경력 기술서

지원자 이권능

1. 업무 요약

제목	포인트 지급 프로세스 자동화 및 운영 효율화
성과	• 포인트 지급 자동화로 한 달 평균 8시간의 수작업 업무를 제거 • 운영 효율성을 높여 개발자와 마케터의 커뮤니케이션 비용을 최소화
업무	• AWS Event Bridge를 활용한 스케줄러 동적 생성 구현 • 포인트 지급 데이터 S3 저장 및 호출 • 스케줄러 동작 시 Lambda로 API 요청 • 지급 결과를 Slack으로 실시간 알림 전송
기술	Spring Boot, Kotlin, AWS (S3, Lambda, EventBridge), Slack API

제목	MongoDB Query Targeting Alert 이슈 해결 및 쿼리 성능 개선
성과	• 쿼리 최적화로 Latency를 평균 750ms에서 6ms로 99.2% 단축 • 문서화를 통해 잘못된 인덱스 사용 사례 공유 및 동일 문제 재발 방지
업무	• Query Targeting Alert 원인을 분석하여 Partial Index를 활용하지 못한 쿼리 조건을 최적화 • Slack Alert 적용으로 동일 문제 발생 시 신속한 대응을 할 수 있도록 개선
기술	Spring, Kotlin, MongoDB

제목	NGINX 무중단 배포 과정의 결함 개선
성과	• 무중단 배포 프로세스 안정성 확보 및 서비스 장애 예방

업무	• 무중단 배포 중 Swagger 및 일부 API 비정상 동작 문제 분석 • 배포 시 JAR 파일 덮어쓰기 문제를 원인으로 식별 • 전환 프로세스 간 파일명 충돌이 발생하지 않도록 배포 스크립트 수정
기술	Jenkins, Shell script

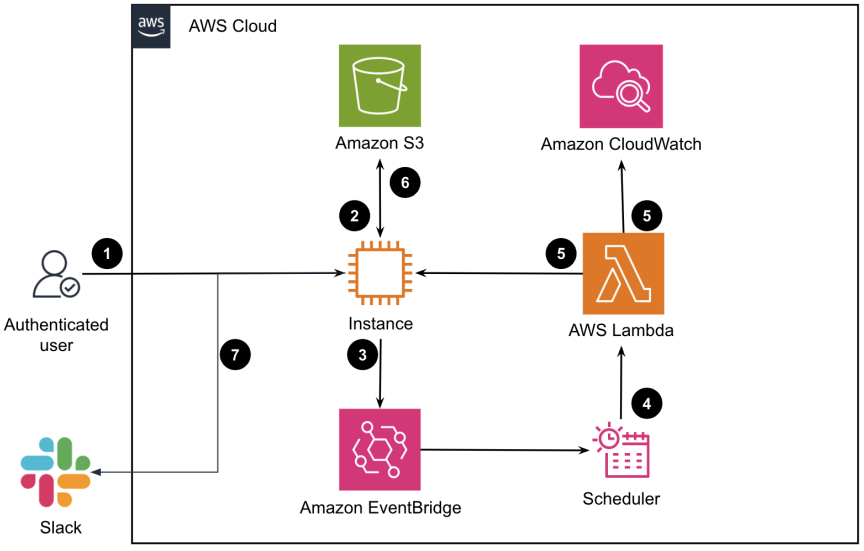
2. 업무 상세 기술

2-1. 포인트 지급 프로세스 자동화 및 운영 효율화

멘토링 세션에서 가장 공감되는 말이 있었습니다. “지나간 업무에 파묻히면 나아갈 동력을 잃게 된다.” 그래서 저는 운영 자동화를 최우선 업무로 선정하고 과제를 진행했습니다.

저희 서비스에서 가장 비효율적으로 느껴지는 운영 업무는 온체인 포인트 지급 처리입니다. 기존의 업무 프로세스는 개발자가 고객 정보를 받아서 고객 한 명씩 개별적으로 API를 요청하는 수동 방식으로 운영되었습니다. 단순 반복 업무로 개발자의 공수를 낭비하게 하고 마케팅 팀과의 커뮤니케이션 비용까지 필요한 비효율적인 구조였습니다. 그래서 포인트 지급 자동화의 필요성을 절실히 느끼고 다음과 같은 개선 작업을 진행했습니다.

작업 세부 내용



운영자가 어드민 페이지에서 포인트 지급 엑셀 데이터와 날짜를 미리 등록하면, 자동으로 포인트가 지급되고 결과를 Slack으로 전달하는 프로세스를 구상하고 작업했습니다. 포인트는 한 달에 약 1~2회 정도 행사가 진행될 때마다 지급하며, 지급일이 유동적입니다. 그래서 동적으로 등록하고 1회 실행 이후에 삭제할 수 있는 스케줄러가 필요했습니다. 사용 빈도가 높지 않기 때문에 AWS EventBridge를 활용하여 스케줄러를 동적으로 생성하도록 구현했습니다. 스케줄러가 지정된 시간에 동작하면 Lambda가 서버에 API 요청을 전송하여 포인트를 지급합니다. 이 때, S3에 등록했던 엑셀 데이터를 사용합니다. 지급 결과는 Slack 알림을 통해 성공 및 실패 유저 수를 공유하게 됩니다.

성과

한 달 평균 8시간 이상 소요되던 기존의 수동 포인트 지급 프로세스를 자동화하여 개발자의 공수가 들지 않도록 개선했습니다. 마케터가 개발자를 거치지 않고 직접 포인트를 지급할 수 있게 되어서 커뮤니케이션 비용도 최소화 했습니다. 그리고 Slack 알림을 도입함으로써 포인트 지급 실패 이슈에 대해 즉각적으로 대응할 수 있도록 했습니다.

2-2. MongoDB Query Targeting Alert 이슈 해결 및 쿼리 성능 개선

멘토링 세션에서 정확한 수치와 메트릭을 통해 성능을 개선하고 안정적인 운영을 하는 점이 인상 깊었습니다. 아쉽게도 제가 개발하는 환경에서는 실시간 메트릭을 확인할 수 있는 인프라가 많지 않았습니다. 그래도 사용중인 Cloud 서비스에서 제공하는 메트릭을 통해 성능을 개선할 수 있었고, 모니터링을 할 수 있는 환경을 구축하는게 얼마나 중요한지 깨달을 수 있었습니다.

MongoDB Cloud에서 제공하는 메트릭 수치 중에서 Query Targeting 항목이 불안정하게 튀는 현상을 발견할 수 있었습니다. Alert 역시 해당되는 경고가 반복적으로 발생했습니다. Query Targeting: Scanned Objects / Returned has gone above N 경고는 쿼리 실행 시 반환된 문서 수에 비해 지나치게 많은 문서를 스캔할 때 발생합니다. 쿼리 또는 인덱스쪽에 문제가 있을 것으로 예상하고 다음과 같은 개선을 진행했습니다.

작업 세부 내용

원인 분석을 위해 Query Insight의 Profile에서 이상치가 발생한 시간과 타겟한 컬렉션을 확인하고, Cloud Watch에 기록된 로그를 통해서 문제가 된 쿼리를 찾을 수 있었습니다. 이미 인덱스는 걸려있었지만 Partial Index이기 때문에 문제가 발생한 것처럼 보였습니다. 실행 계획을 확인해보니 올바른 인덱스를 활용하지 못하고 35만개의 Document를 검사하고 있었습니다.

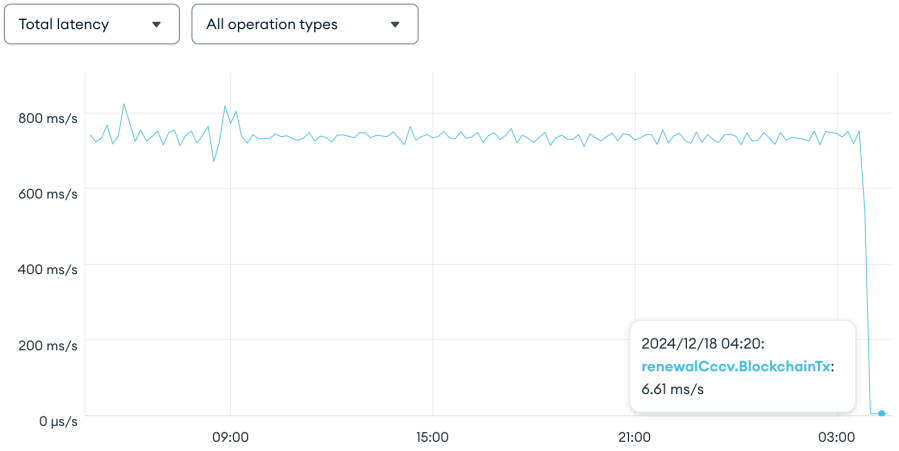
Partial Index는 필터 조건에 해당하는 항목으로만 인덱싱 범위를 제한하는 기능으로, 쿼리 조건이 필터

조건을 포함하지 않을 경우에 인덱스를 사용할 수 없습니다. 문제가 되는 쿼리는 $in 옵션을 사용하지 않아서 필터 조건을 만족하지 못하고 있었습니다. 담당자를 통해 쿼리에 $in 옵션이 필요하지만 쿼리가 잘못된 것을 확인했고 필터 조건을 포함하도록 수정 했습니다. 그 결과 쿼리는 인덱스를 정확히 활용하면서 실행 시간이 크게 단축되었고, MongoDB Cloud 메트릭에서도 Latency가 750ms에서 6ms로 감소하여 99.2% 단축되었습니다.

추가적으로, 이슈 발생 시 실시간으로 알림을 받을 수 있도록 Slack 알림을 적용했습니다. 문제를 파악한 후 가장 놀라운 것은, Alert가 발생했지만 팀원 중 아무도 몰랐다는 사실입니다. Alert가 발생할 때마다 Mail 알람이 오지만, 인지한 사람이 없었습니다. 팀원들과 이슈를 공유하면서 이메일이 메신저 알람에 비해 확인하기 힘들다는 의견을 수렴하고 MongoDB Cloud에서 Slack 알람을 추가했습니다. 이를 통해 동일 문제 발생 시 빠르게 대응할 수 있는 체계를 마련했습니다.

성과

쿼리 최적화로 평균 750ms였던 Latency를 6ms로 99.2% 단축했습니다. Query Targeting 그래프의 이상치를 제거하고, 비효율적 문서 스캔 문제를 해결했습니다. Slack 알림 도입을 통해 쿼리 관련 이슈에 대해 실시간으로 대응 가능하도록 개선했습니다.



2-3. NGINX 무중단 배포 과정의 결함 개선

저희 팀은 NGINX를 사용한 무중단 배포를 도입한 상태였지만 간혹 배포중에 발생한 장애가 문제가 되었습니다. 이슈 공유시에는 NGINX 문제 같다고 전달을 받았습니다. 그리고 최근 저도 환경변수 설정 오류 때문에 배포 중 실행 단계에서 문제가 발생했습니다. 기존에 전달 받은대로 포트포워딩이 아예 되지 않을줄 알았지만 Swagger 새로고침 시 White Label 페이지가 응답했고 API도 일부만 동작하지 않았습니다. NGINX가 아니라 무중단 배포 구현에 문제가 있음을 깨닫고 다음과 같은 개선을 진행했습니다.

작업 세부 내용

가장 먼저 Jenkins pipeline과 배포 스크립트를 검토했습니다. 그 결과로 실패하는 경우를 염두에 두지 않고 jar 파일을 전송할 때 같은 파일명으로 덮어쓰기를 하는 것을 확인했습니다. 그래서 JAR 파일 충돌이 원인이라는 가설을 세우고 검증을 진행했습니다. 배포 스크립트를 수정해서 빌드 직전에 파일명에 포트번호를 더하는 식으로 이름 변경을 해주었습니다. 파일명이 겹치지 않게 조치했더니 배포 실패시에도 문제가 발생하지 않고 서비스가 정상적으로 동작하는 것을 확인했습니다.

Root Cause를 확인하기 위해서 문제를 재현하고 실행 로그를 분석해보니 LaunchedURLClassLoader에서 이미 정의된 패키지를 찾지 못하고 있음을 파악했습니다. 아마도 Class Loader가 런타임에서 클래스를 동적으로 조회할 때 문제가 발생하는 것으로 보였습니다. 실제로 Spring Boot 코드를 살펴보니 definePackageIfNecessary 함수에서 getPackage(packageName)로 패키지를 조회했을 때 null 값을 반환하면 definePackage(className, packageName)로 패키지를 정의하고 있었습니다. 이때, IllegalArgumentException이 반환되면 로그와 동일한 에러가 발생합니다. 그리고 definePackage는 Java 표준 라이브러리 메서드로 이미 정의된 패키지를 재정의 했을 때 IllegalArgumentException을 반환하는 것을 공식문서를 통해 확인했습니다.

하지만 아직 의문이 모두 해결되지는 않았습니다. 왜 getPackage는 기존의 클래스를 못 찾아왔을까요? 배포시에 기존 패키지를 수정하지 않고 신규 패키지만 추가하는 경우도 있는데, 해당 경우에는 문제가 없을지도 궁금했습니다. 그래서 이미 실행중인 jar 파일을 그대로 덮어쓰기해서 테스트를 해본 결과, 동일한 문제가 발생했습니다. 그래서 파일이 덮어씌워질 때마다 iNode 번호가 초기화되지만 실행중인 프로세스의 파일 디스크립터는 그대로 유지된다는 인사이트까지 얻을 수 있었습니다.

성과

배포 스크립트를 수정하여 무중단 배포 중 발생하던 JAR 파일 충돌 문제를 해결했습니다. 그 결과로 배포 실패 시에도 서비스의 안정성을 확보했습니다.