

<프로젝트 기말보고서>

Sauron

AWS CloudWatch와 Lambda를 이용한 사용량 모니터링 시스템

K조 정형식, 황호성

엄현상 교수님

윤석찬 담당자님(Amazon Web Services, Technology Evangelist)

목차

1. Abstract
2. Introduction
3. Background Study
4. Goal/Problem & Requirement
5. Approach
6. Project Architecture
 - a. Architecture Diagram
 - b. Architecture Description
7. Implementation Spec
 - a. Collect
 - b. Analyze
 - c. Visualize
8. Solution
 - a. Implementation Details
 - b. Implementation Issues
9. Results
 - a. Experiments
 - b. Result Analysis and Discussion
10. Division and Assignment of Work
11. Conclusion
- [Appendix] User Manual

1. Abstract

본 프로젝트는 아마존 웹서비스(이하 AWS)가 제공하는 여러가지 클라우드 서비스를 사용해 별도의 서버 인스턴스 없이 사용자가 실제 서비스 중인 서비스 스택 구성요소의 상태를 모니터링하는 것을 목표로 한다. 특히 최근에 출시된 Lambda 서비스를 이용해서 서버 인스턴스(EC2) 구성을 하지 않고 웹페이지에 코드를 입력하는 것으로 서버의 역할을 수행하도록 한다. 또한 모니터링 데이터를 저장할 스토리지와, 모니터링 데이터를 확인하기 위한 웹 대시보드 역시 AWS의 클라우드 서비스를 이용해 정적 호스팅을 해, 사용자 측면에서 서버 관리가 필요 없는 서비스 스택 모니터링 시스템을 제작하도록 계획하였다.

2. Introduction

시스템 운영에 있어서 현재 시스템의 가동 상황을 체크하는 모니터링은 필수적이다. 일반적으로 서비스 스택은 수 대에서 많게는 수백, 수천 대의 서버로 구성되어 있는데, 각 서버들이 실시간으로 생성해내는 각종 지표를 안정적으로 수집하고 보여주기 위해서는 모니터링 시스템이 확장성 있고 고가용성을 보장하도록 구현 및 운영돼야 한다. 이는 서비스를 운영하는 기업에 있어 관리를 위한 시스템이 추가적인 장비와 관리 인력에 대한 비용이 발생하는 것을 의미한다.

AWS의 Lambda 서비스는 어플리케이션의 로직에 해당하는 코드 조각만 입력하면 서버 관리/확장/보안 이슈 없이 코드의 안정적인 동작을 보장한다. 시스템 모니터링 어플리케이션은 높은 안정성이 요구되는 반면 로직은 상대적으로 단순한 편이어서 Lambda 서비스를 이용하기 적합한 경우에 해당한다. 본 프로젝트에서는 Lambda 서비스를 이용하여 현재 사용중인 서비스들의 로그가 단기적으로 저장되는 CloudWatch에서 raw data를 수집, 가공하여 별도의 데이터베이스에 저장하도록 한다.

로그 데이터들이 저장할 스토리지는 AWS에서 최근 출시한 관리형 Elasticsearch 서비스인 'Elasticsearch Service'를 이용해서 일반적으로 로그 스토리지의 scaling 이슈를 최소화하기로 한다. 한편 Elasticsearch에 수집된 메트릭을 실시간으로 조회할 수 있는 대시보드는 별도의 웹서버 자원 없이 AWS의 파일 스토리지 서비스인 'S3'가 제공하는 정적 웹사이트 호스팅 기능을 이용하여 scaling에 따른 관리 비용을 최소화하고자 하였다. 대시보드에서는 메트릭의 기간 구간별 조회, 태그를 이용한 AWS 리소스의 그로핑 및 시각화, 기능/리소스별 메트릭의 차트 표현 등의 기능을 지원하여 로그 데이터를 사용자가 필요로 하는 최적의 형태로 빠르게 제공할 수 있도록 하였다.

3. Background Study

a. 기존 모니터링 솔루션 조사

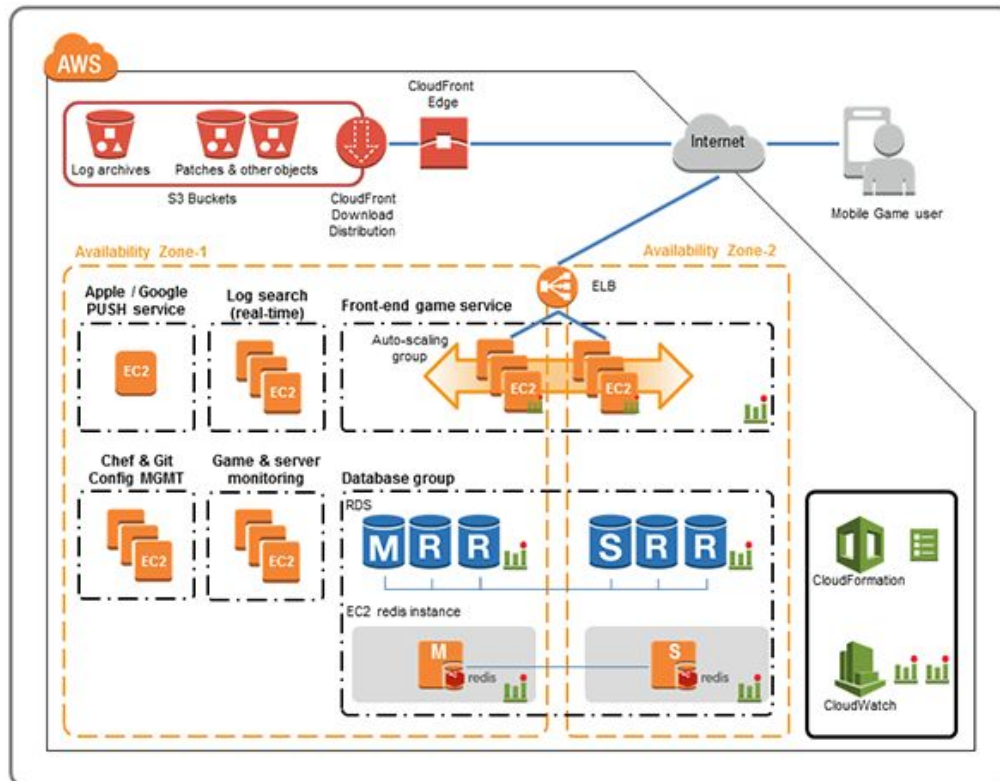
현재 시장에 존재하는 일반적인 모니터링 솔루션(nagios, cacti 등)에서는 수집하고자 하는 개별 머신 단위로 로그 수집 에이전트를 구동하여 필요한 어플리케이션 혹은 시스템 로그를 수집한 뒤, 중앙 집중적인 수집 시스템에 배치로 전송하는 형태로 시스템 지표를 수집한다. 하지만 AWS에서는 로드밸런서, 데이터베이스 등의 솔루션이 머신(가상 혹은 물리)에 설치하는 개념이 아니라 관리형 서비스의 개념으로 제공되기 때문에, 에이전트를 별도로 구동해서 로그를 수집하는 것이 불가능하다. 때문에 기존 솔루션을 이용하면 가상 서버 인스턴스인 EC2 이외의 리소스의 메트릭 수집이 불가능하다는 문제점이 있다.

클라우드 환경에서의 메트릭 수집의 어려움을 해결하기 위해 AWS에서는 사용 중인 클라우드 리소스의 지표 혹은 로그를 API로 노출해주는 CloudWatch라는 서비스를 제공하고 있다. 하지만 CloudWatch는 low-level API로, 개별 인스턴스의 지표만 확인할 수 있기 때문에 서비스 중인 기술 스택 전체 현황을 한 눈에 파악하기에는 부족함이 있다.

b. 클라우드에 배포되는 서비스 스택 구조(on AWS)

AWS 클라우드에 배포되는 어플리케이션은 보통 AWS가 제공하는 여러 서비스들의 조합으로 이루어져 있다. 아래는 AWS 사용 사례로 제시된 데브시스템즈의 아키텍처를 나타낸 다이어그램으로 트래픽을 분산해주는 ELB(로드밸런서)서비스 아래에 게임로직을 담당하는 다수의 EC2(가상서버) 인스턴스, 데이터를 저장할 RDS(관계형 데이터베이스

서비스) 등으로 구성되어 있다.



c. 프로젝트에 사용되는 AWS 서비스

인적&물적 관리 비용이 드는 서버 인스턴스의 활용 없이, 이미 고성능의 가용성이 확보된 AWS 서비스를 빌딩 블록으로 활용하여 scalability 문제를 최소화 하고자 하였고, 때문에 이용하는 AWS의 서비스의 개수를 평가 항목으로도 제시하였다. 본 프로젝트를 구성하는데 사용하기로 한 AWS 서비스와 사용함으로써 얻을 수 있는 이점은 아래와 같다.

CloudWatch: CloudWatch는 AWS 클라우드 리소스와 AWS에서 실행되는 애플리케이션을 위한 모니터링 서비스로 Amazon EC2 인스턴스, Amazon DynamoDB 테이블, Amazon RDS DB 인스턴스 같은 AWS 리소스뿐만 아니라 애플리케이션과 서비스에서 생성된 사용자 지정 지표 및 애플리케이션에서 생성된 모든 로그 파일을 모니터링할 수 있다. 수집된 데이터는 2주 동안 보관되며, 자체제공되는 CloudWatch 대시보드, Management Console, CloudWatch API등 다양한 방법으로 확인할 수 있다. 즉 서비스에서 운영되는 모든 리소스들의 시스템 지표는 CloudWatch가 자동으로 수집하고 단기간동안 저장하게 된다.

Lambda: Lambda는 이벤트에 응답하여 미리 업로드한 코드 조각을 실행하고 컴퓨팅 리소스를 자동으로 관리하는 컴퓨팅 서비스이다. 때문에 논리적인 서버단위에 대한 개념 없이 최소한의 기능 단위의 코드 실행에만 집중할 수 있도록 해준다. 코드조각의 실행을 트리거하는 이벤트로는 S3, SNS, Dynamo DB등 다른 AWS 서비스에서 발생하는 이벤트, 미리 예약된 스케줄 등이 있다. 현재 지원하는 언어로는 Node.js, Java, Python이 있다. 본 프로젝트에서는 Lambda를 이용하여 실시간으로 CloudWatch에 새로 쌓인 시스템 지표를 로그 스토리지에 저장하도록 하였다.

Elasticsearch Service: AWS Elasticsearch Service는 널리 이용되는 오픈소스 분산 검색엔진인 Elasticsearch를 AWS Cloud상에서 손쉽게 배포하고, 운영하고, 확장할 수 있게

해주는 서비스이다. Apache의 검색엔진 프로젝트인 Lucene에 기반한 Elasticsearch는 RESTful API 형태로 동작하여 다른 서비스들과 손쉽게 연동 가능하다. 또한 metric aggregation이 효율적이고 용이하여, 본 프로젝트에서는 검색 엔진의 용도보다는 로그 스토리지로 활용하였다.

S3: S3는 파일 스토리지 서비스로 고가용성의 스토리지를 저렴하고 간편하게 호스팅 할 수 있도록 해준다. 본 프로젝트에서는 대시보드 애플리케이션을 빌드해 S3에 업로드한 후 S3의 정적 웹사이트 호스팅 기능을 이용해 서버 기능 없이 호스팅하여 웹서버 관리 비용을 최소화하였다.

d. 프로젝트 개발환경

AWS Stack: Cloudwatch, Lambda, Elasticsearch Service, S3

Lambda Environment

- AWS sdk
- Elasticsearch library

Dashboard Stack

- Javascript: ES6 + Backbone.js(MVC Framework)
- Build Framework: Webpack
- Styling: SCSS(dynamic CSS language)

Collaboration Tool

- VCS: git
- Issue Tracking: Github Issue, Trello
- Communication: Slack

4. Goal/Problem & Requirements

이번 프로젝트의 목표는 (1)클라우드 자원에 최적화된, (2)관리비용이 최소화된, (3)실시간 모니터링과 장기간 로그조회가 모두 가능한 모니터링 시스템을 구축하는 것이다.

5. Approach

프로젝트 목표의 세가지 항목을 각각 다음과 같은 방법으로 접근한다.

(1) 클라우드 자원에 최적화된 모니터링 시스템

클라우드 상에 배포된 서비스 스택은 캐쉬, 데이터베이스, 서버 등 여러 종류의 서비스들이 인증, 결제, 로깅 등 서로 다른 목적을 위해 그룹화 되어있다. 이에 본 프로젝트에서는 기능에 따른 태그를 이용해서 여러 리소스들을 논리 그룹으로 나누어서 모니터링 할 수 있도록 한다.

(2) 관리비용이 최소화된 모니터링 시스템

CloudWatch로 부터 로그를 수집, 가공하는 로직은 Lambda, 데이터 저장은 Elasticsearch Service, 대시보드 어플리케이션 배포는 S3 등 시스템의 모든 부분을 AWS의 클라우드 서비스를 이용하여 serverless하게 구축, 추가적인 관리 없이 확장성과 안정성이 담보될 수 있도록 한다.

(3) 실시간 모니터링과 장기간 로그조회가 모두 가능한 모니터링 시스템

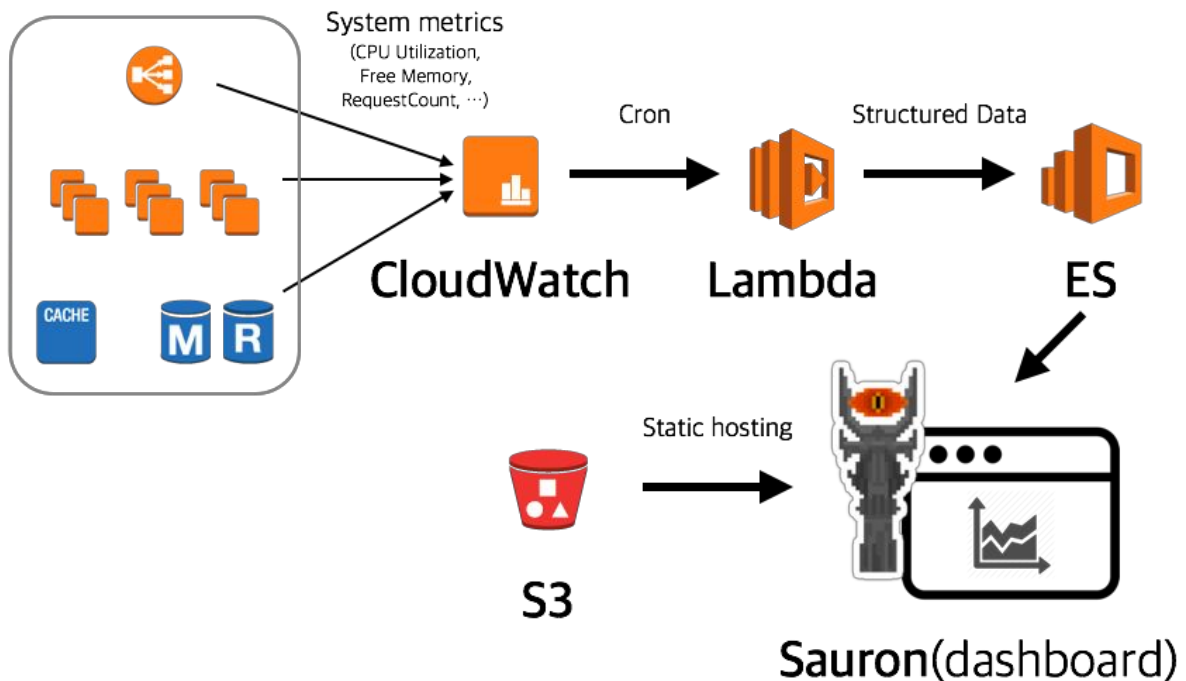
브라우저 상에서 동작하는 대시보드 어플리케이션은 Elasticsearch와 주기적으로 통신해서 계속해서 최신의 metric들을 보여줄 수 있도록 한다. 또한, 기간별 조회를 위한

별도의 인터페이스를 구축하고 Elasticsearch에 저장된 데이터의 보존기간을 늘려서 Cloudwatch의 한계를 보완한다.

6. Project Architecture

A. Architecture Diagram

Authentication Service



B. Architecture Description

본 프로젝트는 크게 COLLECT, ANALYZE, VISUALIZE 세 단계로 구성되어있다.

우선 COLLECT단계에서는 Cloudwatch에 저장되어있는 시스템 metric들을 Lambda 코드를 이용해서 가져온다. Lambda코드는 Javascript(node.js)코드를 이용하여 작성하며, node.js용 AWS SDK를 이용해서 sdk가 제공하는 객체를 통해 정보를 받아올 수 있도록 한다. 이때 정보를 받아와야할 인스턴스들과 필요한 메트릭은 Elasticsearch에 미리 저장된 데이터를 참고하도록 한다. Lambda코드의 trigger event는 'scheduled event'로 하여 현재 지원하는 최소 주기인 5분마다 최신 정보를 수집한다.

두번째 ANALYZE단계에서는 앞서 ANALYZE단계에서 수집한 raw data(json형태)를 본 프로젝트에서 지정한 포맷으로 변환하고, data의 출처에 따라 태그 정보를 더한 후 Elasticsearch에 저장한다.

마지막 VISUALIZE단계는 앞서 두 단계를 거쳐 Elasticsearch에 저장된 정보를 불러와 사용자에게 보여주는 시각화 부분과 함께 본 시스템에서 조회할 인스턴스를 정해주는 설정 부분을 담당한다. 시각화 부분은 크게 실시간 모니터링과 과거 기록 조회로 나눌 수 있으며 사전에 등록된 태그에 따라 그룹화해주는 기능을 제공한다. 설정 부분에서는 현재 모니터링

중인 인스턴스의 목록을 확인/업데이트 할 수 있으며, 변경 사항은 Elasticsearch의 instance 인덱스를 통해 COLLECT단계와 ANALYZE 단계에 반영된다.

7. Implementation Spec

A. Collect

Collect단계는 lambda의 cron기능이 5분마다 발생시키는 event에 따라 동작하는 handler(일종의 메인 함수)와 handler가 실행시키는 함수들로 구성된다. handler는 Elasticsearch에서 수집할 대상을 조회하고(getServices, getInstances), 받아온 목록에 따라 CloudWatch에서 로그를 가져온다(getCloudWatchLogs). 모니터링 대상이 되는 서비스와 관련 메트릭은 다음과 같다.

이 때 조회하는 AWS 서비스의 목록은 다음과 같다.

- ELB(Elastic Load Balancing)
- EC2(Elastic Compute Cloud)
- RDS(Relational Database Service)

서비스별로 조회하는 Metric은 다음과 같다. Metric은 인스턴스의 역할별로 장애 상황이나 이용량을 표현하는데 도움이 되는 메트릭을 다음과 같이 선정하였다.

- ELB(Elastic Load Balancing)
 - *HealthyHostCount*: 해당 로드밸런서에 속한 서버 중 동작하고 있는 서버의 수
 - *UnHealthyHostCount*: 해당 로드밸런서에 속한 서버 중 동작하지 않고 있는 서버의 수. 해당 메트릭이 높아지면 장애 발생의 우려가 높음을 의미한다.
 - *RequestCount*: 로드밸런서가 처리하고 있는 단위시간당 리퀘스트의 수. 현재 로드밸런서에 걸리고 있는 부하를 나타내는 지표다.
 - *Latency*: 리퀘스트에 대한 응답이 나갈 때 까지 걸리는 지연시간을 의미하며, 지연시간이 길어지면 어플리케이션 서버나 데이터베이스 등에서 병목현상 혹은 장애가 발생했을 확률이 있다.
- EC2(Elastic Compute Cloud)
 - *CPUUtilization*: 해당 서버의 cpu 사용량. 해당 지표가 높아지면 병목이 발생하거나 요청이 증가했을 확률이 높다.
 - *NetworkIn*: 해당 서버로 유입되는 네트워크 트래픽량. 해당 지표가 높아지면 요청이 급증했음을 의미한다. 외부적 요인에 의해 이용량이 급증했거나 DDos 공격 등을 당했을 확률이 있다.

- *NetworkOut*: 해당 서버에서 바깥으로 나가는 네트워크 트래픽량. 해당 지표가 높아지면 요청이 급증했음을 의미한다. 외부적 요인에 의해 이용량이 급증했거나 DDos 공격 등을 당했을 확률이 있다.
-
- RDS(Relational Database Service)
 - *CPUUtilization*: 해당 데이터베이스 인스턴스의 cpu 사용량.
 - *FreeableMemory*: 해당 데이터베이스의 여유 메모리량.
 - *ReadThroughput*: 디스크에서 초당 read하고 있는 평균 byte.
 - *WriteThroughput*: 디스크에 초당 write하고 있는 평균 byte.
 - *SwapUsage*: 해당 데이터베이스에서 발생하고 있는 스왑 공간의 크기.

B. Analyze

Analyze는 Collect에서 받아온 metric들에 태그 정보를 추가하여 Elasticsearch에 저장할 형태로 변환한 후(metricsToIndexedJSON), 이들을 묶어서 한번에 bulk index로 Elasticsearch에 저장한다(bulkIndex). 이때 Elasticsearch에 저장되는 데이터 포맷은 다음과 같다.

```
{
  InstanceID: Number,
  MetricName: String,
  TimeStamp: Date, // Javascript Date Object
  Value: Number
}
```

C. Visualize

Visualize는 Sauron에 등록된 인스턴스들을 관리하고, 사용자가 지정한 기간에 따라 차트를 그려준다.

1. 인스턴스 관리(getServices, getInstances, renewInstances)
인스턴스들은 각각이 맡은 기능(결제, 인증 등)에 따라 Service에 속하면서 동시에 각각의 기능에 따라 Function값을 갖는다.

각각의 서비스는 다음과 같은 값을 갖는다.

- Name: 서비스의 이름
- Description: 사용자가 작성한 비고

각각의 인스턴스는 다음과 같은 정보를 포함한다.

- instanceId: AWS에서 리소스를 지칭하는 유니크한 identifier.
- tags: 해당 인스턴스에 등록된 태그 object로, { tagName: tagValue }의 목록.
- function: 해당 인스턴스의 역할로, AWS 서비스별로 function값이 부여되며, 그 규칙은 다음과 같다.

- ELB: *LB*(로드밸런서)
- EC2: *WAS*(웹 어플리케이션 서버)
- RDS: *DB*(관계형 데이터베이스)
- metricNames: 해당 인스턴스가 수집하는 메트릭 종류로, 인스턴스의 역할별(function)로 장애상황이나 이용량을 표현하는 데 도움이 되는 메트릭을 다음과 같이 선정하였다.

2. Chart Visualization

차트를 각 인스턴스의 Function에 따라, 혹은 특정 개별 인스턴스의 차트를 그릴 수 있도록 한다.(drawMetricChartByFunction, drawMetricChartByInstance). 이 두 함수는 getMetrics라는 공통의 함수를 사용해서 Elasticsearch에서 자료를 받아온다. 차트 라이브러리는 chart.js를 사용하도록 한다.

8. Solution

A. Implementation Details

1. Collector

index.js - 이벤트가 발생했을 때 데이터를 가져와야 할 목록을 받고, Elasticsearch에 업로드하는 핵심모듈

exports.handler	event를 받아 EC2, RDS, ELB 각각 조회할 타입을 수집하고, 그 타입들을 cloudwatch에서 조회해서 인덱싱함
getInstanceMetrics	실제 데이터 조회와 인덱싱이 이루어지는 함수. exports.handler를 통해서 실행
dimensionParams	인스턴스의 타입에 따라 dimensionParams를 완성
instanceId	인스턴스 고유의 id를 EC2, RDS, ELB 각각의 경우에 따라 반환
filteredParamList	타입에 맞는 패러미터만 남은 paramsList를 반환
makeActions	ElasticSearch에 bulk API로 인덱싱할 메트릭 패러미터를 생성
constructRDSInstanceARN	RDS Instance의 arn(Amazon Resource Name)을 만들어줌
upsertResourceQuery	ElasticSearch에 Resource를 Indexing할 쿼리를 반환
fetchEC2Resources	ElasticSearch로부터 조회해야할 EC2 Instance 목록을 가져옴
fetchRDSResources	ElasticSearch로부터 조회해야할 RDS Instance 목록을 가져옴
fetchELBResources	ElasticSearch로부터 조회해야할 ELB Instance 목록을 가져옴

metricList.js

리소스 타입별로 조회해야할 지표를 정의한 json객체

runLocally.js

Lambda에 코드를 배포하지 않고 로컬환경에서 index.js의 handler에 event와 context를 제공해서 테스트하기 위해 사용되는 스크립트

pack.sh

코드를 Lambda에 손쉽게 배포하기 위해 Lambda 코드베이스를 압축하는 스크립트

2. Dashboard

App 대쉬보드 전역에서 공유될 객체들을 정의하는 모듈

var app	전체에서 공유할 app객체.
var esClient	elasticsearch의 node.js용 모듈에서 elasticsearch 클라이언트 생성해서 singleton 스타일로 사용할 수 있도록 함.

index.js 대쉬보드가 실행되는 일종의 main script

app.globalEvents	앱 어디서나 이벤트를 주고받을 수 있도록하는 버스
app.appView	앱의 현재 뷰 상태
AppRouter	앱의 라우팅을 담당하는 라우팅모듈

ESSync ES의 인덱스를 저장소로 활용하여 모델, 컬렉션을 sync해주는 모듈로. Backbone에서 기본적으로 제공하는 REST API 인터페이스를 가정하고 작성된 Backbone.sync 모듈을 대체하였다.

case 'read'	모델의 indexName과 typeName에 따라 ES를 조회
case 'create'	모델의 indexName과 typeName에 따라 새로운 document를 삽입

AppRouter

app.js 앱의 라우팅을 담당하는 backbone의 라우터 모듈

constructor(options)	모듈의 생성자. 주소에 해당하는 결과들을 정의
setView(view)	새로운 뷰를 그리고, app.currentView에 저장하는 helper 함수
root()	초기화면을 띄우는 함수
setting()	세팅화면을 띄우는 함수
charts()	차트화면을 띄우는 함수
singleChart(id)	'id'에 해당하는 차트 하나를 크게 띄우는 함수

<models>

class **ESModel** ES와 연동가능한 형태의 Backbone 모델

get idAttribute ()	모델의 idAttribute의 getter. ES6의 getter 기능을 활용
get indexName ()	모델의 indexName의 getter. ES6의 getter 기능을 활용
get typeName ()	모델의 typeName의 getter. ES6의 getter 기능을 활용
ensureIndex()	모델의 indexName이 ES에 존재하는지를 확인
createIndex()	모델의 indexName을 ES에 생성
removeIndex()	모델의 indexName을 ES에서 제거
sync()	모델의 현재 상태를 해당하는 ES에 반영

class **ESCollection** ESModel의 Backbone Collection

get indexName ()	컬렉션의 indexName의 getter
get typeName ()	컬렉션의 typeName의 getter
sync ()	컬렉션의 현재 상태를 해당하는 ES에 반영
parse(resp)	ES의 response를 쓰기 쉬운 상태로 변형

class **AWSResourceModel** AWSResource를 나타내는 ESModel

idAttribute	id를 어떻게 사용할지 지정하는 값
indexName	ES상의 이 모델의 index명

class **AWSResourceCollection** AWSResource의 Collection

indexName	AWSResourceModel의 indexName과 동일
typeName	AWSResourceModel의 typeName과 동일

class **EC2ResourceModel** EC2를 위한 AWSResourceCollection

get typeName ()	'ec2'를 typeName으로 반환하는 getter
-----------------	-------------------------------

class **EC2ResourceCollection** EC2를 위한 AWSResourceCollection

get typeName ()	'ec2'를 typeName으로 반환하는 getter
-----------------	-------------------------------

class **ELBResourceModel** ELB를 위한 AWSResourceCollection

get typeName ()	‘elb’를 typeName으로 반환하는 getter
-----------------	-------------------------------

class **ELBResourceCollection** ELB를 위한 AWSResourceCollection

get typeName ()	‘elb’를 typeName으로 반환하는 getter
-----------------	-------------------------------

class **RDSResourceModel** RDS를 위한 AWSResourceCollection

get typeName ()	‘rds’를 typeName으로 반환하는 getter
-----------------	-------------------------------

class **RDSResourceCollection** RDS를 위한 AWSResourceCollection

get typeName ()	‘rds’를 typeName으로 반환하는 getter
-----------------	-------------------------------

class **MetricModel** 차트로 그릴 두 있는 형태의 데이터를 담은 Model

fetch()	ES에서 원하는 데이터를 가져와서 chart.j 데이터 형식으로 저장하는 함수
---------	---

class **MetricCollection** metric의 Collection

class **ServiceModel** 서비스(EC2Resource, ELBResource, RDSResource의 묶음)의 ESModel

constructor()	세가지 리소스의 컬렉션을 ec2Instances, rdsInstances, elbInstances로 가진다.
fetchAWSResources()	EC2, RDS, ELB resource를 fetch한다
fetchEC2Resources()	‘name’에 해당하는 ec2 instance들을 ec2Instances에 담음
fetchRDSResources()	‘name’에 해당하는 rds instance들을 rdsInstances에 담음
fetchELBResources()	‘name’에 해당하는 elb instance들을 elbInstances에 담음

class **ServiceCollection** serviceModel의 ESCollection

<data>
getLogs

지정한 타입의 지정한 메트릭을 기간에 따라 조회하고, 차트에 뿌리기 적합한 형태로 aggregate해주는 모듈

var fromString	SQL에서 FROM에 해당하는 부분을 ES의 date format으로 지정 eg) "now-30m"
var toString	SQL에서 TO에 해당하는 부분을 ES의 date format으로 지정 eg) "now"
var binString	aggregate할 단위를 "{ { n } }m" 형태로 지정
bucket2LabelValue()	ES쿼리에 대한 응답을 차트에 넣을 label과 value형태로 변환

chartjsDataConverter

label과 value를 받아 색상, 옵션까지 더해진 chartjs 데이터로 만들어주는 함수

<views>

AppView

등록된 서비스를 확인, 추가하는 뷰

initialize	뷰를 초기화하고 현재 등록되어있는 서비스의 목록을 ES에서 가져옴
openNewService	새로운 서비스 등록화면을 띄움
setService	현재 모니터링 중인 서비스를 변경
updateDuration	현재 차트가 나타내는 범위를 바꿈

MetricView

차트 하나의 뷰

initialize	뷰를 초기화, 데이터를 가져오고 기타 event에 bind
render	차트 명과 타입을 그려줌
drawChart	뷰의 로딩이 완료되면 chartjs를 이용해서 차트를 그림
newCtx	새로운 차트를 그리기 위해 캔버스 DOM을 삭제하고 새로운 캔버스의 ctx를 반환
updateChart	duration이 바뀔때 따라 모델을 바꿈

MetricsView

metric 뷰 여러개를 그림

initialize	뷰를 초기화. 인스턴스 종류별 항목들의 차트를 꼭 그려줌
metricNames	인스턴스별로 차트를 그려야 할 metric을 반환

NewServiceView

render	템플릿을 \$el에 그려줌
open	새로운 서비스를 추가하는 모달을 띄움
close	새로운 서비스를 추가하는 모달을 닫음
create	새로운 서비스를 저장

ResourceView

차트를 띄울 인스턴스를 정하는 버튼의 뷰

initialize	뷰의 초기화
render	버튼을 그림

ResourcesView

차트를 띄울 인스턴스를 정하는 버튼을 여러 개 그려주는 뷰

initialize	뷰의 초기화
addAll	resource 버튼을 여러 개 그림

B. Implementation Issues

1. Collect

Collect의 핵심은 별도의 등록절차 없이, AWS Console에서 등록한 태그만으로 Sauron에서 모니터링할 서비스와 그 아래의 인스턴스를 할 수 있게 하는 것이었다. 이를 위해 EC2, ELB, RDS 각각의 sdk를 이용해서 현재 사용자의 계정 아래에 등록되어있는 전체 인스턴스 중 본 프로젝트에서 사용하는 'instanceId' 항목을 태그로 가지고 있는 인스턴스들의 목록을 수집하였다.

2. Dashboard

Dashboard는 관리 cost를 최소화하기위해 정적 웹페이지 형태로 제작되어야했고, 의존성 관리 및 배포의 편의를 위해 webpack으로 컴파일한 최종 결과물을 S3에 업로드했다. 기간별, 서비스별 등 다양한 형태의 data를 가공할 때 elasticsearch의 queryDSL을 최대한 이용해서 query하여 유지보수의 용이성과 성능 모두를 충족시킬 수 있었다.

9. Results

본 프로젝트의 결과물을 이번 프로젝트의 목표였던

1. 클라우드 자원에 최적화된,
2. 관리비용이 최소화된,
3. 실시간 모니터링과 장기간 로그조회가 모두 가능(성능지표)

에 비추어 살펴보았다.

1. 클라우드 자원에 최적화

로그 수집 대상, 로그 수집, 로그 확인 등 전체 flow가 모두 클라우드 상에서 이루어지는 cloud-native 아키텍처를 완성하였다.

2. 관리비용이 최소화

로그가 1차적으로 수집되는 Cloudwatch, 로그를 수집, 가공하는 Lambda, 로그를 저장, 인덱싱하는 Elasticsearch, 그리고 대쉬보드를 호스팅하는 s3 전부 아마존에서 제공하는 서비스이기 때문에 관리비용이 최소화되었으며, 확장성 면에서도 훨씬 유연한 시스템을 구현하였다.

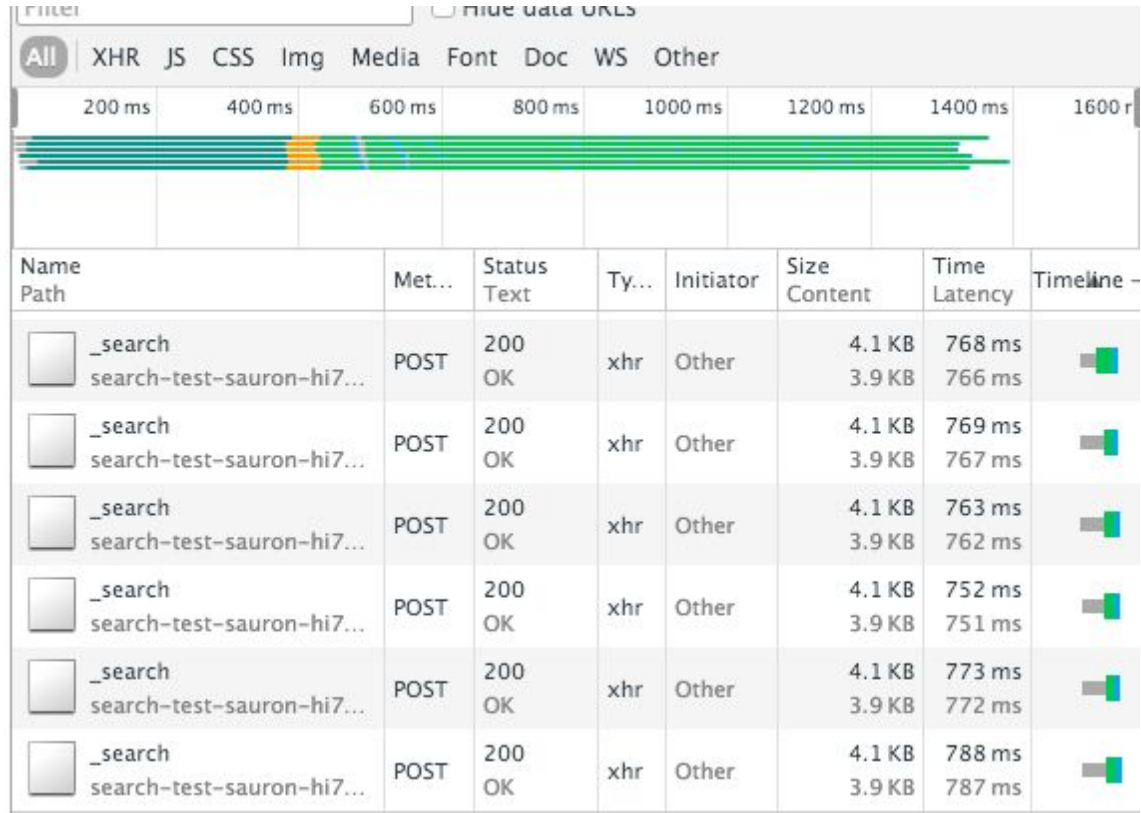
3. 성능 지표 만족(실시간 모니터링과 장기간 로그 조회가 모두 가능)

해당 requirement를 테스트하기 위해 랜덤하게 메트릭을 생성해서 Elasticsearch에 인덱싱을 하는 부하테스트 스크립트를 작성하였다. 부하는 약 10개의 리소스에서 16종의 시스템 지표를 1분 단위로 생성해내는 것을 1달동안 하는 것을 상정하였다.

```
{
  - _shards: {
    total: 10,
    successful: 10,
    failed: 0
  },
  - _all: {
    - primaries: {
      - docs: {
        count: 2767680,
        deleted: 0
      },
      - store: {
        size_in_bytes: 306639988,
        throttle_time_in_millis: 74007
      },
      - indexing: {
        index_total: 2767680,
        index_time_in_millis: 272879,
        index_current: 0,
        delete_total: 0,
        delete_time_in_millis: 0,
        delete_current: 0,
        noop_update_total: 0,
        is_throttled: false,
        throttle_time_in_millis: 0
      },
    },
  },
}
```

위 그림은 1달치의 로그를 모두 인덱싱한 이후의 ES 인덱스 크기를 나타내며, 약 276만건의 raw 메트릭 데이터가 300MB 가량을 차지함을 확인할 수 있다. 로그 스토리지의 스펙은 이를 기준으로 산정할 수 있는데, 예를들어 100여대의 인스턴스로 구성된 시스템의 로그를 수집할 경우 1달 기준 3GB의 스토리지가 필요함을 알 수 있다. ES의 경우 인덱스를 동적으로 분리할 수 있으며, 인덱스를 언제든지 백업 및 복원할 수 있으므로 스토리지의 한계로 오래된 메트릭 인덱스를 아카이브했다고 하더라도 필요한 경우 복원하여 해당 시점의 지표를 확인하는 것이 용이하다.

사우론 대시보드에서는 기본적으로 최근 30분, 6시간, 하루, 1주일, 1달 단위로 기간을 변경해가며 지표를 조회할 수 있으며, 그래프 출력은 지정한 기간을 대략 30개 정도의 datapoint로 bucketing을 하여 이뤄진다. 실제로 조회기간을 1주일, 1달 등으로



변경해가며 테스트해도 1000ms, 즉 1초 내외의 시간 안에 14개의 메트릭에 대한 지표 모두 갱신이 완료됨을 확인할 수 있었다.

10. Division & Assignment of Work

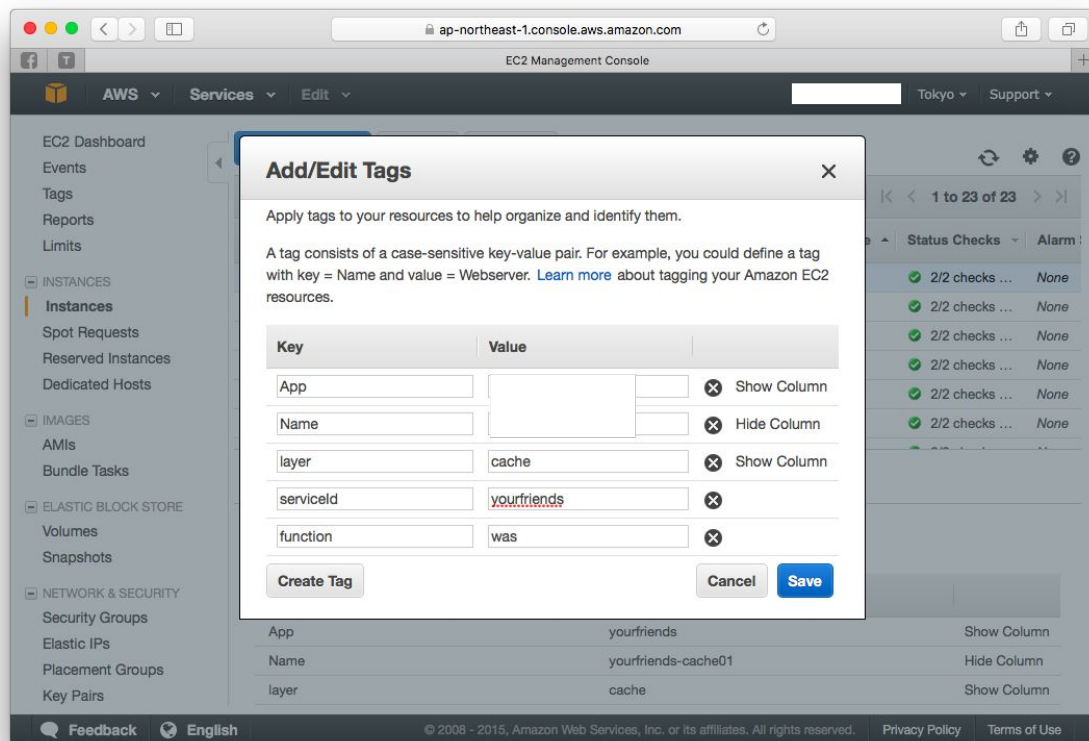
구분	항목	담당자
Collect	CloudWatch 알림 조건 설정 및 SNS 연동	황호성
	Lambda로 위 SNS를 구독 & metric을 파싱	황호성
Analyse	저장할 metric structure 정의	정형식
	Lambda로 metric aggregation	정형식
Visualize	Dashboard Bootstrapping	황호성
	Write DynamoDB Queries /w aggregation	정형식
	Chart Visualisation	정형식

[Appendix.A] User Manual

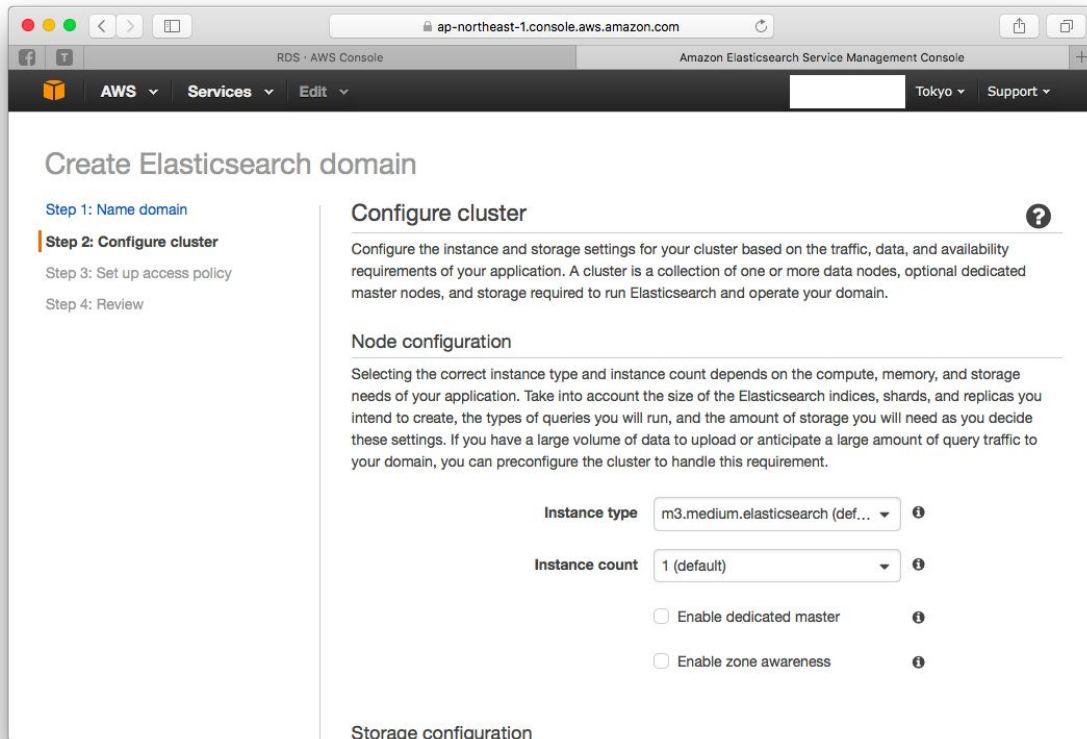
<Sauron>은 사용자의 AWS 계정 아래에서 동작하는 서비스들의 로그를 별도의 솔루션 설치 없이, 태그 추가와 AWS서비스 등록만으로 손쉽게 확인할 수 있게해주는 솔루션입니다.

이용방법

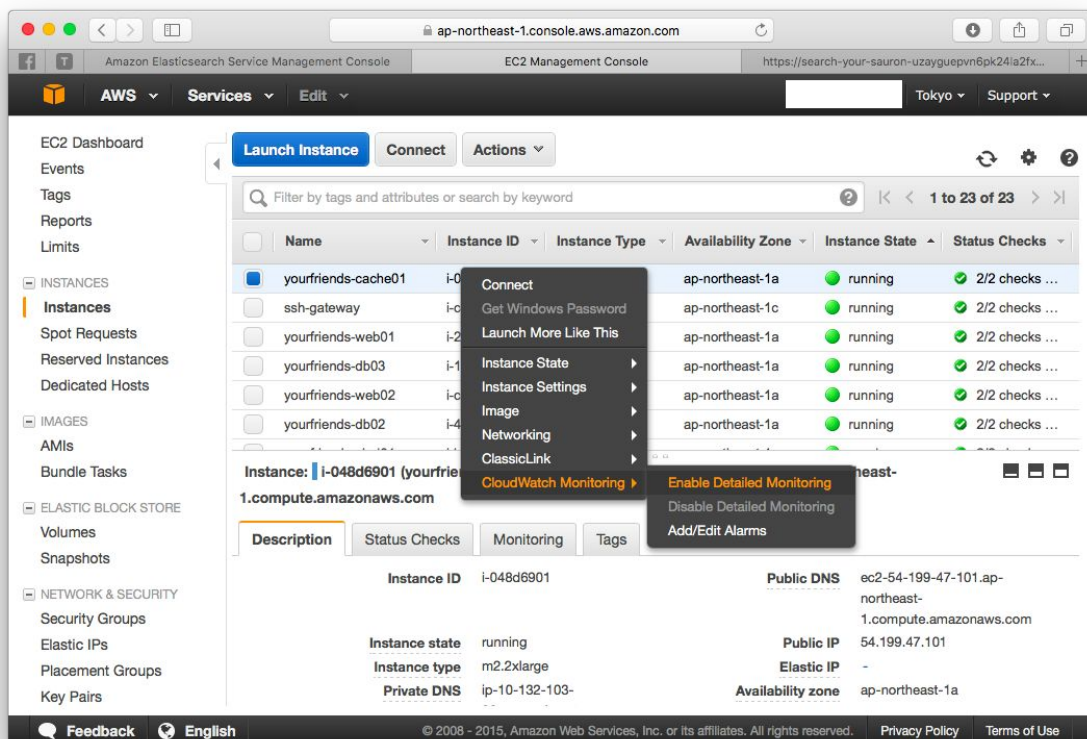
1. AWS 콘솔을 이용해서 인스턴스들에 태그를 등록합니다. 사우론은 'serviceld'항목을 기준으로 서비스를 구분합니다.



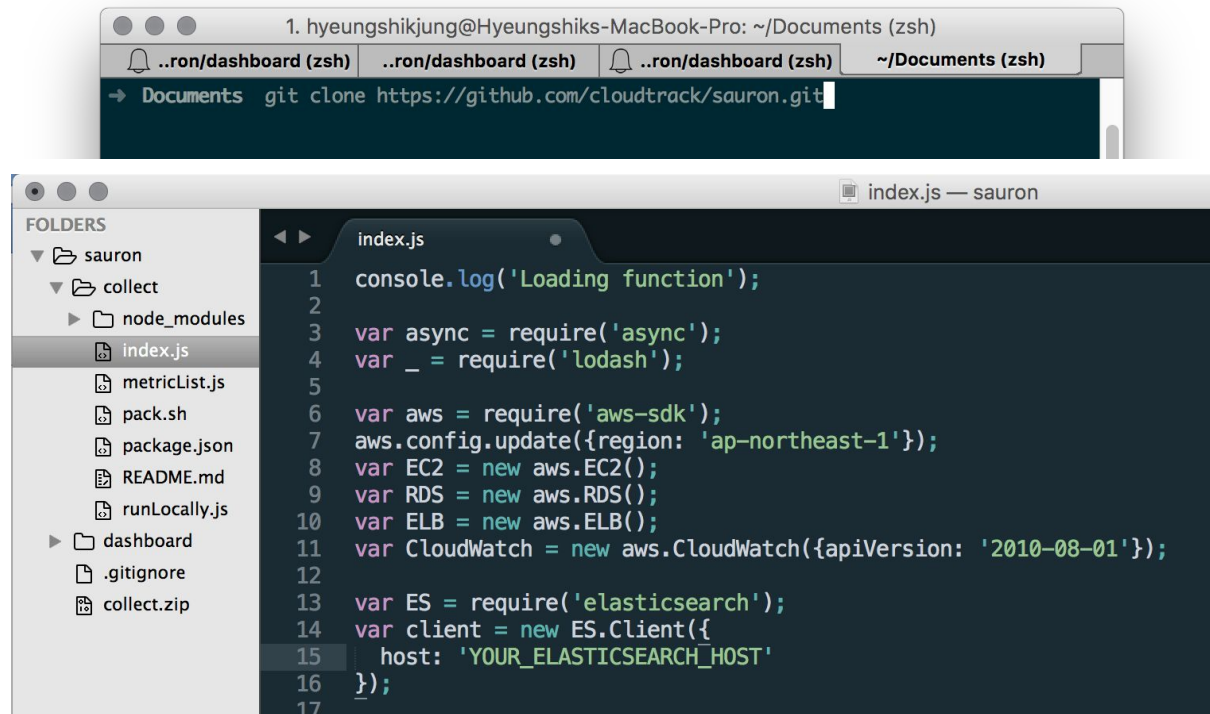
2. Elasticsearch 인스턴스를 생성합니다.



3. 로그를 1분 주기로 수집하기 위해 AWS Console에서 'Detailed Monitoring'을 활성화합니다.



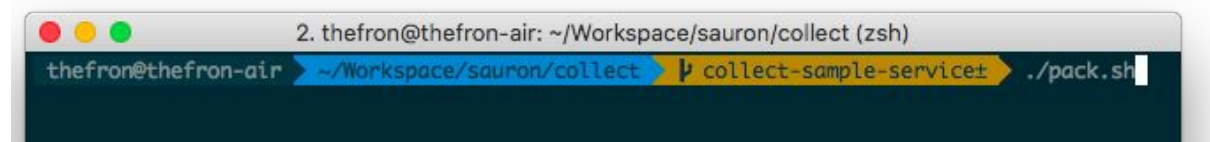
4. 사우론 repository를 클론받고, collect/index.js의 Elasticsearch 주소부분을 방금 만든 인스턴스의 주소로 대체합니다.



The image shows two screenshots. The top screenshot is a terminal window with the command `git clone https://github.com/cloudtrack/sauron.git` being executed. The bottom screenshot is a code editor showing the file `index.js` in the `sauron` project. The file contains JavaScript code for initializing AWS services and Elasticsearch. Line 15 shows the `host` property of the `ES.Client` constructor set to `'YOUR_ELASTICSEARCH_HOST'`.

```
1 console.log('Loading function');
2
3 var async = require('async');
4 var _ = require('lodash');
5
6 var aws = require('aws-sdk');
7 aws.config.update({region: 'ap-northeast-1'});
8 var EC2 = new aws.EC2();
9 var RDS = new aws.RDS();
10 var ELB = new aws.ELB();
11 var CloudWatch = new aws.CloudWatch({apiVersion: '2010-08-01'});
12
13 var ES = require('elasticsearch');
14 var client = new ES.Client({
15   host: 'YOUR_ELASTICSEARCH_HOST'
16 });
17
```

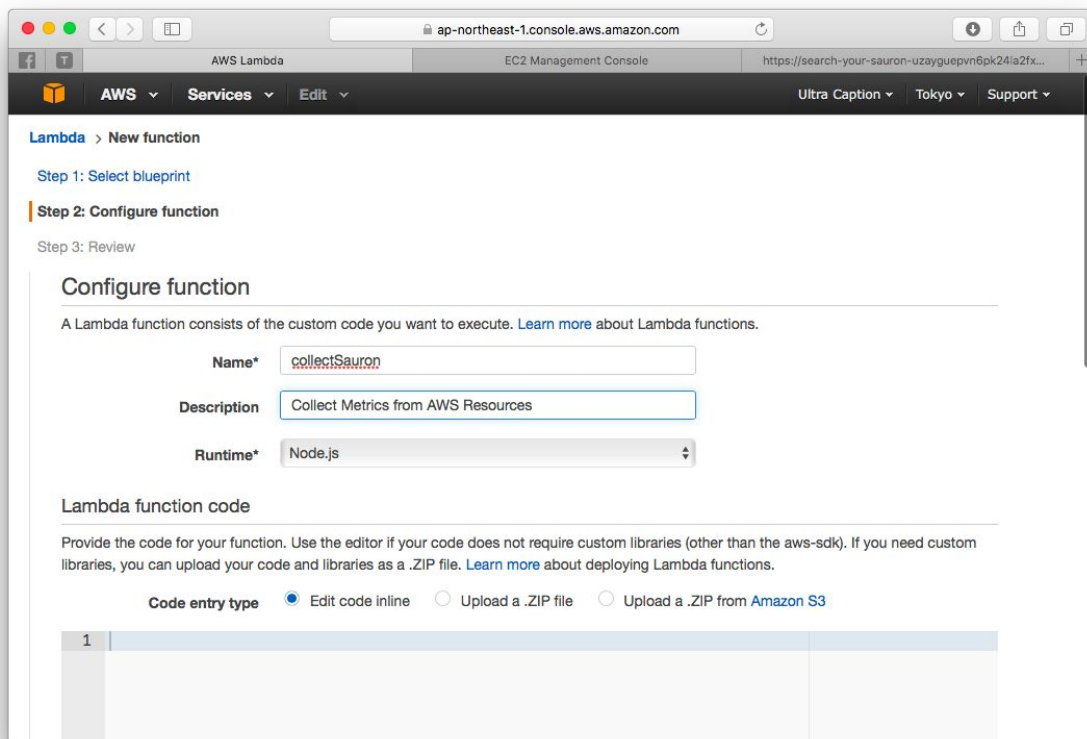
5. 동봉된 pack.sh을 이용해서 collect를 압축합니다.



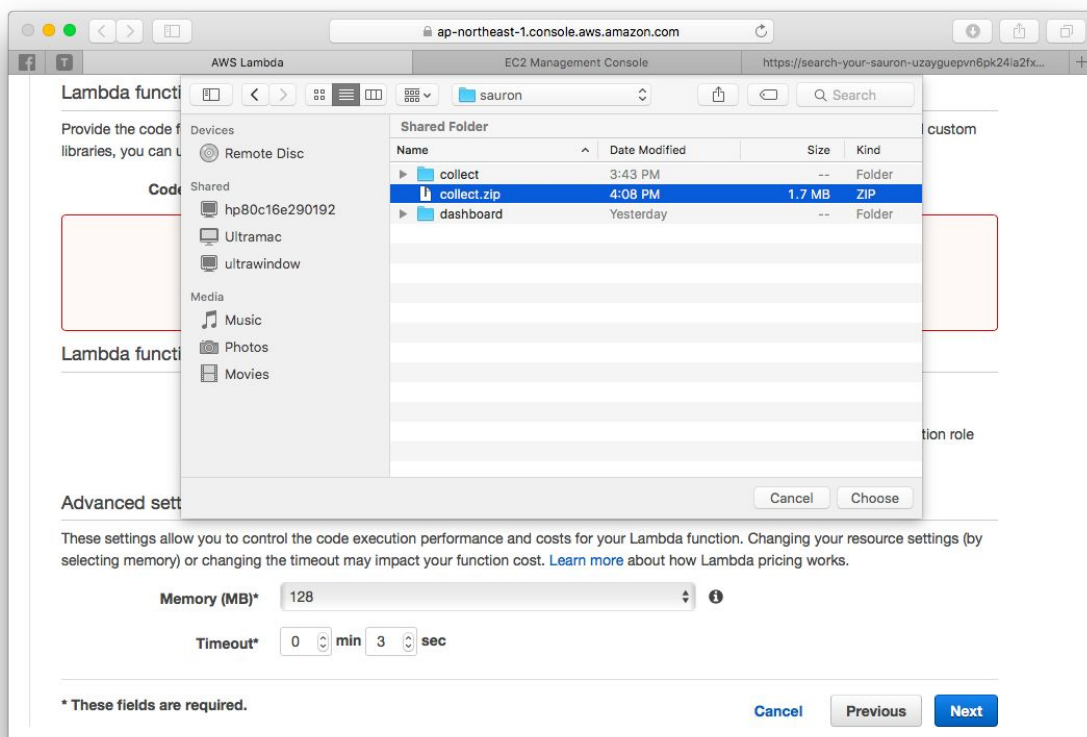
The image shows a terminal window with the command `./pack.sh` being executed in the `~/Workspace/sauron/collect` directory. The prompt shows the user is `thefron` on the `thefron-air` machine.

```
thefron@thefron-air ~/Workspace/sauron/collect$ ./pack.sh
```

6. AWS 콘솔의 람다에 들어가서 새로운 함수를 만듭니다.



7. pack.sh로 만들어진 collect.zip을 업로드합니다



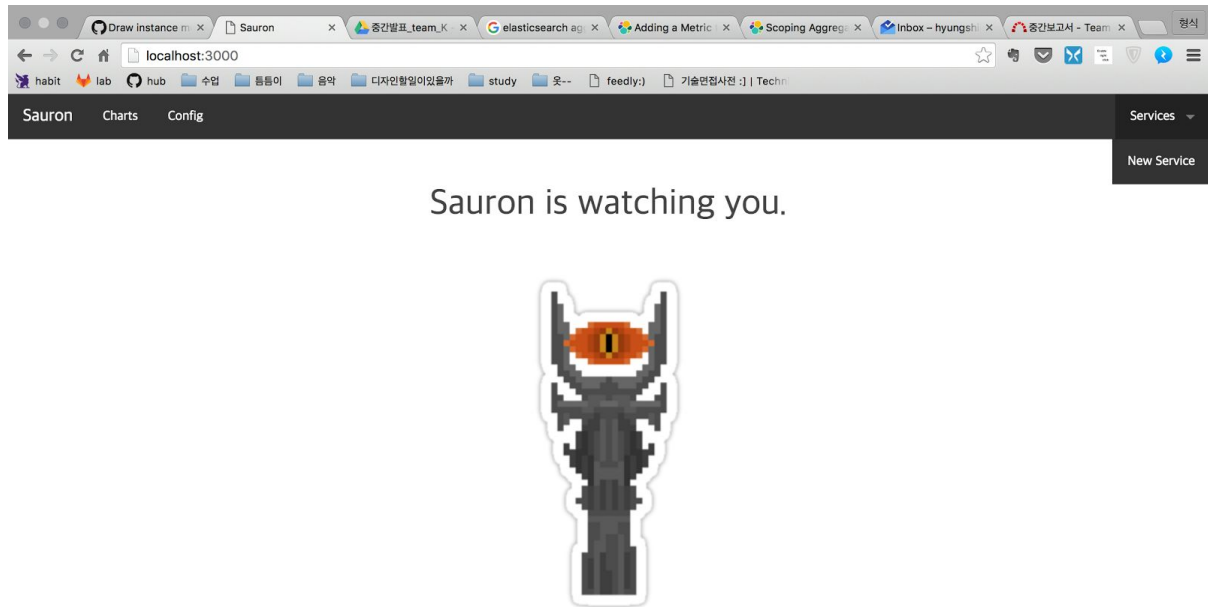
8. role(Lambda function의 권한)은 AWS에서 제공하는 lambda_basic_execution, timeout(함수의 최대 실행시간)은 30초로 설정합니다.

The screenshot shows the AWS Lambda console configuration page. Under 'Lambda function handler and role', the 'Handler' is set to 'index.handler' and the 'Role' is set to 'lambda_basic_execution'. Below this, a note states: 'Ensure that popups are enabled to create a new role. [Learn more](#) about Lambda execution roles.' The 'Advanced settings' section explains that these settings control code execution performance and costs. It shows 'Memory (MB)' set to 128 and 'Timeout' set to 0 min and 30 sec. At the bottom, there are 'Cancel', 'Previous', and 'Next' buttons, and a note: '* These fields are required.'

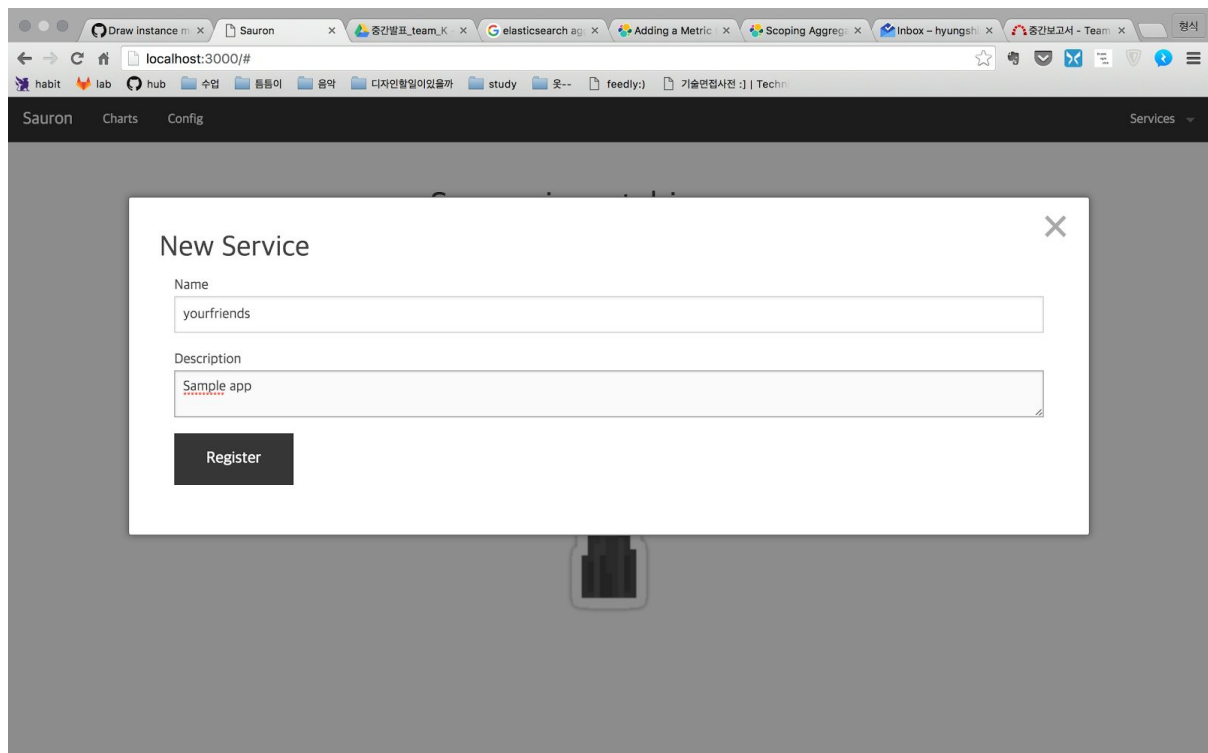
9. Lambda의 event source 타입은 'Scheduled Event', 주기는 '5분'으로 설정해줍니다

The screenshot shows the 'Add event source' dialog box in the AWS Lambda console. It prompts the user to configure the Lambda function to respond to events from the listed sources. The 'Event source type' is set to 'Scheduled Event'. The 'Name' is 'metric-cron', the 'Description' is 'Trigger periodically', and the 'Schedule expression' is 'rate(5 minutes)'. A note states: 'Lambda will add the necessary permissions to invoke your Lambda function on a schedule. [Learn more](#) about the Lambda permissions model.' At the bottom, there are radio buttons for 'Enable now' (selected) and 'Enable later', and 'Cancel' and 'Submit' buttons.

10. dashboard를 띄우면 현재 아무런 서비스도 등록되지 않은 상태입니다.



11. 'New Service'를 클릭해서 서비스를 등록합니다.



12. 등록한 서비스의 메트릭을 확인할 수 있습니다.

