

Part 8

Devops

시스템 운영을 위한 방법론인 Devops에 대해서 알아본다.
시스템 모니터링, 로깅뿐만 아니라 운영팀을 운영하기 위한
문화와 Devops팀이 어떤 일을 하는지, 그리고 시스템을
모니터링하기 위한 지표 정의 방법과 이를 구현하기 위한
도구들을 알아본다.



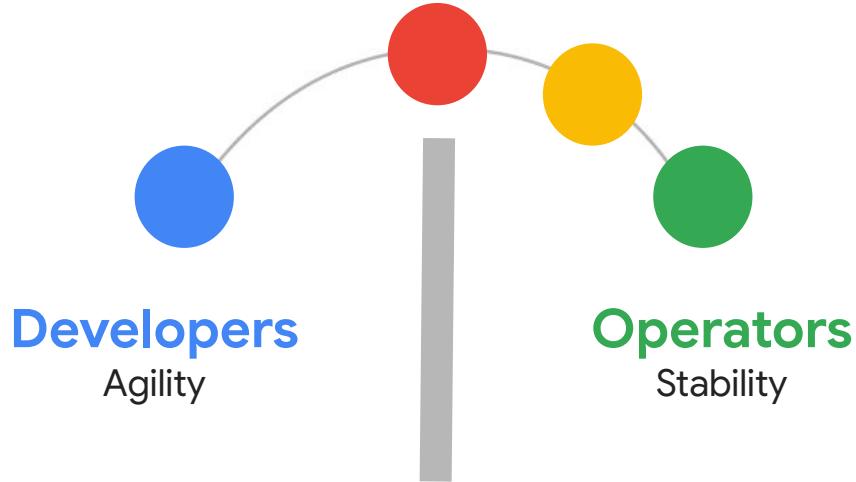
Devops

시스템을 운영하기 위한 프로세스와,
모니터링 지표 정의 방법 및 필요한 도구에
대해서 알아본다.

Devops란?

Devops란 무엇인가?

DevOps는 소프트웨어 개발과 시스템 운영을 결합하여 시스템 개발 주기를 단축하고, 비즈니스 목표에 밀접하게 맞추어 기능, 수정, 업데이트를 자주 제공하는 소프트웨어 개발 프렉티스의 집합이다. (출처: 위키백과)



Devops에 대해서 조사할때..

Devops를 어떻게 하고 계세요?

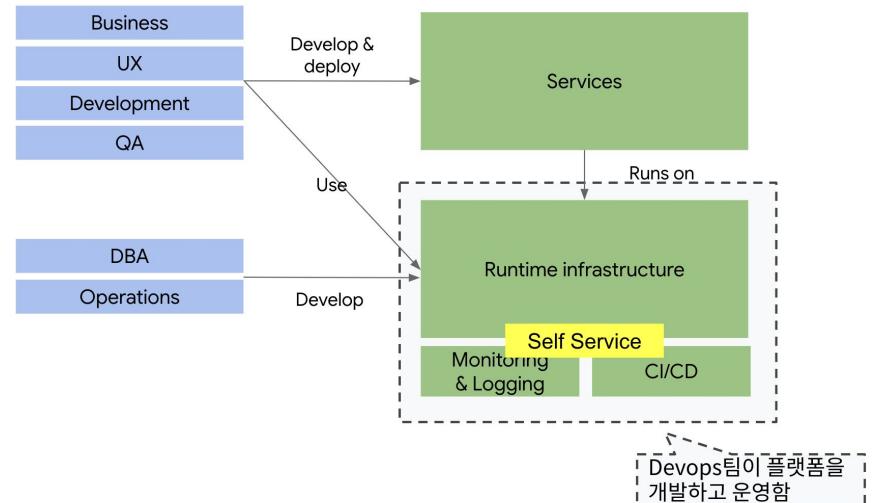
- 개발팀과 운영팀이 있는데,
Devops팀이 추가됨.
- 개발팀과 Devops팀이 있음.
운영팀이 Devops팀으로 이름만 바꿈

Devops는 어떻게 하는 걸까?



그러면 Devops는 어떻게 할까?

- 개발팀이 하드웨어부터 네트워크까지 모두 셋업할 수 없다.
- 마치 클라우드 서비스처럼, 누군가가 기본적인 플랫폼 (런타임, DB, 미들웨어, 네트워크, CI/CD, 모니터링 도구등)을 제공하고 운영해줘야 한다.
- Devops팀은 개발팀이 스스로 개발 및 운영을 할 수 있는 플랫폼 환경을 제공한다.



Devops팀의 기본 문화

바람직한 Devops팀이 되기 위해서 갖추어야 하는 자세

데이터 기반의 의사 결정 Data based decision

모든 의사결정은 데이터를
기반으로 한다.

고객 지향적 User centric

고객이 서비스가 느리다면,
모니터링 시스템의 지표가
정상이라도 느린것이다. 모니터링
시스템을 의심해봐야 한다.

비난하지 않고, 책임을 공유하는 Blameless & shared responsibility

개발 운영뿐만 아니라 비즈니스 조직을 포함하여 조직
전반에 걸쳐 책임감을 공유하고, 실수에 대해서
비난하지 않는 문화.
장애는 운영상 사람의 실수가 아니라, 실수를 하게한
프로세스에 문제가 있다.

Devops 팀이 하는일?

메트릭 정의 및 모니터링



- SLO
- Dashboard
- Analytics

용량 관리



- Forecasting
- Demand-driven
- Performance

변화 관리



- Release process
- Consulting design
- Automations

장애 대응



- Oncall
- Incident analysis
- Postmortems

조직 문화



- Toil management
- Blamelessness
- Share responsibility

모니터링과 알람

모든것을 수치화하여 모니터링

- Devops에서는 모든 결정은 수치화된 데이터를 기반으로 한다.
- 옳은 의사결정을 할 수 있는 유의미한 메트릭을 정하는 것이 중요함
- SLI,SLO>Error budget, Toil

메트릭은 자동으로 수집

모니터링 메트릭은 자동으로 수집되어야 하며, 필요한 사람이 필요한 시점에 다시 보드를 통해서 쉽게 보고 해석할 수 있어야 한다.

알람 : 장애나 특정 이벤트에 대한 알람

- 알람 : 사람의 즉시 개입이 필요할 경우, 즉시 알람을 할 수 있어야 함.
- 티켓 : 사람의 개입이 필요하지만 당장 필요하지 않은 경우 자동으로 티켓 생성

용량 관리 - 수요 예측

자연 증가에 대한 예측

사용자가 늘어남에 따라서
자연적으로 증가되는 필요
용량에 대한 예측 및 대응

이벤트성 용량 증가 예측

마케팅 캠페인이나, 새로운 기능
배포 등 특정 이벤트로 인하여
급격하게 증가될 수 있는 용량
예측



수요 예측에 대한 계약 모델

클라우드의 경우 Commit 계약을 제공함

- 1년, 3년 단위로 사용 금액을 약정하면 할인을 제공
 - 실사용량 보다 많이 약정하면, 돈이 낭비됨
 - 실사용량 보다 적게 약정하면, 그만큼 할인 기회를 잃게 됨
- 보통 예측치의 80%를 약정
- CPU나 하드웨어 성능이 변화됨을 감안해야 함

클라우드 사용 금액 예측 (분기별)



용량 관리 - 비용 관리

기존의 있는 인프라에 대한 사용률을 높이고
비용을 절감 - Finops

신규로 필요한 용량을 추가 확충하는 것도
중요하지만, 기존에 있는 인프라를 최적화하여,
사용률을 최적화하는 것도 중요함.

- 소프트웨어 최적화
- 불필요한 인스턴스, 스토리지 삭제
- 인프라 변경 (인텔 → AMD)



변화관리

70%¹ 의 장애는 시스템 변경(업데이트,보수) 과정에서 일어남.

장애를 줄이는 방법

- 단계적 배포 방식 사용
(Canary, Rolling update)
- 빠르게 비정상 상황을
잡아낼 수 있는 구조 구현
(모니터링/로깅/알럿)

배포과정에서 자동화를 통해서 사람을 최대한 배제

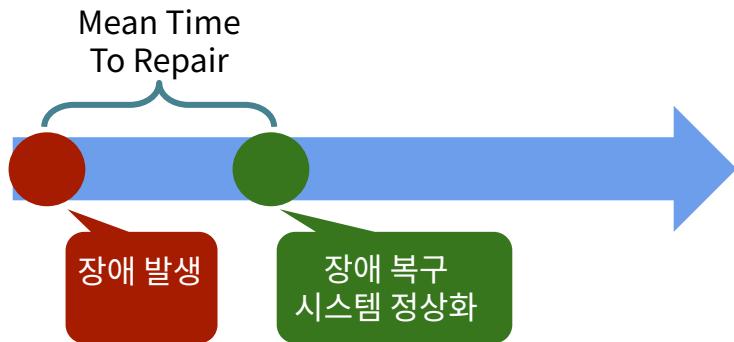
- 에러 감소
- 피로도 감소
- 속도 증가

아키텍처 디자인 가이드

- Devops팀은 개발
설계단에서부터 참여하여,
장애 회피를 위한 설계에
대한 가이드를 제공

¹ Analysis of Google internal data, 2011-2018

장애 대응



장애 대응의 목표는 MTTR 시간 (다운타임)을 최소화 하는 것

- 사람이 프로세스에 있으면, 복구 시간이 늘어남
- 소방관(수퍼맨)이 문제를 모두 해결할 수 있지만, 장애 대응 매뉴얼이 있으면, 장애 복구를 3배 빨리 할 수 있음
- 장애 대응에 대해서 지속적인 훈련을 하는 것이 중요함

※ 테스트 방법론의 카오스 엔지니어링 참고

장애 대응 - 회고 (Postmortem)

장애 대응 후에, 회고를 하는 이유

- 모든 장애는 문서화 되어야 한다. (JIRA와 같은 이슈 트래킹 시스템 사용 권장)
- 장애에 대한 원인 분석과 대응 방법이 제시 되어야 한다.
- 동일한 장애에 대한 발생 가능성을 낮추거나 발생했을 경우 파급효과를 낮출 수 있는 방안이 제시 되어야 한다.

장애 회고 회의는 장애 발생시 반드시 수행되어야 함

- 장애 문서화 및 회고가 처벌처럼 사용되면 안됨.

장애 대응 - 회고 (Postmortem)

장애 회고 미팅에 대한 문화

- 자아 비판, 책임 전가, 마녀 사냥의 자리가 되지 말아야함.
- 특정 개인이나, 팀을 지칭하지 말아야함.
- 순수하게 문제에 초점을 맞춰서 원인 파악 및 재발 방지책 토론의 자리로 만들어야함
- Blameless culture and shared responsibility
- 사람이 실수하는 것은 개인의 실수가 아니라, 그 사람이 실수를 할 수 있는 기회를 만든 시스템의 문제이다.
- 사람은 고칠 수 없지만, 시스템은 개선할 수 있다.
- 비난하고 책임을 전가하는 문화에서는, 문제가 있을때 아무도 그 문제를 들고나와서 해결하려고 하지 않는다.

FinOps

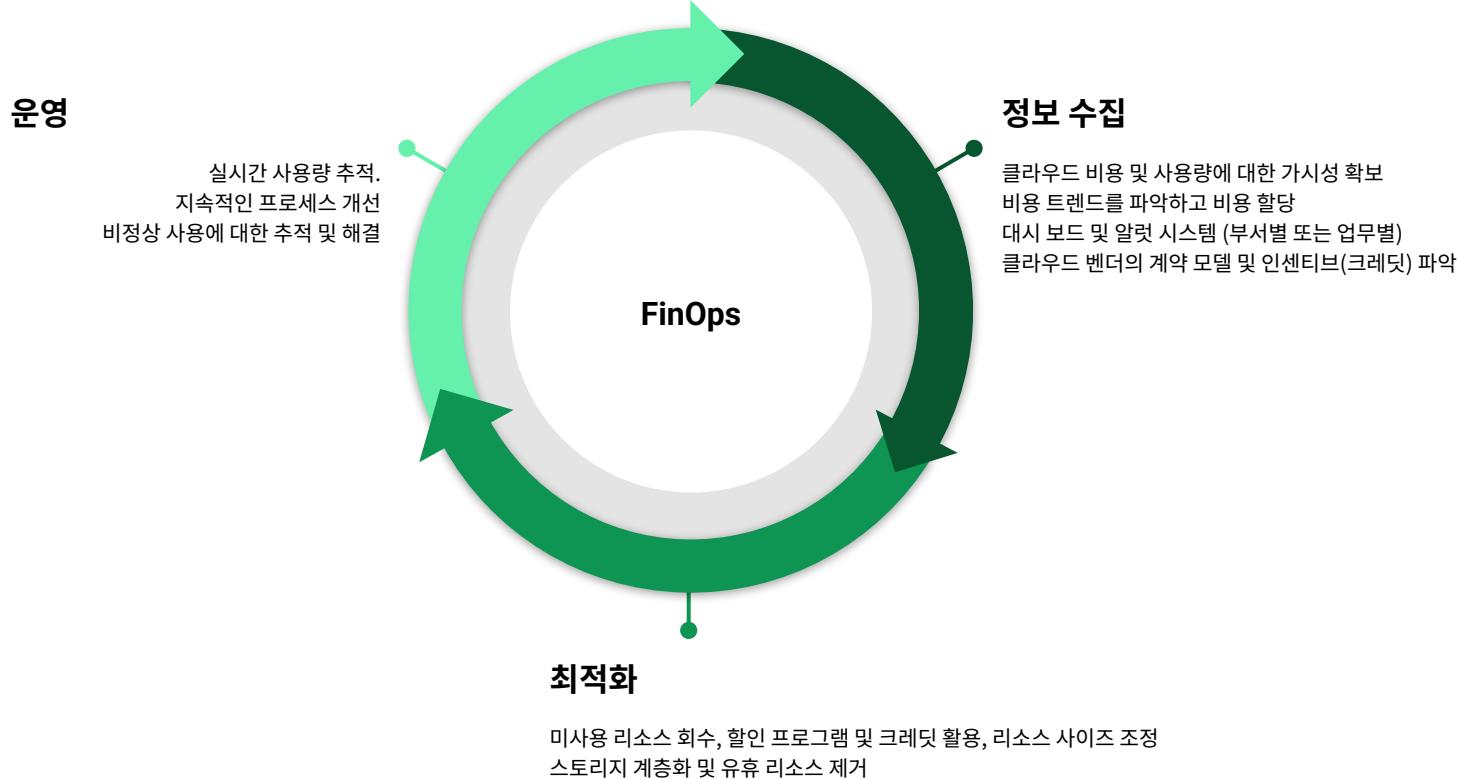
FinOps란?

- FinOps(Financial Operations)는 클라우드 지출을 관리하고 최적화하는 새로운 운영 모델이다.
- FinOps는 재무, 기술, 비즈니스가 협력하여 데이터 기반 의사 결정을 내릴 수 있도록 지원한다.
- FinOps는 클라우드 사용에 대한 책임을 공유하고 비용을 최적화하며 비즈니스 가치를 극대화하는 것을 목표로 한다.
- FinOps를 통해 기업은 클라우드 비용을 예측하고 관리하여 예산 초과를 방지하고 효율성을 높일 수 있다

FinOps의 핵심 원칙

- **협업**: FinOps는 재무, 기술, 비즈니스 팀 간의 긴밀한 협업을 요구한다.
- **가시성**: 모든 이해 관계자는 클라우드 비용 및 사용량에 대한 정확하고 시기적절한 정보에 액세스할 수 있어야 한다.
- **책임**: 팀과 개인은 자신의 클라우드 사용량에 대해 책임을 져야 한다.
- **최적화**: 지속적인 최적화 노력을 통해 비용을 절감하고 효율성을 높여야 한다.
- **가치 기반 의사 결정**: 클라우드 투자는 비즈니스 가치와 전략적 목표에 따라 이루어져야 한다.

FinOps 수행



비용 절감 전략

계약 관점

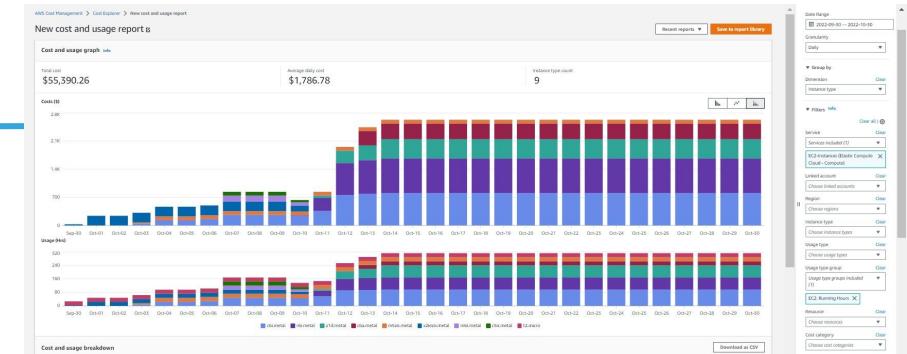
- (1년,3년) Reserved Instance, Spot instance 적극 활용
- EA 계약 활용
- 제품 단위 디스카운트
- 크레딧 프로그램 활용

운영 관점

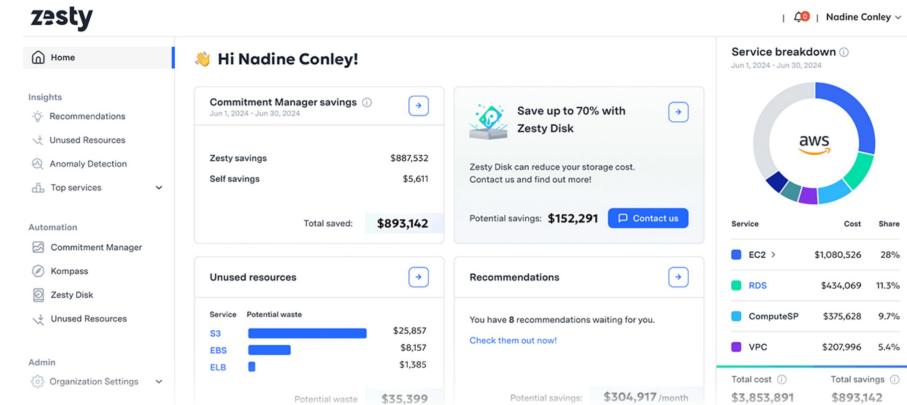
- 적절한 인스턴스 사이즈
- 적절한 CPU 플랫폼
- 적절한 IO, 네트워크 대역폭
- 오토스케일링, 사용하지 않는 인스턴스 다운
- 주기적인 파일 청소, 스토리지 티어링

FinOps 도구

- 클라우드의 Billing & Cost 모니터링 도구 사용
- 데이터를 export하여 별도의 대쉬 보드를 만드는 것을 권장
- Zestly, FinOut, Spot (by Netapp)등의 전용도구등이 있음



출처 : aws.com



출처 : zestly.co

시스템 모니터링 및 매트릭스 정의 방법

SLI (Service Level Indicator)

SLI는 시스템을 모니터링하기 위한 지표로, 서비스를 정량적인 수치나 수준(Level)로 정의하는 것

- **Request latency(응답 시간)**: 요청에 대한 응답을 반환하는 데 걸리는 시간
- **Error rate(오류율)**: 성공적인 요청의 비율
- **Throughput(처리량)**: 초당 처리되는 양을 측정하며, 예를 들어 TPS(초당 처리량), QPS(초당 퀴리 수)와 같습니다
- **Availability(가용성)**: 프로덕션에서 서비스 가능한 시간과 같은 시스템의 가동 시간
- **Durability(내구성, 스토리지 시스템 전용)**: 데이터가 오랜 기간 동안 유실없이 보존될 가능성

워크로드 타입에 따른 SLI

- 사용자 서비스 시스템 (웹, API 서버): availability, latency, and throughput
- 스토리지 시스템: latency, availability, and durability
- 데이터 분석 시스템: throughput, end-to-end latency
- 머신러닝 시스템: latency, availability, throughput, accuracy (prediction) and training time (training)

SLI 수집

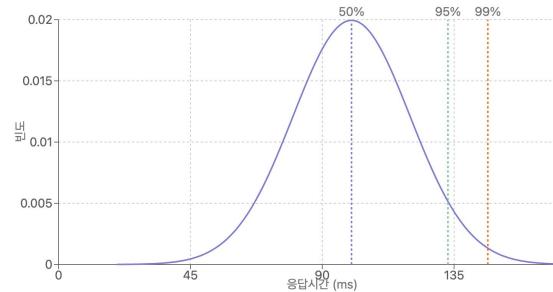
집계 방법

- 일정 시간동안의 여러 지표를 대표값으로 집계 하는 방법
- 합산, 평균, 중간값 등을 사용
- 퍼센타일 기반의 지표가 평균/중간값보다 효율적

퍼센타일

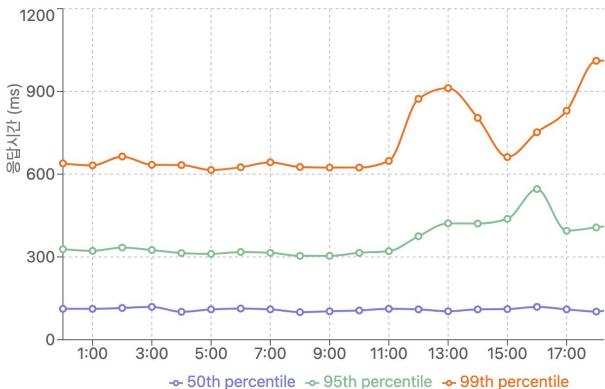
- 중간값이나 평균값은 이상적인 지표를 보여주는 것으로, 문제가 있는 구간의 값을 대표하지 않음
- **95%, 99%** 높은 퍼센타일 구간에 있는 지표가 최악의 케이스를 보여주는 경우가 많음
- Devops는 95%, 99% 구간에 집중하는 것이 좋음

응답시간 정규 분포와 퍼센타일



50th percentile (중앙값): 100ms
95th percentile: 133ms
99th percentile: 147ms

시스템 응답시간 모니터링



SLI 표준화

- 같은 응답시간이라도 사람마다 그 의미가 다른
 - 모바일 개발자 : 단말에서 측정한 응답시간
 - 서버 개발자 : 서버 로그에서 측정한 응답시간
 - 인프라 운영 : 로드밸런서에서 측정한 응답시간
- 지표에 대한 표준화가 매우 중요.
- 표준 템플릿을 만들어서 조직내에서 사용하는 것을 권장

항목	설명
집계 주기	1분간의 평균값
수집 위치	한국 클러스터
수집 주기	매 10 초마다 수집
어떤 요청	GET /users/{userid}
데이터 수집 방법	서버 로그 파일에서 응답시간 필드로 수집

SLI 정의

3-5 SLIs^{*}

* 하나의 사용자 스토리당

SLO (Service Level Objectives)

SLI에 목표값을 지정한 값

SLO = SLI + 목표값 + 측정 기간

- **지표**: 어떤 메트릭을 모니터링 할 것인지 (SLI)
 - **목표 수준** : 타겟값으로, 성능 평가의 목표
 - **측정 기간** : 성능 평가 기간



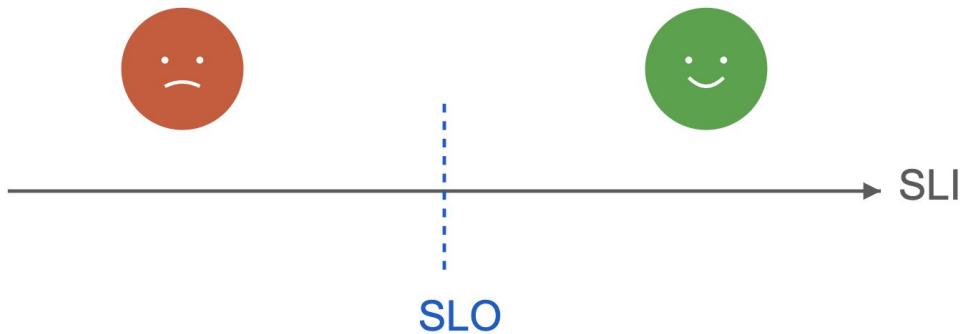
SLO

좋은 SLO?

- 고객 피드백을 이용하여 고객이 불만이 없는 구간이 타겟값

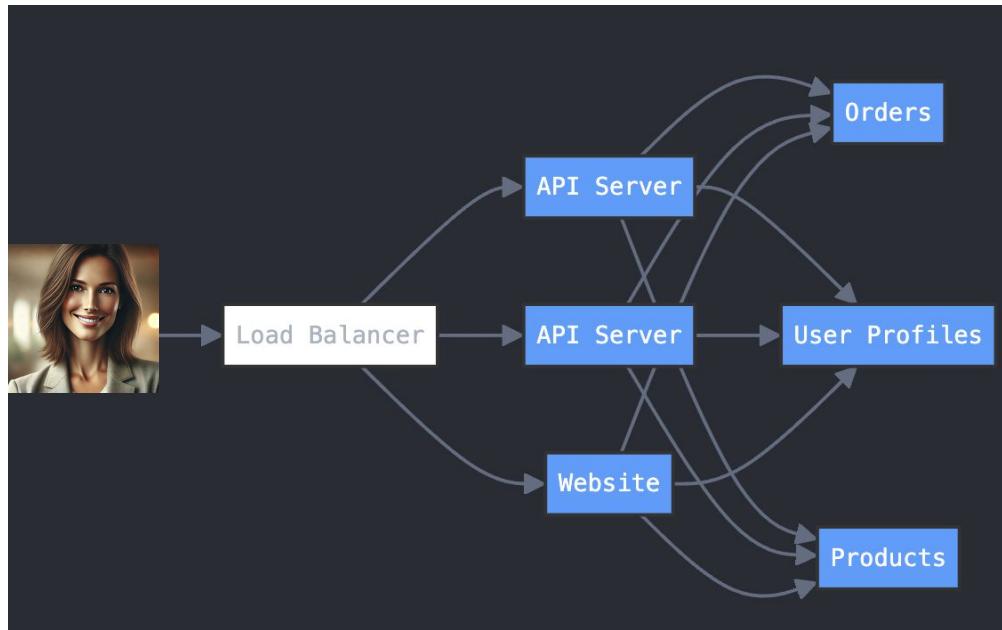
SLO 정의 방식

- 되도록 단순하게
- 완벽하려고 하지 말것
- 가능하면 적은 SLO가 좋음
- 피드백을 통해서 SLO는 점차적으로 변경
- 업계에 알려진 표준값 참고



SLO 정의 예제

온라인 쇼핑몰 시나리오를 통해서 어떻게 SLI, SLO를 지정하는지를 알아본다.



User profile service

<https://myshop.com/profile/Lora>

Lora's Profile

Name: Lora Chang

Email Address: user@example.com

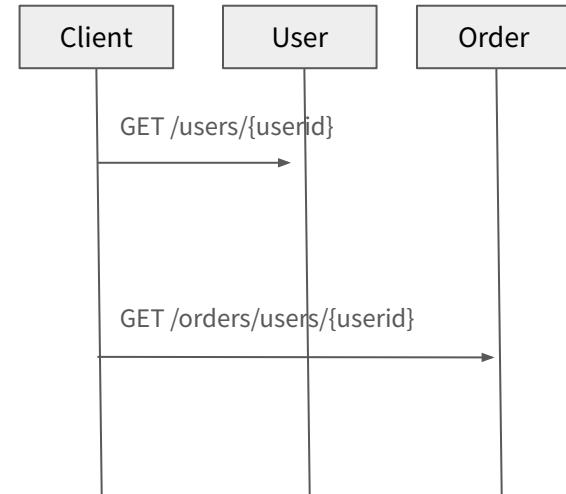
Shipping Address: 644 San Jose, CA, US

Shopping list

- Jesus TV * 1
- DioL bag * 1
- Coke * 10

Update

Orders



워크로드 패턴 선택

앞에서 정의한 워크로드 패턴 중에서 일치하는 워크로드를 선택하여, SLI를 선택

- 사용자 서비스 시스템 (웹, API 서버): availabilities, latency, and throughput
- 스토리지 시스템: latency, availability, and durability
- 데이터 분석 시스템: throughput, end-to-end latency
- 머신러닝 시스템: latency, availability, throughput, accuracy (prediction) and training time (training)

SLI와 SLO 정의 예시

Availability

profile page 가 성공적으로 로딩되었는가?

Latency

profile page 얼마나 빨리 로딩 되었는가?

SLI와 SLO 정의 예시

Availability

profile page 가 성공적으로 로딩되었는가?

- 성공을 어떻게 정의할것인가?
- 어디서 메트릭을 수집할것인가?

Latency

profile page 얼마나 빨리 로딩 되었는가?

- 빨리라는 정의는 무엇인가?
- 언제부터 언제까지 응답시간을 측정할것인가?

SLI와 SLO 정의 예시

Availability

profile page 가 성공적으로 로딩되었는가?

- 성공을 어떻게 정의할것인가?
- 어디서 메트릭을 수집할것인가?

응답중에서 정상적인(에러포함) 응답을 받은 비율

Latency

profile page 얼마나 빨리 로딩 되었는가?

- 빨리라는 정의는 무엇인가?
- 언제부터 언제까지 응답시간을 측정할것인가?

정상적인 응답중에서, 목표 응답시간보다 빠른 응답시간의 비중

SLI와 SLO 정의 예시

Availability

profile page 가 성공적으로 로딩되었는가?

- 성공을 어떻게 정의할것인가?
- 어디서 메트릭을 수집할것인가?

HTTP GET /users/{userid}와
/orders/users/{userid} 가 정상적으로 응답을 준 비율

Latency

profile page 얼마나 빨리 로딩 되었는가?

- 빨리라는 정의는 무엇인가?
- 언제부터 언제까지 응답시간을 측정할것인가?

HTTP GET /users/{userid}와
/orders/users/{userid} 의 응답시간이 목표값보다
빠른 비율

SLI와 SLO 정의 예시

Availability

profile page 가 성공적으로 로딩되었는가?

- 성공을 어떻게 정의할것인가?
- 어디서 메트릭을 수집할것인가?

HTTP GET /users/{userid}와
/orders/users/{userid} 가 HTTP response 코드
2xx, 3xx, 4xx 코드를 받은 비율 (5xx 제외)

Latency

profile page 얼마나 빨리 로딩 되었는가?

- 빨리라는 정의는 무엇인가?
- 언제부터 언제까지 응답시간을
측정할것인가?

HTTP GET /users/{userid}와
/orders/users/{userid} 가 각각 100 ms 내의
응답을 한 비율

SLI와 SLO 정의 예시

Availability

HTTP GET /users/{userid}와
/orders/users/{userid} 가 HTTP response
코드를 Loadbalancer에서 측정했을때,
**code가 2xx, 3xx, 4xx 코드를 받은 비율 (5xx
제외)**

Latency

HTTP GET /users/{userid}와
/orders/users/{userid} 호출의 응답 시간을
Load balancer에서 측정했을때 **100 ms**
내의 응답을 한 비율

SLI와 SLO 정의 예시

여기에 목표값에 기간을 더해서 SLO를 정의한다.

Service	SLO Type	Objective
Web: User Profile	Availability	30일 동안 99.95% 성공
Web: User Profile	Latency	30일 동안 90%의 응답이 100ms 이하
...	...	

Good Valid SLI

Good의 의미

Good은 특정 요청 또는 이벤트가 서비스의 성공 기준을 충족했음을 의미한다.

예를 들어, **Good**으로 간주되려면 아래 조건을 만족해야 할 수 있다:

- HTTP 요청이 200 상태 코드로 응답했다.
- 데이터베이스 쿼리가 지정된 시간(예: 100ms) 이내에 완료되었다.
- 웹페이지 로딩 시간이 SLA에서 정한 임계값 (예: 2초) 이내였다.

Valid의 의미

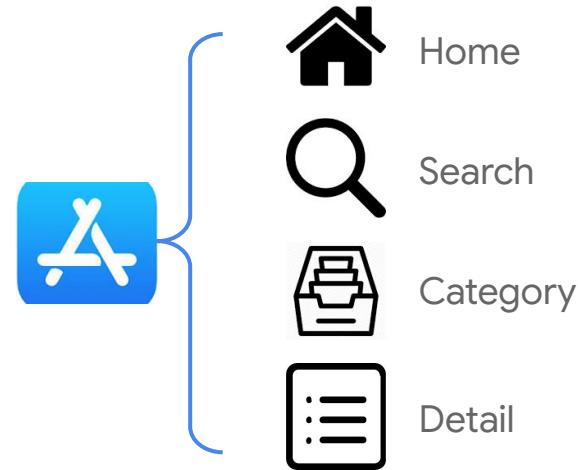
- **Valid**는 해당 요청 또는 이벤트가 SLI를 측정할 때 평가 가능한 유효한 범위 안에 있는지를 나타낸다.(주로 에러)
- 요청이 **Valid**하지 않다면, 이는 SLI 계산에서 제외된다. 예를 들어:
 - 클라이언트가 잘못된 요청을 보냈을 때 (예: 400 Bad Request).
 - 요청이 정상적인 서비스 범위가 아닌 외부적인 요인에 의해 영향을 받았을 때 (예: 네트워크 장애).

실제 예시

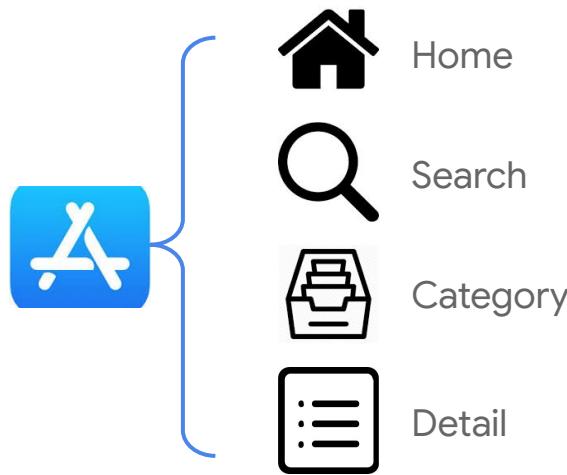
실제 시스템에서는 사용자 시나리오가 많고,
여러개의 SLI가 사용된다.

여러 SLI를 각각 사용할 수 있지만 전체 시스템의
상태를 한눈에 알아보기 어렵다.

Aggregating multiple SLI를 사용한다.



실제 예시



	Good	Valid
Home	9994	10000
Search	9989	10000
Category	9997	10000
Detail	10000	10000
Total	33980	40000
browse	99.95%	

→ Aggregated SLI

에러 예산 (Error Budget)

Error budget

에러 버짓은 시스템이 허용할 수 있는 다운 타임을 정의한다.

$$\text{Error budget} = [100\% - \text{availability target}]$$

예를 들어 Availability SLO가 99.999%인 경우 Error Budget은 100%-99.999%로 0.001%이며, 한달에 25.9초가 된다.

Availability level	Allowed unavailability window		
	per year	per quarter	per 30 days
90%	36.5 days	9 days	3 days
95%	18.25 days	4.5 days	1.5 days
99%	3.65 days	21.6 hours	7.2 hours
99.5%	1.83 days	10.8 hours	3.6 hours
99.9%	8.76 hours	2.16 hours	43.2 minutes
99.95%	4.38 hours	1.08 hours	21.6 minutes
99.99%	52.6 minutes	12.96 minutes	4.32 minutes
99.999%	5.26 minutes	1.30 minutes	25.9 seconds

Error Budget

Error Budget은 어떻게 활용할 수 있는가?

아래와 같은 상황에서 시스템이 다운되었을 경우에 Error Budget을 차감한다.

- 장애
- 예정된 유지보수 (설날 업데이트)
- 배포로 인한 시스템 정지
- 장애 대응 훈련

즉 시스템의 다운이 발생하는 경우, 다운 타임 만큼 Error Budget을 차감

Error budget이 소진 되었을 경우?

회사마다 정책을 정할 수 있음

- 새로운 기능 개발을 멈추고, 안정성을 높이는 작업에만 집중
- Error budget이 소모된 팀/개인에 대해서는 릴리즈마다 고강도 코드리뷰를 실행



Error Budget의 장점

- ▶ 개발과 운영팀에 명시적으로 허용된 장애 시간 제공

새로운 기능을 개발할것인지, 시스템 안정성을

- ▶ ~~개발과 운영팀이~~ ~~제공으로~~ 위기를 관리할 수 있는 기회가 생김

남은 Error budget을 보고, 위기 관리를 할 수 있음

- ▶ 도달 불가능한 목표는 의욕을 감소 시킴

0%, 99.999% 와 같은 수치는 매우 높은 수치로 현실 가능한 목표를 수립하도록. (예외 : 미션 크리티컬 시스템 - 은행등)

- ▶ Error budget내에서 팀이 스스로 규칙을 지킴

수치화된 다운타임을 통해서, 팀이 이를 스스로

- ▶ ~~제공하는~~ ~~다운타임으로~~, 조직내의 책임 공유

Error budget내에서 장애들은, 조직에서 인정해주고, 절대 탓하지 않음.

Error budget의 범위

컴포넌트별 Error budget

시스템을 구성하는 각 서비스별로 Error Budget을 할당하여 관리

장점:

- 각 컴포넌트의 안정성을 개별적으로 관리하고 개선하는 데 집중할 수 있음
- 특정 컴포넌트의 장애가 다른 컴포넌트에 미치는 영향을 제한하여 전체 시스템의 안정성을 높일 수 있음
- 컴포넌트 담당 팀에게 명확한 책임과 권한을 부여하여 ownership을 강화할 수 있음

단점:

- 컴포넌트 간의 의존성을 고려하지 않으면 전체 시스템의 Error Budget을 초과하는 상황이 발생할 수 있음
- 컴포넌트별 Error Budget을 관리하는 데 추가적인 오버헤드가 발생할 수 있음

전체 시스템에 대한 Error budget

전체 시스템을 하나의 Error Budget으로 관리

장점:

- 전체 시스템의 안정성을 단일 지표로 관리하여 목표 달성을 여부를 명확하게 파악할 수 있음.
- 컴포넌트 간의 의존성을 고려하여 Error Budget을 효율적으로 관리할 수 있음.

단점:

- 개별 컴포넌트의 안정성 문제를 파악하고 개선하기 어려울 수 있음.
- 특정 컴포넌트의 장애가 전체 시스템에 미치는 영향을 제한하기 어려울 수 있음.

Error budget의 범위

컴포넌트별 Error budget

시스템을 구성하는 각 서비스별로 Error Budget을 할당하여 관리

전체 시스템에 대한 Error budget

전체 시스템을 하나의 Error Budget으로 관리

실제로는 혼합하여 사용하는 경우가 많음

- 예를 들어, 전체 시스템에 대한 Error Budget을 설정하고, 중요 컴포넌트에 대해서는 별도의 Error Budget을 할당하여 관리
- 컴포넌트별 Error Budget을 설정하고, 이를 종합하여 전체 시스템 Error Budget을 계산할 수도 있음 (좋지 않음)

토일(Toil)

Toil

반복적이고 수동으로 해야하는 작업

- 경비처리나 입사 면접등은 Toil이 아님.
- 시스템 운영에 해당하는 작업만.
- 수동 배포, 수동 패치, 장애 대응



Toil 이란?

- **Toil은 사람이 수행하며 반복되는 작업이다.**

Toil은 사람이 수행하는 반복적인 작업으로, 지속적으로 발생한다.

- **비즈니스에 가치를 더하지 않는다.**

사람이 Toil 작업을 수행해도 비즈니스에 추가적인 가치를 제공하지 않는다. Toil 작업은 시스템 상태를 이전과 이후 사이에 변화시키지 않는다. 예를 들어, 인시던트(incident) 처리 작업은 Toil에 해당하지만, 시스템 상태를 변경하지 않고 사용자에게 추가적인 가치를 제공하지 않는다.

- **서비스 성장과 함께 증가한다.**

Toil 작업의 양은 비즈니스 성장과 함께 증가한다. 비즈니스 성장은 시스템 규모의 확대를 의미하며, 이는 추가적인 운영 작업을 초래한다. 예를 들어, 더 많은 가상 머신에 애플리케이션을 배포해야 하는 작업이 필요해진다.

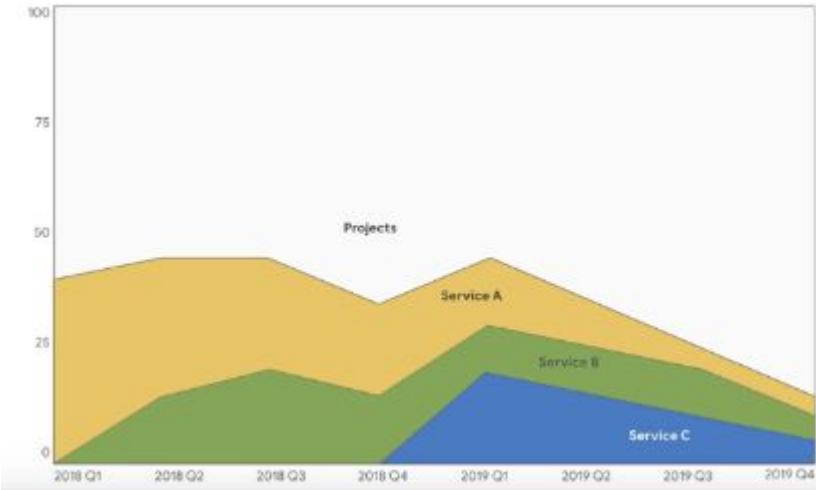
- **자동화가 가능하다.**

Toil의 가장 중요한 점은 이러한 작업이 자동화될 수 있다는 것이다. 하지만 자원의 제약(시간 문제)으로 인해 자동화되지 못한 경우가 많다. 이는 DevOps 세계에서 흔히 말하는 기술 부채(technical-debt)와 유사하다.

Toil를 측정하는 이유?

Toil을 제거하여 확보된 시간으로 비즈니스 가치가 있는 작업을 할 수 있다

- 자동화를 통해 Toil을 제거할 수 있다.
Toil 작업은 자동화를 통해 제거할 수 있다.
- 모든 Toil을 제거할 필요는 없다. (ROI를 고려하라)
모든 Toil을 제거하는 것은 비효율적일 수 있다.
ROI(투자 대비 수익)를 고려하여 적절한 수준의 Toil을 유지하는 것이 좋다.
- 권장 비율은 Toil을 30~50% 유지하는 것이다.
한 서비스에서 Toil을 줄여 시간을 확보한 뒤, 이 시간을 다른 서비스를 관리하는 데 활용하라.



Toil를 측정하는 법?

- 장애 해결 시간 측정
- 주간 업무 일지를 통해서 측정 (생각보다 잘 안됨)
- 팀 설문조사 (생각보다 잘됨)



시스템 모니터링과 장애 대응

시스템 모니터링을 하는 이유?

시스템을 장애 없이 안정적으로 운영하기 위함

01

시스템 안정성과 성능 유지

- 자원 사용량과 응답시간 모니터링을 통하여 과부하나 병목 제가 리소스 사용량을 최적화 하여, 용량 계획 수립

02

장애 예방과 신속 대응

- 장애 예측
장애 발생시 신속한 해결 및 원인 파악

03

보안 및 데이터 분석

- 보안 위협 탐지
- 규제 준수 여부에 대한 지원
보안, 규제, 용량 계획 등을 위한 장기적인 데이터 수집

모니터링 및 장애 대응

대상 시스템 아키텍처 이해

모니터링 대상
시스템의 기능과
아키텍처를 이해

시스템 모니터링 지표와 로그 수집

지표와 로그로 부터
SLI SLO 추출

SLI,SLO를 기반으로
대시 보드를 통한
시각화

장애에 대한 알림

이벤트 관리

배포,유지 보수등
계획된 이벤트에 대한
관리

장애 지원

장애 복구 (롤백)
장애 내용을 티켓으로
기록하여 추적

모니터링/로깅을
통하여 원인 분석

회고

장애에 대한 회고
회의를 통하여 재발
방지책 수립

모니터링 시스템 구축시 중요사항

도구



방법론

- 어떤 정보 구조를 가지고 모니터링 할것인가?
- 어떤 지표를 모니터링 할것인가?
- 어떻게 장애를 예방하고 신속 대응할 수 있는가?
- 시스템 안정성, 성능에 따른 비즈니스 영향도

모니터링 정보 구조

비즈니스 시스템 구조에 따른 모니터링과, 인프라 관점에서 기술분류별 모니터링

아키텍처 구조에 따른 서비스단위 모니터링



인프라/기술에 구조에 따른 모니터링

애플리케이션

미들웨어

하드웨어

모니터링 대시 보드

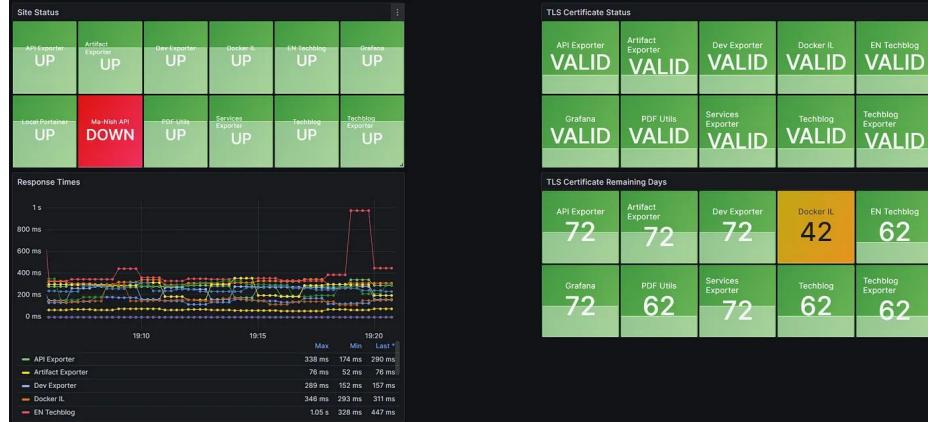
사용하는 사람과, 목적에 따라서 다른 대시 보드가 필요함

대시 보드 종류	내용	사용 대상	노트
실시간 대시 보드	현재 서비스의 전반적인 상태를 모니터링 함	Devops	
애플리케이션 모니터링	애플리케이션 성능 모니터링. 병목. 에러로그	개발자	
인프라 대시 보드	하드웨어 및 미들웨어 인프라 상태 모니터링	Devops	
비용 대시 보드	클라우드 서비스등의 사용 금액을 모니터링	Devops,CFO	
보안 모니터링	서버의 보안 위협 및 로그 분석	Devops,CSO	IPS,IDS,WAF

실시간 모니터링

아키텍처 구조에 대한 업타임 대시보드

- 한화면에서, 전체 서비스가 정상인지 비정상인지 모니터링할 수 있는 단순한 뷰를 제공
- 시스템이 클 경우, 각 서브 서비스별로 별도의 업타임 대시보드를 제공할 수 있음
- SLI/SLO 사용
- OSS : Kuma
- 상용 : DataDog



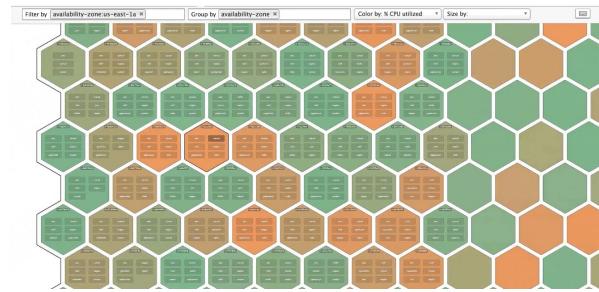
<https://github.com/louislam/uptime-kuma>

인프라 뷰

인프라에 대한 상황을 한눈에 볼 수 있는 뷰

- 하드웨어 및 미들웨어를 같이 볼 수 있어야 함
- 서버, 스토리지, 네트워크, 가상화 및 컨테이너(쿠버네티스), 매니지드 서비스 모니터링
- OSS : Prometheus + Grafana, Zabbix, Nagios
- 상용 : DataDog, NewRelic

데이터독 버드뷰



출처 : datadoghq.com

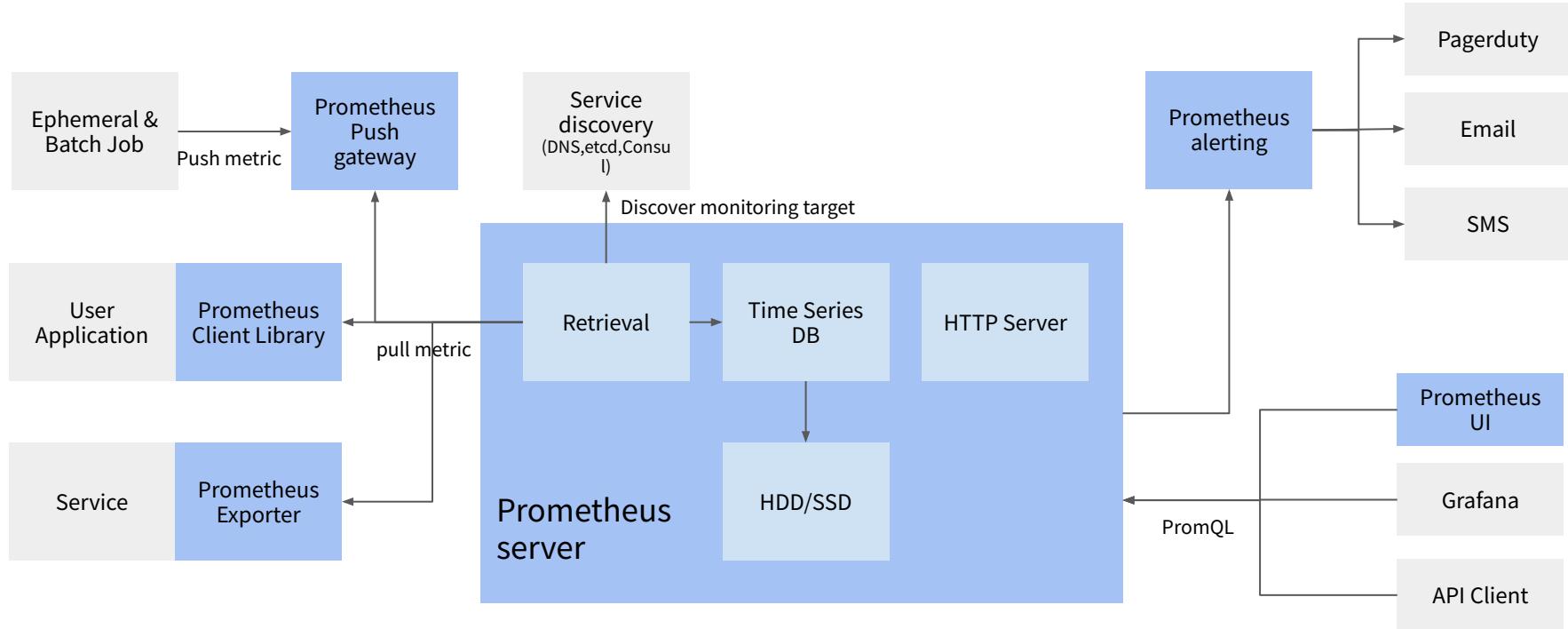
프로메테우스

Prometheus는 원래 SoundCloud에서
개발된 오픈 소스 시스템 모니터링
도구이다.

Prometheus는 Kubernetes에 이어
2016년에 클라우드 네이티브 컴퓨팅 재단
(Cloud Native Computing Foundation)에 두
번째로 호스팅된 프로젝트로 합류하였다.

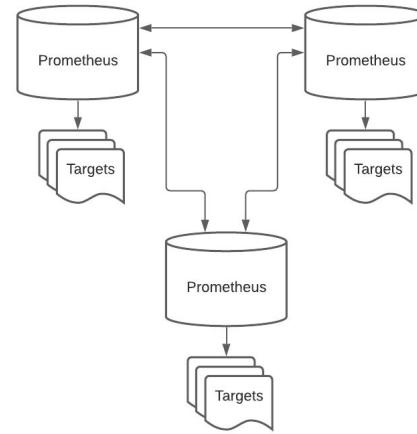


프로메테우스 아키텍처



프로메테우스 제약

- 프로메테우스는 싱글 인스턴스
 - 클러스터링을 통한 스케일링
 - 고가용성 보장이 되지 않음
 - 용량의 한계가 있음
- Sharding
프로메테우스 인스턴스별로 각각 다른 시스템을 모니터링 해서 용량의 한계를 극복
- Federation
각각의 프로메테우스의 메트릭 데이터를 요약하여 중앙 시스템으로 수집하여 중앙 프로메테우스에서 전체 시스템을 모니터링



Thanos

여러개의 프로메테우스 인스턴스를 묶어서

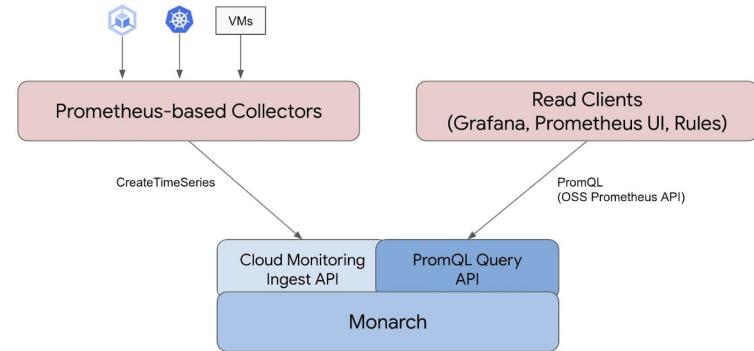
- 스케일링
- 고가용성 지원
- 용량 한계에 대한 Long Term Storage를 S3,GCS, Azure Blob을 통하여 극복



클라우드 프로메테우스

클라우드 서비스의 매니지드 프로메테우스

- 인터페이스는 OSS 프로메테우스와 동일
- 내부 저장구조와 엔진을 자체 모니터링 시스템으로 바꿔서, 기존 프로메테우스의 한계 (고가용성, 용량) 등을 극복함



출처 :
<https://cloud.google.com/stackdriver/docs/managed-prometheus>

알림 서비스

장애 발생시 알림

- 이메일, 슬랙, SMS 등 다양한 채널 지원 필요
 - 24x7 온콜 로테이션 지원 (담당자 지원)
 - 에스컬레이션
담당자가 연락이 안되거나, 지정 시간내에 해결이 안될 경우 차상위로 에스컬레이션
 - 상용 솔루션 : PagerDuty

pagerduty

장애 티켓 관리

장애를 티켓으로 관리

- 기존의 bug 추적 시스템이나 이슈 트랙킹 시스템을 함께 사용하는것이 프로세스 관리상 효과적
- 향후 분류 및 검색이 가능해야 함 (유사 장애 대응 참고)
- 모니터링 로깅 시스템과 연동하여 자동 장애 티켓 생성
- 상용 : JIRA

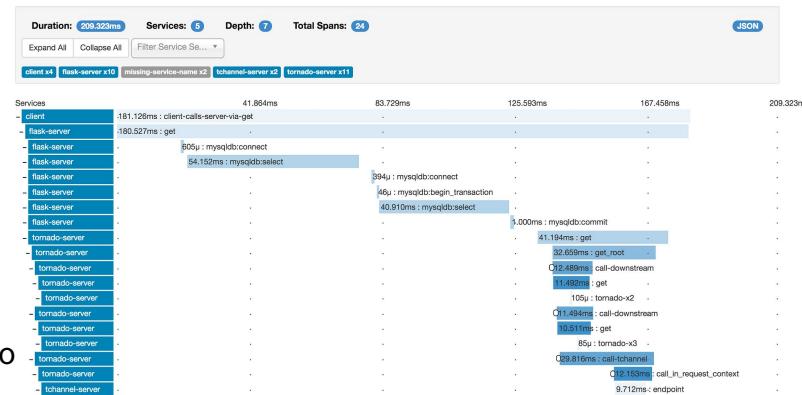
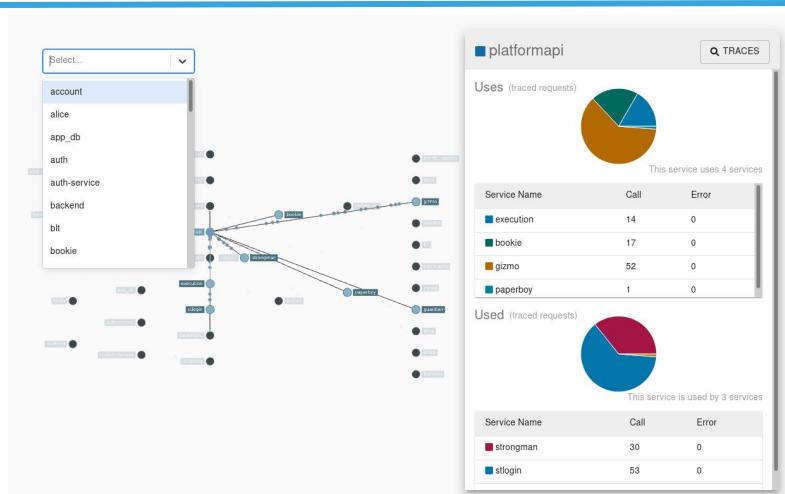
The screenshot shows the Opsgenie web interface with the title "Ongoing major incidents". A message states, "Opsgenie is showing these major disruptions (incidents) right now." Below this is a table listing 14 incidents. The columns are: Key, Message, Opsgenie service/s, Responders, Status, Linked, and Created.

P	Key	Message	Opsgenie service/s	Responders	Status	Linked	Created
P2	#52	Space shuttle won't start	third service	Second team	OPEN	1 request	19/Jul/20 11:47 PM
P3	#51	Problem with rocket thrusters		Confluence Team	OPEN	5 requests	17/Jul/20 5:45 AM
P3	#50	Someone stole the Moon	second service	Test Team	OPEN	5 requests	16/Jul/20 3:11 AM
P3	#49	Robot overboard!	second service	Test Team	OPEN	3 requests	16/Jul/20 3:07 AM
P2	#48	Wormhole is down	Trading Service	Athena	OPEN	0 requests	16/Jul/20 1:49 AM
P4	#47	Where's my ship	Translations Service	Translations Team	OPEN	3 requests	16/Jul/20 12:38 AM
P3	#46	Refunds system is down	Translations Service	Translations Team	OPEN	0 requests	15/Jul/20 1:28 AM
P3	#45	Oxygen depletion		Jira Team	OPEN	3 requests	15/Jul/20 1:25 AM
P3	#44	Mars office is dusty	Confluence Service	Confluence Team	OPEN	3 requests	15/Jul/20 1:19 AM
P1	#43	Incident Management has problems	Sample Service	Test Team	OPEN	5 requests	14/Jul/20 12:30 AM
P5	#42	Aliens are attacking!			OPEN	0 requests	11/Jul/20 8:07 AM
P1	#41	All your base are belong to us	Refunds Service	Billing Team	OPEN	1 request	09/Jul/20 4:40 AM

애플리케이션 뷰

애플리케이션 성능 모니터링

- 어떤 마이크로 서비스가 어떤 마이크로 서비스를 부르는지 토플로지 추적 가능
- 코드 (함수) 레벨로 소요시간 모니터링을 통한 병목 확인
- OSS : Zipkin, Jaeger
- 상용 : Jennifer, DataDog, NewRelic

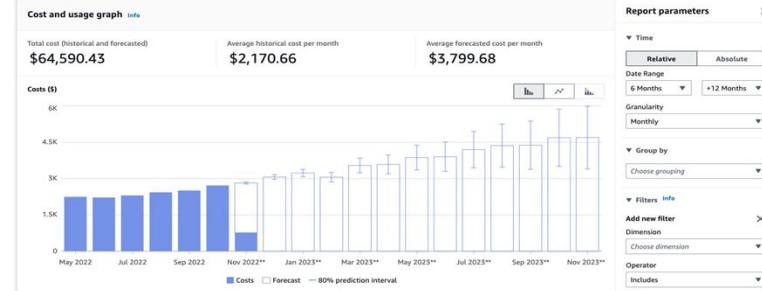
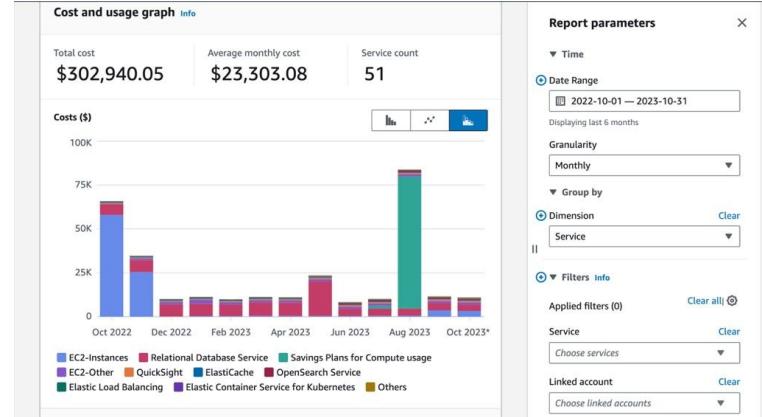


출처 : zipkin.io

비용 관리

클라우드 또는 서비스에 대한 비용 상황 모니터링

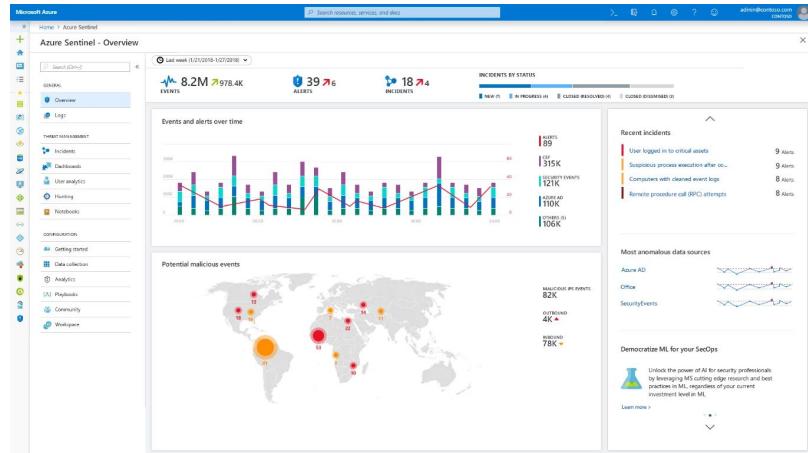
- 기간, 서비스별 세부 항목
- 기존 사용량 기준으로 예측 기능 제공
- 알림 기능 설정 필수 (원하지 않는 스파이크 사용)
- 대부분의 클라우드가 제공은 하지만, 여러 클라우드간 통합된 기능은 제공하지 않음.
특히 3rd 파티 SaaS 서비스에 대한 비용 관리 통합 필요



보안 모니터링

SIEM (Security Information and Event Management)

- 로그와 기타 연동 시스템에서 다양한 이벤트를 수집
- 이벤트의 연관성을 분석하여 보안 위협 탐지
- 규정준수 지원 : 로그데이터를 보관하고 보고서를 생성하여 규정준수를 돋는다.
- 상용 : Microsoft Sentinel,



Microsoft Sentinel

출처 : microsoft.com

모니터링 솔루션

상용

- DataDog
- New Relic
- Cloud Service (Cloud Watch etc)

OSS

- 지표 수집 : Prometheus
- 시각화 : Grafana
- 업타임 대시 보드 : Kuma
<https://github.com/louislam/uptime-kuma>
- 애플리케이션 트레이싱 : Zipkin, Jaeger

모니터링 시스템

- 지표에 대한 표준화
- 모니터링 정보 모델 정의
- 모든 지표를 단일 시스템에서 수집해서 모니터링
- 멀티 클라우드 하이브리드 클라우드 지원
- 알림 시스템 구축 필수

로깅

JSON 로깅

JSON 로깅은 사람이 이해하기도 편하지만, 로깅 시스템 (ELK, Splunk, DataDog)과 연동이 용이함

- Log Level
- Timestamp
- Application context 포함 (어떤 마이크로 서비스인지)
- 사용자 정보 : user id
- Request id : request 고유 id
- 보안 및 민감정보 : REDACTED 된 형태로 로그에 표현하지 않음
- 디버깅용 로그에는 request body를 포함

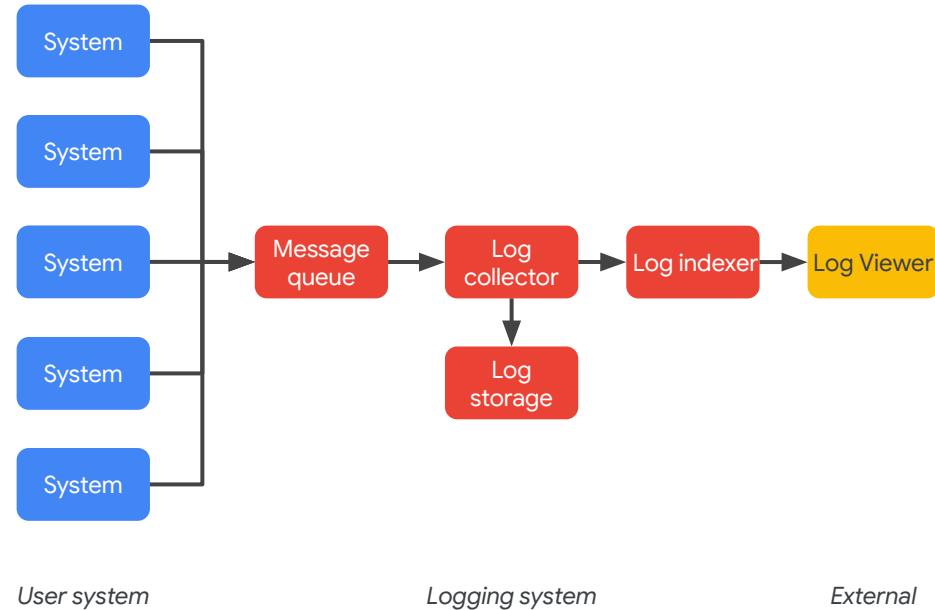
```
{  
    "timestamp": "2024-11-16T12:34:56.789Z",  
    "level": "INFO",  
    "application": "my-app",  
    "message": "Request processed successfully",  
    "user_id": "12345",  
    "request_id": "abc-123",  
    "response_code": 200,  
    "execution_time_ms": 150,  
    "additional_info": {  
        "endpoint": "/api/v1/login",  
        "method": "POST"  
    }  
}
```

```
{  
    "timestamp": "2024-11-16T12:35:00.123Z",  
    "level": "ERROR",  
    "application": "my-app",  
    "message": "Unhandled exception occurred",  
    "error_message": "NullPointerException",  
    "stack_trace": "at com.example.MyClass.method(MyClass.java:45)",  
    "user_id": "12345",  
    "request_id": "abc-123",  
    "additional_info": {  
        "endpoint": "/api/v1/login",  
        "method": "POST"  
    }  
}
```

로깅 시스템

분산 로그 시스템 : 여러개의 분산된 시스템에서 하나의 저장소로 로그를 수집하여 분석 및 모니터링이 가능하게 함.

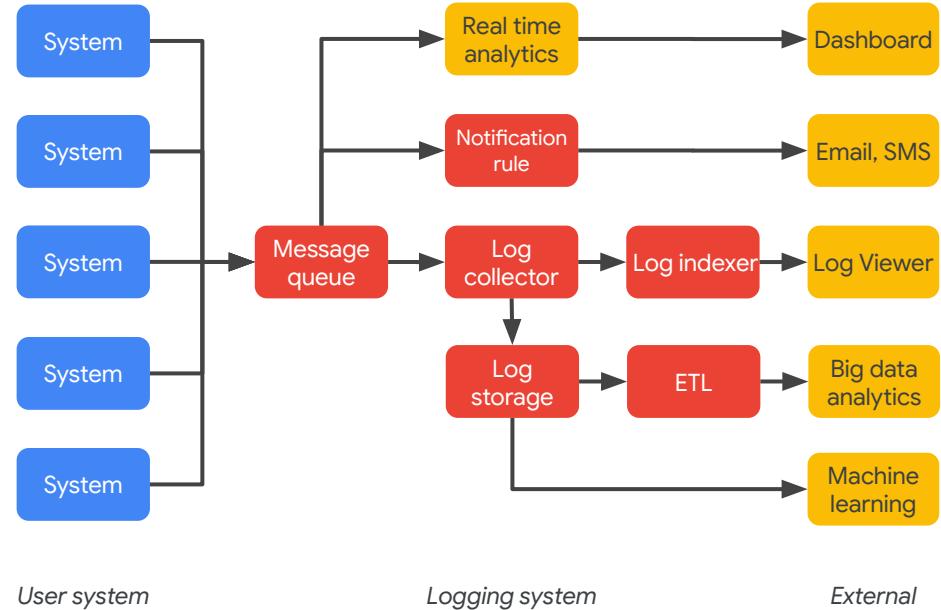
- 각 시스템에서 로그를 수집하여 메시지 큐로 전송한다.
- 메시지 큐는 대량의 로그를 비동기적으로 처리하는데 사용된다.
- 로그는 로그 수집 서버에 의해 수집되어 로그 저장소에 저장된다.
- 로그 인덱서는 검색을 위해 인덱스를 생성한다.
- 사용자는 로그 뷰어에서 로그를 조회하고 검색할 수 있다



현대의 로깅 시스템

현대의 로깅 시스템은 모니터링 용도로만 사용되지 않고, 데이터 파이프라인으로도 사용됨

- 로그 스트림은 실시간 데이터 분석을 위해 실시간 분석 시스템으로 내보내야 한다.
- 이상 징후를 감지하기 위해 로그를 관찰하고 사용자에게 알림을 전송한다.
- 데이터 분석 팀은 인사이트를 찾기 위해 로그를 필요로 한다.
- 머신 러닝 시스템도 학습을 위해 로그가 필요하다.



로깅 시스템

로그는 비싸고 중요한 자원

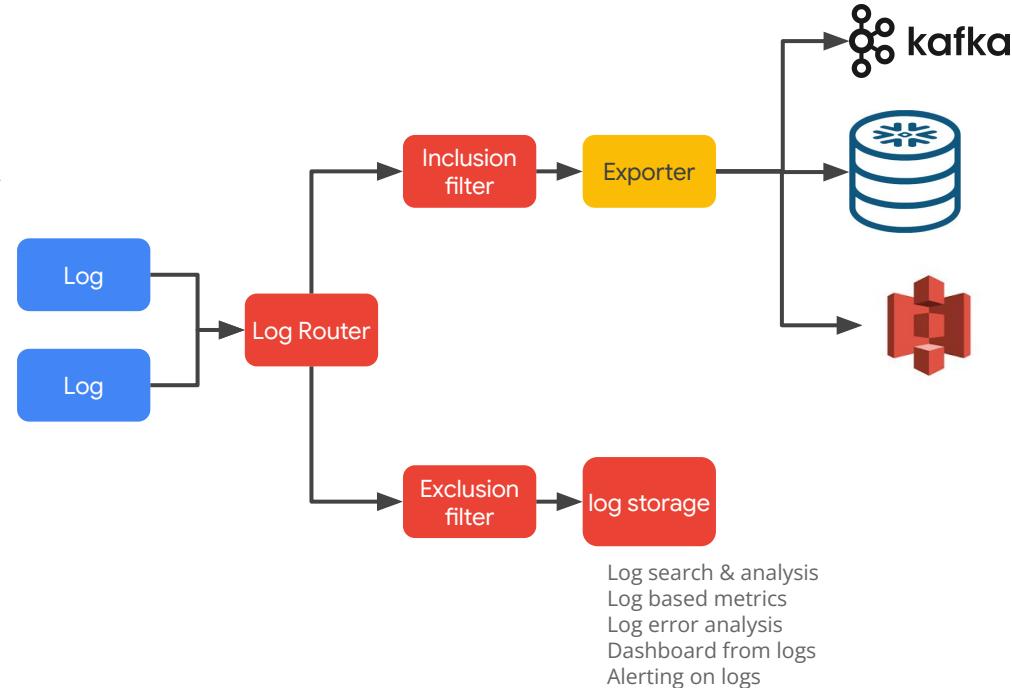
- **Exclusion**

불필요한 로그를 Exclusion 필터를 통해서 로깅 시스템에 저장하지 않음

- **Export**

로그의 용도에 따라 로그를 라우팅

- 실시간 분석 : 메시지 큐
- 추가 데이터 분석 : 데이터 레이크
- 아카이브 : S3,GCS 등 Blob 저장소

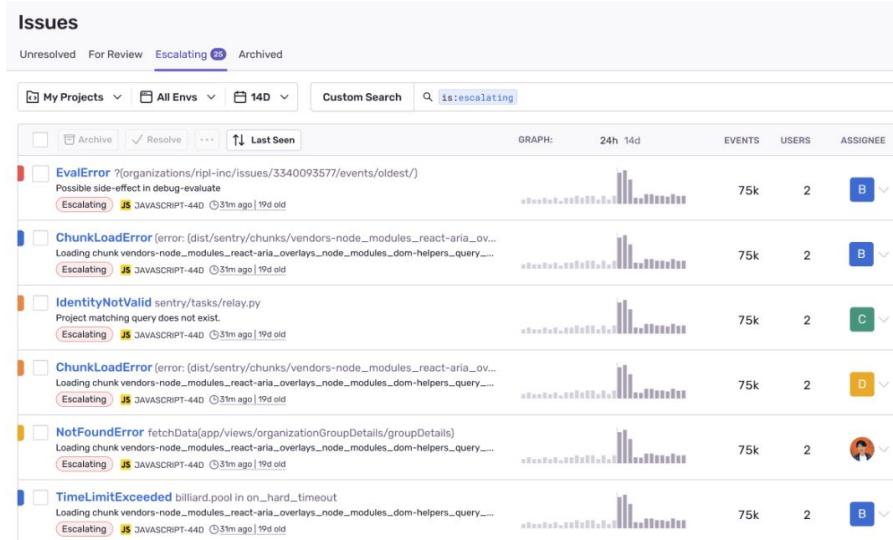


로깅 & 에러 로깅

로깅 시스템은 일반 로그 시스템과 에러 로그 시스템으로 나눠서 운영 가능

- 에러 로그 시스템

- 에러 메세지만 통합해서 관리
- 같은 에러에 대한 히스토리, 발생 빈도를 관리
- 같은 에러메시지에 대해서 티켓 시스템과 연계 가능.
- OSS : Sentry



Sentry 기반의 에러 로그 모니터링

출처 : <https://open.sentry.io/>

로그 메시지 기반 메트릭

로그메시지에서 메트릭을 정의할 수 있다

- 로그 문자열에서 특정 부분을 파싱해서 모니터링 메트릭으로 정의
예) HTTP 로그에서 응답시간 필드를 추출
- 로그의 수를 메트릭으로 정의할 수 있다.
예) 에러 로그의 수

PREVIEW

DATE	HOST	SERVICE	CONTENT
Jan 15 14:11:28.000	fse-ldmx-f7-18	fse-auto-process	> GET /api/auto-process/refund/break HTTP/1.1
Jan 15 14:11:25.000	fse-ldmx-f7-18	fse-auto-process	> GET /api/auto-process/refund/1428 HTTP/1.1
Jan 15 14:11:25.000	fse-ldmx-f7-18	fse-auto-process	> GET /api/auto-process/refund/6887 HTTP/1.1
Jan 15 14:11:25.000	fse-ldmx-f7-18	fse-auto-process	> GET /api/auto-process/payment/5194 HTTP/1.1
Jan 15 14:11:25.000	fse-ldmx-f7-18	fse-auto-process	> GET /order/2243 HTTP/1.1
Jan 15 14:11:25.000	fse-ldmx-f7-18	fse-auto-process	> 2021/01/15 19:11:25 [error] 6#6: *45823773 open() "/var/www/_
Jan 15 14:11:25.000	fse-ldmx-f7-18	fse-auto-process	> GET /send/notice HTTP/1.1
Jan 15 14:11:25.000	fse-ldmx-f7-18	fse-auto-process	> 2021/01/15 19:11:25 [error] 6#6: *45823772 open() "/var/www/_
Jan 15 14:11:25.000	fse-ldmx-f7-18	fse-auto-process	> GET /bad HTTP/1.1
Jan 15 14:11:25.000	fse-ldmx-f7-18	fse-auto-process	> 2021/01/15 19:11:25 [error] 6#6: *45823750 open() "/var/www/_

Live Tail

1 Define query [?](#)

Source:nginx X

Measure Duration group by status X

Calculate percentiles [?](#)

- p50, p75, p90, p95, and p99 aggregations will be created for the applied tags
- Query these aggregations in dashboards and notebooks
- [Read more in the docs](#)

2 Name [?](#)

nginx.request_duration

Create Metric

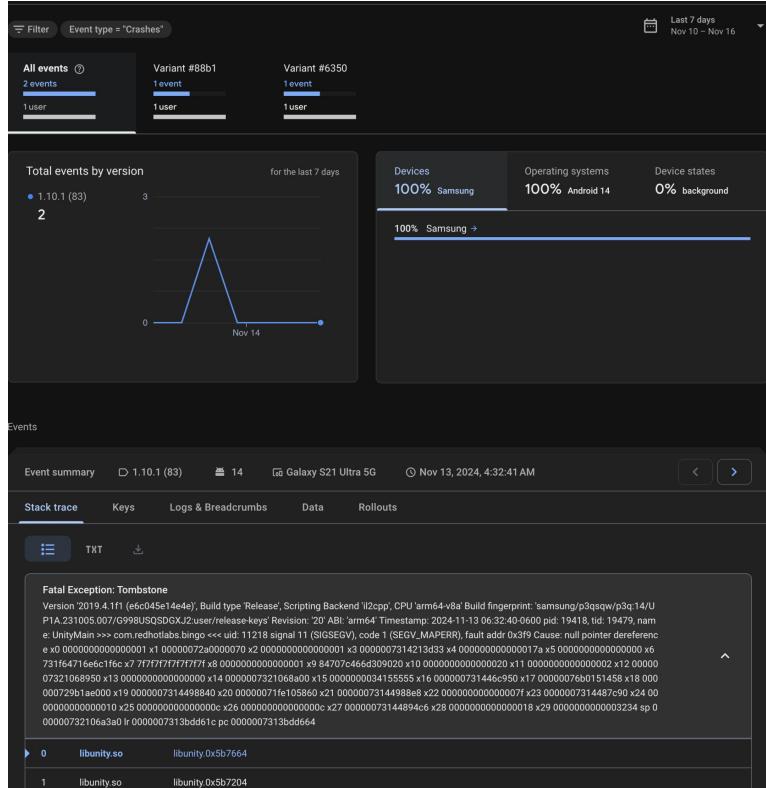
DataDog에서 Log에서 매트릭을 정의하는 화면

출처 : datadoghq.com

모바일 에러 수집

모바일에서 로그는 원격환경이기 때문에
수집하기가 어려움

- 원격으로, 에러로그만 수집
- 에러 로그 수집시, 단말 정보, SDK 정보,
앱 버전등을 포함해서 수집해야 함
(특정 단말이나 SDK 버전에 따라서만
발생하는 경우가 있기 때문에, 수정할지
말지를 결정해야 함)

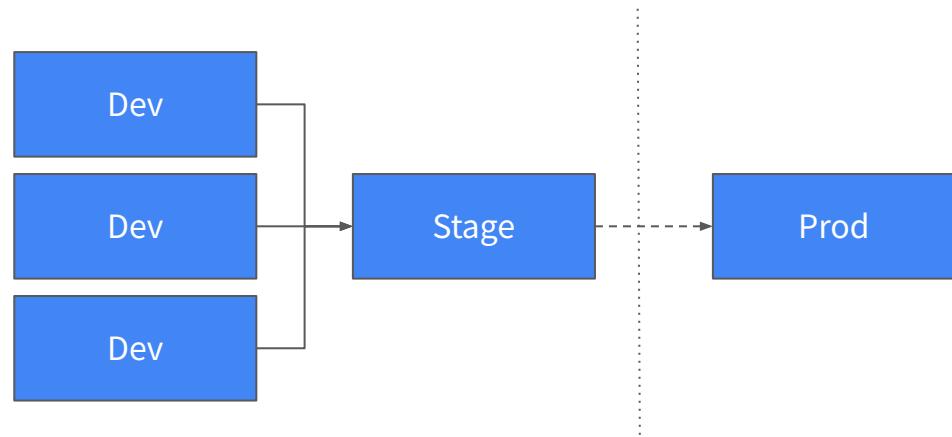


Firebase crashlytics 에러 모니터링 화면

CI/CD

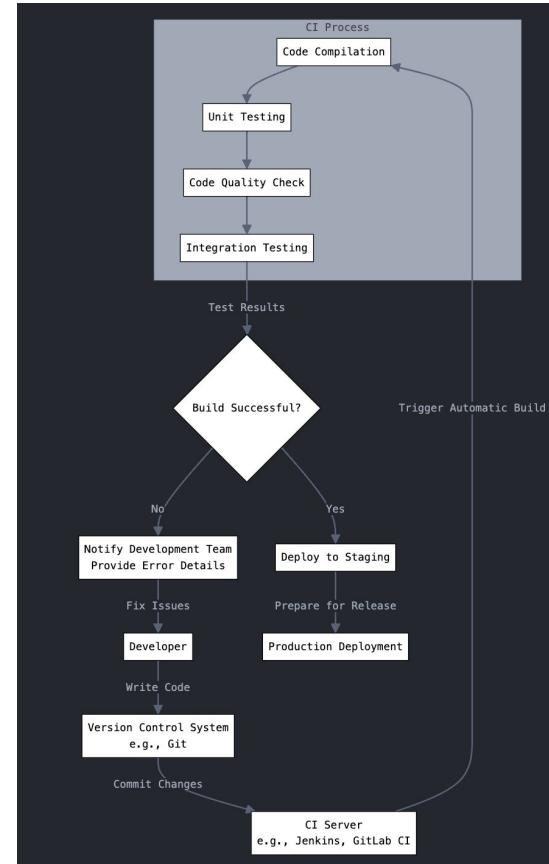
환경

- **Dev(개발 환경)**
개발을 위한 환경
팀마다 별도의 환경을
제공하기도 함
- **Staging (통합 테스트 환경)**
운영 환경과 거의 유사한
환경으로, 코드 통합 후에, 각종
테스트를 수행함.
운영 환경과 유사한 환경을
다운사이징 해서 배포
- **Prod (운영 환경)**
운영 환경.
보안등의 이유로 운영팀만 접근
가능하게 함



CI (Continuous Integration)

- 코드 커밋 시마다 자동으로 빌드를 수행
- 통합 비용 절감
잦은 코드 통합을 통해서 프로젝트 말에 대규모 통합으로 인한 문제를 줄인다.
- 개발속도 향상
안정적인 코드베이스를 유지함으로써 개발자들이 새기능에 집중할 수 있도록 함



CI 도구들

- **버전 관리 시스템**

코드를 저장하는 시스템 (git)

- **빌드 자동화 시스템**

변경된 코드에 대한 빌드를 수행하고,
빌드에 필요한 기타 과정을 자동으로
수행하는 시스템 (Jenkins, Circle CI,
Github Action)

- **테스트 자동화**

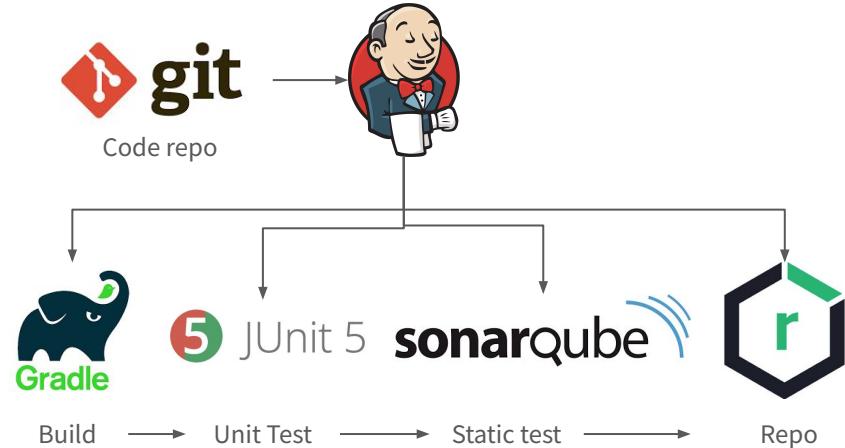
빌드된 코드에 대한 품질 검증

- 단위 테스트 : JUnit
- Web UI : Selenium
- Static Code analysis : Sonarqube (오픈소스
라이센스)

- **아티팩트 리포**

빌드된 아티팩트를 저장, 컨테이너 (Nexus)

- 클라우드 기반 CI (리소스 문제)



CD

Continuous Delivery

- Continuous Integration(지속적 통합) 위에 추가로 릴리스 프로세스를 자동화하는 것이다.
- 사용자는 버튼을 클릭하여 언제든지 애플리케이션을 배포할 수 있다.

Continuous Deployment

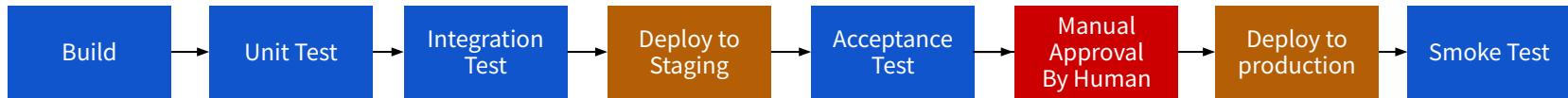
- 사람의 개입 없이 더욱 성숙한 릴리스 자동화(완전 자동화, 버튼 클릭 없이 진행).
- 모든 코드 변경 사항이 자동으로 최종 사용자 서비스에 적용된다.

CD

Continuous integration



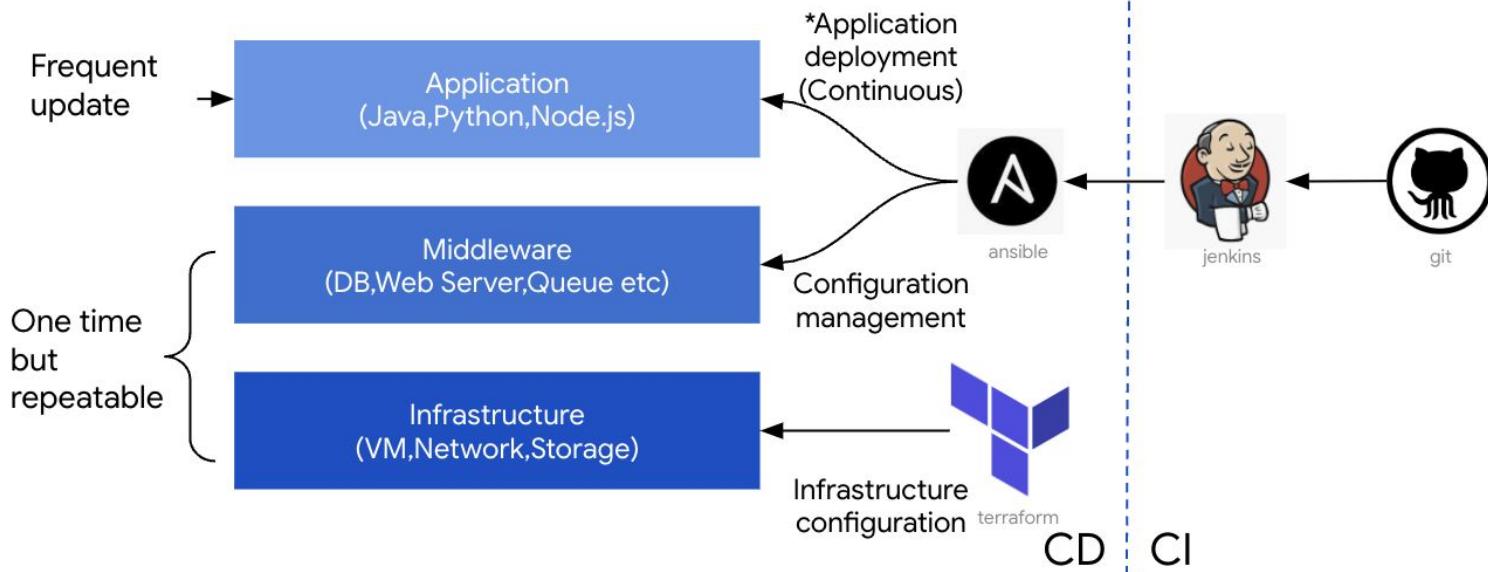
Continuous delivery



Continuous deployment



VM 기반의 CI/CD 파이프라인



스노우플레이크 패턴

- 서버 패치나 미들웨어 업그레이드와 애플리케이션 업데이트가 분리되어 있음
- 애플리케이션만 지속적으로 업데이트 하는 경우
- 미들웨어, 서버 일부만 업데이트 하는 경우
- 서버마다 형상이 다르게 되서 운영상 문제가 발생함
(패치 불일치등)
- 눈송이 같이 모든 서버의 설정이 다른 “스노우 플레이크” 패턴 (안티 패턴)

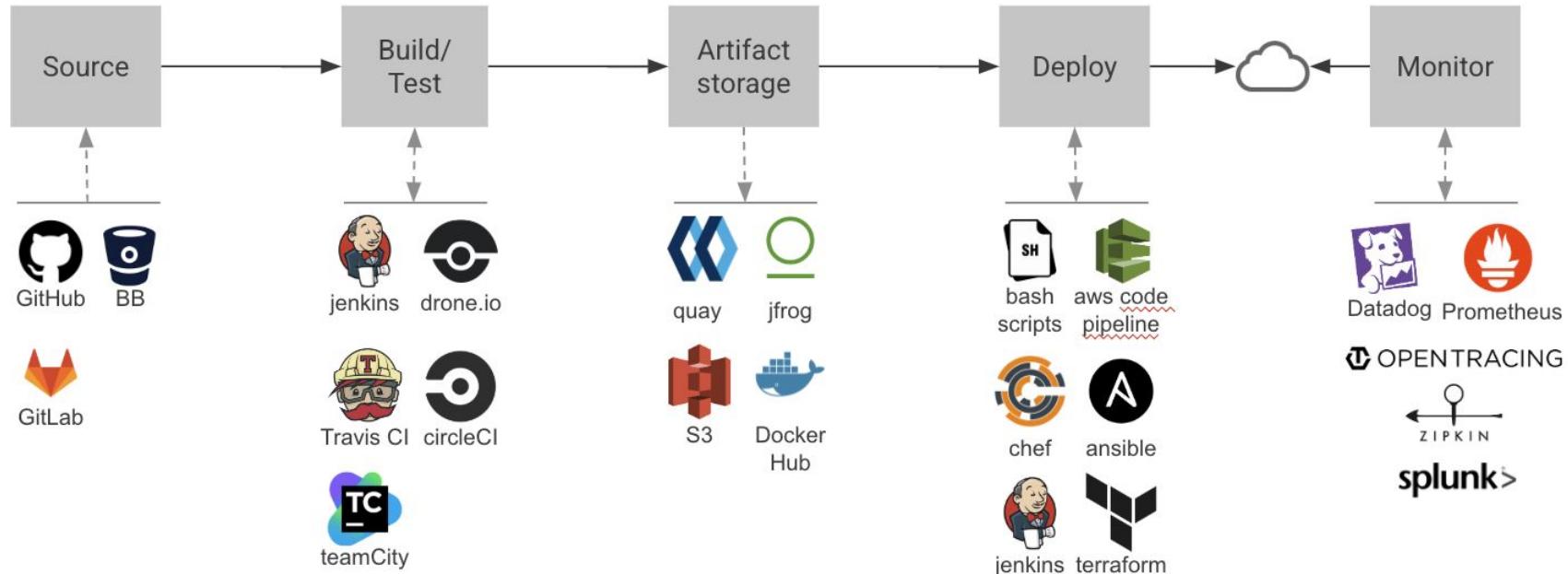


피닉스 패턴

- 서버의 형상을 모두 일치시키기 위한 패턴
- 애플리케이션을 업데이트할때에도 OS에서부터 미들웨어,애플리케이션 전체 스택을 처음부터 다시 인스톨 함
- VM에서는 Packer등을 이용해서 이미지 부터 새로 Baking
- Container/Kubernetes에서는 자동을 피닉스 패턴이 적용됨 (OS 베이스 이미지부터 새로 인스톨)

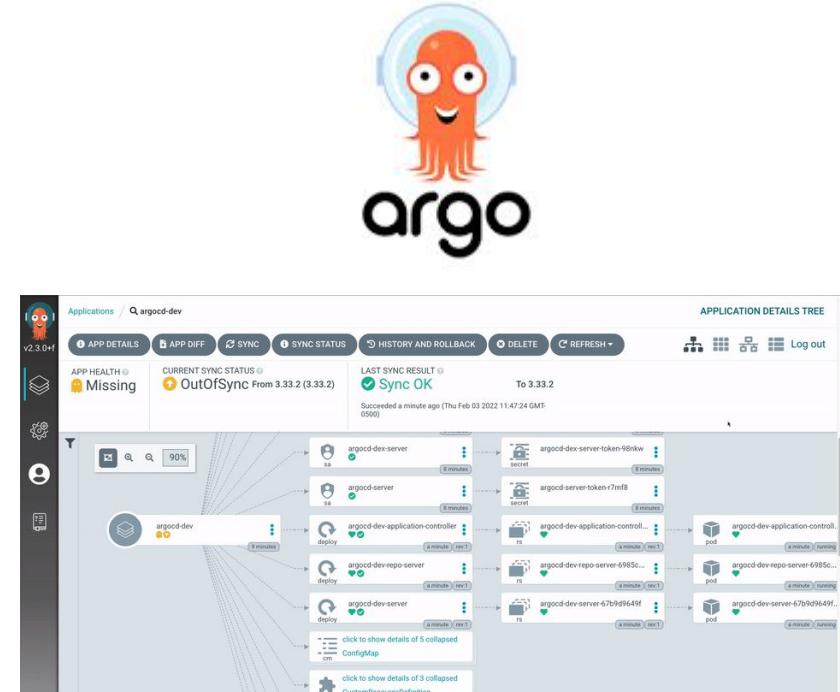


CI/CD 파이프라인



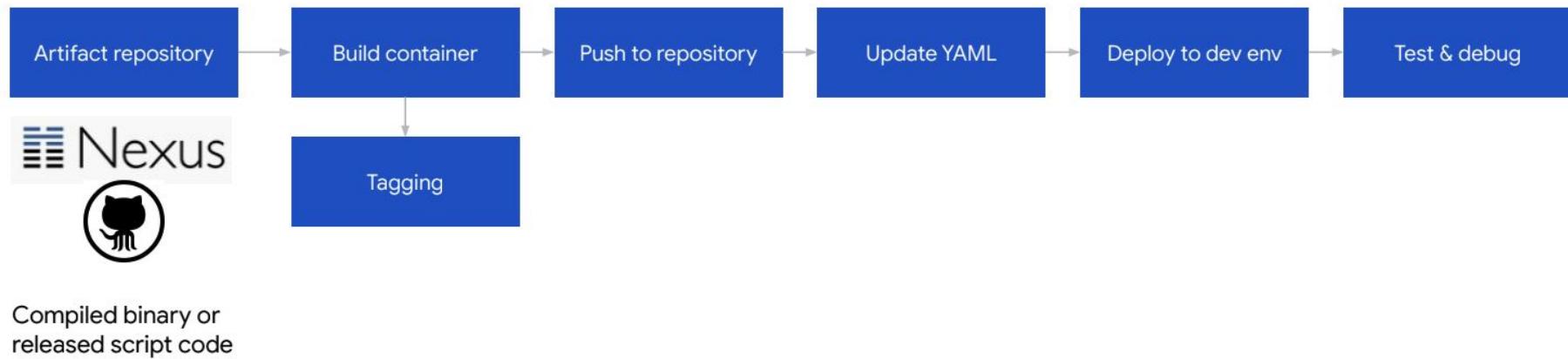
CD 도구

- Spinnaker
멀티클라우드 지원.
복잡한 파이프라인 지원 (Blue/Green 카나리, 사용자 승인, 배포 윈도우등)
대규모 시스템에서 복잡한 배포 관리용 (어려움)
- GitHub Action
Git Hub 사용시 손쉽게 Git와 통합 가능
소규모 팀에 적절
- ArgoCD
쿠버네티스에서 가장 많이 사용됨
Helm,Kustomize,YAML 등 기존
Kubernetes 이코 시스템과 연동이 잘됨

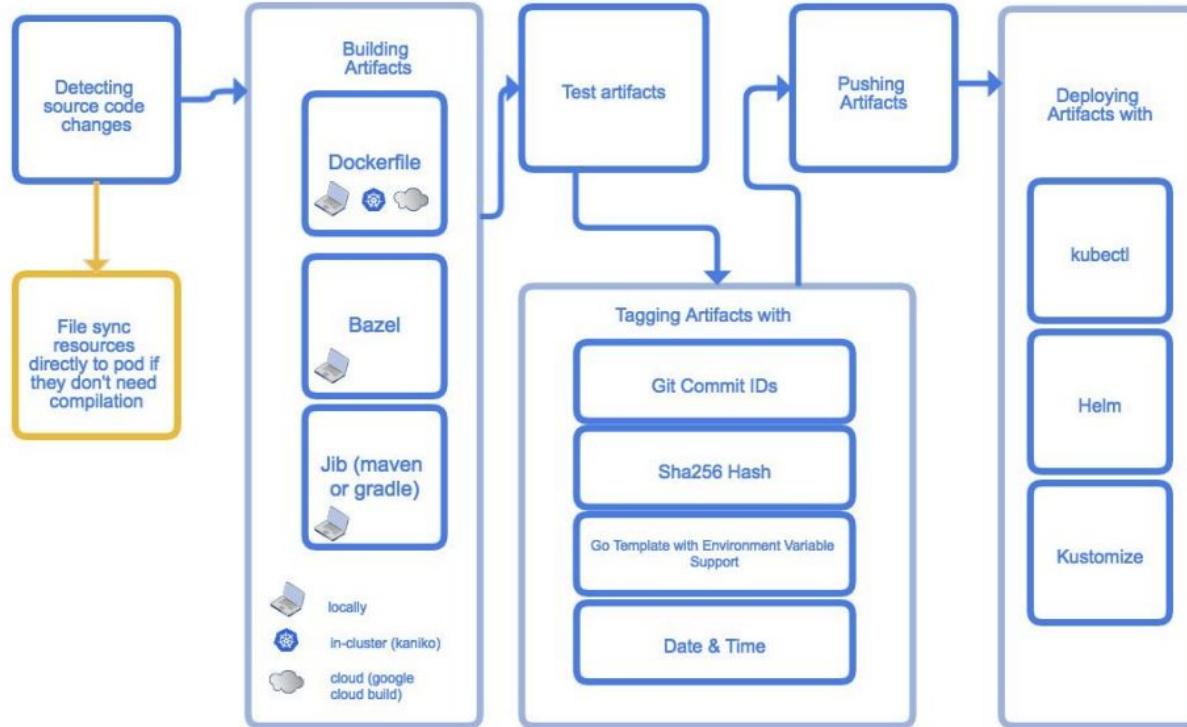


출처 : <https://argoproj.github.io/cd/>

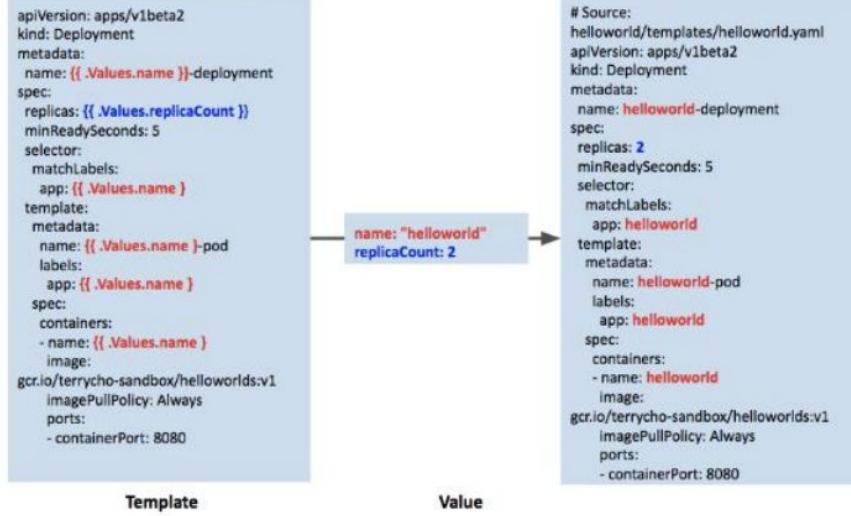
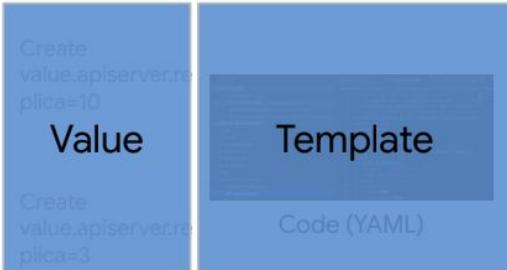
Kubernetes CD



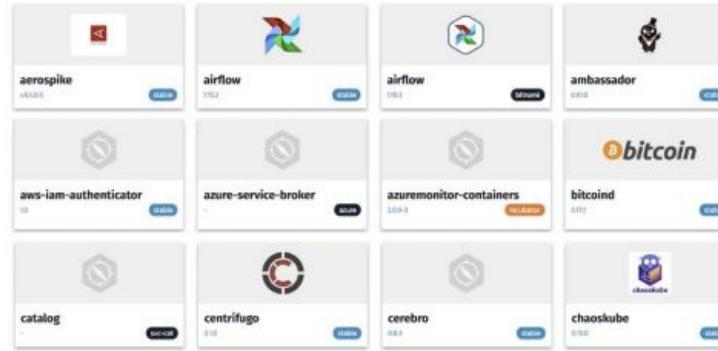
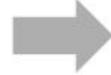
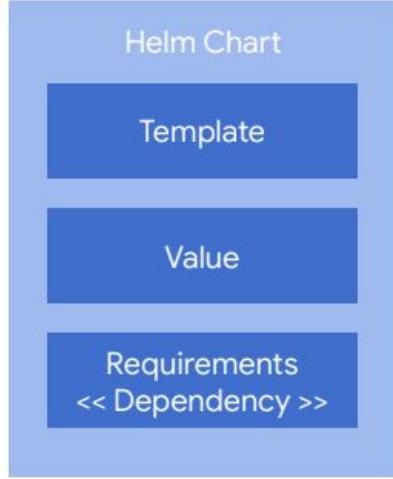
Kubernetes Skaffold



Helm



Helm Chart



MySQL install chart, redis chart, Tomcat chart, your application chart ...



감사합니다.

CI/CD 도 포함할것.

모니터링에는 zipkin, Istio 등 포함할것