

# 유휴 마일리지 포인트를 이용한 스포츠 모의배팅 사이트

Terraform으로 자동화된 인프라 구축



GitHub

<https://github.com/NoJamBean/Revolution>

# 목차

**1. Cloud Architecture**

---

**2. Application**

---

**3. Backend Service**

---

**4. Log Service**

# 팀원 소개



**정원빈** 조장

Backend 담당

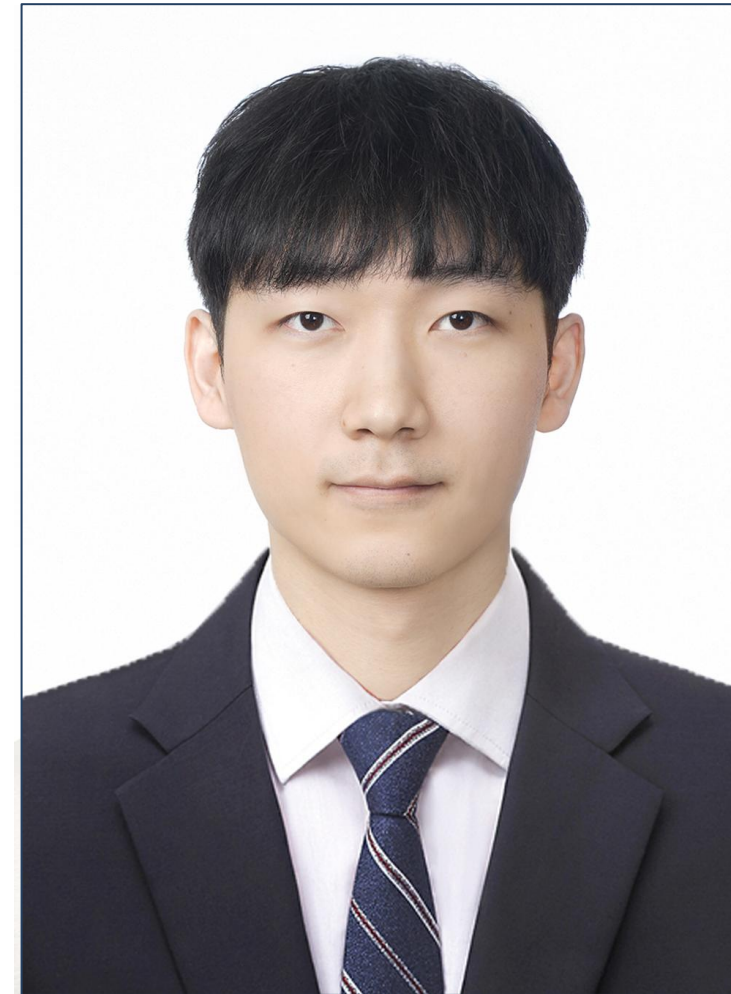
- DBMS (RDS Service) 구성
- API Server 구성



**송현섭**

Frontend 담당

- 페이지 구성
- 기능 구현



**김주관**

Cloud Architecture 담당

- AWS Cloud 환경 구성
- Auzre Cloud 환경 구성



**이정규**

Log Service 담당

- Opensearch 서비스 구성
- Dashoboard 구성

# 핵심 역량



## Terraform을 이용한 구축

멀티 클라우드 기반 웹 서비스 배포를 위해 Terraform을 도입하였으며, 코드 기반 인프라 구성 방식을 통해 다양한 클라우드 환경에서도 자동화된 배포와 확장성을 확보하였습니다.

또한 모듈화를 통해 인프라 구성을 표준화하고, 상태 관리 기능을 이용해 변경 이력을 체계적으로 관리함으로써 안정적인 서비스 운영 기반을 마련하였습니다.



## 스포츠 모의배팅 사이트

본 프로젝트는 실제 금전 거래 없이 마일리지 포인트를 통해 스포츠 경기 결과를 예측하고, 이를 기반으로 포인트를 활용하는 건전한 모의배팅 플랫폼이며 도박적 요소는 배제하고 건전한 여가 활동으로 자리매김할 수 있도록 기획되었습니다.

또한 서비스 전반은 확장성과 안정성을 고려하여 설계되었으며, 다양한 경기 종목과 통계 API 연동을 통해 지속적인 콘텐츠 확장도 가능하도록 구현하였습니다.



## Cloud Architecture

본 서비스는 멀티 클라우드 환경에 기반한고가용성·확장성 중심 아키텍처로 구축되었으며, 컨테이너 기반 마이크로서비스와 관리형 RDS, 오토스케일링 및 로드밸런싱 구조를 통해 안정적인 운영이 가능합니다.

실제 운영 환경 기반 테스트 환경을 구축해, 새로운 기능 배포 전 검증과 서비스 안정성을 확보할 수 있도록 하였으며 통합 모니터링 시스템을 통해 장애 대응이 가능하고 변화하는 트래픽에도 유연하게 대응할 수 있도록 설계되었습니다.

# Cloud Architecture

1. 전체 구성

---

2. AWS 클라우드 구성

---

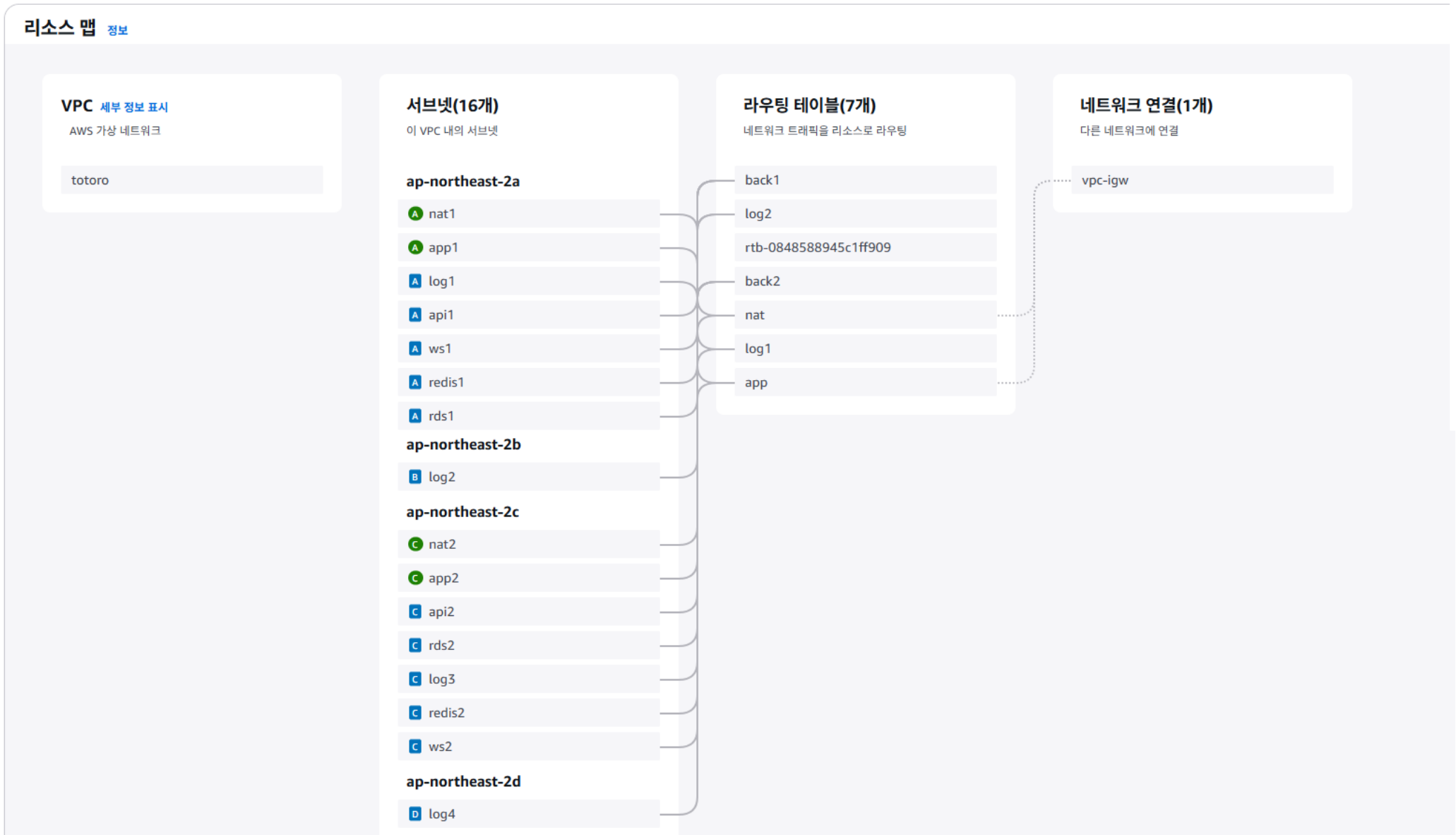
3. Azure 클라우드 구성

---

4. 결과



# AWS Cloud 구성

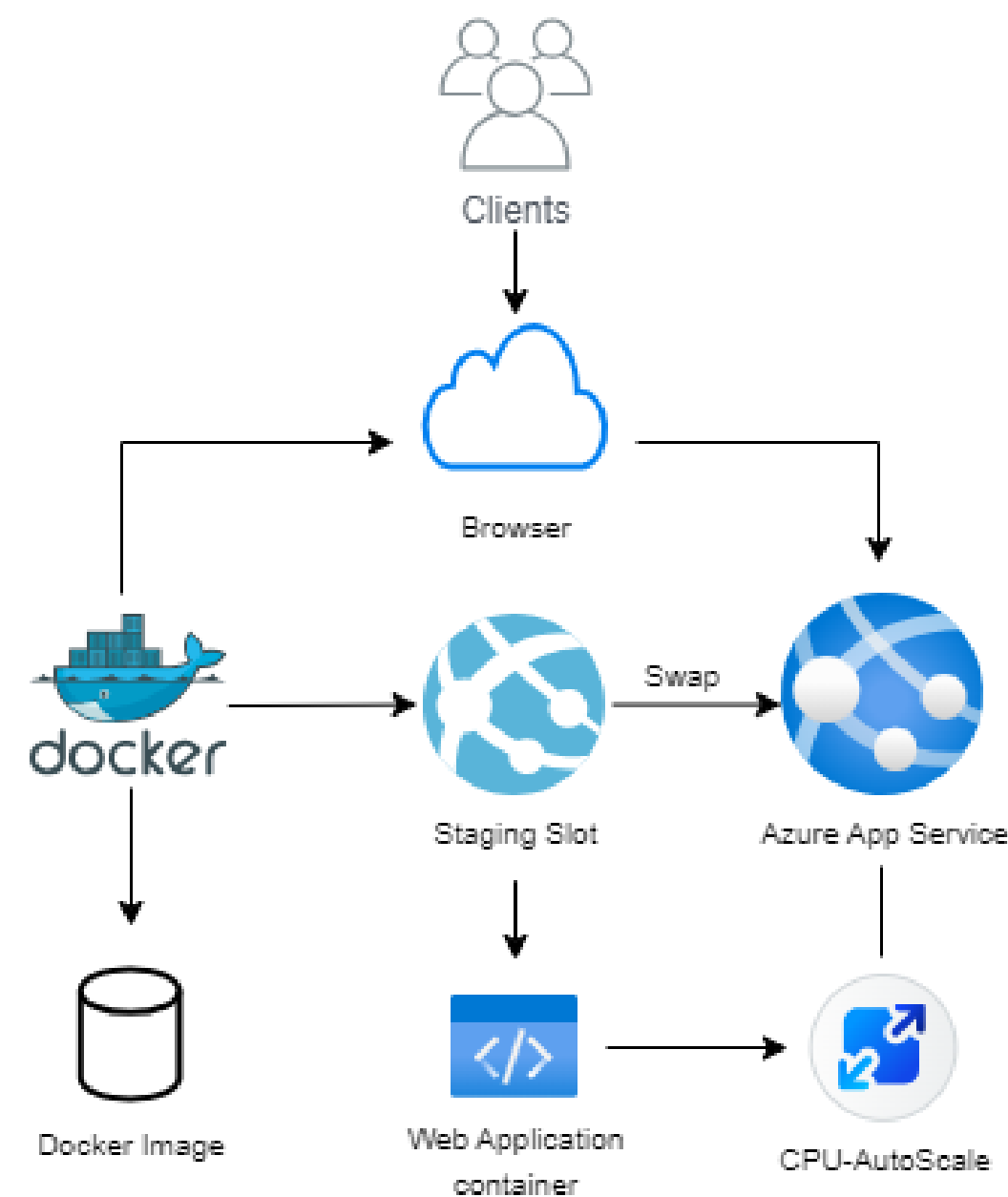














AWS 클라우드 환경은 Auto Scaling과 Application Load Balancer를 기반으로고가용성과 확장성에 중점을 둔 아키텍처로 구축하였습니다. 모든 인프라 구성은 모듈화된 Terraform 코드를 통해 자동화되어 있으며, 재사용성과 유지보수성을 고려해 설계되었습니다.

또한, AWS와 Azure 간 유기적인 통신이 가능하도록 Route 53을 활용하여 DNS 기반의 지리적 라우팅, 페일오버, 인증서 검증, 리전 간 확장을 지원하는 구조를 마련하였습니다. 이를 통해 안정적인 서비스 운영과 클라우드 간 연동이 가능한 네트워크 환경을 구현하였습니다.



# Azure Cloud 구성



Azure Resource	
	app-service-webapp
	staging(app-service-webapp/staging)
	asp-appserviceplan
	aws-cgw-1
	aws-cgw-2
	aws-connection-1
	aws-connection-2
	example-route-table
	vnet-gateway
	vnet-webapp
	vpn-gateway-pip
	vpn-gateway-pip-2

Azure Cloud 환경에서 Docker 기반 Linux App Service를 활용해 프론트엔드 웹 애플리케이션을 배포하였으며, 스테이징 슬롯을 활용한 블루-그린 방식의 자동 배포 전략을 적용하였습니다. 애플리케이션은 컨테이너 이미지로 Docker Hub에 업로드한 후 Pull 방식으로 배포되며, CPU 사용률 기반의 오토스케일 정책을 통해 탄력적인 서비스 운영이 가능하도록 구성하였습니다.



# 결과

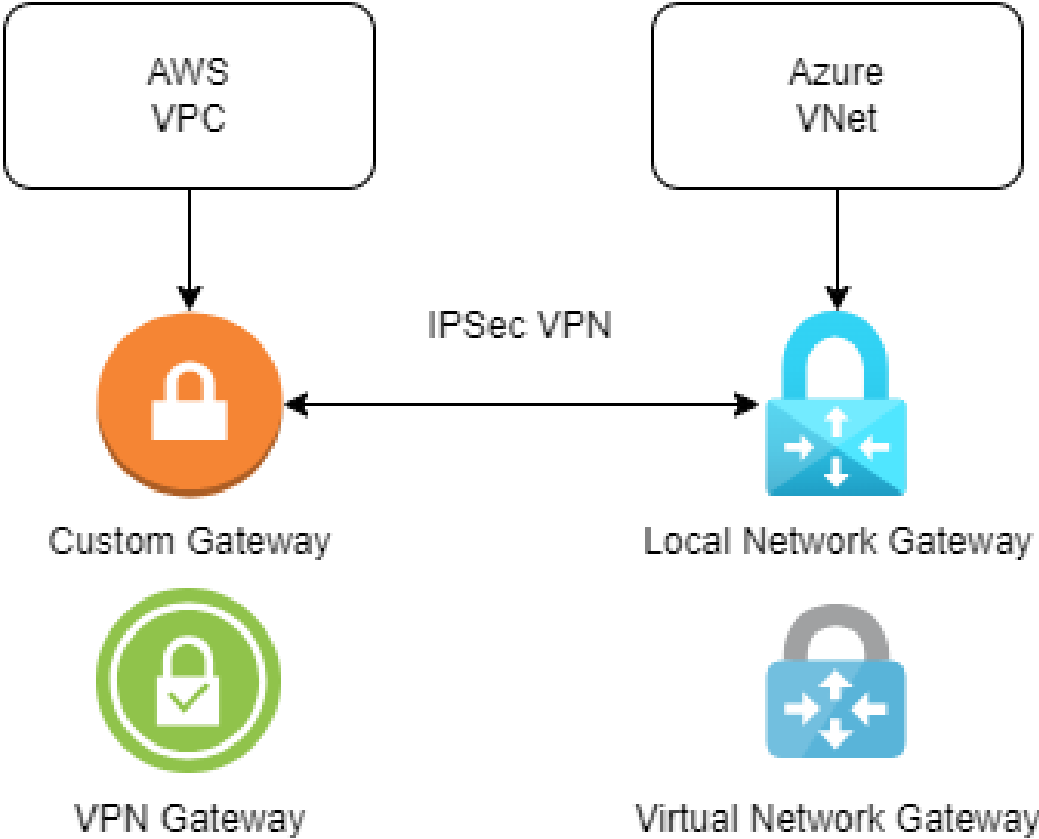
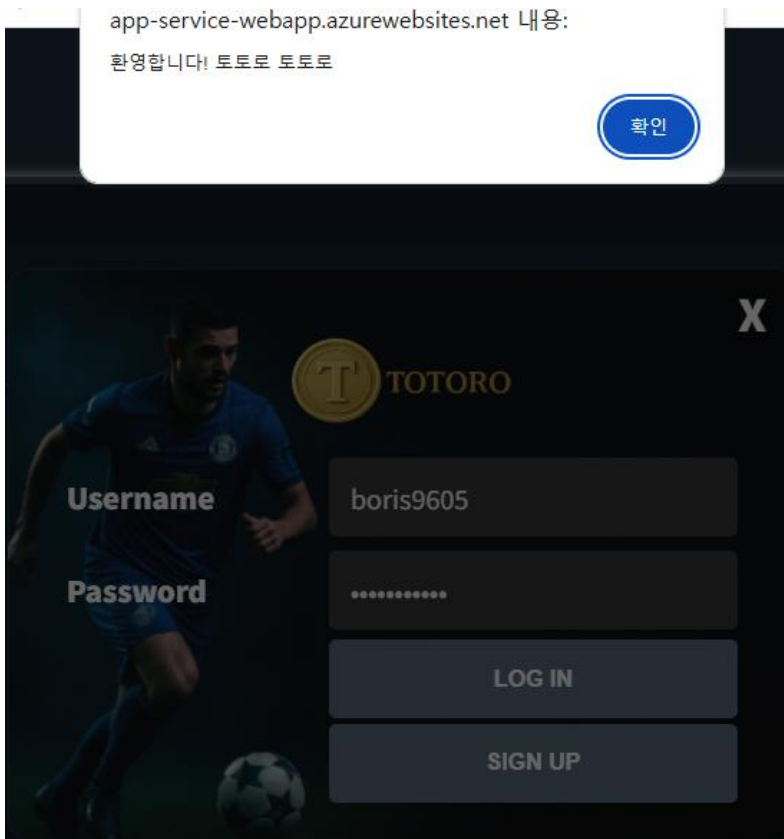


## VPN 연결 vpn-069fc4b4dc527c8f6 / vpn-aws-to-azure

세부 정보 | 터널 세부 정보 | 정적 경로 | 태그

### 터널 상태

터널 번호	외부 IP 주소	내부 IPv4 CIDR	내부 IPv6 CIDR	상태
Tunnel 1	15.164.192.51	169.254.21.0/30	-	✔ 위로
Tunnel 2	43.202.80.187	169.254.22.0/30	-	✔ 위로



멀티 클라우드 구성은 AWS와 Azure를 VPN으로 연동했으며 AWS에서는 인프라 자원을 구성하고, Azure에서는 App Service 및 관련 리소스를 배치해 웹 어플리케이션을 배포하는 방식을 이용했습니다. VPN 연결을 통해 두 클라우드 간 사설 IP 기반의 안정적인 통신 경로를 확보했으며, DNS는 Route53 Inbound Resolver를 활용해 Azure 측에서도 AWS 리소스의 내부 도메인을 조회할 수 있도록 구성하였습니다. 또한 확장성과 가용성을 고려해 양쪽 모두 Auto Scaling 및 로드밸런싱 구조를 도입했으며 모든 인프라는 Terraform을 통해 모듈화된 방식으로 코드화해 자동화 및 관리 효율성을 극대화하였습니다.

# Application

1. 전체 구성

---

2. 페이지 구성

---

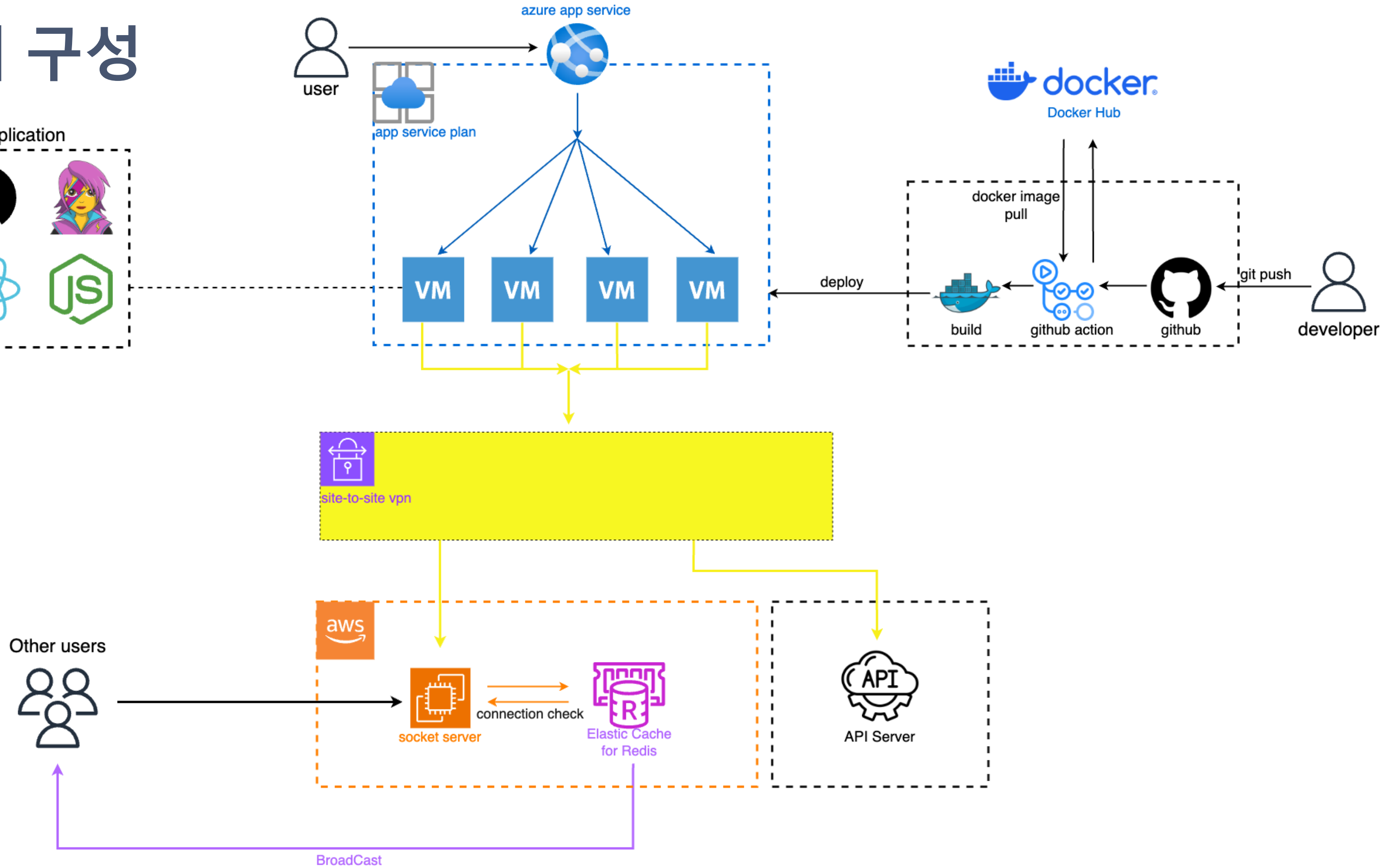
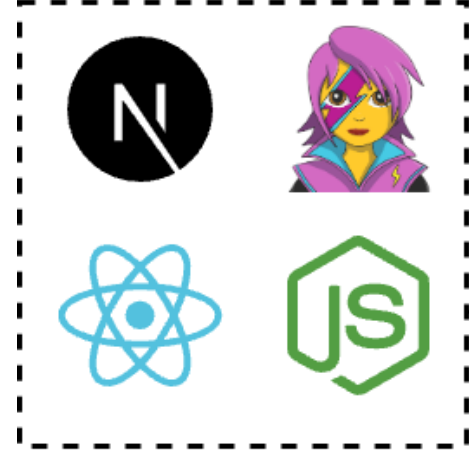
3. 기능 구현

---

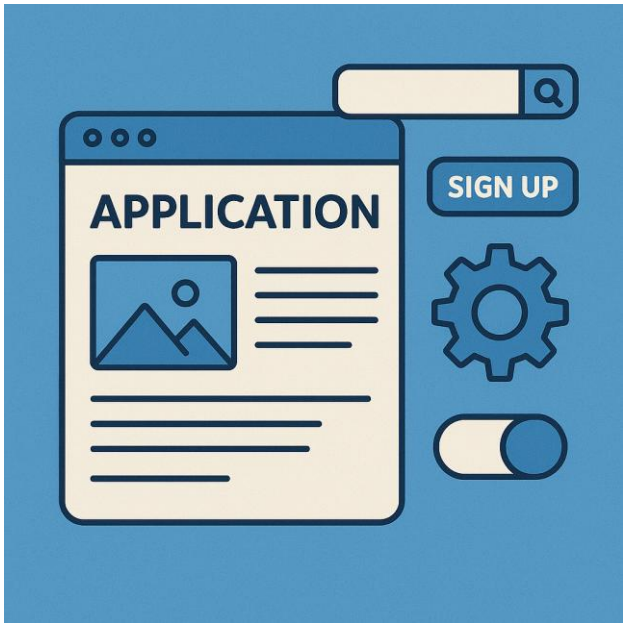
4. 결과

# 전체 구성

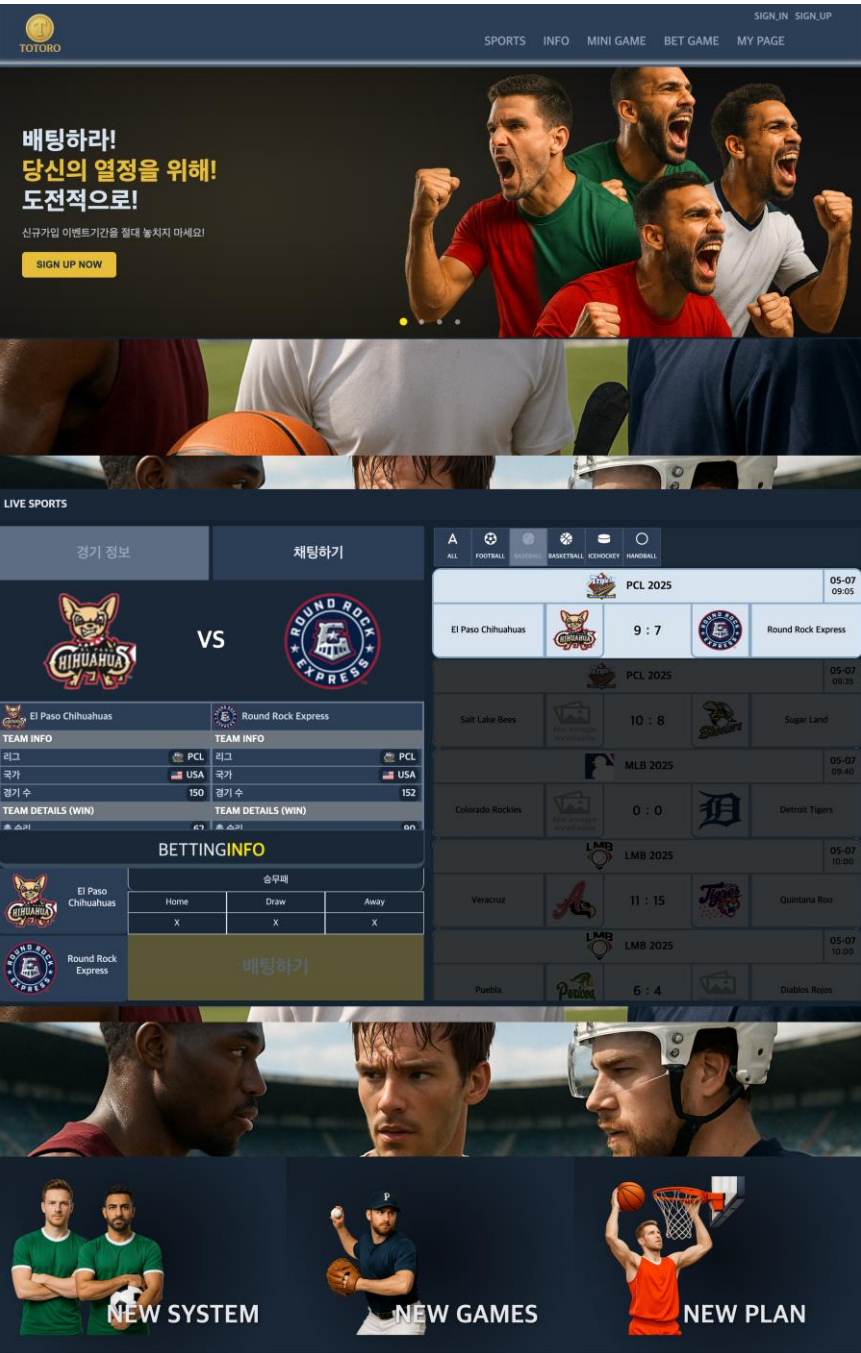
## Web Application



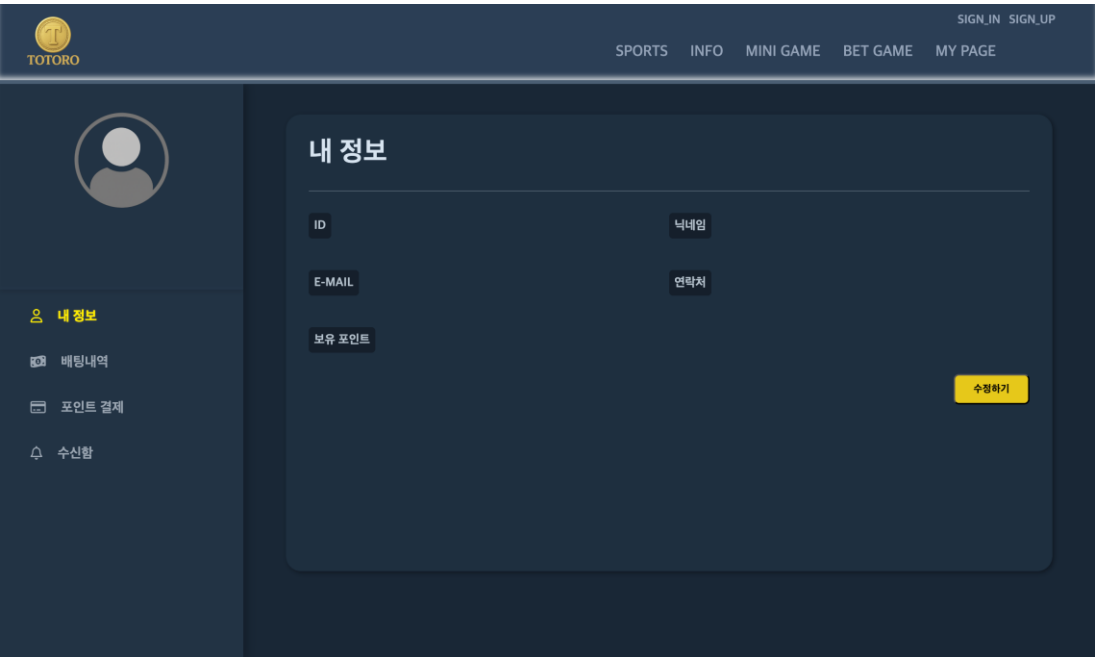
# 페이지 구성



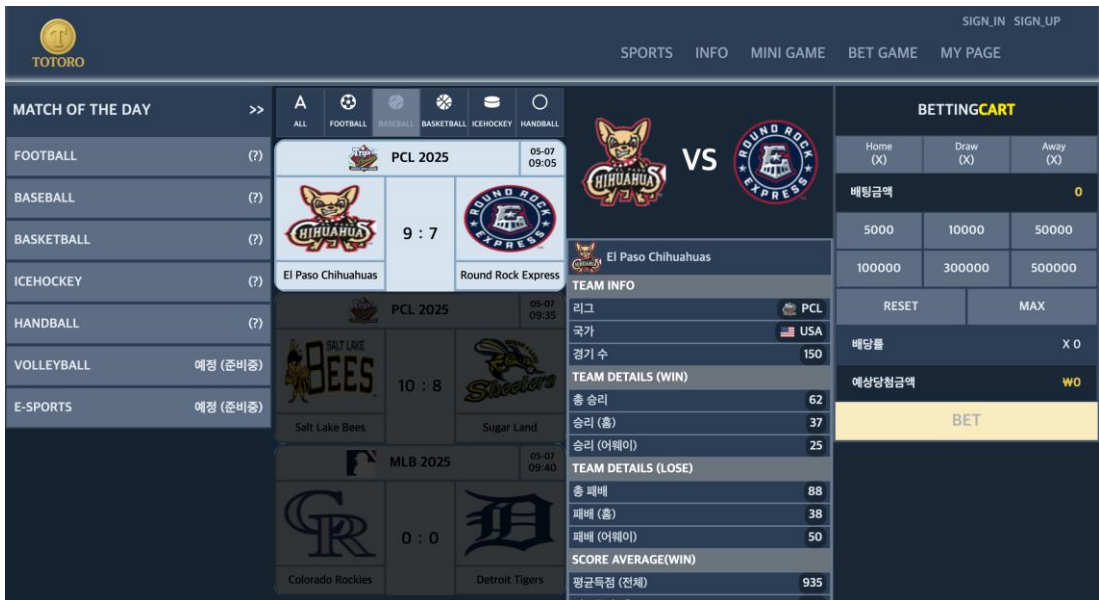
## 메인 페이지 (/main)



## 사용자 정보 페이지 (/mypage)



## 배팅 페이지 (/bet)



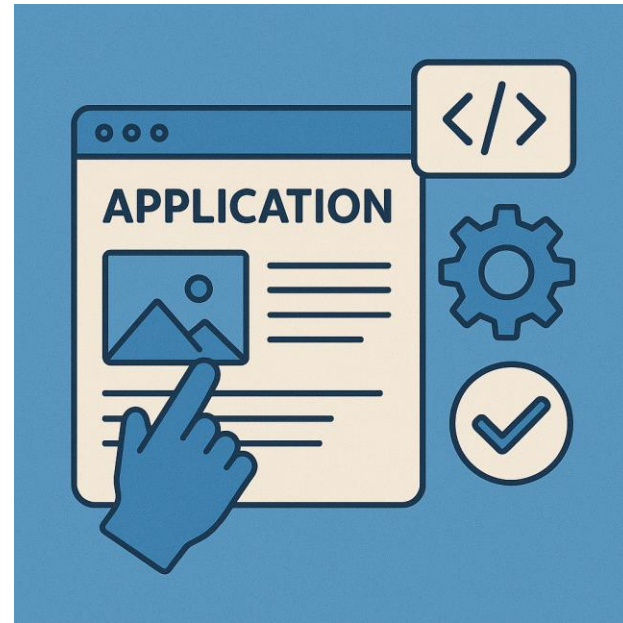
/main 페이지에서는 Sport 종목에 대한 데이터를 제공하는 open API를 활용하여 사용자가 페이지 접속 후 받아온 각 종목별 경기 리스트를 체크할 수 있도록 구성했습니다. Next.js 프레임워크의 강점을 살려, shallow routing 기능을 활용해 경기 선택 시 배팅페이지로 바로 이동되어 선택한 경기에 대한 배팅을 바로 진행 가능하도록 사용자 경험을 향상시켰습니다.

/bet 페이지에서는 종목별 오늘 경기 수, 선택된 경기의 각 팀의 역대 전적, 오늘 경기에 대한 배당률 확인이 가능합니다. 각각의 경기 데이터는 open API에서 제공하는 데이터를 통해 실시간으로 받아오며, 사용자가 매번 페이지에 접속할 때마다 최신 정보를 기반으로 UI가 동적으로 렌더링됩니다.

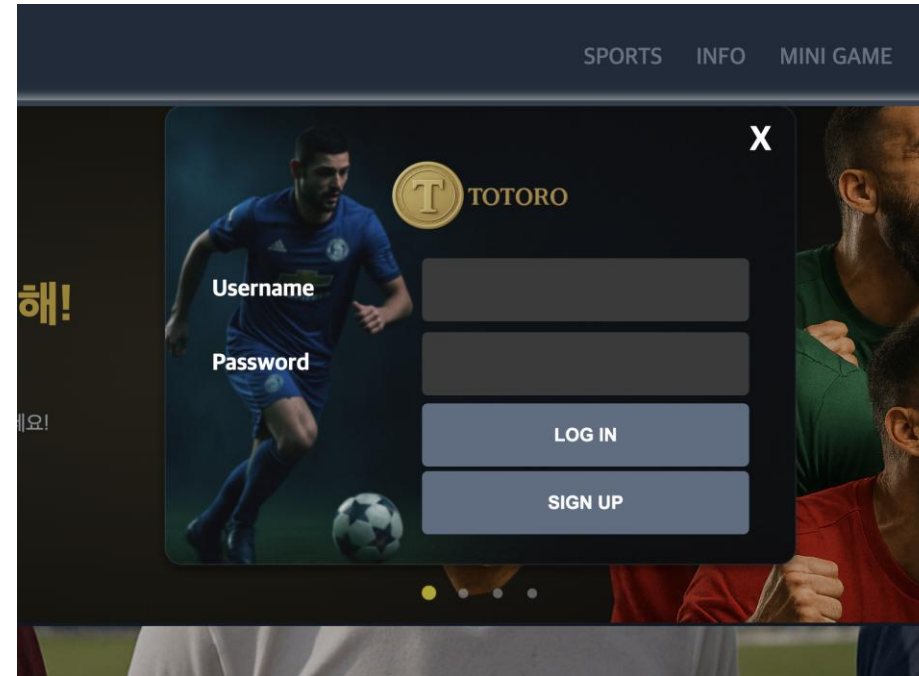
/mypage 페이지에서는 회원가입 시 입력한 사용자 정보, 배팅내역, 보유 잔액 등을 확인 가능합니다. 개인정보는 Cognito에서 받아온 토큰 값을 복호화하여 얻게 되는 userData 정보를 이용해 각 사용자들의 개인 입력정보를 화면상에 보여주는 식으로 구성되어 있습니다. 일정금액을 넣고 배팅을 한 후 배팅 내역에서 각 경기정보를 확인 가능하며, 게임의 진행 상태에 따라 before, playing, Finished 의 각 현재 경기 진행도를 확인 가능합니다.



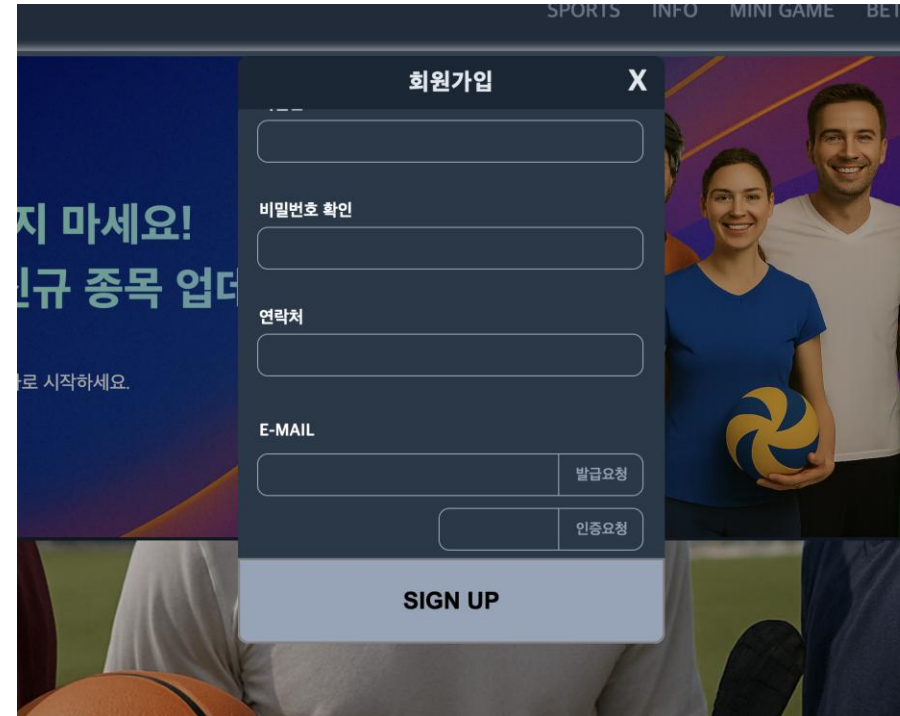
# 기능 구현



## 로그인 (Modal)



## 회원가입 (Modal)



## 채팅 (Web socket)

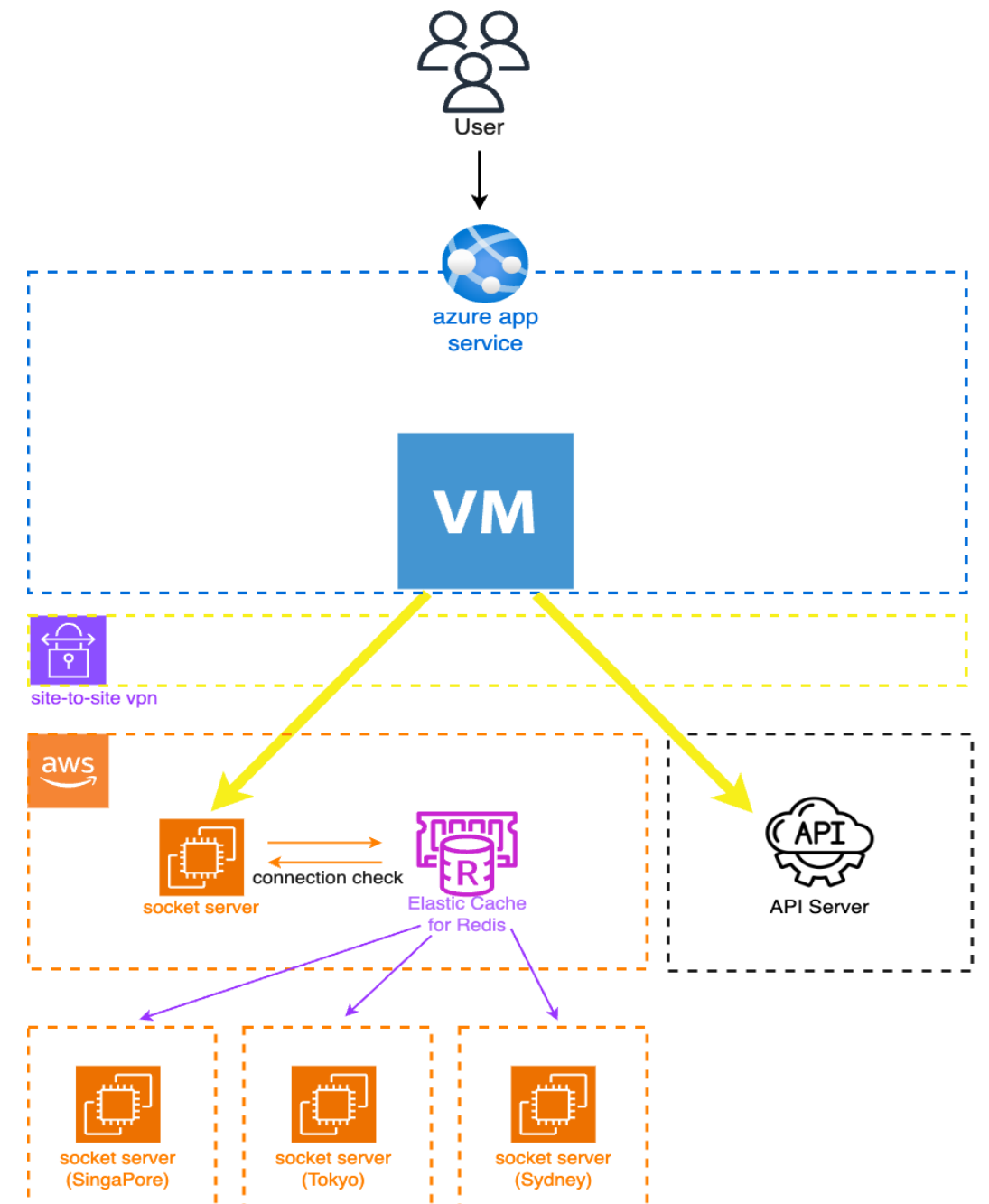
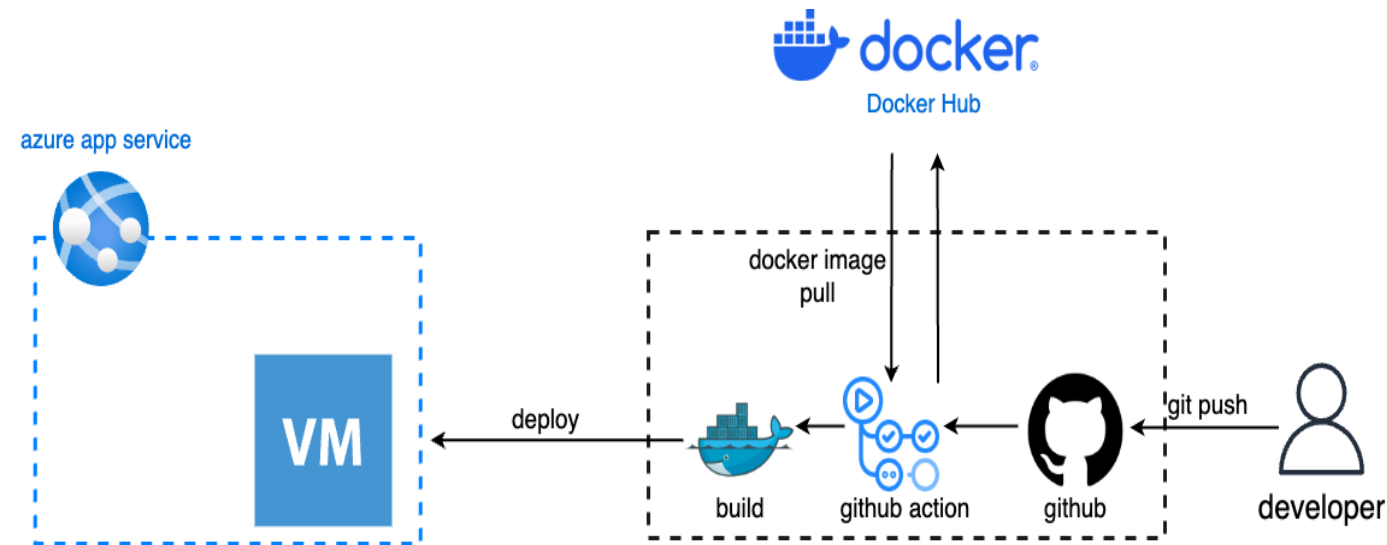


사용자 인터페이스와 함께 핵심 기능들을 안정적으로 구현하기 위해 다양한 기술 스택을 적용하였습니다. 로그인 및 회원가입 기능은 하나의 공용 모달 컴포넌트에서 상태 전환 방식으로 구현되었으며, 이를 통해 UI 내에서 별도의 페이지 이동 없이 부드럽고 일관된 사용자 경험을 제공할 수 있도록 했습니다. 사용자가 로그인에 성공하면, 연결된 Cognito service로 부터 토큰을 응답받고, 해당 토큰은 localStorage에 저장되며, 이를 기반으로 페이지 이동 시에도 로그인 상태가 유지되도록 설계되었습니다.

실시간 채팅 기능은 WebSocket과 AWS ElastiCache for Redis를 연동하여 구현되었습니다. 사용자가 특정 경기에서 채팅에 참여하면, roomId를 기준으로 WebSocket 서버에서 Redis Pub/Sub 구조를 통해 메시지를 송수신하며, 다수의 사용자가 같은 방에 접속해도 안정적으로 브로드캐스팅이 이루어지도록 구성되어 있습니다. 또한 방에 처음 입장한 사용자가 있는 경우에만 해당 방 정보를 생성하고, 모든 사용자가 퇴장하면 자동으로 방 정보를 제거하는 로직을 구현하여 서버 자원을 효율적으로 관리할 수 있도록 했습니다. 이 구조는 AWS EC2에 배포된 WebSocket 서버와 연결되어 있으며, 추후 Redis의 Broadcast를 통해 멀티 리전 환경으로의 확장을 고려했습니다.

서비스 배포는 Azure App Service를 통해 이루어졌으며, 사용자 접근은 Azure 측에서 담당하고, API 서버와 WebSocket 서버는 AWS 상에 위치한 인프라로 구성했습니다. 두 클라우드 간의 연결은 VPN 터널을 통해 이루어지며, Azure 앱에서 들어온 요청은 내부적으로 VPN을 타고 AWS 내부의 프라이빗 서브넷 내 API 서버, WebSocket 서버로 전달됩니다. 또한 GitHub Actions를 통해 Azure에 배포되는 웹 애플리케이션은 push 때마다 트리거로 인해 자동으로 빌드 및 배포가 진행되어, 끊임없이 지속적 배포가 가능하게 구현되었습니다.

# 결과



본 프로젝트는 단순한 기능 구현에 그치지 않고, 실제 서비스 운영을 고려한 구조적 설계와 클라우드 기반 확장성을 갖춘 웹 애플리케이션으로 완성되었습니다. 프론트엔드는 Azure App Service를 통해 배포되며, 사용자 접근을 담당하도록 설계되었습니다. 반면, 핵심 로직을 담당하는 WebSocket 서버 및 API 서버는 AWS 인프라 내 EC2 인스턴스에 위치해두고, 두 클라우드를 VPN으로 연결하여 보안성과 구조적 분리를 모두 확보한 하이브리드 아키텍처를 구성하였습니다. 이러한 구조는 클라우드 자원 분산, 네트워크 격리, 유지보수 편의성 측면에서 실제 운영 환경에 적합한 구성을 반영하여 설계되었습니다.

또한 채팅 기능은 AWS ElastiCache for Redis의 Pub/Sub 구조를 기반으로 하여 구현되었고, 다수의 사용자가 실시간으로 참여해도 안정적으로 메시지를 브로드캐스트할 수 있는 구조를 갖추었습니다. 특히 이 구조는 리전 간 Redis 클러스터 분리 및 글로벌 서버 확장도 고려하여 설계되었으며, 향후 멀티리전 확장이 가능하도록 미리 그 기반을 마련하는 것에 중점을 두었다고 볼 수 있습니다.

CI/CD는 GitHub Actions와 Docker 기반의 파이프라인을 통해 개발부터 배포까지 자동화되었으며, 빌드된 Docker 이미지는 Docker Hub에 푸시된 뒤 Azure App Service에서 자동으로 배포됩니다. 이를 통해 효율적인 운영과 빠른 피드백 사이클을 가능하게 했습니다. 결과적으로 기능, 구조, 인프라, 배포에 이르기까지 서비스 전반을 통합적으로 고려한 실전형 웹 플랫폼으로 구현되었습니다.

# Backend Service

1. 전체 구성

---

2. API Server 구성

---

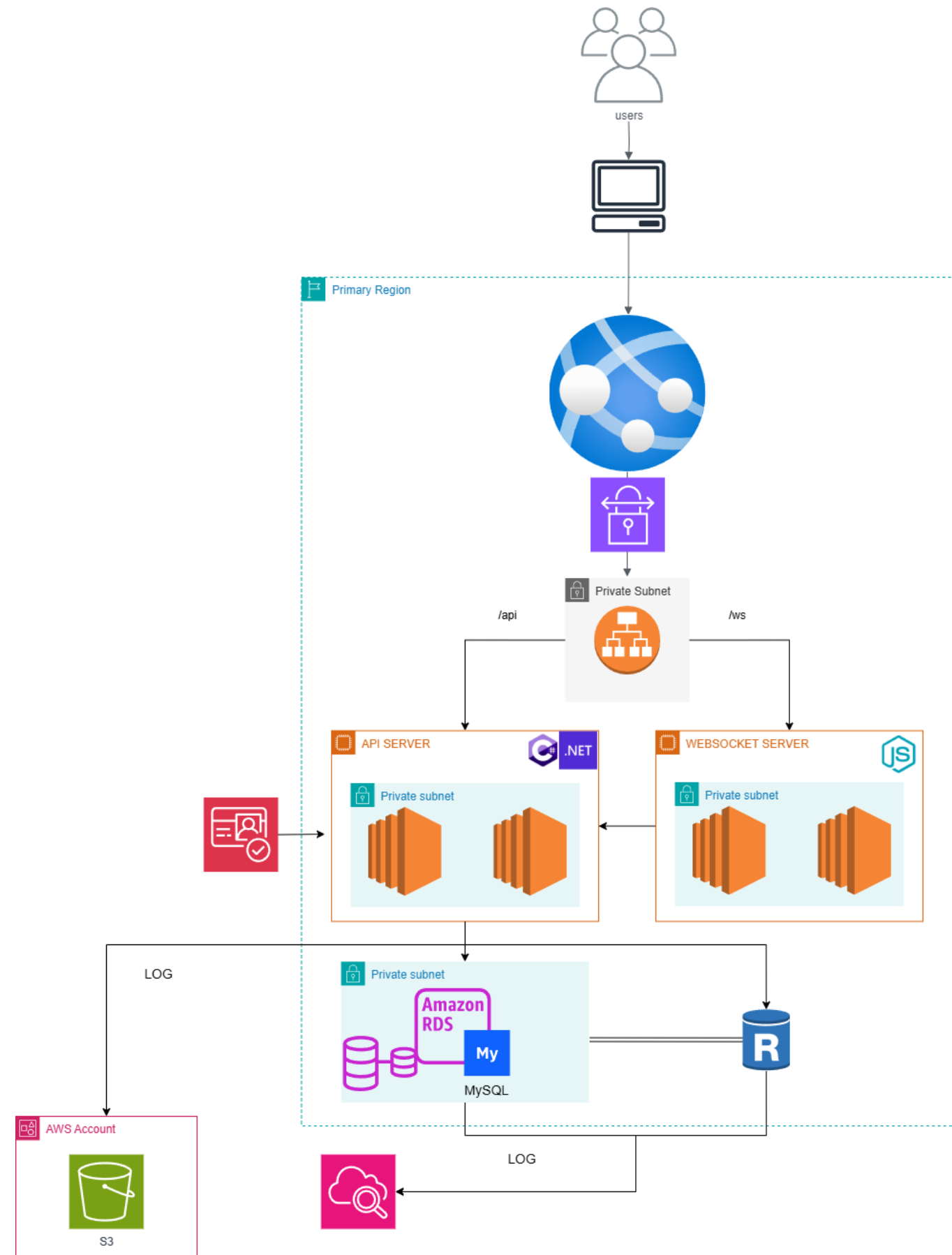
3. RDS Service 구성

---

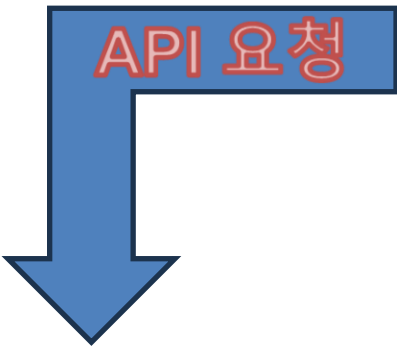
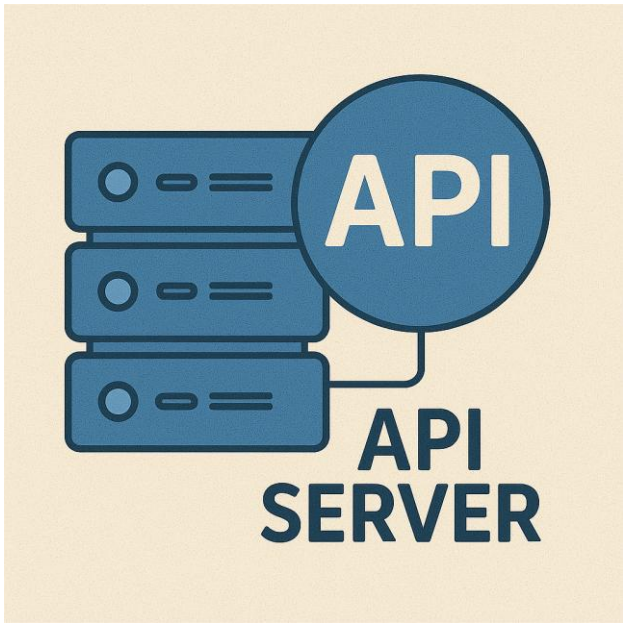
4. 결과



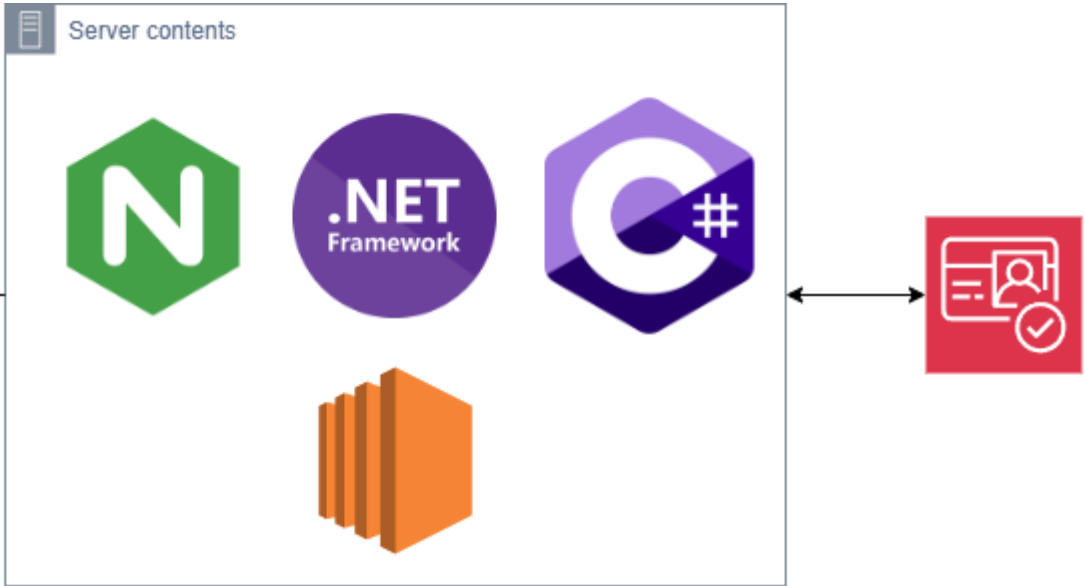
# 전체 구성



# API Server 구성



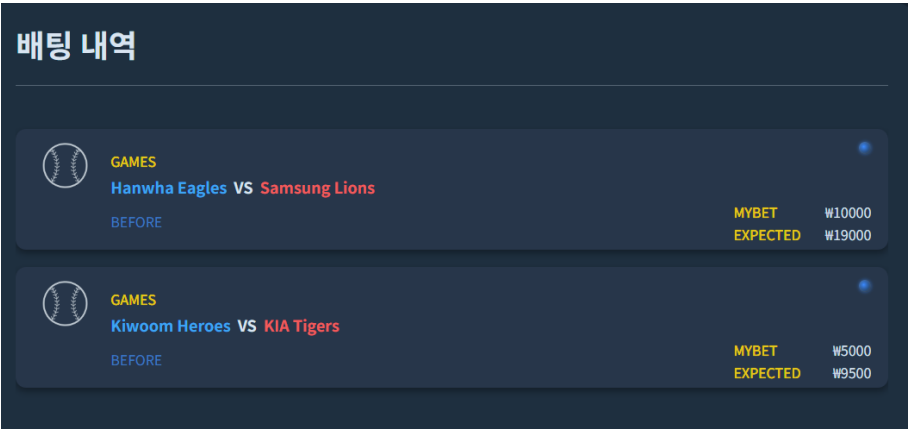
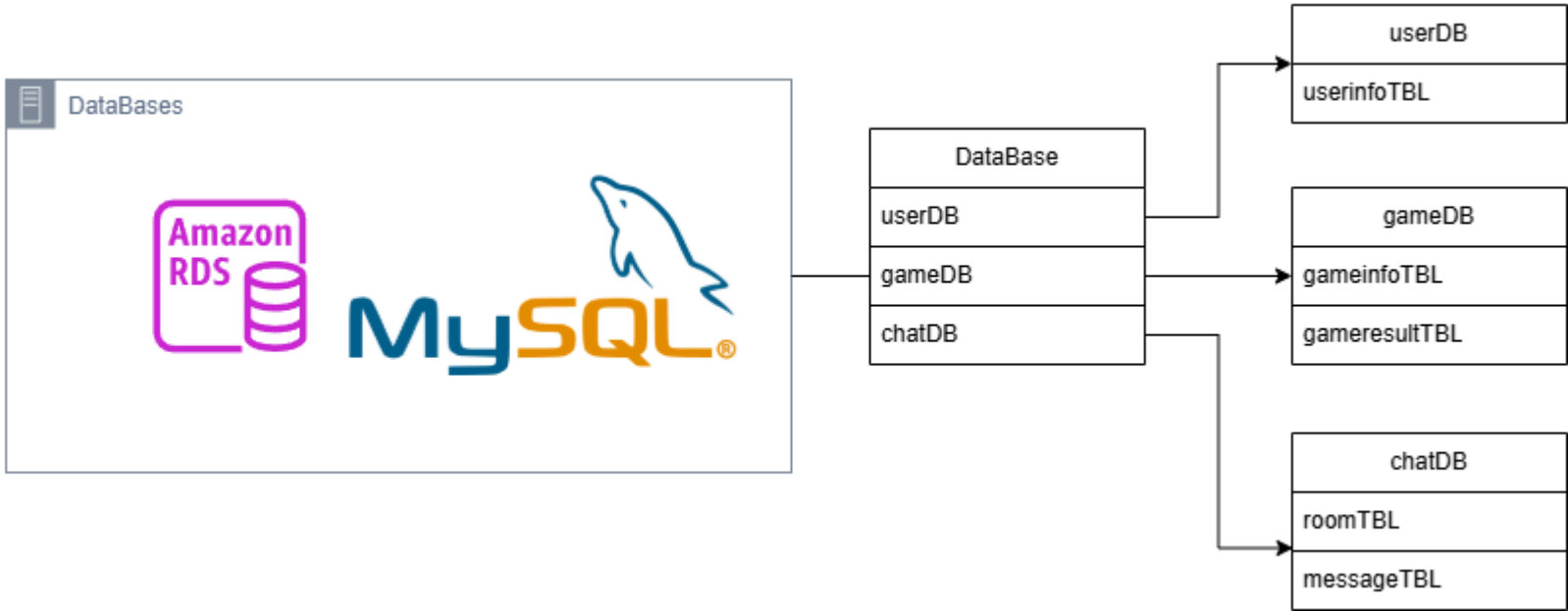
API
GET
POST



```
[ec2-user@ip-10-0-21-100 ~]$ curl -v http://alb.backend.internal/api/users/register/idcheck/aaaa
* Trying 10.0.101.17:80...
* Connected to alb.backend.internal (10.0.101.17) port 80
> GET /api/users/register/idcheck/aaaa HTTP/1.1
> Host: alb.backend.internal
> User-Agent: curl/8.3.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Wed, 07 May 2025 02:09:58 GMT
< Content-Type: application/json; charset=utf-8
< Transfer-Encoding: chunked
< Connection: keep-alive
< Server: nginx/1.26.3
<
{
  "message": "사용 가능한 아이디입니다."
}
* Connection #0 to host alb.backend.internal left intact
```

API Server는 AWS 환경에서 .NET 6 기반 API 서버를 EC2 인스턴스로 구성하고, Nginx를 리버스 프록시로 활용해 외부 요청을 효율적으로 분산 처리하도록 설계했습니다. 또한, 사용자 인증은 AWS Cognito를 연동하여 API 서버 내에서 직접 처리함으로써, 인증 보안성과 통합 관리의 효율성을 동시에 확보하였습니다. 서버리스나 컨테이너 대신 EC2 기반으로 구성하여 고정적인 트래픽 환경에서 비용 절감 효과도 볼 수 있는 구조로 구축하였습니다.

# RDS Service 구성

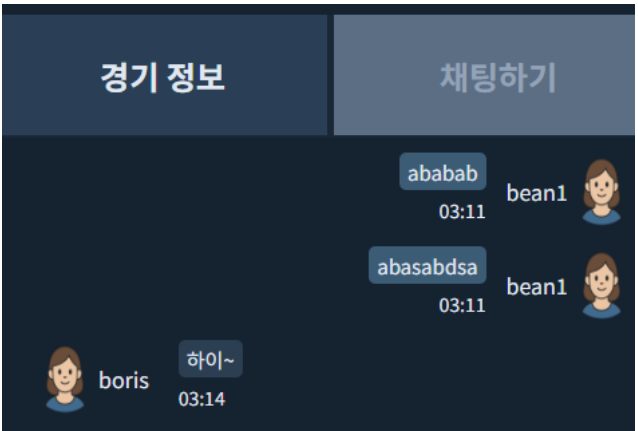


id	nickname	password	e_mail	phone_number	balance	modified_date
gnfkqh	bean1	\$2a\$11\$6dB20jBS1nNMbZ7Cs.4IqO4WqUJrlzru3AJs5QWpPy9WYdLkT3dgC	gesep48817@firain.com	111-1111-1111	0	2025-05-05 23:45:12
manner	songseop	\$2a\$11\$7f1fgZ0old8F9Q/QSdRhauXoilnoMF5TedO50uVE6UIYjGF3wiBEe	gustjq99455352@gmail.com	010-1233-4444	0	2025-05-05 20:26:38

seq	id	type	matchid	gameDate	home	away	wdl	odds	price	status	modified_date
1	gnfkqh	BASEBALL	166918	2025-05-07 18:30:00	Hanwha Eagles	Samsung Lions	HOME	1.90	10000	BEFORE	2025-05-07 16:24:27
2	gnfkqh	BASEBALL	166919	2025-05-07 18:30:00	Kiwoom Heroes	KIA Tigers		1.90	5000	BEFORE	2025-05-07 17:11:15

roomid	modified_date
162517	2025-05-07 12:09:16

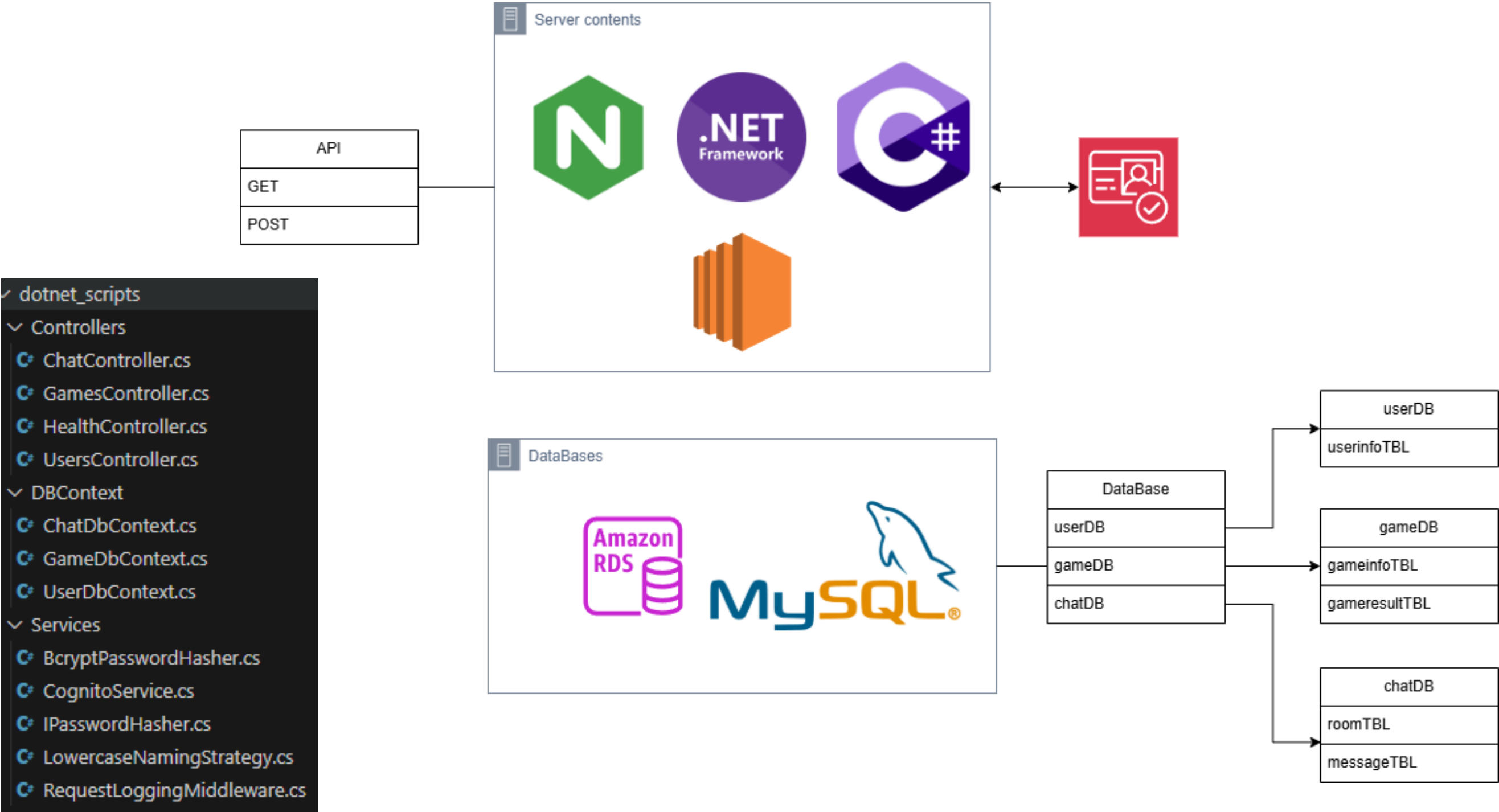
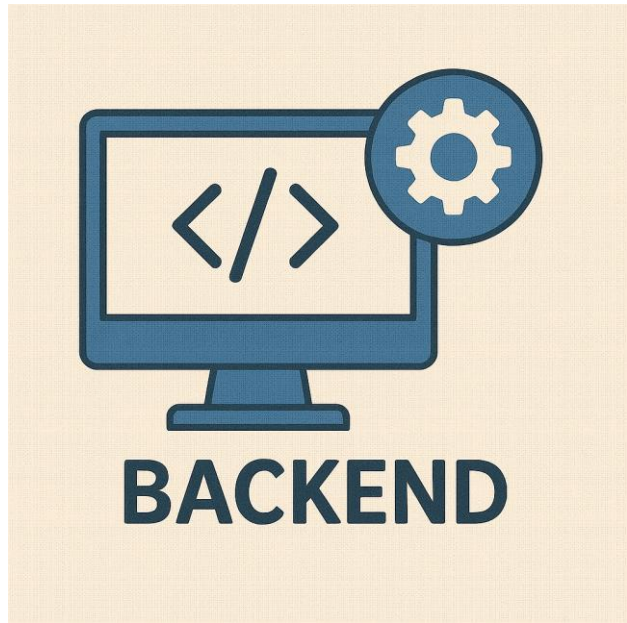
roomid	id	content	time
162517	bean1	ababab	2025-05-07 12:11:18.810016
162517	bean1	abasabdsa	2025-05-07 12:11:20.827572
162517	boris	하이~	2025-05-07 12:14:30.194081



RDS Service는 AWS RDS 인스턴스를 기반으로 구축하여 안정성과 확장성을 확보하였으며, 트래픽 분산과 빠른 응답 속도를 위해 읽기 전용 복제본 Read Replica 도 함께 구성하였습니다. 이를 통해 읽기와 쓰기 요청을 분리 처리함으로써 전체 시스템 성능을 향상시켰습니다.

데이터는 서비스의 기능과 역할에 따라 용도별로 별도의 데이터베이스 DB 로 체계적으로 분리하였으며, 각 데이터베이스 내에서도 기능 단위로 테이블을 나누어 구성함으로써 유지보수의 효율성과 데이터 접근의 명확성을 높였습니다. 이러한 설계는 향후 서비스 확장 및 데이터 분석에도 유리한 구조를 제공합니다.

# 결과



Backend 서비스는 읽기 전용 복제본(Read Replica) 구성과 기능별 데이터베이스 분리를 통해 성능과 확장성을 확보하고, 데이터 접근성과 유지보수 효율을 극대화했습니다. API 서버는 .NET 6 기반으로 AWS EC2에 구축하고 Nginx 리버스 프록시를 통해 안정적인 트래픽 분산 처리를 구현했으며, 사용자 인증은 AWS Cognito를 연동해 보안성과 통합 관리를 강화했습니다. 또한, API는 기능과 목적에 따라 파일을 체계적으로 분리하여 유지보수와 협업 효율을 높인 구조로 설계했습니다.

# Log Service

1. 전체 구성

---

2. Opensearch Service

---

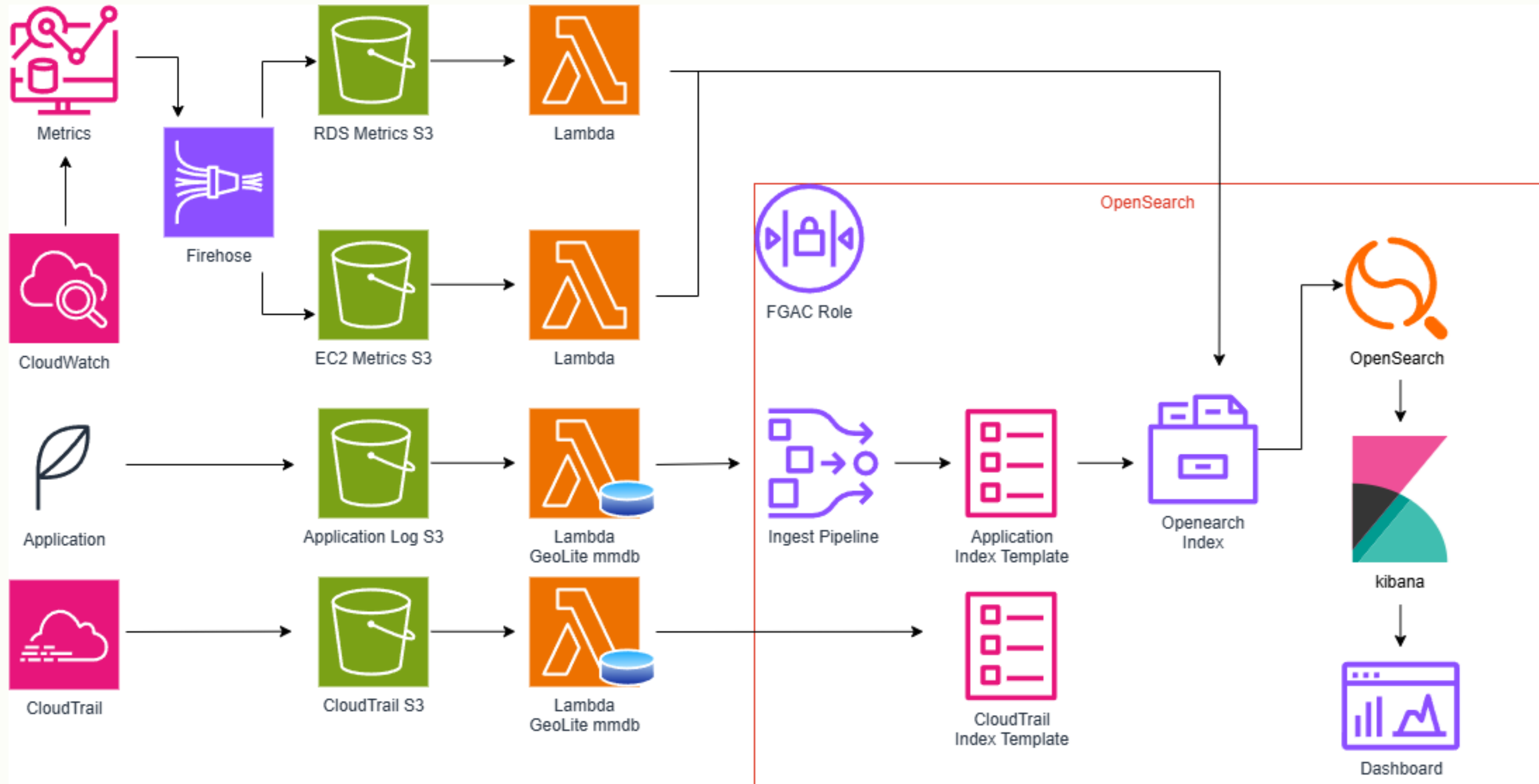
3. Geopoint

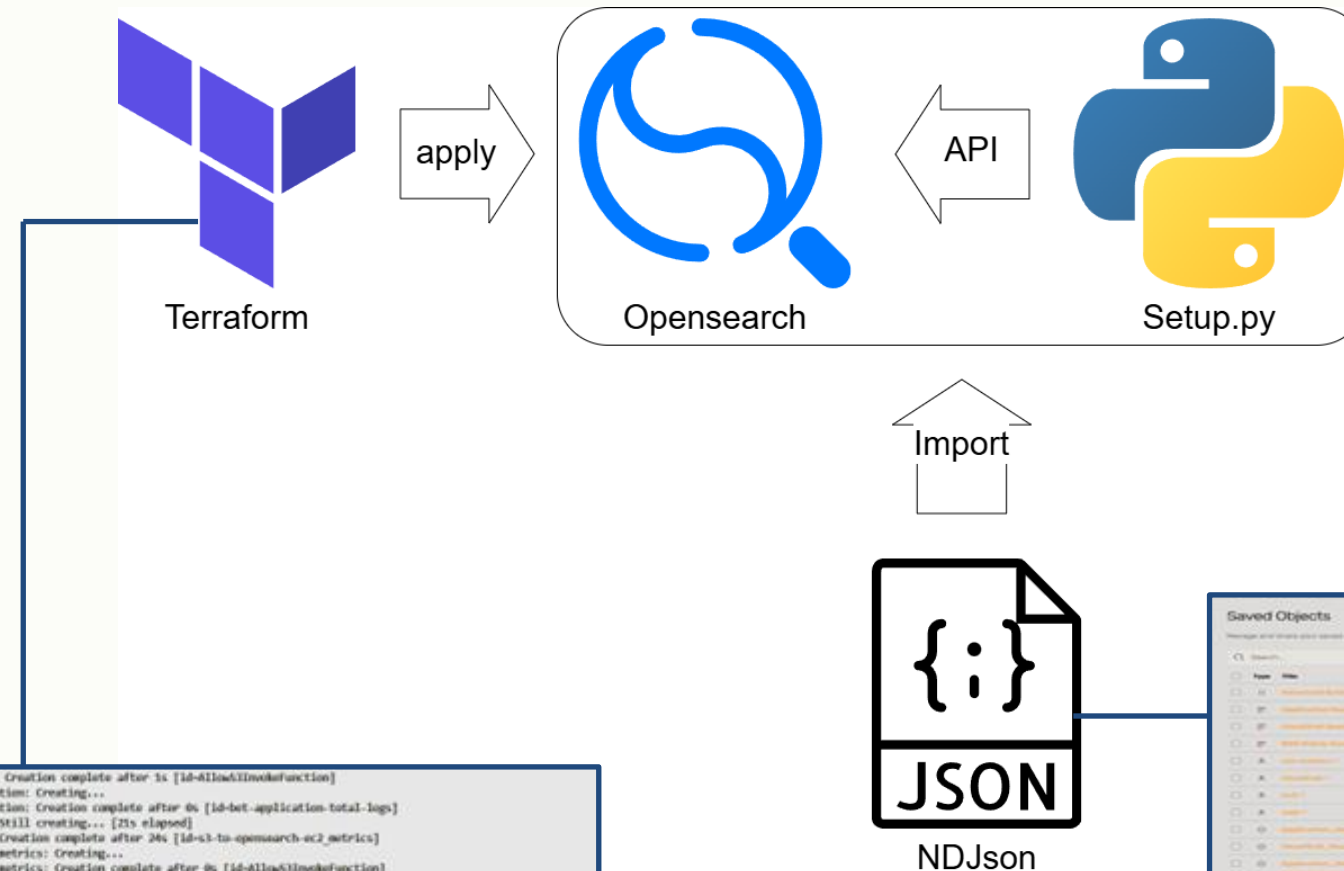
---

4. 결과



# 전체 구성





```

root@DESKTOP-K037QDH:~/git/Revolution/Log/opensearch# python3 setup.py

성공적으로 가져온 Terraform 출력 값:
- opensearch_domain endpoint: search-integration-log-timang-pmq42otk4e4kzasqldlnbpkgey.ap-northeast-2.es.amazonaws.com
- lambda_iam_role_arn: arn:aws:iam::248189921892:role/lambda-s3-opensearch-role
- firehose_iam_role_arn: arn:aws:iam::248189921892:role/firehose-s3-delivery-role

스크립트의 다른 부분에서 값 사용 가능:
OpenSearch Endpoint: search-integration-log-timang-pmq42otk4e4kzasqldlnbpkgey.ap-northeast-2.es.amazonaws.com
Lambda Role ARN: arn:aws:iam::248189921892:role/lambda-s3-opensearch-role
Firehose Role ARN: arn:aws:iam::248189921892:role/firehose-s3-delivery-role
인제스트 파이프라인 'parse_querystring_final' 생성을 시도합니다...
성공! 응답 코드: 200

인덱스 템플릿 'web-*' 생성을 시도합니다...
대상 URL: https://search-integration-log-timang-pmq42otk4e4kzasqldlnbpkgey.ap-northeast-2.es.amazonaws.com/_index_template/web_logs_template
성공! 응답 코드: 200

원수 호출 성공.

인덱스 템플릿 'cloudtrail_logs_template' 생성을 시도합니다...
대상 URL: https://search-integration-log-timang-pmq42otk4e4kzasqldlnbpkgey.ap-northeast-2.es.amazonaws.com/_index_template/cloudtrail_logs_template
성공! 응답 코드: 200

'cloudtrail_logs_template' 템플릿 생성/업데이트 성공.
응답 내용: {
  "acknowledged": true
}

사용할 값 확인:
OpenSearch Endpoint: search-integration-log-timang-pmq42otk4e4kzasqldlnbpkgey.ap-northeast-2.es.amazonaws.com
Lambda Role ARN: arn:aws:iam::248189921892:role/lambda-s3-opensearch-role
Firehose Role ARN: arn:aws:iam::248189921892:role/firehose-s3-delivery-role
Target OpenSearch Role: all_access
Admin User: admin

'all_access' 역할에 IAM 역할 'arn:aws:iam::248189921892:role/lambda-s3-opensearch-role' 매핑을 시도합니다...
대상 URL: https://search-integration-log-timang-pmq42otk4e4kzasqldlnbpkgey.ap-northeast-2.es.amazonaws.com/_plugins/_security/api/rolesmapping/all_access
성공! 응답 코드: 200
응답 내용:
{
  "status": "OK",
  "message": "'all_access' updated."
}

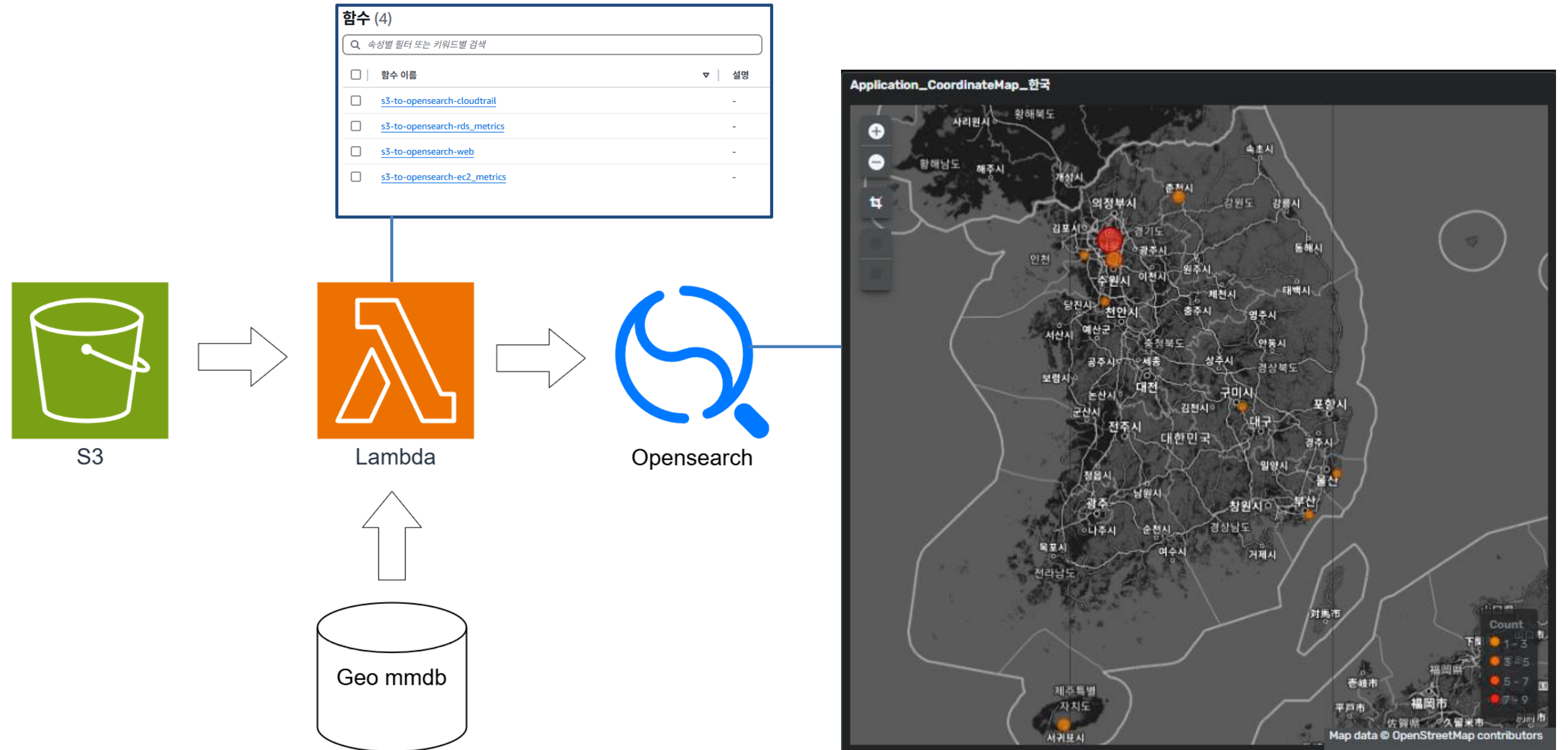
```

[illegible]

파편화된 운영 데이터 문제를 해결하기 위해 웹 애플리케이션 접근 로그, AWS CloudTrail 로그(API 활동 감사), EC2 및 RDS 인스턴스의 주요 성능 메트릭을 통합합니다. 이러한 다양한 데이터 소스를 단일 OpenSearch 클러스터로 통합함으로써, 통합된 검색, 분석, 시각화를 제공합니다. 운영 가시성을 크게 향상시키고, 문제 해결 속도를 높일 수 있고 AWS 인프라의 상태 및 사용량에 대한 Insight를 제공합니다.

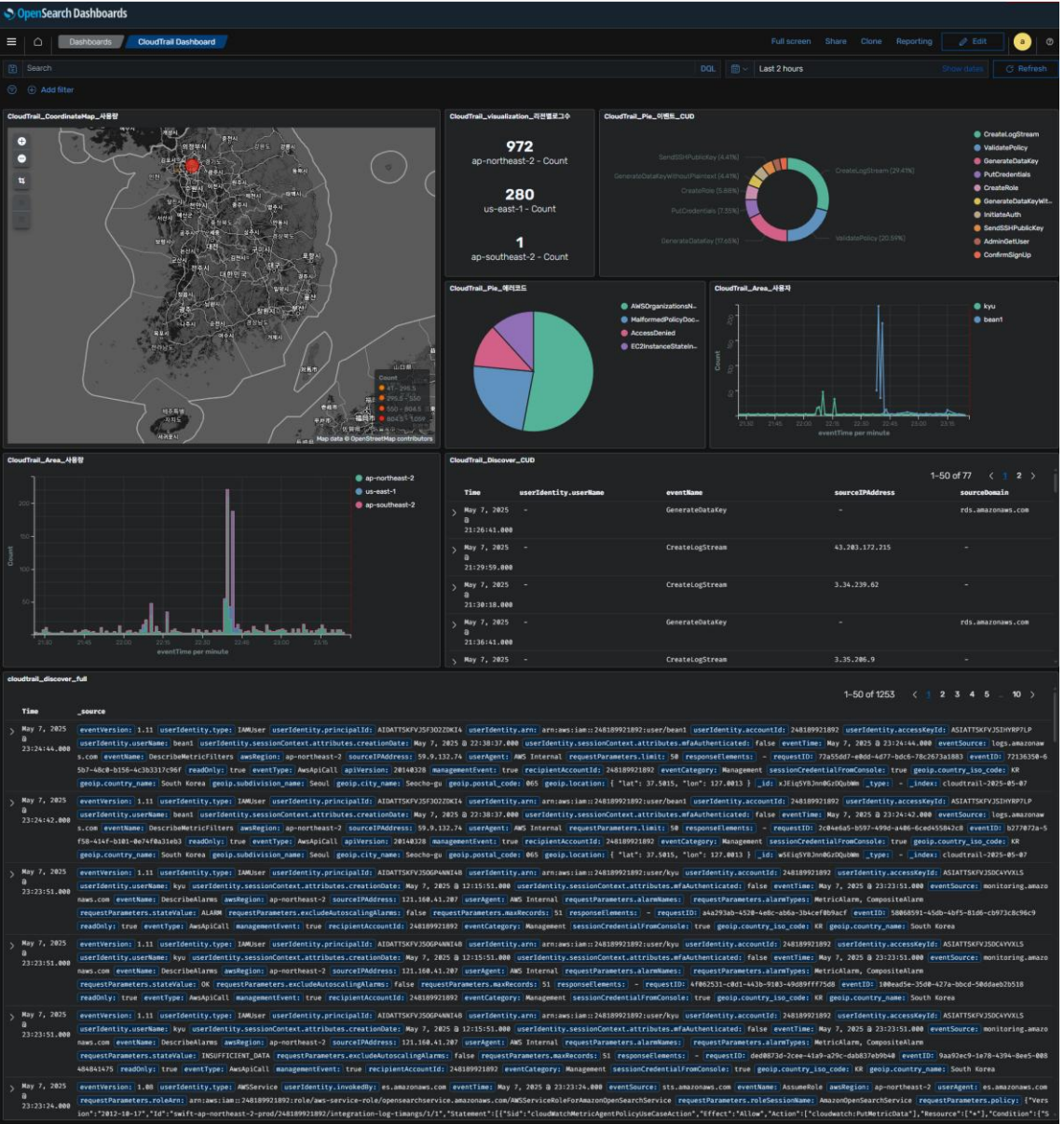
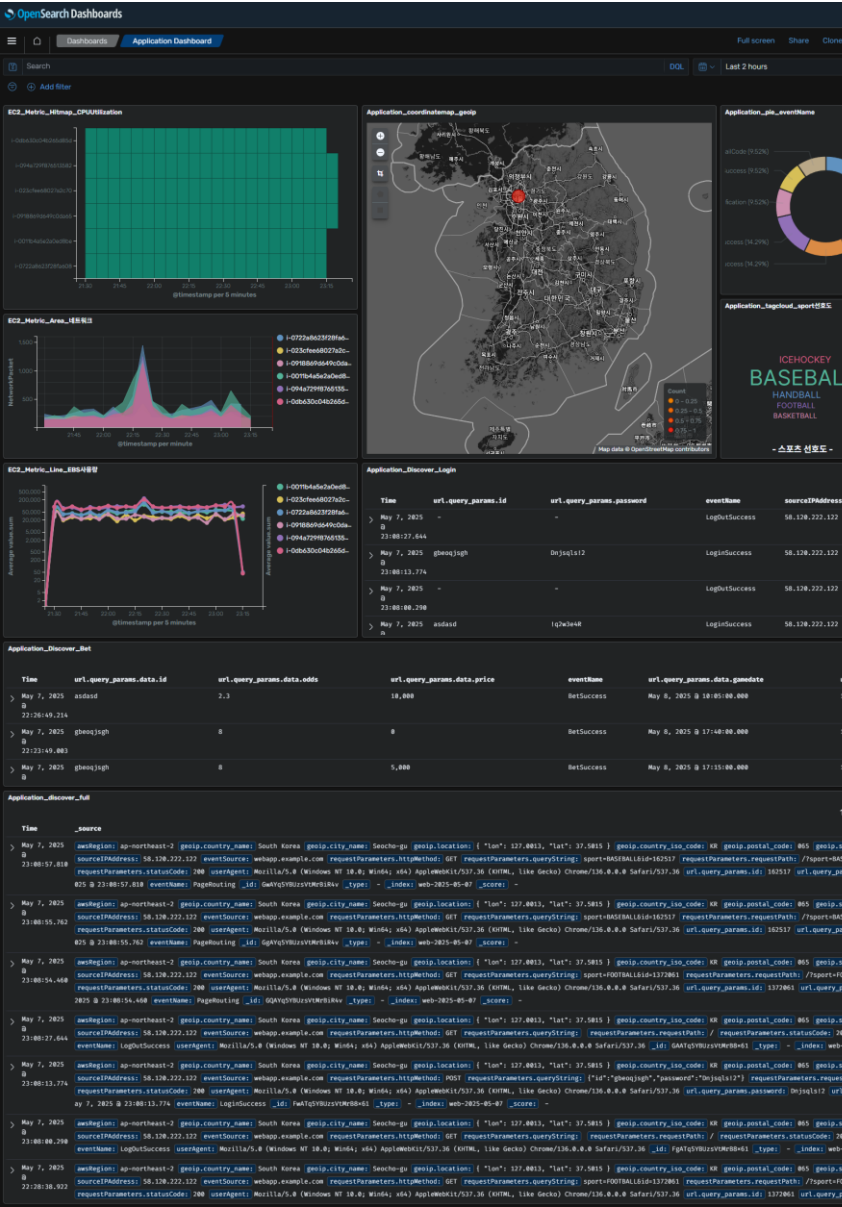


# GeoPoint



본 프로젝트는 웹 애플리케이션 사용자의 지리적 위치 정보를 활용하여 보안 및 운영 가시성을 향상시키기 위해 AWS OpenSearch 내에서 Geopoint 데이터 분석을 구현했습니다. 수집된 웹 애플리케이션 접근 로그에 포함된 IP 주소를 기반으로 지리적 위치 정보를 연계하여 OpenSearch에 geo\_point 필드로 저장하고 인덱싱합니다. 이를 통해 특정 지역에서의 접근 패턴 분석, 비정상적인 위치에서의 로그인 시도 감지, 지역별 트래픽 분포 시각화 등 다양한 위치 기반 분석이 가능해집니다. OpenSearch Dashboards의 지도 시각화 기능을 활용하여 직관적으로 사용자 접근 발생 지역을 파악하고 잠재적 보안 위협 또는 지역별 성능 문제를 신속하게 식별하는 데 기여합니다.





대규모 로그 및 메트릭 데이터에서 유의미한 통찰력을 추출하고 시스템 상태를 효율적으로 모니터링하기 위해, 본 프로젝트는 AWS OpenSearch Dashboards를 활용한 맞춤형 대시보드 구축에 중점을 두었습니다. 웹 애플리케이션 접근 로그, CloudTrail 이벤트, EC2/RDS 메트릭 등 OpenSearch에 수집된 다양한 데이터를 기반으로 운영, 보안, 성능 관리를 위한 여러 시각화 요소를 설계 및 구성했습니다. 이를 통해 시스템 핵심 지표(KPI), 보안 이벤트 현황, 리소스 사용량 추이 등을 단일 화면에서 실시간으로 파악할 수 있게 되었습니다. 강력한 대시보드는 데이터 기반 의사결정을 지원하고, 문제를 사전에 감지하며, 팀 간의 정보 공유를 촉진하여 전반적인 시스템 관리 효율성을 극대화합니다

# 감사합니다.

	Git	Phone	Email
정원빈	<a href="https://github.com/NoJamBean">https://github.com/NoJamBean</a>	010-9287-1157	gnfkqh@gmail.com
송현섭	<a href="https://github.com/Songhyunseop">https://github.com/Songhyunseop</a>	010-9945-5352	gustjq99455352@gmail.com
김주관	<a href="https://github.com/kindread11">https://github.com/kindread11</a>	010-8001-9511	boris9605@gmail.com
이정규	<a href="https://github.com/timangs">https://github.com/timangs</a>	010-3080-2937	jeongkyu1145@gmail.com