



Dive into AWS Lambda

Limber up - What is AWS Lambda?

Serverless and AWS Lambda

Working with AWS Lambda

AWS Lambda Use-Case

Deep Dive 0 - How Lambda works (in the view of AWS)

Architecture of AWS Lambda

Invocation Types

Execution Environment

Deep Dive 1 - Cold Start

Deep Dive 2 - Concurrency and Auto Scaling

Auto Scaling (in the view of AWS)

Function Scaling with Concurrency limit

오늘 다루고 싶었지만 하지 못한 이야기

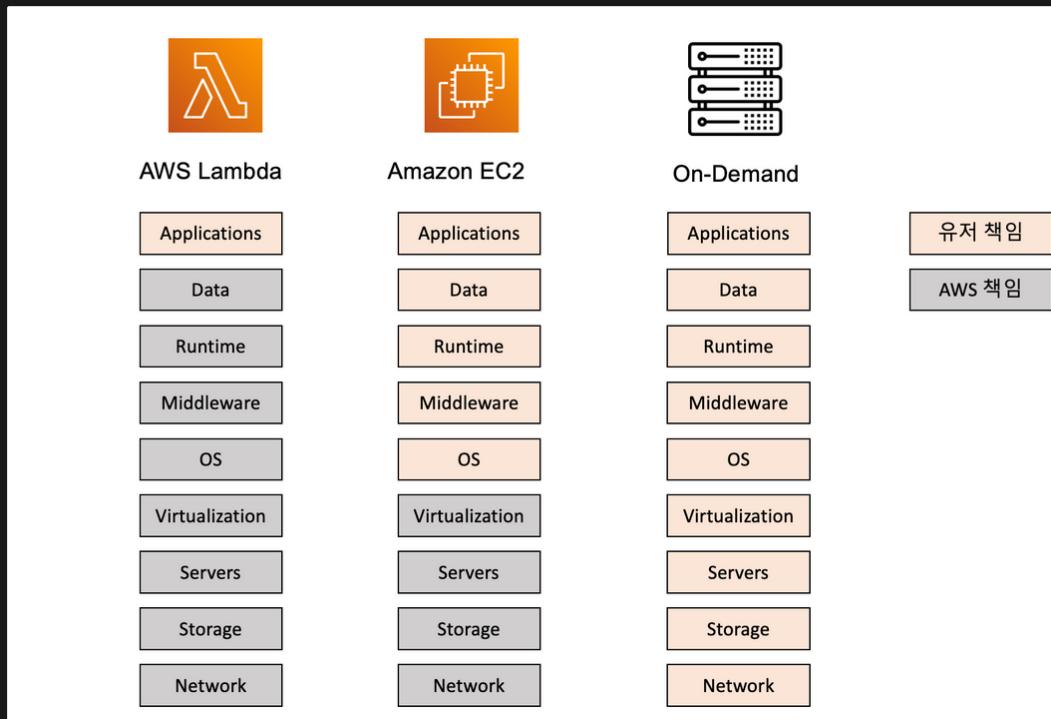
참고 자료

Limber up - What is AWS Lambda?

Serverless and AWS Lambda

AWS Lambda는 AWS에서 제공하는 서버리스 컴퓨팅 FaaS 상품입니다. 이때 서비스란 개발자가 인프라 환경 구축, OS 설치, 런타임 환경 설정 등을 할 필요 없다는 의미입니다. 또한 오토스케일링, 보안 등을 책임져 준다는 의미이기도 합니다. AWS Lambda를 이용할 경우 서버 프로비저닝과 관리를 모두 AWS 측이 해주기 때문에 개발자는 코드 혹은 비즈니스 로직에만 신경 쓸 수 있습니다.

그럼 저희가 잘 아는 EC2와 어떤 차이일까요? AWS가 책임져주는 범위와의 비교를 통해 둘을 비교해볼 수 있습니다. 아래 그림을 보면 확실히 AWS Lambda에서 유저가 신경써야 할 것들이 적음을 확인할 수 있습니다. 물론 실행 시간이나 사용 가능한 스토리지 등에서도 차이가 있는데, 이는 이후에 좀 더 자세하게 다루겠습니다.



<https://medium.com/harrythegreat/내게-알맞는-aws-컴퓨팅-서비스-찾기-bfd2c409273c의 표 재가공>

다시 AWS Lambda 소개로 돌아와서, Lambda의 특징을 요약해보자면 다음과 같습니다

- Function-as-a-Service
- 프로비저닝 혹은 서버 관리, 보안 관리를 신경 쓰지 않고 애플리케이션 운영
- 오토스케일링과 로드 밸런싱
- 실패 혹은 에러 핸들링 지원
- 사용한 만큼만 비용 차징(~~과다소 헤자스러운 비용정책~~)

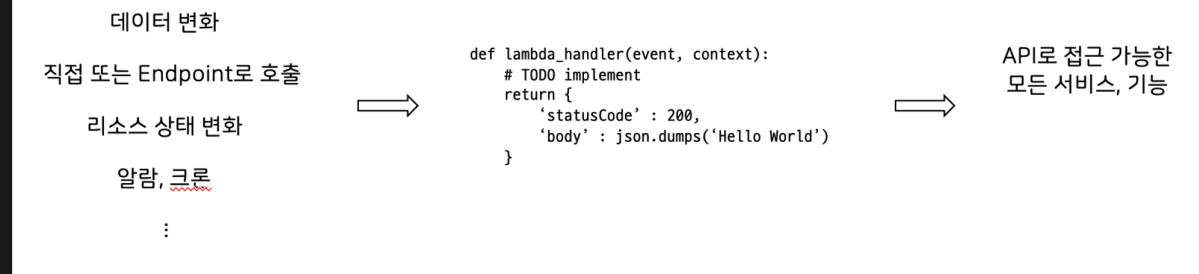
Working with AWS Lambda

Lambda를 어떻게 사용할 수 있는지 알아보겠습니다. Lambda를 사용하는 과정은 아래와 같이 표현할 수 있습니다.

이벤트 소스

Lambda Function

외부 서비스



아까 Lambda의 특징 중 하나를 Function-as-a-Service라고 했었죠? Lambda는 기본적으로 코드를 함수 형태로 실행합니다. 사용자는 lambda의 handler 함수를 무조건 정의해야 합니다 (즉, 코드의 최종적인 실행 형태를 함수로 실행하는 형태로 바꿔줘야 합니다). handler 함수는 이벤트 소스에서 발생한 이벤트를 받아 function을 실행하고, API 형태로 외부 서비스 혹은 기능을 호출하게 됩니다.

Lambda Handler Function

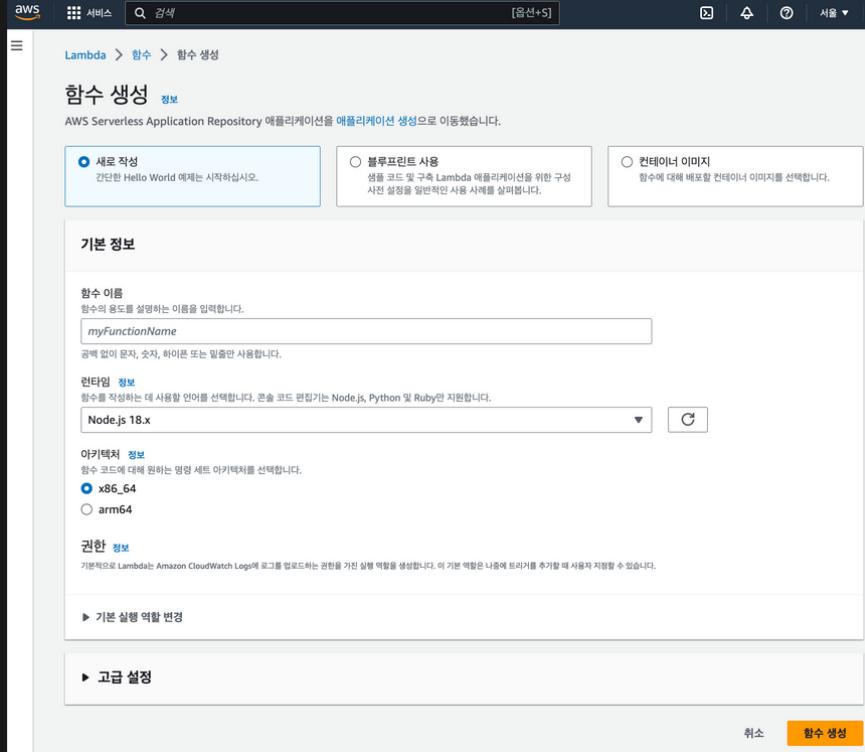
람다의 핵심이 되는 핸들러 함수를 자세히 봅시다.

```
import json def lambda_handler(event, context): # TODO implement return {
'statusCode': 200, 'body': json.dumps('Hello World') }
```

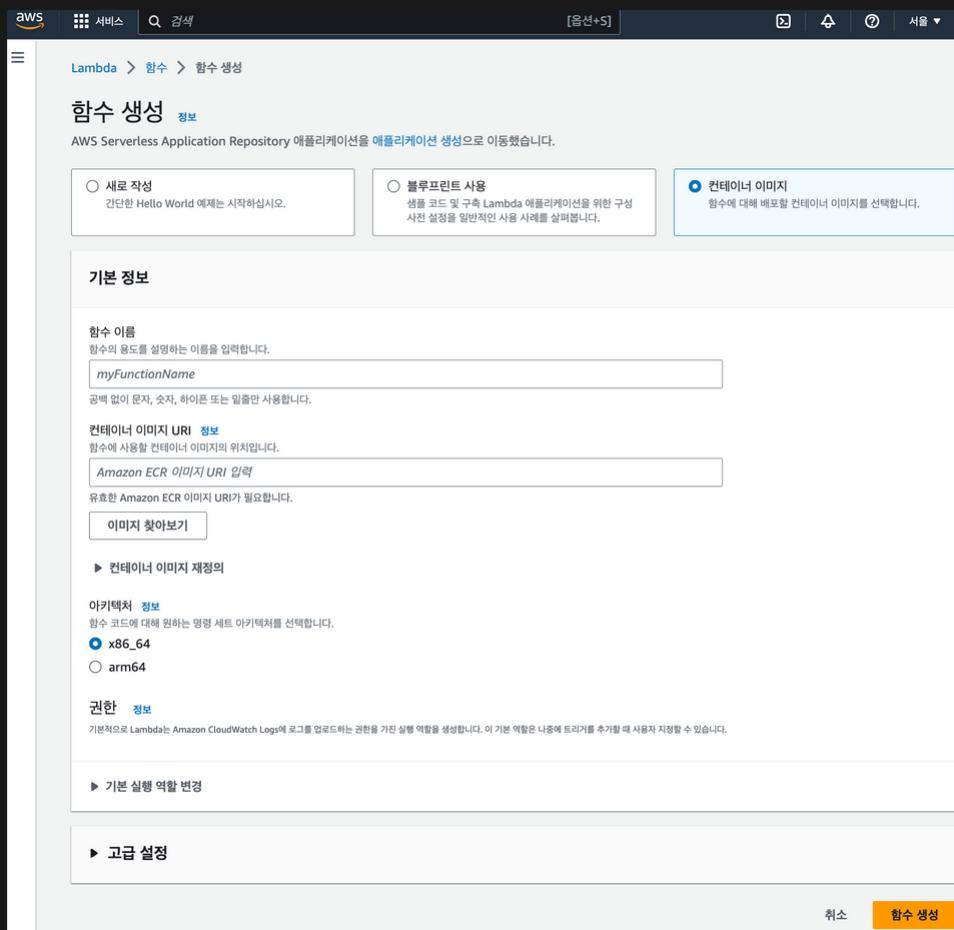
람다는 람다 함수를 트리거하는 event 객체, 람다의 런타임 정보(request ID, log group 등)를 가지고 있는 context 객체를 받습니다. 이벤트 소스에서 이벤트 객체를 받으면(트리거 되면) 람다 함수가 실행됩니다. 이 때 return만 하고 끝낼 수도 있고, API로 외부 서비스 혹은 기능을 사용할 수도 있습니다.

람다 함수의 경우 AWS 웹에서 직접 코드를 타이핑할 수도 있고, Zip 형태로 만들어 업로드할 수 있습니다. 2023년 9월 기준 람다는 다음 런타임 환경을 제공합니다

- .Net, Go, Python, Ruby, Java, Node.js (콘솔 코드 편집기는 Node.js, Python, Ruby 만 지원)



람다 함수는 컨테이너 이미지로도 배포 가능합니다. 컨테이너 이미지로 배포할 경우 AWS의 이미지 허브인 Amazon ECR에 이미지를 업로드하고, 람다에서 ECR 이미지 URI를 입력하여 사용할 수 있습니다.



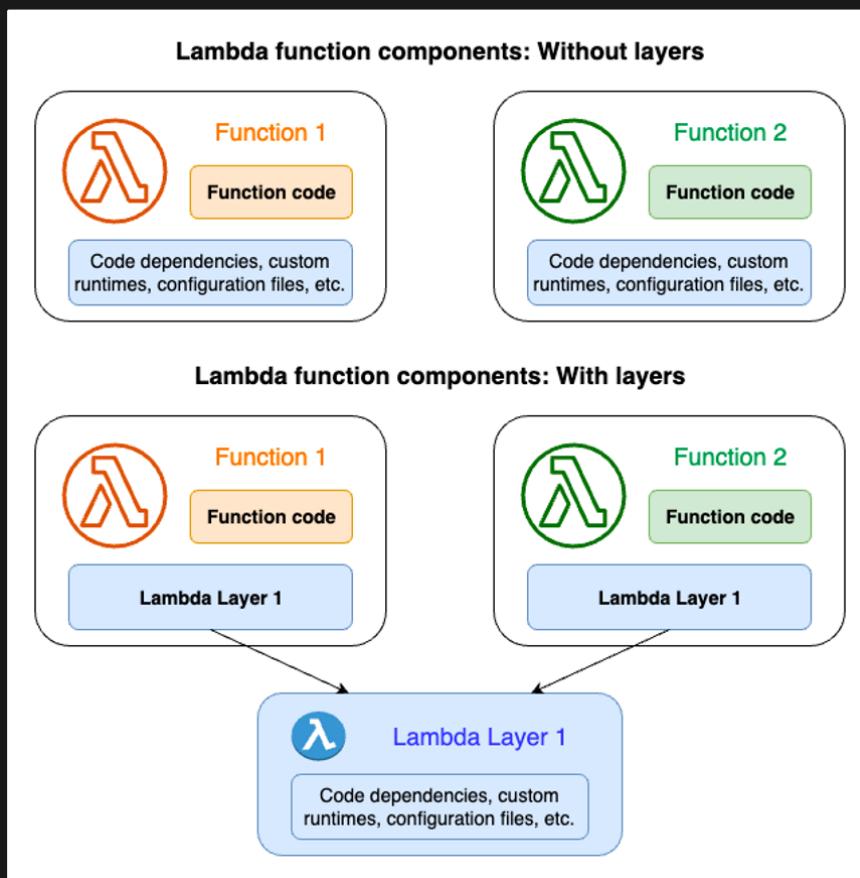
Lambda Layers

또한 람다의 Layers를 통해 크기가 너무 큰, 혹은 람다 함수들 간에 공통적으로 사용하는 의존성들을 따로 빼내어 사용할 수 있습니다.

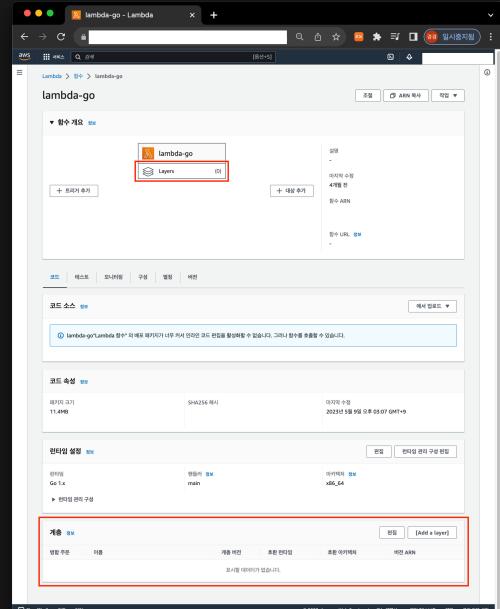
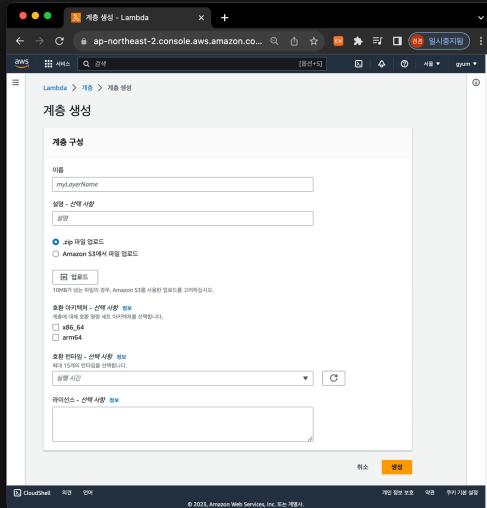
Layers 기능이 출시되기 전에는 람다 함수를 위한 모든 패키지 파일이 함께 압축되어 람다 함수에 업로드 되었어야 했습니다. 그것도 최대 250MB 안으로요. 그렇기 때문에 이전에는 아래와 같은 불편함이 있었습니다.

- 가장 중심이 되는 코드의 용량이 상당부분 제한되며
- 코드, I/O, RAM 사용 제약으로 람다 함수 실행 도중 패키지 설치는 거의 불가능에 가까움
- 코드 수정할 때마다 매번 패키지 파일과 같이 압축해서 업로드해야 함

하지만 Layers 기능이 출시되어 함수 간에 공통적으로 사용하는 패키지 파일 등은 따로 압축하여 업로드하고, 람다 함수를 작성할 때 사용하고자 하는 Layers를 골라 적용 시킬 수 있게 되었습니다. Layers로 등록된 압축 파일은 람다 함수가 실행될 때 `/opt` 폴더에 압축이 해제되고, 해당 데이터와 코드를 함수가 실행되는 중에 사용할 수 있습니다.



아래 왼쪽 그림과 같이 '계층' 혹은 'Layers'로 계층을 생성할 수 있습니다. 또한 오른쪽 그림과 같이 함수를 작성할 때 미리 만들어둔 Layers를 사용할 수 있습니다.



💡 분량 상 생략하지만 위에서 나온 것 같이 AWS 웹으로 조작할 수도 있지만, 여러 서비스들을 구성하고 설정할 때는 다소 불편할 수 있습니다. AWS에서는 SAM 템플릿을 활용하여 여러 AWS 서비스들을 각기 다른 구성으로 한 번에 배포할 수 있습니다.

SAM templates

```

1 AWSTemplateFormatVersion: '2010-09-09'
2 Transform: AWS::Serverless-2016-10-31
3 Resources:
4   GetProductsFunction:
5     Type: AWS::Serverless::Function
6     Properties:
7       CodeUri: src/
8       Handler: app.lambda_handler
9       Runtime: python3.8
10      Policies:
11        - DynamoDBReadPolicy:
12          TableName: !Ref ProductTable
13      Events:
14        HelloWorld:
15          Type: HttpApi
16          Properties:
17            Path: /products/{productId}
18            Method: get
19      ProductTable:
20        Type: AWS::Serverless::SimpleTable

```

Just 20 lines to create:

- Lambda function
- IAM role
- API Gateway
- DynamoDB table

SAM – Deployment Configuration

```

Resources:
MyLambdaFunction:
  Type: AWS::Serverless::Function
  Properties:
    Handler: index.handler
    Runtime: nodejs.x
    CodeUri: s3://bucket/code.zip

AutoPublishAlias: live

DeploymentPreference:
  Type: Canary10Percent10Minutes
  Alarms:
    - !Ref AliasErrorMetricGreaterThanZeroAlarm
    - !Ref LatestVersionErrorMetricGreaterThanZeroAlarm
  Hooks:
    PreTraffic: !Ref PreTrafficLambdaFunction
    PostTraffic: !Ref PostTrafficLambdaFunction

```

Just put additional properties
in SAM template file

© 2022, Amazon Web Services, Inc. or its Affiliates.



AWS SAM Transform

SAM template

```

1 AWSTemplateFormatVersion: '2010-09-09'
2 Transform: AWS::Serverless-2016-10-31
3 Resources:
4   GetProductsFunction:
5     Type: AWS::Serverless::Function
6     Properties:
7       CodeUri: src/
8       Handler: app.lambda_handler
9       Runtime: python3.8
10      Policies:
11        - DynamoDBReadPolicy:
12          TableName: !Ref ProductTable
13      Events:
14        HelloWorld:
15          Type: HttpApi
16          Properties:
17            Path: /products/{productId}
18            Method: get
19      ProductTable:
20        Type: AWS::Serverless::SimpleTable

```



CloudFormation template

```

1 {
2   "AWSTemplateFormatVersion": "2010-09-09",
3   "Resources": {
4     "ServerlessHttpApi": {
5       "Type": "AWS::ApigatewayV2::Api",
6       "Properties": ...
7     },
8     "ProductTable": {
9       "Type": "AWS::DynamoDB::Table",
10      "Properties": ...
11    },
12    "ServerlessHttpApiGatewayDefaultStage": {
13      "Type": "AWS::ApigatewayV2::Stage",
14      "Properties": ...
15    },
16    "GetProductsFunction": {
17      "Type": "AWS::Lambda::Function",
18      "Properties": ...
19    },
20    "GetProductsFunctionHelloWorldPermission": {
21      "Type": "AWS::Lambda::Permission",
22      "Properties": ...
23    },
24    "GetProductsFunctionRole": {
25      "Type": "AWS::IAM::Role",
26      "Properties": ...
27    }
28  }
29 }

```

193 lines in total

Limitations of Lambda

람다의 경우 컴퓨팅 스펙 관련하여 많은 제약 사항들이 존재합니다. 람다 함수를 작성 할 경우 아래 사항들을 유의하여 작업해야 합니다.

1. 실행시간

- 람다의 기본 실행 시간은 3초, 최대 실행 시간은 900초(15분)입니다.
- 실행 시간은 '구성 > 일반 구성'에서 편집 가능합니다.
- 실행 시간 동안 함수의 실행이 완료되지 않을 경우 람다는 함수를 실패 처리 합니다. 즉, 람다는 함수의 실행이 완료될 때까지 기다려주지 않습니다.

2. 메모리(와 CPU)

- 람다 함수를 실행하는데 사용하는 메모리 스펙은 기본 128MB이며, 최대 10240MB까지 설정 가능합니다. (왜인지 모르겠지만 과거 최대치인 3008MB 까지만 할당 가능할 때가 있습니다)
- 람다의 vCPU 스펙은 메모리 할당에 비례하여 할당됩니다. 공식적으로 1,769MB를 할당하면 1개의 CPU 스펙을 온전히 활용할 수 있다는 정도만 알려져 있습니다. 그 이상은 많은 이들이 실험을 통해 경험적으로 밝혀내고 있습니다.
 - Proportional CPU Allocation과 관련된 [미디엄 글](#)이 있으니 따로 참고해보셔도 좋을 것 같습니다
- 메모리 할당은 비용과 직결($\text{비용} = \text{memory} * \text{execution time}$)되기 때문에 성능과 비용 사이의 trade-off를 잘 고려해야 합니다

3. 임시 파일 시스템

- 람다의 실행 환경에서 사용할 수 있는 임시 파일 공간이 있습니다.
- 폴더 명은 `/tmp`이며, 람다가 실행되면서 저장되는 파일 등은 이 공간 안에 저장됩니다.
- 512MB가 기본이며 10,240MB까지 늘릴 수 있습니다.

4. 배포 패키지 사이즈

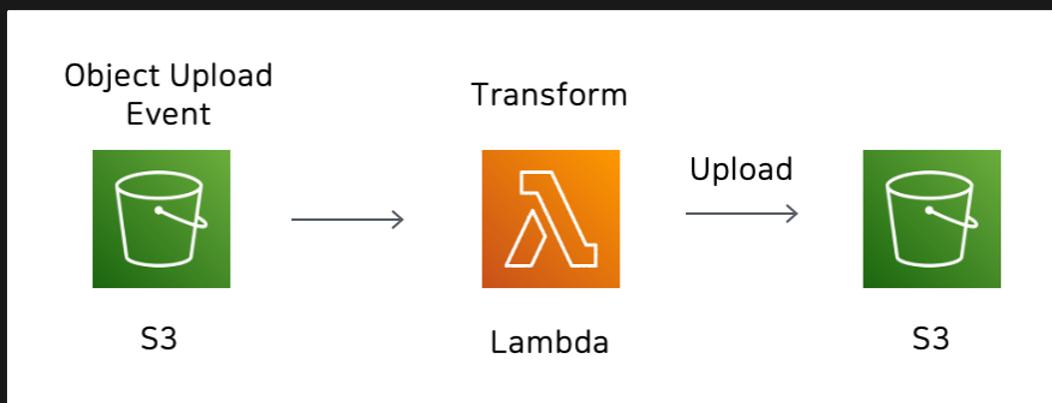
- 람다 함수에 올릴 수 있는 코드는 (레이어와 커스텀 런타임을 포함) 최대 250MB, 압축했을 경우 50MB까지 가능합니다.
- 컨테이너 이미지의 경우 10GB까지 가능합니다

AWS Lambda Use-Case

그렇다면 Lambda가 실제 어떻게 사용되는지 알아봅시다.

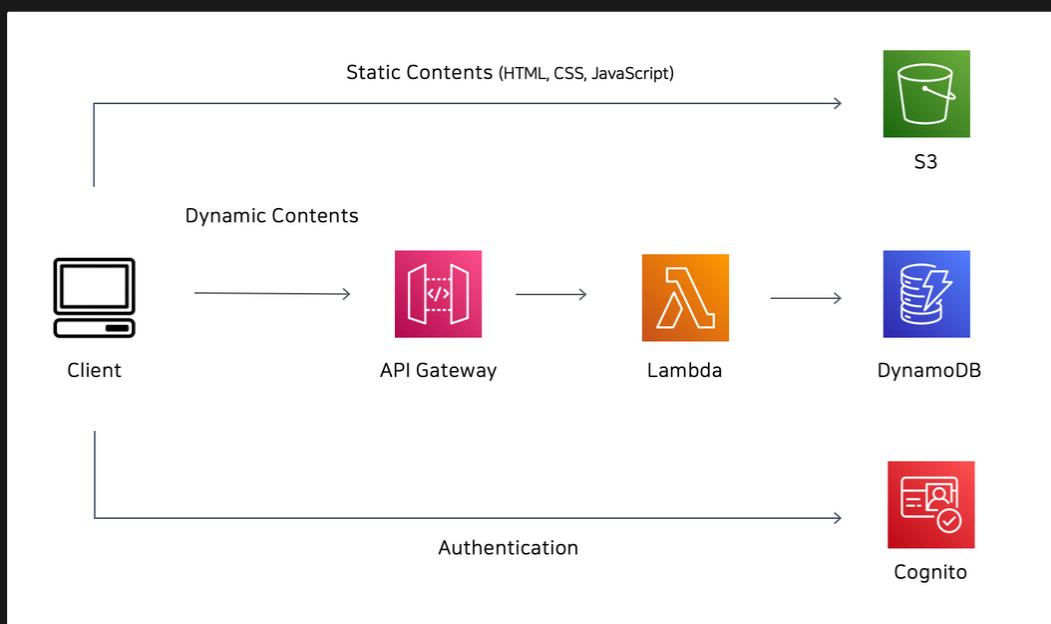
Case 1 - Data Transform Automation

가장 간단하게는 S3와 연동하여 데이터 변환을 자동화할 수 있습니다. S3에 객체가 업로드 되는 것을 트리거로 업로드 된 객체를 Lambda에서 적절히 변환, 다시 S3에 업로드합니다.



Case 2 - Web Application

HTML, CSS와 같은 정적인 콘텐츠는 S3에서, 프로그램이 필요한 동적인 콘텐츠는 Lambda와 API Gateway를 연동하여 사용할 수 있습니다. 인증으로는 cognito를 별도로 사용하고요.



Case 3 - Monitoring

CloudWatch에서 에러 혹은 알람이 발생하였을 경우 Lambda에서 이를 받아 가공, SNS에 등록된 이메일로 받아볼 수 있습니다. Slack webhook을 사용하여 알림과 에러 내용을 Slack 봇으로 받아볼 수도 있습니다.

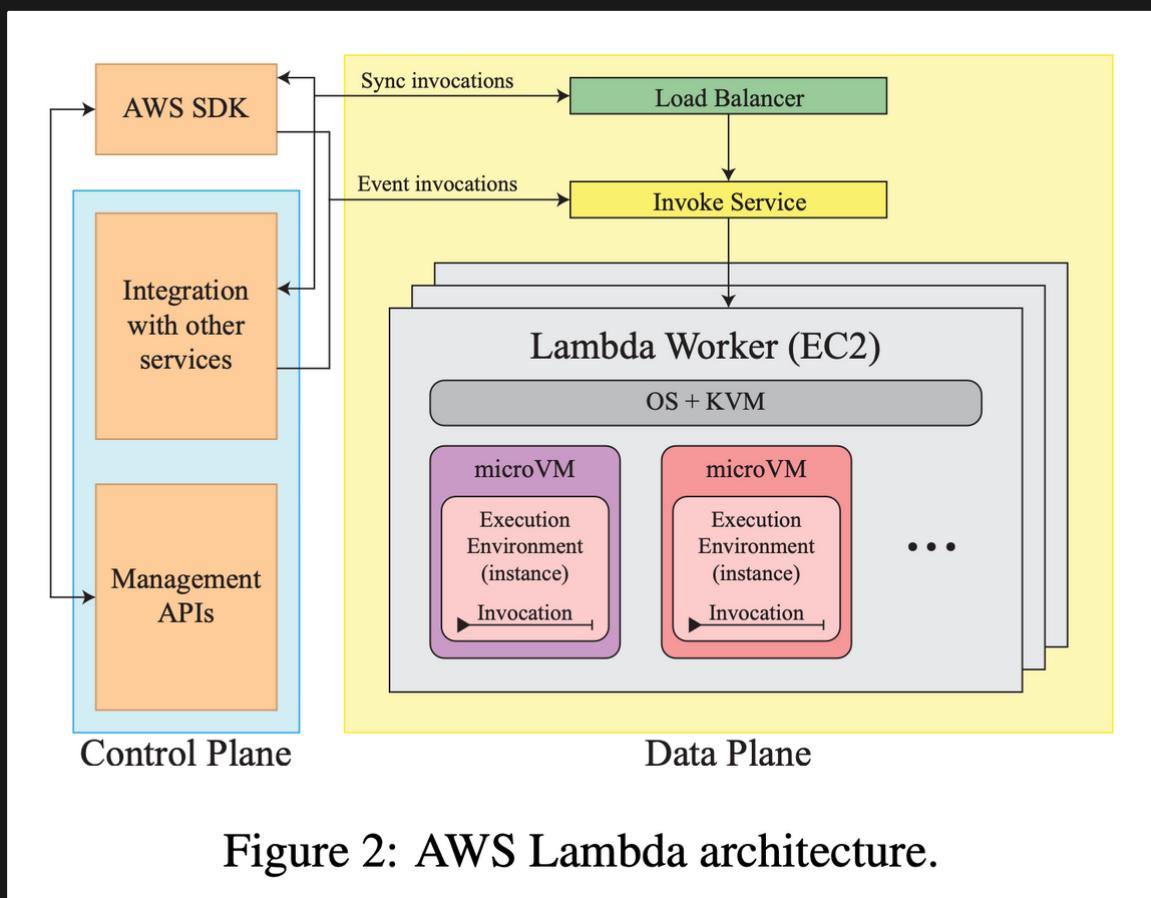


Deep Dive 0 - How Lambda works (in the view of AWS)

위까지의 내용은 AWS Lambda를 한 번이라도 써본 사람이라면 얼추 다 아는 내용입니다. 하지만 지금까지 우리는 람다 함수를 작성하기만 했지, 람다라는 서비스가 구체적으로 어떻게 구성되어 있는지 알아보지는 않았습니다. AWS Lambda에 대해서 조금 더 심층적으로 알아보기 위해, AWS Lambda가 어떻게 동작하는지 보다 상세하게 알아보는 시간을 가져보도록 하겠습니다.

Architecture of AWS Lambda

람다의 아키텍쳐는 블랙박스로 취급되었으나 최근 AWS에서 [Security Overview of AWS Lambda](#)라는 백서를 통해 그 모습이 꽤 상세하게 그릴 수 있게 되었습니다. 또한 2018년 이후 AWS Fargate와 Lambda에 적용된 Firecracker 기술에 대한 논문 ([Firecracker: Lightweight Virtualization for Serverless Applications](#))을 통해 Lambda의 대략적인 모습을 알 수 있습니다. 두 문서를 바탕으로 Daniel과 Pedro는 아래와 같은 Lambda의 아키텍쳐 그림을 제시합니다.



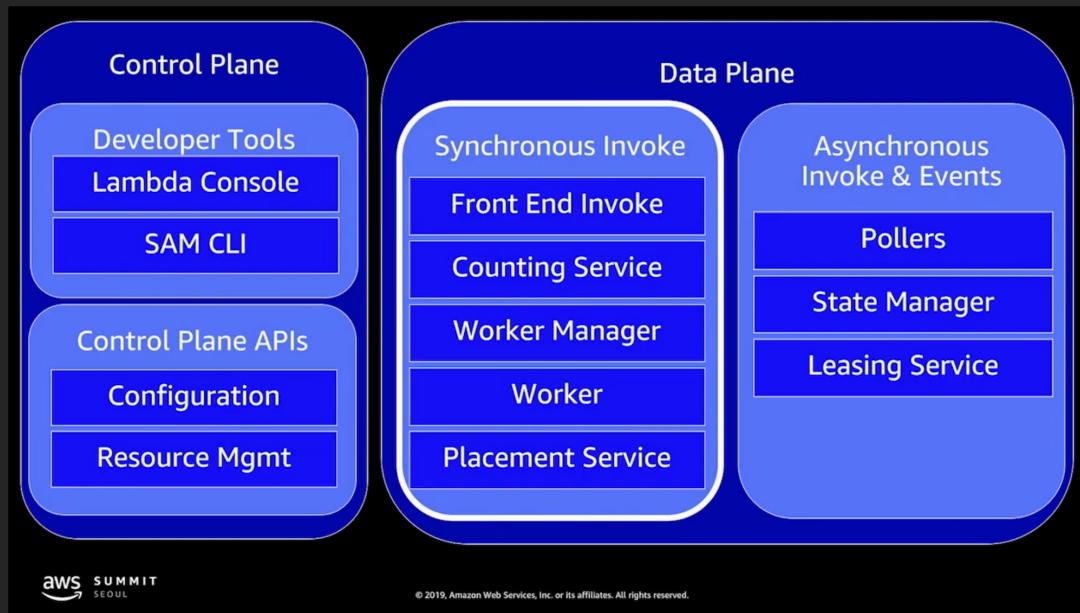
쿠버네티스를 공부해보신 분들에게는 꽤 익숙한 구조일 것이라 생각합니다. 람다는 크게 Control Plane과 Data Plane으로 나뉘어져 있습니다. Control Plane에서는 람다 함수의 생성이나 업데이트를 위한 Management API(ex. CreateFunction, UpdateFunctionCode, PublishLayerVersion)를 관리합니다. 다른 클라우드 서비스와의 integration도 여기에서 control plane에서 관리를 합니다.

Data Plane은 실제 람다 함수가 돌아가는 곳입니다. 람다 함수를 돌리라고 우리가 요청하는 것을 함수 호출(function invocation)이라고 하는데요, Data Plane에서는 이 함수 호출의 종류에 따라 요청이 Load Balancer 혹은 Invoke Service를 거쳐 Lambda Worker 안에 Execution Environment(실행 환경)에 함수가 할당되는 프로세스가 이루어집니다. Lambda Worker는 베어메탈 EC2 Nitro 인스턴스로 이루어져 있으며, 실행 환경은 이 EC2 인스턴스 위에서 Firecracker 기반으로 생성된 hardware-virtualized microVM을 뜻합니다. 람다 함수는 실제적으로는 이 microVM 위에서 작동이 되는 것입니다. (그렇기 때문에 많은 Lambda 문서에서 실행 환경이 container 기반이라고 설명합니다)

그럼 이제 위 아키텍처를 기반으로 Lambda 함수의 실행이 요청, 처리되는 흐름을 따라 조금 더 상세하게 살펴보도록 하겠습니다.

💡 하... 자료 다 만들고 한국 AWS Summit 2019에서 발표된 자료를 발견하여... 내용 추가합니다

거의 대부분의 내용을 같고요, 함수 호출과 페이로드를 워커에 할당하는 부분에 있어서 살짝 더 디테일하게 설명이 들어갔습니다.



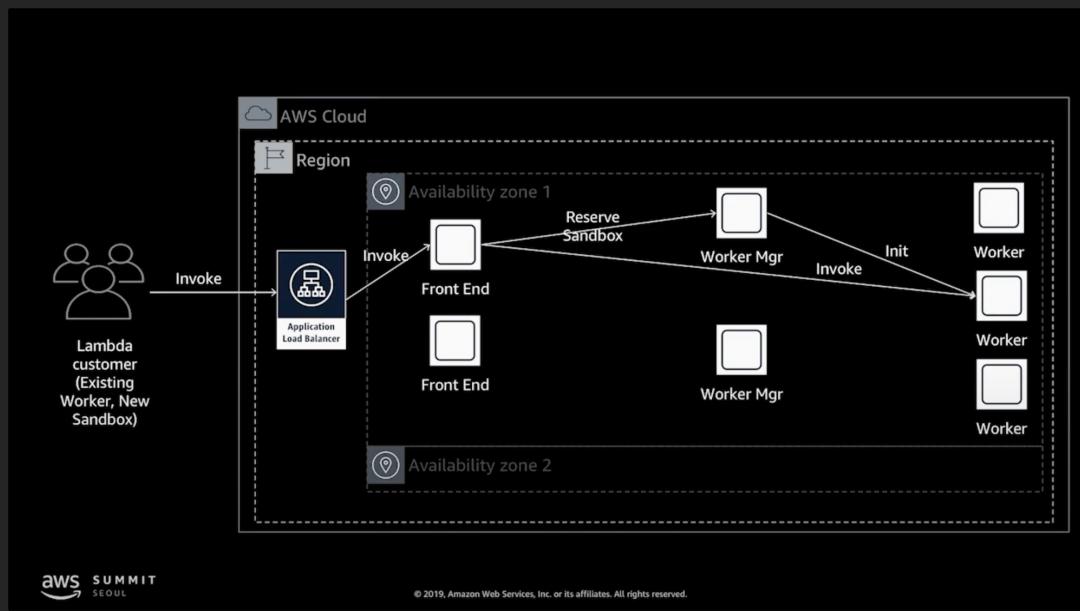
Front End Invoke: 동기, 비동기 호출을 모두 관리하며 API call을 처음 받은 컴포넌트입니다. 세계의 많은 유저들이 람다를 호출할 때 이것을 밸런싱하는 역할을 합니다

Counting Service: 사용자가 얼마나 많은 API 요청을 하고 있는지 모니터링하고 제한 기능을 제공합니다.

Worker Manager: 실제 컨테이너의 상태를 관리하고 API 요청을 가능화하는 컨테이너로 중계

Placement Service: 워커에 Sandbox 구성을 자원 활용률이 높고, 고객 서비스 영향이 없도록 관리 (워커의 상태를 관리)

Load Balancing



고가용성, 성능을 위해 Front End와 Worker Manager도 여러 개가 존재합니다. 로드밸런서를 통과한 요청을 Front End에서 받고, Front End는 Worker Manager에게 현재 사용 가능한 Worker가 있는지 확인 요청합니다. 만약 있다면 바로 할당하고, 없다면 Worker Manager가 새로운 Worker를 생성하고 그 후에 함수 호출이 할당 됩니다.

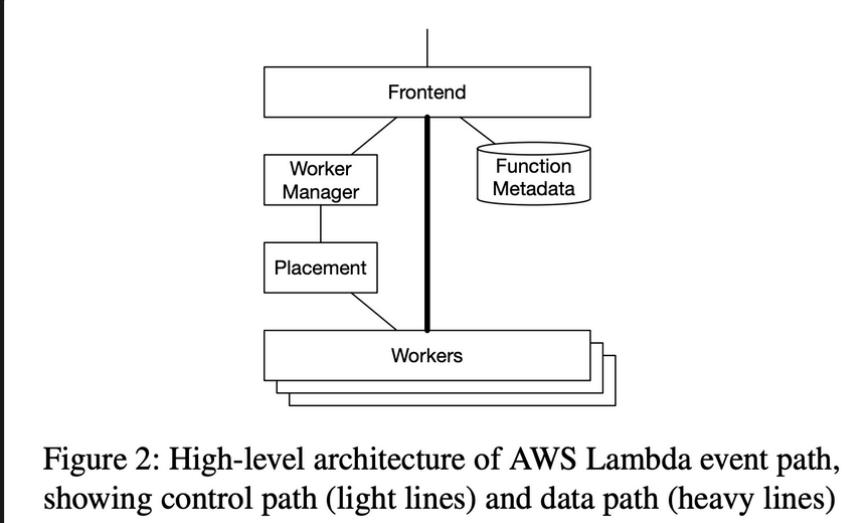


Figure 2: High-level architecture of AWS Lambda event path, showing control path (light lines) and data path (heavy lines)

Invocation Types

Data Plane에서는 이 함수 호출의 종류에 따라 요청이 Load Balancer 혹은 Invoke Service로 전달된다고 한 바 있습니다. 이는 Invoke API에 Request-Response Mode와 Event Mode 두 가지 모드가 있기 때문입니다. 각각 아래 그림의 Sync Invocations와 Event Invocations에 대응해서 생각해주시면 될 것 같습니다.

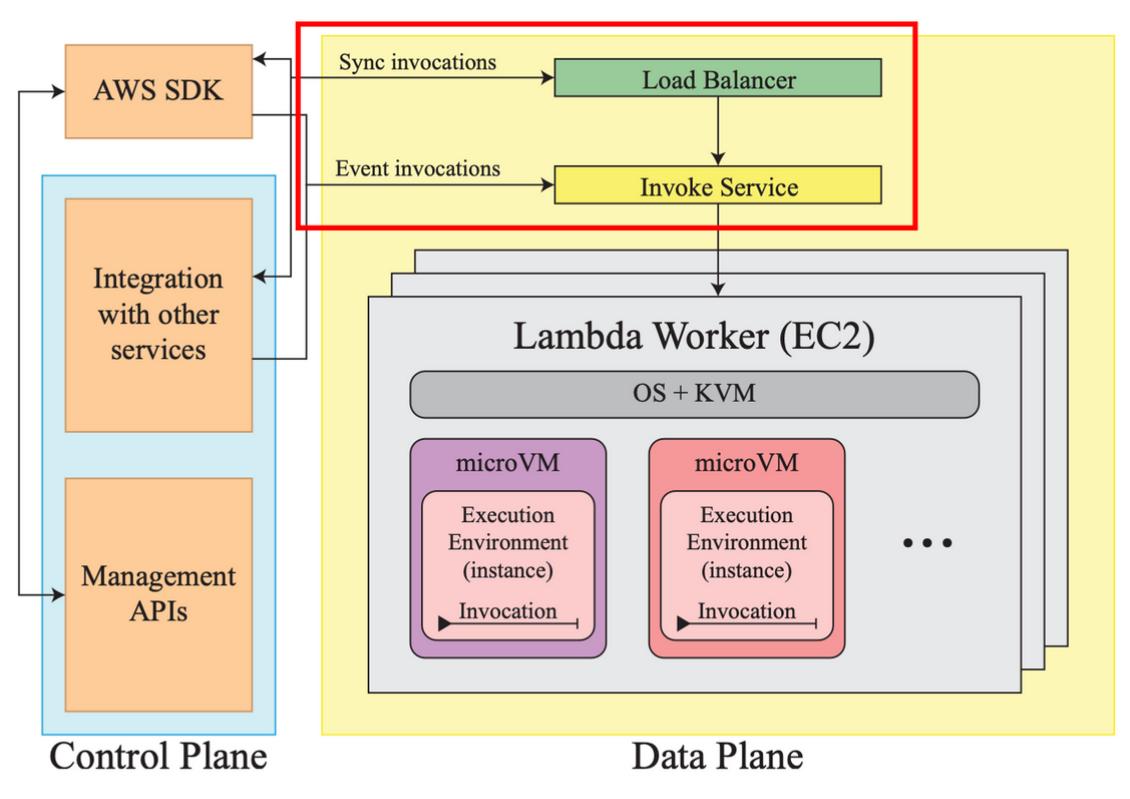
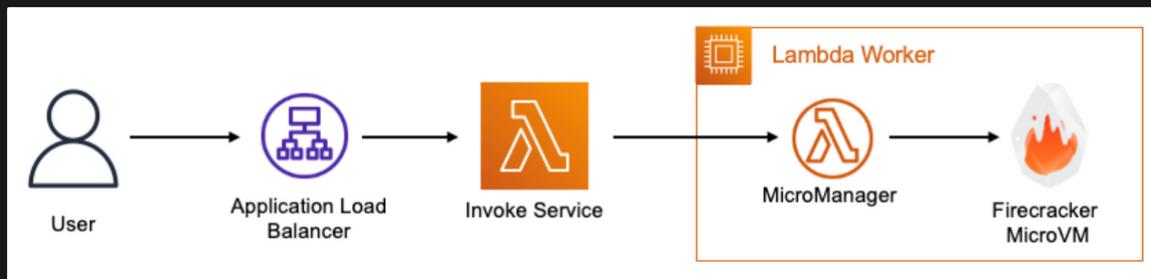


Figure 2: AWS Lambda architecture.

D. Barcelona-Pons, P. García-López, Benchmarking Parallelism in FaaS Platforms (2020)

Request-Response Mode

요청-응답 모드는 함수를 동기적으로(synchronously) 호출하고 즉시 응답을 반환하는 모드입니다. 함수 호출과 페이로드는 애플리케이션 로드밸런서를 거쳐 바로 Invoke Service로 보내지는데요, 이때 Invoke Service가 사용 불가하다면 클라이언트 사이드에서 페이로드를 잠시 큐에 넣어놓고 함수 호출을 재시도하도록 설정할 수 있습니다. 페이로드가 invoke service에 전달이 되면, 그때 invoke service가 사용 가능한 execution environment를 찾아 이를 할당하게 됩니다. (마땅한 execution environment가 없다면 새로 생성하여 할당합니다)



Event Mode

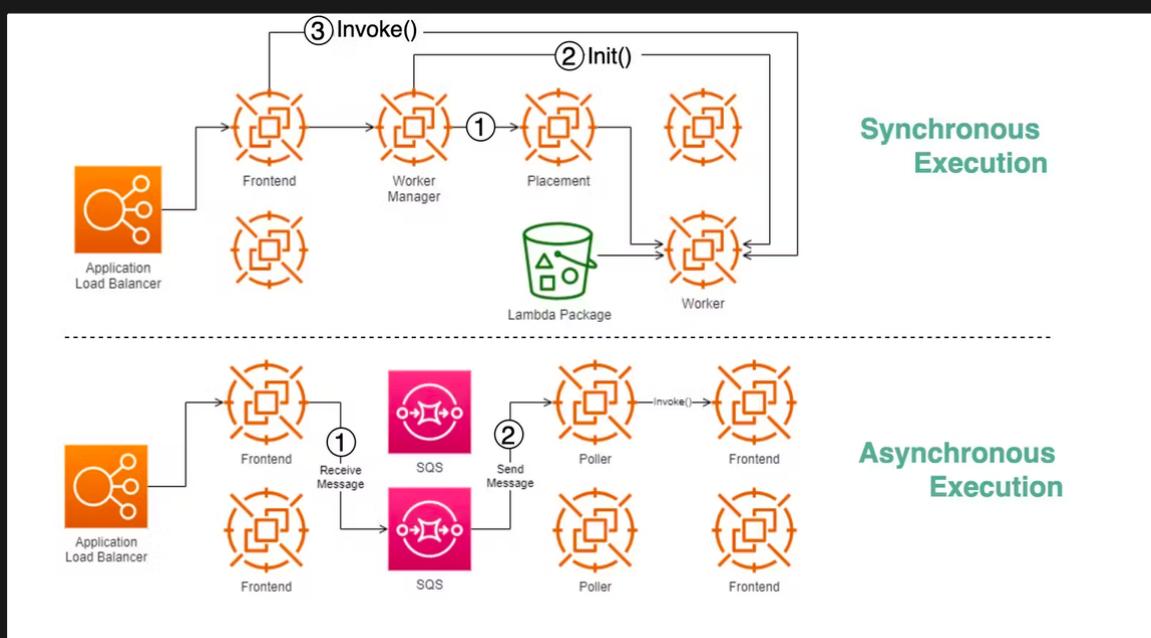
이벤트 모드는 함수가 비동기적으로(asynchronously) 호출되어 있을 때 사용되는 모드입니다. 이벤트 모드에서는 함수 호출이 처리되기 전 페이로드가 큐 됩니다(참고로 이때 Amazon SQS queue를 사용하며, 공유 큐에서 대기할 수도 있지만 호출 속도, 이벤트 크기 등에 따라 전용 큐에 할당되거나 마이그레이션 될 수도 있습니다).



대충 이런 느낌?

큐된 이벤트는 람다의 poller fleet(큐에서 아직 처리하지 않은 것들을 처리하기 위한 EC2 인스턴스 그룹)에 의해 배치 형태로 반환되고 invoke service에 전달됩니다. 만약 invoke service가 사용 불가하거나 호출된 함수가 실행 불가한 상태라면 poller fleet에서 인메모리 형태로 이벤트를 보관, 재시도를 요청하고, 실행 가능한 상황이라면 할당된 execution environment로 이벤트가 전달되어 람다 함수가 실행됩니다.

만약 함수 호출이 완전히 실패할 경우 람다는 이를 버리게 되는데, 이때 dead letter queue (DLQ) 기능을 사용하면 처리 실패한 이벤트를 Amazon SQS나 Amazon SNS에 보내는 방식으로 핸들링할 수도 있습니다.



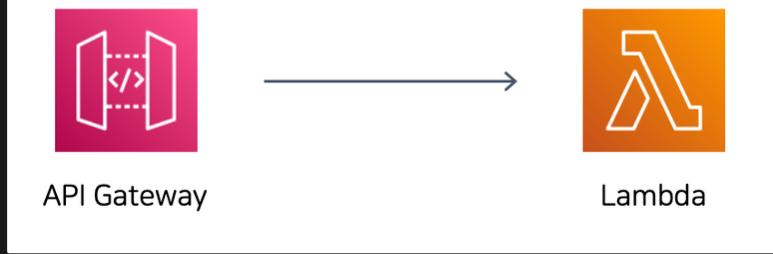
Invocation Types with Examples

위에만 봐서는 실제로는 뭐가 어떻게 다른지 사실 잘 감이 안 오죠? 실제 예시를 통해 알아봅시다.

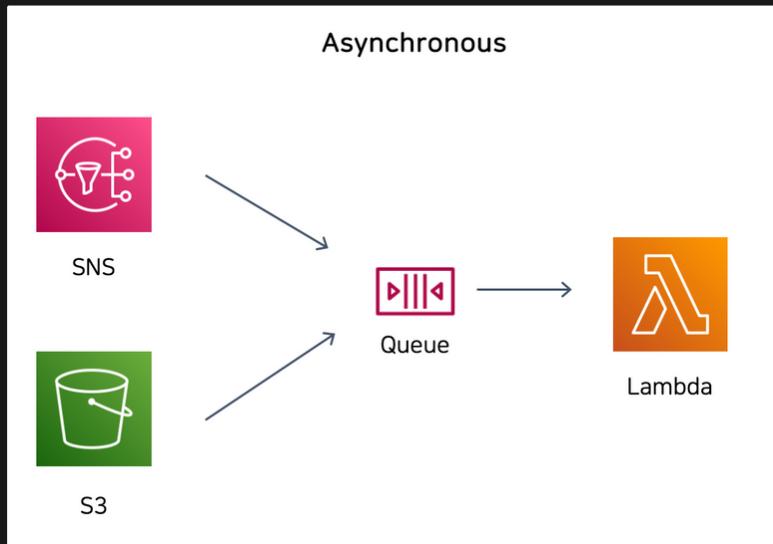
우선 동기적 호출과 그에 따른 Request-response 모드에 대해서 알아봅시다. 위 방식으로 람다 함수를 호출할 경우, 호출자는 람다 함수가 응답(response)를 보낼 때까지 대기하게 됩니다. 응답에 대한 세부 내용은 응답의 헤더와 바디 안에 들어와서 전달됩니다(이는 곧 함수를 호출한 곳에서 직접 이 응답을 분석하고 처리해주어야 한다는 의미이기도 합니다)

람다 함수의 URL HTTP(S) 엔드포인트, Lambda API, AWS CLI 등을 이용하여 람다 함수를 호출하는 경우, 혹은 CloudFront, API GateWay, ELB와 같은 AWS 서비스를 사용하는 경우가 여기에 속합니다.

Synchronous

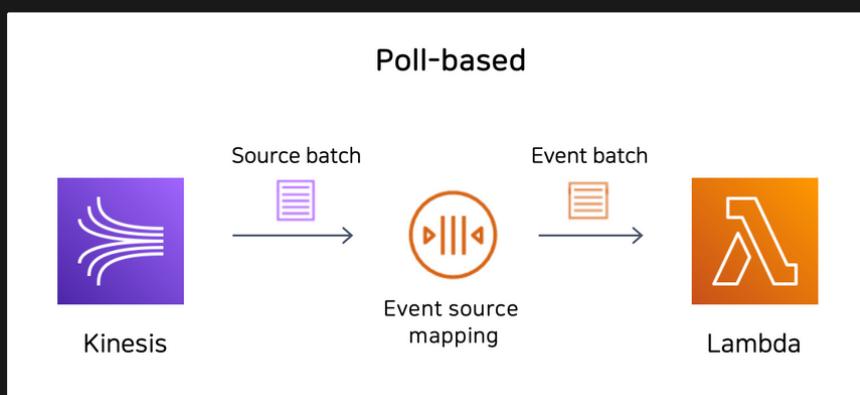


다음은 비동기적 호출과 Event 모드입니다. 비동기적으로 호출할 경우, 호출만 던져두고 따로 람다 함수의 응답을 기다리지 않습니다. 던져진 이벤트는 위에서 설명했던 것처럼 큐에 넣어졌다가 람다의 실행 환경에 전달됩니다. S3나, SNS, CloudWatch Logs나 CodeCommit 등의 이벤트로 람다 함수를 호출하는 경우가 여기에 속합니다.



그런데 위 비동기적 처리 예시로 분류하기 살짝 애매한 상황도 있습니다. 바로 Kinesis나 SQS와 같은 스트림이나 큐를 기반으로 한 서비스가 람다 함수를 호출하는 상황, 즉 Poll-Based Invoke 상황인데요. 이때 람다는 사용자를 대신해서 폴링하고, 레코드를 retrieve하여 함수를 동기적으로 호출합니다. 이 과정을 이벤트 소스 매핑(Event Source Mapping)이라고 합니다.

이벤트 소스 매핑은 타깃 이벤트 소스에서 아이템을 읽고, 배치 형태로 람다 함수에 전달하게 됩니다. 배치 크기는 기본 1이지만, 이벤트 소스에 따라 최대최소 배치 크기가 달라질 수 있습니다. 이렇게 만들어진 이벤트 배치(event batch)가 람다 함수에 전달되며, 만약 함수에서 에러가 날 경우 이 배치 전체에 대해 재실행하게 됩니다.



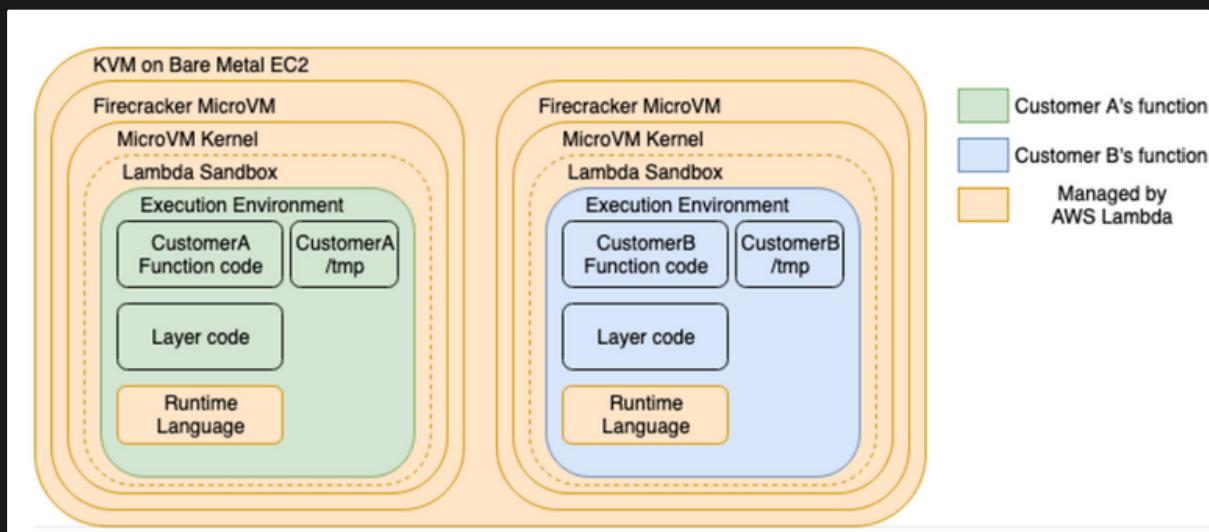
Execution Environment

지금까지 함수 호출 과정에 대해서 알아보았습니다. 지금부터는 함수 호출과 페이로드가 Lambda Worker에서 어떤 방식으로 처리되는지 알아보겠습니다.

Lambda Worker

본격적으로 Execution Environment를 설명하기 앞서, 해당 환경을 감싸고 있는 Lambda Worker에 대해서 자세히 알아보겠습니다.

아키텍쳐 파트에서 invoke service를 통해 함수 호출과 페이로드가 lambda worker 위에 있는 execution environment로 전달된다고 설명한 바 있습니다. Lambda Worker는 Amazon EC2 AWS Nitro 인스턴스이고, 하나의 인스턴스 위에 여러 개의 Micro Virtual Machine(MVM)이 돌아가는 형태입니다. 이 가상 머신은 리눅스 커널을 바탕으로 만든 가상머신 소프트웨어(KVM)인 Firecracker를 이용하여 만들어집니다.



Security Overview of AWS Lambda AWS Whitepaper

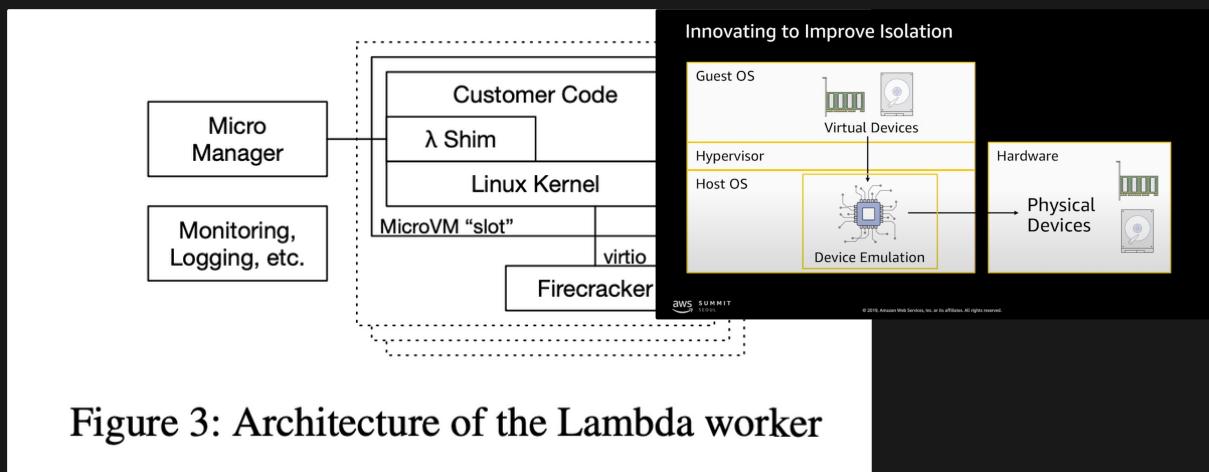


Figure 3: Architecture of the Lambda worker

아래와 같은 기술들이 사용되어 유저들마다 격리된 공간 안에서 함수를 실행할 수 있게 되었습니다. 참고하시면 람다를 좀 더 깊게 이해하는데 도움이 될 것이라 생각합니다

- Control groups (cgroups) – Used to constrain the function's access to CPU and memory. (각 환경에 리소스 할당)
- Namespaces – Each execution environment runs in a dedicated namespace. We do this by having unique group process IDs, user IDs, network interfaces, and other resources managed by the Linux kernel.
- seccomp-bpf – To limit the system calls (syscalls) that can be used from within the execution environment. (보안 관련)
- iptables and routing tables – To prevent ingress network communications and to isolate network connections between MVMs.
- chroot – Provide scoped access to the underlying filesystem. (루트 파일 시스템 내에서 독립된 루트 파일 시스템을 만들어 별도의 공간으로 사용 가능하게 함)
- Firecracker configuration – Used to rate limit block device and network device throughput.
- Firecracker security features – For more information about Firecracker's current security design, refer to Firecracker's latest design document.

참고로 람다에서는 격리 및 보안을 위해 `/dev/shm` (리눅스 환경에서의 공유 메모리 파일 시스템)을 사용할 수 없기 때문에 람다 환경에서 Python Multiprocessing 모듈에서 Pool은 사용 불가하며, chromium 사용시 일정 부분 제약이 있습니다.

 워커의 임대시간은 최대 14시간 정도입니다. 워커가 최대 임대 시간에 도달하거나, 더 이상 함수 호출이 할당되지 않게 되면 Micro Virtual Machine은 graceful하게 종료되고 워커 인스턴스 또한 종료됩니다.

 2014년 람다가 처음 공개되었을 때는 Firecracker가 아닌 QEMU와 리눅스 컨테이너 기반으로 격리, 가상화를 실행했습니다. 이럴 경우 하나의 가상 머신 안에서는 완전한 격리가 불가하기 때문에 계정당 각기 다른 가상 머신을 생성해야 했습니다. (보안과 효율성의 Trade-off)

하지만 2018년 12월부터 KVM은 유지하되 그 안에서 돌아가는 QEMU를 Firecracker MicroVM으로 교체하여 하나의 KVM 안에서 다른 계정의 람다 실행 환경이 만들어질 수 있게끔 변경 되었습니다.



Rust로 만들어졌다고 합니다... 간지...

Firecracker 덕분에 하나의 호스트에서 수천개의 VM이 돌아갈 수 있을만큼 가볍고 성능이 향상되었다고 합니다.

Execution Environment

워커 위에 새로운 실행 환경을 생성할 때 워커에 유저의 함수 아티팩트(zip이나 이미지 형식으로 올렸을 경우 한 번 최적화된 형태의 것)에 대해서 임시 접근 권한을 갖게 됩니다. 생성된 실행 환경 안에서 런타임이 부트스트랩되고, 람다 함수 코드가 실행됩니다.

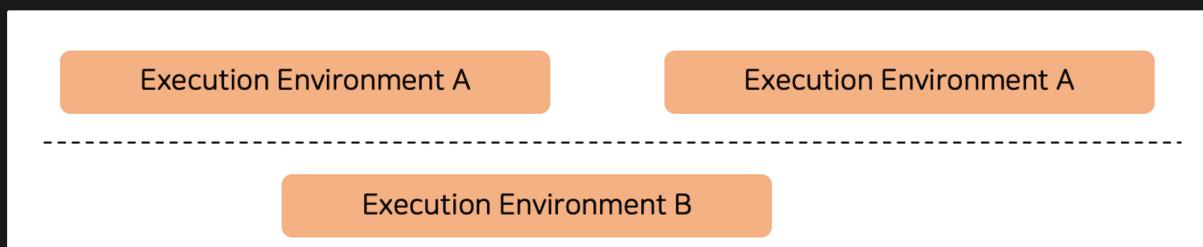


생성된 실행 환경은 다음과 같은 요소들이 있습니다.

- Amazon Linux 2 기반의 (최소한의) 리눅스 유저 스페이스
- 함수 코드
- AWS Lambda Layers
- 런타임 (ex. Go 1.X, Java 11, Node.js 12, Python 3.8, 커스텀)
- 쓰기 가능한 `/tmp` 디렉토리

생성된 실행 환경(컨테이너 환경)에서는 하나의 함수가 실행됩니다(one concurrent invocation at a time). 이때 사용된 실행 환경은 다른 함수 혹은 같은 함수이나 다른 버전의 함수는 실행 불가합니다. 다만 이전에 실행 되었던 것과 같은 버전의 같은 함수를 실행하는데 있어서는 재사용될 수 있습니다.

처음 람다 함수 호출을 처리하기 위해 실행 환경 A가 만들어졌다고 가정 해봅시다. 실행 환경 A에서 함수가 돌아가는 동안 동일한 함수 호출이 하나 더 들어오게 된다면, 람다는 자동으로 실행환경 B를 만들어 해당 요청을 처리합니다. 만약 실행환경 A에서 작업이 끝났고, B에서는 아직 안 끝난 상황에서 3번째 호출이 들어오게 된다면 람다는 실행 환경 A를 사용하여 함수를 실행합니다.



유의할 것은 같은 실행 환경을 재사용 할 때 해당 환경에서의 `/tmp` 혹은 인메모리 데이터(or 상태)는 그대로 남아 있다는 점입니다. 남아 있는 데이터가 함수 실행에 영향을 미칠 수 있으므로 이를 유의하여 (특히 같은 요청을 병렬적으로 처리해야 하는) 함수를 작성해야 합니다.

동일한 실행 환경 내에서 여러 함수 호출은 하나의 프로세스로 처리(하나의 함수 안에서 멀티 프로세싱을 못한다는 뜻은 아닙니다)되기 때문에 Java의 static state와 같이 process-wide한 상태들의 경우 재사용할 수 있습니다. 또한 람다 함수가 쓰기 권한을 지닌 `/tmp` 폴더 안 데이터를 공유하는 것도 가능합니다.

남아 있는 데이터가 함수 실행에 영향을 미칠 수 있는 가능성이 충분히 존재하기 때문에 이를 유의하여 함수를 작성해야 합니다. 반대로 말하자면 계속해서 사용해야 하는 큰 데이터를 `/tmp`에 임시 저장하거나 로컬 캐시, long-lived connections를 사용하여 람다 함수의 처리 성능을 향상시킬 수도 있습니다.

런타임 환경의 경우 AWS 측에서 각 언어별 업데이트 혹은 보안 패치 등을 관리해주기 때문에 개발자들은 코드 작성 및 에러 핸들링에만 집중할 수 있습니다. 다만 람다에서 특정 런타임 버전을 지원하지 않는 경우, 람다 함수 실행이 되지 않기 때문에 이 경우 코드를 업데이트 해줘야 합니다. 그 외에 개발자가 직접 액션을 취해야 하는 경우 이메일 등을 통해 알림을 전송합니다.

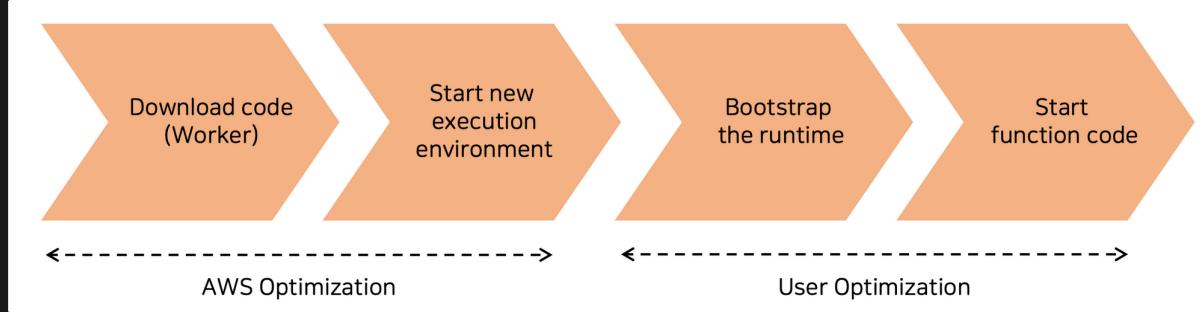
Deep Dive 1 - Cold Start

서비스 혹은 람다를 이용할 경우 가장 흔하게 겪게되는 문제입니다. 람다는 EC2처럼 24시간 내내 서버가 세팅되어 돌아가지 않고, 일정 시간 동안 함수를 돌릴 수 있는 환경을 잠시 빌리는 서비스입니다. 즉, 일정 시간이 되면 띄워졌던 컨테이너가 사라지고, 우리는 다시 모든 것을 처음부터 돌려야 한다는 의미입니다.

앞서 함수 호출이 들어오면, 함수 코드가 실행되기 전 워커에서 새로운 microVM을 띄우고, 사용자의 코드를 다운 받는 등의 초기화 작업이 선행된다고 설명한 바 있습니다.



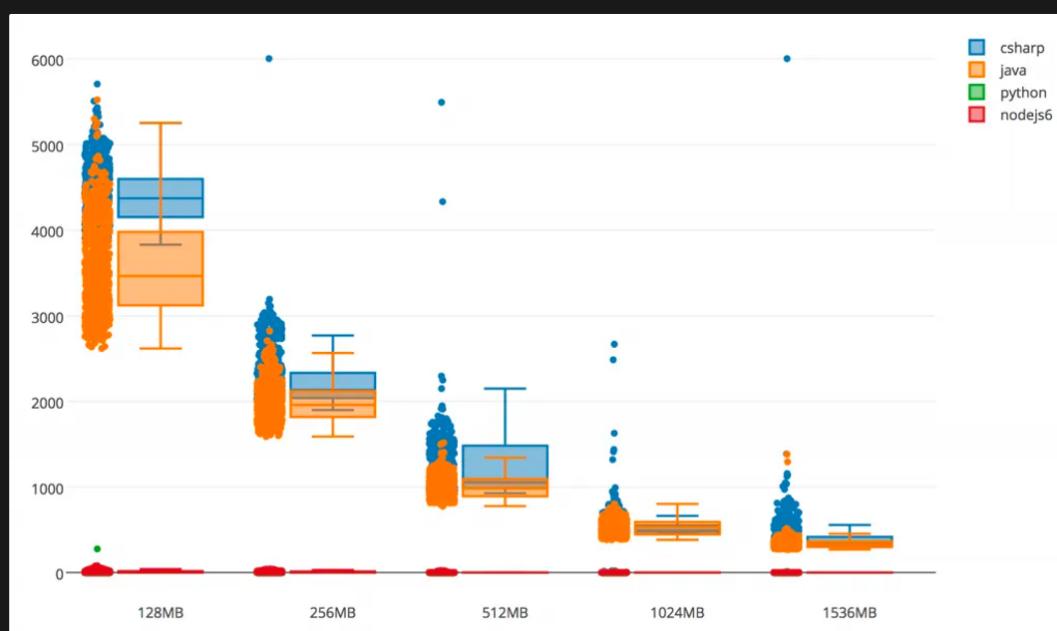
이는 다시 말해 사용자는 AWS가 실행 환경을 만들어줄 때까지 잠시 대기하고 있어야 한다는 것입니다. 이 초기화 과정을 거치는 것을 통상 콜드 스타트(cold start)라고 부릅니다. 이 때문에 요청에 대한 응답에 지역 시간이 발생하고, 15분 내에 함수를 다 돌리지 못하는 문제가 발생하기도 합니다.



콜드 스타트라는 것은 서비스 로직 상 필연적으로 발생할 수밖에 없기 때문에, 콜드 스타트 시간을 최대한 줄이는 방법 혹은 실행 환경이 종료되지 않도록 계속 트리거를 주는 방법을 도입해야 합니다.

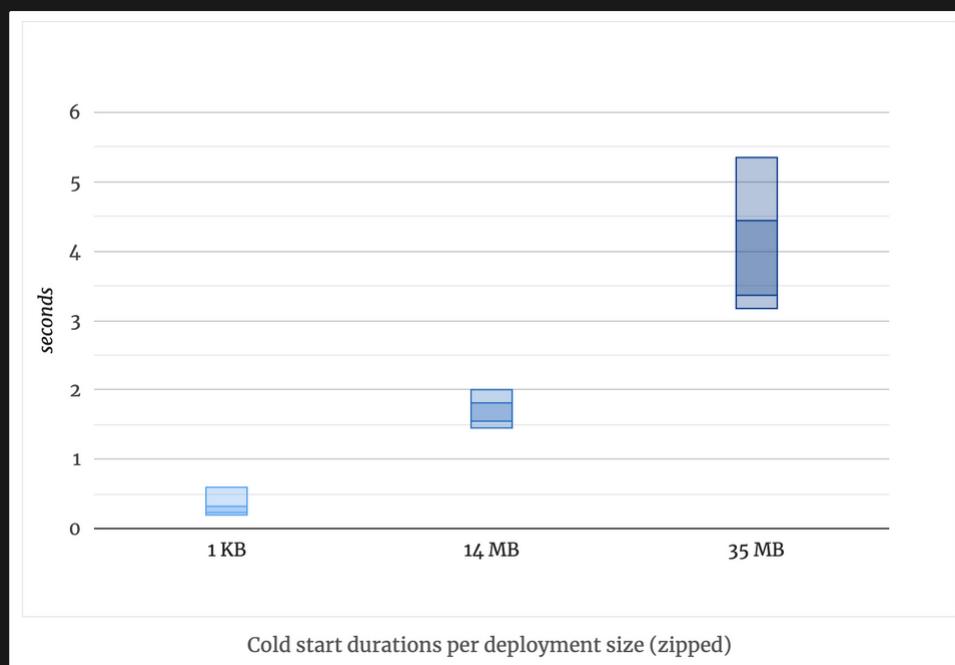
우선 최적화를 통해 콜드 스타트 시간을 줄이는 방법을 알아봅시다.

- 가장 쉽고 빠르게 시간을 줄이는 것은 컴퓨팅 파워를 늘리는 것, 즉 더 좋은 CPU를 할당하는 것입니다. 램다에서는 메모리 할당에 비례하여 CPU가 할당되기 때문에 더 많은 메모리를 할당함으로써 이를 수행할 수 있습니다. (다만 비용 문제와 직접적으로 연관되어 있기 때문에 주의해야 합니다)
- 리플렉션을 되도록 지양하는 것이 좋습니다. 특히 자바와 하이버네이트(자바 언어를 위한 ORM 프레임워크)의 경우 리플렉션을 많이 사용하고, 내부적으로도 stateful한 것들을 많이 사용하기 때문에 되도록 자바스크립트 혹은 파이썬으로 개발하는 것을 추천하고 있습니다.
 - 자바의 경우 .jar 파일들을 별도의 `/lib` 디렉토리에 배치하는 방법도 있습니다



<https://theburningmonk.com/2017/06/aws-lambda-compare-coldstart-time-with-different-languages-memory-and-code-sizes/>

- 람다 함수를 실행하기 위해 코드를 다운받는 시간이 존재하기 때문에, 다운받을 패키지의 사이즈를 줄임으로써 속도를 개선할 수 있습니다.



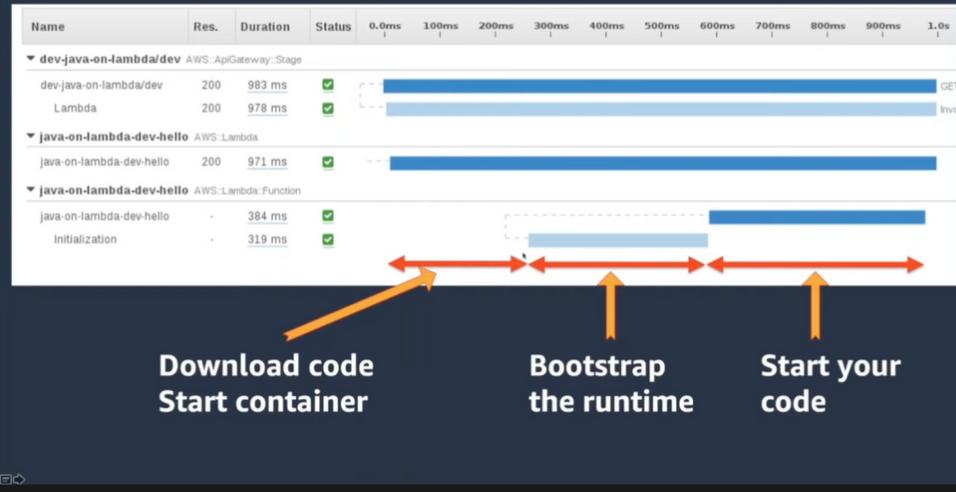
<https://mikhail.io/serverless/coldstarts/aws/>

다음은 콜드 스타트가 아니라 이미 람다의 실행 환경을 어느정도 구축해놓는 방법, 즉 웜 스타트를 할 수 있는 방법들입니다.

- 다른 서비스를 사용하여 지속적으로 람다 함수 호출하기
 - (람다 함수의 실행이 로직에 직접적인 영향을 미치거나 하는 경우 사용 불가)
 - CloudWatch를 통해 람다를 5분에 한 번씩 호출해주기
 - Route53 health check를 람다 함수 API Gateway Endpoint로 설정하기
 - 기타 플러그인 사용하기 ([예시](#))
- 전역 변수, `/tmp` 폴더 사용하기
 - 위에서 설명했듯, 실행환경에서 `/tmp` 폴더와 핸들러 함수 밖에서 생성된 변수의 경우 실행 환경에 남습니다. 이를 이용하여 최초의 콜드 스타트 외에는 효율적으로 코드를 실행
 - 하지만 반드시 어떤 호출이 이 실행 환경 안에서 실행될 것이라는 보장이 없다는 것을 유의
- 프로비저닝된 동시성(Provisioned Concurrency) 사용
 - 미리 프로비저닝된 실행 환경을 예약해두는 기능. 설정한 수만큼 미리 인스턴스를 초기화 해두고 대기하고 있고, 함수 호출이 들어오는 경우 프로비저닝 프로세스 생략 후 바로 코드 실행
 - 이는 곧 인스턴스를 미리 켜두는 것을 의미하기 때문에 함수 호출이 없더라도 비용 청구 됨

AWS X-Ray를 바탕으로 람다 함수가 프로비저닝되고 돌아가는 과정에서 얼마나 많은 시간이 걸렸는지 알 수 있기 때문에, 분석하여 초기화 과정을 최적화할 수 있습니다.

Seeing a cold start in AWS X-Ray



- 💡 사실 제가 람다를 딥하게 써보진 않아서 콜드 스타트가 얼마나 심각한지 체감이 잘 안 되긴 합니다. 우선 문서에 따르면 약 300ms 안으로 나온다고 하는데 실제는 잘 모르겠네요 (실험과 서비스는 또 많이 다르니까요)

Deep Dive 2 - Concurrency and Auto Scaling

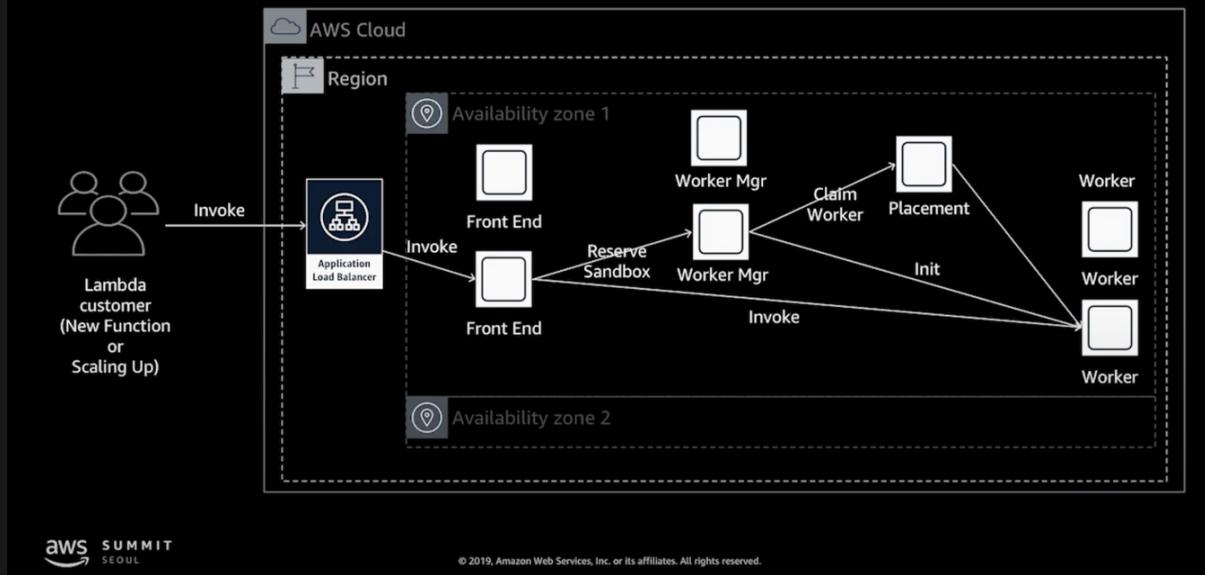
동일한 요청(ex. 갑자기 트래픽이 급증하여 API Gateway를 통해 수많은 (동일한) 함수 호출이 발생한다고 생각해봅시다. 한 실행 환경 안에서 람다 함수가 다 돌아가기도 전에 수많은 호출이 동시에 오는 상황입니다. 이때 람다는 이 함수 호출들을 적절히 처리할 수 있을 만큼 처리 환경 컨테이너를 자동으로 띄워줍니다.

- 💡 참고로 필요한 동시성을 계산하는 방법은 아래와 같습니다.

$$\text{Concurrency} = (\text{average requests per second}) * (\text{average request duration in seconds})$$

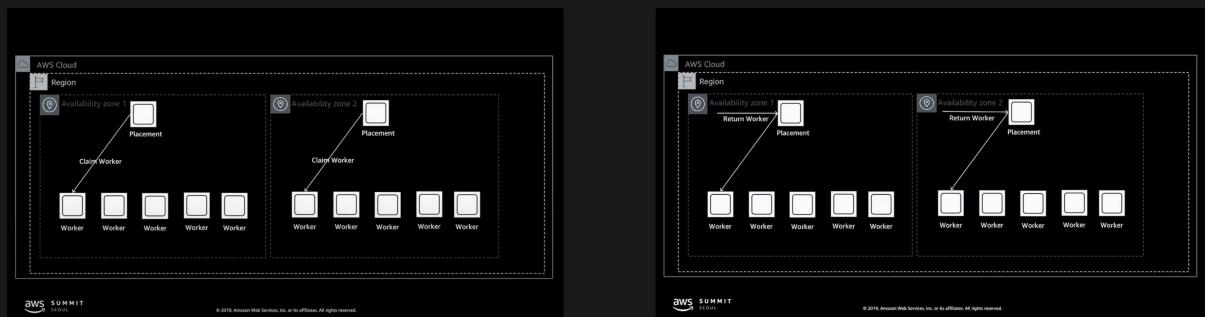
Auto Scaling (in the view of AWS)

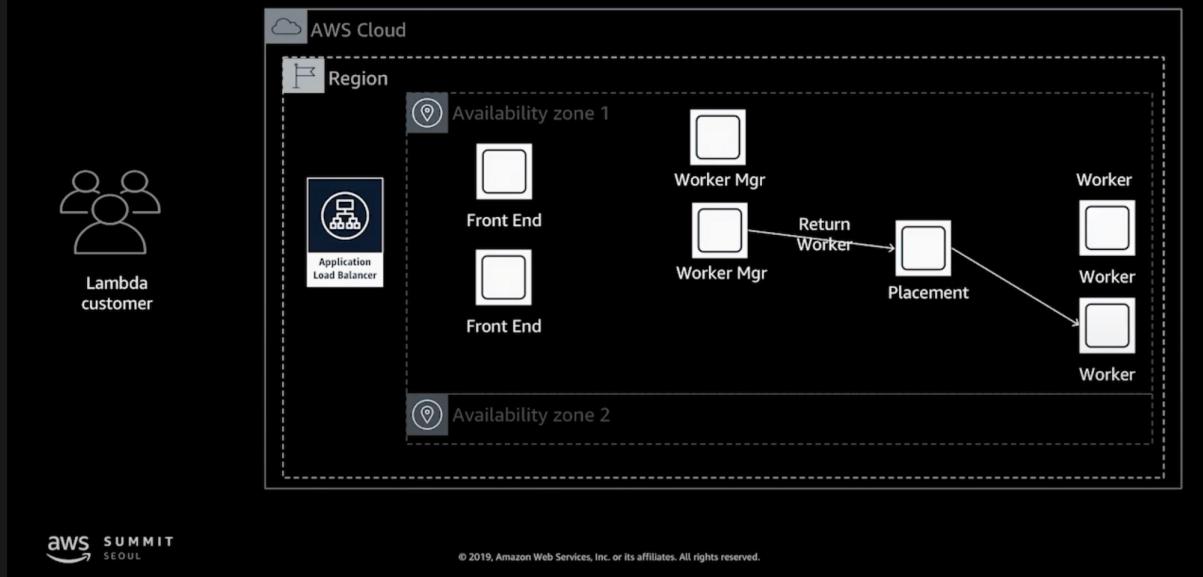
람다 아키텍쳐의 관점에서 오토 스케일링이 이루어지는 흐름은 아래 그림과 같습니다.



위에서 언급하였듯, 로드밸런서를 통과한 요청을 Front End에서 받습니다. 이때 Front End는 Worker가 얼마나 바쁜지, 함수 호출을 더 할당해도 되는지에 대한 정보를 Placement에게서 받습니다. Placement가 적합한 워커의 정보를 알려주게 되면, Worker Manager는 해당 워커에 함수 호출을 할당하고 없다면 새로 워커를 띄워 함수를 할당하게 됩니다. 정보를 획득한 Manager에게 현재 사용 가능한 Worker가 있는지 확인 요청합니다.

Placement는 현재 워커에 대한 정보를 계속 모니터링하고 있습니다. 또한 워커의 실행이 종료되고 리턴하는 리소스 또한 placement에서 담당하게 됩니다.

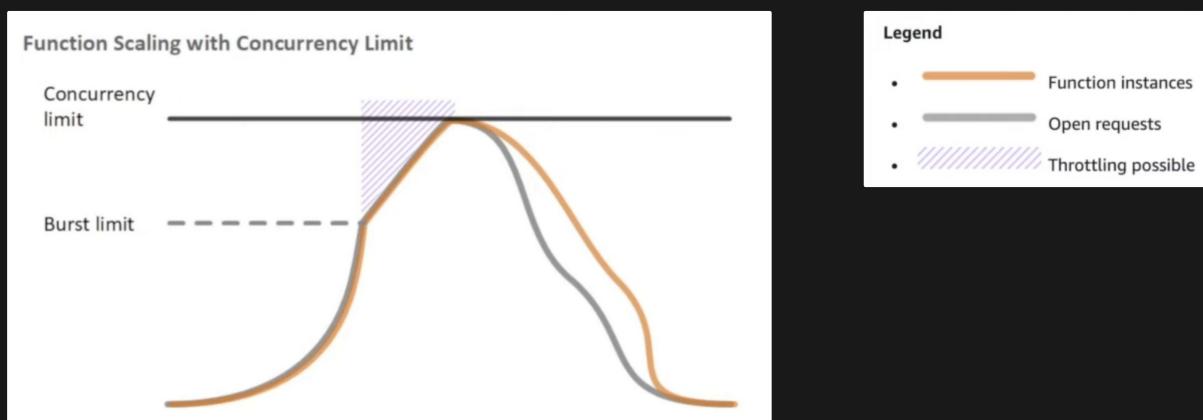




Function Scaling with Concurrency limit

그렇다면 이 오토 스케일링으로 어디까지 동시 처리를 할 수 있을까요? 그리고 어떤 속도로 오토 스케일링이 진행될까요?

Concurrency Limit



우선 동시성 한도(Concurrency limit)는 얼마만큼 동시성 처리를 할 수 있는가에 대한 제한입니다. 동시성 한도를 넘어서는 함수 요청이 들어올 경우 램다 함수가 요청 삭제를 하는 등 쓰로틀링이 발생합니다. 동시성 한도는 계정 안에서 공유되며, 기본 1000 개가 할당됩니다(더 필요하면 AWS에 요청할 수 있음)

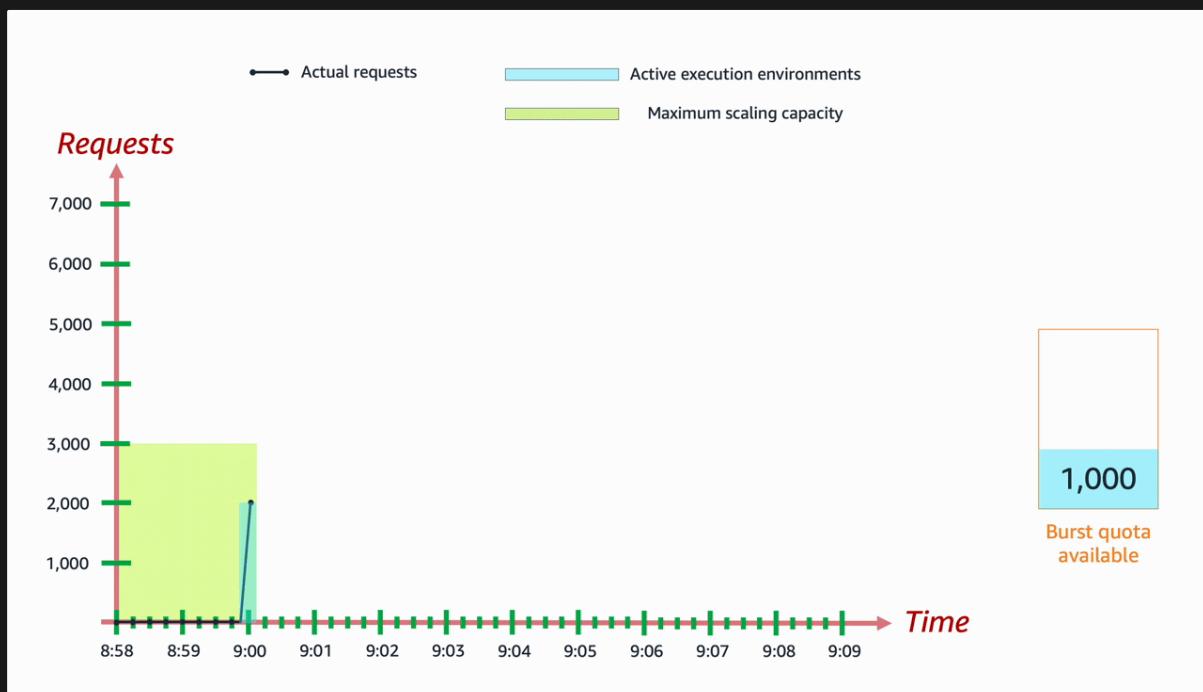
Burst Concurrency

갑작스러운 트래픽 버스트가 발생할 경우, 이를 스무스하게 처리하기 위해 버스트 동시성(Burst Concurrency)이 존재합니다. 짧은 시간 안에 동시 처리를 위한 스케일업이 되는, 말 그대로 버스트하게 동시성 처리를 해주는 것이죠. 위 그림에서도 Burst limit이라고 적혀있는 곳까지는 빠르게 function instance가 늘어나는 것이 보이실 겁니다.

하지만 burst limit이라는 단어에서 볼 수 있듯, 버스트 동시성에도 한도가 존재하며 이는 오버 스케일링을 방지하기 위함입니다. 버스트 동시성 한도는 리전별로 다르지만 최대 500~3000개가 가능합니다

| 리전 | 버스트 동시성 한도 |
|--------------------------------------|------------|
| 미국 서부(오레곤), 미국 동부(버지니아 북부), 유럽(아일랜드) | 3,000 |
| 아시아 태평양(도쿄), 유럽(프랑크푸르트), 미국 동부(오하이오) | 1,000 |
| 기타 모든 리전 | 500 |

이때 버스트 동시성 한도라는 개념은 한 번에 충전 가능한 버스트 동시성으로 이해하는 것이 편할 것 같습니다. 최초의 버스트가 진행된 이후에 계속 요청이 대량으로 들어올 경우 사용 가능한 버스트 동시성은 1분에 500개씩 늘어납니다. 게임으로 비유하자면 버스트 동시성 한도가 최대 MP 수치이고, 1분에 한 번 MP가 500씩 회복되는 것입니다. 아래 gif는 동시성 한도가 10000, 버스트 한도가 3000이며, 09:00에 2000개의 버스트가 발생한 이후에 과정을 보여줍니다.



Throttling

만약 동시성 한도보다 많은 요청이 들어올 경우, 동시성 한도는 아직 도달하지 못했지만 요청 증가의 속도가 동시성 용량(concurrency capacity) 속도를 넘을 경우 쓰로틀링이 발생합니다. 이 경우 앞서 실행되고 있는 실행 환경에서의 작업이 끝나는대로, 혹은 1분 후 burst quota가 증가하는 대로 요청이 처리됩니다

