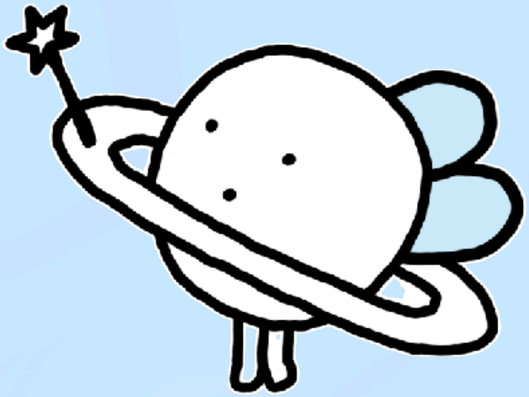
10만명의 사용자를 받아들일 수 있는 아키텍처

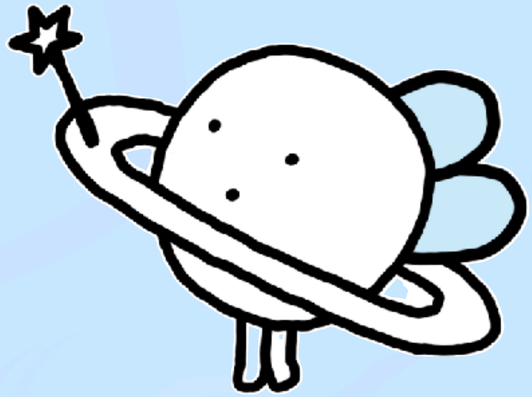
BE 7기 밍트

개요



1. 현재 상황 & 문제점
2. 10만 명을 위한 아키텍처 변화 4가지
3. 트레이드오프 분석
4. 결론

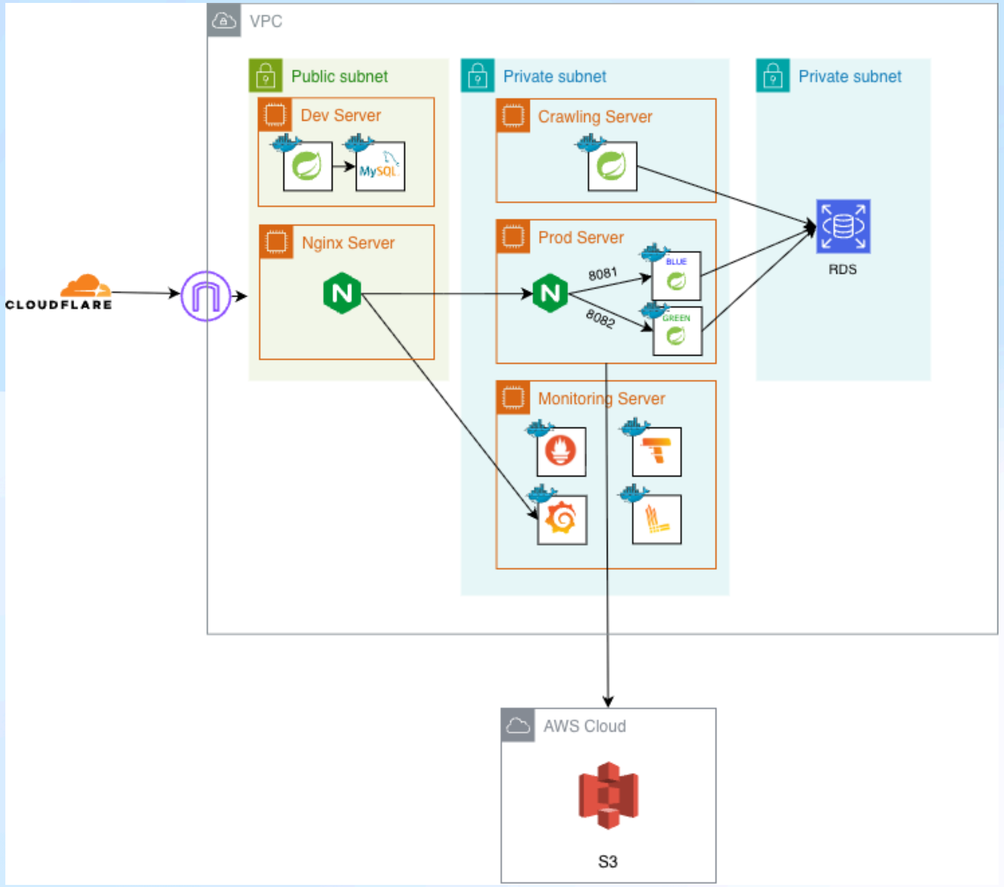
개요



1. 현재 상황 & 문제점
2. 10만 명을 위한 아키텍처 변화 4가지
3. 트레이드오프 분석
4. 결론

1. 현재 상황 & 문제점

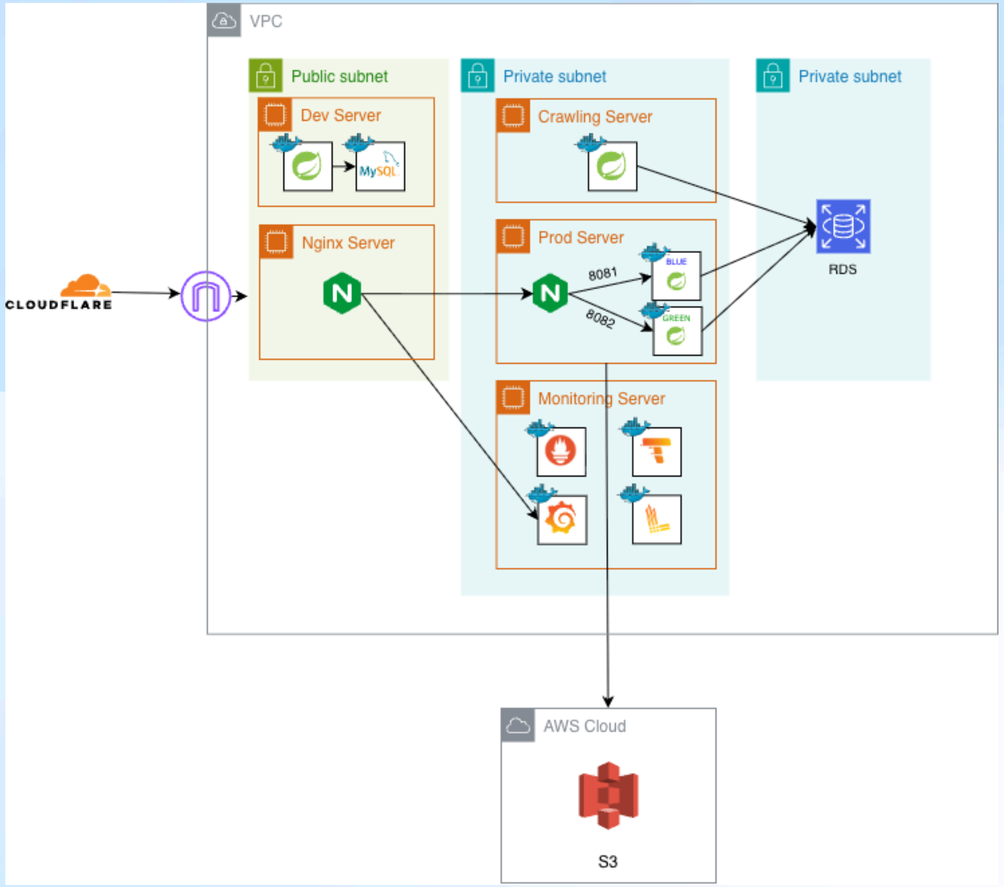
현재 아키텍처



- CloudFlare : DDoS 보호 + WAF 기반 보안 게이트웨이
- Nginx → EC2 → RDS(Mysql)
- Max TPS : 80
- 응답 시간 : 200~500ms

1. 현재 상황 & 문제점

10만명이 접속한다면?



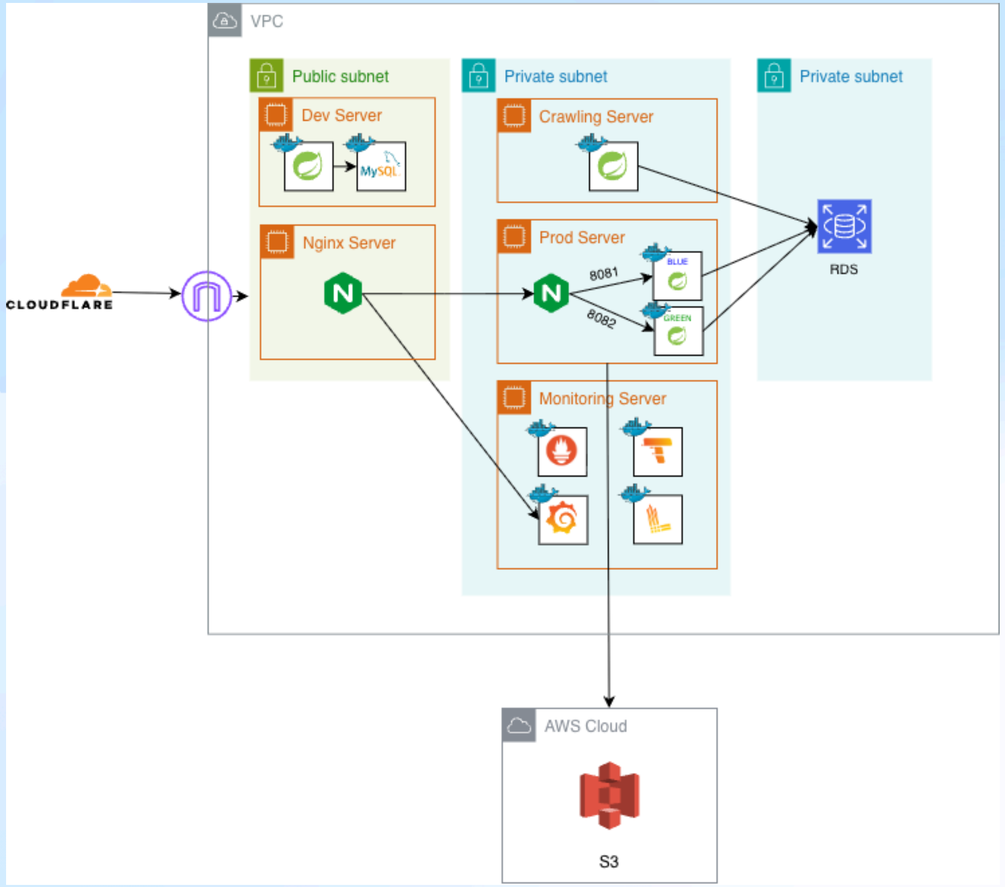
항목	현재	10만 명	증가율
MAU	140명	100,000명	714배
DAU	15~20명	15,000명	750~1,000배
피크 동시 접속	5~10명	1,500명	150~300배
피크 TPS	2~3	100	33~50배
경기당 인증	3~5명	1,500명	300~500배

✅ 현재는 매우 안정적 (리소스 여유)

10만명 규모라면..? ☠️

1. 현재 상황 & 문제점

10만명이 접속한다면?



1. SPOF 문제 (단일 장애 지점)

- Nginx : 단일 진입점
- EC2 1대 : 장애 시 전체 서비스 중단

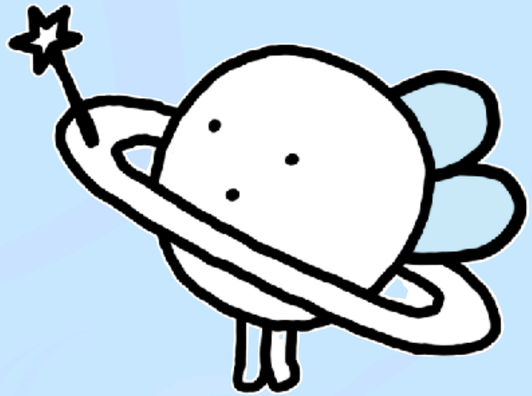
2. 서버 확장성 부족

- TPS 100 이상 감당 불가
- CPU & 스레드 한계

3. 데이터베이스 읽기 부하 & 쓰기 병목

- 읽기 부하 → 쿼리 대기 / 응답 지연
- 쓰기 폭주 → 락 경합 → 커넥션 풀 포화 / 타임아웃

개요

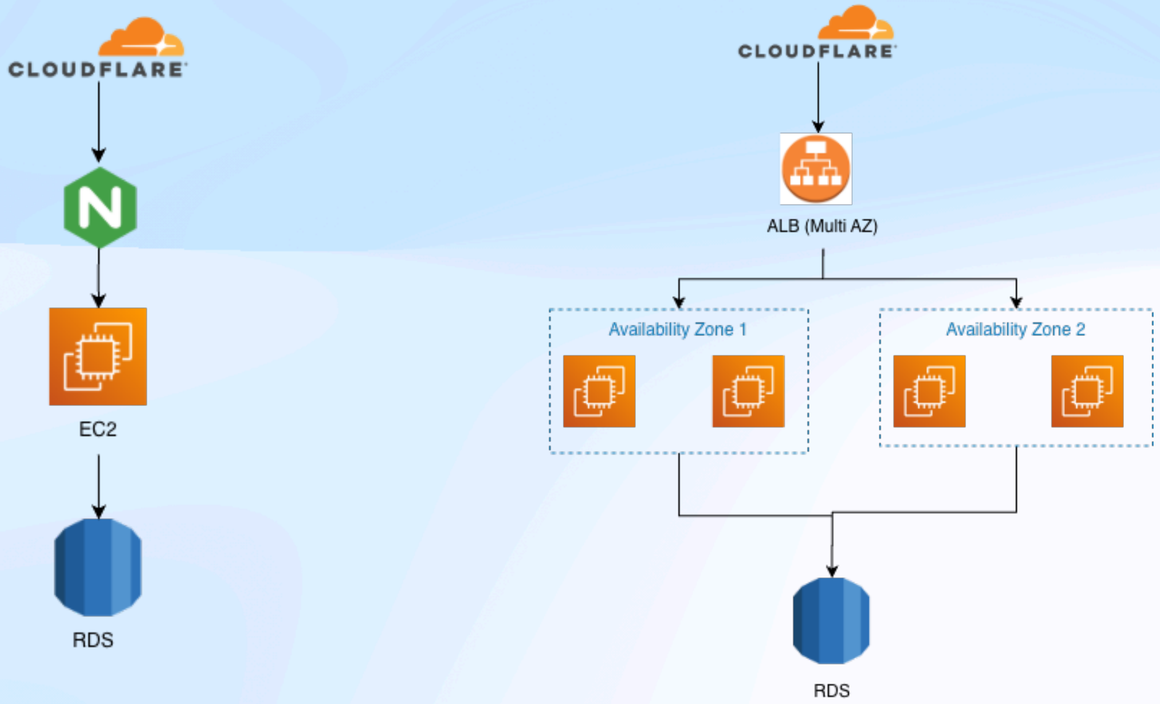


1. 현재 상황 & 문제점
2. **10만 명을 위한 아키텍처 변화 4가지**
3. 트레이드오프 분석
4. 결론

2. 10만 명을 위한 아키텍처 변화 4가지

1️⃣ SPOF 제거 + 서버 확장

Before → After



ALB (Application Load Balancer)

- Multi-Az로 구성
- 헬스 체크를 통한 장애 서버 제거
- 트래픽 자동 분산

Auto Scaling

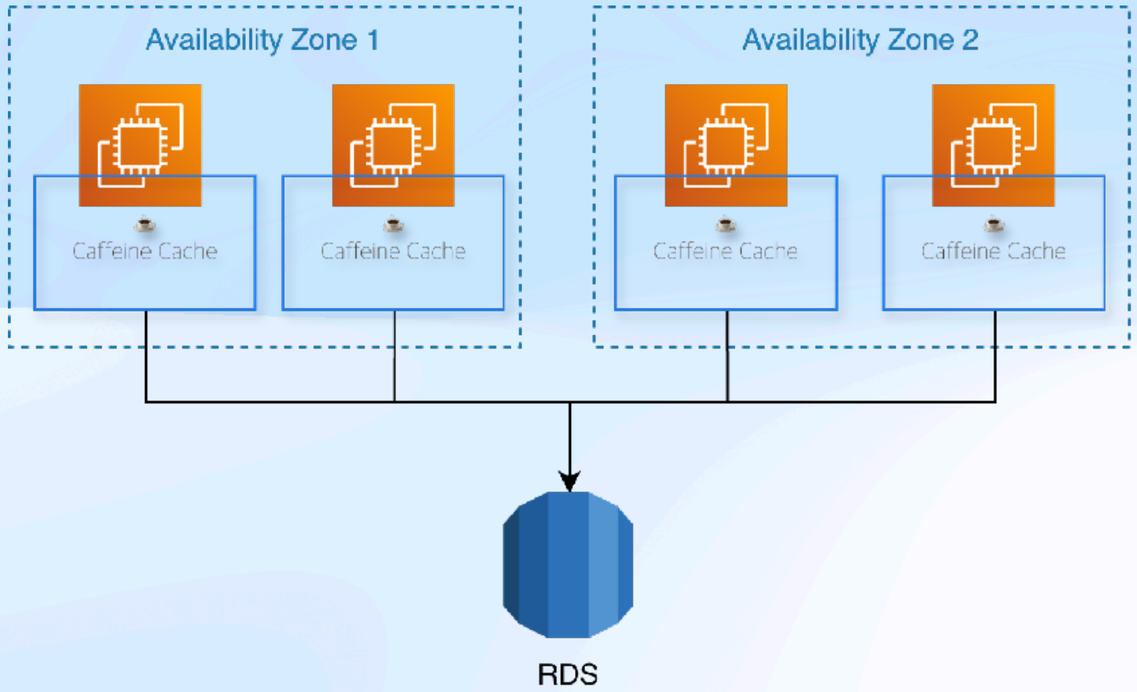
- CPU 70% 이상 → 서버 자동 추가
- CPU 30% 이하 → 서버 자동 축소
- Scheduled Scaling: 경기 시작 1시간 전 5대로 증설

고가용성(HA), 장애 조치 🆚 비용 증가, 관리 복잡도 증가

+ TPS ⬆️

2. 10만 명을 위한 아키텍처 변화 4가지

2️⃣ Caching 전략 수정



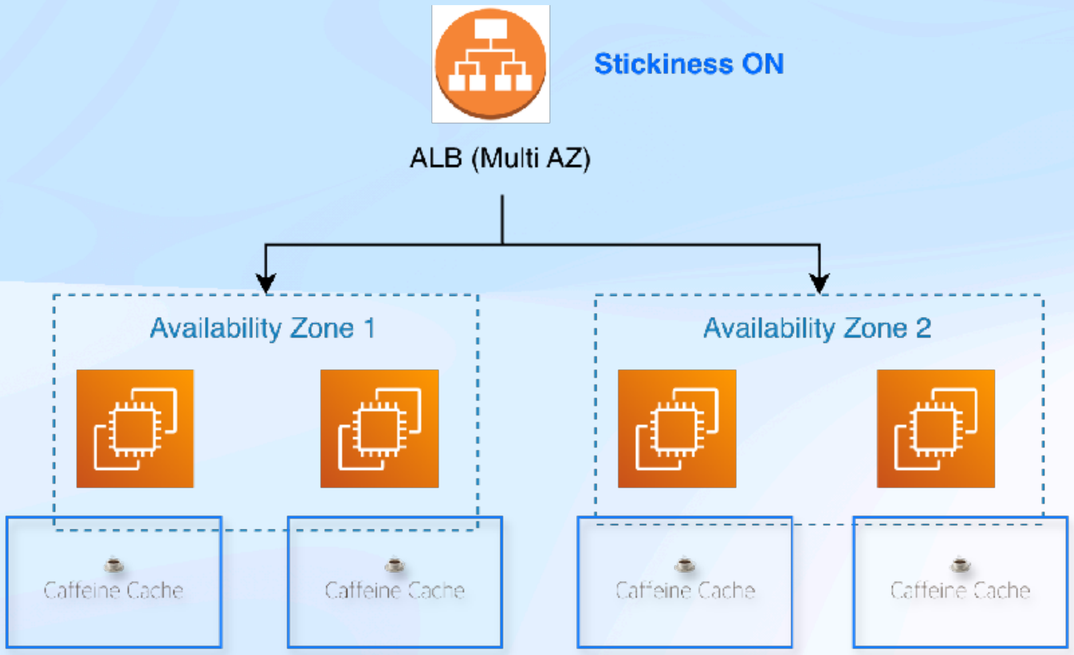
로컬 캐시가 서버마다 존재한다면?

- 서버마다 다른 캐시 → **데이터 일관성** ❌
- 캐시 중복 저장 → **비효율적인 메모리 사용**
- 서버 재시작 시 **Cache Stampede**

→ **Cache Hit Rate** ⏬

2. 10만 명을 위한 아키텍처 변화 4가지

2️⃣ Caching 전략 수정

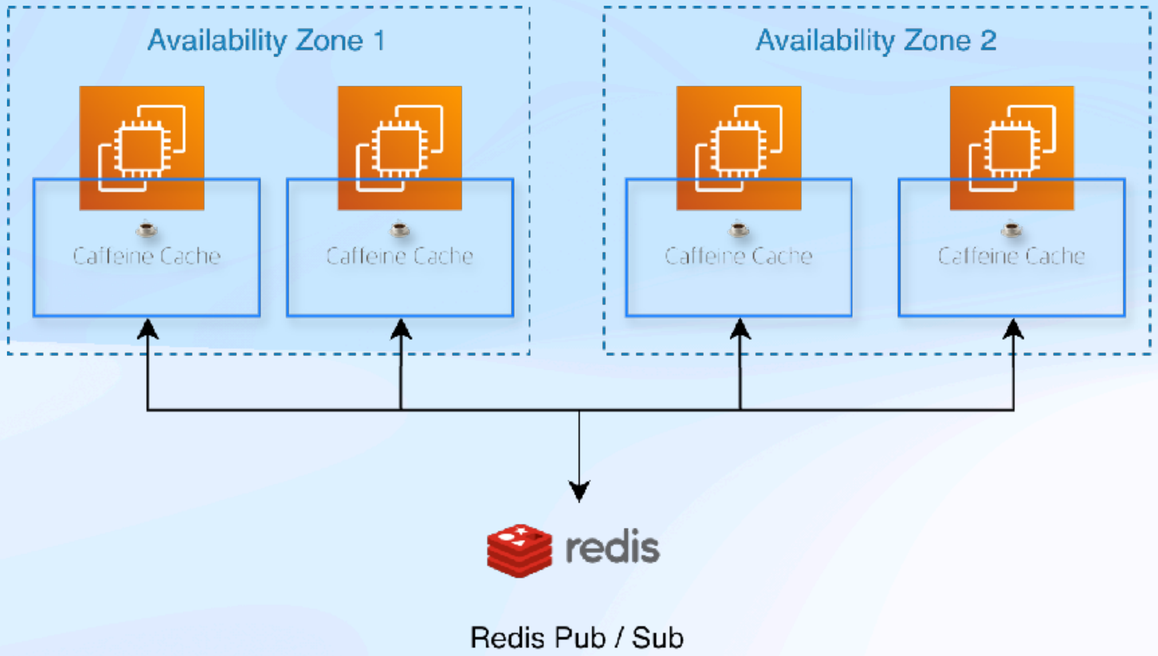


1) Sticky Session ?

- Load Balancer가 항상 같은 사용자를 같은 WAS로 보냄
- 장점
  - 응답 속도 빠름
  - 비용 0달러, 구현 단순
- 단점
  - 데이터 일관성 ❌ (사용자별 데이터만 일관성 보장)
  - 부하 불균형 발생
  - WAS 재시작시 캐시 소실
  - Auto Scaling시 세션 재분배 복잡

2. 10만 명을 위한 아키텍처 변화 4가지

2 Caching 전략 수정

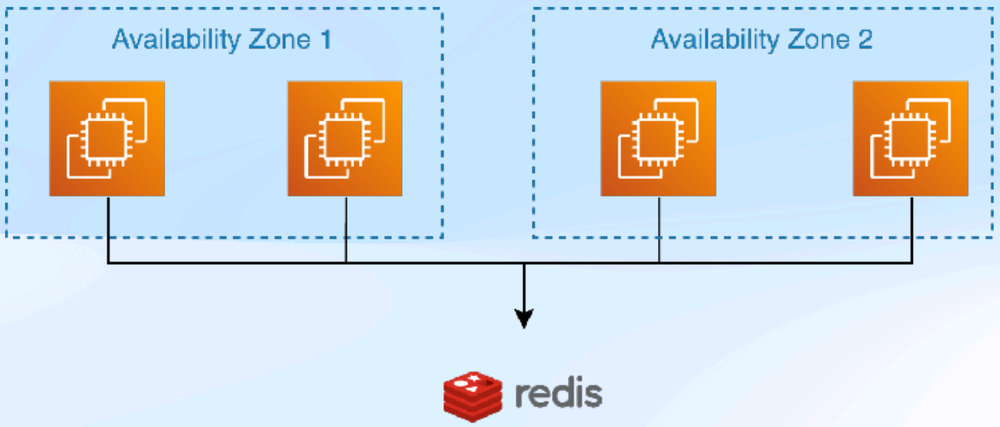


2) Redis Pub / Sub

- 각 WAS는 독립적인 캐시 보유
- Redis는 **동기화 메세지(캐시 무효화)만 전달**
- 장점: 로컬 캐시 유지 → 응답 속도 높음
- 단점: 동기화로 인한 지연 → 일시적 캐시 불일치

2. 10만 명을 위한 아키텍처 변화 4가지

2️⃣ Caching 전략 수정

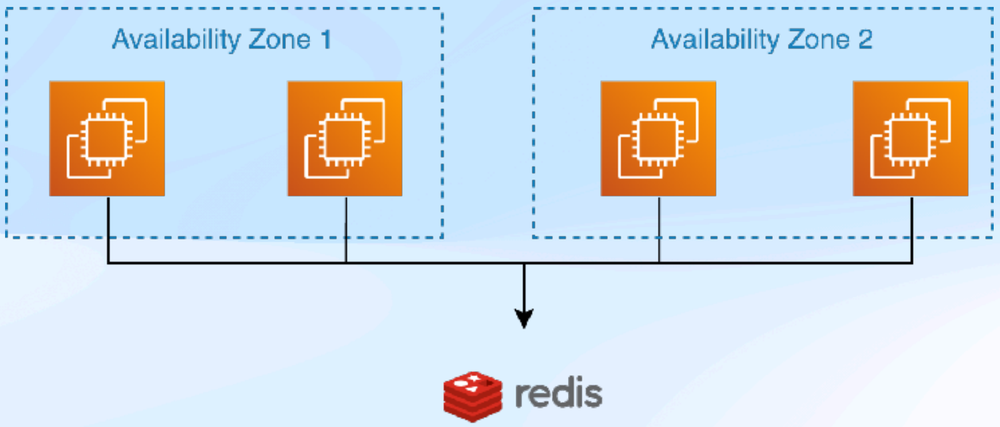


3) Redis에 캐시 저장

- 모든 캐시를 Redis에만 저장 (로컬 캐시 제거)
- 장점
  - **데이터 일관성 유지**
  - Cache Hit Rate ⬆ (여러 서버가 하나의 캐시 공유)
  - 관리 쉬움
- 단점
  - 네트워크 레이턴시 존재
  - 비용 증가

2. 10만 명을 위한 아키텍처 변화 4가지

2️⃣ Caching 전략 수정 - 3) Redis에 캐시 저장 채택 ✅



1) **Auto Scaling 대응**

WAS를 증설할 때 새 WAS가 Redis 캐시 즉시 사용 가능

2) **운영 편의성**

캐시 모니터링, 무효화를 Redis 한 곳에서만 수행

3) **부하 분산**

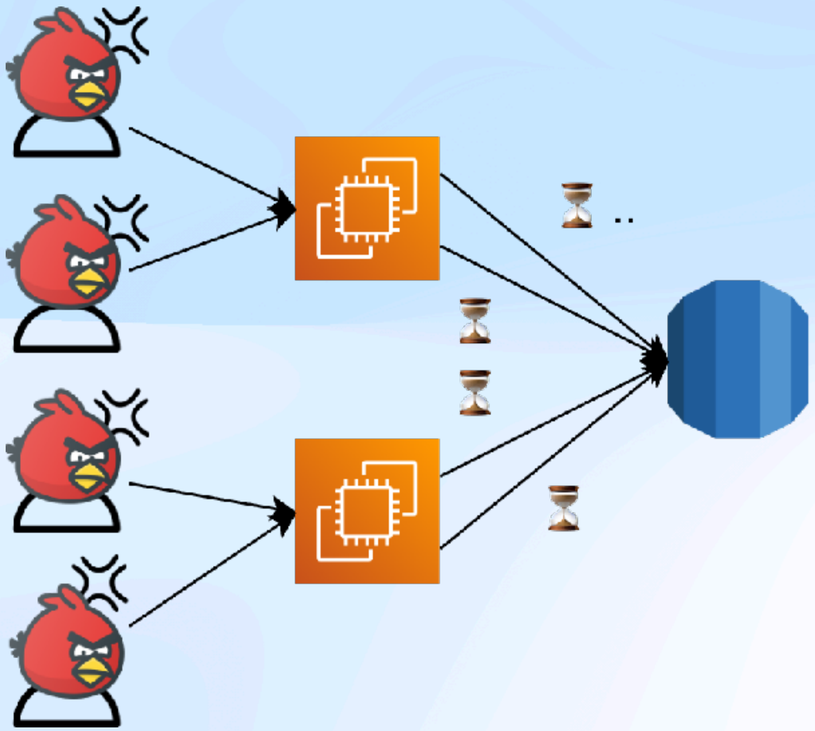
트래픽을 균등하게 분산

- **트레이드오프**
  - 네트워크 레이턴시 & 비용 OK

2. 10만 명을 위한 아키텍처 변화 4가지

2000명이 동시에 요청을 보낸다면? - DB 병목

Before



모든 요청이 동시에 MySQL 도착

→ MySQL 동시 쓰기 2,000개 처리 필요

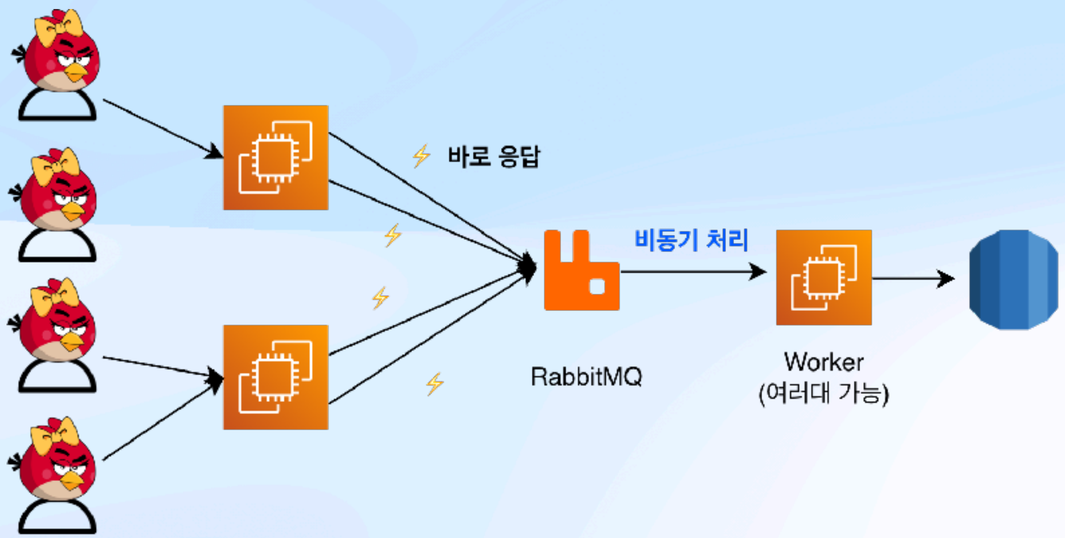
발생 가능한 문제

- 커넥션 풀 부족
- 응답 시간 1초 이상
- 타임아웃 에러 발생
- 사용자 불만 증가

2. 10만 명을 위한 아키텍처 변화 4가지

3 비동기 메세지 큐 도입

After



실시간이 아니어도 되는 일은 분리 (비동기 처리)

요청 순서대로 메세지 큐에 저장 & Worker 순차 처리

개선 효과

- 즉시 응답 (사용자 만족도 ⬆)
- DB 부하 분산 & 타임아웃 감소

단점

- 실시간 반영 ❌
- 운영 복잡도 ⬆, 비용 증가

2. 10만 명을 위한 아키텍처 변화 4가지

3️⃣ 비동기 메세지 큐 도입

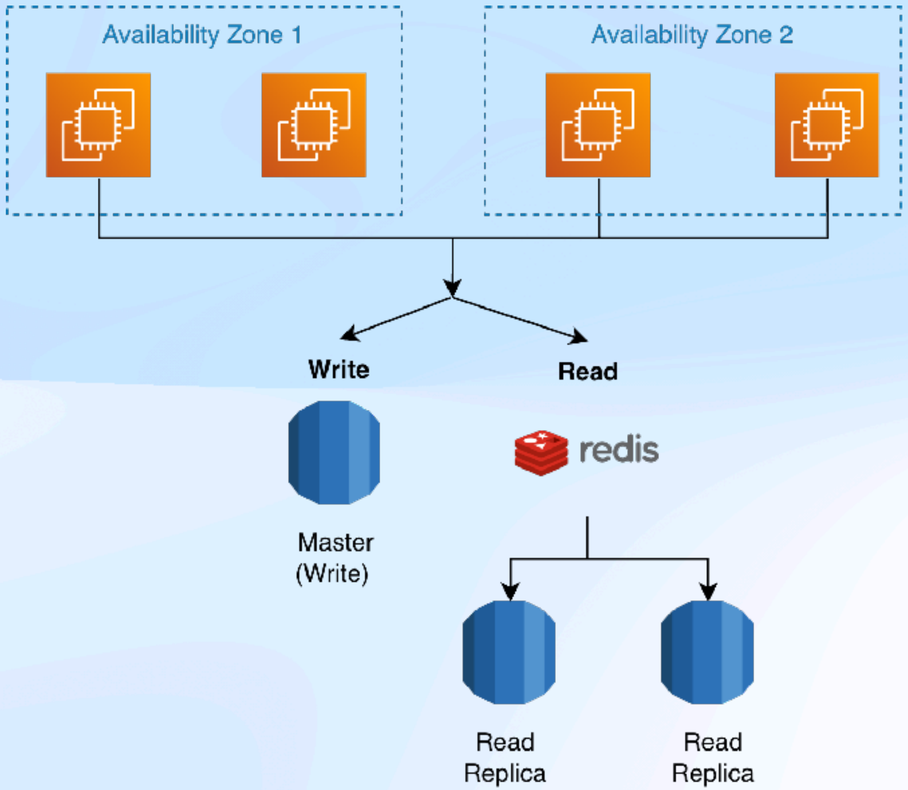
RabbitMQ vs Kafka

항목	RabbitMQ	Kafka
처리량	약 10만 msg/s	약 100만 msg/s
구축 난이도	쉬움	어려움
운영 복잡도	낮음	높음
메시지 영속성(유실 안되는 정도)	약함 (일반 용도 충분)	강함 (로그/이벤트 스트림)
10만 명 규모 적합성	적합 (✅ 충분)	과도 (❌ 오버스펙)

Kafka는 10만 규모에서 오버엔지니어링

2. 10만 명을 위한 아키텍처 변화 4가지

4 Read Replica 구성



사용자의 80%는 읽기 작업 수행

→ 읽기 부하 높음

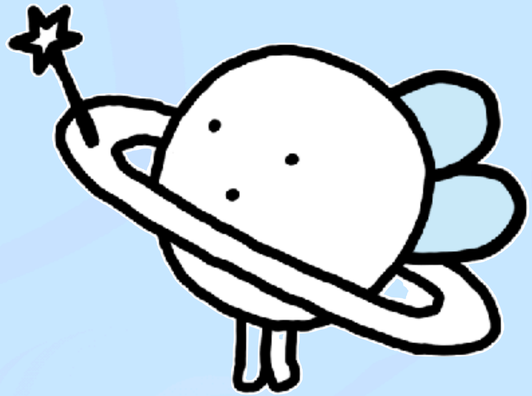
장점

- 읽기 성능 향상 (replica 수만큼)
- DB 부하 분산, 확장 가능

단점

- 복제 지연
- 비용 증가

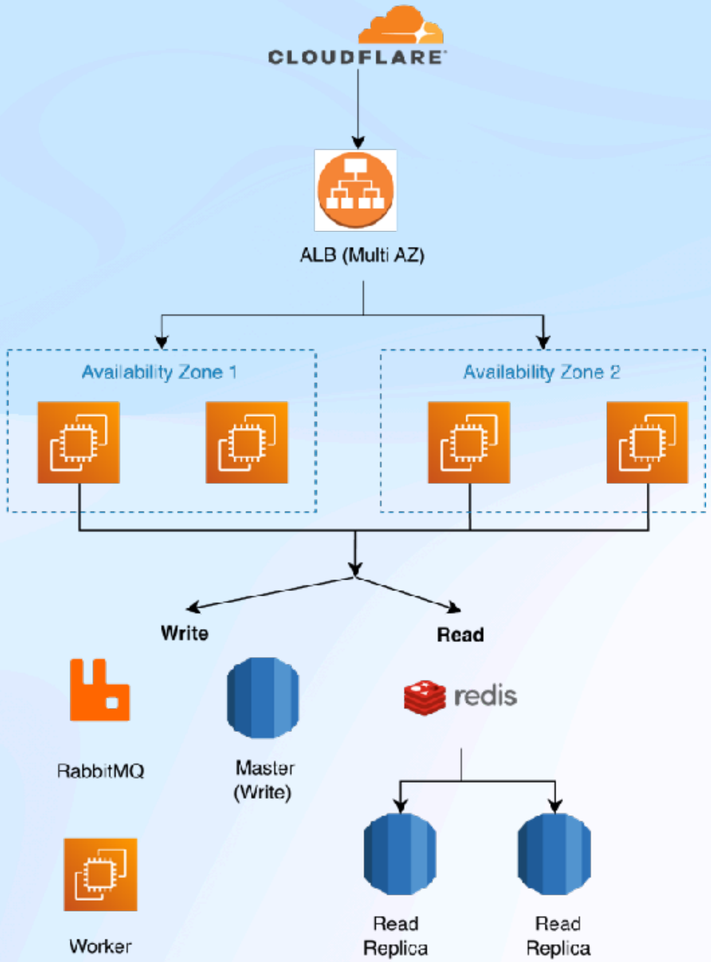
개요



1. 현재 상황 & 문제점
2. 10만 명을 위한 아키텍처 변화 4가지
3. **트레이드오프 분석**
4. 결론

3. 트레이드오프 분석

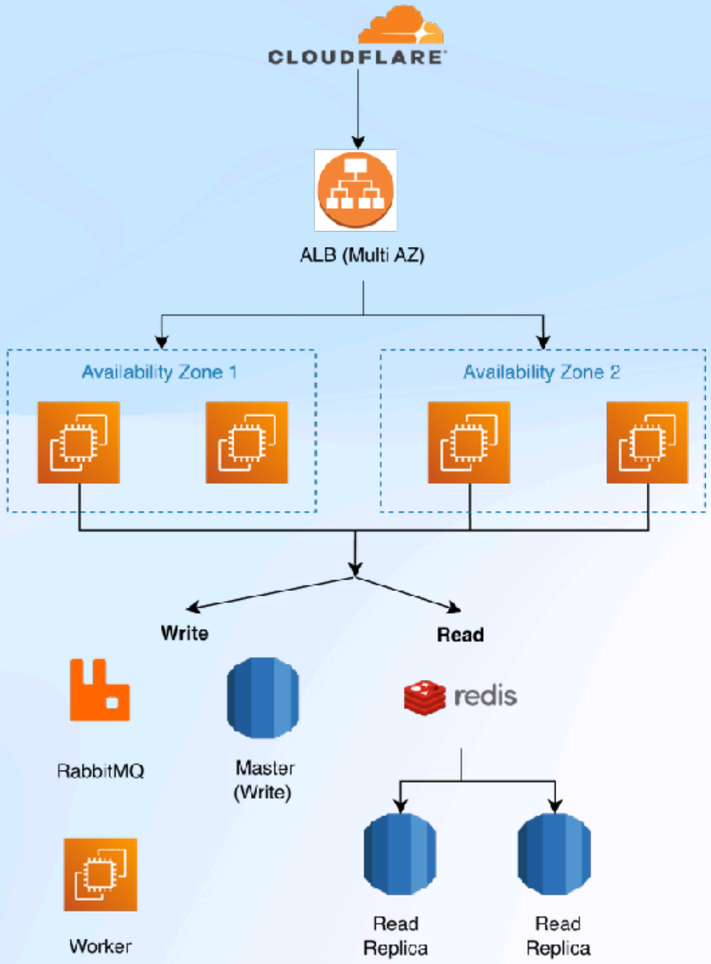
최종 아키텍처



1. ALB & Auto Scaling
2. Redis Caching
3. 메세지 큐 도입 (RabbitMQ)
4. Read Replica 도입

3. 트레이드오프 분석

비용 vs 성능, 가용성



비용 3배 증가

VS

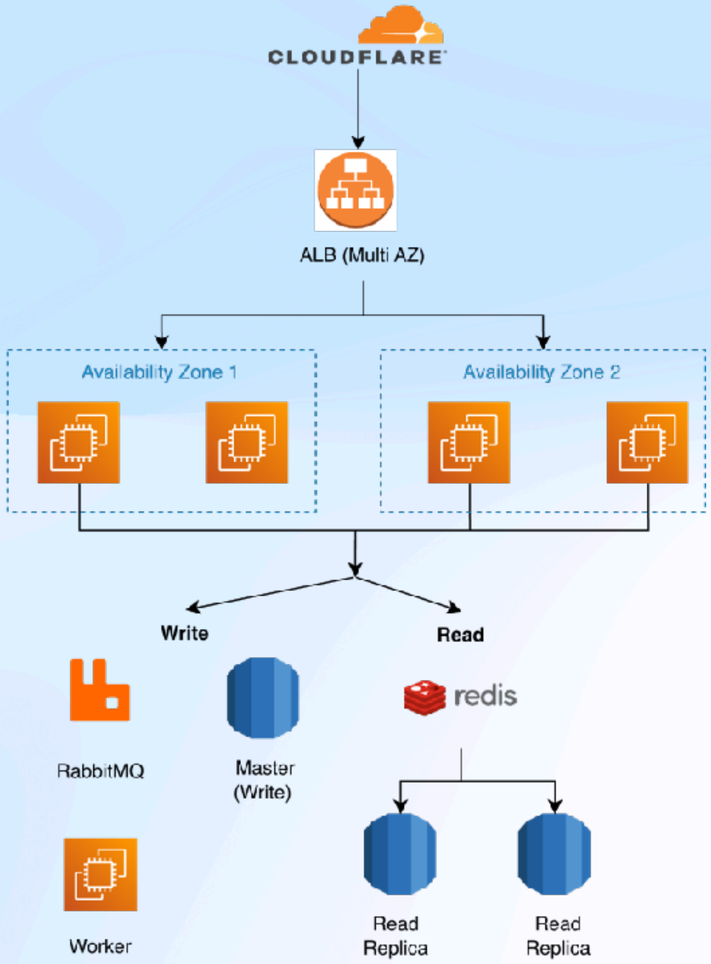
가용성 99.99% & TPS 증가

조회 속도 증가 (Redis)

쓰기 응답 속도 증가 (메세지 큐)

3. 트레이드오프 분석

실시간성 (동기) vs 안정성 (비동기)



바로 DB에 반영되어야하는 작업

최신 데이터를 반드시 조회해야하는 작업

ex) 로그인, 직관 인증 여부 조회 → Master에서 바로 조회

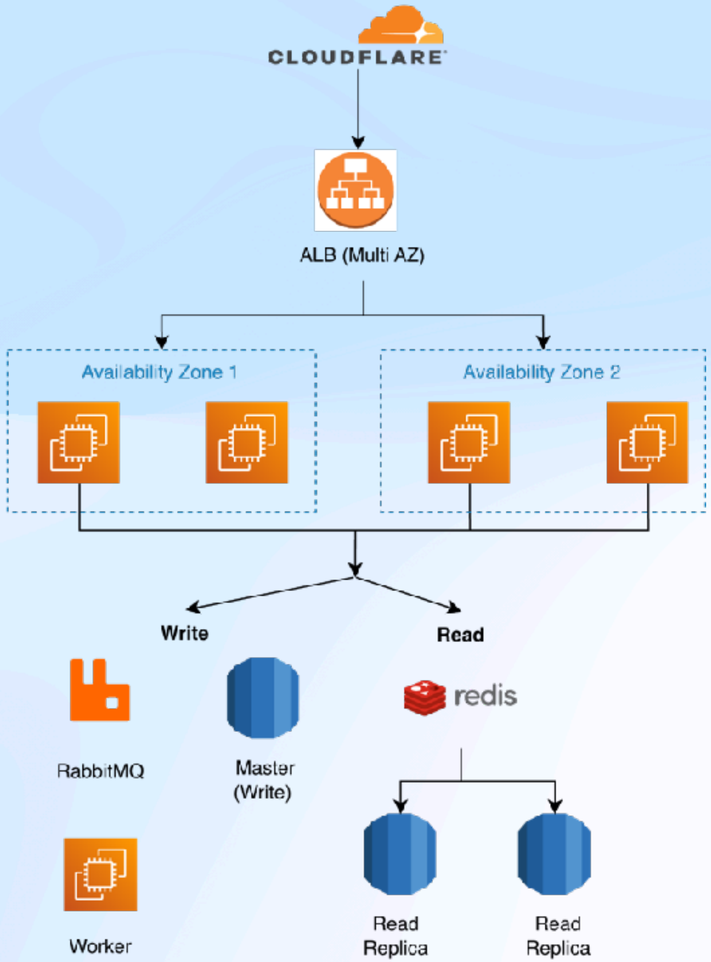
VS

DB에 천천히 안전하게 반영

ex) 좋아요, 채팅 → 메세지 큐

3. 트레이드오프 분석

허용 가능한 것 / 허용 불가능한 것



비용 (과도 X)

데이터 지연 (일부 기능 제외)

운영 복잡도 증가

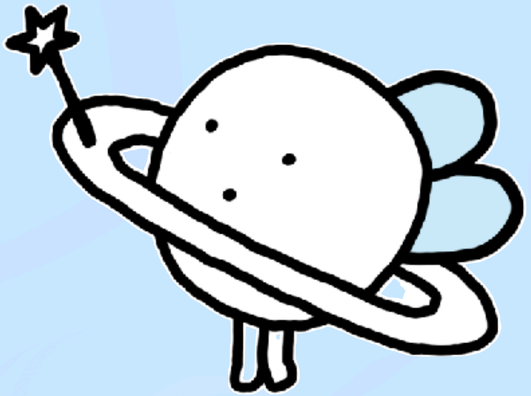
VS

가용성 저하

응답 시간 1초 이상

데이터 유실

개요



1. 현재 상황 & 문제점
2. 10만 명을 위한 아키텍처 변화 4가지
3. 트레이드오프 분석
4. 결론

3. 트레이드오프 분석

결론

단계적으로 확장하자.

추측하지 말고, 모니터링을 기반으로 결정하자.

오버엔지니어링 하지 말자.

3. 트레이드오프 분석

100만명의 사용자를

받아들일 수 있는 아키텍처도

많관부

꼿