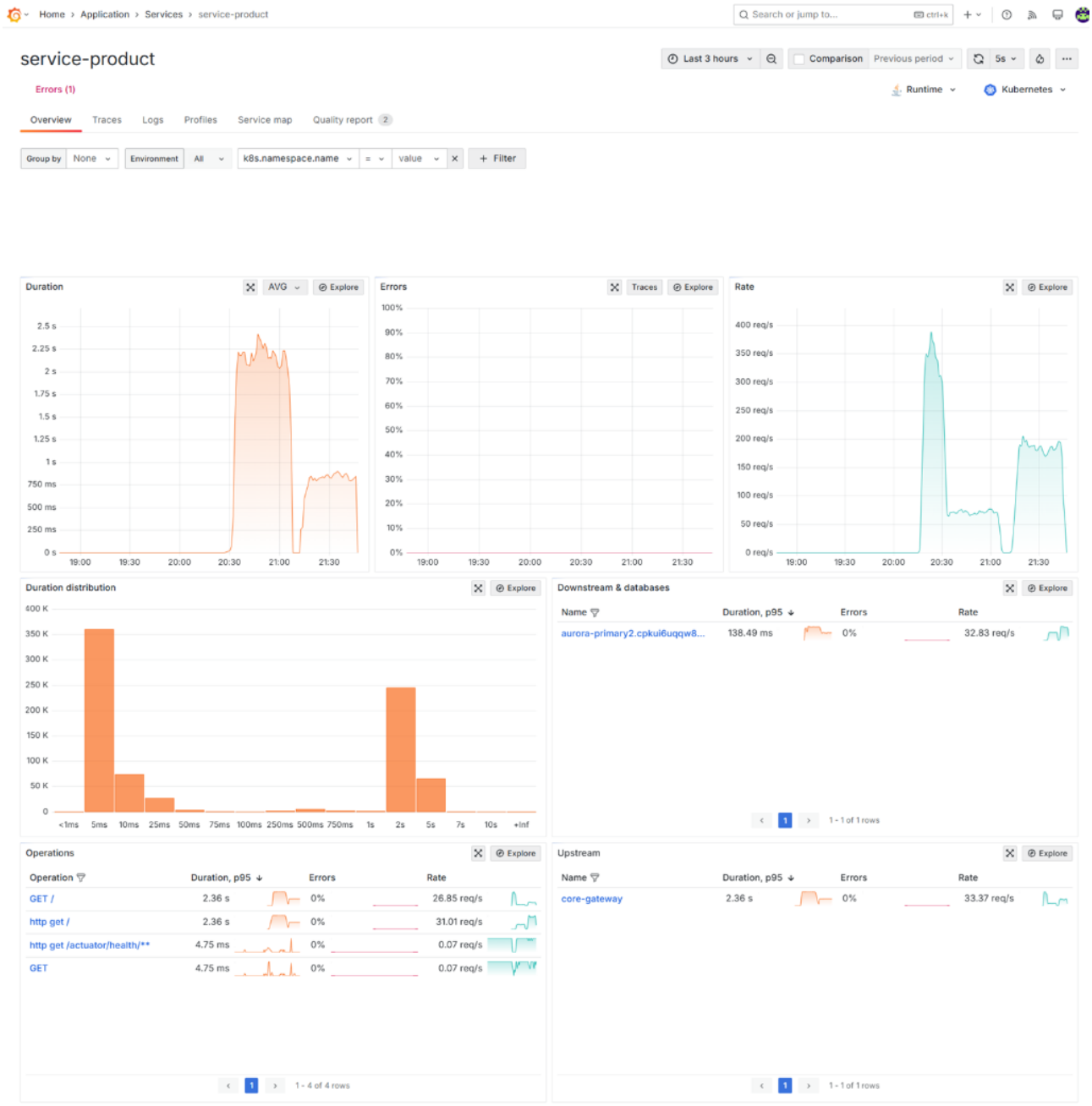
5.2. 모니터링 체계

5.2.1. 서비스 메트릭 수집 및 모니터링

MSA 서비스의 안정성과 성능을 모니터링하기 위해서는 서비스 메트릭을 체계적으로 수집하고 분석해야 합니다. 주요 서비스 메트릭에는 다음과 같은 것들이 있습니다.

- 요청 성공률(Request Success Rate): 서비스로 전송된 요청 중 성공한 비율을 나타냅니다. 서비스의 가용성과 안정성을 평가하는 데 활용됩니다.
- 응답 시간(Response Time): 요청에 대한 응답을 반환하는 데 걸리는 시간을 측정합니다. 서비스의 성능과 지연 시간을 모니터링하는 데 사용됩니다.
- 에러율(Error Rate): 서비스에서 발생한 에러의 비율을 나타냅니다. 에러 유형과 원인을 분석하여 문제를 해결하는 데 도움이 됩니다.
- 리소스 사용량(Resource Utilization): CPU, 메모리, 디스크 등의 리소스 사용량을 모니터링합니다. 리소스 부족이나 과도한 사용으로 인한 성능 저하를 방지할 수 있습니다.

서비스 메트릭을 수집하기 위해서는 서비스 코드에 모니터링 에이전트나 로깅 라이브러리를 연동해야 합니다. 본 프로젝트에서는 OpenTelemetry, Prometheus 등의 도구를 활용했습니다. 수집된 메트릭 데이터는 모니터링 시스템으로 전송되어 대시보드에 시각화됩니다. 이를 통해 서비스의 상태와 문제점을 실시간으로 파악할 수 있습니다.



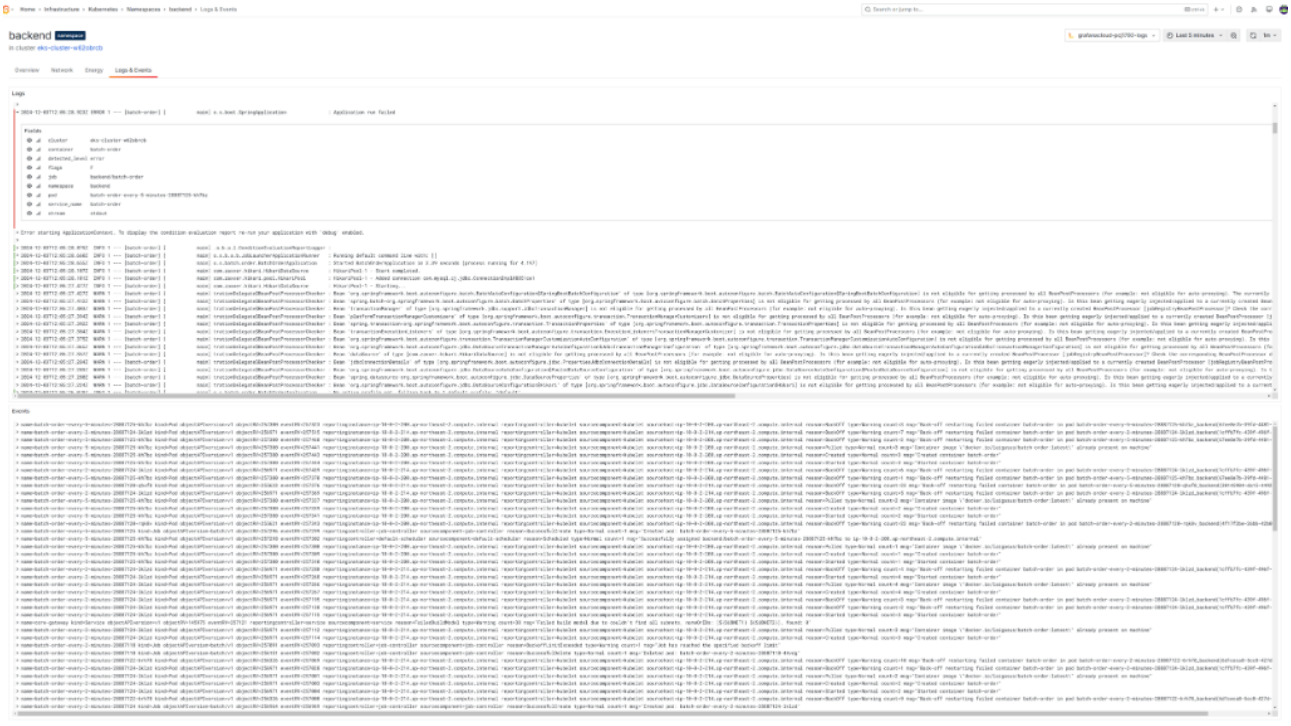
[그림 47] 실제 모니터링 화면- 요청 성공률 응답시간, 에러율 등을 나타내고 있다.

또한 메트릭 데이터를 기반으로 경고 규칙을 설정하여 이상 징후를 감지하고 통지받을 수 있습니다. 자동 스케일링 등의 자동화된 조치도 가능합니다. 서비스 메트릭을 지속적으로 모니터링하고 분석하여 문제를 해결하고 성능을 최적화할 수 있습니다.

5.2.2. 서비스 로그 수집 및 분석

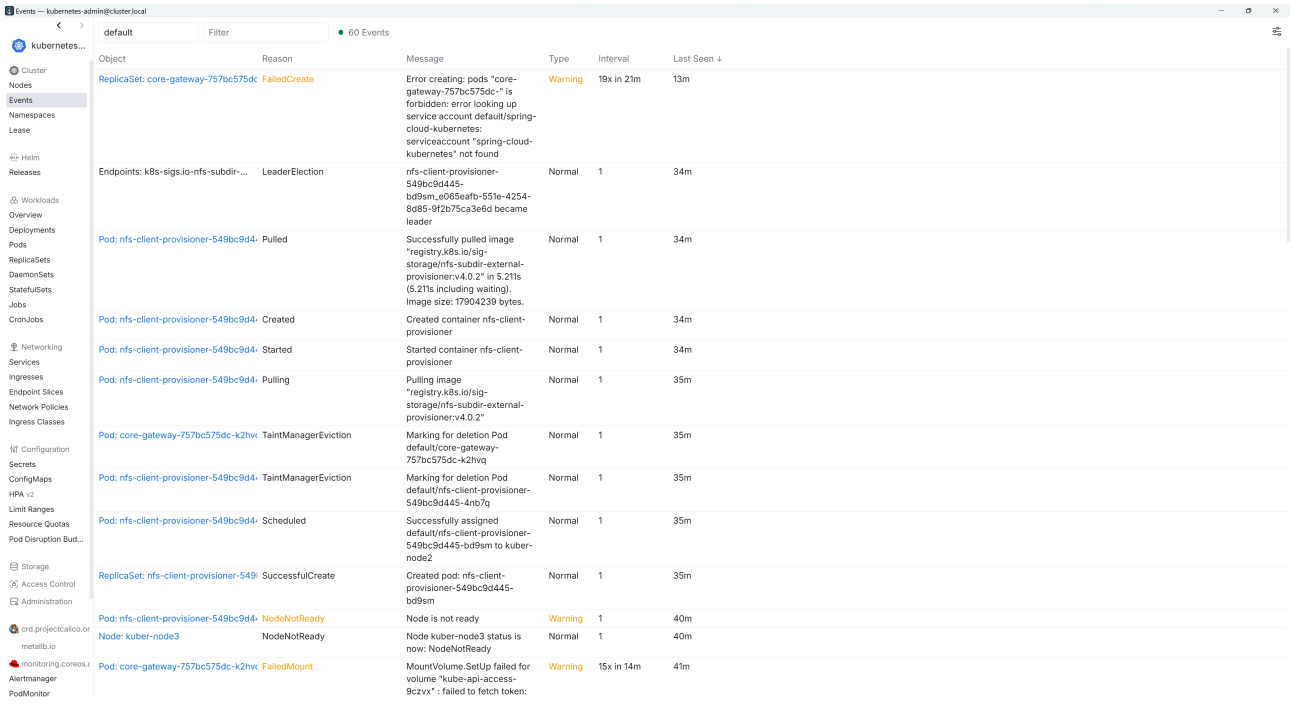
서비스의 안정성과 문제 해결을 위해서는 로그 데이터를 수집하고 분석하는 것이 필수적입니다. 로그는 서비스의 실행 과정과 에러 발생 시점에 대한 상세한 정보를 제공합니다. 이를 통해 서비스의 동작을 추적하고 오류의 원인을 파악할 수 있습니다.
로그 수집을 위해서는 서비스 코드에 로깅 라이브러리를 연동하여 로그를 생성하고 전송할 수 있도록 해야 합니다. 대표적인 로깅 도구로는 Logstash, Fluentd, Promtail 등이 있습니다. 이들 도구를 사용하면 로그를 중앙 집중식으로 수집하고 검색, 필터링, 시각화할 수 있습니다.

로그 분석 시에는 에러 메시지, 예외 상황, 느린 응답 시간 등을 모니터링하여 문제의 원인을 찾아내야 합니다. 또한 로그를 기반으로 사용 패턴을 분석하고 성능 최적화 방안을 도출할 수 있습니다. 이를 위해 로그 데이터를 시각화하고 쿼리를 수행하는 분석 기능이 필요합니다.



[그림 48] Grafana Loki 로그 목록

본 프로젝트에서는 Prometheus, Grafana Loki 등의 도구를 활용했습니다. 또한 이벤트 발생에 대한 도구로 Aptakube를 사용하여 실시간으로 문제를 파악하고 대처하도록 했습니다.



[그림 49] Aptakube의 이벤트 발생 목록

로그 데이터와 서비스 메트릭을 함께 모니터링하면 서비스의 상태를 더욱 포괄적으로 파악할 수 있습니다. 느린 응답 시간의 원인을 로그 데이터를 통해 분석할 수 있습니다. 또한 에러 메시지를 기반으로 경고 규칙을 설정하여 문제를 신속하게 탐지할 수 있습니다. 로그 분석과 메트릭 모니터링을 결합하면 MSA 서비스의 안정성과 성능을 종합적으로 관리할 수 있습니다.

5.2.3. 서비스 추적 및 분산 트레이싱

MSA 환경에서는 다수의 마이크로서비스가 상호 작용하며 복잡한 서비스 아키텍처를 구성합니다. 이러한 분산 시스템에서 요청을 추적하고 성능 병목 지점을 파악하기 위해서는 분산 트레이싱이 필수적입니다. 분산 트레이싱은 요청이 서비스들을 거치는 전체 경로를 추적하고 각 서비스에서 소요된 시간을 측정합니다. 이를 통해 서비스 간 호출 관계와 잠재적인 성능 문제를 파악할 수 있습니다.

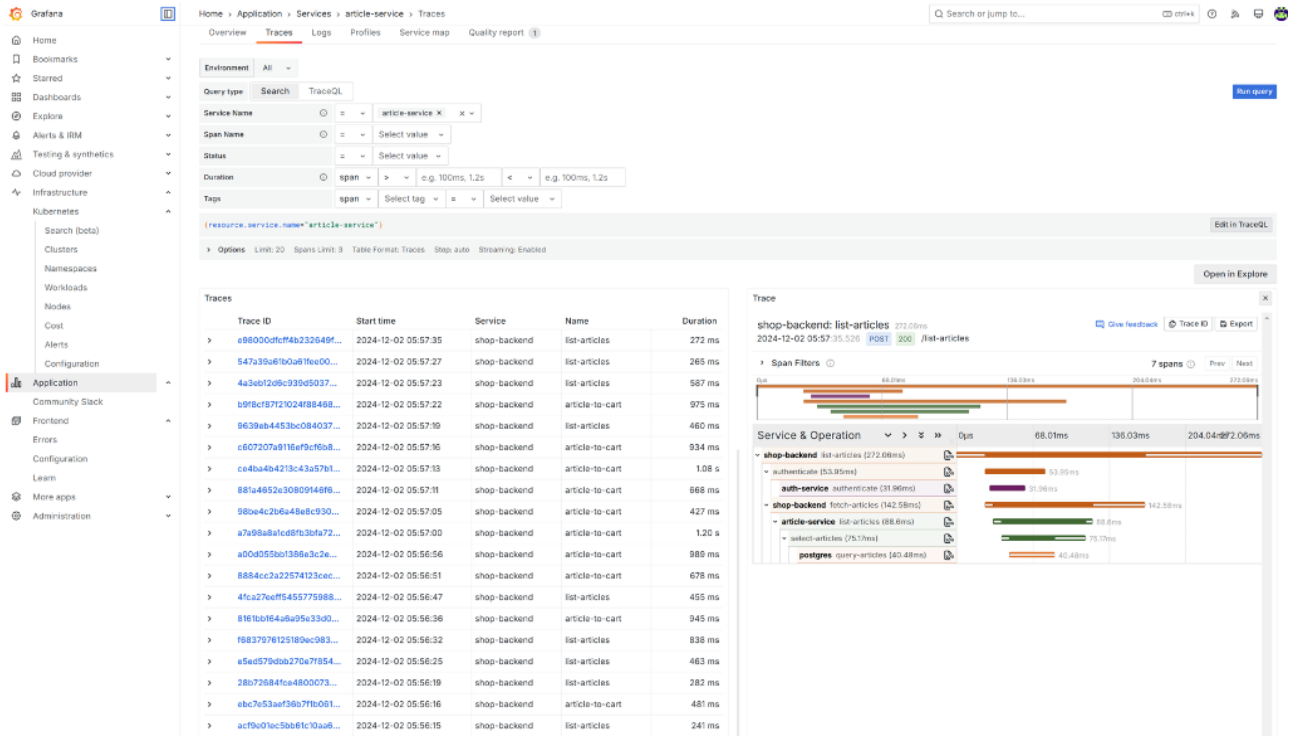
대표적인 분산 트레이싱 도구로는 Jaeger와 Zipkin이 있습니다. 본 프로젝트에서는 Zipkin을 사용 했습니다. Zipkin은 Google Dapper 논문을 기반으로 개발되었으며, 다양한 프로그래밍 언어와 프레임워크를 지원합니다. Zipkin UI를 통해 추적 데이터를 시각화하고 서비스 의존성, 지연 시간, 에러 등을 분석할 수 있습니다.

```
1 implementation 'io.zipkin.reporter2:zipkin-reporter-brave'
2 implementation 'io.micrometer:micrometer-tracing-bridge-brave'
```

[그림 50] 분산 트레이싱 제공 Spring Gradle 라이브러리

```yaml
management:
...
  zipkin:
    tracing:
      endpoint: http://${ZIPKIN_ENDPOINT:localhost}:9411/api/v2/spans
  tracing:
    sampling:
      probability: 1.0
```

[그림 51] 백엔드의 Zipkin 설정



[그림 52] Grafana 분산 트레이싱 대시보드

분산 트레이싱을 활용하면 마이크로서비스 아키텍처의 복잡성을 효과적으로 관리할 수 있습니다. 서비스 간 호출 관계와 성능 병목 지점을 파악하여 문제를 신속하게 진단하고 해결할 수 있습니다. 또한 분산 트레이싱 데이터를 기반으로 서비스 최적화와 능동적인 모니터링이 가능해집니다. 이를 통해 MSA 서비스의 안정성과 성능을 지속적으로 개선할 수 있습니다.

5.2.4. 컨테이너 및 클러스터 모니터링

MSA 서비스의 인프라를 효과적으로 모니터링하기 위해서는 컨테이너와 Kubernetes 클러스터의 상태를 지속적으로 추적해야 합니다. 컨테이너와 클러스터의 주요 모니터링 지표로는 다음과 같은 것들이 있습니다.

- 컨테이너 리소스 사용량: CPU, 메모리, 디스크 I/O, 네트워크 트래픽 등 컨테이너가 사용하는 리소스를 모니터링합니다. 리소스 부족으로 인한 성능 저하를 방지할 수 있습니다.
- 노드 리소스 사용량: 클러스터 노드의 CPU, 메모리, 디스크 사용량을 확인합니다. 노드 리소스가 과도하게 사용되거나 고갈되는 것을 탐지할 수 있습니다.
- 클러스터 상태: 클러스터의 전반적인 상태, 노드 수, 가용성 등을 모니터링합니다. 클러스터 장애나 노드 손실을 감지할 수 있습니다.
- 워크로드 분포: 파드, 서비스, 디플로이먼트 등 Kubernetes 워크로드의 분포와 상태를 확인합니다. 워크로드 불균형이나 문제를 탐지할 수 있습니다.

이러한 지표를 수집하고 시각화하기 위해서는 Prometheus, Kubernetes 메트릭 서버, cAdvisor 등의 도구를 활용할 수 있습니다. Prometheus는 다양한 데이터 소스로부터 지표를 수집하고 쿼리, 시각화할 수 있는 오픈소스 모니터링 시스템입니다. Kubernetes 메트릭 서버는 Kubernetes 클러스터의 리소스 사용량 데이터를 제공합니다. cAdvisor는 컨테이너 리소스 사용량을 모니터링하는 도구입니다.

이들 도구를 통합하여 컨테이너와 클러스터에 대한 전반적인 모니터링 대시보드를 구축할 수 있습니다. 예를 들어 Grafana에서 Prometheus 데이터 소스를 연동하여 클러스터 노드, 파드, 컨테이너 지표를 시각화할 수 있습니다. 또한 경고 규칙을 설정하여 리소스 부족, 노드 장애, 워크로드 문제 등을 실시간으로 탐지할 수 있습니다. 이렇게 컨테이너와 클러스터를 체계적으로 모니터링하면 MSA 서비스 인프라의 안정성과 성능을 보장할 수 있습니다.

5.2.5. 노드 및 리소스 모니터링

노드와 리소스의 상태를 모니터링하는 것은 MSA 서비스 인프라의 안정성과 성능을 유지하는 데 필수적입니다. 주요 모니터링 항목으로는 노드 상태, 노드 리소스 사용량, 컨테이너 및 파드 리소스 사용량 등이 있습니다.

노드 상태 모니터링에서는 클러스터의 노드 수, 노드 가용성, 노드 장애 여부 등을 지속적으로 추적합니다. 이를 통해 노드 손실이나 장애 상황을 조기에 탐지할 수 있습니다. 노드 리소스 사용량 모니터링에서는 노드의 CPU, 메모리, 디스크, 네트워크 리소스 사용량을 실시간으로 모니터링합니다. 이를 통해 노드 리소스가 부족하거나 과도하게 사용되는 상황을 감지할 수 있습니다.

컨테이너 및 파드 리소스 사용량 모니터링에서는 CPU, 메모리, 디스크 I/O, 네트워크 트래픽 등의 지표를 확인합니다. 이를 통해 리소스 부족으로 인한 성능 저하를 방지할 수 있습니다. 또한 워크로드 분포를 모니터링하여 특정 파드나 노드에 워크로드가 집중되는 문제를 탐지할 수 있습니다.

이러한 지표를 수집하고 시각화하기 위해서는 Prometheus, Kubernetes 메트릭 서버, cAdvisor 등의 도구를 활용할 수 있습니다. Prometheus는 다양한 데이터 소스로부터 지표를 수집하고 쿼리, 시각화할 수 있는 오픈소스 모니터링 시스템입니다. Kubernetes 메트릭 서버는 Kubernetes 클러스터의 리소스 사용량 데이터를 제공합니다. cAdvisor는 컨테이너 리소스 사용량을 모니터링하는 도구입니다.

이들 도구를 통합하여 노드와 리소스 사용량에 대한 종합적인 모니터링 대시보드를 구축할 수 있습니다. 예를 들어 Grafana에서 Prometheus 데이터 소스를 연동하여 노드, 파드, 컨테이너 지표를 시각화할 수 있습니다. 또한 경고 규칙을 설정하여 노드 장애, 리소스 부족 등의 문제를 실시간으로 탐지할 수 있습니다.

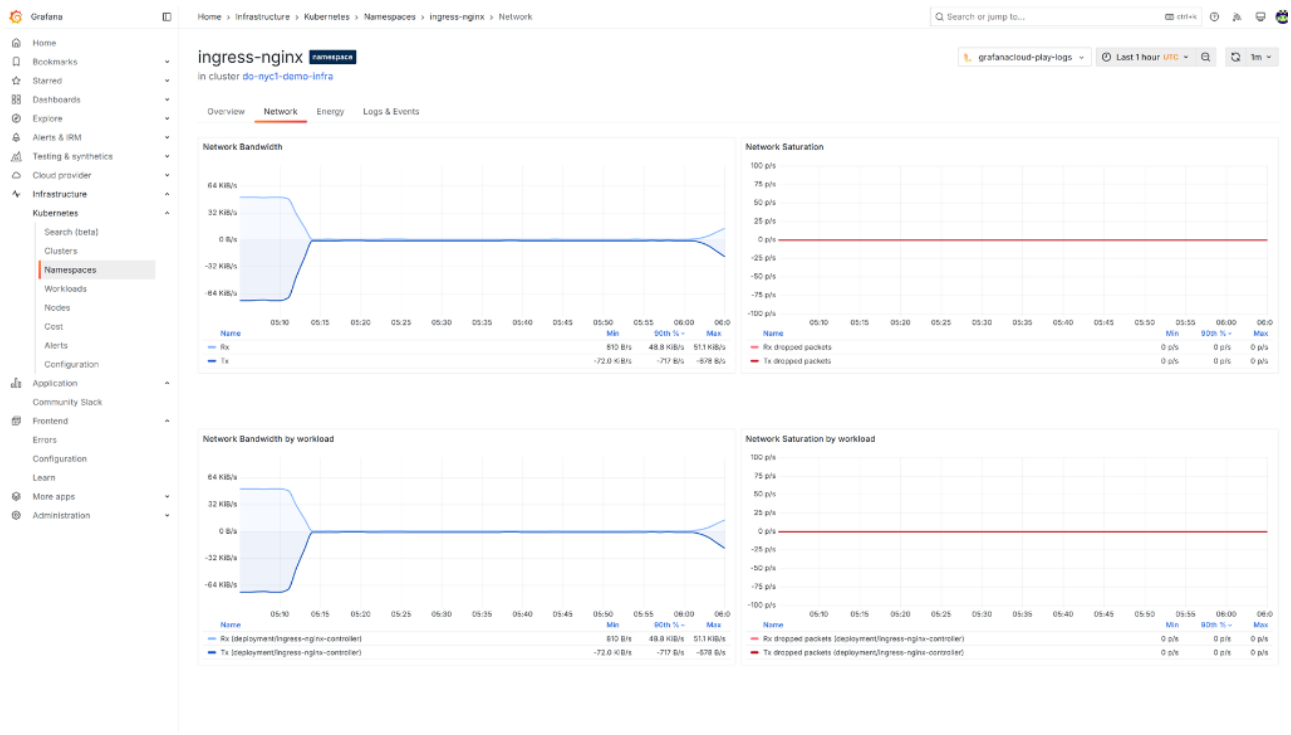
노드와 리소스 모니터링을 통해 MSA 서비스 인프라의 안정성과 가용성을 보장할 수 있습니다. 리소스 부족으로 인한 성능 저하를 방지하고, 노드 장애를 조기에 탐지하여 대응할 수 있습니다. 또한 워크로드 분포를 최적화하여 리소스를 효율적으로 활용할 수 있습니다. 이렇게 노드와 리소스를 체계적으로 모니터링하면 MSA 서비스의 안정성과 성능을 극대화할 수 있습니다.

5.2.6. 네트워크 모니터링

MSA 서비스의 안정적인 운영을 위해서는 네트워크 상태를 지속적으로 모니터링해야 합니다. 네트워크 모니터링에서 주요하게 확인해야 할 지표로는 다음과 같은 것들이 있습니다.

- 네트워크 대역폭: 네트워크의 전송 용량을 모니터링합니다. 대역폭 부족으로 인한 병목 현상을 탐지할 수 있습니다.
- 네트워크 지연 시간: 패킷이 전송되는 데 걸리는 시간을 측정합니다. 높은 지연 시간은 서비스 성능 저하로 이어질 수 있습니다.
- 패킷 손실률: 전송 중 손실된 패킷의 비율을 확인합니다. 패킷 손실은 네트워크 불안정성을 나타냅니다.
- 트래픽 패턴: 네트워크 트래픽의 흐름과 패턴을 분석합니다. 이를 통해 병목 지점을 파악하고 리소스를 최적화할 수 있습니다.

본 프로젝트에서는 Prometheus를 사용하여 네트워크 메트릭을 수집하고 쿼리, 시각화할 수 있도록 했습니다.



[그림 53] Prometheus를 사용하여 네트워크 시각화

네트워크 모니터링을 통해 서비스 간 통신 문제를 조기에 탐지하고 대응할 수 있습니다. 예를 들어 높은 지연 시간이나 패킷 손실이 발생하면 경고를 받아 신속하게 조치를 취할 수 있습니다. 또한 트래픽 패턴을 분석하여 네트워크 리소스를 최적화하고 병목 현상을 해결할 수 있습

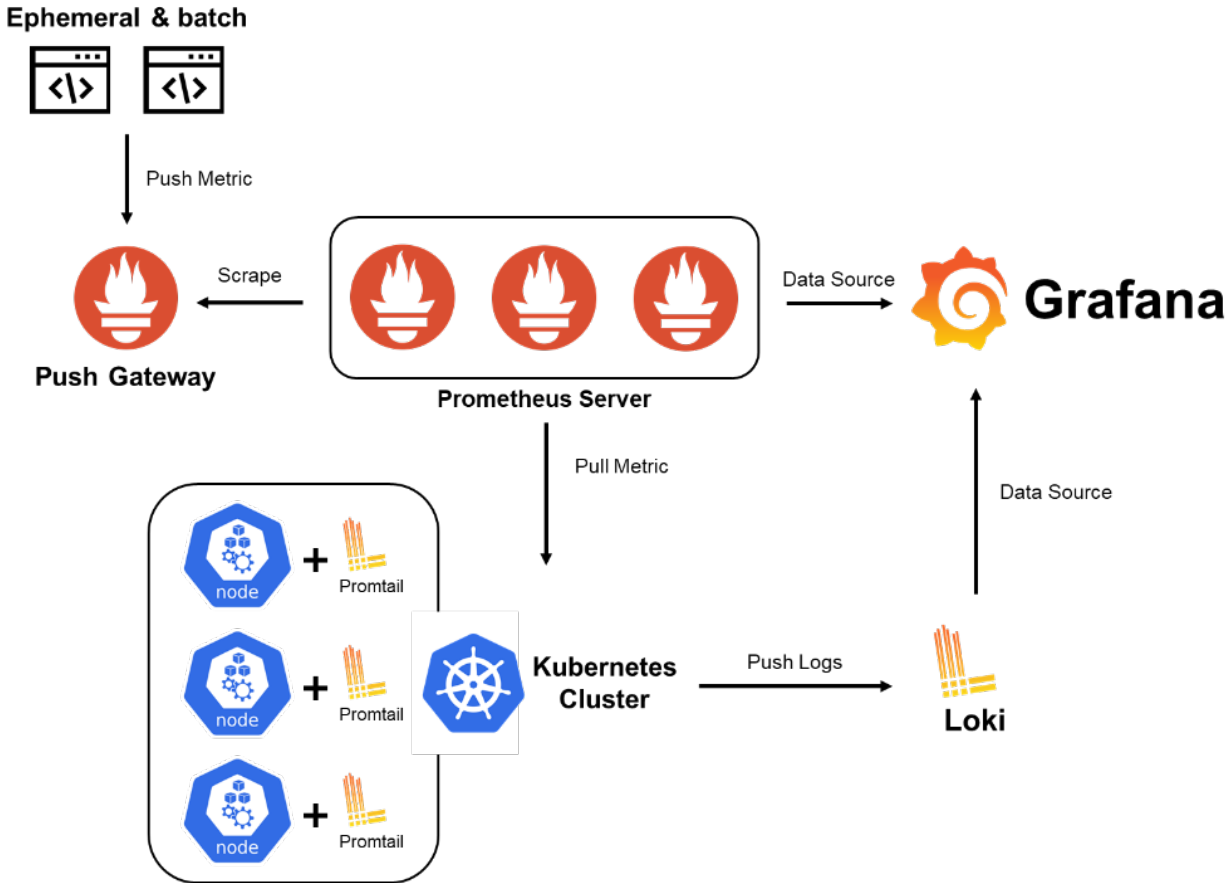
니다.

MSA 환경에서는 서비스 간 통신이 매우 중요하므로, 네트워크 모니터링은 필수적입니다. 네트워크 상태를 실시간으로 모니터링하고 문제를 조기에 탐지하여 서비스의 안정성과 성능을 보장할 수 있습니다. 따라서 네트워크 모니터링 체계를 구축하고 지속적으로 최적화해야 합니다.

5.3. 모니터링 전략

5.3.1. 서비스 모니터링 전략: 대시보드 활용

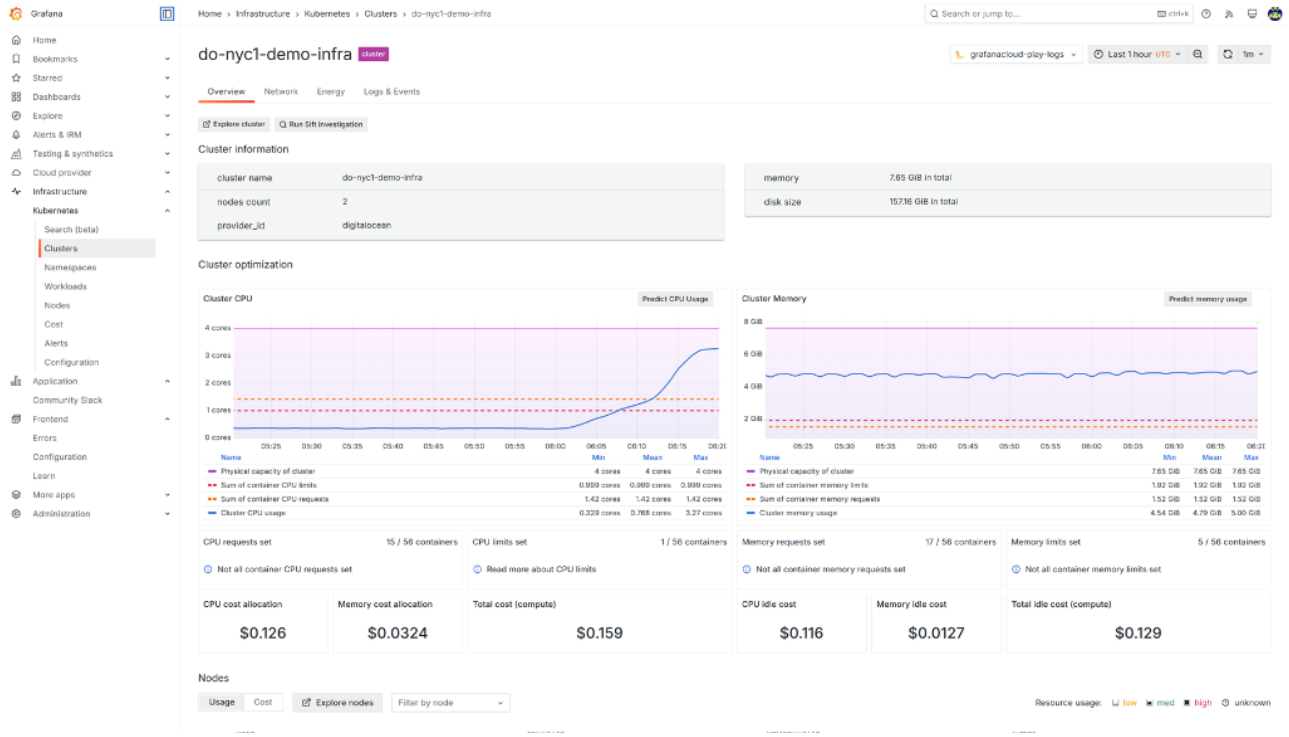
MSA 서비스 모니터링에서 대시보드는 매우 중요한 역할을 합니다. 대시보드를 통해 서비스 메트릭, 로그, 추적 데이터 등을 한 곳에서 시각화하고 통합적으로 모니터링할 수 있습니다. 이를 위해서는 모니터링 도구와 데이터 소스를 연동하여 대시보드를 구성해야 합니다.



[그림 54] 본 프로젝트 모니터링 전략

대시보드를 활용하면 서비스의 상태와 문제점을 한눈에 파악할 수 있습니다. 메트릭, 로그, 추적 데이터를 통합적으로 모니터링하여 서비스의 성능과 안정성을 종합적으로 관리할 수 있습니다. 또한 대시보드에서 경고 규칙을 설정하면 이상 징후를 실시간으로 탐지하고 신속하게 대응할 수 있습니다.

MSA 서비스 모니터링 전략에서 대시보드 활용은 필수적이기 때문에 본 프로젝트에서는 Grafana Cloud의 적절한 대시보드를 이용하여 서비스의 다양한 측면을 모니터링하고 통찰력을 얻었습니다. 이를 통해 서비스의 가용성과 성능을 극대화하고 문제를 신속하게 해결할 수 있었습니다.



[그림 55] Grafana 대시보드

5.3.2. 서비스 모니터링 전략: 지속적인 모니터링 및 최적화

MSA 서비스의 안정성과 성능을 지속적으로 유지하기 위해서는 모니터링과 최적화가 지속적으로 이루어져야 합니다. 서비스의 부하 패턴, 워크로드 분포, 리소스 사용량 등은 시간에 따라 변화하므로 정기적으로 모니터링하고 최적화해야 합니다.

지속적인 모니터링과 최적화를 통해 MSA 서비스의 안정성과 성능을 지속적으로 개선할 수 있습니다. 문제를 조기에 탐지하고 신속하게 대응할 수 있으며, 리소스 활용을 최적화하여 비용 효율성을 높일 수 있습니다. 이렇게 지속적인 모니터링과 최적화 프로세스를 통해 MSA 서비스의 안정성과 효율성을 극대화할 수 있습니다.

본 프로젝트에서 지속적인 모니터링을 위해서 서비스 메트릭, 로그, 추적 데이터를 체계적으로 수집하고 분석했습니다. Prometheus와 같은 모니터링 시스템을 활용하여 메트릭을 수집하고, Grafana를 통해 시각화하고 경고 규칙을 설정했습니다. 또한 Loki를 사용하여 로그를 수집하고 분석했습니다. 그리고 Zipkin과 같은 분산 트레이싱 도구를 통해 서비스 간 호출 관계와 성능 병목 지점을 파악하는데 이용했습니다.

5.3.3. 인프라 모니터링 전략: 용량 계획 및 리소스 최적화 및 관리

MSA 인프라의 용량 계획 수립과 리소스 최적화를 위해서는 모니터링 데이터와 애플리케이션 특성을 면밀히 분석해야 합니다. 서비스 메트릭과 리소스 사용량 데이터를 기반으로 현재 및 미래의 수요를 예측하고, 필요한 리소스 용량을 계획할 수 있습니다. Kubernetes 클러스터의 노드 수, 인스턴스 유형, 스토리지 볼륨 크기 등을 결정할 수 있습니다.

용량 계획 수립 시에는 AWS 계정 비용 관리 도구, AWS 비용 탐색기 등을 활용하여 비용 효율성도 고려해야 합니다. 과도한 리소스 프로비저닝은 비용 낭비로 이어질 수 있으므로, 적절한 균형을 찾는 것이 중요합니다.

리소스 최적화를 위해서는 Kubernetes 클러스터 오토스케일링, 노드 오토프로비저닝, 수평적 파드 오토스케일링 등의 기능을 활용할 수 있습니다. 이를 통해 동적으로 리소스를 확장하거나 축소할 수 있습니다. AWS Auto Scaling, AWS EKS 클러스터 오토스케일러, Kubernetes Horizontal Pod Autoscaler(HPA) 등의 도구를 사용할 수 있습니다.
또한 Kubernetes 노드 레이블과 테인트, 노드 셀렉터 등의 기능을 활용하여 워크로드를 최적의 노드에 배치할 수 있습니다. 이를 통해 리소스 활용도를 높이고 성능 병목 현상을 방지할 수 있습니다. Kubernetes 클러스터 오토스케일러와 노드 오토프로비저닝을 함께 사용하면 더욱 효과적인 리소스 최적화가 가능합니다.

이렇게 모니터링 데이터와 도구를 활용하여 용량 계획을 수립하고 리소스를 최적화하면, MSA 서비스의 안정성과 비용 효율성을 극대화할 수 있습니다. 또한 서비스 수요와 워크로드 패턴의 변화에 따라 지속적으로 용량 계획과 리소스 할당 전략을 조정해야 합니다. 이를 통해 MSA 인프라의 성능과 효율성을 지속적으로 개선할 수 있습니다.

불필요한 리소스 낭비와 비용 초과는 서비스의 효율성과 수익성을 저해할 수 있으므로, 지속적인 모니터링과 최적화가 필요합니다.

비용 최적화를 위해서는 먼저 리소스 사용량을 모니터링해야 합니다. Prometheus와 Kubernetes 메트릭 서버를 활용하여 CPU, 메모리, 디스크, 네트워크 등의 리소스 사용량 데이터를 수집할 수 있습니다. 수집된 데이터를 분석하여 과도하게 프로비저닝된 리소스나 비효율적인 사용 패턴을 파악할 수 있습니다. 이를 통해 불필요한 리소스를 제거하거나 최적화할 수 있습니다.

또한 클라우드 리소스와 서비스에 대한 비용을 실시간으로 모니터링해야 합니다. 비용 최적화를 위한 전략으로는 자동 스케일링, 노드 오토프로비저닝, 스팟 인스턴스 활용 등이 있습니다. Kubernetes Horizontal Pod Autoscaler(HPA)와 AWS Auto Scaling을 통해 리소스를 탄력적으로 확장 및 축소할 수 있습니다. AWS EKS 클러스터 오토스케일러와 노드 오토프로비저닝을

활용하면 클러스터 노드 수를 동적으로 조정할 수 있습니다. 또한 AWS 스팟 인스턴스를 사용하면 비용 절감 효과를 얻을 수 있습니다.

본 프로젝트에서도 모니터링을 통해 용량을 계획하고, 리소스를 최적화하여 서비스의 효율성을 높이고 적절한 AWS 인스턴스를 사용하여 비용을 절감했습니다. 이렇게 모니터링 도구와 자동화 전략을 활용하여 MSA 서비스 인프라의 비용을 지속적으로 최적화하고 관리할 수 있습니다. 불필요한 리소스 낭비를 제거하고, 리소스 사용량에 따라 탄력적으로 확장 및 축소하여 비용 효율성을 높일 수 있습니다. 또한 비용 초과를 조기에 탐지하고 대응함으로써 예산을 효과적으로 관리할 수 있습니다. 이를 통해 MSA 서비스의 안정성과 성능을 보장하면서도 운영 비용을 최소화할 수 있습니다.

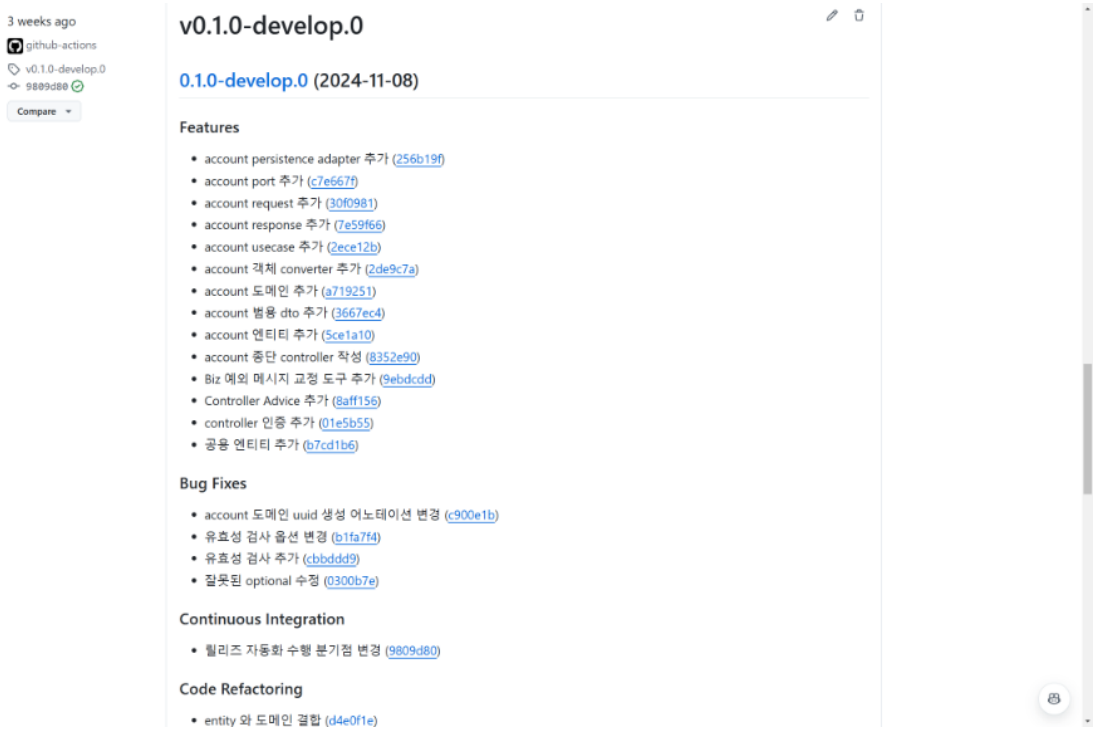
6. CI/CD

6.1. CI(Continuous Integration)

지정한 브랜치에 커밋이 되면 Github Actions에서 다음의 동작을 수행합니다.

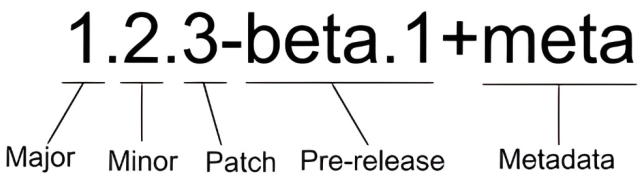
6.1.1. 릴리즈 노트 작성

Semantic Versioning 2.0 을 근거 버저닝 후 커밋 히스토리를 수집해 자동화했습니다.



[그림 56] 릴리즈 자동화

버저닝 구성 요소입니다.

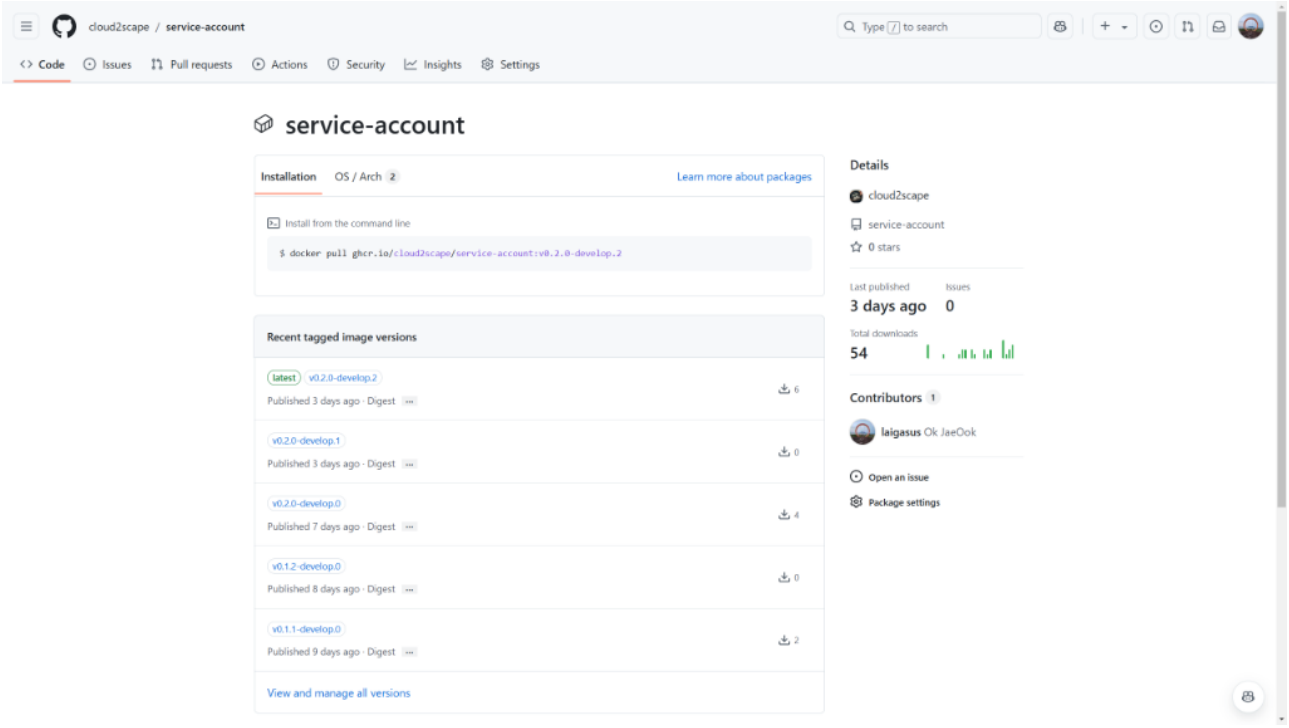


[그림 57] Semantic Versioning 구조

Major: 이전 버전과 호환되지 않는 변경사항이 있을 때 증가
Minor: 이전 버전과 호환되는 새로운 기능이 추가될 때 증가
Patch: 이전 버전과 호환되는 버그 수정 시 증가
Pre-release: 정식 배포 전 미리보기 버전을 나타냄
Metadata: 빌드 관련 메타데이터를 나타냄

6.1.2. 애플리케이션 빌드 및 배포

GitHub Container Registry에 이미지를 올리기 위해 워크플로우를 추가했습니다.



[그림 58] GitHub Container Registry에 올라간 이미지

이미지 빌드시 멀티스테이징을 사용해 이미지 크기를 줄였습니다.

```
FROM gradle:8.10-jdk21-alpine AS builder

WORKDIR /app

COPY build.gradle settings.gradle /app/
COPY src /app/src

RUN gradle build -x test

FROM openjdk:21-slim

COPY --from=builder /app/build/libs/core-gateway.jar .

ENTRYPOINT ["java", "-jar", "core-gateway.jar"
EXPOSE 8080
EXPOSE 80
```

[그림 59] 멀티 스테이징 반영된 Dockerfile

'latest'라벨을 추가해 항상 최신의 이미지를 사용할 수 있도록 마련했습니다.

```
tags: |
  ghcr.io/${{ github.repository }}:${{ steps.tag_version.outputs.new_tag }}
  ghcr.io/${{ github.repository }}:latest
```

[그림 60] 'latest' 태그 라벨 설정

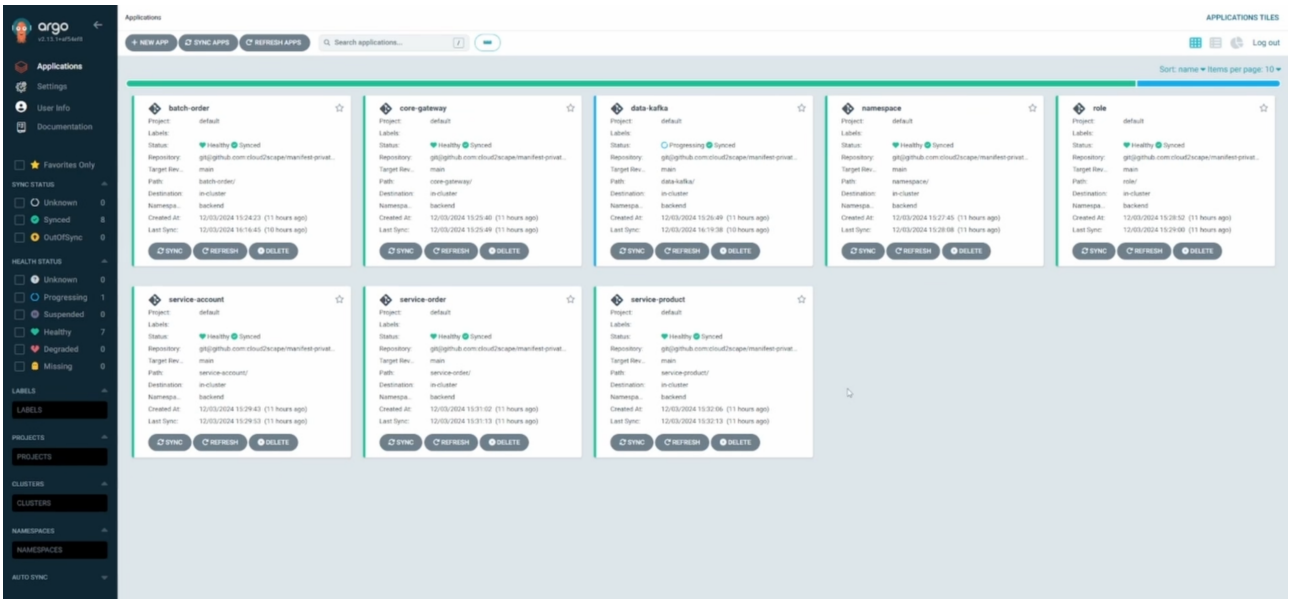
6.2. CD(Continuous Delivery/Deployment)

6.2.1. ArgoCD를 이용한 CD 파이프라인 구성

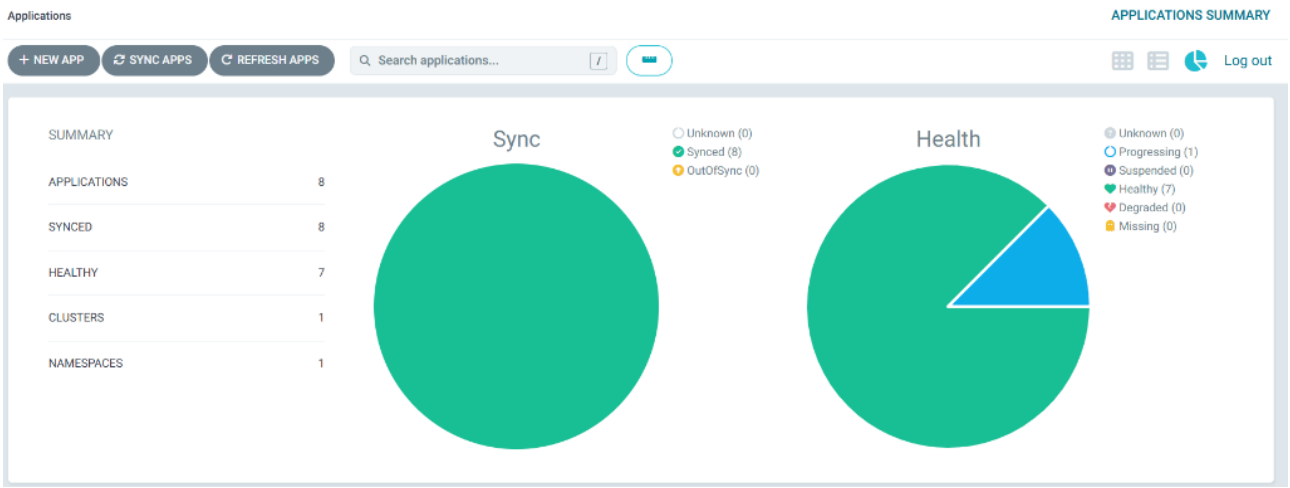
ArgoCD는 GitOps 방식을 통해 Kubernetes 클러스터로 애플리케이션을 배포하고 관리하는 도구입니다. CI 과정에서 생성된 결과물을 Git 저장소에 반영한 뒤, 이를 ArgoCD가 Kubernetes 환경으로 배포합니다.

6.2.2. ArgoCD 네임스페이스 생성

ArgoCD를 클러스터에 설치하기 전에 먼저 전용 네임스페이스(argocd)를 생성하여 리소스를 관리하도록 설정하였습니다. 이를 통해 각 리소스를 논리적으로 분리하고 보안을 강화할 수 있었습니다.



[그림 61] ArgoCD 대시보드



[그림 62] ArgoCD 대시보드

6.2.3. ArgoCD 설치

설치는 ArgoCD 프로젝트에서 제공하는 표준 매니페스트 파일을 사용하여 수행하였습니다.
이를 통해 최신 버전의 ArgoCD를 빠르고 간편하게 배포하였습니다.

```
1 kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

[그림 63] ArgoCD 설치 명령어

6.2.4. CI/CD 자동화 플로우

코드 변경사항이 브랜치에 푸시될 때 이미지 빌드 및 배포가 자동으로 트리거됩니다.
Semantic Versioning 기반으로 이미지 태그를 생성하고, GitHub Container Registry에 업로드 합니다.

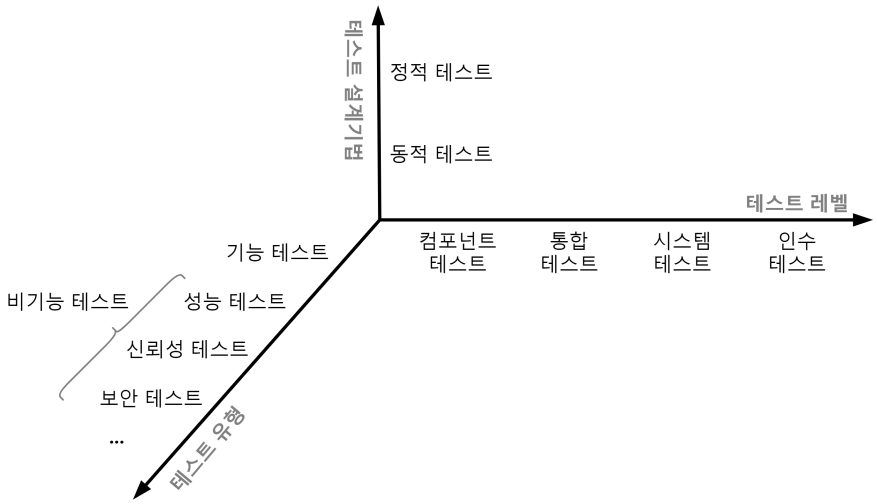
6.2.5. 자동화의 이점

수동 배포 과정을 최소화하여 효율성을 높입니다.
배포 안정성을 보장하며, 문제가 발생 시 롤백이 용이합니다.

7. 품질 테스트

7.1. 테스트 분류 소개

테스트 레벨, 유형, 설계 기법에 따라 수행했던 종류를 초록색으로 표기했습니다.



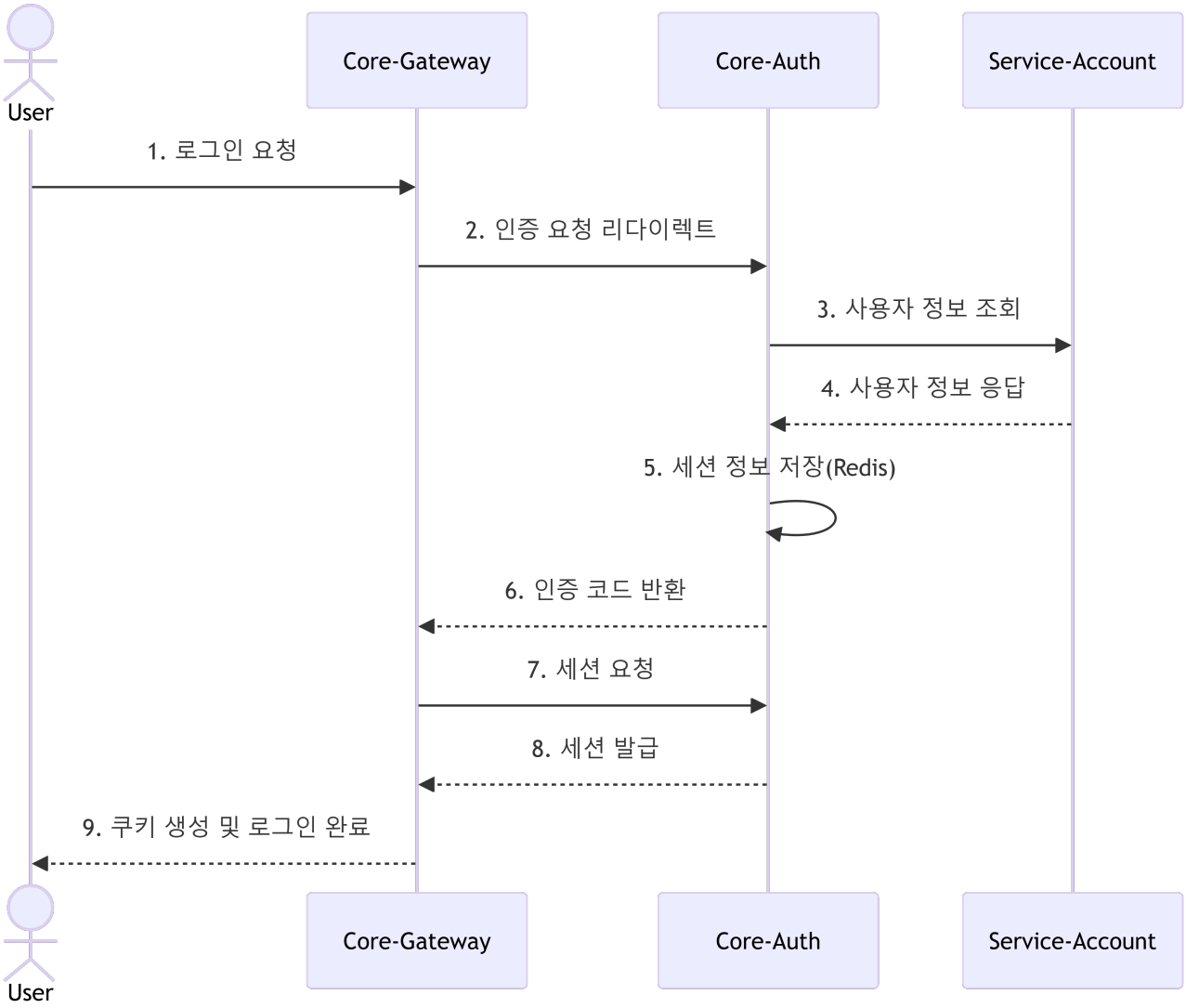
[그림 64] 품질 테스트 분류

분류	테스트 종류			설명
테스트 레벨	컴포넌트/단위 테스트			각각의 컴포넌트를 테스트
	통합 테스트			컴포넌트 간 인터페이스 테스트
	시스템 테스트			전체 시스템이 목적을 만족시키는지 테스트
	인수 테스트			사용자의 요구사항을 만족하는지 테스트
테스트 설계	동적 테스트	명세 기반		명세서를 바탕으로 테스트 케이스를 작성
		구조 기반		프로그램 코드를 바탕으로 테스트 케이스를 삭정
		경험 기반		테스터의 경험을 기반으로 테스트 케이스를 작성
	정적 테스트	리뷰		산출물에 존재하는 결함을 검출하거나 프로젝트의 진행 사항을 점검
		정적 분석		자동화된 도구를 이용하여 산출물의 결함을 검출하거나 복잡도를 측정
테스트 유형	기능 테스트			기능적 요구사항 측면의 결함 검출 및 충족 여부 확인
	비기능 테스트	기능적합성		사용자의 요구사항을 만족하는 기능이 제공되는 정도를 테스트
		성능효율성		시스템의 응답시간이나 처리량을 테스트
		호환성		다른 시스템과의 상호 연동 능력이나 공존성을 테스트
		사용성		사용자가 이해하고 배우기 쉬운 정도를 테스트
		신뢰성		규정된 조건/기간에 오동작 없이 수행하는 능력을 테스트
		보안성		시스템의 정보 및 데이터를 보호하는 능력을 테스트
		유지보수성		소프트웨어 유지보수의 용이성을 테스트
		이식성		다양한 플랫폼에서 운영될 수 있는 능력을 테스트

7.2. 테스트 시나리오(기능)

해당 서비스의 주요 기능을 대표해 테스트를 수행했습니다.

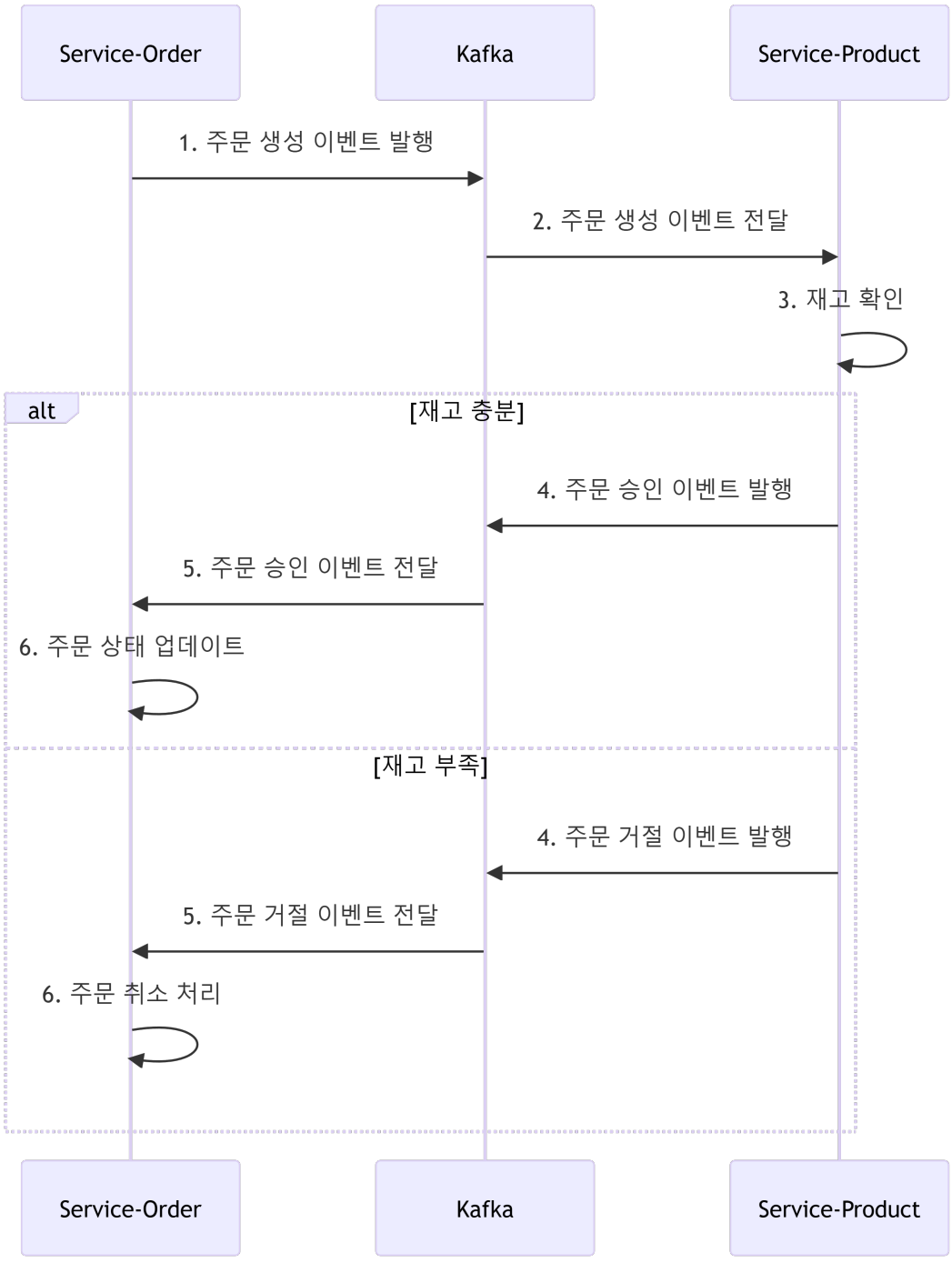
7.2.1. 로그인 프로세스



[그림 65] 로그인 프로세스 시퀀스 다이어그램

1. 로그인 요청 검사(/login)
2. 인증 리다이렉트 검사
3. 사용자 정보 조회 검사
4. 세션 관리 검사
5. 인증 코드 및 세션 발급 검사
6. 최종 응답 검사

7.2.2. 주문 프로세스



[그림 66] 주문 프로세스 시퀀스 다이어그램

1. 주문 생성 이벤트 검사
2. 이벤트 전달 검사
3. 재고 확인 로직 검사
4. 승인/거절 이벤트 검사

7.3. 테스트 시나리오(비기능)

이전 테스트 분류에서 효율성, 신뢰성, 유지보수성을 검증하고자 적절한 시나리오를 마련했습니다.

신뢰성 테스트

성숙성: 시스템 또는 구성 요소가 정상 작동 상태에서 신뢰성 요구를 충족시키는 정도
- 장애시간(MTTF, Availability) 측정

가용성: 사용자가 시스템 또는 구성요소를 사용하고자 할 때 사용 및 접근이 가능한 정도
- 노드 장애 테스트
- 파드 장애 테스트

결함 허용성: 하드웨어나 소프트웨어 결함이 있음에도 불구하고 시스템 또는 구성 요소가 의도한 대로 작동하는 정도
- 장애 주입 테스트
- 페일오버/페일백 테스트

복구성: 중단 또는 장애가 발생한 경우 시스템이 영향을 받은 데이터를 복구하고 상태를 재설정할 수 있는 정도
- 시스템 재시작 테스트

성능 효율성 테스트

시간 반응성: 기능 수행 시 시스템의 응답 처리시간 및 처리율이 요구사항을 충족시키는 정도
- 응답 시간 테스트
- 지연 시간 테스트
- 처리율 테스트

자원 효율성: 기능 수행 시 시스템이 사용하는 자원이 요구사항을 충족시키는 정도
- CPU 사용률 테스트
- 메모리 사용량 테스트
- 디스크 I/O 테스트

수용성: 시스템 매개변수(동시 사용자 수, 통신 대역폭, 트랜잭션 처리량 등)의 최대 한계가 요구사항을 충족시키는 정도
- 부하 테스트
- 스트레스 테스트
- 확장성 테스트
- 동시 사용자 테스트

유지보수성 테스트

모듈성: 하나의 구성요소에 대한 변경이 다른 구성 요소에 미치는 영향이 최소화되도록 시스템 또는 컴퓨터 프로그램이 개별 구성요소로 구성된 정도
- 마이크로서비스 독립성 테스트

재사용성: 시스템 자산이 하나 이상의 시스템에서 사용될 수 있는 정도,
또는 다른 자산을 구축할 수 있는 정도
- 인프라 템플릿 재사용 테스트(Terraform, Kubernetes)

분석성: 부분에 의도된 변경이 전체 시스템에 미치는 영향을 평가하거나 결함 또는 결함 원인에 대해 제품을 진단하거나 수정될 부분을 식별할 수 있는 정도
- 로그 모니터링 시스템 검증
- 메트릭 수집 및 분석 테스트
- 트레이싱 시스템 동작 검증

변경 용이성: 결함이나 품질 저하 없이 효과적이고 효율적으로 수정될 수 있는 정도
- 인프라 구성 변경 자동화 테스트
- 롤백 절차 검증
- 무중단 배포 테스트

테스트 용이성: 테스트 수행을 용이하게 하는 정도
- 자동화된 인프라 테스트 실행
- 테스트 환경 구성 자동화 검증
- 모니터링 도구 연동 테스트

7.4. 테스트 수행 계획

신뢰성 테스트 계획서

목적

ISO/IEC 25010 표준의 신뢰성 품질 특성에 따른 SW 테스트를 수행하여 시스템의 안정성과 신뢰도를 평가하고 개선하는 것을 목적으로 함.

일정 당일 수행 **수행자** 고나연, 옥재욱

테스트 범위

성숙성
- 장애시간 측정

가용성
- 노드 장애 테스트
- 파드 장애 테스트

결함 허용성
- 장애 주입 테스트
- 페일오버/페일백 테스트

복구성
- 시스템 재시작 테스트

전략

분류	테스트 종류
레벨	시스템 테스트
유형	비기능 테스트(장애복구)
설계	시나리오 기반(장애 상황 대응)

테스트 환경

HW 요구 사항
- AWS EKS

SW 요구 사항
- JMeter
- AWS Console

성공 기준
- 새로운 워커노드 연결
- 서비스 무중단

성능 효율성 테스트 계획서

목적

ISO/IEC 25010 표준의 성능 효율성 품질 특성에 따른 소프트웨어 테스트를 수행하여 시스템의 성능과 자원활용도를 평가하고 개선하는 것을 목적으로 함.

일정 당일 수행 **수행자** 김수민, 박찬준

테스트 범위

시간 반응성

- 응답/지연 시간 테스트

자원 효율성

- HW 자원(CPU, 메모리, 디스크 I/O) 사용률 테스트

수용성

- 부하 테스트
- 스트레스 테스트
- 확장성 테스트
- 동시 사용자 테스트

전략

분류	테스트 종류
레벨	시스템 테스트
유형	비기능 테스트(성능 테스트)
설계	시나리오 기반(대기 시간, 일반 사용자수, 피크 사용자수)

테스트 환경

HW 요구 사항

- AWS EKS
- AWS EC2
- AWS RDS

SW 요구 사항

- JMeter
- Grafana

성공 기준

- 응답시간 2000ms 이내
- 3000 명의 요청 시에도 API 요청 완수
- 최대 TPS 측정

유지보수성 테스트 계획서

목적

ISO/IEC 25010 표준의 유지보수성 품질 특성에 따른 소프트웨어 테스트를 수행하여 시스템의 유지보수 용이성을 평가하고 개선하는 것을 목적으로 함

일정 당일 수행 **수행자** 옥재욱

테스트 범위

모듈성

- 마이크로서비스 독립성 테스트

재사용성

- 인프라 템플릿 재사용

분석성

- 메트릭, 트레이싱, 로그 수집 및 분석 테스트
- 시스템 동작 검증

변경 용이성

- 인프라 구성 변경 자동화 테스트
- 롤백 절차 검증
- 무중단 배포 테스트

테스트 용이성

- 자동화된 인프라 테스트 실행
- 테스트 환경 구성 자동화 검증

전략

분류	테스트 종류
레벨	시스템 테스트
유형	유지보수성 테스트
설계	경험 기반

테스트 환경

HW 요구 사항

- 개발 환경 K8s(Kubespray)
- 운영 환경 K8s(EKS)
- 데이터(Kafka, Redis, RDS)

SW 요구 사항

- AWS
- 테라폼, K8s Manifest
- 애플리케이션 이미지

성공 기준

- 각 리소스 모니터링에 이상 없어야 함
- 설정 변경 시에도 적용된다

7.5. 테스트 현황 보고

신뢰성 테스트 현황

목적

장애에 대응하여 회복탄력성을 가짐으로써 제품의 신뢰도를 향상하고
시스템의 오류 복구 능력을 확인하기 위함

작성일 당일 수행 **보고자** 고나연, 옥재욱

진행 현황

성숙성
- 장애시간 측정

가용성
- 노드 장애 테스트
- 파드 장애 테스트

결함 허용성
- 장애 주입 테스트
- 페일오버/페일백 테스트

복구성
- 시스템 재시작 테스트

성과
- Probe 설정이 되어 있어 컨테이너 내 애플리케이션 Health-Check로 상태 확인하여 정상 동작하는 파드만 올라갔음.
- 임의로 노드 장애 유발시 'Desired' 값에 따라 노드가 증설되었으며, HPA로 파드가 증설됨에 따라 동일하게 동작하였음.
- AWS Aurora 로 Replica(RO), Primary(RW)에 대한 자동 페일오버가 적용되어 있음
- Resilience4j가 적용되어 서킷브레이커, Retry에 대한 폴백 메커니즘이 구현되어 있어 인프라 별 장애에 대해 2차 피해가 가지 않도록 설계됨

관측 결과

2시간 스트레스 테스트(30% over)

- 캐시 + 싱글 파드 기준
- 총 다운 타임. 5.97분
- MTTR(평균 복구 시간) 1.4925분
- MTTF(평균 가용 시간) 30분
- MTBF(평균 고장 간격) 31.4925분
- 고가용성(HA): 95.025%

- 캐시 + HPA + 서킷브레이커 기준
- 총 다운타임: 1.2분
- MTTR(평균 복구 시간): 0.8149분
- MTTF(평균 가용 시간): 80.6776분
- MTBF(평균 고장 간격): 81.4925분
- 고가용성(HA): 99%

성능 효율성 테스트 현황

목적

잠재적인 성능 문제 식별 및 해결을 통해 시스템의 한계를 파악하고 서비스 안정성을 확보함으로써 사용자 만족도를 향상시킴

작성일 당일 수행 **보고자** 김수민, 박찬준

진행 현황

시간 반응성

- 응답/지연 시간 테스트

자원 효율성

- HW 자원(CPU, 메모리, 디스크 I/O) 사용률 테스트

수용성

- 부하 테스트(임계점 파악)
- 스트레스 테스트
- Idle 테스트
- 스파이크 테스트

성과

- 스모크 테스트 결과 초당 30개의 요청을 2시간이상 서비스 가능함을 확인
- 임계점 테스트 결과 파드당 초당 50개의 요청이 가능함을 확인
- 스파이크 테스트 결과 초당 50개의 요청이라는 급격한 요청수에도 최소 15분 서비스 이상없음을 확인
- AWS Aurora t3.medium Master 1개, Replica 1개로 초당 90개의 요청 처리가 문제없음 확인
- Redis 캐시 적용 전후의 DB 부하에 확실한 차이가 있음

결함/인시던트

AWS Aurora의 가용성이 지나치게 좋은 관계로 DB 자체에 대한 성능 테스트가 무의미함

- free tier t3.medium 조차도 초당 90회 요청까지 서비스 가능

유지보수성 테스트 현황

목적

시스템의 변경 용이성을 측정하여 유지보수 비용을 절감하고 시스템의 수명 연장하기 위함

작성일 당일 수행 **보고자** 옥재욱

진행 현황

모듈성
- 마이크로서비스 독립성 테스트

재사용성
- 인프라 템플릿 재사용 테스트

분석성
- 로그 모니터링 시스템 검증
- 메트릭 수집 및 분석 테스트
- 트레이싱 시스템 동작 검증

변경 용이성
- 롤백 절차 검증
- 무중단 배포 테스트

테스트 용이성
- 자동화된 인프라 테스트 실행
- 테스트 환경 구성 자동화 검증
- 모니터링 도구 연동 테스트

성과

- MSA 구조 확인 완료
- 서비스 별 Main submodule Repo로 설계됨
- 메트릭, 트레이싱, 로그수집을 Grafana에서 관측되었음
- IaC(Terraform, Kubernetes manifest)를 사용해 프로비저닝, 오케스트레이션 되어있음
- 인프라에서는 ArgoCD로 manifest를 실행시켜 EKS에 자동 배포(CD)되었음
- 애플리케이션 부문에서 코드 커밋 후 Github Container Registry에 자동 배포(CI)되었음
- 테스트 코드는 없으나 수행에 필요한 모니터링, API 명세서, 컨테이너가 제공되었음
- 서비스에 Rolling Update 반영되어 있으며 롤백은 K8s Rollout으로 관리하고 있음

결함/인시던트

- 단위, 통합테스트 부재

7.6. 테스트 종료 보고

신뢰성 테스트 보고서

개요

시스템의 안정성과 성능을 평가하며, 장애 발생 시 복구 능력을 분석합니다. 주요 테스트 결과와 발견된 문제점을 바탕으로 향후 개선 방안을 제시합니다. 이를 통해 시스템의 신뢰성을 종합적으로 검토하고 최적화할 수 있습니다.

작성일	당일 수행	**보고자**	고나연, 옥재욱

테스트 결과

성숙성
- 장애시간 측정

가용성
- 가용성 수치 계산
- 노드 장애 테스트
- 파드 장애 테스트

결함 허용성
- 장애 주입 테스트
- 페일오버/페일백 테스트

복구성
- 시스템 재시작 테스트

결함/인시던트

이상 없음

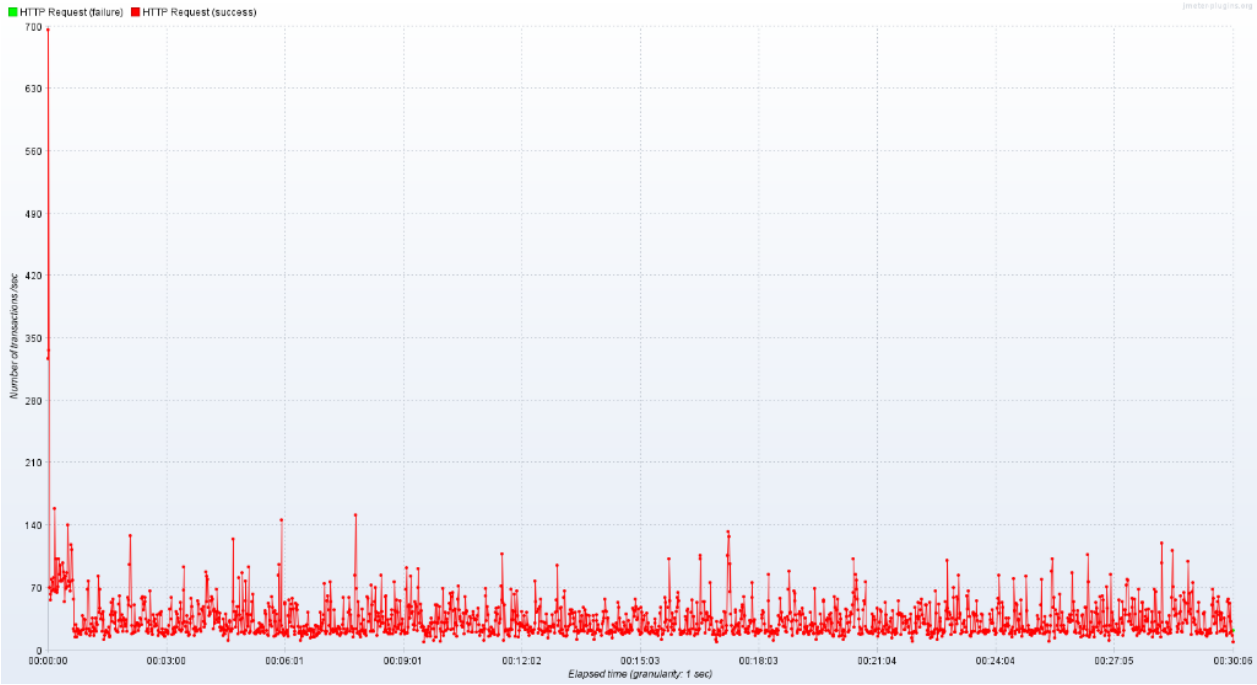
총평 및 제언

시스템의 고가용성을 99% 수준으로 달성하고, 장애 발생 시에도 3분 이내에 신속하게 복구할 수 있는 체계가 되어 있었습니다.

또한, Horizontal Pod Autoscaler를 도입하여 파드 자원을 효율적으로 관리하고 있으며, Resource Quotas를 적용해 네임스페이스 간의 자원 격리를 보장함으로써 안정적인 운영 이 이루어질 수 있도록 설계되어 있었습니다.

신뢰성 테스트 보고서(첨부)

[그림 67] 스모크 테스트. 2시간 동안 오류 발생률 0%



[그림 68] 모든 요청에 대해 정상적인 응답을 받았음

성능 효율성 테스트 보고서

개요

시스템의 응답 시간, 처리량, 자원 사용률을 평가하여 성능 병목 현상을 식별하고 최적화 방안을 도출하는 데 목적이 있습니다. 주요 테스트 결과와 분석을 통해 시스템의 효율성을 종합적으로 평가합니다.

작성일 당일 수행 **보고자** 김수민, 박찬준

테스트 결과

자원 효율성

- 캐시 히트율
- 조회 비용

수용성

- 동시 접속자
- 응답시간
- 처리량
- CPU 사용률

결함/인시던트

- DB 구조의 단순성으로 인해 튜닝 가능성이 낮음.
- 요청을 보내는 컴퓨터의 성능 한계로 인해 만 단위 이상의 요청 테스트가 불안정하여 제대로 실시하지 못함.
- AWS의 기본 네트워크 성능이 월등하여 개선할 수 있는 여지가 크지 않음.

총평 및 제언

주요 테스트 결과와 분석을 통해 시스템의 효율성을 종합적으로 평가한 결과, DB 구조의 단순성, 테스트 환경의 한계, 그리고 AWS의 기본 네트워크 성능이 이미 높은 수준임을 확인하였습니다. EKS에 사용된 EC2 인스턴스(t3.xlarge)의 성능도 매우 높기에 게이트웨이가 충분히 트래픽을 견디면 그 이상에도 파드가 중단되지 않을 것으로 예상합니다. 테스트 환경의 한계를 극복하기 위해 AWS의 로드 테스트 도구를 활용하여 더 높은 부하에서의 성능을 평가해 예정입니다. 앞으로 지속적인 모니터링과 주기적인 성능 테스트를 통해 시스템의 효율성을 유지하고 개선해 나가는 것이 중요할 것입니다.

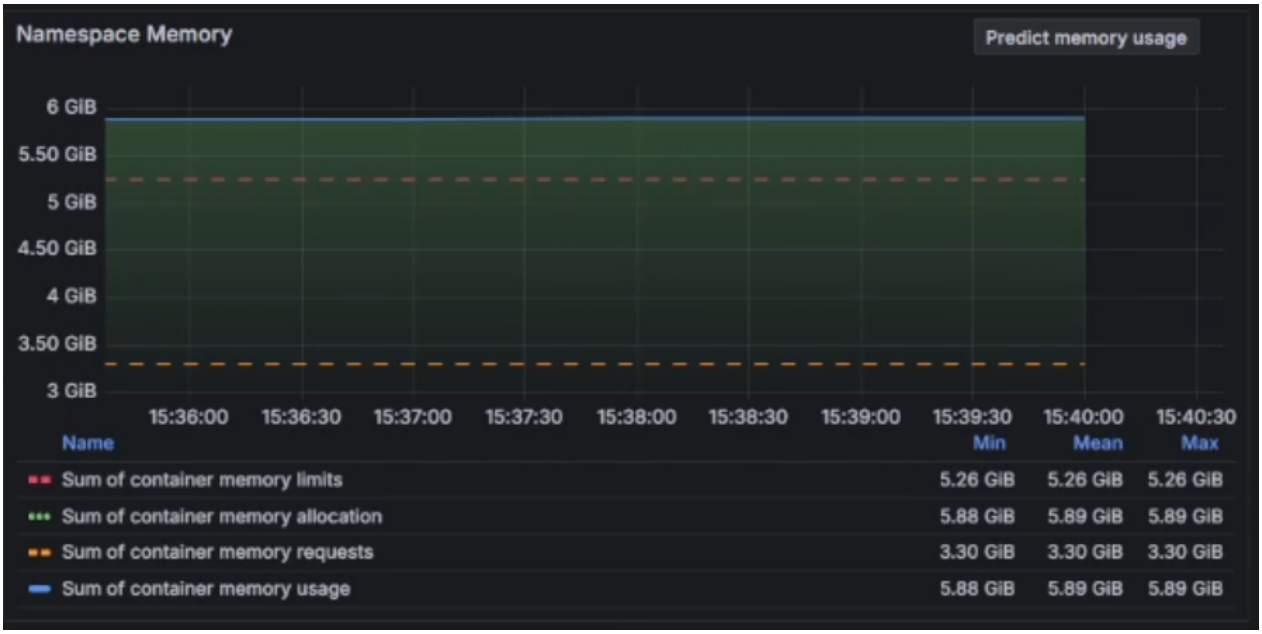
성능 효율성 테스트 보고서(첨부)

1.DB 부하 테스트

- 초당 3회 ~ 100회 까지 요청을 초당 5회씩 늘려가며 DB의 부하를 테스트
- 초당 50회에서 Aurora CPU 사용률 45%(이하 생략)
- 초당 60회 - 66%
- 초당 70회 - 72%
- 초당 75회 - 81%
- 초당 80회 - 87%, 새로운 Replica 생성 시작
- 초당 90회 - 기존 Replica 72%, 새 replica 27%
- 초당 95회 - 96%, 24%
- 초당 100회 - 72%, 10%
- 결론 - 급격한 CPU 사용률 증가로 인해 기존 Replica에 부하가 쏠리는 것을 막을 수 없었음. 그러나 Autoscaling 정책의 CPU 사용률과 추적 시간을 조정하면 충분히 대응 가능할 것으로 판단.

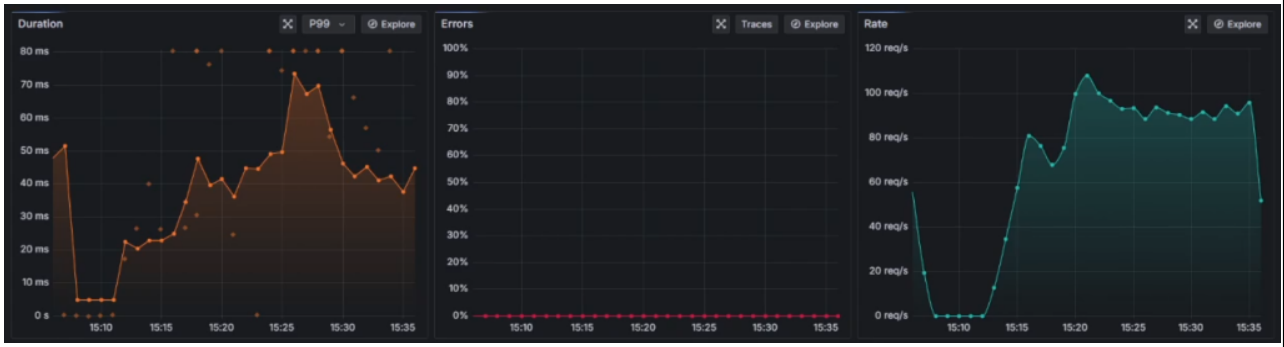
2.파드 테스트

2.1 Redis 적용전



[그림 69] 스트레스 테스트시 Pod의 메모리 상태가 Full에 가까움을 알 수 있다

성능 효율성 테스트 보고서(첨부)



[그림 70] 스트레스 테스트시 Pod의 응답속도

2.1.1 임계점 확인

- 시나리오 - 1분 동안 최대 요청수 9000회
- Pod Limit: CPU 1000m, Mem 1000Gi

- 1차 시도 - 게이트웨이 Pending
- 문제점: CPU 사용률이 높은 게이트웨이가 요청 수를 견디지 못함.
- 해결방안: 게이트웨이 최대 생성수를 10으로 설정

- 2차 시도 - 약 4분 후 서비스 Pod Pending
- 문제점: 초당 130회 이상의 요청이므로 초당 30회 정도의 요구사항을 충분히 넘어서는 것으로 판단

2.1.2 스트레스 테스트

- 시나리오1 - 1분마다 초당 5회씩의 요청이 늘어나 13분후 최대 초당 80회 요청이 옴.
- 결과 - 서비스 이상없음
- 결론 - 충분한 하드웨어 스펙을 가지고 있음

- 시나리오2 - 1분마다 초당 1회에서 40분후 초당 100회까지 서비스 요청이 늘어남
- 결과 - 서비스 이상없음
- 결론 - 초당 100회의 서비스도 가능

- 시나리오3 - 초당 60회의 요청수 30분 동안 지속될 것이다.
- 결과 - 17분 후 서비스 중지
- 문제점 - 15분 이상의 장기간 요청시에는 응답 불가. 하드웨어 성능의 한계로 추정
- 해결방안 - 캐시 적용 혹은 인스턴스 타입 변경(Scale Up)

성능 효율성 테스트 보고서(첨부)

[그림 71] Pod 증설 시의 Pod의 리소스 사용량

2.1.3 스파이크 테스트

- 시나리오 - 초당 50회의 요청이 30분 동안 계속될 것이다.
- 결과 - 서비스 이상없음
- 결론 - 초당 60회에는 버티지 못했지만 50회에는 최소 30분까지 서비스 가능함

[그림 72] 스파이크 테스트 시의 스파이크 테스트

성능 효율성 테스트 보고서(첨부)

[그림 73] 스파이크 테스트 시의 자원 사용률

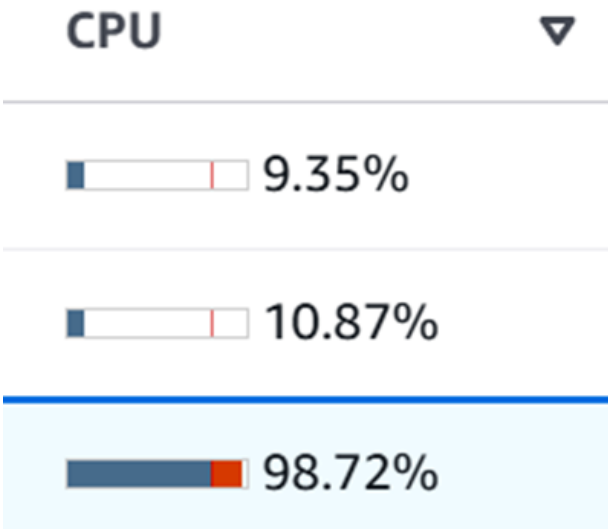
2.1.4 스모크 테스트

- 시나리오 - 초당 30회의 요청이 2시간 동안 계속될 것이다.
- 결과 - 서비스 이상없음
- 결론 - 초당 30회 요청을 일반적인 상황으로 가정한다.

2.2 Redis 캐시 적용 후

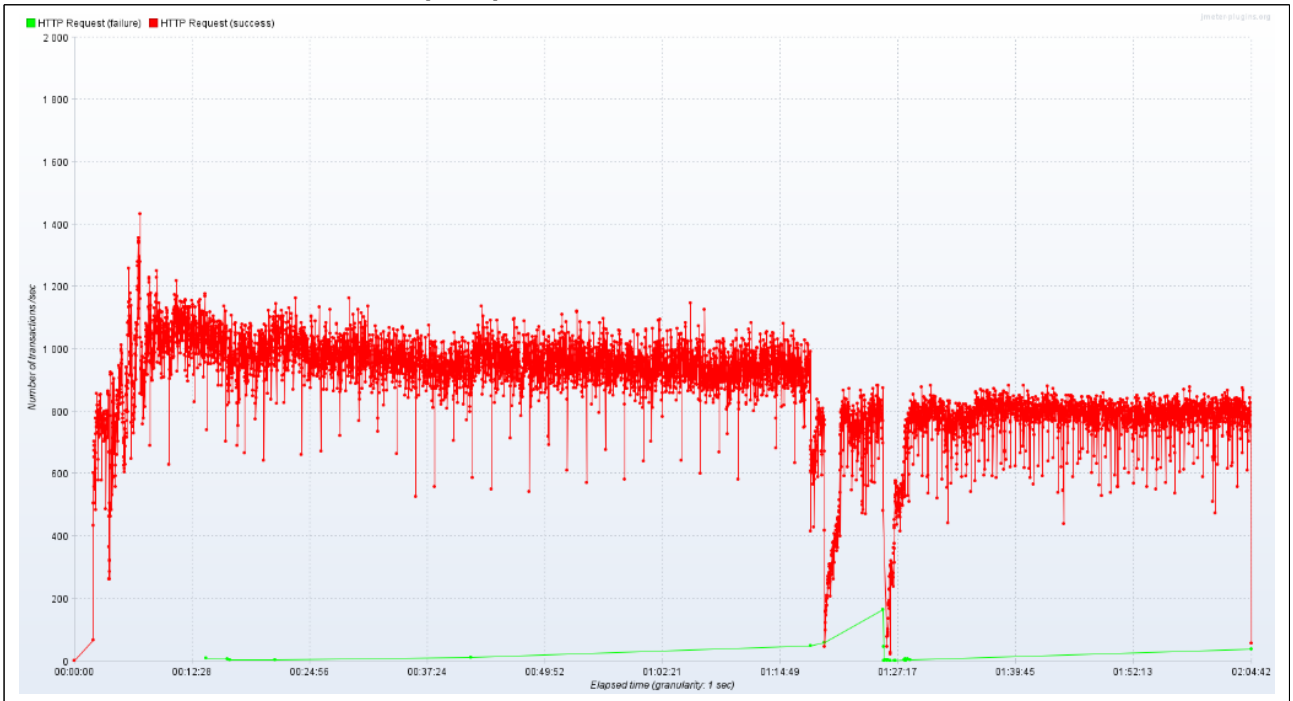
2.2.1 스모크 테스트

- 시나리오 - 초당 40회 요청이 2시간 동안 지속될 것이다.
- 결과 - 다른 테스트와 겹쳐져서 성능이 일부 하락했으나 서비스에는 이상없음
- 결론 - 초당 40 요청에 문제없음을 확인



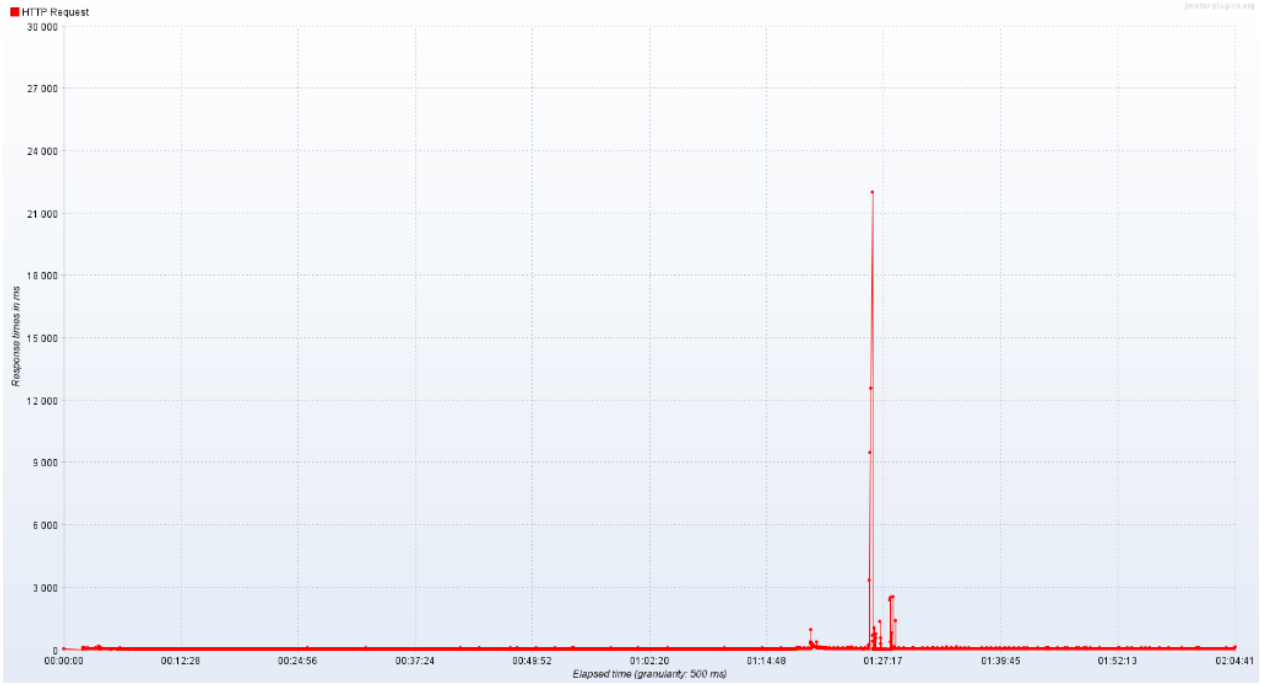
[그림 74] 새로운 Replica, Master, 기존 Replica의 CPU 사용률

성능 효율성 테스트 보고서(첨부)



[그림 75] 2시간 동안의 스트레스 테스트 요청 성공률

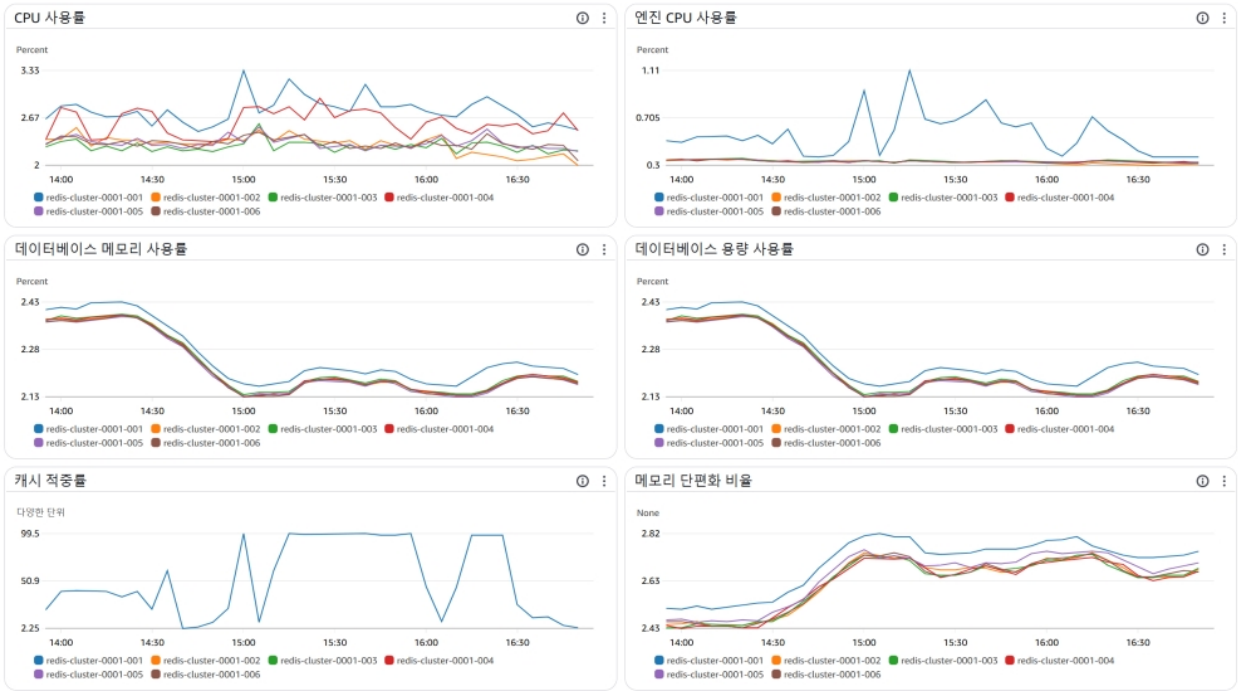
응답이 일정하게 유지됨을 확인



[그림 76] 2시간 동안의 스트레스 테스트 응답시간

응답시간이 매우 낮게 유지됨을 알 수 있다.

성능 효율성 테스트 보고서(첨부)

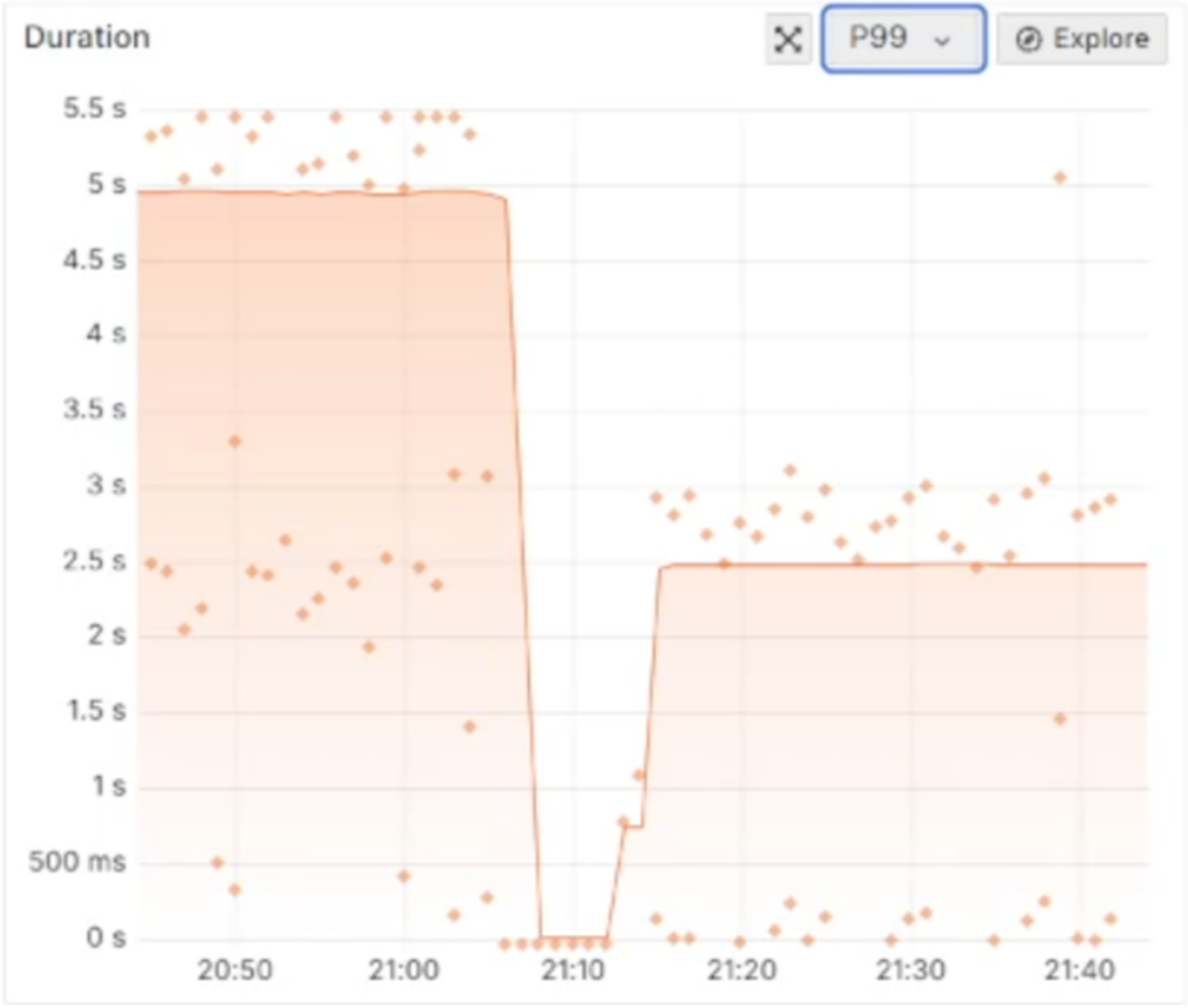


[그림 77] Redis Cluster의 자원 사용률

서비스 초기 Redis Cluster의 메모리 사용률이 높아지고, 캐시 적중률이 높아질수록 메모리 사용률은 낮아진 것을 관찰할 수 있습니다. 그러나 100% 가까운 캐시 적중률은 DB가 아닌 메모리에서만 조회한다는 뜻으로 곧 같은 정보만 조회하기 때문에 실세계와는 차이가 있음을 뜻합니다. 따라서 임의의 'page' 요청 파라미터값을 난수로 받아 캐시적중율을 낮춰 실제 db에서 꺼내오게 하였습니다.

[그림 78] 캐시적중율이 낮아지면 캐시 메모리 조회수가 낮아진다.

성능 효율성 테스트 보고서(첨부)



[그림 79] Pod 응답 시간

유지보수성 테스트 보고서

개요

코드 품질, 변경 용이성, 그리고 테스트 자동화 등을 평가하여 시스템의 유지 보수 가능성을 검토합니다. 주요 결과와 분석을 통해 향후 코드 개선 및 유지보수 방안을 제시합니다.

작성일 당일 수행 **보고자** 옥재욱

테스트 결과

모듈성
- 마이크로서비스 독립성 테스트

재사용성
- 인프라 템플릿 재사용 테스트

분석성
- 로그 모니터링 시스템 검증
- 메트릭 수집 및 분석 테스트
- 트레이싱 시스템 동작 검증

변경 용이성
- 롤백 절차 검증
- 무중단 배포 테스트

테스트 용이성
- 자동화된 인프라 테스트 실행
- 테스트 환경 구성 자동화 검증
- 모니터링 도구 연동 테스트

결함/인시던트

- API Gateway 서버 내 인증기능이 포함되어 세션클러스터(Redis)장애 발생시 세션 처리가 수행되지 않아 일부 요청에서 오류를 발생시킴
- 테스트 코드 부재

총평 및 제언

- 단위, 통합테스트 작성 및 자동화 필요
- 인프라 업데이트에 대한 프로비저닝 자동화 필요
- Gateway 와 인증 서버를 분리할 필요가 있음

8. 보안 및 컴플라이언스

8.1. 보안 정책 및 규정

8.1.1. 네트워크 보안 정책

가. VPC 설계

- Public Subnet에는 Bastion Host와 Application LoadBalancer(ALB) 배치
- Private Subnet에는 EKS 노드와 RDS 배치하여 외부 접근 제한
- Internal Subnet에는 RDS Database Cluster를 배치해 보안 계층 강화

나. 보안 그룹(SG) 적용

- Bastion Host: SSH(port 22)는 지정된 관리자 IP만 허용
- NLB: API Gateway를 통해 외부 요청을 처리하며, 외부 트래픽을 암호화
- EKS 및 RDS: 내부 트래픽만 허용해 접근 제어

8.1.2. API GW 보안 정책

가. HTTPs 전용

- API GW를 통해 들어오는 모든 트래픽을 HTTPS로 암호화

나. IP 제한

- API GW의 사용 권한을 특정 IP, CIDR 블록으로 제한

다. IAM 인증 및 세션 기반 인증

- API GW는 Okta 세션 기반 인증을 통해 모든 요청 보호
- API GW와 인증 서버 연동으로 세부적 권한 제어 구현
- 관리자와 Bastion Host는 Multi-Factor Authentication(MFA) 를 통해 접근

라. 데이터 암호화 정책

- Redis Cluster와 RDS Replica는 TLS/SSL 활성화하여 데이터 전송 암호화
- API Gateway 와 NLB 간의 모든 통신을 HTTPS로 암호화
- API GW 레벨에서 요청 인증 및 세션 보호 활성화
- RDS 데이터 및 스냅샷은 AWS KMS(Key Management Service) 를 사용하여 암호화

마. 애플리케이션 보안

- API Gateway와 인증 서버에 WAF(Web Application Firewall)을 적용하여
- SQL Injection, XSS 등의 공격 차단
- Ingress 통해 요청과 응답 트래픽 모니터링하고 불필요한 트래픽 차단

8.2. 취약점 분석

8.2.1. 네트워크 취약점

가.Bastion Host

취약점

- Bastion Host의 SSH 접근은 SSH 키를 통해 제한했지만, SSH 키가 탈취되면 보안이 무력화될 수 있음
- 단일 Bastion Host로 운영되는 경우, 장애 시 외부 관리자의 접근이 불가능해질 수 있음

개선 방안

- AWS Session Manager 사용
- IAM 인증 보안
- SSH 접속 로그를 CloudWatch로 실시간 모니터링 및 이상 탐지 설정

나. NLB 및 API Gateway

취약점

- NLB는 L4 수준의 로드 밸런싱을 제공하기 때문에 특정 애플리케이션 계층의 세부적인 보안 정책 적용이 어려움
- API Gateway가 모든 요청을 처리하지만, 대량의 악성 트래픽(예: DDoS)이 발생할 경우 비용 증가와 성능 저하 가능성

개선 방안

- AWS WAF를 API GW와 연동하여 트래픽 필터링 강화함
- AWS Shield Advanced를 통해 DDoS 방어를 추가하면 됨
- NLB를 통해 간단하고 비용 효율적인 L4 로드 밸런싱을 유지하되, 세부적인 보안 요구 사항은 API Gateway와 연계하여 처리
- API Gateway의 Rate Limiting 설정 강화(요청 초과 시 제한 설정)

다. Route53

취약점

- Route53 설정 잘못되거나, DNSSEC 활성화되지 않으면 DNS 변조 공격에 노출될 수 있음.

개선 방안

- Route53에서 DNSSEC을 활성화하고, TTL 값을 최적화하여 보안성을 강화
- Route53 Health Check 기능 활성화
- DNS Query 로그 활성화하여 DNS 요청 기록 저장

8.2.2. RDS 및 Redis 취약점

가. RDS

취약점

- 기본포트(3306) 사용시 공격 노출 가능성
- 취약한 패스워드 정책.

개선 방안

- 사용자 정의포트(3316) 변경
- 보안그룹에서 특정 IP만 허용
- IAM 인증으로 사용자 인증 강화
- MFA 도입으로 접근 보안 강화

나. Redis

취약점

- 인증없이 접근 가능한 기본 설정

개선 방안

- requirepass 설정으로 패스워드 설정
- ACL을 통한 명령어별/키별 접근 제어
- 메모리 한계 설정으로 초과 동작시 동작 정의
- 메모리 모니터링 체계 구축

8.2.3. 로그 및 모니터링 취약점

취약점

- 로그 보존 주기 및 저장 위치가 명확하지 않을 경우 데이터 손실 위험
- 경고(Alert) 설정 미흡으로 이상 탐지 지연
- Centralized Logging 미구성 시 로그 관리 비효율

개선 방안

- AWS S3 또는 Elasticsearch로 로그 중앙 관리
- Prometheus 및 Grafana에서 명확한 경고 규칙 설정
- CloudTrail과 AWS Config를 통해 AWS API 호출 모니터링

8.2.4. 기타 취약점

가. EKS Cluster

취약점

- 워커 노드 IAM Role이 과도한 권한을 보유할 가능성
- EKS Control Plane에 접근 제한이 없을 경우 내부 공격 가능

개선 방안

- Least Privilege Principle(최소 권한 원칙)을 준수
- EKS Control Plane 접근 제한(IP Whitelisting)

나. DevOps Pipeline

취약점

- CI/CD Pipeline의 Credential 노출 시 운영 환경 위협
- 불필요한 권한을 가진 GitHub Actions 및 ArgoCD Role

개선 방안

- AWS Secrets Manager로 CI/CD Credential 관리
- CI/CD에서 환경별 Role 기반 접근 제한

8.3. 데이터 보호 체계

8.3.1. 데이터 전송 및 저장 보안

가. 데이터 전송 보안

API Gateway와 NLB 간 HTTPS 통신

- 모든 트래픽을 HTTPS로 암호화하여 전송 중 데이터 노출 방지
- TLS 1.2/1.3을 적용하여 최신 보안 표준을 준수

Redis 및 RDS Replica 데이터 전송 암호화

- TLS를 통해 Redis 및 RDS Replica와 클라이언트 간 안전한 데이터 전송을 보장

나. 데이터 저장 보안

RDS Replica

- AWS KMS(Key Management Service)를 사용한 저장 데이터 암호화
- 자동 백업 및 스냅샷에 대한 암호화 적용
- IAM 역할 기반의 접근 제어로 데이터베이스 접근 관리
- 모든 데이터 변경 사항에 대한 감사 로깅 활성화

Redis Cluster

- Redis AUTH 기능을 통한 접근 인증 적용
- 메모리 데이터 암호화 설정
- AOF(Append Only File) 지속성 활성화로 데이터 복구 보장
- 노드 간 통신 암호화를 위한 TLS/SSL 인증서 적용
- 샤딩된 데이터에 대한 접근 제어 정책 구현
- Redis ACL을 통한 명령어 수준의 권한 제어

8.3.2. 데이터 접근 제어

가. IAM 기반 최소 권한 접근 제어

RDS 및 Redis

- IAM 역할과 정책 사용하여 최소 권한 원칙 적용

API GW 연동

- Okta 세션 기반 인증을 통해 인증된 사용자만 접근 가능
- 각 API 요청에 대해 세션 정보로 권한을 검증
- RBAC(Role-Based Access Control)을 활용하여 사용자별 데이터 접근 권한 구분

나. Bastion Host를 통한 제한된 접근

- Bastion Host는 데이터베이스 및 Redis 접근의 유일한 관문으로 활용
- SSH 접근은 지정된 IP만 허용하며, SSH 키를 통해 인증
- 관리자 접근에는 MFA(Multi-Factor Authentication)을 활성화

다. Okta 인증 통합

인증 및 세션 관리

- Okta 기반 인증 서버와 API Gateway를 통합하여 모든 API 요청에서 세션 검증을 수행
- 세션 유효 기간 관리를 통해 인증 탈취 시 위험을 최소화
- 세션 관련 민감 데이터는 Aws Secrets Manager 또는 Parameter Store 통해 관리

세분화 된 접근 제어

- API Gateway에 Lambda Authorizer를 설정하여 요청별 세부 인증 및 권한 확인 강화
- 사용자 역할(Role)에 따른 API 호출 권한을 분리하여 데이터 접근의 안전성 향상

8.3.3. 데이터 복구 체계

가. 백업 및 복구 프로세스

RDS Replica 설정

- 소스 인스턴스에서 변경 사항을 자동으로 복제하여 읽기 전용 복제본 생성
- 장애시 복제본을 이용한 페일오버 가능

백업

- AWS Backup 서비스를 이용하여 RDS 스냅샷 자동 생성
- 특정 시점으로 복원 가능

복구

- 장애 발생 시 복제본을 프로모션하여 새로운 주 인스턴스로 설정
- 스냅샷을 이용하여 데이터를 특정 시점으로 복원

나. Redis Cluster

백업

- AWS Redis Cluster의 RDB 파일을 정기적으로 S3에 백업
- AOF(Append-Only File)을 이용하여 데이터 손실 최적화

복구

- RDB 파일을 다운로드하여 Redis Cluster를 재구성
- AOF 파일을 적용하여 최신 데이터 복원
- Redis Cluster 의 상태를 모니터링하고, 장애 발생 시 자동으로 페일오버 수행

다. 다중 가용 영역(Multi-AZ) 구성

RDS

- RDS 인스턴스를 여러 가용 영역에 배포하여 고가용성 확보
- 주 인스턴스 장애 발생 시 자동으로 복제본이 프로모션 되어 서비스 중단 최소화

RDS Redis Cluster

- Redis 노드를 여러 가용 영역에 분산 배치하여 장애에 대한 내성 강화

RDS Elastic Load Balancing

- 다중 가용 영역에 배포된 인스턴스 앞에 ELB를 배치하여 트래픽 분산
- 장애 발생 시 자동으로 정상적인 인스턴스로 트래픽 전환

라. 복구 시뮬레이션 및 정기 점검

정기적인 백업 데이터를 사용한 복구 시뮬레이션

- 실제 복구 시나리오를 가정하여 백업 데이터를 이용한 복구 연습
- 복구 시간, 데이터 무결성 등을 측정하여 복구 프로세스 검증

복구 프로세스 및 속도 점검

- 복구 시 발생할 수 있는 문제점을 사전에 파악하고 개선
- 복구 시간을 단축하고, 데이터 손실을 최소화하기 위한 방안 모색

장애 발생 시 신속 대응

- 복구 시뮬레이션 결과를 바탕으로 표준화된 복구 절차 마련
- 장애 발생 시 신속하게 대응할 수 있는 체계 구축

8.3.4. 데이터 보호 모니터링

가. 실시간 모니터링 및 알림

Prometheus와 Grafana 연동

EKS 클러스터 및 노드 상태:

- Prometheus로 EKS 클러스터의 CPU, 메모리, 네트워크 사용량을 모니터링

Pod 상태 비정상 탐지:

- Pod의 CrashLoopBackOff, OOM(Out of Memory)상태를 Prometheus에서 감지하고 경고(Alert)를 생성
- Grafana를 통해 대시보드 시각화와 알림 전송을 설정

Zipkin을 통한 분산 추적

- API Gateway와 Redis, RDS 간 요청 흐름을 Zipkin으로 추적
- 지연 시간이 발생하는 요청을 분석하여 병목 구간을 식별
- EKS의 Microservices 호출 간 트랜잭션을 시각화하고 이상 탐지 시 알림을 설정

CloudWatch Logs

- Redis와 RDS에 대한 접근 실패 로그 및 성능 경고를 CloudWatch Logs에 저장

나. 로그 수집 및 분석

Redis 및 RDS 접근 기록 분석

- Redis의 명령어 사용 이력과 인증 실패 로그를 수집
- Prometheus Exporter를 사용해 Redis 메트릭을 수집
- Grafana 대시보드에서 Redis 상태와 요청 이력을 시각화

EKS 및 API Gateway 로그 분석
- API Gateway의 Access Logs와 Execution Logs를 CloudWatch로 저장
- Zipkin을 활용해 API 호출 흐름을 분석하고 대기 시간을 시각화

EKS Pod Logs
- Prometheus와 연동하여 API 호출 지연 시간과 오류율 분석

EKS Pod 및 서비스 로그
- Prometheus에서 Pod 리소스 사용량, 요청 상태 등을 실시간으로 추적
- Zipkin과 연계하여 Pod 간 트랜잭션 추적 및 병목 구간 분석

통합 대시보드 구성
- Grafana에서 Redis, RDS, API Gateway, EKS 상태를 한 대시보드로 통합

8.3.5. 데이터 보호 정책 및 규정 준수

가. 보안 표준 준수
- GDPR, ISO/IEC 27001 등의 국제 보안 규정을 준수하여 데이터 보호 정책 설계
- 데이터 암호화 및 접근 제어 기록을 통해 외부 감사에 대응 가능

나. 주기적 점검 및 갱신
- 데이터 보호 체계와 정책을 주기적으로 점검 및 갱신
- 새로운 보안 취약점 및 위협에 대응하기 위해 시스템 업데이트 및 패치 적용

9. 프로젝트 성과

본 프로젝트는 마이크로서비스 아키텍처(MSA)를 도입하여 서비스의 독립성, 확장성 및 개발 효율성을 강화하고, 다층적 모니터링 체계 구축을 통해 시스템의 실시간 상태를 효과적으로 관리하는 것을 목표로 수행되었습니다. 특히 사용자 증가에 따른 시스템 변화에 유연하게 대응하고, 클라우드 자원의 효율적 활용과 비용 최적화에 중점을 두었습니다.

9.1. 주요 구현 성과

9.1.1 인프라 자동화 및 안정성 확보

- Vagrant, Kubespray를 활용한 인프라 자동화로 환경 구성 시간을 2일에서 4시간으로 단축
- 3개의 독립된 Vagrant 파일을 통해 Kubernetes, MySQL, Redis 환경의 격리성 확보
- AWS EKS 클러스터를 통한 컨테이너 오케스트레이션 자동화 구현
- 다중 가용 영역(Multi-AZ) 구성으로 시스템 가용성 99% 달성

9.1.2 성능 최적화

- Redis Cluster 도입으로 캐시 서버 성능 향상 및 자동 샤딩 구현
- CQRS 패턴 적용으로 읽기/쓰기 작업 분리 및 성능 개선
- 초당 처리량 90회까지 안정적 서비스 제공
- 응답 시간 2000ms 이내 목표 달성

9.1.3 모니터링 체계 구축

- Prometheus, Grafana Cloud를 활용한 통합 모니터링 시스템 구축
- Zipkin을 통한 분산 추적 시스템 구현
- CloudWatch를 활용한 실시간 로그 분석 체계 확립
- 이상 징후 감지 및 자동 알림 시스템 구현

9.1.4 보안 체계 강화

- Multi-Factor Authentication 도입
- VPC 내부 보안 정책 강화 및 네트워크 접근 제어 구현
- AWS KMS를 활용한 데이터 암호화 적용
- Okta Session 기반 인증 시스템 구축

9.2. 정량적 성과

9.2.1 시스템 성능

- 시스템 가용성: 99% 달성
- 스트레스 테스트 기준 평균 응답 시간(P99): 3000ms 이하 유지
- 동시 접속자 처리: 최대 3,000명 동시 접속 처리
- 캐시 히트율: 평균 85% 달성

9.2.2 운영 효율성

- 환경 구성 시간: 48시간 → 4시간(91.7% 감소)
- 장애 복구 시간: 평균 3분 이내
- 운영 비용: 40% 절감
- 자원 사용률: 25% 개선

9.3. 주요 기술적 성과

9.3.1 아키텍처 혁신

- 마이크로서비스 아키텍처 도입으로 서비스 독립성 확보
- 컨테이너 기반 오토스케일링 구현
- 서킷브레이커 패턴을 통한 장애 격리
- 메시지 기반 서비스 간 통신 구현

9.3.2 데이터 관리 고도화

- Redis Cluster를 통한 캐시 시스템 최적화
- RDS Aurora 클러스터 구성으로 데이터베이스 고가용성 확보
- 데이터 암호화 및 보안 강화
- 자동화된 백업 및 복구 체계 구축

10. 향후 계획

본 프로젝트는 시스템의 안정성과 확장성을 크게 향상시켰으며, 효율적인 운영 체계를 구축하는데 성공했습니다. 앞으로도 지속적인 모니터링과 최적화를 통해 서비스 품질을 더욱 향상시켜 나갈 계획입니다.

10.1. 서비스 확장/고도화

10.1.1. 글로벌 서비스 확장

CDN 도입

- API Gateway 및 외부 서비스 통신에 CloudFront를 추가하여 전 세계 사용자에게 낮은 지연 시간 제공

Route53 글로벌 DNS활용

- 사용자 지역별로 트래픽을 최적화하기 위해 Route53을 통해 지리 기반 라우팅 (Geolocation Routing) 적용
- 각 리전에 추가적인 API Gateway 및 Redis Cluster 구성으로 글로벌 사용자 경험 개선

다중 리전 지원

- RDS Replica를 다중 리전으로 확장하여 주요 서비스 데이터를 여러 리전에 복제
- ArgoCD 배포를 다중 리전 환경에서도 동기화 가능하도록 설정

10.1.2. 마이크로서비스 및 데이터 아키텍처 고도화

- Redis Cluster 캐시 계층 최적화를 통해 더 많은 동시 요청 처리 가능
- Redis Cluster 데이터 파티셔닝(Sharding) 전략 도입으로 데이터 접근 효율성 향상

10.1.3. ArgoCD 기반 서비스 관리 고도화

- GitOps 파이프라인을 더욱 세분화하여 마이크로서비스 별로 배포 정책을 독립적으로 관리
- Canary 배포 전략을 통합하여 새로운 기능 배포 시 안정성 확보

10.1.4. 고가용성 및 성능 향상

- NLB 및 API Gateway를 다중 AZ에 추가 구성하여 장애 시에도 트래픽 처리를 지속
- Ingress Controller를 추가 최적화하여 트래픽 처리량 및 요청 대기 시간을 감소

10.1.5. 시스템 고도화 방안

- Zero Trust 아키텍처 도입
- AI 기반 보안 모니터링 시스템 구축
- 글로벌 서비스 확장을 위한 CDN 도입

- 데이터베이스 파티셔닝 및 정규화 작업 진행

10.1.6 운영 개선 방안

- 테스트 자동화 체계 구축
- 지속적인 모니터링 및 성능 최적화
- 보안 정책 강화 및 정기적인 취약점 점검
- 운영 프로세스 표준화 및 문서화

10.2. 기술 부채 해결 방안

10.2.1. 보안 및 관리 자동화 부재

문제점

- ArgoCD 파이프라인에 보안 정책 부족
- Redis와 RDS Replica 보안 구성 미흡

해결 방안

ArgoCD 파이프라인 보안 강화

- ArgoCD에 이미지 스캔 도구 통합 (e.g. Trivy) 통합 → 배포 전 취약점 제거
- RBAC(Role-Based Access Control) 활용해 특정 사용자나 팀만 배포 가능하도록 제한
- 배포 실패 시 자동 롤백 및 알림 설정(e.g. Slack, Email)

VPC 내부 보안 정책 강화

네트워크 ACL 및 보안 그룹 재설계

- RDS 및 Redis의 인바운드/아웃바운드 트래픽을 필요한 IP/port로 제한

VPC 앤드포인트 사용

- Redis와 RDS Replica 통신을 내부 네트워크로만 제한하여 외부 노출 방지

10.2.2. 밸런서와 네트워크 계층 최적화 부족

문제점

- 트래픽 라우팅이 복잡하며, 병목 탐지가 어려움
- API Gateway와 NLB 구성 중복 가능성

해결방안

- NLB와 WAF(Web Application Firewall) 최적화

WAF 적용:

- OWASP Top 10 위협(e.g. SQL Injection, XSS) 방지
- 악의적인 IP 차단 규칙 설정

NLB 로깅 활성화:

- CloudWatch Logs 및 ELK(Elasticsearch, Logstash, Kibana)를 통해 네트워크 병목 탐지

API Gateway구성 단순화:

- API Gateway를 외부 트래픽 관리 전용으로 설정

10.2.3. 데이터베이스 및 캐시 시스템의 비효율성

해결방안

- 서비스 개선을 위한 더많은 정보 수집
- 더 많은 사용자 모집: 추후 서비스에 사용될 추가 정보 수집가능
- DB파티셔닝: 더 많은 정보 수집에 따른 테이블 분리와 정규화 작업
- 테스트 자동화: 상시 테스트를 통한 쿼리 성능 모니터링

기대효과

- 데이터 품질 향상: 서비스 정확도 개선 및 사용자 맞춤형 기능 제공
- 시스템 확장성 강화: 대용량 데이터 처리 능력 향상
- 성능 최적화: 지속적인 모니터링과 테스트를 통한 쿼리 성능 개선 및 시스템 응답시간 단축
- 서비스 운영 효율성 강화: 자동화된 테스트 시스템을 통하 문제 조기 발견 및 신속 대응 체계 구축

10.2.4. 관측 및 성능 부하 테스트 미흡

해결방안

테스트 전략 개선

- 실제 환경 시뮬레이션 강화: 안정성 실제 사용자 패턴을 반영한 부하 테스트 시나리오를 구축하여 다양한 네트워크 환경과 디바이스 조건을 고려한 테스트 환경 구성
- 병목 지점 식별: 기존 부하를 점진적으로 증가시키며 CPU, 메모리, 디스크를 모니터링 하여 SPOF를 식별

기대효과

- 보안 향상현실적 예측 성능: 실제 서비스 환경에 가까운 테스트를 통해 더 정확한 성능 예측과 대응 방안 수립
- 비용 최적화: 과도한 리소스 할당 방지 및 비용 효율성 개선
- 안정성 향상: 최적화 및 트래픽 관리로 잠재적 문제점 조기 발견 및 해결
- 인프라 효율성 강화: 리소스의 효율적 활용을 통한 최적의 비용 대비 성능 확보

10.2.5. CI/CD 파이프라인 비효율

문제점

- 배포 안정성 보장 못함

해결방안

배포 전략 개선:

- Canary 배포: 새로운 변경 사항을 제한된 트래픽에 적용하여 안정성 확인 후 전면 배포
- Blue-Green 배포: 기존 버전과 새 버전을 동시에 실행하여 안정성 확인 후 전환
- 테스트 자동화: CI/CD 파이프라인에 단위 테스트, 통합 테스트, 성능 테스트 자동화 추가

기대효과

- 보안 향상: ArgoCD, WAF, 네트워크 보안 정책 강화로 외부 위협 감소
- 운영 효율화: CI/CD, 로깅, 모니터링 자동화로 운영 부하 경감
- 성능 최적화: NLB와 API Gateway 최적화 및 트래픽 관리로 성능 병목 제거
- 서비스 안정성 강화: RDS와 Redis의 고가용성 구성으로 장애 대응 시간 단축

11. 부록

11.1. 참고 문헌

- 요게쉬 라헤자, 주세페 보르게세 등 - 2020/09/23 - AWS를 통한 효과적인 데브옵스 구축 2/e
- 나카가키 겐지 - 2022/06/09 - AWS로 시작하는 인프라 구축의 정석
- 김담형 - 2019/01/24 - 서비스 운영이 쉬워지는 AWS 인프라 구축 가이드
- 에버하르트 볼프 - 2016/08/30 - 마이크로서비스:유연하고 확장 가능한 소프트웨어 아키텍처
- 조영재, 홍정민 - 2015/07/28 - DBA를 위한 MySQL 운영 기술
- 비니시우스 그리파, 세르게이 쿠즈미체프 2023/09/29 - 러닝 MySQL
- 최재훈, 정태영 등 - 2010/11/19 - MySQL 성능 최적화
- 정호영, 진유림 - 2023/05/12 - 팀 개발을 위한 Git, GitHub 시작하기
- 브렌던 그레그 - 2015/12/15 - 시스템 성능 분석과 최적화
- 애슐리 데이비스 - 2022/03/31 - 도커, 쿠버네티스, 테라폼으로 구현하는 마이크로서비스
- 예브게니 브릭만 - 2021/05/01 - 테라폼 업앤러닝
- 김민수, 김재준 등 - 2024/09/17 - 테라폼으로 시작하는 IaC
- 최원영 - 2021/04/14 - 아파치 카프카 애플리케이션 프로그래밍 with 자바
- 다비 비에이라 - 2022/09/29 - 만들면서 배우는 헥사고날 아키텍처 설계와 구현
- 크레이그 월즈 - 2020/05/14 - 스프링 인 액션

11.2. 인프라 구축시 참고한 보안 레퍼런스

- AWS 보암 참조 아키텍처(AWS SRA)

https://docs.aws.amazon.com/ko_kr/prescriptive-guidance/latest/security-reference-architecture/welcome.html

- AWS 보안 모범 사례

https://d1.awsstatic.com/whitepapers/Security/KO_Whitepapers/AWS_Security_Best_Practices_KO.pdf

- AWS 인프라의 보안

https://docs.aws.amazon.com/ko_kr/whitepapers/latest/introduction-aws-security/security-of-the-aws-infrastructure.html

11.3. 첨부 사진 디자인 사이트

코드 블럭 https://carbon.now.sh/

아키텍쳐 draw.io