Datadog Live Seoul 2024

12년차 스타트업의 아키텍처 리팩토링 돌아보기

이정민 (Tony)
DevOps Tech Lead
드라마앤컴퍼니 (리멤버)

Agenda

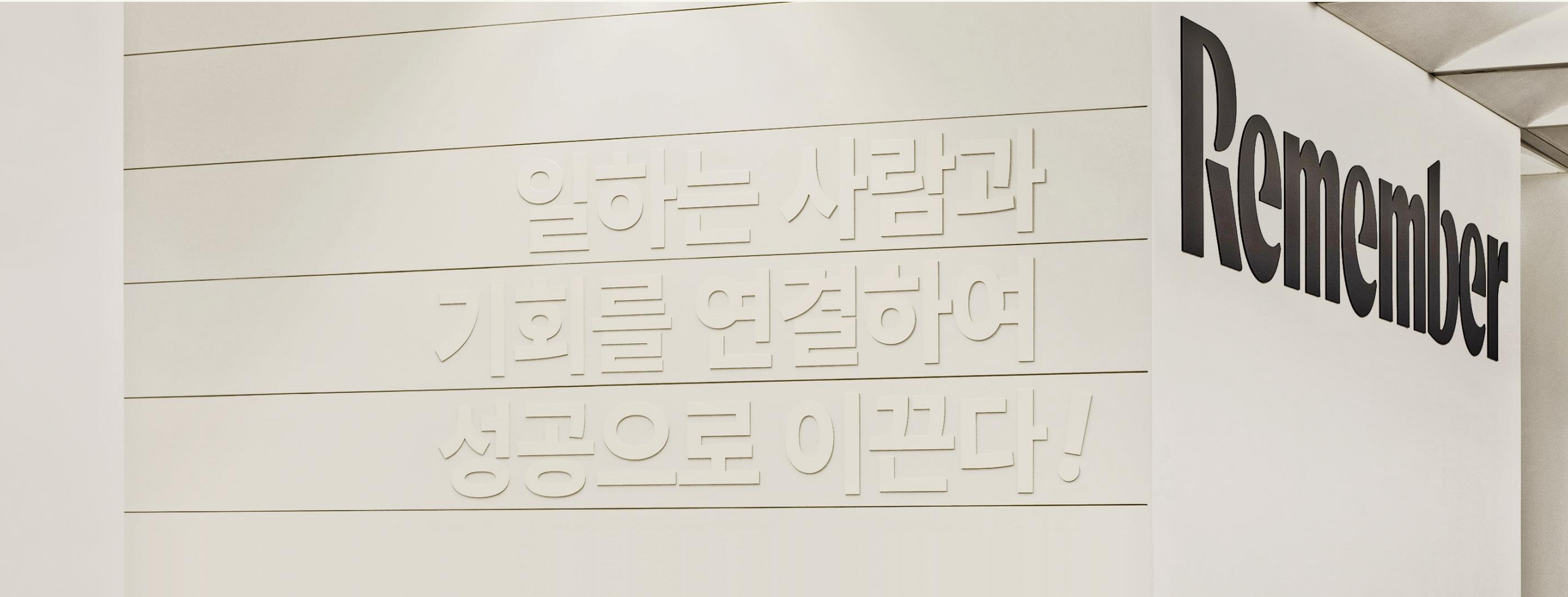
회사 소개

DATADOG

Remember

일하는 사람과 기회를 연결하여 성공으로 이끈다

리멤버는 인맥 관리부터 커리어 성장, 전문가 네트워킹까지 '**프로를 위한 모든 기회**'를
연결하는
프로페셔널 네트워크 서비스(Professional Network Service, PNS)입니다.



누적 가입회원 수

450만 명

누적 스카웃 제안 수

700만 건

Remember Services

B2C Service →
B2B Service ↓

채용공고

명함관리

커뮤니티

프롤로그

채용솔루션

헤드헌팅

리서치 서비스

광고 상품

팀 명함첩

입사 1일차

DATADOG

설레는 첫 출근

- 리멤버의 첫 DevOps 엔지니어
- 출근 첫 날 AWS 계정을 받아보게 되는데..

출처: https://www.irasutoya.com/2017/01/blog-post_657.html

첫 만남은 너무 어려워

AS-IS 현황을 파악해보기

실전 압축된 AWS 계정

모든 워크로드가 Default VPC에 배포되어 있음
CDN을 사용하지 않고 정적 컨텐츠를 S3에서 서빙

부족한 가시성으로 인한 MTTA 증가

호스트 내부, CloudWatch, APM 등 로그가 산발적으로 흩어져 있음
인프라, 클라이언트의 모니터링이 부족함

ISMS 인증 의무 대상

사용자와 매출액 증가로 1년 내 ISMS 인증을 취득해야 함

설렜던 첫 출근

일주일 동안 리팩토링 로드맵을 계획

그리고 쏟아지는 질문들

- AWS 계정을 굳이 나눠야 하나요?
- 서브넷을 나누지 말고 VPC로 구분하면 안되나요?
- IAM은 어떻게 세팅하시려고요?

출처: https://www.irasutoya.com/2018/03/blog-post_56.html

누구나 그럴듯한 계획은 있다

시도해보기 전까지는

확장 가능한 인프라 아키텍처

Provisioning

IaC & GitOps 도입

Prerequisite

- 모든 리소스는 IaC로 관리되어야 함
- 여러 작업자가 작업하더라도 인프라의 형상이 무너지지 않도록 대비해야 함
- GUI로 작업하는 것만큼 쉽게 배포할 수 있어야 함

IaC & GitOps 도입

AS-IS

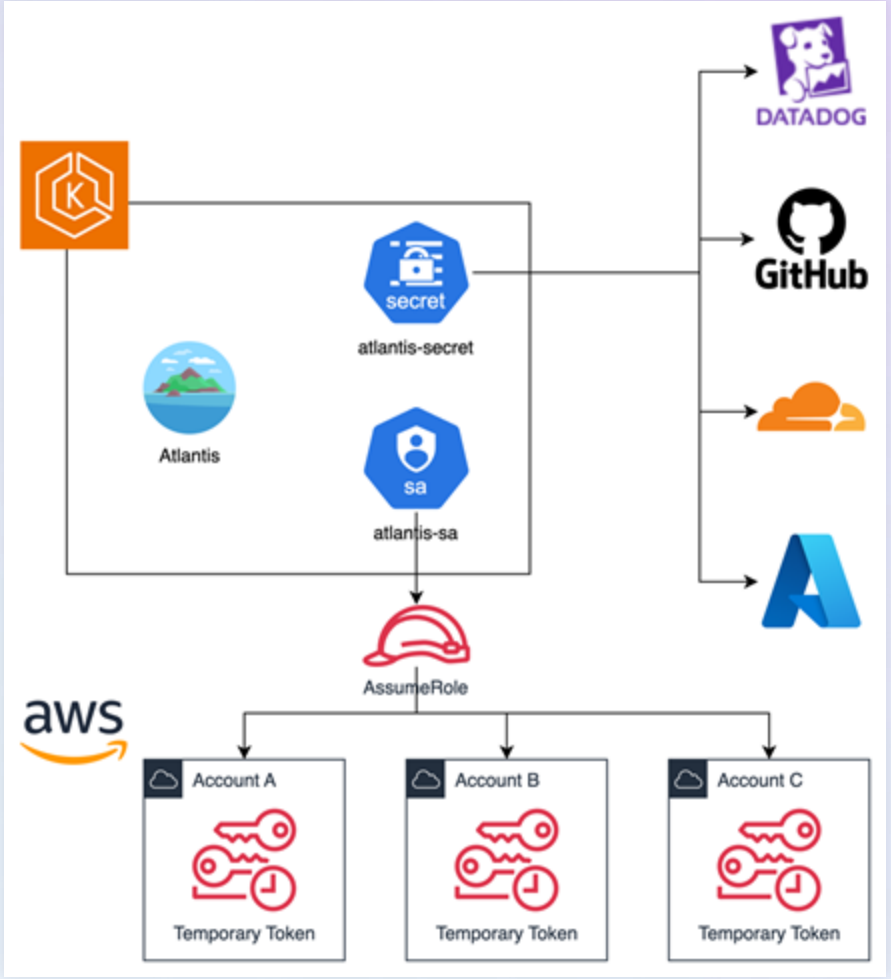
- 소스 코드와 함께 Terraform 리소스를 관리
  - Terraform 코드와 tfstate를 repo에 저장
- 개발자 로컬 환경에서 Terraform 코드를 배포
  - 모든 개발자에게 배포에 필요한 IAM 권한을 부여
- 일부 리소스만 Terraform으로 관리되고 있음
  - ECS Task Definition, Redis 등

IaC & GitOps 도입

- IaC 도구로 Terraform을 선택
  - 개발팀 대부분이 Terraform 사용 경험이 있어 러닝 커브가 적었음
- 배포 중앙화 및 GitOps를 위해 Atlantis를 도입
- 가능한 모든 것을 IaC로 배포
  - AWS, Azure, Github, Datadog, Cloudflare 등
- 주로 사용하는 리소스는 테라폼 모듈로 작성

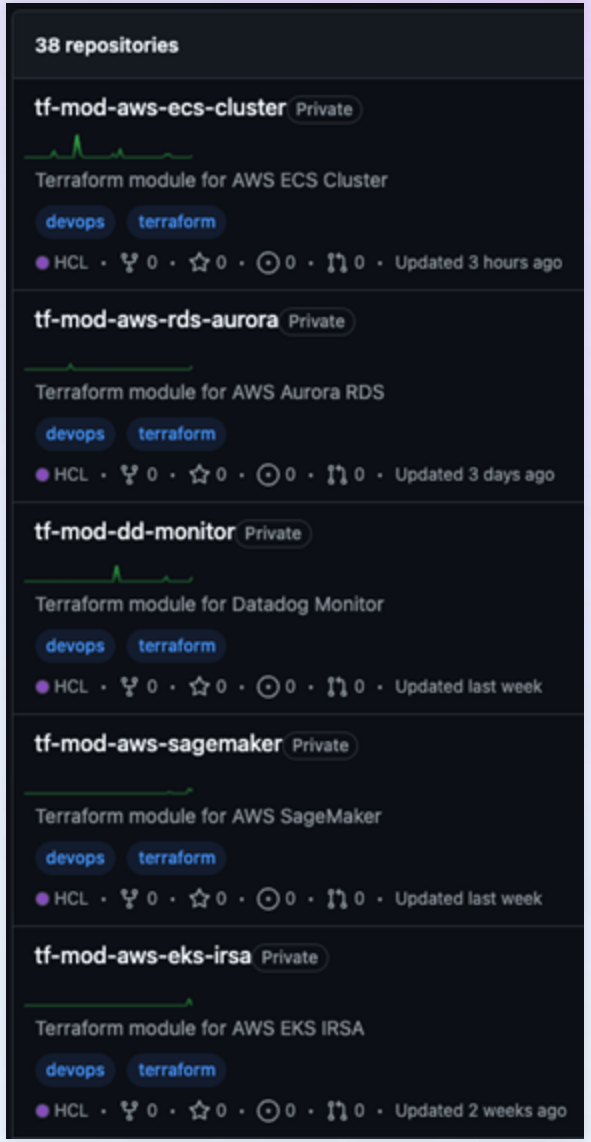
IaC & GitOps 도입

- IaC 도구로 Terraform을 선택
  - 개발팀 대부분이 Terraform 사용 경험이 있어 러닝 커브가 적었음
- 배포 중앙화 및 GitOps를 위해 Atlantis를 도입
- 가능한 모든 것을 IaC로 배포
  - AWS, Azure, Github, Datadog, Cloudflare 등
- 주로 사용하는 리소스는 테라폼 모듈로 작성



IaC & GitOps 도입

- IaC 도구로 Terraform을 선택
  - 개발팀 대부분이 Terraform 사용 경험이 있어 러닝 커브가 적었음
- 배포 중앙화 및 GitOps를 위해 Atlantis를 도입
- 가능한 모든 것을 IaC로 배포
  - AWS, Azure, Github, Datadog, Cloudflare 등
- 주로 사용하는 리소스는 테라폼 모듈로 작성



IaC & GitOps 도입

AS-IS

- 소스 코드와 함께 Terraform 리소스를 관리
  - Terraform 코드와 tfstate를 repo에 저장
- 개발자 로컬 환경에서 Terraform 코드를 배포
  - 모든 개발자에게 배포에 필요한 IAM 권한을 부여
- 일부 리소스만 Terraform으로 관리되고 있음
  - ECS Task Definition, Redis 등

TO-BE

- Terraform 코드를 관리하는 별도의 repo 생성
  - tfstate는 외부 Backend Storage에 저장
  - DynamoDB를 사용하여 State Lock 설정
- Git PR을 통해 Terraform 코드를 배포
  - Atlantis에 배포를 위한 IAM 권한을 부여
  - PR을 통해 리소스를 배포할 수 있는 조건을 제한
- 가능한 많은 리소스를 코드화하여 관리
  - AWS 리소스 **99.x%** 이상 코드화하여 운영

확장 가능한 인프라 아키텍처

AWS Account & Network Architecture

DATADOG

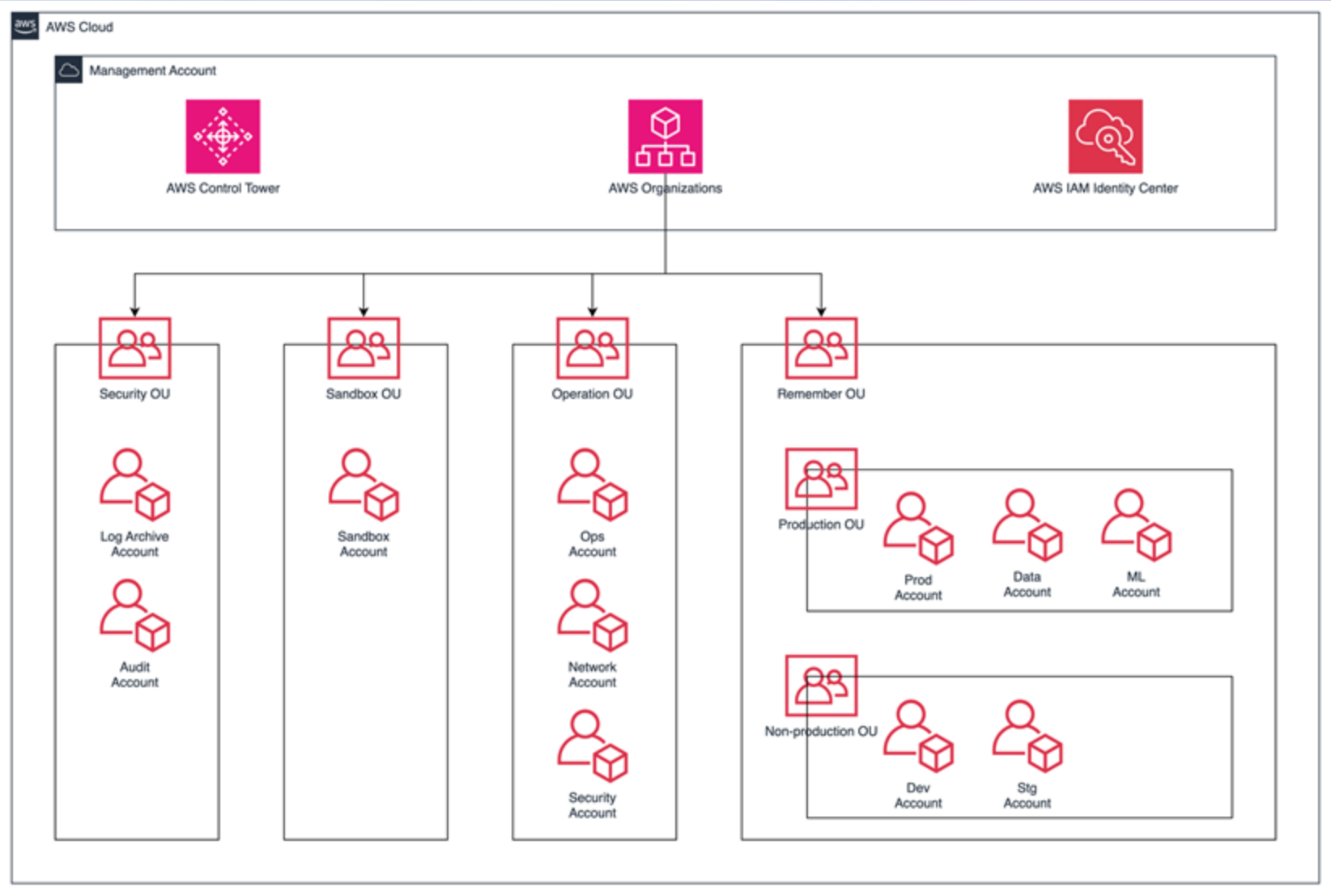
Landing Zone 구성

Prerequisite

- 빠르게 증가하는 서비스 도메인을 고려해야 함
  - 명함, 채용, 광고, 리서치, 커뮤니티, 선물하기
- 다양한 개발 Stage를 분리해야 함
  - 개발, QA, 운영
- 세분화되는 기능 조직 단위에 알맞는 구조를 채택해야 함
  - 애플리케이션, Data, AI/ML, 네트워크, 보안
- Guardrail을 확실하게 설정하되, Guardrail 내에서는 자유롭게 사용할 수 있어야 함
- 비용 효율적으로 구성해야 함

Landing Zone 구성

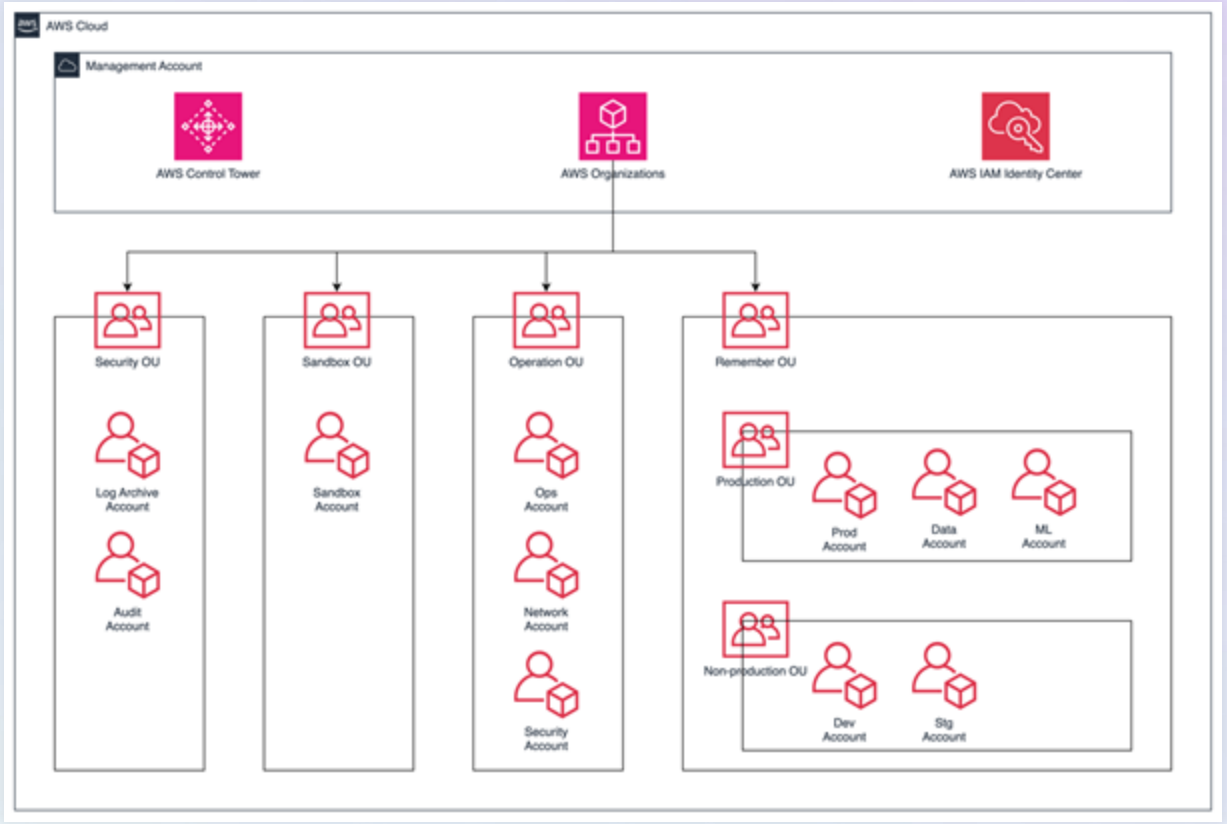
Landing Zone 아키텍처



Landing Zone 구성

AWS Control Tower를 활용

- 도메인 별 계정 분리를 통해 관리 편의성 증가
  - 네트워크 및 데이터 격리
  - 업무 담당자에 한해 접근 권한 부여
- 비용 추적성 증가
  - 도메인, 팀, Stage 단위의 운영 비용 확인 가능
  - 태깅만으로는 비용 추적이 제한적임



리전 마이그레이션

Tokyo to Seoul

- 리멤버 창립: 2013. 07. 03
- AWS 서울 리전 런칭: 2016. 01. 07
- 10년간 사용했으니, 이제는 옮길 때가 되었다
  - AWS 계정을 분리하며 리전을 옮기는 것은 크게 어려운 작업이 아님
  - 한국 사용자가 대다수이기 때문에 서울 리전 사용 시 **사용자 경험이 크게 향상됨**

리전 마이그레이션

Latency Benchmark

- API 서버를 각각 도쿄, 서울에 배포
- 로컬 환경(PC 및 모바일)에서 Latency 체크
- 단일 요청에 대해 서울 리전이 더 빠르게 응답
    - API Call이 중첩되어 발생하기 때문에
      서비스 사용 시 실질적 체감 속도는 더 증가함

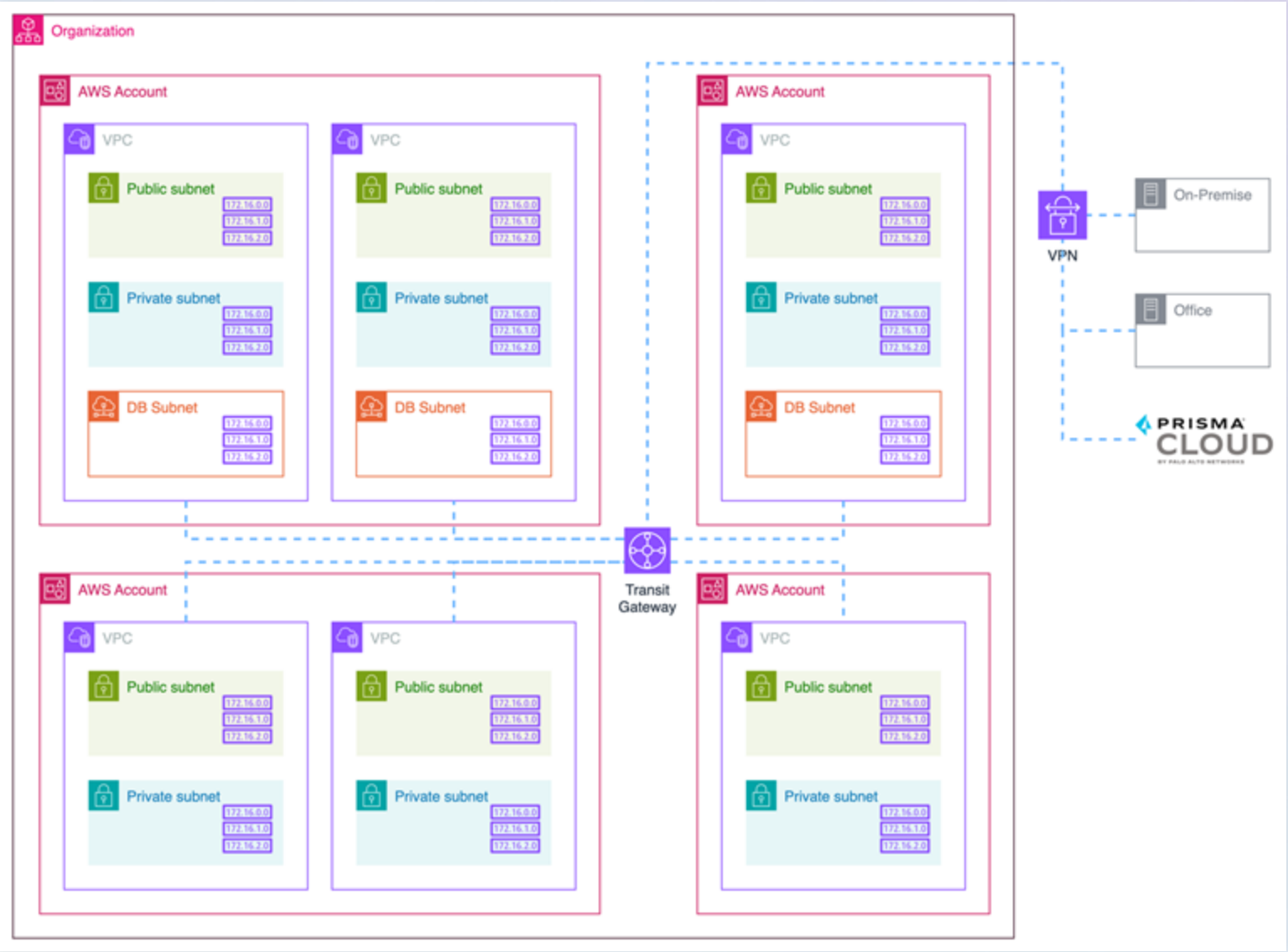
	Mean	Median	Min	Max
Seoul ap-northeast-2	29 ms	29 ms	21 ms	38 ms
Tokyo ap-northeast-1	128 ms	127 ms	120 ms	135 ms
Diff	77.3%	77.1%	82.5%	71.8%

네트워크 아키텍처 재정비

Complexity

- 네트워크 구간이 매우 다양함
  - N개의 사무실, IDC, VPN(SASE) 등
- 복잡한 Route Path, 모든 네트워크가 AWS에 Destination을 가지고 있음
  - 내부 업무 시스템, LDAP 등

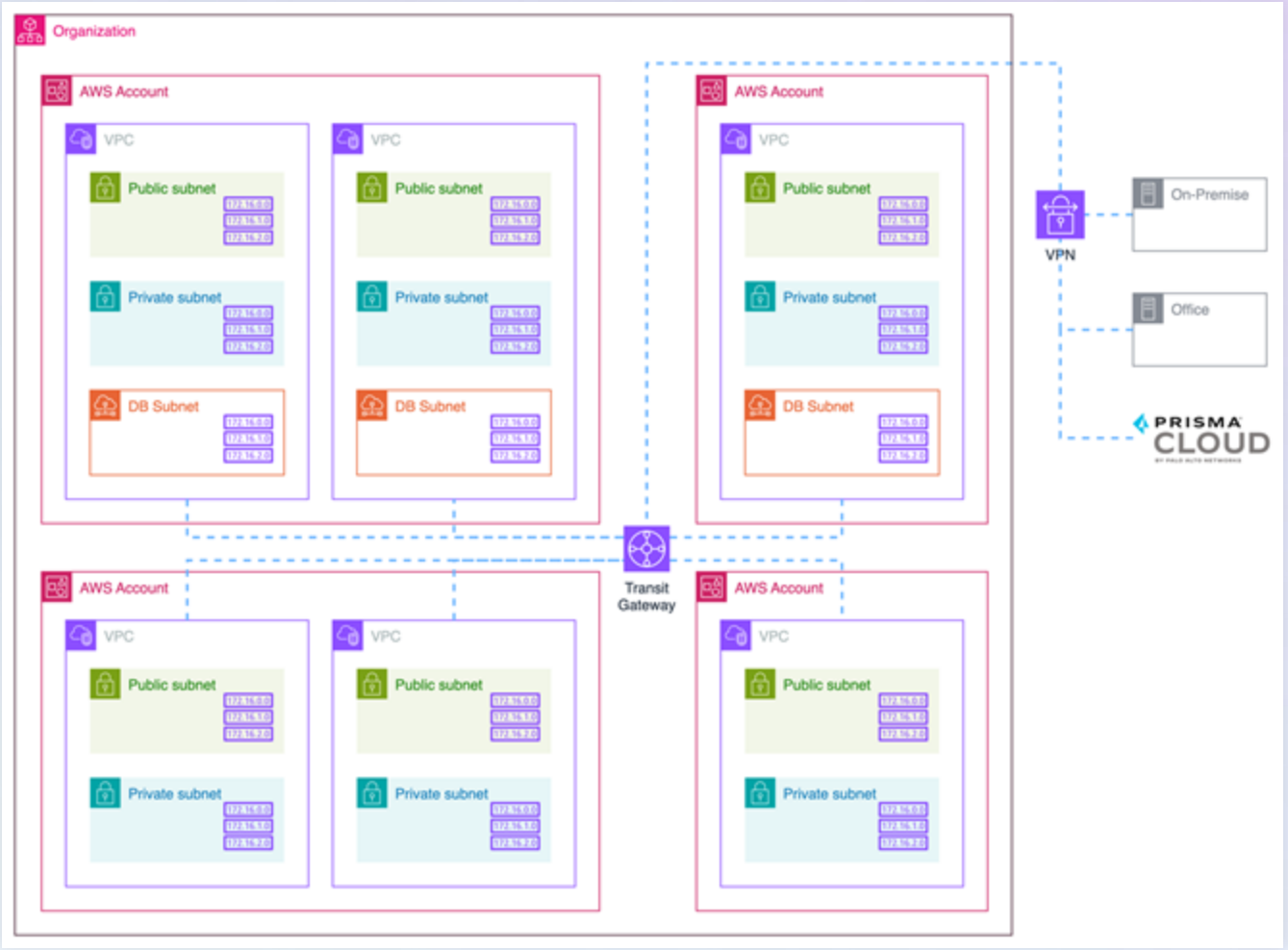
네트워크 아키텍처 재정비



네트워크 아키텍처 재정비

Let’s Hub & Spoke

- VPC에 연결된 구간별 VPN Connection 제거
- 모든 연결을 AWS Transit Gateway로 중앙화
  - VPC to VPC
  - VPN to VPC
  - TGW to TGW (Cross-region)
- TGW Route Table에서 라우팅 정책 관리



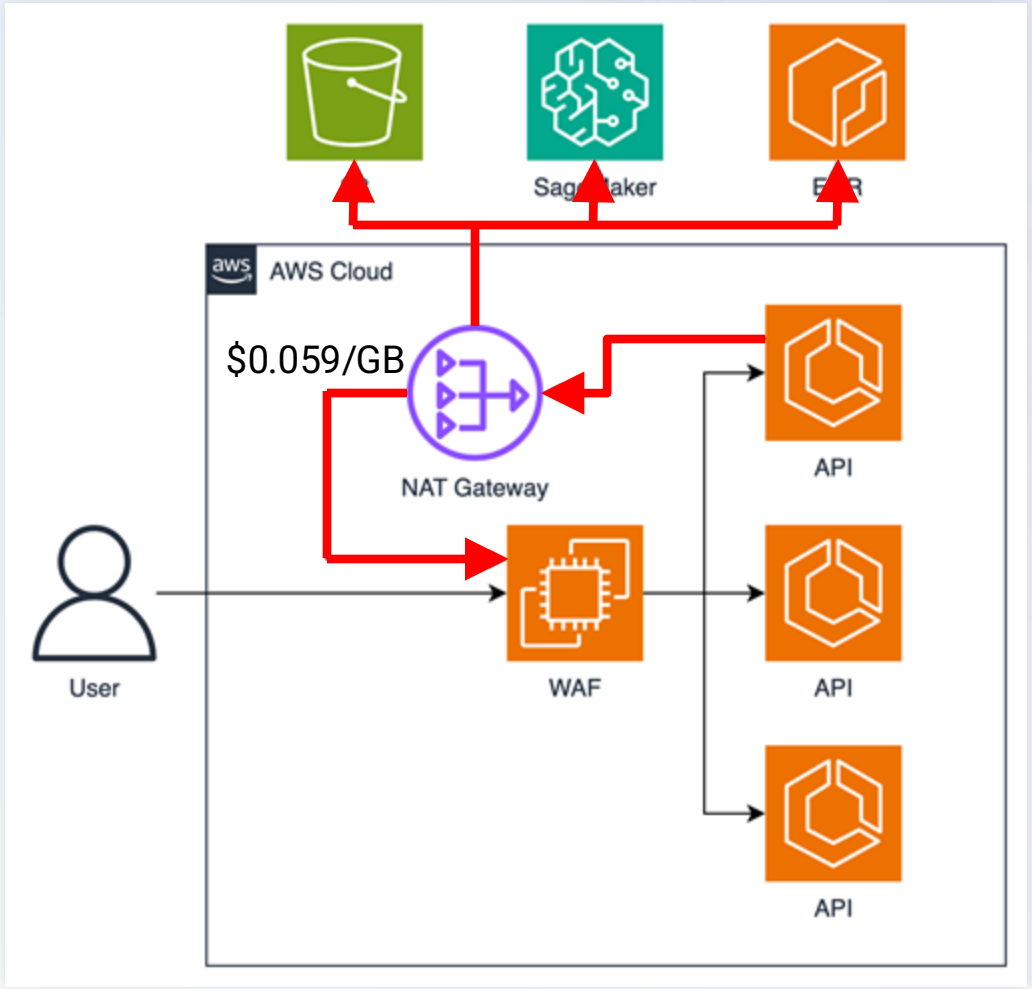
네트워크 아키텍처 재정비

DataTransfer 비용의 정상화

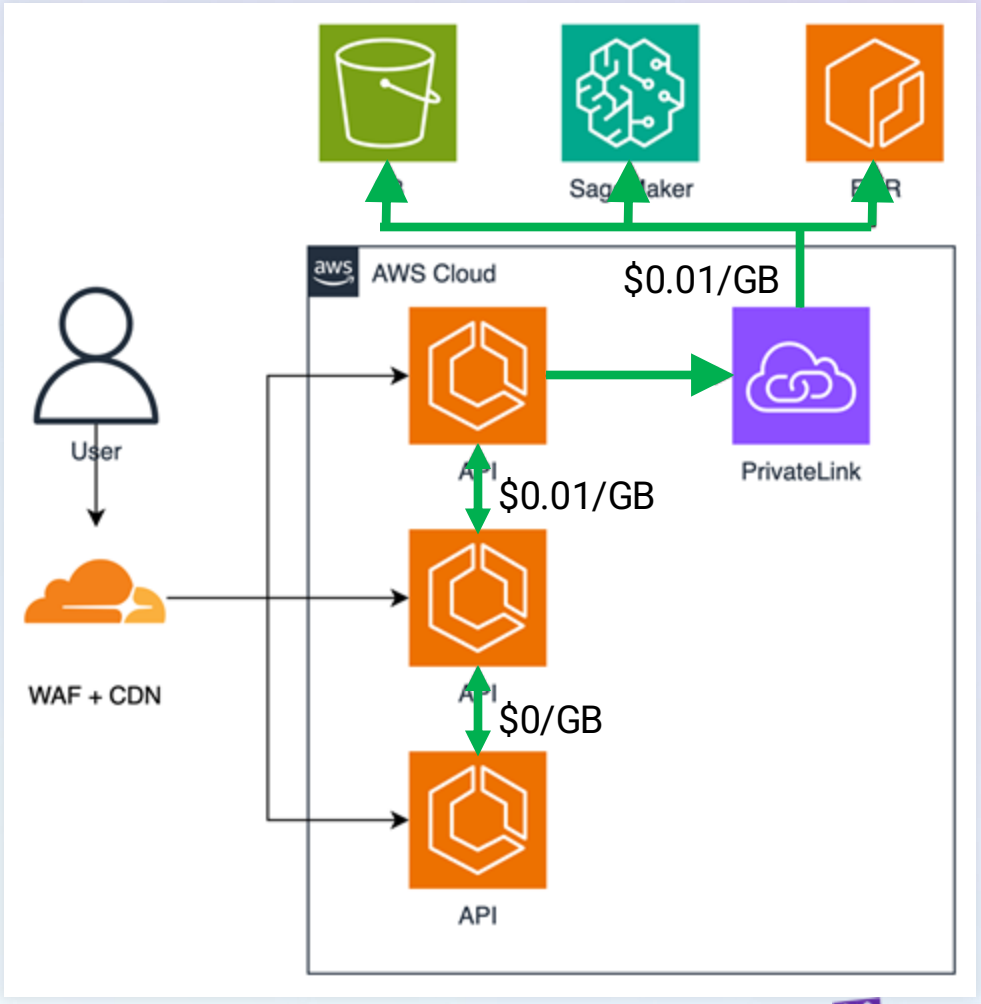
- 트래픽 비용이 비정상적으로 많이 발생함
  - API to API 호출이 Public하게 이루어지면서 NAT Processing 비용이 발생
  - Static Object가 S3에서 바로 서빙되어 캐싱이 이루어지지 않고, S3 API + Outbound DataTransfer 비용이 발생
- 트래픽 비용을 감소시켜보자
  - API 간 Internal 통신하도록 구조 변경
  - 애플리케이션 - AWS 통신 구간 VPC Endpoint로 라우팅 변경
  - 전체 서비스 엔드포인트에 CDN 도입

네트워크 아키텍처 재정비

AS-IS



TO-BE



그 외에도..

Landing Zone 환경을 잘 사용할 수 있도록

- CI/CD 파이프라인 변경 (AWS CodePipeline -> Github)
- 개발 환경 도메인 구조 변경
  - xxx-api-dev.remember.co.kr -> xxx-api.dev.remember.co.kr
  - xxx-api-stg.remember.co.kr -> xxx-api.stg.remember.co.kr
- DNS(Route 53) 이전
- 모든 프로젝트 Containerization

작업은 이런 타임라인으로 진행

Control Tower 배포
2023. 05

TGW 기반 네트워크 구성
2023. 05 ~ 2024. 01

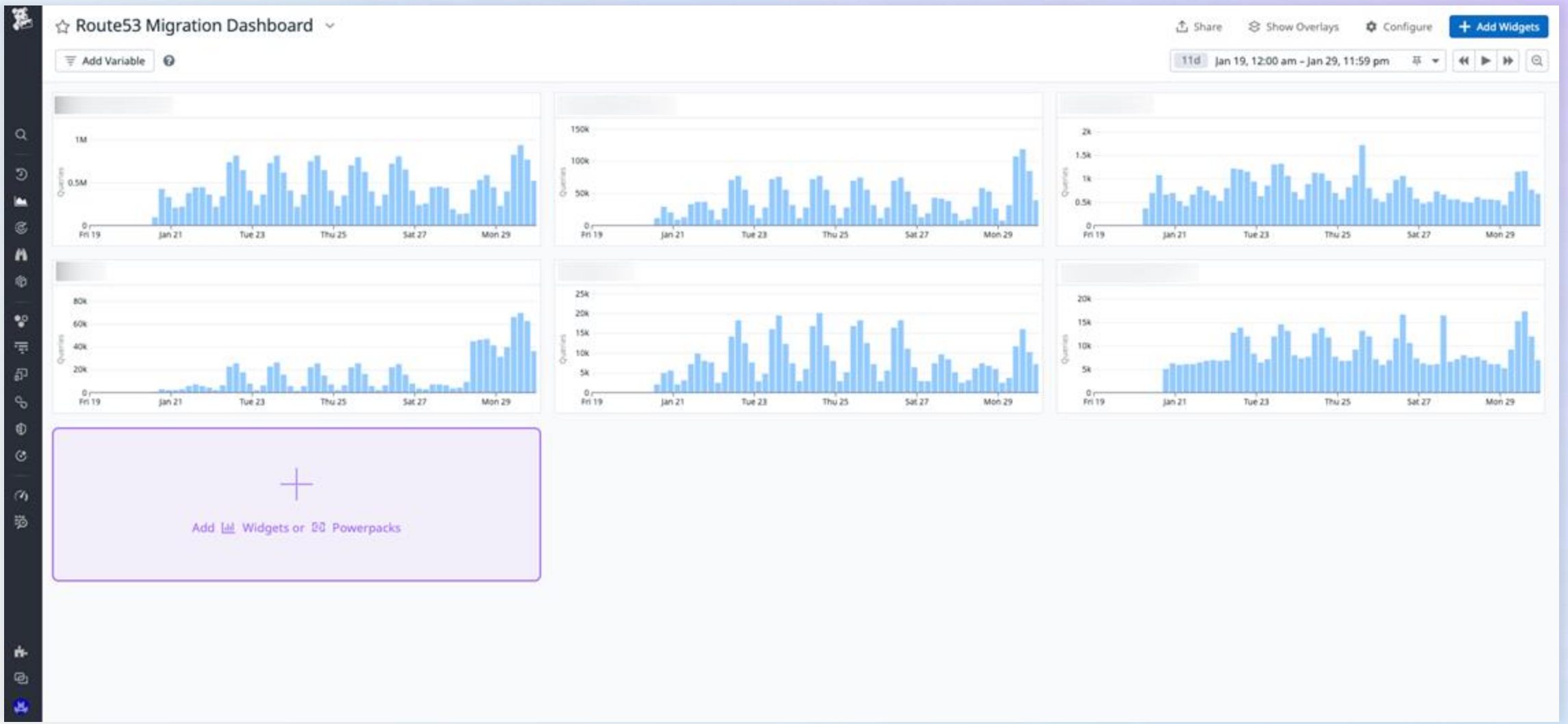
개발, QA 환경 구축
2023. 05 ~ 2023. 11

Data / ML 환경 구축
2023. 12 ~ 2024. 01

운영 환경 마이그레이션
2024. 01

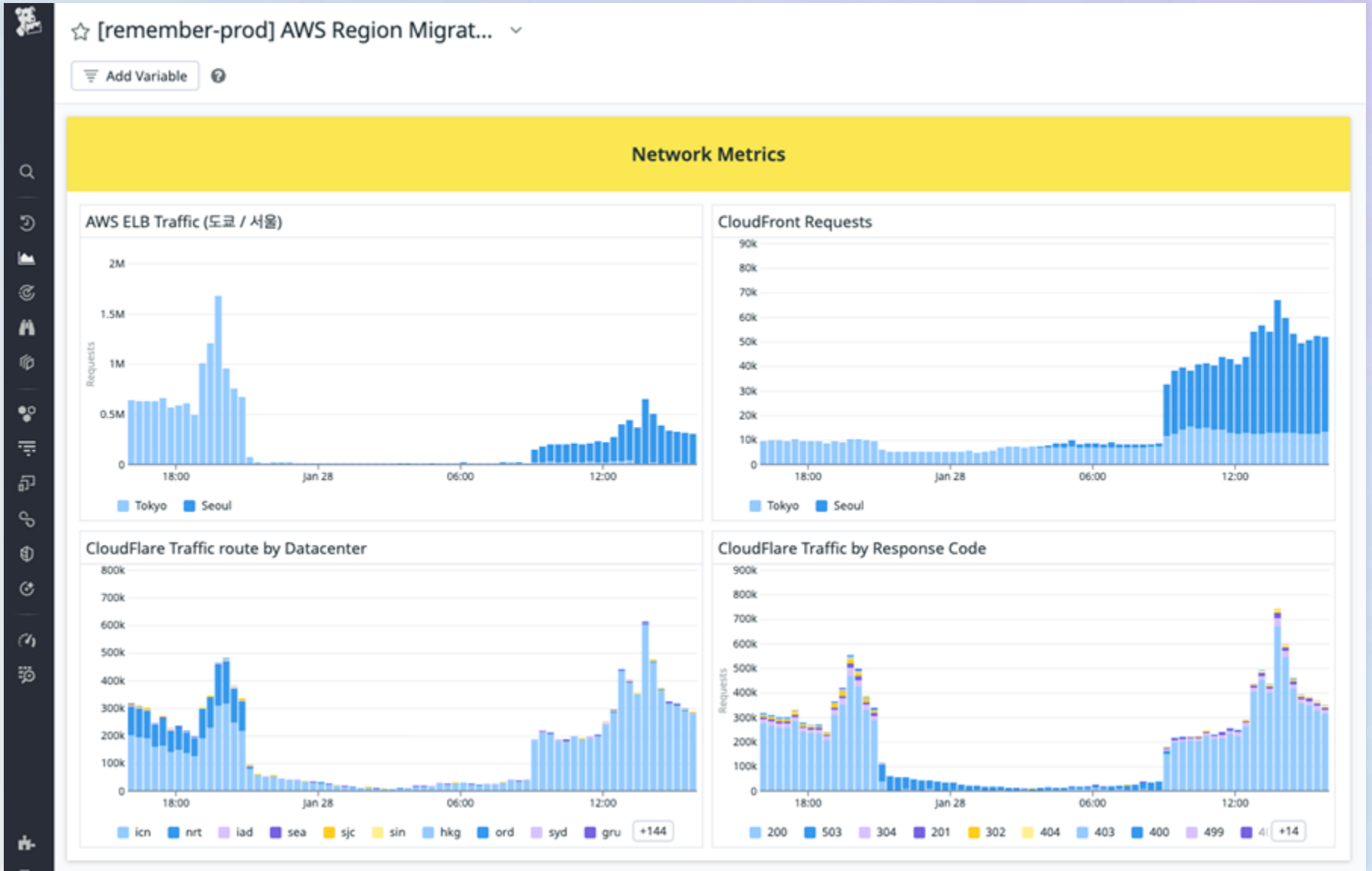
마이그레이션 과정 지표

DNS 마이그레이션



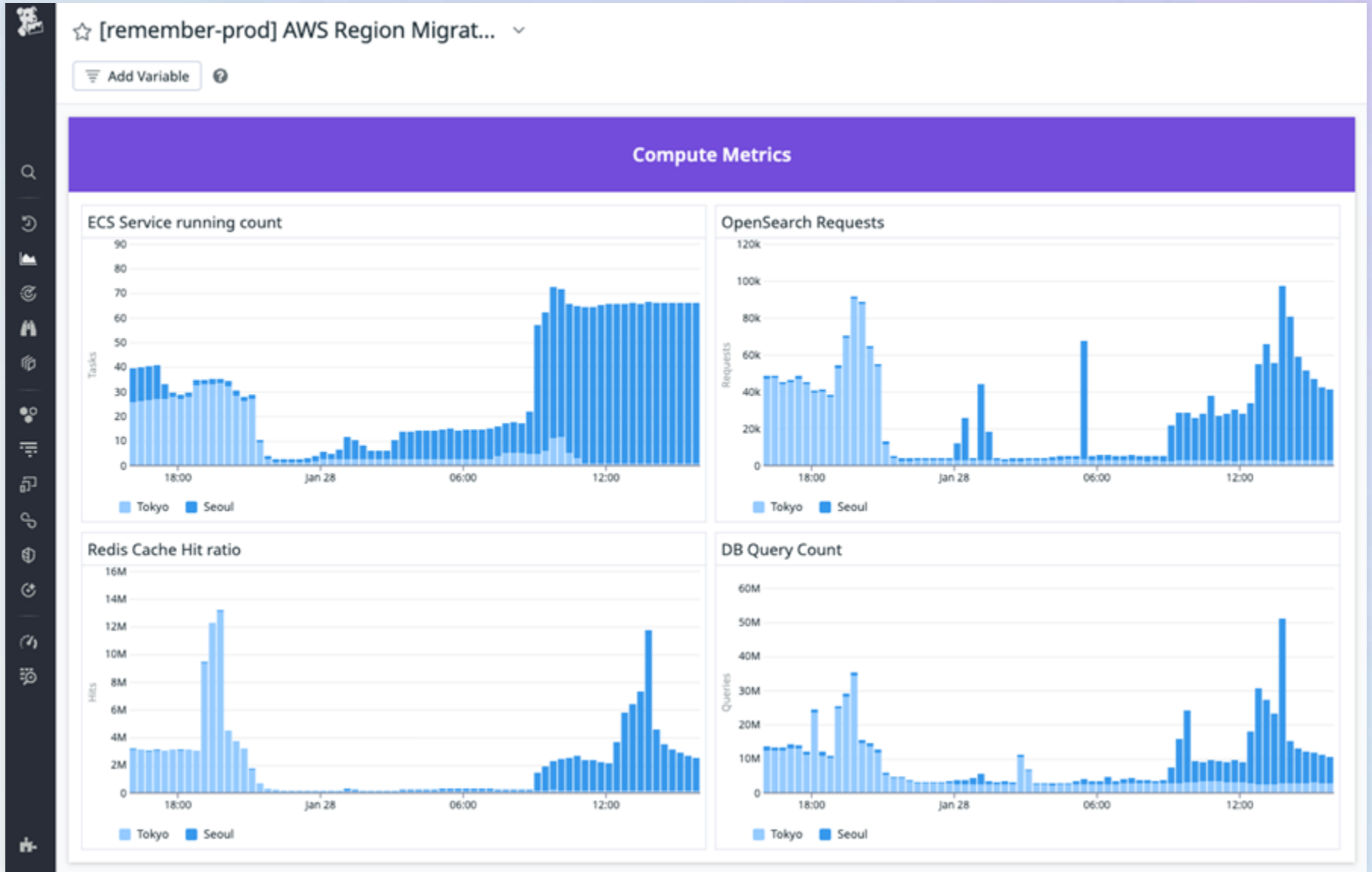
마이그레이션 과정 지표

서비스 중단 후 트래픽 마이그레이션



마이그레이션 과정 지표

서비스 중단 후 트래픽 마이그레이션

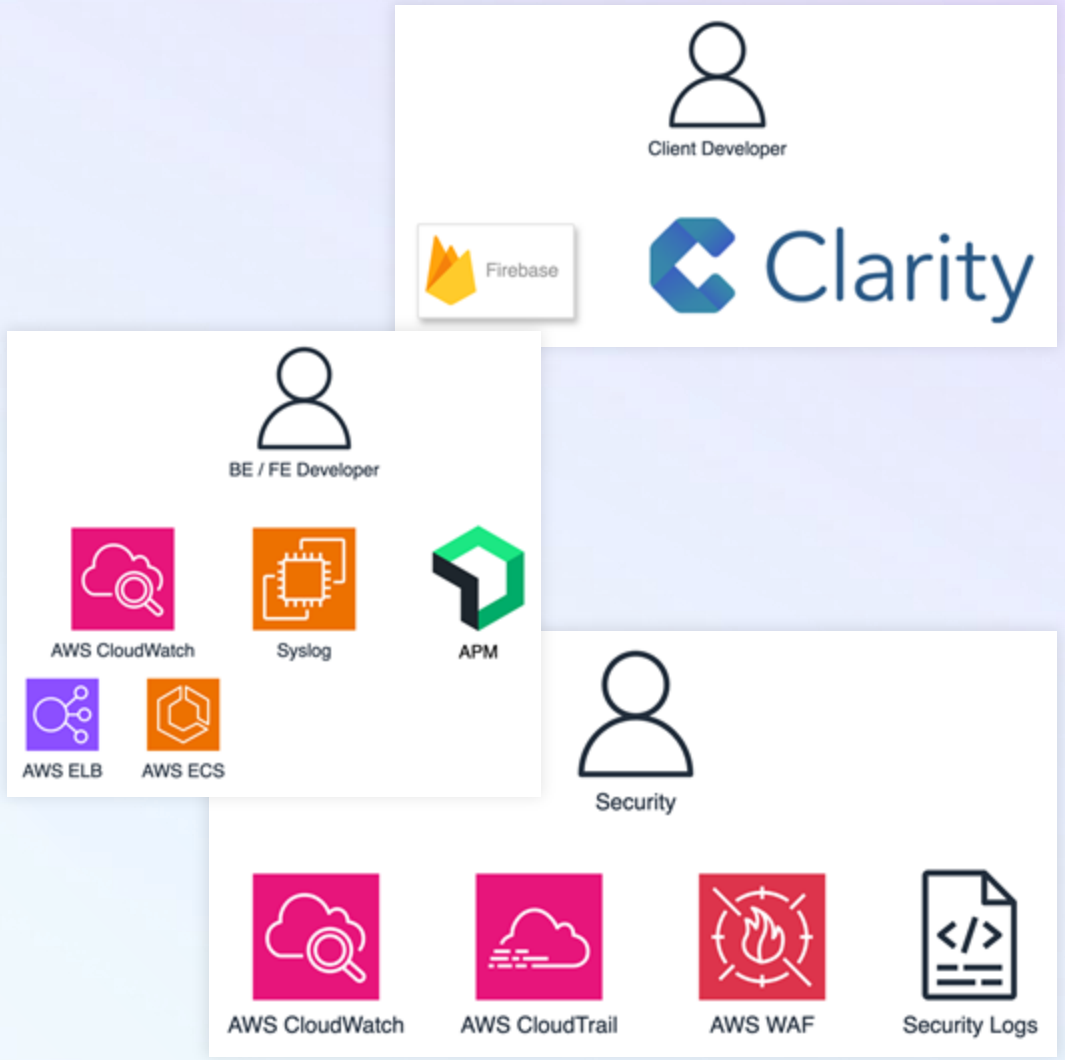


통합 가시성 확보

흩어진 정보

데이터는 많은데 보기가 너무 어려워

- 담당자마다 모니터링을 위해 사용하는 툴이 다름
- 수집되지 않고 있는 로그 혹은 메트릭도 다수 존재



흩어진 정보

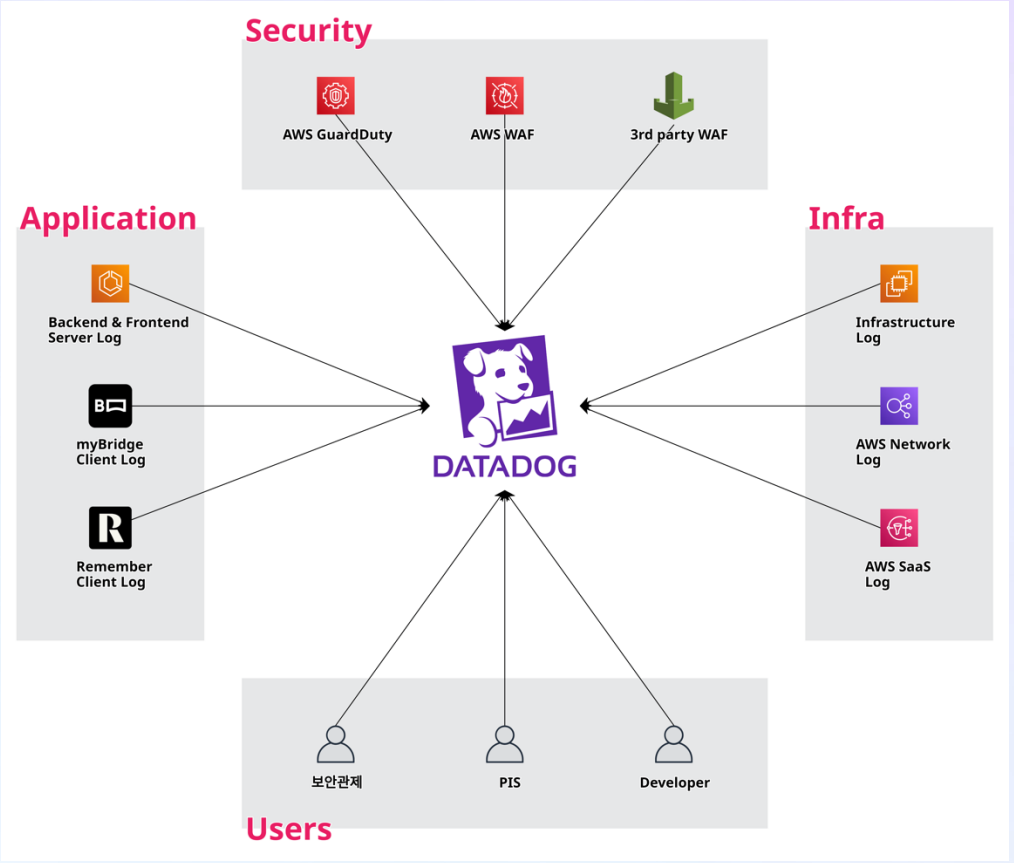
이미 구축된 APM이 있었지만

- 백엔드 위주의 일부 애플리케이션만 연동됨
- 인프라, 네트워크, 보안 지표는 거의 수집되지 않음
- 조직 규모가 크면 비용 부담이 급격히 증가함
- 러닝 커브가 있고 백엔드 개발자 친화적으로 느껴지는 UX
  - UI를 통해 정보를 조회하는 것에 한계가 명확함
  - Query Language를 사용할 줄 안다면 사용성이 급격히 좋아짐
    그 탓인지 백엔드 개발자들은 잘 사용함

흩어진 정보

모두가 같은 화면을 보며 이야기할 수 있도록

- 모든 정보를 단일 창구로 모으는 것이 목표
  - (Integration) Application, Security, Infra
  - (Custom) Service Metrics, Business Metrics
- 누구나 쉽게 사용할 수 있어야 함
  - (Dev) BE, FE, iOS, AOS, DevOps, Security
  - (Non-dev) PM, PO, Marketing, CX
- 디버깅을 넘어 서비스 안정성을 향상시켜야 함
  - Forecast, Anomaly Detection
  - SLO, SLA Management



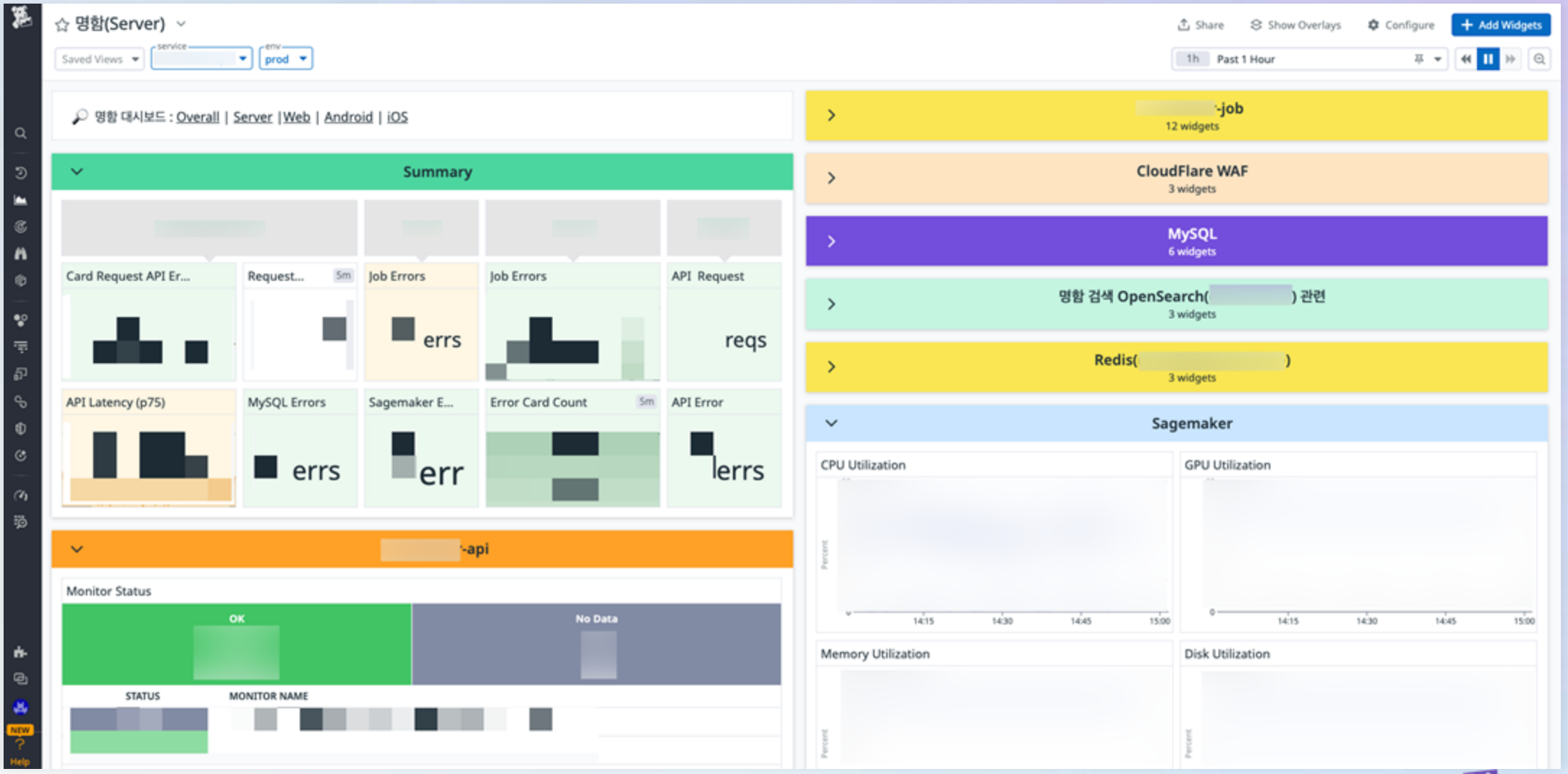
지금은 이렇게 쓰고 있어요

모두가 같은 화면을 보고 있어요

- End-to-End 모니터링 환경 구현
  - 모든 Backend 프로젝트에 APM 연동
  - Frontend, Mobile RUM 및 Session Replay 기록
  - Frontend - Backend - Infra 의 Correlation 설정

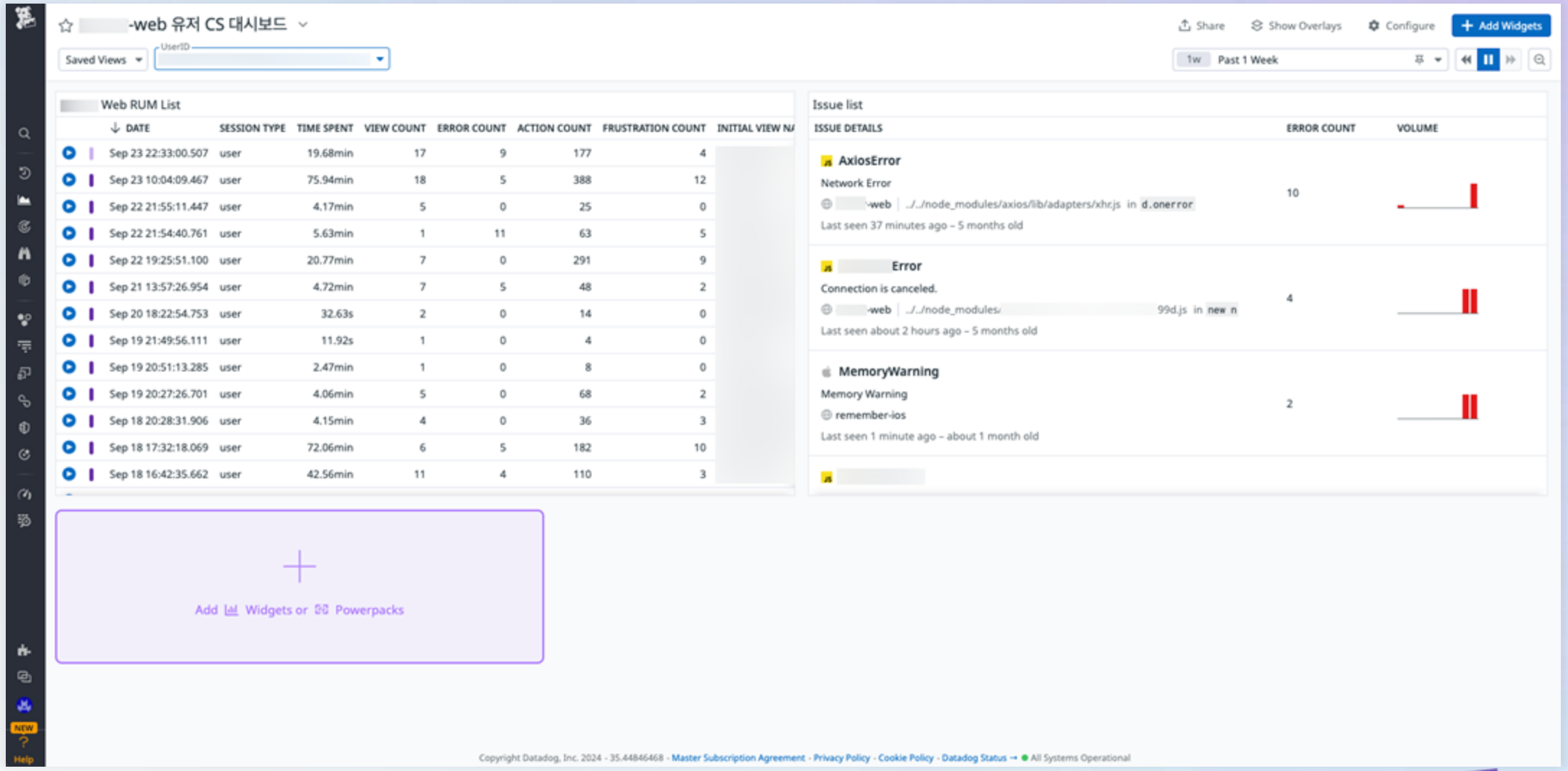
지금은 이렇게 쓰고 있어요

도메인 별 주요 지표 대시보드



지금은 이렇게 쓰고 있어요

CS 대응을 위한 RUM / Session Replay 대시보드



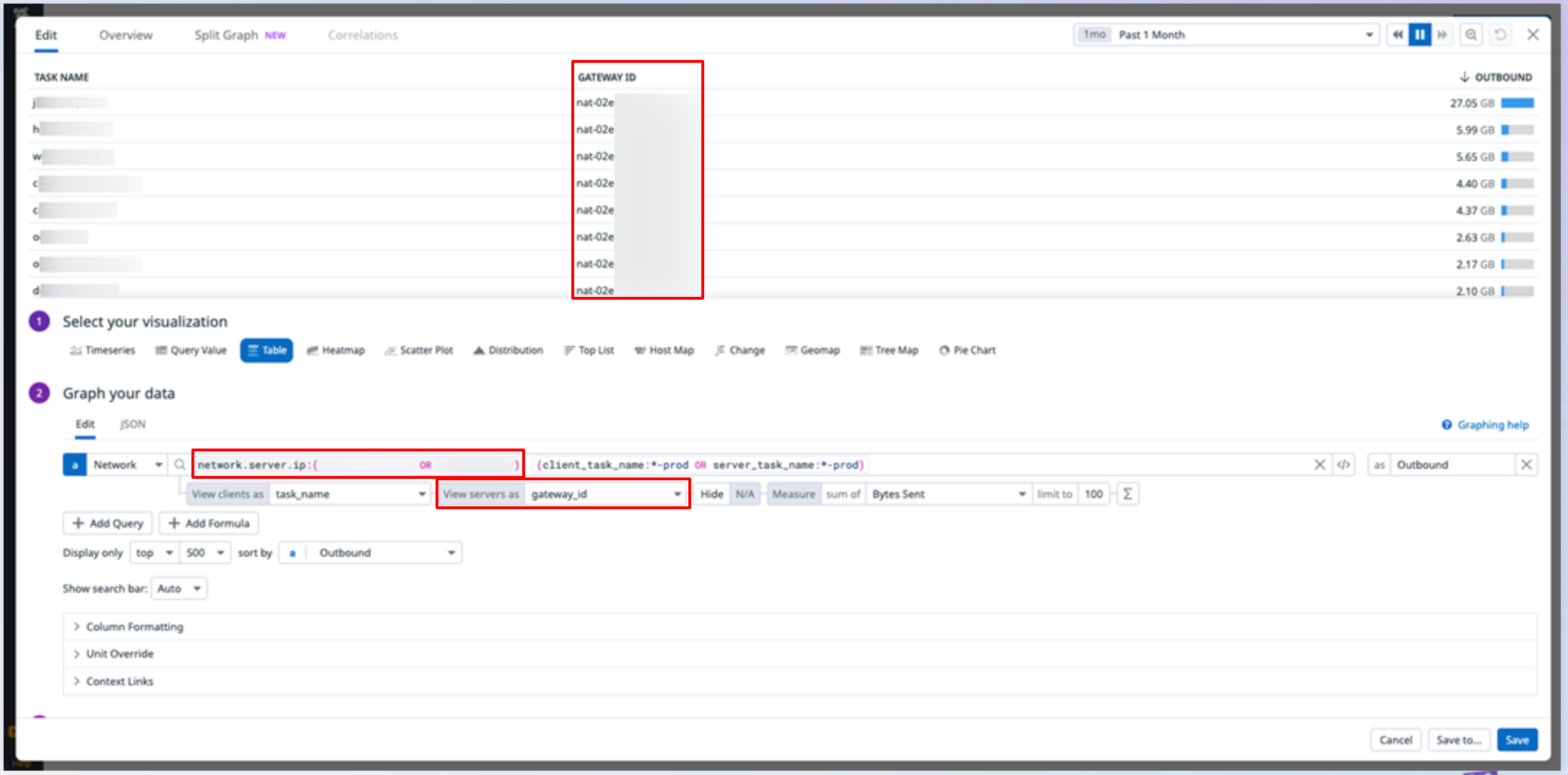
지금은 이렇게 쓰고 있어요

모두가 같은 화면을 보고 있어요

- 인프라 역시 End-to-End 모니터링 환경 구현
  - WAF, CDN, Host, Container, Cache, DB
- Network Performance Monitoring 을 매우 유용하게 사용
  - API 간 통신이 NAT를 경유하는 경우
  - AWS Endpoint(S3, SES, KMS 등) 통신이 NAT를 경유하는 경우

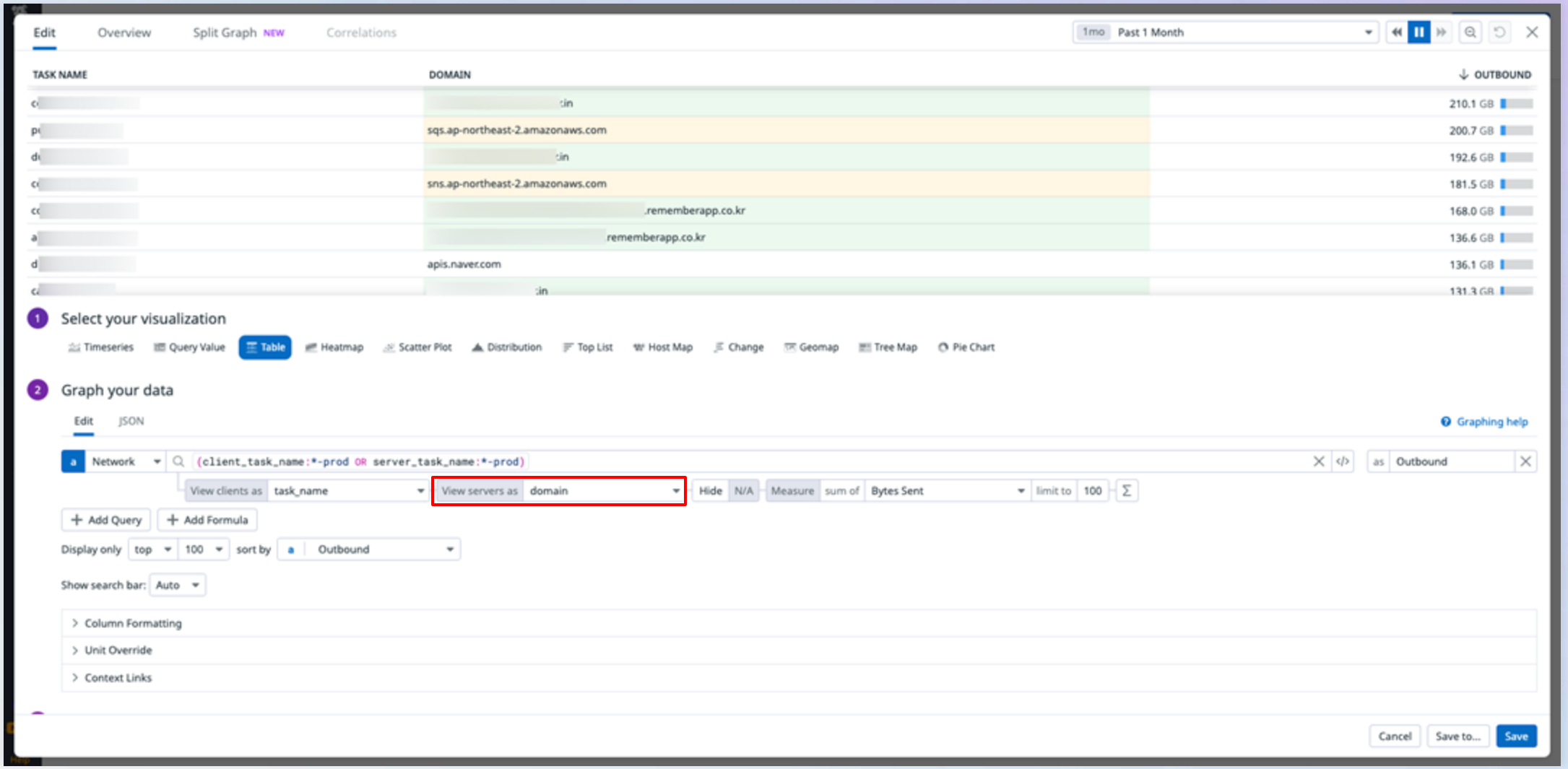
지금은 이렇게 쓰고 있어요

내부 API끼리 공인망(NAT)을 통해 통신하는 케이스



지금은 이렇게 쓰고 있어요

도착지 도메인으로 필터링하여 불필요한 외부 통신 제거



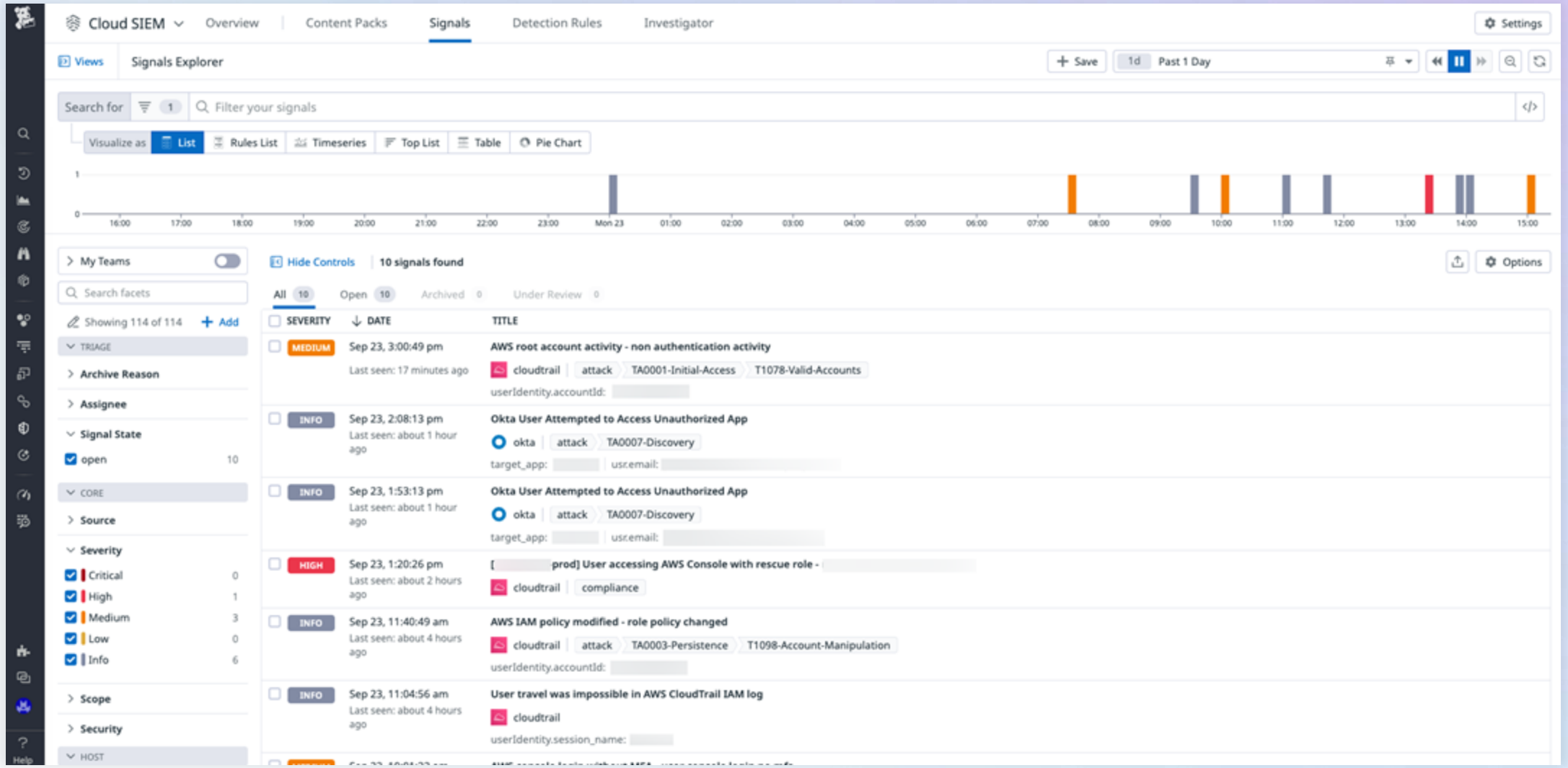
지금은 이렇게 쓰고 있어요

모두가 같은 화면을 보고 있어요

- 마침내 Security Log를 한데 모아서 확인
  - WAF Log
  - Audit(CloudTrail) Log
  - SaaS Logs
- CSM을 사용하여 컴플라이언스 위반 사항 탐지

지금은 이렇게 쓰고 있어요

SIEM에 적재된 로그 기반으로 이상 행위 탐지



지금은 이렇게 쓰고 있어요

모두가 같은 화면을 보고 있어요

- Datadog을 모니터링을 위한 Single Source of Truth로 사용
- 제품에 관여하는 모두가 Datadog을 사용하게 됨
  - PO, PM, 디자이너
  - CS
  - Data Analyst
  - DevOps, Security
- 조금 비싸지만 RUM과 Session Replay는 개발자, 비개발자 모두에게 아주 유용함

보안성 향상

언제나 골치 아픈 IAM

User Side

- 개발자마다 IAM User를 생성하여 권한 부여
- IAM Access Key를 Rotate하지 않음
- 간혹 Access Key가 애플리케이션 코드 혹은 서버에 하드코딩된 경우를 발견
- 같은 업무를 하는 개발자이지만 서로 다른 Role을 가지고 있는 경우
- 개발자가 스스로 본인 IAM User에 권한을 추가/삭제할 수 있음

언제나 골치 아픈 IAM

User Side

- AWS 접근 인원이 50+ 인데, IAM User 관리는 불가능에 가까움
- AWS Account가 늘어나면 IAM User 도 늘어남
- 퇴사, 혹은 조직 이동과 같은 사용자 정보 변경에 민감하게 대응하지 못함

언제나 골치 아픈 IAM

User Side

- AWS IAM Identity Center를 사용
- 사내 SSO 솔루션과 연동하여 계정 관리를 일원화
- SSO 솔루션에 기록된 부서 정보를 기반으로 사용자에게 권한 부여
- Temporary Token을 사용하기 때문에 Key Rotate를 하지 않아도 됨

언제나 골치 아픈 IAM

Application Side

- 애플리케이션을 개발한 담당자에 따라 권한을 부여하는 방식이 다름
  - Host에 IAM Access Key를 하드코딩
  - EC2 Instance Profile
  - ECS Service의 Task Role
  - EKS IAM Roles for Service Accounts

언제나 골치 아픈 IAM

Application Side

- Containerized된 애플리케이션에는 ECS Execution Role 혹은 EKS IRSA를 사용
- EC2 Instance에는 인프라 운영에 필요한 필수 Role만 부여
- 외부 어플리케이션은 OIDC로 인증 후 Role 부여

이제는 걱정 없는 IAM

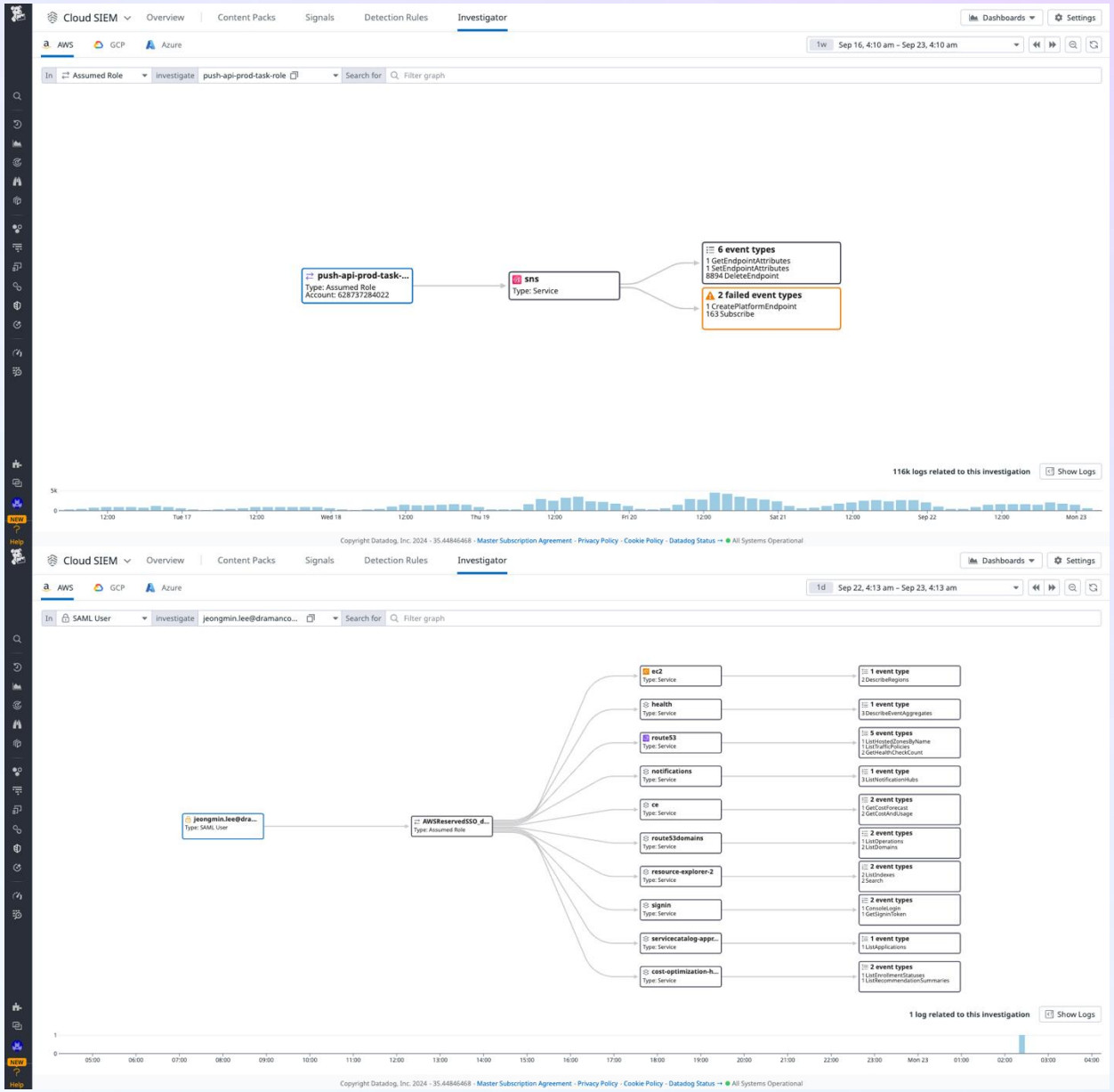
Conclusion

- 현재는 IAM User를 전혀 생성하지 않음
  - 유저 별 MFA를 관리하지 않아도 되며
  - Access Key Rotation 또한 신경쓰지 않아도 됨
- IAM Permission에 * 혹은 FullAccess Policy를 사용하지 않음
- 미사용 Permission은 주기적으로 확인 후 제거

언제나 골치 아픈 IAM

Conclusion

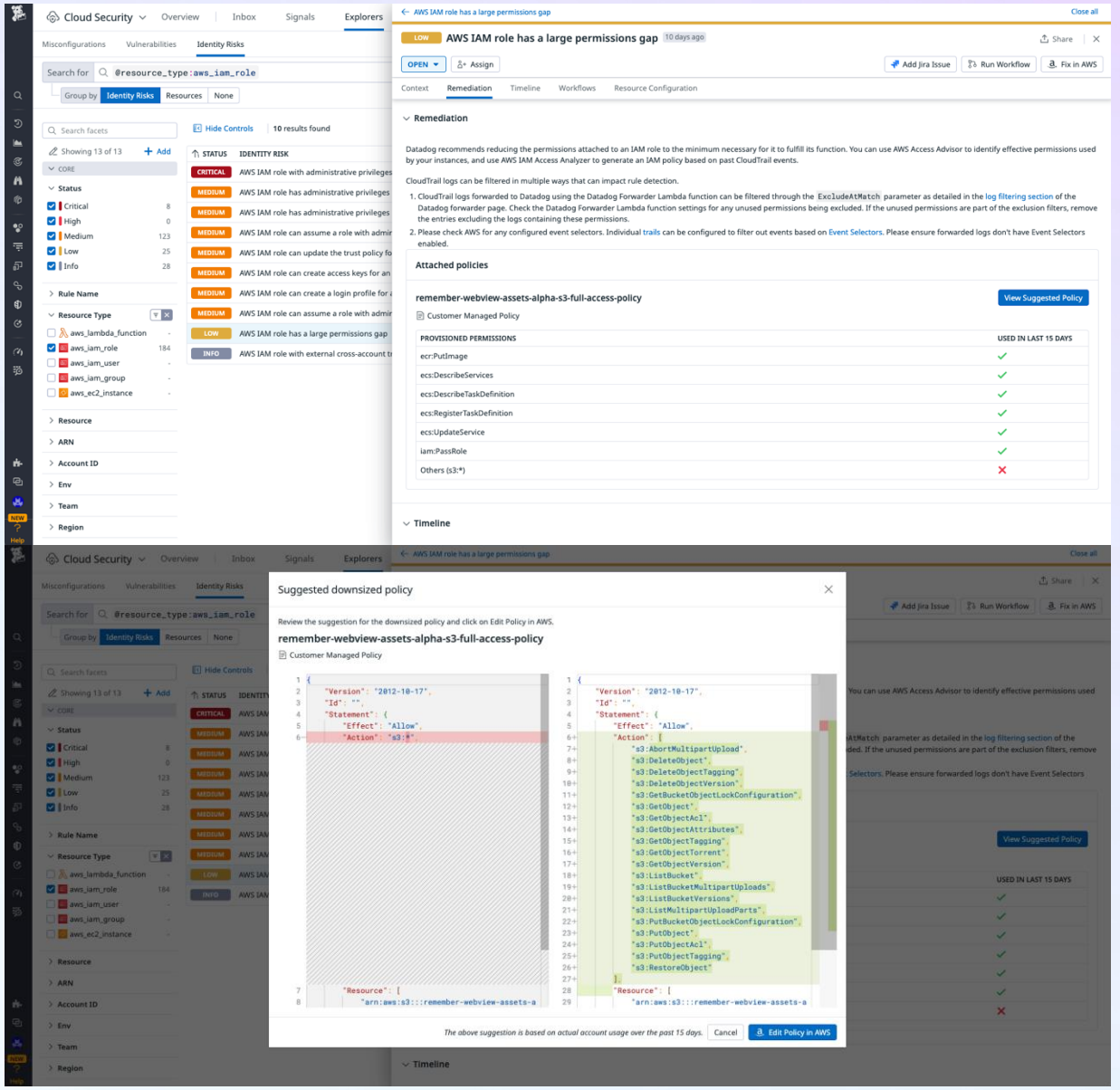
- SIEM Investigator로 사용자 행동 확인
  - IAM User / Access Key
  - Assume Role
  - SAML User



언제나 골치 아픈 IAM

Conclusion

- CSM이 미사용 Permission을 제거하도록 제안
  - AWS IAM role has a large permission gap
  - 미사용 Permission을 식별하는 것은 매우 힘든 일



회고

그래서 지금 운영환경은?

1 → 16
AWS Account 개수

100+ → 0
AWS IAM User 개수

₩100M 절감
네트워크 트래픽 비용

30% | 77% 감소
Datadog 도입 이후
MTTA, MTTR

99.9% 이상
리멤버 SLI

∞
개발 생산성 증가

만약에 다시 한번 작업한다면

Conclusion

- 서비스 무중단 마이그레이션을 해보고 싶음
  - 데이터 마이그레이션(특히 S3 Object Replication)에 대한 확신이 없었음
- 마이그레이션 진행 상황을 확인할 수 있는 대시보드를 더 자세히 제작
  - 작업 당시 모니터링 툴이 여러개여서 Context Switching이 잦았음
  - 서비스 오픈 이후 APM이 연동되어 있었다면 인지할 수 있는 작은 장애들을 겪음

그래도 잘 했다고 생각하는 점

Conclusion

- 훌륭한 동료들의 도움 덕분에 무사히 마이그레이션을 완료함
- 애플리케이션 리팩토링을 위한 기초 공사가 끝남
- Datadog을 적절한 시기에 도입한 것
  - 개발자들이 커뮤니케이션을 위해 대시보드를 제작함
  - 데이터독이 없었다면 마이그레이션 과정이 매우 험난했을 것

- Tip: Datadog과 커뮤니케이션을 열심히&긴밀히 하자
  - Infra, APM, Security 모든 분야의 전문가
  - Integration, 대시보드 제작 등 초반 세팅에 매우 많은 도움을 주심

앞으로 더 개선하고 싶은 것

Conclusion

- ECS - EKS 마이그레이션
  - 프로젝트 규모가 빠르게 커지고 있음 (70+ 서비스)
  - Datadog의 인프라 제품군 또한 k8s 환경에 더 많은 정보를 제공
  - 핵심 워크로드를 제외하면 모두 EKS를 사용하고 있음
- 모니터링 고도화
  - Custom Metrics를 이용한 서비스 지표 수집
  - Workflow를 이용하여 SRV4~5 등급 장애 Runbook 자동화

Thank you

DATADOG