ChatGPT

AKS 기반 하이브리드 마이크로서비스 아키텍처 및 운영 전략

1. 프로젝트 요약

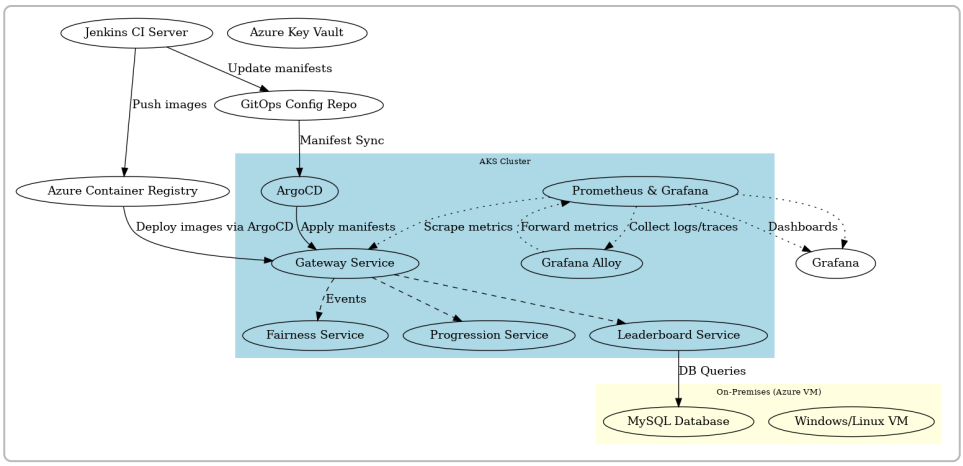
go-microservices 프로젝트는 게임 **LiveOps** 환경의 핵심 기능들을 마이크로서비스로 구현한 백엔드 시스템입니다 [1]. Gateway, Fairness, Progression, Leaderboard 총 **4개의 Go 마이크로서비스**로 구성되어 있으며, 각 서비스는 게임 이벤트 처리와 실시간 모니터링/공정성 보장을 위해 협업합니다. 주요 구성과 역할은 다음과 같습니다:

- **Gateway 서비스 (포트 8080)** – 클라이언트/게임에서 발생하는 이벤트를 수집하는 진입점입니다. 요청에 대한 **HMAC-SHA256 서명 검증**과 **중복 요청 방지 (Idempotency 키)**를 수행하고, 유효한 이벤트를 메시지 버스로 전달합니다 [2] [3]. 이를 통해 이벤트 위변조나 폭주를 초기에 차단합니다.
- **Fairness 서비스 (포트 8081)** – 게임 **이벤트율과 점수의 이상 패턴**을 감지하여 **부정행위**나 비정상 상황을 탐지합니다. 플레이어별 초당 이벤트 발생률을 모니터링하고 비정상 점수 상승을 감지해 정책에 따라 차단(HTTP 429/403 응답)합니다. 또한 차단된 이벤트 수(`dropped_events_total`)나 이상징후 건수(`anomaly_flags_total`) 등의 **Prometheus 메트릭**을 노출하여 시스템 공정성 상태를 관측합니다 [4].
- **Progression 서비스 (포트 8083)** – 경험치 증가, 보스 처치, 아이템 획득 등 **게임 내 진척도 관련 이벤트**를 처리합니다. **Redis 캐시**를 활용해 실시간 처리를 하고, 최종 데이터는 **PostgreSQL**에 영구 저장합니다 [5]. 또한 보상 지급 요청 시 **HMAC 서명된 영수증**을 발행하여 보상 무결성을 보장합니다 [6].
- **Leaderboard 서비스 (포트 8082)** – 일간/주간/시즌별 **랭킹 집계**를 담당합니다. Redis의 **정렬된 집합 (ZSET)** 구조를 이용해 실시간 순위를 계산하고 상위 N명을 제공하며 [7] [8], 일정 주기마다 Redis 데이터를 PostgreSQL에 스냅샷으로 아카이빙하여 영구 보관합니다 [9]. 다양한 기간의 랭킹 윈도우(daily/weekly/seasonal)를 지원하고, 명예의 전당 데이터는 주기적으로 저장소에 적재됩니다.

이러한 마이크로서비스들은 **경량 메시지 버스**를 통해 느슨하게 결합되어 이벤트를 전달받고 처리합니다. 초기 개발 단계에서는 단순 인메모리 버스를 사용하지만, 클라우드 환경에서는 **AWS SQS 또는 Azure Service Bus**로 손쉽게 대체할 수 있도록 추상화되어 있습니다 [10] [11]. 공통 **코어 패키지**에서는 데이터 모델 정의, 이벤트 검증 로직, HMAC 서명/검증, 영수증 생성 등의 로직을 제공하여 각 서비스가 일관된 방식으로 동작하도록 합니다 [12]. 또한 모든 서비스에 **헬스체크** 엔드포인트(`/healthz`)와 **구조화된 JSON 로그**, **Prometheus 매트릭 노출** 기능을 포함하여 운영 시 관측성을 확보했습니다 [13].

> **요약:** go-microservices 프로젝트는 멀티 플레이어 게임의 **공정성(Fairness) 보장**과 **진척도 관리**를 위해 4개의 마이크로서비스로 구성된 백엔드이며, **HMAC 보안, Redis 캐시, 이벤트 큐, Prometheus 모니터링** 등의 기술을 활용하여 대규모 트래픽에서도 신뢰성과 성능을 달성했습니다 [14]. 모든 서비스는 **Docker 컨테이너**로 패키징 가능하며, AWS와 Azure를 모두 지원하는 멀티클라우드 어댑터를 통해 다양한 인프라에 배포할 수 있는 형태로 구현되었습니다 [15].

2. 전체 인프라 아키텍처 구성



AKS 기반 하이브리드 인프라 아키텍처 개요: Azure 클라우드의 AKS 클러스터와 온프레미스 VM(MySQL 서버)을 결합한 구성.

위 다이어그램은 **Azure Kubernetes Service(AKS)**를 중심으로 한 하이브리드 아키텍처를 나타냅니다. **Azure 클라우드** 상의 AKS 쿠버네티스 클러스터에는 앞서 설명한 4개의 마이크로서비스(Gateway, Fairness, Progression, Leaderboard)가 **컨테이너**로 배포되어 있으며, 온프레미스 환경을 시뮬레이션하기 위해 **Azure VM** 한 대가 별도로 구성되어 **MySQL 데이터베이스**를 구동하고 있습니다 [16] [17]. 이는 실제 기업 환경에서 사내 IDC에 존재하는 **레거시 DB**나 서비스와 클라우드 네이티브 애플리케이션이 동시에 사용되는 시나리오를 반영한 것입니다 (예: 클라우드의 마이크로서비스가 온프레미스 DB에 연결) [18].

AKS 클러스터와 온프레미스 VM은 동일한 **가상 네트워크(VNet)**에 속하며, **VPN 또는 ExpressRoute**를 통해 안전하게 연결된 형태로 가정합니다. VM에는 MySQL 서버가 설치되어 있으며, AKS 클러스터에서 해당 VM의 DB 포트(예: 3306)가 접근 가능하도록 **NSG(Network Security Group)** 규칙이 설정되어 있습니다 [19].

클러스터 내부에는 애플리케이션 외에도 GitOps CD 도구인 **ArgoCD**, 모니터링 스택(**Prometheus, Grafana, Alertmanager, Grafana Alloy**) 등이 함께 실행됩니다. 개발자는 Git 저장소에 Kubernetes 매니페스트를 커밋하며, ArgoCD는 이를 감지하여 클러스터 상태를 자동으로 업데이트합니다 [20] [21]. 또한 CI 파이프라인을 담당하는 **Jenkins 서버**가 별도로 구성되어 (예: VM이나 Jenkins 서비스 이용) 애플리케이션 코드를 빌드하고 **Docker 이미지**를 생성한 뒤 Azure의 **컨테이너 레지스트리(ACR)**에 푸시합니다. 새로운 이미지가 빌드되면, ArgoCD는 해당 이미지를 적용하도록 매니페스트를 동기화하여 배포를 트리거합니다 (GitOps 방식) [21].

정리하면, **클라우드(AKS)**의 확장성과 **온프레미스 VM**의 레거시 연동을 모두 활용한 구조입니다. AKS 마이크로서비스들은 MySQL DB가 동작하는 VM과 통신하여 데이터 조회/저장을 수행하고, 모니터링 에이전트(Alloy)가 양쪽 환경의 메트릭을 수집하여 중앙 **Grafana 대시보드**로 시각화합니다. 이러한 구성은 펄어비스처럼 자체 IDC와 Azure 클라우드를 혼용하는 환경에서도 일관된 운영 관리가 가능하도록 설계되었습니다 [22].

3. 주요 리소스 구성 🗃️

이 아키텍처에서 활용되는 **주요 Azure 리소스 및 구성요소**는 다음과 같습니다:

- **Azure Kubernetes Service (AKS)**: 마이크로서비스들을 배포/운영하는 **관리형 Kubernetes 클러스터**입니다. 예시로 Standard_DS3_v2 노드 3개로 구성된 클러스터를 사용하며, Azure CNI 기반 VNet 통합으로 온프레미스 VM과 동일 가상네트워크 상에서 통신합니다. 클러스터는 **매니지드 서비스**로서 제어 플레인을 Azure가

관리하고, 노드 풀에 대해서만 관리하면 되므로 운영 부담을 줄여줍니다 [23] . 또한 AKS에 **Managed Identity**를 부여하여 ACR, Key Vault 등의 자원 접근 권한을 직접 가지도록 설정했습니다 [24] .

- **Azure Container Registry (ACR)**: Docker 이미지를 저장하는 **프라이빗 레지스트리**입니다. Jenkins에서 빌드한 각 마이크로서비스의 이미지를 ACR에 푸시하며, AKS는 Managed Identity를 통해 해당 이미지를 풀(Pull) 받아 배포합니다 [25] [26] . ACR를 사용함으로써 이미지 관리가 Azure AD 권한으로 일원화되고, 네트워크 상에서도 Private Link 등을 통해 안전하게 클러스터와 연계할 수 있습니다.
- **데이터베이스 (온프레미스 MySQL)**: 온프레미스 시뮬레이션으로 구성한 **Azure VM** (예: Windows Server 2022 또는 Ubuntu 22.04)에 MySQL 서버를 설치해 사용합니다 [27] [28] . Leaderboard 등 일부 서비스의 영구 저장소로 활용되며, AKS 클러스터에서 해당 VM의 MySQL로 **직접 쿼리**를 수행합니다. 이를 위해 **DB 연결 문자열(DB_URL)** 및 인증 정보는 Kubernetes Secret/환경변수로 배포되며, 민감 정보는 Key Vault에 안전하게 저장됩니다 (아래 **보안 구성** 참조). VM 상의 DB 포트(3306)는 NSG 규칙을 통해 AKS 노드에서만 접근 허용하여 보안을 유지합니다 [19] . **참고:** 기존 코드베이스는 PostgreSQL을 사용하도록 작성되었으나 [29] , Azure 환경에 맞춰 MySQL로 마이그레이션하여 온프레미스 DB 시나리오를 구현했습니다. (Azure Database for PostgreSQL 또는 Azure SQL로 대체 가능하며, 요구사항에 따라 DB 종류는 변경될 수 있습니다.)
- **Azure Cache for Redis** (또는 Redis on AKS): **Redis 인메모리 데이터베이스**는 Leaderboard와 Progression 서비스에서 캐시 및 순위 계산 용도로 사용됩니다 [30] [8] . Azure에서는 매니지드 Redis 서비스를 이용하거나, 개발/테스트 목적으론 AKS 내에 Redis Pod를 직접 구동할 수도 있습니다. Redis는 **순위 리스트(ZSET)** 구현과 빠른 키-값 저장에 활용되어, 대량의 랭킹 계산을 실시간으로 처리하는데 기여합니다 [7] [8] .
- **메시지 브로커**: 서비스 간 비동기 이벤트 전달을 위해 **Azure Service Bus**를 사용합니다. 본 프로젝트의 메시징 추상계층은 `BUS_KIND` 설정에 따라 인메모리/SQS/Service Bus를 선택할 수 있으므로, Azure 배포 시 **Service Bus Queue/Topic**를 생성하고 연결 정보(connection string)를 제공하면 마이크로서비스 간 이벤트큐로 활용됩니다 [31] . 예를 들어 Gateway 서비스는 수집한 이벤트를 Service Bus Queue에 넣고, Progression 서비스 등이 해당 메시지를 비동기로 처리할 수 있습니다. (만약 간단한 시나리오로 인메모리 버스를 사용할 경우 별도 Azure 리소스 없이 Kubernetes 내 메모리로 동작하지만, 확장성과 내구성을 위해 프로덕션에서는 Service Bus 사용을 권장합니다.)
- **Azure Virtual Network (VNet) 및 NSG**: AKS와 VM을 포함하는 **가상 네트워크**를 구성하여 클라우드와 온프레미스 리소스가 단일 네트워크 상에서 통신하도록 합니다 [32] . 서브넷 레벨로 **네트워크 보안 그룹(NSG)** 규칙을 적용하여 불필요한 포트는 차단하고, MySQL이 구동되는 VM의 3306 포트는 AKS 노드 서브넷에서 오는 트래픽만 허용하는 식으로 제어합니다 [19] . 또한 온프레미스 환경과 연동을 위해 VPN Gateway 또는 ExpressRoute를 통해 기업 내부망과 VNet을 연결할 수 있습니다 (본 PoC에서는 Azure VM 자체를 IDC로 가정하므로 생략).
- **Azure Key Vault**: **애플리케이션 비밀 관리**를 담당합니다. 데이터베이스 비밀번호, Service Bus 연결 키, HMAC 서명 키(`HMAC_SECRET`) 등 민감한 구성값을 Key Vault에 안전하게 저장하고, AKS에서는 **Secrets Store CSI 드라이버**를 통해 Pod 시작 시 해당 값을 마운트하거나 환경변수로 주입받습니다 [33] . 이를 통해 Git 리포지토리나 Kubernetes manifest에 비밀이 노출되지 않고도 애플리케이션이 필요한 시크릿 값을 안전히 사용하게 합니다. Azure Key Vault는 **중앙 집중형 암호화 저장소**로서 데이터 암호화 및 접근 제어/감사를 제공하여, 클라우드 네이티브 환경에서 **보안 표준 준수**를 지원합니다 [34] [35] .
- **기타**: 그 외에 Ingress Controller(예: NGINX Ingress)로 AKS 내 서비스의 HTTP 엔드포인트를 노출하고 (Gateway 서비스의 `/events` API 등을 외부 접근 가능하도록), Azure Monitor 또는 Application Insights를 옵션으로 활성화하여 Azure 플랫폼 레벨의 로그/메트릭을 수집할 수도 있습니다. 그러나 본 아키텍처에서는 주로 **오픈소스 Observability 스택**(Prometheus/Grafana 등)을 사용하므로 Azure Monitor는 보조적으로 활용합니다.

요약하면, AKS를 중심으로 **컨테이너 이미지 레지스트리(ACR)**, **데이터베이스(MySQL/Postgres)**, **캐시(Redis)**, **메시지큐(Service Bus)**, **시크릿 관리(Key Vault)**, **네트워킹(VNet, NSG)** 등의 자원을 조합하여 전체 시스템을 구성합니다 [36] [28] . 각 구성요소는 Terraform 등의 IaC로 자동 프로비저닝되어 일관성 있게 관리됩니다 (아래 **IaC 구성 전략** 참조).

4. GitOps 배포 구조 및 구성 전략

이 프로젝트는 **GitOps** 원칙에 따라 **지속적 배포(Continuous Deployment)**가 이루어지도록 설계되었습니다. 핵심 구성요소는 **ArgoCD**이며, 애플리케이션의 선언형 매니페스트를 Git 리포지토리에서 관리하고, Git의 상태 변경이 곧바로 클러스터 배포 상태에 반영되도록 합니다 [20] [21].

- **Manifest 리포지토리 구조**: 코드 저장소와 별도로 Kubernetes 배포용 매니페스트를 위한 **Git 리포지토리**를 운용합니다. 이 리포지토리는 예를 들어 다음과 같은 구조를 갖습니다:

```
manifests/
├── base/                  # 기본 K8s manifest (공통 설정)
│   ├── gateway-deploy.yaml
│   ├── fairness-deploy.yaml
│   └── ... etc
└── overlays/
    ├── dev/               # 개발 환경 오버레이
    │   ├── kustomization.yaml
    │   └── *.patch.yaml (다른 이미지 태그 등)
    └── prod/              # 운영 환경 오버레이
        ├── kustomization.yaml
        └── *.patch.yaml
```

  각 마이크로서비스별 **Deployment, Service, ConfigMap, Secret** 등의 manifest를 base 디렉토리에 정의하고, 환경별로 이미지 태그나 리소스 개수를 달리할 경우 overlays에서 **Kustomize** 패치를 적용합니다. 이러한 구조를 통해 동일한 애플리케이션 구성을 개발/스테이징/프로덕션에 재사용하되, 환경 차이만 선언적으로 관리합니다.
- **ArgoCD 구성**: AKS 클러스터에 ArgoCD를 설치하고 (쿠버네티스 `argocd` 네임스페이스에 배포), 위 Git 저장소와 연동합니다 [21]. ArgoCD의 **Application 객체**를 생성하여, 예컨대 `applications.yaml`에 각 서비스에 대한 Application 리소스를 정의합니다. 한 가지 방법은 **App of Apps 패턴**으로 최상위 Application이 전체 매니페스트 repo를 동기화하도록 하는 것이고, 단순히 각 서비스별로 Application을 만들어도 무방합니다. 본 프로젝트에서는 하나의 Application에 전체 manifests 디렉토리를 지정해 4개 서비스를 함께 관리하거나, 필요시 서비스별 Application으로 나눌 수 있습니다.
- **동작 방식**: ArgoCD는 기본적으로 Git 리포지토리의 HEAD 상태를 주기적으로 폴링하거나 webhook을 통해 변화가 생김을 감지합니다. 만약 Git의 manifest가 업데이트되면, ArgoCD는 해당 변경(commit)을 감지하여 **자동 동기화(auto-sync)**를 수행, Kubernetes 클러스터에 필요한 `kubectl apply`를 해줍니다 [21]. 이를 통해 개발자가 **git push**를 하는 행위 자체가 배포를 트리거하는 효과를 얻습니다 [21]. (물론 ArgoCD UI나 CLI를 통해 수동 Sync도 가능하며, auto-sync는 필요시 disable할 수 있습니다.)
- **Kustomize 통합**: ArgoCD는 Kustomize를 네이티브하게 지원하므로, Application 리소스에 overlays 경로를 지정하면 ArgoCD가 자동으로 Kustomize 빌드를 수행하여 해당 결과물을 배포합니다. 예를 들어 `prod` 브랜치 혹은 overlay를 바라보도록 설정하면, 운영 환경에 맞는 설정으로 배포됩니다. 이런 방식으로 **브랜치=환경** 매핑 또는 **디렉토리=환경** 매핑 전략을 사용해 GitOps를 구현합니다.
- **구성 전략**: 모든 Kubernetes 리소스(manifest)는 Git에서 **단일 진실 공급원(Single Source of Truth)**으로 관리되므로, 클러스터 내 실환경 설정과 Git간 **drift**가 발생하지 않도록 합니다 [37]. 개발자는 클러스터를 직접 조작하지 않고 Git PR/Merge를 통해서만 변경을 가하며, ArgoCD 대시보드에서 어떤 리소스가 동기화되었는지, 만약 이탈 상태라면 어떤 차이가 있는지 가시적으로 볼 수 있습니다 [38].

또한 **권한 분리** 측면에서, CI 파이프라인(Jenkins)은 Kubernetes 접근 권한 없이 Git 권한만으로 배포에 관여하고, ArgoCD는 읽기 전용 토큰으로 GitRepo를 모니터링하며 Kubernetes 배포 권한을 갖습니다. 이렇게 함으로써 CI 시스템과 클러스터 간의 직접적인 자격 증명 공유를 최소화하고 보안을 높입니다. ArgoCD 자체는 admin 계정으로 운영되

며, 필요에 따라 SSO 연동 및 RBAC을 설정해 특정 팀이 특정 네임스페이스의 Application만 관리하도록 세분화할 수 있습니다.

> **요약: Kustomize + ArgoCD 기반 GitOps**를 통해 **배포 자동화 파이프라인**을 구축하였습니다 [20]. 개발자가 애플리케이션 코드를 수정하고 이미지를 새로 만들면 (CI 파트), **manifest의 이미지 태그만 변경하여 Git에 커밋**하면 됩니다. 그러면 ArgoCD가 이를 감지하여 **자동으로 Kubernetes에 배포**하므로, 사람의 개입 없이도 일관되고 신속한 배포가 가능합니다. 이러한 GitOps 구조는 인프라 구성 관리에도 동일하게 적용되어, 예를 들어 **클러스터 설정**이나 **ConfigMap 값** 변경도 Pull Request로 코드 리뷰 후 배포할 수 있게 합니다.

5. 모니터링 및 보안 구성 방안

Observability (가시성) 강화와 **보안**은 이 아키텍처의 중요한 요소입니다. 프로덕션 수준의 안정적 운영을 위해 다음과 같은 **모니터링 스택**과 **시크릿 관리/보안 대책**을 구성했습니다:

- **Prometheus & Grafana**: 클러스터 내에 **프로메테우스(Prometheus)**를 설치하여 모든 서비스와 인프라의 메트릭을 수집합니다. 마이크로서비스들은 `/metrics` 엔드포인트를 통해 애플리케이션 메트릭(예: 요청 처리량, 응답 지연, 도메인별 지표 등)을 노출하며, Prometheus는 스크랩 타겟으로 Kubernetes `ServiceMonitor` 등을 설정해 주기적으로 수집합니다 [39]. 또한 Kubernetes 노드 메트릭(node_exporter)과 **Windows VM 메트릭(windows_exporter)**도 수집 대상으로 포함되어, **온프레미스 VM의 CPU, 메모리, 디스크 IO, 네트워크** 사용량과 **MySQL DB**의 지표까지 한눈에 모니터링 가능합니다 [40]. 시계열 메트릭 DB인 Prometheus와 시각화 도구 **Grafana**를 연동하여 대시보드를 구성했습니다. Grafana에서는 **AKS 애플리케이션 메트릭 + 온프레미스 VM 메트릭**을 통합한 대시보드를 만들어, 예컨대 **Leaderboard API의 응답 지연(p95)**과 **MySQL DB의 쿼리 성능**을 한 화면에서 상관분석할 수 있습니다 [41]. (windows_exporter의 MSSQL 콜렉터 또는 MySQL용 exporter를 사용해 **Slow Query**, **Active Connections** 등의 DB 성능지표도 수집합니다.)
- **Grafana Alloy (오텔 컬렉터)**: Grafana Labs의 **Alloy 에이전트**는 **오픈텔레메트리 수집기(OpenTelemetry Collector)**의 배포판으로, 멀티클라우드/하이브리드 환경의 데이터를 일관되게 모니터링하기 위해 권장되는 방식입니다 [42]. 본 아키텍처에서는 **Grafana Alloy 에이전트**를 AKS 클러스터의 **DaemonSet**으로 배포하여, 클러스터 내 컨테이너의 로그/메트릭/트레이스를 수집하고 동시에 온프레미스 VM의 exporter 엔드포인트까지 스크레이핑하도록 구성했습니다 [43]. Alloy는 수집한 메트릭을 필요에 따라 Prometheus 서버로 **원격 전송(remote_write)**하거나, Loki/Tempo와 같은 백엔드로 보내는 파이프라인을 갖출 수 있습니다. 이를 통해 **클라우드(AKS)**와 **온프레미스(IDC)**에 분산된 telemetry 데이터를 하나의 통합 파이프라인으로 수집/가공하여 중앙 관제할 수 있습니다 [20].
- **로그 및 트레이싱**: 컨테이너 로그는 **Grafana Loki**를 활용하여 수집/검색할 수 있도록 구성했습니다. 쿠버네티스 각 노드에 **Promtail**을 사이드카로 두거나, Alloy를 통해 로그를 수집하여 Loki에 저장합니다. 이렇게 하면 애플리케이션의 구조화된 JSON 로그를 모아서 **중앙에서 검색 및 분석**할 수 있어 장애 원인 파악에 용이합니다. 또한, 서비스 간 분산 추적(Distributed Tracing)을 위해 **Grafana Tempo**를 구축하여, OpenTelemetry 프로토콜로 수집된 트레이스를 보관합니다. go-microservices 코드는 이미 OpenTelemetry SDK와 Jaeger 익스포터를 통해 트레이싱을 구현했는데 [44], 이를 **OTLP 프로토콜**로 전환함으로써 Tempo와 연계했습니다. Grafana UI에서 애플리케이션 지도(Service Map)와 트레이스 세부 내역을 확인할 수 있어, 마이크로서비스 간 호출 관계와 지연 구간을 가시화합니다.
- **Alertmanager 경보**: **프로메테우스 Alertmanager**를 구성하여 각종 임계치 초과 상황에 자동 알람이 발생하도록 했습니다. 예를 들어 **Gateway 서비스의 5분 평균 p95 응답시간이 500ms를 넘으면 경보**를 발하거나, **온프레미스 MySQL 서버의 CPU 사용률이 일정 시간 90% 이상이면 경보**를 울리도록 규칙을 정했습니다 [45]. 경보는 우선순위(Level)에 따라 구분되며, **Slack 웹훅**이나 이메일로 담당자에게 전파됩니다. 이를 통해 문제가 발생하기 전에 징후를 파악하고 선제 조치하거나, 장애 발생시 신속하게 대응하는 체계를 마련했습니다 [46]. 나아가 Alertmanager를 활용한 **자동화 대응**도 고려하고 있습니다 (예: 디스크 용량 경고 시 자동 정리 스크립트 실행 등) [47].

- **시크릿 & 보안 관리**: 앞서 언급한 **Azure Key Vault** 통합을 통해 애플리케이션 시크릿을 안전하게 관리합니다. AKS에 **Secrets Store CSI Driver**를 설치하여 Pod 배포 시 필요한 시크릿을 Key Vault로부터 마운트하거나 환경변수로 주입하며, 애플리케이션은 평문 시크릿 대신 이 경로를 통해 비밀값을 로드합니다 [33]. 또한 컨테이너 이미지 보안을 위해 **CI 단계에서 Trivy** 등의 도구로 이미지 취약점 스캔을 실행하고, 중대한 취약점 발견 시 빌드 프로세스를 실패시켜 **취약 이미지의 Registry 푸시 차단**을 구현했습니다 [48]. 쿠버네티스 측면에서는 **Pod 보안 컨텍스트(SecurityContext)**를 적용하여 컨테이너가 root 권한으로 실행되지 않도록 하고, read-only filesystem 옵션을 사용하는 등 **최소 권한 원칙**을 따랐습니다. 네트워크 보안을 위해 AKS의 네임스페이스별 네트워크 폴리시를 적용, 불필요한 서비스 간 통신을 제한하고 외부로의 아웃바운드 트래픽도 허용된 도메인만 통신하도록 설정했습니다.

이러한 관측성 및 보안 구성은 **게임 서비스의 안정적인 운영**에 필수적인 요소입니다. 운영자는 Grafana 대시보드를 통해 **실시간으로 클라우드+온프레미스 시스템 전체를 모니터링**하고, 이상징후 발생 시 Alertmanager 알림으로 즉시 인지하며, Key Vault 기반 비밀 관리로 **보안 사고 없이** 구성 관리를 할 수 있습니다 [46]. 결과적으로 **Four Golden Signals (지연, 트래픽, 에러율, 자원)**에 대한 모니터링과 **End-to-End 트레이싱**, 그리고 **안전한 시크릿 관리**를 달성하여 프로덕션 환경에서 요구되는 **가용성**과 **신뢰성**, **보안성**을 확보했습니다.

6. CI 자동화 구성 ⚙

지속적 통합(**CI**) 단계에서는 **Jenkins**를 활용하여 코드 빌드, 테스트, 컨테이너 이미지 생성 및 Registry 푸시까지 자동화합니다. 전체 흐름은 다음과 같습니다:

1. **코드 변경 트리거**: 개발자가 Git 저장소의 애플리케이션 코드 (go-microservices 레포지토리)에 변경을 푸시하면, Jenkins가 Webhook 또는 폴링을 통해 이를 감지합니다. 브랜치별로 CI 파이프라인이 구성되어 있어, 예를 들어 `main` 브랜치 푸시 시 프로덕션 이미지를 빌드하도록 설정합니다.
2. **빌드 및 테스트 단계**: Jenkins 파이프라인은 Go 애플리케이션을 빌드하고 (예: `go build` 또는 `make build` 실행 [49]), **단위/통합 테스트**를 수행합니다 [50]. 프로젝트 내 Makefile과 테스트 스크립트를 활용하여 모든 서비스에 대한 테스트가 통과하면 다음 단계로 진행합니다. 만약 테스트 실패 또는 빌드 에러가 발생하면 파이프라인이 중단되고 개발자에게 피드백이 제공됩니다 (콘솔 로그 또는 Slack 알림).
3. **Docker 이미지 생성**: 빌드된 바이너리를 기반으로 각 서비스별 **Docker 이미지를 생성**합니다. Jenkins의 워커 노드에 Docker 환경을 설치하거나, **Kaniko**와 같은 도구를 사용해도 됩니다. 이 단계에서는 루트 디렉토리의 Dockerfile(들)을 빌드 컨텍스트로 사용하며, 멀티스테이지 Dockerfile을 통해 불필요한 파일을 제외하고 경량 이미지를 생성합니다. 또한 **이미지 태깅**은 Git 커밋 SHA 또는 빌드 넘버를 사용하여 일관된 버전으로 부여합니다 (예: `myapp/gateway:build-{BUILD_ID}` 또는 semver 태그).
4. **ACR 푸시**: 태그된 이미지를 Azure 인증을 거쳐 **Azure Container Registry(ACR)**에 푸시합니다. Jenkins에는 Azure 서비스 원장(credentials)이 등록되어 있어, 파이프라인 스크립트에서 `$ az acr login` 또는 Azure CLI를 통해 ACR에 로그인한 뒤 `docker push`를 수행합니다 [51]. 또는 Azure 전용 플러그인을 사용해도 무방합니다. 빌드된 **네 개 서비스 각각의 이미지**가 모두 ACR에 저장되면, Jenkins는 다음 작업으로 이동합니다.
5. **매니페스트 업데이트 (GitOps 연계)**: 이미지 배포를 위해, ArgoCD가 모니터링 중인 **GitOps manifest 리포지토리**에 새로운 이미지 태그를 반영합니다. 예를 들어 Kustomize overlay의 `image.tag` 필드를 갱신하거나, Helm Chart 값을 업데이트하는 식입니다. Jenkins 파이프라인은 이를 자동화하기 위해 manifest 레포지토리에 대한 push 권한을 가지고 있으며, 스크립트로 YAML 파일을 수정한 후 **Git commit & push**를 수행합니다 [52] [53]. 이렇게 하면 ArgoCD가 변경된 매니페스트를 감지하여 앞서 설명한 CD 파이프라인이 실행됩니다. (※ 만약 GitOps가 아닌 직접 ArgoCD 트리거를 원한다면 Jenkins에서 ArgoCD API/CLI를 호출해 `argocd app sync`를 할 수도 있지만, 권장되는 패턴은 매니페스트를 소스로 유지하는 것입니다.)
6. **피드백 및 알림**: CI/CD의 최종 결과를 관련자에게 알립니다. Jenkins는 빌드 성공/실패 여부를 Slack 채널이나 이메일로 통보하고, ArgoCD 상태(동기화 여부, 실패시 원인 등)는 ArgoCD 대시보드를 통해 모니터링합니다. 또한 Grafana Loki에 CI 로그를 남기거나, Jira 등과 연계하여 배포 노트를 자동 생성하는 등 추가 자동화도 고려할 수 있습니다.

Jenkins 파이프라인은 **선언형 Jenkinsfile**로 관리되며, 단계별 타임아웃, 병렬 테스트 실행, 오류 핸들링 등이 명시되어 있습니다 [54] [55]. 예를 들어 이미지 스캔(Trivy)은 빌드 후 단계에 조건부로 포함하고, 취약점 발견시 실패를 리턴하도록 해 **보안 검증**을 CI에 녹여냈습니다 [48].

이러한 CI 설정을 통해 **개발에서 배포까지의 사이클타임**을 크게 단축할 수 있습니다. 개발자의 커밋이 자동으로 빌드되고 테스트되어, 몇 분 안에 새로운 컨테이너 이미지가 레지스트리에 올라가며, 이어서 GitOps 배포까지 연계되므로 **완전한 CI/CD 파이프라인**이 완성됩니다. 특히 Jenkins 기반 CI는 유연성이 높아, 추후 Azure DevOps Pipeline이나 GitHub Actions로 대체해도 개념적으로 동일한 단계를 구현할 수 있습니다. 핵심은 **빌드 아티팩트(Docker 이미지)의 자동생성 및 등록**, 그리고 **배포 매니페스트의 자동 업데이트**이며, 이를 통해 수작업 없이도 일관된 배포 프로세스를 보장합니다.

7. IaC 구성 전략 및 권장 파일 구조 📦

Infrastructure as Code (IaC)를 통해 인프라 자원 provisioning을 자동화하고, 코드로서 버전관리합니다. 클라우드 리소스는 Terraform 또는 Azure Bicep 템플릿으로 선언되며, Git 리포지토리에서 다른 구성과 함께 추적됩니다 [56]. 권장 IaC 구성 전략은 다음과 같습니다:

- **Terraform 모듈화**: Terraform을 사용하는 경우, 프로젝트 루트에 `infra/` 디렉토리를 만들고, 주요 자원별로 모듈을 작성합니다 (또는 공식 모듈 활용). 예를 들어 `modules/aks`, `modules/vnet`, `modules/acr`, `modules/vm`, `modules/monitoring` 등의 하위 디렉토리를 만들고 각 모듈에서 해당 리소스들을 정의합니다. 상위에는 `env/prod/main.tf` 등 환경별로 모듈을 인스턴스화하는 구성파일을 둡니다:

```
infra/
├── modules/
│   ├── aks/
│   │    └── main.tf (azurerm_kubernetes_cluster 리소스 정의)
│   ├── acr/
│   ├── vnet/
│   └── vm/
├── env/
│   └── prod/
│        ├── main.tf (모듈 호출 및 변수 설정)
│        └── variables.tf
└── terraform.tfstate (원격 상태 백엔드)
```

  이렇게 모듈로 나누면, 예를 들어 **AKS 클러스터** 생성 모듈에는 노드풀 구성, RBAC, Managed Identity 등의 옵션을 넣고, **VM 모듈**에는 Windows/Linux VM 생성과 초기 스크립트(클라우드 이니션으로 MySQL 설치) 등을 포함시킵니다. 최종 `prod/main.tf`에서는 각 모듈을 호출하여 인프라 전체를 구성합니다. **변수**를 사용해 환경별로 크기나 갯수 등을 조정할 수 있으며, 동일한 구성을 `dev`, `staging` 등에 재사용할 수 있습니다. IaC 변경은 Pull Request로 코드 리뷰를 거친 후 `terraform plan/apply`를 CI 도구에서 실행하는 GitOps 흐름으로 관리합니다.
- **Azure Bicep 대안**: Azure 네이티브 IaC로 Bicep을 선택할 수도 있습니다. 이 경우 폴더 구조는 Terraform과 유사하게 `bicep/` 아래 서비스별 Bicep 템플릿 파일(.bicep)들을 두고, 필요한 모듈은 Bicep Module로 분리하는 방식입니다. 예를 들면 `aks.bicep`, `acr.bicep`, `vm_mysql.bicep`, `monitoring.bicep` 등을 작성하고, 최상위에 `main.bicep`이 이것들을 param으로 호출하여 배포합니다. Bicep은 Azure Resource Manager와 통합되어 배포하므로 `az deployment` CLI 명령으로 배포 스크립트를 작성합니다. Bicep의 장점은 Azure에 최적화된 스키마와 검사 기능이며, Terraform의 장점은 멀티 클라우드와 풍부한 모듈 생태계이므로, 팀 역량과 프로젝트 요구에 따라 선택합니다.

- **Git 관리 및 환경 격리**: IaC 코드 역시 Git으로 형상 관리되며, 개발/프로덕션 환경의 정의를 분리합니다. 예를 들어 `prod` 브랜치에는 실제 프로덕션 인프라 설정이 있고, `dev` 브랜치에는 개발용 경량 설정이 있도록 운영할 수 있습니다. Terraform 원격 상태(tfstate)는 Azure Storage나 Terraform Cloud를 사용해 안전하게 저장하고 락킹을 적용합니다.
- **자동화 적용**: IaC 템플릿 변경이 발생하면 GitHub Actions 또는 Jenkins를 통해 `terraform plan`을 실행하여 변경사항을 검토하고, 승인이 나면 `terraform apply`를 자동 수행하도록 CICD 파이프라인을 구축합니다. 이를 통해 인프라 변경도 어플리케이션 코드처럼 **코드리뷰 + 배포** 프로세스로 관리되어, 수동 실수나 구성 Drift를 방지할 수 있습니다 [56]. 예를 들어 새로운 마이크로서비스를 추가하려면 Terraform 모듈에 Deployment를 추가하고 ArgoCD manifests에 Application을 추가하는 커밋을 하면, 해당 리소스들이 자동 생성/반영됩니다.

권장 파일 구조는 위와 같으며, 한 폴더에 모든 것을 넣기보다 **책임별로 디렉토리 구분**하는 것이 좋습니다. 또한 중요한 것은 **일관성 유지**입니다. IaC를 적용하면 불과 10~20분만에 동일한 인프라 스택을 새로운 리소스 그룹이나 구독에 복제할 수 있고, 재해 발생 시 신속히 복원할 수 있습니다 [56]. 결국 IaC 전략을 통해 인프라 구성을 **문서화 + 자동화**하여, 개발-스테이징-프로덕션 환경이 동일하게 구축되고 관리되는 DevOps 기반을 마련합니다.

8. 기존 인프라 계획 대비 변경점 요약

초기(또는 타 클라우드) 인프라 설계와 비교해, **Azure/AKS 중심으로 변경된 사항**은 다음과 같이 정리할 수 있습니다:

- **쿠버네티스 클러스터**: AWS의 EKS에서 **Azure AKS**로 변경되었습니다. 관리형 Kubernetes라는 점은 동일하지만, IAM 연동이 AWS IAM에서 Azure AD로 바뀌고, 노드그룹 관리 방식 등 세부 차이가 있습니다 (예: EKS는 Managed Node Group/Fargate, AKS는 Node Pool/Virtual Nodes). 결과적으로 클러스터 운영은 Azure Portal 및 `az aks` CLI로 이루어지며, Azure AD로 권한 제어를 합니다.
- **컨테이너 이미지 레지스트리**: AWS ECR에서 **Azure Container Registry(ACR)**로 교체되었습니다. CI 파이프라인에서 자격증명만 Azure 쪽으로 바뀌었을 뿐 사용 방식은 유사합니다. AKS가 ACR 이미지를 Pull할 수 있도록 ACR Pull 권한을 AKS Managed Identity에 부여했습니다 [26].
- **스토리지**: AWS S3로 가정했던 오브젝트 스토리지가 **Azure Blob Storage**로 대체되었습니다. (현재 프로젝트에서 직접적으로 오브젝트 스토리지를 사용하진 않지만, 예를 들어 랭킹 보관 파일이나 이미지 업로드 기능이 있었다면 S3 -> Blob으로 변경되는 셈입니다.) Azure Blob은 S3와 비슷한 REST API를 제공하며, 인증 체계만 SAS키 또는 Azure AD 기반으로 달라집니다.
- **메시징 큐**: **AWS SQS**를 사용하도록 했던 부분은 **Azure Service Bus Queue/Topic**으로 변경되었습니다. 코드 레벨에서는 `BUS_KIND=servicebus`와 Azure SDK를 사용하여 구현하였고, 설정 값으로 connection string을 주입하여 동작합니다. Service Bus는 FIFO Queue 및 Pub/Sub Topic을 지원하므로, SQS+SNS 조합을 단일 서비스로 처리하는 효과가 있습니다.
- **데이터베이스**: 원래 AWS RDS (PostgreSQL)이나 Aurora 등을 고려했다면, Azure 환경에서는 **Azure Database for PostgreSQL** 또는 **Azure SQL** 등이 대응됩니다. 그러나 본 시나리오에서는 **온프레미스 VM의 MySQL**로 변경하여 IDC 연동을 보여주었습니다. 이는 PostgreSQL -> MySQL DBMS 변경과 클라우드 DB -> 온프레미스 DB 변경이 혼합된 것이므로, 데이터 스키마 마이그레이션과 연결 방식 변경이 수반되었습니다. 만약 Azure PaaS DB를 사용했다면 연결 문자열 주입 방식 외에는 애플리케이션 로직 변경은 거의 없었을 것입니다.
- **캐시**: AWS ElastiCache for Redis 대신 **Azure Cache for Redis** (PaaS)로 대체하거나, Kubernetes 내 Redis로 임시 대체했습니다. ElastiCache와 유사하게 Azure Cache도 매니지드 Redis여서, 코드 변경은 필요 없고 엔드포인트만 변경됩니다.
- **시크릿 매니저**: AWS Secrets Manager 혹은 SSM Parameter Store를 쓰던 것을 **Azure Key Vault**로 변경했습니다. Key Vault 연계 방식은 앞서 설명한 CSI driver를 사용하며, AWS 대비 이식성만 달라졌을 뿐 보안 비밀 관리 개념은 동일합니다.
- **모니터링**: AWS CloudWatch, X-Ray 기반 모니터링에서 벗어나 오픈소스 **Prometheus/Grafana/Loki/Tempo 스택**으로 전환했습니다. Azure에서는 Azure Monitor(Application Insights)도 제공되지만, 보다 클라우드 중립적이고 커스터마이즈가 쉬운 OSS 스택을 활용했습니다. 이는 모니터링 구성을 코드로 관리하고, 클

라우드 종속성을 줄이는 효과가 있습니다. (또한 Grafana Labs의 Alloy로 OTel 기반 수집을 강조한 점이 차별화입니다.)
- **스토리지 (파일)**: AWS EFS 등은 Azure에서는 Azure Files or NFS on Azure로 치환될 수 있으나, 본 프로젝트에는 해당 없음.
- **CI/CD**: 만약 AWS CodePipeline/CodeBuild를 사용했다면, Azure DevOps나 GitHub Actions, Jenkins로 전환하는 변경이 있습니다. 여기서는 **Jenkins**로 표기했으나, Azure DevOps Services (ADO)의 Pipeline을 사용해도 무방합니다. 펄어비스 사례에서는 Jenkins 등 자체 CI 도구를 선호할 수 있어 이를 반영했습니다. CD는 ArgoCD로서 AWS CodeDeploy 등을 대체합니다.
- **기타**: AWS에서 사용하던 ALB/ELB(Ingress) -> Azure Load Balancer/Ingress Controller, Route53(DNS) -> Azure DNS, CloudFormation -> Bicep/Terraform 등의 일대일 대응 변경이 있습니다. 요컨대 클라우드 제공자에 특화된 managed 서비스들을 Azure의 대응 서비스로 치환한 것입니다. 다행히 본 프로젝트의 코드베이스는 **멀티클라우드 추상화**를 염두에 두었기에 (S3나 DynamoDB 등에 종속적이지 않음) 변경 작업이 최소화되었습니다.

以上와 같이, **AWS 중심 아키텍처를 Azure로 옮기는 작업**은 대부분 서비스명과 설정의 변경으로 수렴되고, 핵심 로직은 수정 없이도 동일하게 구동될 수 있음을 확인했습니다. 이것은 설계 단계에서 클라우드 종속성을 낮추고 표준 프로토콜(예: HTTP, 오픈소스 컴포넌트)을 활용한 덕분입니다.

9. go-microservices 코드 수정이 필요한 항목 목록

마지막으로, go-microservices 코드를 Azure AKS 하이브리드 환경에 최적화하여 배포하기 위해 **수정 또는 점검이 필요한 사항**을 정리합니다:

- **Dockerfile 개선**: 모든 마이크로서비스에 대해 경량화된 Dockerfile을 사용해야 합니다. 멀티스테이지 빌드로 Go 바이너리만 추출해 Alpine 기반 이미지에 담고, 실행 사용자도 `NON_ROOT`로 설정합니다. 또한 Kubernetes 환경에서 필요한 포트 설정(EXPOSE)과 환경변수(`ENV`) 설정이 올바른지 확인합니다. 예를 들어 현재 Dockerfile이 있다면 **CI에서 build 시 빌드 캐시를 활용**하도록 하고, 이미지 내 불필요한 파일 (예: 소스코드)을 복사하지 않도록 최종 스테이지를 최소화합니다.
- **환경변수 및 구성**: Kubernetes에서 주입할 각종 환경변수 키를 코드에서 제대로 참조하도록 합니다. `config.Load()` 등의 함수가 `.env`뿐 아니라 실제 OS 환경변수를 읽도록 변경하거나, 12-Factor App 원칙에 따라 **환경변수 우선**으로 설정되게 합니다. 주요 환경변수로는 `CLOUD`, `DB_URL`, `REDIS_URL`, `BUS_KIND`, `HMAC_SECRET` 등이 있으며 [57], 이 값들이 **명령행 인자나 하드코딩**되지 않고 환경에서 로드되도록 확인해야 합니다. 특히 Azure 배포에서는 Key Vault CSI로 비밀 값이 제공되므로, 해당 경로(예: `/mnt/secrets/...`)를 읽도록 경로를 조정하거나, Key Vault 값을 sync해서 Kubernetes Secret으로 넣을 경우 그냥 ENV로 받으면 됩니다.
- **다중 클라우드 플래그 처리**: 코드 내 `CLOUD=azure` 또는 `CLOUD=aws` 플래그에 따른 분기 로직이 정확히 구현돼야 합니다. 현재 adapters 패키지에 AWS SQS와 Azure Service Bus 구현이 모두 있으므로, `CLOUD` 값에 따라 **올바른 드라이버**를 선택하도록 확인합니다 [58] [31]. Azure 선택 시에는 Service Bus 연결 문자열 등 **필요한 환경변수값(KEY)**도 정의되어야 합니다 (예: `AZURE_SERVICEBUS_CONNECTION_STRING` 등, 코드에 해당 키 추가 필요 시 추가).
- **데이터베이스 드라이버 교체**: PostgreSQL용으로 작성된 ORM (Gorm) 구성을 MySQL에 맞게 조정합니다. 구체적으로, Gorm 초기화시 사용하는 DSN을 MySQL 형식 (`user:pass@tcp(host:3306)/dbname`)으로 바꾸고 Gorm MySQL 드라이버 모듈을 import해야 합니다. 또한 PostgreSQL에 특화된 SQL 문이나 데이터 타입이 있다면 MySQL 호환 형태로 수정해야 합니다. 예를 들어 `UUID` 컬럼 타입이나 `SERIAL` 사용 등을 MySQL에서는 `CHAR(36)`이나 `AUTO_INCREMENT`로 대체해야 할 수 있습니다. 다행히 도메인 모델이 단순하다면 변경이 크지 않을 것입니다. (만약 PostgreSQL 그대로 사용한다면 Azure Database for PostgreSQL 연결 문자열을 ENV에 넣고 동일하게 쓰면 되므로 이 항목은 해당없습니다.)
- **외부 연동 URL**: 혹시 코드에서 AWS S3 버킷 URL, AWS API 엔드포인트 등을 호출하는 부분이 있다면 Azure로 변경에 따라 해당 URL이나 SDK를 변경해야 합니다. 예를 들어 S3 presigned URL을 만들던 로직이

있었다면 Azure Blob SAS URL 생성 로직으로 대체해야 합니다. 현재 프로젝트에서는 such external calls 보다는 내부 로직이 대부분이라 특별한 AWS API 콜은 없었습니다.

- **보안 설정 강화**: HMAC 시크릿 키(`HMAC_SECRET`)가 최소 32바이트 이상인지 체크하고, 가능하다면 Key Vault에서 불러오도록 코드 구현을 추가합니다. 예를 들어 현재는 .env에서 키를 읽겠지만, Key Vault CSI로 마운트된 파일 경로로부터 읽는 옵션을 추가할 수 있습니다. 또 한가지, **TLS/SSL 적용** 여부입니다. Gateway 서비스가 외부에 노출될 경우 TLS 종료는 Ingress Controller가 담당하지만, 서비스 간 통신이나 DB 통신에도 가능하면 SSL을 쓰도록 설정합니다 (MySQL SSL 모드 설정 등).
- **경로 및 운영 환경**: 운영 환경에서는 로컬 파일시스템을 쓰지 않도록 코드에서 경로 참조를 점검합니다. 예를 들어 샘플 이벤트 JSON 파일이나 로그파일을 디스크에 쓰는 부분이 있다면, 쿠버네티스에서는 ephemeral 하거나 권한 문제 소지가 있으므로 사용을 피해야 합니다. 현재 프로젝트에선 대부분 메모리 혹은 DB를 사용하지만, `sample_events/` 경로 참조는 개발용일 뿐 운영에 사용되지 않음을 확인합니다.
- **테스트/빌드 스크립트 수정**: CI 파이프라인에서 go test가 원활히 동작하도록, 외부 의존성을 ENV로 주입하거나 Mock으로 대체해야 합니다. 특히 Service Bus나 Redis, MySQL 등을 필요로 하는 통합테스트가 있다면, Azure Pipelines 또는 Jenkins에서 별도의 테스트 스텝 전에 로컬컨테이너를 기동하거나, 해당 테스트를 스킵하도록 조정합니다.
- **프로메테우스 메트릭 경로**: 각 서비스의 `/metrics` 엔드포인트가 기본 포트에 잘 열려있는지, 또한 기본 레지스트리(default registry)를 사용하고 있다면 서비스 인스턴스별 metric prefix 등을 고려해 구분되는지 확인합니다. 필요하면 microservice명 라벨을 추가하거나, metric namespace를 서비스별로 달리 설정해주는 개선을 할 수 있습니다. 이는 운영상의 편의를 위한 코드 개선 포인트입니다.
- **클린업 및 최적화**: 끝으로, 사용하지 않는 코드나 불필요한 패키지 import를 제거하고 [59], 로그 레벨이나 설정값을 환경변수로 조정 가능하게 (예: DEBUG 모드 활성화 여부) 해두면 운영 시 유용합니다. 특히 메모리 누수나 goroutine 잠재 문제에 대비해 서비스 종료 시그널 (SIGTERM) 처리 루틴이 잘 작동하는지 재점검하여, Kubernetes에서 컨테이너 재시작/종료 시 **graceful shutdown**이 이뤄지도록 합니다 [60].

위 항목들을 적용함으로써, go-microservices 애플리케이션은 Azure AKS 환경에서 **무중단 배포**, **안전한 구성**, **효율적인 모니터링 연동**을 갖춘 상태로 구동될 것입니다. 특히 **코드 수정사항이 최소화**되도록 설계되었으므로 (주로 환경설정의 변경), 인프라의 변화에 비해 애플리케이션 로직의 안정성은 유지됩니다. 이러한 개선과 점검을 거친 후, 전체 시스템을 배포하면 펄어비스가 요구하는 **"게임 인프라 시스템 구축 및 서비스 운영"** 역량을 잘 보여줄 수 있는 데모/포트폴리오가 완성될 것입니다. [22] [36]

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [29] [30] [31] [49] [50] [57] [58] [59] [60] PROJECT_SUMMARY.md
file://file-NUL8DiHBbZWSGdpABVnnLR

[16] [17] [18] [19] [20] [21] [22] [27] [28] [32] [36] [37] [39] [40] [41] [43] [45] [46] [47] [48] [56] azure project.pdf
file://file-Mqh8M6JakGS5prmQLqncBM

[23] [24] Microservices Architecture on Azure Kubernetes Service - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/containers/aks-microservices/aks-microservices

[25] [26] [51] [52] [53] [54] [55] Automated image push to ACR and deployed to AKS using ArgoCD CLI and Terraform backend | by Liliane Konissi | Medium
https://medium.com/@lilnya79/automated-image-push-to-acr-and-deployed-to-aks-using-argocd-cli-and-terraform-backend-3f4ded327e59

[33] [34] [35] Azure Key Vault provider for Secrets Store CSI Driver in an Azure Kubernetes Service (AKS) | by Mehmet kanus | Hedgus | Medium
https://medium.com/hedgus/azure-key-vault-provider-for-secrets-store-csi-driver-in-an-azure-kubernetes-service-aks-56de3fe6c9b4

38 GitOps for Azure Kubernetes Service - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/example-scenario/gitops-aks/gitops-blueprint-aks

42 Set up Grafana Alloy for Application Observability | OpenTelemetry documentation
https://grafana.com/docs/opentelemetry/collector/grafana-alloy/

44 GitHub - meysamhadeli/shop-golang-microservices: Practical microservices based on different software architecture and technologies like Golang, CQRS, Vertical Slice Architecture, Docker, RabbitMQ, OpenTelemetry and Postgresql.
https://github.com/meysamhadeli/shop-golang-microservices