2026.02.09.월

생성일	@2026년 2월 9일 오전 11:48
태그	AWS

목차

▼ 1) AWS 로드밸런서 + 오토스케일링

표준적인 AWS 아키텍처

```
사용자
   ↓
Route53 (DNS)
   ↓
ALB (로드밸런서)
   ↓
Auto Scaling Group
   ↓
EC2 인스턴스들
   ↓
DB (RDS 등)
```

---

1. VPC 및 네트워크

역할: AWS 내 가상 데이터센터, IP 관리 및 트래픽 제어

- **Public Subnet**: ALB, NAT Gateway (외부 접근 가능)
- **Private Subnet**: EC2, DB (외부 차단)
- **없으면**: 서버 인터넷 노출, 보안 붕괴, 장애 시 전체 중단

---

2. 보안 그룹

역할: AWS 방화벽, 포트 및 접근 제어

- 계층형 보안: 외부 → ALB → EC2 → DB
- **없으면**: 서버 직접 노출, 해킹 위험

---

3. Launch Template

역할: 서버 생성 설계도

- 서버 구성 표준화 (OS, 패키지, 설정)
- Auto Scaling 시 동일 환경 서버 자동 복제
- **없으면**: 서버마다 설정 달라짐, 자동 확장 불가

---

4. Target Group

역할: 로드밸런서가 트래픽 보낼 서버 목록 + 상태 관리

- Health Check로 서버 정상 여부 확인
- 트래픽 분산 정책 관리
- **없으면**: 장애 서버에도 트래픽 전달

---

5. ALB (Application Load Balancer)

역할: 트래픽 분산 및 고가용성 확보

- 서버 간 요청 균등 분배
- 장애 서버 자동 차단
- SSL 종료, Layer 7 라우팅
- **없으면**: 서버 하나 장애 시 전체 서비스 다운

---

6. Auto Scaling Group

역할: 서버 수 자동 조절

- 트래픽 증가 → 서버 자동 생성
- 트래픽 감소 → 서버 축소로 비용 절감
- 장애 서버 자동 교체
- **없으면**: 트래픽 급증 시 다운, 수동 복구 필요

---

7. Auto Scaling 정책

역할: 서버 확장/축소 판단 기준

- 예: CPU 70% 이상 → 서버 증가
- 실시간 트래픽 대응
- **없으면**: 확장 타이밍 실패, 비용 낭비

---

8. Health Check

역할: 장애 감지 및 자동 복구

- 서버 상태 모니터링
- 장애 서버 트래픽 차단
- ASG가 새 서버 자동 생성
- **없으면**: 사용자에게 오류 전달, 신뢰성 하락

💡 AWS 로드밸런서 + 오토스케일링 3가지 관점 분석

1️⃣ 아키텍처 관점 (Infrastructure Design)

계층별 구조

```
[Internet Gateway]
      ↓
[Public Subnet - ALB]
      ↓
[Private Subnet - EC2 Auto Scaling Group]
      ↓
[Private Subnet - RDS]
```

각 계층의 설계 목적

네트워크 레이어 (VPC)

- 논리적 격리: 외부와 내부 분리
- Multi-AZ 구성: 가용영역별 서브넷 분산 → 물리적 장애 대응
- CIDR 설계: IP 주소 공간 관리

보안 레이어 (Security Groups)

- Defense in Depth: 다층 방어
- Least Privilege: 최소 권한 원칙
- Stateful 방화벽: 응답 트래픽 자동 허용

컴퓨팅 레이어 (EC2 + ASG)

- Cattle, not Pets: 서버를 교체 가능한 자원으로 취급
- Immutable Infrastructure: 서버 수정 대신 교체
- Horizontal Scaling: 서버 개수로 확장

트래픽 제어 레이어 (ALB + Target Group)

- Single Point of Entry: 단일 진입점
- Service Discovery: 동적 서버 등록/해제
- Health-based Routing: 정상 서버로만 라우팅

고가용성 설계 원칙

```
AZ-A: ALB (Active) + EC2 (2개)
AZ-B: ALB (Active) + EC2 (2개)
```

- **단일 장애점 제거**: 모든 컴포넌트 이중화
- **자동 장애 조치**: Health Check → 자동 복구
- **탄력성**: 트래픽 변화에 자동 대응

2️⃣ 사용자 흐름 관점 (User Journey)

정상 시나리오

```
1. 사용자가 example.com 접속
   ↓
2. DNS가 ALB의 Public IP 반환
   ↓
3. ALB가 요청 수신 (HTTPS)
   ↓
4. ALB가 Target Group의 Health Check 결과 확인
   ↓
5. 정상 EC2 중 하나 선택 (라운드 로빈)
```

```
   ↓
6. EC2가 요청 처리
   ↓
7. 응답을 ALB → 사용자에게 전달
```

사용자 경험: 투명함 (어떤 서버가 처리했는지 모름)

트래픽 급증 시나리오

```
1. 사용자 100명 → 1000명 급증
   ↓
2. EC2 CPU 사용률 70% 초과
   ↓
3. CloudWatch Alarm 발동
   ↓
4. Auto Scaling Policy 실행
   ↓
5. Launch Template으로 새 EC2 2개 생성
   ↓
6. 새 EC2가 Health Check 통과
   ↓
7. Target Group에 자동 등록
   ↓
8. ALB가 4개 서버로 트래픽 분산
```

사용자 경험: 지연 없이 정상 서비스 (자동 확장)

서버 장애 시나리오

```
1. EC2-1번 서버 장애 발생
   ↓
2. ALB Health Check 실패 (3회 연속)
   ↓
3. Target Group에서 EC2-1 제외
   ↓
4. 남은 서버로만 트래픽 분산
   ↓
5. ASG가 Unhealthy 인스턴스 감지
   ↓
6. EC2-1 종료 후 새 인스턴스 생성
   ↓
7. 새 인스턴스 Health Check 통과
   ↓
8. Target Group에 재등록
```

사용자 경험: 일부 요청만 실패 (대부분 정상)

3️⃣ 데이터 플로우 관점 (Data Flow)

HTTP 요청 데이터 흐름

```
[사용자 브라우저]
  | HTTP GET /products
  ↓
[Internet Gateway]
  | 패킷 라우팅
  ↓
[ALB - Public Subnet]
  | - SSL 종료 (HTTPS → HTTP)
  | - Connection Pooling
  | - X-Forwarded-For 헤더 추가
```

```
  ↓
[Security Group 검증]
  | 소스: ALB Security Group
  | 포트: 80
  ↓
[EC2 Instance - Private Subnet]
  | - 애플리케이션 처리
  | - DB 쿼리
  ↓
[RDS - Private Subnet]
  | SQL 쿼리 실행
  ↓
[응답 역순 전달]
  EC2 → ALB → 사용자
```

Health Check 데이터 흐름

```
[ALB]
  | 매 30초마다
  ↓
[각 EC2에 Health Check 요청]
  GET /health
  ↓
[EC2 응답]
  - 200 OK → Healthy
  - Timeout/5xx → Unhealthy
  ↓
[Target Group 상태 업데이트]
  | 연속 2회 성공 → Healthy
  | 연속 3회 실패 → Unhealthy
  ↓
[ALB 라우팅 테이블 갱신]
```

데이터: HTTP Status Code만 전달 (최소 트래픽)

메트릭 데이터 흐름

```
[EC2 Instance]
  | CPU, Memory, Network 사용률
  ↓
[CloudWatch Agent]
  | 1분마다 수집
  ↓
[CloudWatch Metrics]
  | 데이터 저장 및 집계
  ↓
[CloudWatch Alarm]
  | 조건 평가: CPU > 70%
  ↓
[Auto Scaling Policy]
  | Scale Out 명령
  ↓
[EC2 Auto Scaling]
  | Launch Template 기반 인스턴스 생성
```

데이터: 시계열 메트릭 (시간별 집계)

로그 데이터 흐름

```
[ALB Access Logs]
  | 모든 요청 기록
  ↓
[S3 Bucket]
  | 압축 저장 (Gzip)
  ↓
[Athena / CloudWatch Logs Insights]
  | SQL 쿼리로 분석
```

```
[EC2 Application Logs]
  ↓
[CloudWatch Logs Agent]
  ↓
[CloudWatch Log Groups]
  ↓
[CloudWatch Insights / Lambda]
  | 실시간 분석 및 알림##
```

❓ CloudWatch와 AutoScaling의 관계

컴포넌트	역할	비유
EC2	메트릭 발생	환자 (체온 측정)
CloudWatch	메트릭 수집 및 저장	체온계
CloudWatch Alarm	임계값 감지 및 경고	경보기
Auto Scaling Policy	대응 방법 정의	처방전
Auto Scaling Group	실제 액션 실행	의사 (처방 실행)

정책별 관리 방식

정책 유형	CloudWatch 생성	정책 변경 시 반영	관리 난이도
Target Tracking	자동	✅ 자동 반영	쉬움
Step Scaling	수동	❌ 수동 수정	어려움
Simple Scaling	수동	❌ 수동 수정	어려움
Scheduled	사용 안 함	N/A	중간

Target Tracking, Step/Simple Scaling

구분	Target Tracking	Step Scaling	Simple Scaling
설정	목표 값 하나	여러 단계 규칙	조건 하나, 액션 하나
판단	AWS가 자동	사용자 정의	고정 액션
유연성	낮음 (AWS 방식)	높음 (세밀 제어)	없음 (고정)
복잡도	쉬움	어려움	매우 쉬움
Scale In	자동	별도 정책 필요	별도 정책 필요
추천	일반적인 경우	복잡한 요구사항	거의 사용 안 함

▼ 2) CLI로 키 페어 생성

```
# RSA 2048비트 SSH 키 쌍(개인키/공개키)을 .ssh 폴더에 생성하고 패스프레이즈 없이 저장한다.
ssh-keygen -t rsa -b 2048 -C "" -f "/Users/sanggyoon/.ssh/kakao_keypair_ksg" -N ""

# 생성된 개인키 파일을 .pem 확장자 파일로 복사한다. 꼭 필요한 명령어는 아님
cp /Users/sanggyoon/.ssh/kakao_keypair_ksg /Users/sanggyoon/.ssh/kakao_keypair_ksg.pem

# 로컬에 생성된 공개키를 AWS EC2 Key Pair로 등록한다.
aws ec2 import-key-pair --key-name "kakao_keypair_ksg" --public-key-material fileb://$HOME/.ssh/kakao_
```

```
keypair_ksg.pub
# 기본 리전에 등록되지만 리전을 특정할 수 있는 명령어를 추가할 수 있다. --region us-east-1 (버지니아 리전에 공개키 등록)

# 개인키 파일을 사용자 본인만 읽을 수 있도록 권한을 제한한다.
chmod 400 ~/.ssh/kakao_keypair_ksg

# .pem 개인키 파일도 동일하게 보안 권한을 제한한다.
chmod 400 ~/.ssh/kakao_keypair_ksg.pem
```

▼ 3) Bastion 중계서버

네트워크 → 공개 서브넷의 중계 서버(pem 프라이빗 키 소유) → 프라이빗 서브넷

❓ **Public subnet의 ec2가 갖고있는 pem은 결국 위험에 노출되어 있는거 아닌가?**

→ 맞음. 그래서 bastion 서버는 비공개키를 갖고 있지않고 다른 곳에 존재하는 pem을 사용한다.

SSH Agent

클라이언트 pc의 메모리에 비공개키를 저장 (파일로 저장 x)

인증 요청을 할 때 사용

```
# 전통적인 접근
[사용자 브라우저]
        ↓
[Load Balancer]
        ↓
[App Server (프론트 + 백엔드)]
        ↓
[DB]

# 요즘 사용하는 웹 서비스의 고객 접근 경로
사용자 브라우저
        ↓
   CDN (CloudFront 등)
        ↓
   프론트 정적 파일
        ↓
브라우저 실행
        ↓
   API 호출
        ↓
Load Balancer
        ↓
Backend(App 서버)
        ↓
DB

# 결론: 정적 페이지는 LB앞에 있고 서비스 로직은 뒤에 존재함
```