서버리스 애플리케이션 렌즈

AWS Well-Architected 프레임워크

2019 년 12 월

This paper has been archived.

The latest version is now available at:

https://docs.aws.amazon.com/ko_kr/wellarchitected/latest/serverless-applications-lens/welcome.html

고지 사항

고객에게는 본 문서에 포함된 정보를 독립적으로 평가할 책임이 있습니다. 본 문서는 (a) 정보 제공만을 위한 것이며, (b)사전 고지 없이 변경될 수 있는 현재의 AWS 제품 제공 서비스 및 사례를 보여 주며, (c) AWS 및 자회사, 공급업체 또는 라이선스 제공자로부터 어떠한 약정 또는 보증도 하지 않습니다. AWS 제품 또는 서비스는 명시적이든 묵시적이든 어떠한 종류의 보증, 진술 또는 조건 없이 "있는 그대로" 제공됩니다. 고객에 대한 AWS 의 책임과 법적 책임은 AWS 계약서에 준하며 본 문서는 AWS 와 고객 간의 어떠한 계약도 구성하거나 수정하지 않습니다.

목차

요약

이 문서에서는 AWS Well-Architected 프레임워크의 **서버리스 애플리케이션 렌즈**를 설명합니다. 일반적인 서버리스 애플리케이션 시나리오를 살펴보고, 모범 사례에 따라 워크로드 아키텍처를 설계하는 데 필요한 주요 요소를 알아봅니다.

소개

AWS Well-Architected 프레임워크는 AWS 에서 시스템을 구축할 때 내리는 의사 결정의 장점과 단점을 이해하는 데 도움이 됩니다.[1] 이 프레임워크를 사용하면 클라우드에서 안정적이고 안전하며 효율적이고 경제적인 시스템을 설계하고 운영하기 위한 설계 모범 사례를 알아볼 수 있습니다. 이 프레임워크는 모범 사례를 기준으로 아키텍처를 일관적으로 측정하고 개선 영역을 식별하는 방법을 제공합니다. 제대로 설계된 시스템을 갖추면 비즈니스 성공 가능성이 높아집니다.

이 "렌즈"에서는 AWS 클라우드 기반 **서버리스 애플리케이션 워크로드**의 설계, 배포 및 아키텍처 구성 방법을 집중적으로 살펴봅니다. 간결성을 위해 이 문서에서는 서버리스 워크로드와 관련된 Well-Architected 프레임워크의 세부 정보만 다룹니다. 그러나 아키텍처를 설계할 때는 이 문서에 포함되지 않은 모범 사례와 질문도 고려해야 합니다. AWS Well-Architected 프레임워크 백서를 읽어보십시오.[2]

이 문서는 CTO(최고 기술 책임자), 아키텍트, 개발자, 운영 팀 팀원 등 기술 업무 담당자를 위해 작성되었습니다. 이 문서를 읽으면 서버리스 애플리케이션의 아키텍처를 설계할 때 사용할 AWS 모범 사례와 전략을 이해할 수 있습니다.

정의

AWS Well-Architected 프레임워크는 운영 우수성, 보안, 안정성, 성능 효율성, 비용 최적화라는 5 가지 핵심 요소를 기반으로 합니다. AWS 는 서버리스 워크로드를 위한 다수의 핵심 구성 요소(서버리스 및 비 서버리스)를 제공합니다. 이러한 핵심 구성 요소를 사용하면 서버리스 애플리케이션을 위한 강력한 아키텍처를 설계할 수 있습니다. 이 섹션에는 이 문서 전체에서 사용할 서비스의 개요가 나와 있습니다. 서버리스 워크로드를 구축할 때는 7 가지 영역을 고려해야 합니다.

- 컴퓨팅 계층
- 데이터 계층
- 메시징 및 스트리밍 계층
- 사용자 관리 및 자격 증명 계층

- 엣지 계층
- 시스템 모니터링 및 배포
- 배포 접근 방식

컴퓨팅 계층

워크로드의 컴퓨팅 계층은 외부 시스템의 요청을 관리하여 액세스를 제어하고 요청이 적절히 승인될 수 있도록 합니다. 이 계층에는 비즈니스 로직이 배포되고 실행되는 실행 시간 환경이 포함됩니다.

AWS Lambda 를 사용하면 함수 계층에서 마이크로서비스 아키텍처, 배포 및 실행 관리를 지원하는 관리형 플랫폼에서 상태 비저장 서버리스 애플리케이션을 실행할 수 있습니다.

Amazon API Gateway 를 사용하면 Lambda 와 통합하여 비즈니스 로직을 실행하고, 트래픽 관리, 권한 부여 및 액세스 제어, 모니터링 및 API 버전 관리 기능을 제공하는 완전관리형 REST API 를 실행할 수 있습니다.

AWS Step Functions 는 서버리스 워크플로를 오케스트레이션합니다. 예를 들어 조정, 상태 및 함수 체이닝을 수행하고, Lambda 실행 제한 내에서 지원되지 않는 장기 실행을 여러 단계로 분할하거나 Amazon Elastic Compute Cloud(Amazon EC2) 인스턴스 또는 온프레미스에서 작업자를 호출하여 결합합니다.

데이터 계층

워크로드의 데이터 계층은 시스템 내부의 영구 스토리지를 관리합니다. 비즈니스 로직에 필요한 상태를 저장하는 보안 메커니즘과 데이터 변경에 대한 응답으로 이벤트를 트리거하는 메커니즘을 제공합니다.

Amazon DynamoDB 는 영구 스토리지를 위한 관리형 NoSQL 데이터베이스를 제공하여 서버리스 애플리케이션을 쉽게 구축할 수 있도록 합니다. **DynamoDB Streams** 와 함께 사용하면 Lambda 함수를 호출하여 DynamoDB 테이블의 변경 사항에 거의 실시간으로 응답할 수 있습니다. **DynamoDB Accelerator**(DAX)는 DynamoDB 용 고가용성 인메모리 캐시를 추가하여 밀리초 단위에서 마이크로초 단위로 최대 10 배 향상된 성능을 제공합니다.

Amazon Simple Storage Service(Amazon S3)는 **Amazon CloudFront** 같은 CDN(콘텐츠 전송 네트워크)을 통해 정적 자산을 처리하는 고가용성 키-값 스토어를 제공하여 서버리스 웹 애플리케이션 및 웹 사이트를 구축할 수 있도록 합니다.

Amazon Elasticsearch Service(Amazon ES)는 로그 분석, 전체 텍스트 검색, 애플리케이션 모니터링을 위해 Elasticsearch 를 손쉽게 배포, 보호, 운영 및 조정할 수 있는 기능을 제공합니다. Amazon ES 는 검색 엔진과 분석 도구를 모두 제공하는 완전관리형 서비스입니다.

AWS AppSync 는 실시간 및 오프라인 기능과 엔터프라이즈급 보안 제어를 통해 애플리케이션 개발을 간소화하는 관리형 GraphQL 서비스입니다. AWS AppSync 는 애플리케이션 및 디바이스에서 DynamoDB, Amazon ES 및 Amazon S3 와 같은 서비스에 연결할 때 사용되는 데이터 기반 API 와 일관적인 프로그래밍 언어를 제공합니다.

메시징 및 스트리밍 계층

워크로드의 메시징 계층은 구성 요소 간 통신을 관리합니다. 스트리밍 계층은 스트리밍 데이터의 실시간 분석 및 처리를 관리합니다.

Amazon Simple Notification Service(Amazon SNS)는 마이크로서비스, 분산 시스템 및 서버리스 애플리케이션을 위한 비동기식 이벤트 알림 및 모바일 푸시 알림을 사용하여 게시/구독 패턴에 대한 완전관리형 메시징 서비스를 제공합니다.

Amazon Kinesis 를 사용하면 실시간 스트리밍 데이터를 손쉽게 수집, 처리 및 분석할 수 있습니다. **Amazon Kinesis Data Analytics** 를 사용하면 표준 SQL 을 실행하거나 SQL 을 사용하여 전체 스트리밍 애플리케이션을 구축할 수 있습니다.

Amazon Kinesis Data Firehose 는 스트리밍 데이터를 캡처 및 변환하고 Kinesis Data Analytics, Amazon S3, Amazon Redshift 및 Amazon ES 로 로드하여 기존 비즈니스 인텔리전스 도구를 통한 근실시간 분석을 지원합니다.

사용자 관리 및 자격 증명 계층

워크로드의 사용자 관리 및 자격 증명 계층은 워크로드 인터페이스의 외부 및 내부 고객을 위한 자격 증명, 인증 및 권한 부여를 제공합니다.

Amazon Cognito 를 사용하면 서버리스 애플리케이션에 사용자 등록, 가입 및 데이터 동기화를 손쉽게 추가할 수 있습니다. **Amazon Cognito** 사용자 풀은 기본 제공되는 로그인 화면과 Facebook, Google, Amazon 및 SAML(Security Assertion Markup Language)과의 연동을 제공합니다. **Amazon Cognito 연동 자격 증명**을 사용하면 서버리스 아키텍처의 일부인 AWS 리소스에 대해 범위가 지정된 액세스를 안전하게 제공할 수 있습니다.

엣지 계층

워크로드의 엣지 계층은 프레젠테이션 계층과 외부 고객에 대한 연결성을 관리합니다. 별개의 지리적 위치에 상주하는 외부 고객에게 효율적인 전송 방법을 제공합니다.

Amazon CloudFront 는 짧은 지연 시간과 빠른 전송 속도로 웹 애플리케이션 콘텐츠와 데이터를 안전하게 전송하는 CDN 을 제공합니다.

시스템 모니터링 및 배포

워크로드의 시스템 모니터링 계층은 지표를 통해 시스템 가시성을 관리하고 시스템의 시간대별 운영 및 작동을 컨텍스트를 기반으로 인식합니다. 배포 계층은 릴리스 관리 프로세스를 통해 워크로드 변경을 승격하는 방법을 정의합니다.

Amazon CloudWatch 를 사용하면 사용하는 모든 AWS 서비스의 시스템 지표에 액세스하고, 시스템 및 애플리케이션 수준 로그를 통합하고, 비즈니스 KPI(핵심 성능 지표)를 특정 요구 사항에 대한 사용자 지정 지표로 생성할 수 있습니다. 이 서비스는 플랫폼에서 자동화된 작업을 트리거할 수 있는 대시보드 및 알림을 제공합니다.

AWS X-Ray 에서는 분산 추적 및 서비스 맵을 통해 요청을 전체적으로 시각화하여 성능 병목 현상을 손쉽게 식별함으로써 서버리스 애플리케이션을 분석하고 디버깅할 수 있습니다.

AWS Serverless Application Model(AWS SAM)은 서버리스 애플리케이션을 패키징, 테스트 및 배포하는 데 사용되는 AWS CloudFormation 의 확장입니다. 또한 AWS SAM CLI 를 사용하면 Lambda 함수를 로컬에서 개발할 때 디버깅 주기를 가속화할 수 있습니다.

배포 접근 방식

마이크로서비스 아키텍처의 배포 접근 방식은 변경 사항이 소비자의 서비스 계약을 위반하지 않도록 하는 것입니다. API 소유자가 서비스 계약을 위반하는 변경을 수행하고 소비자가 이 변경에 준비되어 있지 않으면 오류가 발생할 수 있습니다.

안전한 배포를 위한 첫 번째 단계는 소비자가 어떤 API 를 사용하고 있는지 아는 것입니다. 소비자와 소비자의 사용량에 대한 메타데이터를 수집하면 데이터를 기반으로 변경의 영향에 대한 의사 결정을 내릴 수 있습니다. API 키는 API 소비자/클라이언트에 대한 메타데이터를 캡처하는 효과적인 방법이며 API 에 호환성 손상을 야기하는 변경이 발생하는 경우 연락처의 형태로 종종 사용됩니다.

호환성이 손상되는 변경에 대한 위험을 방지하는 접근 방식을 취하려는 고객은 API 를 복제하고 고객을 다른 하위 도메인(예: v2.my-service.com)으로 라우팅하여 기존 고객이 영향을 받지 않도록 할 수 있습니다. 이 접근 방식에서는 새로운 배포를 새로운 서비스 계약으로 지원할 수 있지만 이중 API(및 차후의 백엔드 인프라)를 유지하는 오버헤드로 인해 추가 오버헤드가 발생한다는 단점이 있습니다.

다음 표에는 배포에 대한 다양한 접근 방식이 나와 있습니다.

배포	소비자 영향	롤백	이벤트 모델 요소	배포 속도
한 번에 모두	한 번에 모두	이전 버전 다시 배포	동시성 비율이 낮은 모든 이벤트 모델	즉시
블루/그린	일정 수준의 프로덕션 환경 테스트를 사전에 수행한 후 한 번에 모두	트래픽을 이전 환경으로 되돌리기	동시성 워크로드가 중간인 비동기식 및 동기식 이벤트 모델에 적합	몇 분에서 몇 시간의 검증을 거친 후 고객에게 즉시
Canary/선형	1~10%의 일반적인 초기 트래픽 이동 후 단계별로 늘리거나 한 번에 모두	트래픽의 100%를 이전 배포로 되돌리기	동시성이 높은 워크로드에 적합	몇 분에서 몇 시간 사이

한 번에 모두 배포

한 번에 모두 배포 접근 방식에서는 기존 구성을 변경하는 작업이 포함됩니다. 이 배포 스타일의 장점은 관계형 데이터베이스 같은 데이터 스토어에 대한 백엔드 변경에서 훨씬 적은 노력으로 변경 주기 중에 트랜잭션을 조정할 수 있다는 것입니다. 이 유형의 배포 스타일은 작업이 쉽고 동시성이 낮은 모델에서 영향을 최소화할 수 있지만 롤백 시 위험이 추가되고 일반적으로 다운타임이 발생합니다. 이 배포 모델은 사용자 영향이 최소한인 개발 환경과 같은 시나리오에서 사용됩니다.

블루/그린 배포

또 다른 트래픽 이동 패턴은 블루/그린 배포를 활성화하는 것입니다. 다운타임이 거의 없는 이 릴리스를 사용하면 롤백이 필요할 경우를 대비하여 이전 프로덕션 환경(블루)을 예비 상태로 유지하면서 새 라이브 환경(그린)으로 트래픽을 이동할 수 있습니다. 이 스타일의 배포에서는 API Gateway 를 사용하여 특정 환경으로 이동하는 트래픽의 백분율을 정의할 수 있으므로 효과적일 수 있습니다. 블루/그린 배포는 다운타임을 줄이도록 설계되었기 때문에 많은 고객이 프로덕션 변경에 이러한 패턴을 채택하고 있습니다.

상태 비저장 및 멱등성의 모범 사례를 따르는 서버리스 아키텍처는 기반 인프라에 대한 선호도가 없기 때문에 이 배포 스타일에 적합합니다. 필요한 경우 작업 중인 환경으로 쉽게 롤백할 수 있도록 이러한 배포를 더 작은 증분 변경으로 보정해야 합니다.

롤백이 필요한지 여부를 알려면 올바른 지표가 필요합니다. 모범 사례로, CloudWatch 의 고해상도 지표를 사용하는 것이 좋습니다. 이러한 지표는 1 초 간격의 모니터링을 통해 하향 추세를 빠르게 캡처할 수 있습니다. CloudWatch 경보와 함께 사용하면 긴급 롤백을 수행할 수 있습니다. API Gateway, Step Functions, Lambda(사용자 지정 지표 포함) 및 DynamoDB 에서 CloudWatch 지표를 캡처할 수 있습니다.

Canary 배포

Canary 배포는 소프트웨어의 새 릴리스를 제어된 환경에서 활용하여 배포 주기를 가속화하기 위해 점점 더 많이 사용되고 있는 방법 중 하나입니다. Canary 배포에서는 새 변경에 대한 소수의 요청을 배포하여 소수의 사용자에게 미치는 영향을 분석합니다. AWS 클라우드에서는 새로운 배포의 기본 인프라 프로비저닝 및 확장에 대해 더 이상 걱정할 필요가 없으므로 이 배포의 채택을 촉진하는 데 도움이 되었습니다.

API Gateway 에서 Canary 배포를 사용하는 경우 소비자를 위한 동일한 API Gateway HTTP 엔드포인트를 유지하면서 백엔드 엔트포인트(예: Lambda)에 변경을 배포할 수 있습니다. 또한 새 배포로 라우팅되는 트래픽 비율과 제어된 트래픽 전환 비율을 제어할 수 있습니다. Canary 배포는 실제로 새로운 웹 사이트 같은 시나리오에서 사용될 수 있습니다. 모든 트래픽을 새 배포로 이동하기 전에 소수의 최종 사용자에 대한 클릭 수를 모니터링할 수 있습니다.

Lambda 버전 제어 사용

모든 소프트웨어와 마찬가지로 버전 관리를 유지하면 이전의 작동하는 코드를 빠르게 파악하고 새 배포에 실패할 경우 이전 버전으로 되돌릴 수 있습니다. Lambda 를 사용하면 이전 버전을 변경할 수 없도록 개별 Lambda 함수의 변경 불가능한 버전을 하나 이상 게시할 수 있습니다. 각 Lambda 함수 버전에는 고유한 Amazon 리소스 이름(ARN)이 있으며 새로운 버전 변경은 CloudTrail 에 기록될 때 감사 가능합니다. 프로덕션에서는 모범 사례로, 버전 관리를 활성화하여 안정적인 아키텍처의 활용을 최대화해야 합니다.

배포 작업을 간소화하고 오류 위험을 줄이려면 Lambda 별칭을 사용하여 개발 워크플로의 Lambda 함수를 다양하게 변형할 수 있습니다(예: 개발, 베타 및 프로덕션). 예를 들어 API Gateway 와 Lambda 를 통합하여 프로덕션 별칭의 ARN 을 가리킬 수 있습니다. 프로덕션 별칭은 Lambda 버전을 가리킵니다. 이 기술의 가치는 호출자 구성 내의 Lambda 별칭이 정적으로 유지되므로 변경이 줄어들고 새 버전을 라이브 환경으로 승격할 때 안전한 배포가 가능하다는 데 있습니다.

일반 설계 원칙

Well-Architected 프레임워크는 클라우드에서 서버리스 애플리케이션을 제대로 설계할 수 있게 하는 다음과 같은 일반적인 설계 원칙을 식별합니다.

- **단일 용도의 빠르고 단순한 함수를 사용할 것**: 함수는 간결하고 짧으며 단일 용도여야 하고 함수 환경은 함수의 요청 수명 주기에 따릅니다. 트랜잭션 비용은 효율적으로 인식되므로 빠른 실행이 선호됩니다.
- **총 요청 수가 아닌 동시 요청 수를 고려할 것**: 서버리스 애플리케이션은 동시성 모델을 활용하며 설계 수준에서 장/단점은 동시성을 기준으로 평가됩니다.

- **어떤 것도 공유하지 말 것**: 함수 실행 시간 환경 및 기반 인프라의 수명이 짧으므로 임시 스토리지 같은 로컬 리소스가 보장되지 않습니다. 상태는 상태 시스템 실행 수명 주기 내에서 조작 가능하며 내구력을 높이려면 영구 스토리지를 사용하는 것이 좋습니다.
- **하드웨어 선호도를 가정하지 말 것**: 기반 인프라는 변경될 수 있습니다. 따라서 하드웨어에 기반을 두지 않는 코드 또는 종속성을 활용해야 합니다. 예를 들어 CPU 플래그는 일관적으로 제공되지 않을 수 있습니다.
- **애플리케이션 오케스트레이션에 함수 대신 상태 시스템을 사용할 것**: 코드 안에서 Lambda 실행을 연결하여 애플리케이션의 워크플로를 오케스트레이션하면 긴밀하게 결합된 단일 구조의 애플리케이션이 만들어집니다. 그러므로 상태 시스템을 대신 사용하여 트랜잭션 및 통신 흐름을 오케스트레이션하는 것이 좋습니다.
- **이벤트를 사용하여 트랜잭션을 트리거할 것**: 새 Amazon S3 객체 또는 업데이트를 데이터베이스에 쓰는 것과 같은 이벤트를 사용하여 비즈니스 기능에 대한 응답으로 트랜잭션을 실행할 수 있습니다. 이 비동기식 이벤트 동작은 모든 소비자에게 적용이 가능한 경우가 많으며 정시 처리를 촉진하여 간결한 서비스 설계를 보장합니다.
- **오류 및 중복성을 고려하여 설계할 것**: 요청/이벤트에서 트리거되는 작업은 멱등성이 있어야 합니다. 오류가 발생할 경우 지정된 요청/이벤트가 한 번 이상 전송될 수 있기 때문입니다. 다운스트림 호출에 적절한 재시도를 포함하십시오.

시나리오

이 섹션에서는 많은 서버리스 애플리케이션에서 흔히 볼 수 있는 5 가지 주요 시나리오와 이러한 시나리오가 AWS 기반 서버리스 애플리케이션 워크로드의 설계 및 아키텍처에 미치는 영향을 다룹니다. 각 시나리오에 대한 가정, 설계의 일반적인 동인 및 이러한 시나리오를 구현하는 방법에 대한 참조 아키텍처가 제시됩니다.

RESTful 마이크로서비스

마이크로서비스를 구축할 때는 비즈니스 컨텍스트를 재사용 가능한 서비스로 소비자에게 제공할 방법을 고려합니다. 특정 구현은 개별 사용 사례에 맞게 조정되겠지만 마이크로서비스에는 구현의 보안, 복원력 및 최상의 고객 경험을 보장하기 위한 몇 가지 공통된 주제가 있습니다.

AWS 에서 서버리스 마이크로서비스를 구축하면 서버리스 기능 자체를 활용하는 것은 물론, 다른 AWS 서비스 및 기능과 AWS 및 APN(AWS 파트너 네트워크) 도구의 에코시스템까지 활용할 수 있습니다. 서버리스 기술은 내결함성 인프라를 기반으로 구축되므로 미션 크리티컬 워크로드를 위한 안정적인 서비스를 구축할 수 있습니다. 또한 도구 에코시스템을 활용하여 구축을 간소화하고, 작업을 자동화하고, 종속성을 오케스트레이션하고, 마이크로 서비스를 모니터링 및 관리할 수 있습니다. 마지막으로, AWS 서버리스 도구는 사용량에 따라 요금을 지불하므로 비즈니스에 맞춰 서비스를 확장하고 초기 단계와 피크가 아닌 시간에는 비용을 절감할 수 있습니다.

특성:

- 안전하고 조작이 쉬우며 간단히 복제할 수 있고 높은 수준의 복원력 및 가용성을 제공하는 프레임워크가 필요합니다.
- 사용률 및 액세스 패턴을 기록하여 고객 사용량을 지원할 수 있도록 백엔드를 지속적으로 개선해야 합니다.
- 관리형 서비스를 플랫폼에 최대한 활용하여 보안 및 확장성 등 일반적인 플랫폼 관리와 관련된 힘든 작업을 줄여야 합니다.

참조 아키텍처

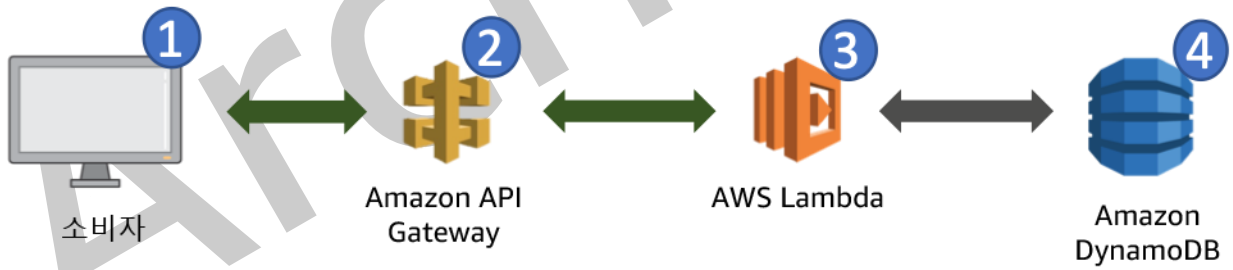

그림 1: RESTful 마이크로서비스의 참조 아키텍처

1. **고객**은 HTTP API 호출을 통해 마이크로서비스를 활용합니다. 이상적으로, 서비스 수준 및 변경 제어에 대한 일관적인 기대치를 달성하려면 소비자의 서비스 계약이 API 에 긴밀하게 연결되어야 합니다.
2. **Amazon API Gateway** 는 RESTful HTTP 요청 및 응답을 고객에게 호스팅합니다. 이 시나리오에서 API Gateway 는 기본적으로 권한 부여, 조절, 보안, 내결함성, 요청/응답 매핑 및 성능 최적화를 제공합니다.

3. **AWS Lambda** 에는 수신 API 호출을 처리하고 DynamoDB 를 영구 스토리지로 활용하는 비즈니스 로직이 포함됩니다.
4. **Amazon DynamoDB** 는 마이크로서비스 데이터를 영구적으로 저장하고 필요에 따라 확장합니다. 마이크로서비스는 한 가지 작업을 수행하도록 설계되기 때문에 스키마가 없는 NoSQL 데이터 스토어가 주기적으로 통합됩니다.

구성 참고 사항:

- API Gateway 로깅을 활용하여 마이크로서비스 소비자의 액세스 동작을 파악합니다. 이 정보는 Amazon CloudWatch Logs 에 표시되며 Log Pivots 를 통해 빠르게 확인하거나 CloudWatch Logs Insights 에서 분석하거나 Amazon ES 또는 Amazon S3(Amazon Athena 사용) 같은 검색 엔진에 제공할 수 있습니다. 전달된 정보는 다음과 같은 주요 가시성을 제공합니다.
    - 백엔드 근접성에 따라 지리적으로 변경될 수 있는 일반적인 고객 위치
    - 고객 입력 요청이 데이터베이스 파티셔닝 방법에 미칠 수 있는 영향
    - 보안 플래그가 될 수 있는 비정상 동작의 의미
    - 구성 최적화에 필요한 오류, 지연 시간 및 캐시 적중/누락에 대한 정보

이 모델은 필요에 따라 확장되는 안전한 환경을 손쉽게 배포하고 유지 관리할 수 있는 프레임워크를 제공합니다.

Alexa 스킬

Alexa 스킬 키트는 자연스럽고 호감이 가는 음성 및 시각적 경험을 구축하여 Alexa 의 기능을 확장할 수 있는 기능을 제공합니다. 성공적인 스킬은 고유한 기능을 제공하여 사용자의 일상적이고 습관적인 사용을 유도하며 새롭고 참신하며 원활한 방법으로 가치를 제공합니다.

사용자는 스킬이 예상대로 작동하지 않을 때와 필요한 작업을 달성하기까지 여러 번의 상호 작용이 필요한 경우 가장 큰 불만을 느낍니다. 일부 사용자는 너무 적게 말하거나 많이 말하거나 예상 외의 말을 할 수 있으므로 음성 상호 작용 모델을 설계하고 이 설계에서 역순으로 작업하는 것이 필수적입니다. 음성 설계 프로세스에는 예상되는 표현과 예상 외의 표현을 생성하고 스크립트로 만들고 계획하는 작업이 포함됩니다.

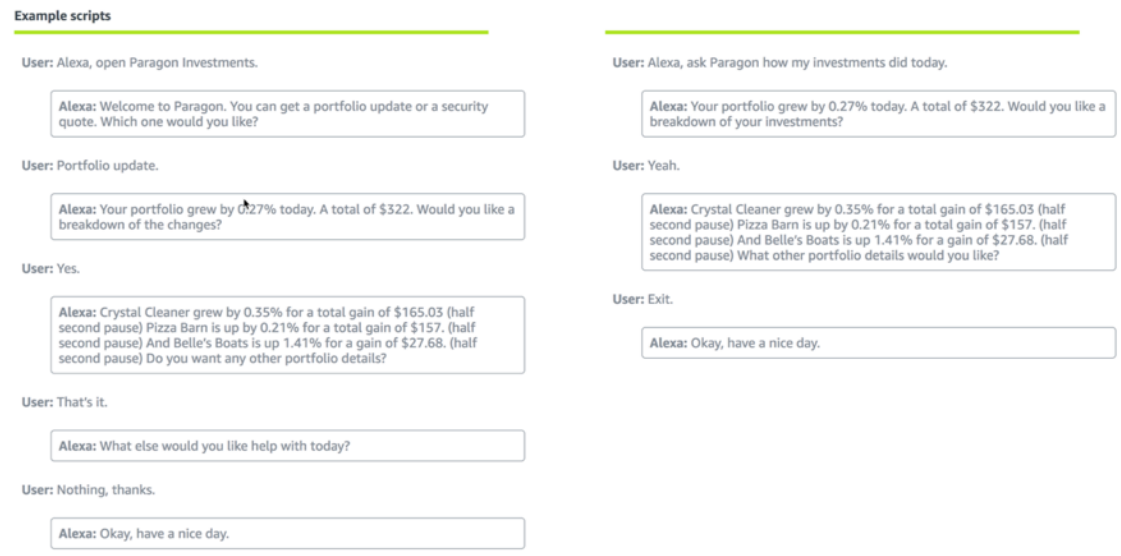

그림 2: Alexa 스킬 예제 설계 스크립트

기본 스크립트를 바탕으로 다음과 같은 기술을 사용하여 스킬 구축을 시작할 수 있습니다.

- **최단 완료 경로 명시**
    - 최단 완료 경로는 일반적으로 스킬의 단일 호출에서 사용자가 모든 정보와 슬롯을 한 번에 제공하고, 관련 계정이 이미 연결되어 있고, 다른 사전 조건이 충족될 때입니다.
- **대체 경로 및 의사 결정 트리 명시**
    - 사용자가 말하는 내용에 요청을 완료하는 데 필요한 모든 정보가 포함되지 않는 경우가 종종 있습니다. 따라서 흐름에서 대체 경로와 사용자 의사 결정을 식별해야 합니다.
- **시스템 로직을 통해 자동으로 수행할 의사 결정 명시**
    - 자동 시스템 의사 결정을 식별합니다(예: 신규 또는 재방문 사용자에 대한 시스템 의사 결정). 백그라운드 시스템 확인을 통해 사용자가 따르는 흐름이 변경될 수 있습니다.
- **스킬이 사용자에게 제공하는 도움 명시**
    - 사용자가 스킬로 할 수 있는 것에 대한 분명한 도움의 방향을 포함합니다. 스킬의 복잡성에 따라 도움에서 단순한 하나의 응답 또는 다수의 응답이 제공될 수 있습니다.

- **계정 연결 프로세스 명시(있는 경우)**
    - 계정 연결에 필요한 정보를 결정합니다. 또한 계정 연결이 완료되지 않은 경우 스킬의 응답 방법도 식별해야 합니다.

특성:

- 모든 인스턴스 또는 서버 관리 없이 완전한 서버리스 아키텍처를 생성해야 합니다.
- 콘텐츠를 최대한 스킬에서 분리해야 합니다.
- API 로 노출되는, 호감이 가는 음성 경험을 제공하여 다양한 범위의 Alexa 디바이스, 리전 및 언어를 사용한 개발 환경을 최적화해야 합니다.
- 사용자 수요에 따라 확장 및 축소되고 예기치 않은 사용량 패턴을 처리하는 탄력성이 필요합니다.

참조 아키텍처

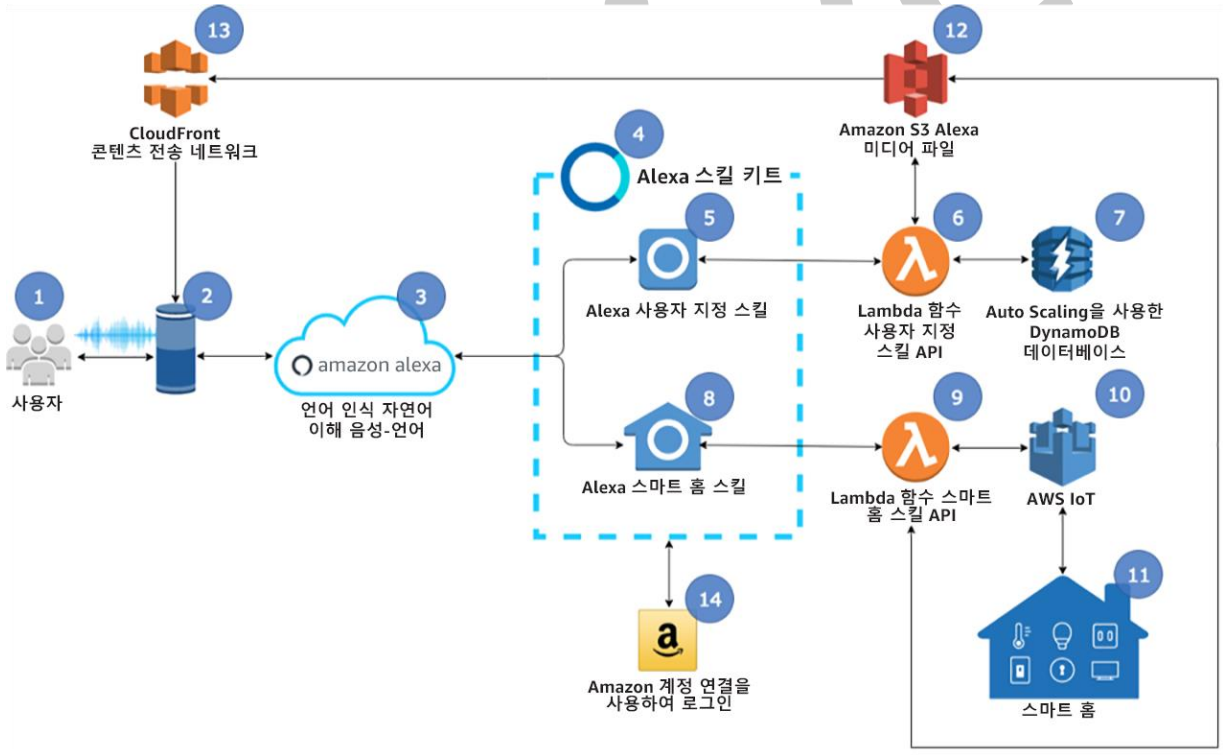

그림 3: Alexa 스킬의 참조 아키텍처

1. **Alexa 사용자**는 음성을 기본적인 상호 작용 방법으로 사용하여 Alexa 지원 디바이스에 말을 함으로써 Alexa 스킬과 상호 작용합니다.
2. **Alexa 지원 디바이스**는 실행 단어의 수신을 대기하다가 실행 단어가 인식되는 즉시 활성화됩니다. 지원되는 실행 단어는 알렉사, 컴퓨터 및 에코입니다.
3. **Alexa 서비스**는 Alexa 스킬을 대신하여 일반적인 SLU(언어 이해) 처리를 수행하는데 여기에는 ASR(자동 언어 인식), NLU(자연어 이해) 및 TTS(텍스트-언어) 변환이 포함됩니다.
4. **ASK(Alexa 스킬 키트)**는 셀프 서비스 API, 도구, 설명서 및 코드 예제의 컬렉션으로, 이 키트를 사용하면 빠르고 쉽게 Alexa 에 스킬을 추가할 수 있습니다. ASK 는 신뢰할 수 있는 AWS Lambda 트리거로, 원활한 통합을 가능하게 합니다.
5. **Alexa 사용자 지정 스킬**을 사용하면 사용자 경험을 제어하여 사용자 지정 상호 작용 모델을 구축할 수 있습니다. 사용자 지정 스킬은 가장 유연하지만 가장 복잡한 유형의 스킬입니다.
6. **Lambda 함수**에 Alexa 스킬 키트를 사용하면 불필요한 복잡성 없이 스킬을 원활하게 구축할 수 있습니다. Alexa 서비스에서 전송되는 다양한 유형의 요청을 처리하고 언어 응답을 구축할 수 있습니다.
7. **DynamoDB 데이터베이스**는 스킬 사용량에 따라 탄력적으로 조정되는 NoSQL 데이터 스토어를 제공합니다. 스킬에서 주로 사용자 상태 및 세션을 유지하는 데 사용됩니다.
8. **Alexa 스마트 홈 스킬**을 사용하면 스마트 홈 API 를 사용하여 조명, 온도 조절 장치, 스마트 TV 등을 제어할 수 있습니다. 스마트 홈 스킬의 경우 상호 작용 모델을 제어할 수 없으므로 사용자 지정 스킬보다 구축이 간단합니다.
9. **Lambda 함수**는 Alexa 서비스의 디바이스 검색 및 제어 요청에 응답하는 데 사용됩니다. 개발자는 이 함수를 사용하여 엔터테인먼트 디바이스, 카메라, 조명, 온도 조절 장치, 잠금 장치 등 다양한 디바이스를 제어할 수 있습니다.
10. 개발자는 **AWS Internet of Things(IoT)**를 사용하여 디바이스를 AWS 에 안전하게 연결하고 Alexa 스킬과 디바이스 간의 상호 작용을 제어할 수 있습니다.
11. Alexa 지원 **스마트 홈**에는 Alexa 스킬의 명령을 수신하고 이에 응답하는 무제한의 IoT 커넥티드 디바이스가 포함될 수 있습니다.

12. **Amazon S3** 에는 이미지, 콘텐츠, 미디어 등의 스킬 정적 자산이 저장됩니다. S3 의 콘텐츠는 CloudFront 를 사용하여 안전하게 제공됩니다.
13. **Amazon CloudFront CDN**(콘텐츠 전송 네트워크)는 지리적으로 분산된 모바일 사용자에게 콘텐츠를 더 빠르게 제공하고 Amazon S3 의 정적 자산에 대한 보안 메커니즘을 포함하는 CDN 을 제공합니다.
14. **계정 연결**은 스킬에서 다른 시스템을 통해 인증해야 하는 경우 필요합니다. 이 작업은 Alexa 사용자를 다른 시스템의 특정 사용자에 연결합니다.

구성 참고 사항:

- 스킬을 통해 Alexa 로 전송되는 모든 가능한 Alexa 스마트 홈 메시지의 JSON 스키마를 기준으로 유효성 검사를 수행하여 스마트 홈의 요청 및 응답 페이로드를 검증합니다.
- Lambda 함수 제한 시간이 8 초 미만이고 이 기간 내에 요청을 처리할 수 있는지 확인합니다. Alexa 서비스의 제한 시간은 8 초입니다.
- DynamoDB 테이블을 생성할 때는 모범 사례 [7] 을 따릅니다. 필요한 읽기/쓰기 용량을 잘 모르는 경우 온디맨드 테이블을 사용합니다. 그렇지 않을 경우 자동 조정이 활성화된 프로비저닝된 용량을 선택합니다. 준비 작업이 많은 스킬의 경우 DynamoDB Accelerator(DAX)를 사용하여 응답 시간을 크게 개선할 수 있습니다.
- 계정 연결을 사용하면 외부 시스템에 저장 가능한 사용자 정보가 제공됩니다. 이 정보를 사용하여 사용자에게 컨텍스트에 적합하고 개인화된 환경을 제공할 수 있습니다. Alexa 의 계정 연결 지침을 사용하여 원활한 경험을 제공하십시오.
- 스킬 베타 테스트 도구를 사용하여 스킬 개발에 대한 피드백을 조기에 수집하고 스킬 버전을 관리하여 이미 라이브 상태인 스킬에 미치는 영향을 최소화할 수 있습니다.
- ASK CLI 를 사용하여 스킬 개발 및 배포를 자동화할 수 있습니다.

모바일 백엔드

빠르고 일관적이며 풍부한 기능의 사용자 경험을 갖춘 모바일 애플리케이션을 기대하는 사용자가 늘어나고 있습니다. 동시에 모바일 사용자 패턴은 피크 사용량을 예측할 수 없는 동적 패턴이며 글로벌 사용자 기반을 보유한 경우가 많습니다.

모바일 사용자의 수요 증가는 백엔드 인프라의 제어 및 유연성을 저해하지 않으면서 원활하게 상호 작용하는 풍부한 모바일 서비스가 애플리케이션에 필요하다는 것을 의미합니다. 모바일 애플리케이션 전반의 특정 기능은 기본적으로 다음과 같습니다.

- 데이터베이스 변경을 쿼리, 변형 및 구독할 수 있는 기능
- 연결 시 데이터를 오프라인으로 유지하고 대역폭을 최적화
- 애플리케이션 데이터의 검색, 필터링 및 검색
- 사용자 동작의 분석
- 여러 채널을 통한 대상 지정 메시징(푸시 알림, SMS, 이메일)
- 이미지 및 비디오 같은 리치 콘텐츠
- 여러 디바이스 및 여러 사용자 전체의 데이터 동기화
- 데이터 보기 및 조작에 대한 세분화된 권한 부여 제어

AWS 에서 서버리스 모바일 백엔드를 구축하면 이러한 기능을 제공하는 동시에 확장성, 탄력성 및 가용성을 효율적이고 경제적인 방법으로 자동으로 관리할 수 있습니다.

특성:

- 클라이언트의 애플리케이션 데이터 동작을 제어하고 API 를 사용하여 얻으려는 데이터를 명시적으로 선택할 수 있어야 합니다.
- 비즈니스 로직을 모바일 애플리케이션에서 최대한 분리해야 합니다.
- 비즈니스 기능을 API 로 제공하여 여러 플랫폼 전반의 개발을 최적화할 수 있어야 합니다.
- 관리형 서비스를 활용하여 모바일 백엔드 인프라 관리와 관련된 차별화되지 않은 힘든 작업을 줄이는 동시에 높은 수준의 확장성 및 가용성을 제공해야 합니다.
- 실제 사용자 수요와 유휴 리소스에 대해 지불하는 비용을 비교하여 모바일 백엔드 비용을 최적화해야 합니다.

참조 아키텍처

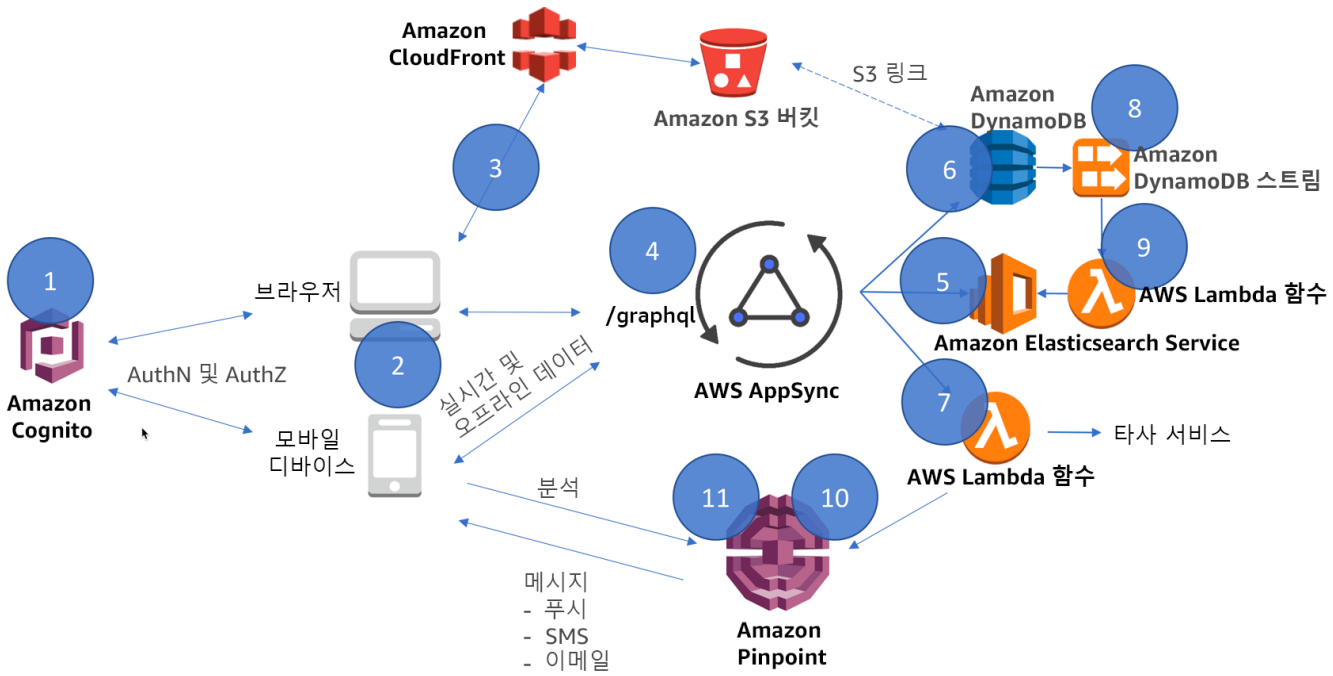

그림 4: 모바일 백엔드의 참조 아키텍처

1. **Amazon Cognito** 는 사용자 관리에 사용되며 모바일 애플리케이션의 자격 증명 공급자 역할을 합니다. 모바일 사용자는 Facebook, Twitter, Google+및 Amazon 과 같은 기존 소셜 자격 증명을 활용하여 로그인할 수 있습니다.
2. **모바일 사용자**는 AWS AppSync 및 AWS 서비스 API(예: Amazon S3 및 Amazon Cognito)에 대해 GraphQL 작업을 수행하여 모바일 애플리케이션 백엔드와 상호 작용합니다.
3. **Amazon S3** 는 프로필 이미지와 같은 특정 모바일 사용자 데이터를 비롯한 모바일 애플리케이션 정적 자산을 저장합니다. S3 의 콘텐츠는 CloudFront 를 통해 안전하게 제공됩니다.
4. **AWS AppSync** 는 모바일 사용자에 대한 GraphQL HTTP 요청 및 응답을 호스팅합니다. 이 시나리오에서 AWS AppSync 의 데이터는 디바이스가 연결될 때 실시간으로 실행되며 오프라인에서도 데이터를 사용할 수 있습니다. 이 시나리오의 데이터 소스는 **Amazon DynamoDB**, **Amazon Elasticsearch Service**, 또는 **AWS Lambda** 함수입니다.

5. **Amazon Elasticsearch Service** 는 모바일 애플리케이션 및 분석을 위한 기본 검색 엔진의 역할을 합니다.
6. **DynamoDB** 는 TTL(Time to Live) 기능을 통해 비활성 모바일 사용자의 불필요한 데이터를 만료시키는 메커니즘을 포함하여 모바일 애플리케이션을 위한 영구 스토리지를 제공합니다.
7. **Lambda** 함수는 다른 타사 서비스와의 상호 작용 또는 클라이언트에 대한 GraphQL 응답의 일부일 수 있는 사용자 지정 흐름에 대한 다른 AWS 서비스 호출을 처리합니다.
8. **DynamoDB Streams** 는 항목 수준 변경 사항을 캡처하고 Lambda 함수를 사용하여 추가 데이터 원본을 업데이트합니다.
9. **Lambda** 함수는 DynamoDB 와 Amazon ES 간의 스트리밍 데이터를 관리하여 고객이 데이터 원본의 논리적 GraphQL 유형 및 작업을 결합할 수 있도록 합니다.
10. **Amazon Pinpoint** 는 사용자 세션 및 애플리케이션 통찰력에 대한 사용자 지정 지표 등 클라이언트의 분석을 캡처합니다.
11. **Amazon Pinpoint** 는 수집된 분석을 기반으로 모든 사용자/디바이스 또는 대상 하위 집합에 메시지를 전송합니다. 메시지는 사용자 지정이 가능하며 푸시 알림, 이메일 또는 SMS 채널을 사용하여 전송될 수 있습니다.

구성 참고 사항:

- 다양한 메모리 및 제한 시간 설정으로 Lambda 함수의 성능을 테스트[3]하여 작업에 가장 적합한 리소스를 사용할 수 있도록 합니다.
- DynamoDB 테이블을 생성할 때는 모범 사례[4]를 따르십시오. 또한 AWS AppSync 를 통해 적절히 분산된 해시 키를 사용하고 작업에 대한 인덱스를 생성하는 GraphQL 스키마에서 테이블을 자동으로 프로비저닝하는 것이 좋습니다. 적절한 응답 시간을 보장하려면 읽기/쓰기 용량과 테이블 파티셔닝을 계산해야 합니다.
- AWS AppSync 서버 측 데이터 캐싱을 사용하여 애플리케이션 경험을 최적화하십시오. 이렇게 하면 API 에 대한 모든 후속 쿼리 요청이 캐시에서 반환되므로 TTL 이 만료되기 전까지 데이터 원본에 직접 연결하지 않아도 됩니다.
- Amazon ES 도메인을 관리할 때는 모범 사례[5]를 따릅니다. 또한 Amazon ES 는 여기에도 적용되는 샤딩 및 액세스 패턴의 설계에 관한 방대한 안내서[6]를 제공합니다.

- 필요한 경우 확인자로 구성되는 AWS AppSync 의 세분화된 액세스 제어를 사용하여 GraphQL 요청을 사용자 단위 또는 그룹 수준으로 필터링합니다. 이 참고 사항은 AWS Identity and Access Management(IAM) 또는 Amazon Cognito 사용자 풀 권한 부여(AWS AppSync 사용)에도 적용될 수 있습니다.
- AWS Amplify 및 Amplify CLI 를 사용하여 애플리케이션을 작성하고 다수의 AWS 서비스와 통합합니다. 또한 Amplify 콘솔에서 스택 배포 및 관리 작업을 처리할 수도 있습니다.

비즈니스 로직이 거의 필요하지 않은 상황에서 짧은 지연 시간에 대한 요구 사항을 충족해야 하는 경우, Amazon Cognito 연동 자격 증명을 사용하여 범위가 지정된 자격 증명을 제공하면 모바일 애플리케이션에서 직접 AWS 서비스와 통신할 수 있습니다. 예를 들어 사용자의 프로필 사진을 업로드하거나 Amazon S3 범위의 검색 메타데이터 파일을 사용자에게 업로드할 수 있습니다.

스트림 처리

실시간 스트리밍 데이터를 수집하고 처리하려면 활동 추적, 트랜잭션 주문 처리, 클릭스트림 분석, 데이터 정리, 지표 생성, 로그 필터링, 인덱싱, 소셜 미디어 분석 및 IoT 디바이스 데이터 원격 분석 및 측정 등 다양한 애플리케이션을 지원할 수 있는 확장성과 짧은 지연 시간이 필요합니다. 이러한 애플리케이션은 종종 사용량이 급증하며 초당 수천 개의 이벤트를 처리하기도 합니다.

AWS Lambda 및 Amazon Kinesis 를 사용하면 서버를 프로비저닝하거나 관리할 필요 없이 자동으로 조정되는 서버리스 스트림 프로세스를 구축할 수 있습니다. AWS Lambda 로 처리된 데이터를 DynamoDB 에 저장하고 나중에 분석할 수 있습니다.

특성:

- 모든 인스턴스 또는 서버 관리 없이 스트리밍 데이터를 처리할 수 있는 완전한 서버리스 아키텍처를 생성해야 합니다.
- Amazon Kinesis Producer Library(KPL)를 사용하여 데이터 생산자 관점에서 데이터 수집을 처리해야 합니다.

참조 아키텍처

여기에는 소셜 미디어 데이터의 분석을 위한 참조 아키텍처인 일반적인 스트림 처리에 대한 시나리오가 나와 있습니다.

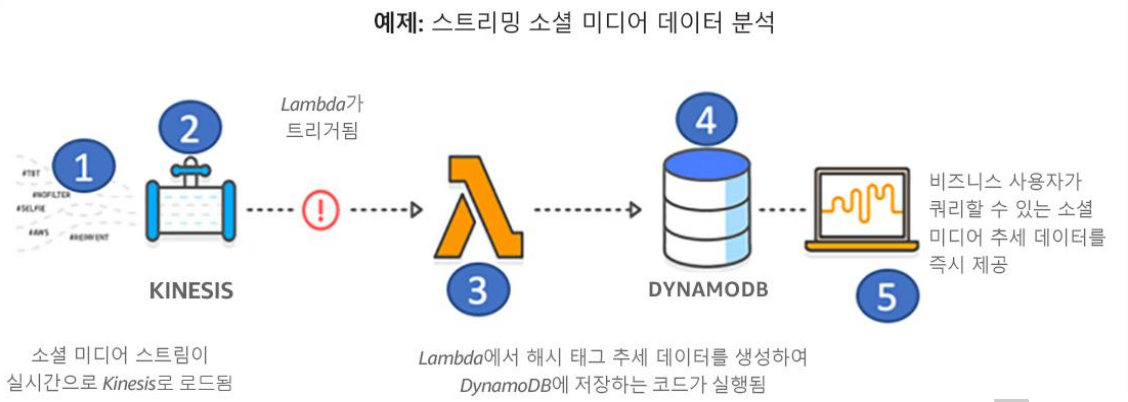

그림 5: 스트림 처리의 참조 아키텍처

1. **데이터 생산자**는 Amazon Kinesis Producer Library(KPL)를 사용하여 소셜 미디어 스트리밍 데이터를 Kinesis 스트림으로 전송합니다. Kinesis API 를 활용하는 Amazon Kinesis 에이전트 및 사용자 지정 데이터 생산자를 사용할 수도 있습니다.
2. **Amazon Kinesis 스트림**은 데이터 생산자가 생산하는 실시간 스트리밍 데이터를 수집, 처리 및 분석합니다. 소비자(이 경우 Lambda)는 스트림으로 수집된 데이터를 처리할 수 있습니다.
3. **AWS Lambda** 는 스트림의 소비자 역할로서, 다수의 수집된 데이터를 단일 이벤트/호출로 수신합니다. 추가 처리는 Lambda 함수를 통해 수행되며 변환된 데이터는 영구 스토리지(이 경우 DynamoDB)에 저장됩니다.
4. **Amazon DynamoDB** 는 AWS Lambda 와 통합하여 이러한 데이터를 모든 위치에서 사용할 수 있도록 하는 트리거를 포함하여 빠르고 유연한 NoSQL 데이터베이스 서비스를 제공합니다.
5. **비즈니스 사용자**는 DynamoDB 기반의 보고 인터페이스를 활용하여 소셜 미디어 추세 데이터에서 통찰력을 수집할 수 있습니다.

구성 참고 사항:

- Kinesis 스트림을 다시 샤딩하여 수집 속도를 높이려는 경우 모범 사례[7]를 따르십시오. 스트림 처리의 동시성은 샤드 수 및 병렬화 요인에 따라 결정됩니다. 그러므로 이러한 요인을 처리량 요구 사항에 따라 조정해야 합니다.
- 스트리밍 데이터 솔루션 백서[8]를 검토하여 배치 처리, 스트림 분석 및 기타 유용한 패턴을 확인하십시오.
- KPL 을 사용하지 않는 경우 PutRecords 같은 비원자성 작업에 대한 부분적 오류를 고려해야 합니다. Kinesis API 는 수집 당시 처리에 성공하고 실패한 모든 레코드[9]를 반환하기 때문입니다.
- 중복된 레코드[10]가 발생할 수 있으므로 소비자 및 생산자용 애플리케이션에서 재시도와 멱등성을 모두 활용해야 합니다.
- 수집된 데이터를 Amazon S3, Amazon Redshift 또는 Amazon ES 로 지속적으로 로드해야 하는 경우 Lambda 보다 Kinesis Data Firehose 를 사용하는 것이 좋습니다.
- 표준 SQL 을 사용하여 스트리밍 데이터를 쿼리하고 쿼리 결과만 Amazon S3, Amazon Redshift, Amazon ES 또는 Kinesis Streams 로 로드하는 경우 Lambda 보다 Kinesis Data Analytics 를 사용하는 것이 좋습니다.
- 배치 크기, 샤드당 동시성 및 모니터링 스트림 처리에 대한 자세한 내용을 다루는 AWS Lambda 스트림 기반 호출[11]의 모범 사례를 따르십시오.
- Lambda 의 최대 재시도 횟수, 최대 레코드 수명, 함수 오류 시 이등분 배치 및 오류 시 대상 오류 제어를 사용하여 스트림 처리 애플리케이션의 복원력을 강화합니다.

웹 애플리케이션

웹 애플리케이션은 일관적이고 안전하며 안정적인 사용자 경험을 보장해야 하므로 일반적으로 요구 사항이 까다롭습니다. 고가용성, 글로벌 가용성 및 수천 또는 잠재적으로 수백만 명의 사용자로 확장할 수 있는 확장성을 보장하기 위해 예상되는 최고 수요의 웹 요청을 처리할 상당한 양의 초과 용량을 예약해야 했습니다. 이를 위해 서버 플릿 및 추가 인프라 구성 요소를 관리해야 했기 때문에 자본 지출이 크게 증가했고 용량 프로비저닝 시간이 길어지는 결과가 발생했습니다.

AWS 에서 서버리스 컴퓨팅을 사용하면 서버 관리, 프로비저닝 용량 추측 또는 유휴 리소스 비용 지출과 관련된 차별화되지 않은 힘든 작업 없이 전체 웹 애플리케이션 스택을 배포할 수 있습니다. 또한 보안, 안정성 또는 성능을 포기하지 않아도 됩니다.

특성:

- 높은 수준의 복원력과 가용성으로 몇 분 이내에 글로벌 수준까지 확장 가능한 웹 애플리케이션을 구축해야 합니다.
- 적절한 응답 시간으로 일관된 사용자 경험을 제공해야 합니다.
- 관리형 서비스를 플랫폼에 최대한 활용하여 일반적인 플랫폼 관리와 관련된 힘든 작업을 제한해야 합니다.
- 실제 사용자 수요와 유휴 리소스에 대해 지불하는 비용을 비교하여 비용을 최적화해야 합니다.
- 설정 및 작동이 쉽고 향후에 최소한의 영향으로 확장할 수 있는 프레임워크를 생성해야 합니다.

참조 아키텍처

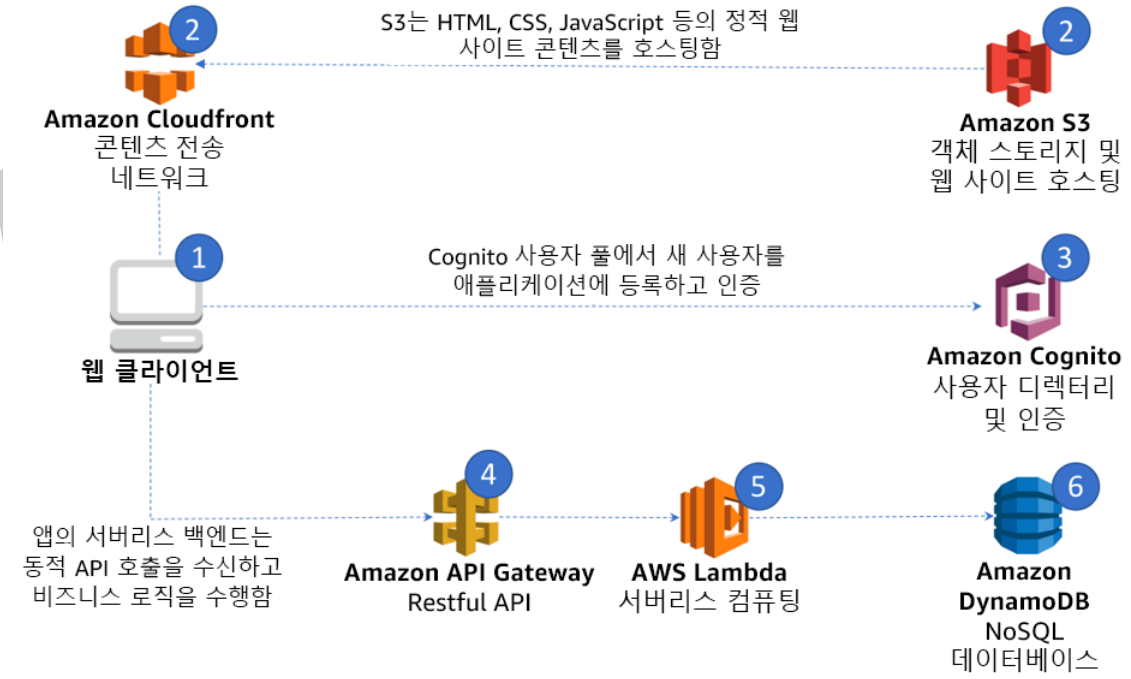

그림 6: 웹 애플리케이션의 참조 아키텍처

1. 이 웹 애플리케이션의 **소비자**는 전 세계에서 지리적으로 집중되어 있거나 분산되어 있을 수 있습니다. Amazon CloudFront 를 활용하면 캐싱 및 최적의 오리진 라우팅을 통해 이러한 소비자에게 더 나은 성능 경험을 제공하는 동시에 백엔드에 대한 중복 호출을 제한할 수 있습니다.
2. **Amazon S3** 는 웹 애플리케이션의 정적 자산을 호스팅하며 CloudFront 를 통해 안전하게 제공됩니다.
3. **Amazon Cognito 사용자 풀**은 웹 애플리케이션을 위한 사용자 관리 및 자격 증명 공급자 기능을 제공합니다.
4. 많은 시나리오에서 소비자는 Amazon S3 에서 정적 콘텐츠를 다운로드하므로 애플리케이션으로 동적 콘텐츠를 전송하거나 수신해야 합니다. 예를 들어 사용자가 양식을 통해 데이터를 제출할 때 **Amazon API Gateway** 는 이러한 호출을 수행하고 웹 애플리케이션을 통해 표시되는 응답을 반환하는 보안 엔드포인트 역할을 합니다.
5. **AWS Lambda** 함수는 DynamoDB 를 기반으로 웹 애플리케이션을 위한 CRUD(생성, 읽기, 업데이트 및 삭제) 작업을 제공합니다.
6. **Amazon DynamoDB** 는 웹 애플리케이션의 트래픽에 따라 탄력적으로 확장되는 백엔드 NoSQL 데이터 스토어를 제공할 수 있습니다.

구성 참고 사항:

- AWS 에 서버리스 웹 애플리케이션 프런트엔드를 배포하는 것과 관련된 모범 사례를 따르십시오. 자세한 내용은 운영 우수성 기반에서 확인할 수 있습니다.
- 단일 페이지 웹 애플리케이션의 경우 AWS Amplify 콘솔을 사용하여 원자성 배포, 캐시 만료, 사용자 지정 도메인 및 UI(사용자 인터페이스) 테스트를 관리할 수 있습니다.
- 인증 및 권한 부여에 대한 권장 사항은 보안 기반을 참조하십시오.
- 웹 애플리케이션 백엔드에 대한 권장 사항은 RESTful 마이크로서비스 시나리오를 참조하십시오.
- 개인화된 서비스를 제공하는 웹 애플리케이션의 경우 API Gateway 사용 계획[12]과 Amazon Cognito 사용자 풀을 활용하여 서로 다른 사용자 세트의 액세스 범위를 지정할 수 있습니다. 예를 들어 프리미엄 사용자에게는 더 높은 API 호출 처리량, 추가 API 액세스 및 추가 스토리지 등이 제공될 수 있습니다.
- 이 시나리오에서 다루지 않는 검색 기능이 애플리케이션에 사용되는 경우 모바일 백엔드 시나리오를 참조하십시오.

Well-Architected 프레임워크의 기반

이 섹션에는 각 기반에 대한 설명과 서버리스 애플리케이션용 솔루션의 설계와 관련된 정의, 모범 사례, 질문, 고려 사항 및 주요 AWS 서비스가 포함되어 있습니다.

간결성을 위해 서버리스 워크로드와 관련된 Well-Architected 프레임워크의 질문만 포함했습니다. 그러나 아키텍처를 설계할 때는 이 문서에 포함되지 않은 질문도 고려해야 합니다. AWS Well-Architected 프레임워크 백서를 읽어보시기 바랍니다.

운영 우수성 기반

운영 우수성 기반에는 비즈니스 가치를 실현하고 지원 프로세스 및 절차를 지속적으로 개선하기 위해 시스템을 실행하고 모니터링하는 기능이 포함됩니다.

정의

클라우드의 운영 우수성에는 3 가지 모범 사례 영역이 있습니다.

- 준비
- 운영
- 개선

Well-Architected 프레임워크에서 프로세스, 런북 및 게임데이와 관련하여 다루는 내용에 더해 서버리스 애플리케이션의 운영 우수성을 촉진하기 위한 특정 영역이 있습니다.

모범 사례

준비

이 하위 섹션에는 서버리스 애플리케이션에 고유한 운영 사례가 없습니다.

운영

> OPS 1: 서버리스 애플리케이션의 상태를 파악할 때 어떤 방법을 사용합니까?

지표 및 알림

사용하려는 모든 AWS 서비스의 동작을 평가하고 필요한 경우 사용자 지정 지표를 추가하는 계획을 마련하려면 이러한 서비스에 대한 Amazon CloudWatch 지표 및 차원을 이해하는 것이 중요합니다.

Amazon CloudWatch 에서는 자동화된 교차 서비스 및 서비스별 대시보드를 통해 사용하는 AWS 서비스의 주요 지표를 쉽게 확인할 수 있습니다. 사용자 지정 지표의 경우 Amazon CloudWatch 의 임베디드 지표 형식을 사용하여 서버리스 애플리케이션의 성능에 영향을 주지 않고 CloudWatch 에서 비동기식으로 처리할 다수의 지표를 기록할 수 있습니다.

지표와 관련하여 대시보드를 생성하거나 신규 및 기존 애플리케이션에 대한 계획을 구성하려는 경우 다음 지침을 사용하십시오.

- **비즈니스 지표**
    - 비즈니스 목표를 기준으로 애플리케이션 성능을 측정하는 비즈니스 KPI 를 나타내며 전체 비즈니스 또는 매출과 관련하여 심각한 영향을 미치는 항목을 파악하는 데 중요합니다.
    - **예**: 제출된 주문, 직불/신용 카드 작업, 구매한 항공권 등
- **고객 경험 지표**
    - 고객 경험 데이터는 UI/UX 의 전반적인 효과를 나타낼 뿐 아니라 애플리케이션의 특정 섹션에서 고객 경험에 영향을 미치는 변경 사항 또는 비정상 동작을 보여줍니다. 이러한 지표는 시간대별 영향 및 고객 기반 전체의 분포 상태를 파악할 때 이상값을 방지하기 위해 백분위수로 측정됩니다.
    - **예**: 감지된 지연 시간, 장바구니에 항목을 추가하거나 체크아웃하는 데 걸리는 시간, 페이지 로드 시간 등
- **시스템 지표**
    - 공급업체 및 애플리케이션 지표는 이전 섹션의 근본 원인을 보강하는 데 중요합니다. 또한 시스템이 정상인지, 위험한 상태인지, 이미 고객이 되었는지 여부를 알려줍니다.
    - **예**: HTTP 오류/성공 비율, 메모리 사용률, 함수 기간/오류/조절, 대기열 길이, 스트림 레코드 길이, 통합 지연 시간 등

- **운영 지표**
    - 운영 지표는 지정된 시스템의 지속 가능성 및 유지 관리를 이해하고 시간대별 안정성의 개선/저하를 파악하는 데 중요합니다.
    - **예**: 티켓 수(해결 성공 및 실패 등), 호출된 직원을 호출한 횟수, 가용성, CI/CD 파이프라인 통계(배포 성공/실패, 피드백 시간, 주기 및 리드 시간 등)

CloudWatch 경보는 개별 수준과 집계 수준 모두에서 구성되어야 합니다. 개별 수준의 예로는 Lambda 의 *Duration* 지표 또는 API Gateway 의 *IntegrationLatency*(API 를 통해 호출된 경우)에 대한 경보가 있는데 애플리케이션의 부분별로 프로필이 다를 수 있기 때문입니다. 이 경우 함수의 실행 시간을 평소보다 길게 하는 잘못된 배포를 빠르게 식별할 수 있습니다.

집계 수준의 예로는 다음 지표에 대한 경보가 포함됩니다.

- **AWS Lambda**: *Duration*, *Errors*, *Throttling* 및 *ConcurrentExecutions*. 스트림 기반 호출의 경우 *IteratorAge* 에 대한 알림을 제공합니다. 비동기식 호출의 경우 *DeadLetterErrors* 에 대한 알림을 제공합니다.
- **Amazon API Gateway**: *IntegrationLatency*, *Latency*, *5XXError*
- **Application Load Balancer**: *HTTPCode_ELB_5XX_Count*, *RejectedConnectionCount*, *HTTPCode_Target_5XX_Count*, *UnHealthyHostCount*, *LambdaInternalError*, *LambdaUserError*
- **AWS AppSync**: 5XX 및 *Latency*
- **Amazon SQS:** *ApproximateAgeOfOldestMessage*
- **Amazon Kinesis Data Streams:** *ReadProvisionedThroughputExceeded*, *WriteProvisionedThroughputExceeded*, *GetRecords.IteratorAgeMilliseconds*, *PutRecord.Success*, *PutRecords.Success*(Kinesis Producer Library 를 사용하는 경우) 및 *GetRecords.Success*
- **Amazon SNS**: *NumberOfNotificationsFailed*, *NumberOfNotificationsFilteredOut-InvalidAttributes*
- **Amazon SES:** *Rejects*, *Bounces*, *Complaints*, *Rendering Failures*
- **AWS Step Functions:** *ExecutionThrottled*, *ExecutionsFailed*, *ExecutionsTimedOut*
- **Amazon EventBridge:** *FailedInvocations*, *ThrottledRules*
- **Amazon S3:** *5xxErrors*, *TotalRequestLatency*
- **Amazon DynamoDB:** *ReadThrottleEvents*, *WriteThrottleEvents*, *SystemErrors*, *ThrottledRequests*, *UserErrors*

중앙 집중식 로깅 및 구조화된 로깅

트랜잭션, 상관 관계 식별자, 구성 요소 전체의 요청 식별자 및 비즈니스 결과에 대한 운영 정보를 제공하도록 애플리케이션 로깅을 표준화합니다. 이 정보를 사용하여 워크로드 상태에 대한 임의 질문에 답합니다.

다음은 JSON 을 출력으로 사용하는 구조화된 로깅의 예입니다.

```
{
   "timestamp":"2019-11-26 18:17:33,774",
   "level":"INFO",
   "location":"cancel.cancel_booking:45",
   "service":"booking",
   "lambda_function_name":"test",
   "lambda_function_memory_size":"128",
   "lambda_function_arn":"arn:aws:lambda:eu-west-1:
12345678910:function:test",
   "lambda_request_id":"52fdfc07-2182-154f-163f-5f0f9a621d72",
   "cold_start": "true",
   "message":{
      "operation":"update_item",
      "details:":{
         "Attributes":{
            "status":"CANCELLED"
         },
         "ResponseMetadata":{

"RequestId":"G7S3SCFDEMEINPG6AOC6CL5IDNVV4KQNSO5AEMVJF66Q9ASUAAJG",
            "HTTPStatusCode":200,
            "HTTPHeaders":{
               "server":"Server",
               "date":"Thu, 26 Nov 2019 18:17:33 GMT",
               "content-type":"application/x-amz-json-1.0",
               "content-length":"43",
               "connection":"keep-alive",
               "x-amzn-
requestid":"G7S3SCFDEMEINPG6AOC6CL5IDNVV4KQNSO5AEMVJF66Q9ASUAAJG",
               "x-amz-crc32":"1848747586"
            },
            "RetryAttempts":0
         }
      }
   }
}
```

중앙 집중식 로깅은 서버리스 애플리케이션 로그를 검색하고 분석하는 데 도움이 됩니다. 구조화된 로깅을 사용하면 애플리케이션 상태에 대한 임의 질문에 답하는 쿼리를 더 쉽게 도출할 수 있습니다. 시스템이 커지고 더 많은 로깅이 수집되면 적절한 로깅 수준과 샘플링 메커니즘을 사용하여 DEBUG 모드에서 소량의 로그를 기록하는 것을 고려하십시오.

분산 추적

비 서버리스 애플리케이션과 마찬가지로 분산 시스템에서는 비정상 동작이 대규모로 발생할 수 있습니다. 서버리스 아키텍처의 경우 아키텍처 특성상 분산 추적을 사용하는 것이 기본입니다.

서버리스 애플리케이션을 변경할 때는 기존 워크로드에 사용되는 배포, 변경 및 릴리스 관리 원칙과 동일한 다수의 원칙이 수반됩니다. 그러나 기존 도구를 사용하여 이러한 원칙을 달성하는 방법에는 미묘한 차이가 있습니다.

AWS X-Ray 를 사용한 활성 추적을 활성화하여 분산 추적 기능을 제공하고 시각적 서비스 맵을 활성화하여 문제 해결 시간을 단축해야 합니다. X-Ray 는 성능 저하를 식별하고 지연 시간 분포를 비롯한 비정상 동작을 신속하게 이해하는 데 도움이 됩니다.

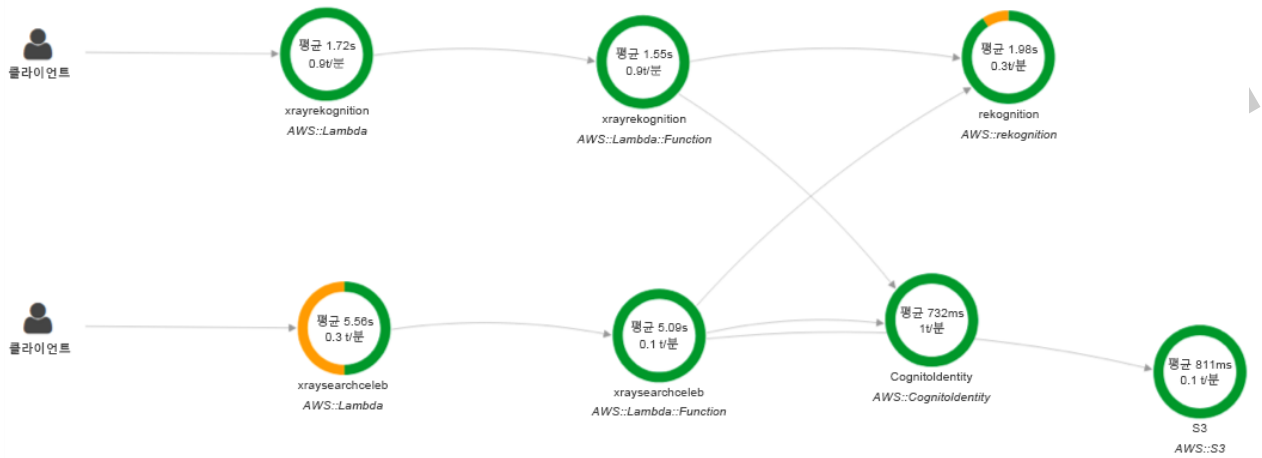

그림 7: 2 개의 서비스를 시각화하는 AWS X-Ray 서비스 맵

서비스 맵은 주의가 필요하고 복원력 사례를 적용해야 하는 통합 지점을 파악하는 데 유용합니다. 통합 호출의 경우 결함이 다운스트림 서비스로 전파되는 것을 방지하는 재시도, 백오프 및 회로 차단기가 필요합니다.

또 다른 예는 네트워킹 이상입니다. 기본 제한 시간 및 재시도 설정에 의존해서는 안 됩니다. 대신, 기본값이 분 단위가 아닌 초 단위인 특정 클라이언트에서 소켓 읽기/쓰기 제한 시간이 발생할 경우 페일 페스트로 전환하십시오.

또한 X-Ray 는 애플리케이션 내의 비정상 동작을 보다 효율적으로 식별할 수 있는 두 가지 강력한 기능인 주석과 하위 세그먼트를 제공합니다.

하위 세그먼트는 애플리케이션 로직의 구성과 통신 대상인 외부 종속성을 파악하는 데 유용합니다. 주석은 AWS X-Ray 에 의해 자동으로 인덱싱되는 문자열, 숫자 또는 부울 값으로 구성된 키 – 값 페어입니다.

이 둘을 함께 사용하면 특정 작업 및 비즈니스 트랜잭션에 대한 성능 통계를 빠르게 식별할 수 있습니다. 예를 들어 데이터베이스 쿼리에 소요되는 시간 또는 많은 사람이 포함된 사진을 처리하는 데 소요되는 시간 등을 파악할 수 있습니다.

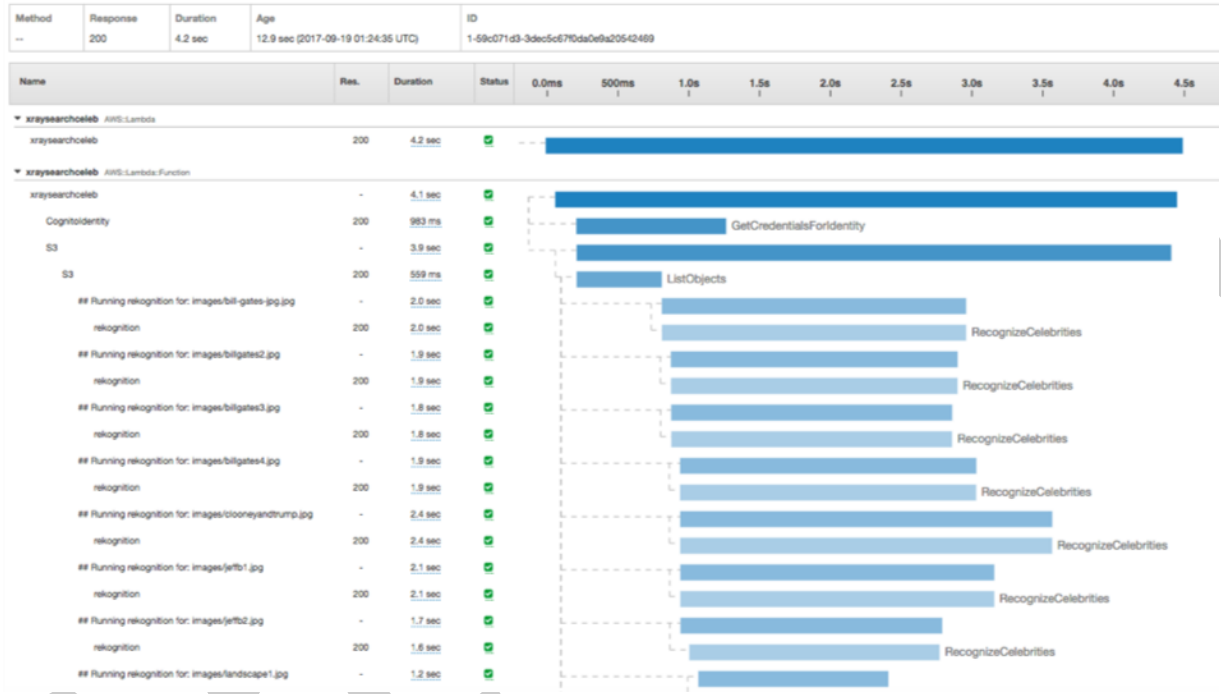

그림 8: ##으로 시작되는 하위 세그먼트가 있는 AWS X-Ray 추적

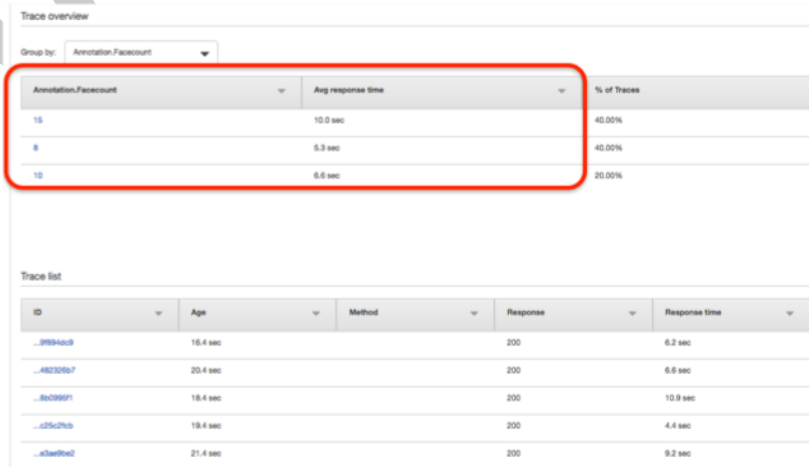

그림 9: 사용자 지정 주석으로 그룹화된 AWS X-Ray 추적

OPS 2: 애플리케이션 수명 주기 관리에는 어떻게 접근합니까?

프로토타입 제작

코드형 인프라를 사용하여 프로토타입을 제작할 새 기능을 위한 임시 환경을 생성하고 작업이 완료되면 폐기합니다. 팀의 규모와 조직 내 자동화 수준에 따라 팀당 또는 개발자별 전용 계정을 사용할 수 있습니다.

임시 환경에서는 관리형 서비스로 작업할 때 충실도를 높일 수 있으며 제어 수준을 강화하여 워크로드가 의도한 대로 통합되고 작동하는지 확인할 수 있습니다.

구성 관리의 경우 로깅 수준 및 데이터베이스 연결 문자열 등 자주 변경되는 항목에 환경 변수를 사용합니다. AWS System Manager Parameter Store 를 동적 구성(예: 기능 전환)에 사용하고 AWS Secrets Manager 를 사용하여 중요한 데이터를 저장합니다.

테스트

테스트는 일반적으로 단위, 통합 및 수락 테스트를 통해 수행됩니다. 강력한 테스트 전략을 개발하면 다양한 로드 및 조건에서 서버리스 애플리케이션을 에뮬레이션할 수 있습니다.

단위 테스트는 비 서버리스 애플리케이션과 다르지 않아야 하므로 변경 없이 로컬로 실행할 수 있습니다.

통합 테스트에서는 사용자가 제어할 수 없는 모의 서비스를 테스트할 수 없습니다. 변경이 발생할 경우 예기치 않은 결과가 나올 수 있기 때문입니다. 이러한 테스트는 실제 서비스를 사용할 때 더 효과적입니다. 프로덕션에서 서버리스 애플리케이션이 요청을 처리할 때 사용하는 것과 동일한 환경을 제공하기 때문입니다.

수락 테스트 또는 엔드투엔드 테스트의 기본 목적은 사용 가능한 외부 인터페이스를 통한 최종 사용자의 작업을 시뮬레이션하는 것입니다. 따라서 이러한 테스트는 변경 없이 수행되어야 합니다. 따라서 여기에서 숙지해야 할 고유한 권장 사항은 없습니다.

일반적으로, 성능 테스트 컨텍스트에서는 AWS Marketplace 에서 제공되는 Lambda 및 타사 도구를 테스트 장치로 사용할 수 있습니다. 다음은 성능 테스트 중 고려해야 할 몇 가지 사항입니다.

- 호출된 최대 메모리 사용량 및 초기화 기간과 같은 지표는 CloudWatch Logs 에서 확인할 수 있습니다. 자세한 내용은 성능 기반 섹션을 참조하십시오.
- Lambda 함수가 Amazon Virtual Private Cloud(VPC) 내에서 실행되는 경우 서브넷 내에서 사용할 수 있는 IP 주소 공간에 주의하십시오.
- 모듈화된 코드를 핸들러 외부에서 별도의 함수로 생성하면 더 많은 함수를 단위 테스트에 사용할 수 있습니다.
- Lambda 함수의 정적 생성자/초기화 코드(즉, 핸들러 외부의 글로벌 범위)에서 참조되는 외부화된 연결 코드(예: 관계형 데이터베이스에 대한 연결 풀)를 설정하면 Lambda 실행 환경을 재사용할 때 외부 연결 임계값에 도달하지 않습니다.
- 성능 테스트가 계정의 현재 제한을 초과하지 않는다면 DynamoDB 온디맨드 테이블을 사용하십시오.
- 성능 테스트 시 서버리스 애플리케이션에서 사용할 수 있는 기타 서비스 제한을 고려하십시오.

배포

코드형 인프라 및 버전 관리를 사용하여 변경 사항 및 릴리스를 추적할 수 있습니다. 개발 단계와 프로덕션 단계를 개별 환경으로 격리합니다. 이렇게 하면 수동 프로세스로 인한 오류가 감소하고 제어 수준이 강화되므로 워크로드가 의도한 대로 작동하는지 확인할 수 있습니다.

서버리스 프레임워크를 사용하여 AWS SAM 또는 Serverless Framework 와 같은 서버리스 애플리케이션의 모델링, 프로토타입 제작, 빌드, 패키징 및 배포를 수행합니다. 코드형 인프라 및 프레임워크를 사용하면 서버리스 애플리케이션과 해당 종속성을 파라미터화하여 격리된 단계와 AWS 계정 전체에 쉽게 배포할 수 있습니다.

예를 들어 CI/CD 파이프라인 베타 단계에서는 *OrderAPIBeta, OrderServiceBeta, OrderStateMachineBeta, OrderBucketBeta, OrderTableBeta, OrderTableBeta 등의 리소스를 베타* AWS 계정에 생성하고 다른 계정에 있을 수 있는 해당하는 단계(감마, 개발, 프로덕션)에도 생성할 수 있습니다.

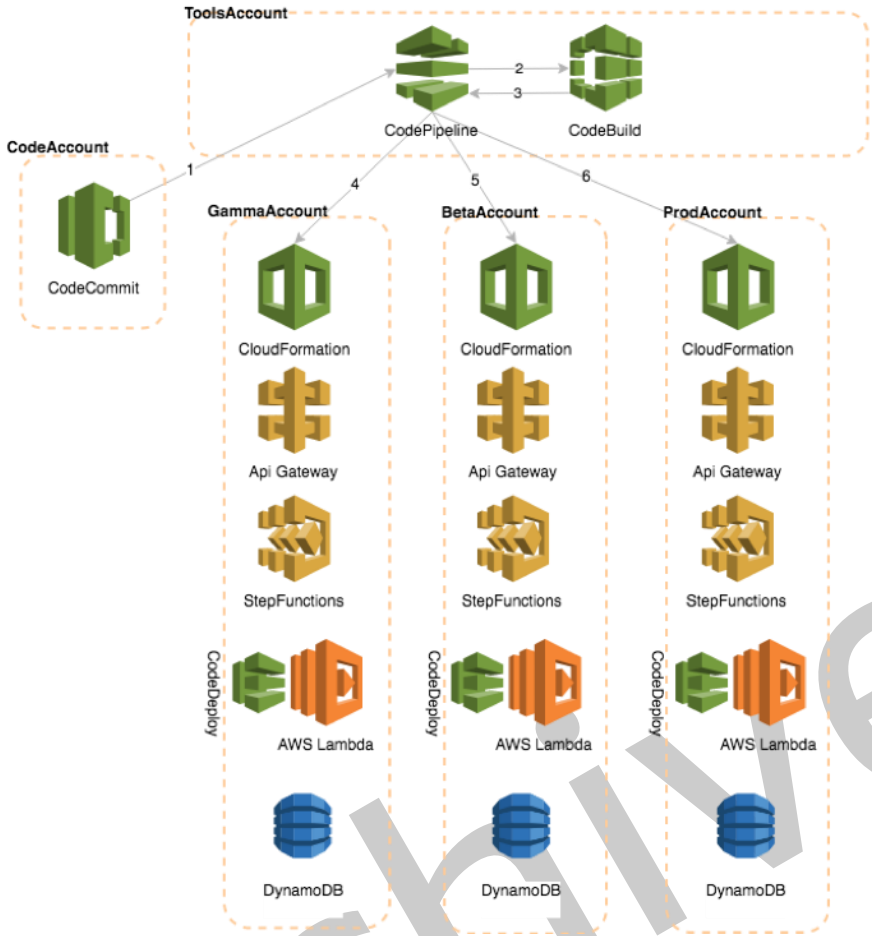

그림 10: 여러 계정의 CI/CD 파이프라인

프로덕션 환경에 배포할 때는 Canary 또는 선형 배포의 경우 새로운 변경 사항이 최종 사용자에게 점진적으로 이동하므로 한 번에 모두 배포하는 것보다 안전하게 배포하는 것이 좋습니다. CodeDeploy 후크(*BeforeAllowTraffic*, *AfterAllowTraffic*) 및 경보를 사용하여 배포 검증, 롤백 및 애플리케이션에 필요할 수 있는 사용자 지정에 대한 제어를 강화합니다.

또한 롤아웃 배포의 일부로 가상 트래픽, 사용자 지정 지표 및 알림을 결합할 수 있습니다. 이렇게 하면 고객 경험에 영향을 미칠 수 있는 새로운 변경 사항으로 인한 오류를 사전에 탐지할 수 있습니다.

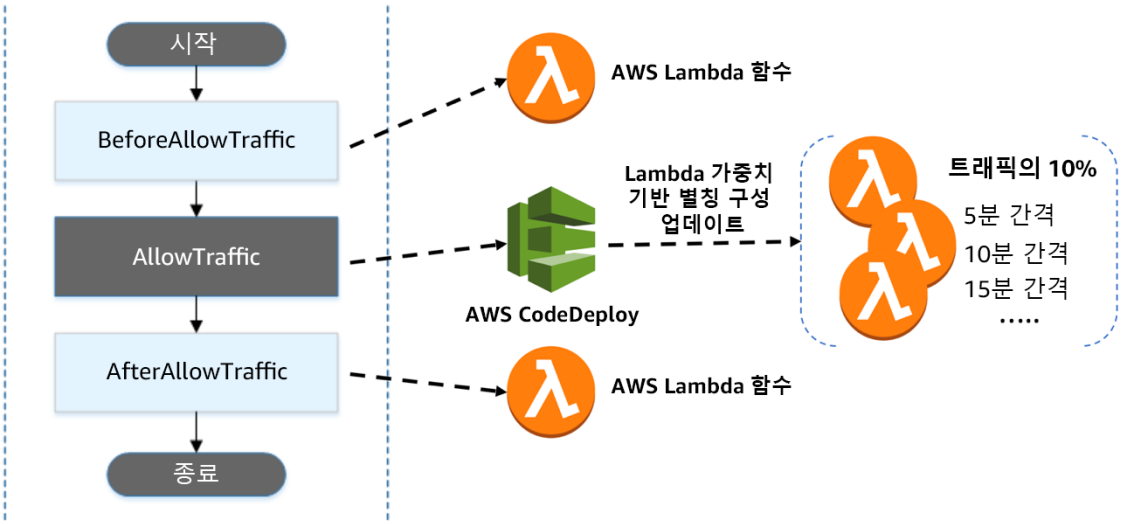

그림 11: AWS CodeDeploy Lambda 배포 및 후크

개선

이 하위 섹션에는 서버리스 애플리케이션에 고유한 운영 사례가 없습니다.

주요 AWS 서비스

운영 우수성을 위한 주요 AWS 서비스에는 AWS Systems Manager Parameter Store, AWS SAM, CloudWatch, AWS CodePipeline, AWS X-Ray, Lambda 및 API Gateway 가 포함됩니다.

리소스

운영 우수성 모범 사례에 대한 자세한 내용은 다음 리소스를 참조하십시오.

설명서 및 블로그

- API Gateway 단계 변수[13]
- Lambda 환경 변수[14]
- AWS SAM CLI[15]
- X-Ray 지연 시간 분포[16]
- X-Ray 를 사용하여 Lambda 기반 애플리케이션 문제 해결[17]
- System Manager(SSM) Parameter Store[18]

- 서버리스 애플리케이션을 위한 지속적 배포 블로그 게시물[19]
- SamFarm: CI/CD 예제[20]
- CI/CD 를 사용하는 서버리스 애플리케이션 예제
- 알림 및 대시보드 자동화를 위한 서버리스 애플리케이션 예제
- Python 용 CloudWatch 임베디드 지표 형식 라이브러리
- Node.js 용 CloudWatch 임베디드 지표 형식 라이브러리
- 추적, 구조화된 로깅 및 사용자 지정 지표를 구현하기 위한 라이브러리 예제
- 일반적인 AWS 제한
- Stackery: 다중 계정 모범 사례

백서

- AWS 의 지속적 통합/지속적 전달 사례[21]

타사 도구

- 타사 프레임워크/도구가 포함된 서버리스 개발자 도구 페이지[22]
- Stelligent: 운영 지표에 대한 CodePipeline 대시보드

보안 기반

보안 기반에는 정보, 시스템, 자산을 보호하는 동시에 위험 진단과 문제 해결 전략을 통해 비즈니스 가치를 제공하는 기능이 포함됩니다.

정의

클라우드의 보안에는 5 가지 모범 사례 영역이 있습니다.

- 자격 증명 및 액세스 관리
- 탐지 제어
- 인프라 보호

- 데이터 보호
- 인시던트 대응

서버리스에서는 운영 체제 패치 적용, 바이너리 업데이트 등과 같은 인프라 관리 작업이 필요하지 않으므로 오늘날 가장 큰 보안 문제의 일부가 해결됩니다. 비 서버리스 아키텍처에 비해 공격 표면은 작지만 OWASP(Open Web Application Security Project) 및 애플리케이션 보안 모범 사례는 서버리스에도 마찬가지로 적용됩니다.

이 섹션에는 공격자가 액세스 권한을 획득하거나 시스템 남용을 위해 잘못 구성된 권한을 악용할 때 사용하는 특정 방법을 해결하기 위한 질문이 포함되어 있습니다. 이 섹션에 설명된 사례는 전체 클라우드 플랫폼의 보안에 많은 영향을 미치므로 신중히 확인하고 자주 검토해야 합니다.

인시던트 대응 범주의 경우 AWS Well-Architected 프레임워크의 사례가 여전히 적용되므로 이 문서에서 설명하지 않습니다.

모범 사례

자격 증명 및 액세스 관리

SEC 1: 서버리스 API 에 대한 액세스를 어떻게 제어합니까?

API 는 API 로 수행할 수 있는 작업과 얻을 수 있는 중요한 데이터 때문에 종종 공격의 대상이 됩니다. 이러한 공격으로부터 API 를 방어할 수 있는 다양한 보안 모범 사례가 있습니다.

인증/권한 부여 측면에서 현재 API Gateway 내에서 API 호출을 승인할 때는 4 가지 메커니즘이 사용됩니다.

- AWS_IAM 권한 부여
- Amazon Cognito 사용자 풀
- API Gateway Lambda 권한 부여자
- 리소스 정책

기본적으로 이러한 메커니즘이 구현되는지 여부와 구현 방법을 이해해야 합니다. 현재 AWS 환경 내에 있거나 AWS Identity and Access Management(IAM) 임시 자격 증명을 검색하여 환경에

액세스할 수 있는 수단을 보유한 소비자의 경우 AWS_IAM 권한 부여를 사용하고 해당 IAM 역할에 최소 권한 권한을 추가하여 API 를 안전하게 호출할 수 있습니다.

다음 다이어그램은 이 컨텍스트에서 AWS_IAM 권한 부여를 사용하는 방법을 보여줍니다.

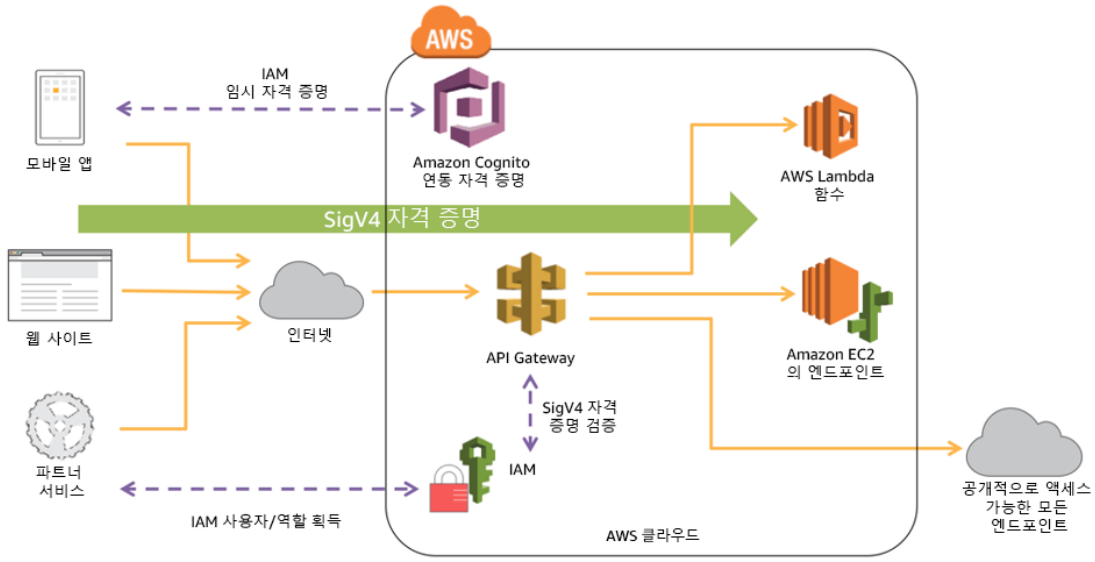

그림 12: AWS_IAM 권한 부여

기존 IdP(자격 증명 공급자)가 있는 경우 API Gateway Lambda 권한 부여자를 사용하여 Lambda 함수를 호출하고 지정된 사용자를 IdP 에 대해 인증/검증할 수 있습니다. 자격 증명 메타데이터를 기반으로 하는 사용자 지정 검증 로직에 Lambda 권한 부여자를 사용할 수 있습니다.

Lambda 권한 부여자는 전달자 토큰에서 파생된 추가 정보 또는 요청 컨텍스트 값을 백엔드 서비스로 전송할 수 있습니다. 예를 들어 권한 부여자는 사용자 ID, 사용자 이름 및 범위가 포함된 맵을 반환할 수 있습니다. Lambda 권한 부여자를 사용하면 백엔드에서 권한 부여 토큰을 사용자 중심 데이터에 매핑할 필요가 없으므로 이러한 정보의 노출을 권한 부여 함수로 제한할 수 있습니다.

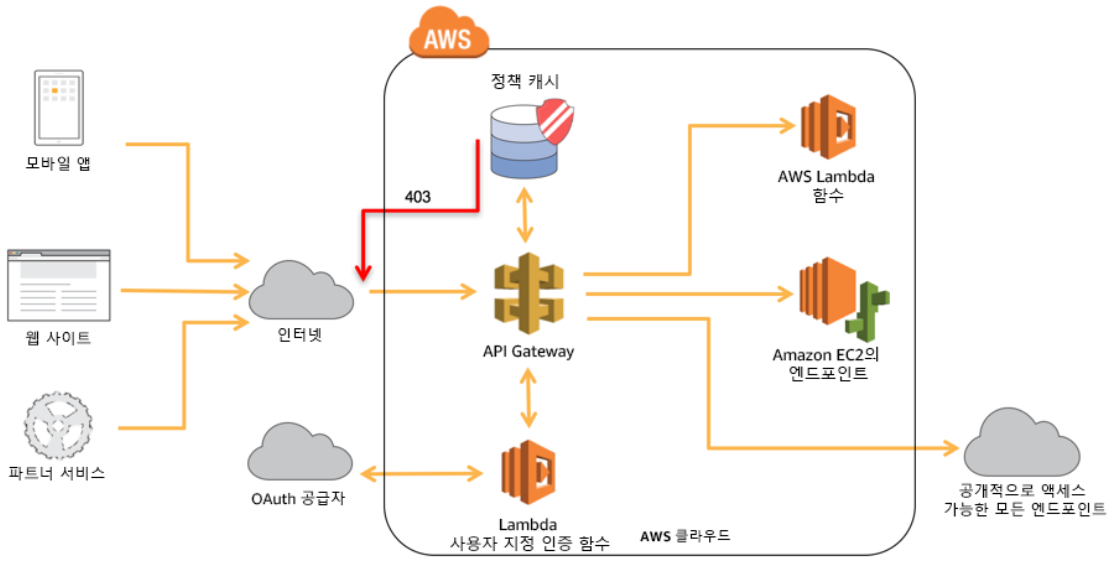

그림 13: API Gateway Lambda 권한 부여자

IdP 가 없는 경우 Amazon Cognito 사용자 풀을 활용하여 기본 제공되는 사용자 관리 기능을 제공하거나 Facebook, Twitter, Google+및 Amazon 과 같은 외부 자격 증명 공급자와 통합할 수 있습니다.

이는 사용자가 이메일 주소/사용자 이름으로 등록/로그인할 수 있는 동시에 소셜 미디어 플랫폼의 기존 계정을 사용하여 인증하는 모바일 백엔드 시나리오에서 흔히 볼 수 있습니다. 이 접근 방식은 OAuth 범위를 통한 세분화된 권한 부여도 제공합니다.

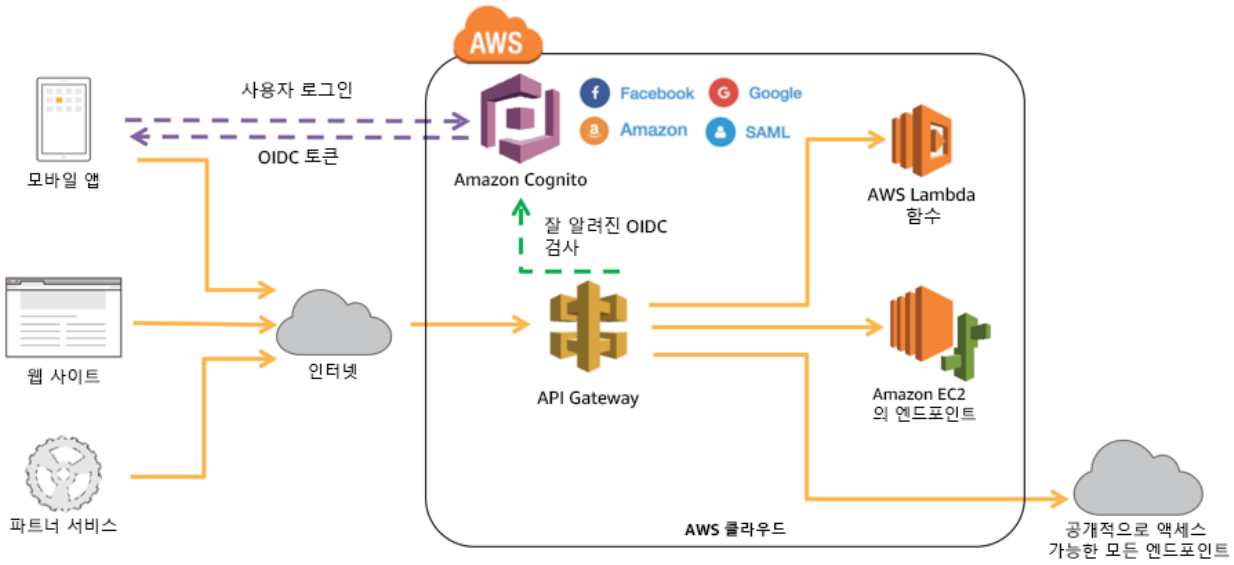

그림 14: Amazon Cognito 사용자 풀

API Gateway API 키는 보안 메커니즘이 아니며 퍼블릭 API 가 아닌 한 권한 부여에 사용해서는 안 됩니다. API 전체의 소비자 사용량을 추적하는 데 주로 사용되어야 하며, 이 섹션의 앞에서 언급한 권한 부여자에 더해 사용될 수 있습니다.

Lambda 권한 부여자를 사용할 때는 쿼리 문자열 파라미터 또는 헤더를 통해 자격 증명 또는 민감한 데이터를 전달하지 않는 것이 좋습니다. 그렇지 않으면 시스템 침해가 발생할 수 있습니다.

Amazon API Gateway 리소스 정책은 API 에 연결하여 지정된 AWS 보안 주체의 API 호출 가능성 여부를 제어할 수 있는 JSON 정책 문서입니다.

이 메커니즘을 사용하면 다음을 기준으로 API 호출을 제한할 수 있습니다.

- 지정된 AWS 계정의 사용자 또는 모든 AWS IAM 자격 증명
- 지정된 소스 IP 주소 범위 또는 CIDR 블록
- 지정된 Virtual Private Cloud(VPC) 또는 VPC 종단점(모든 계정)

리소스 정책을 사용하면 일반적인 시나리오를 제한할 수 있습니다. 예를 들어 특정 IP 범위를 사용하는 알려진 클라이언트 또는 다른 AWS 계정의 요청만 허용할 수 있습니다. 프라이빗 IP 주소의 요청을 제한하려면 API Gateway 프라이빗 엔드포인트를 대신 사용하는 것이 좋습니다.

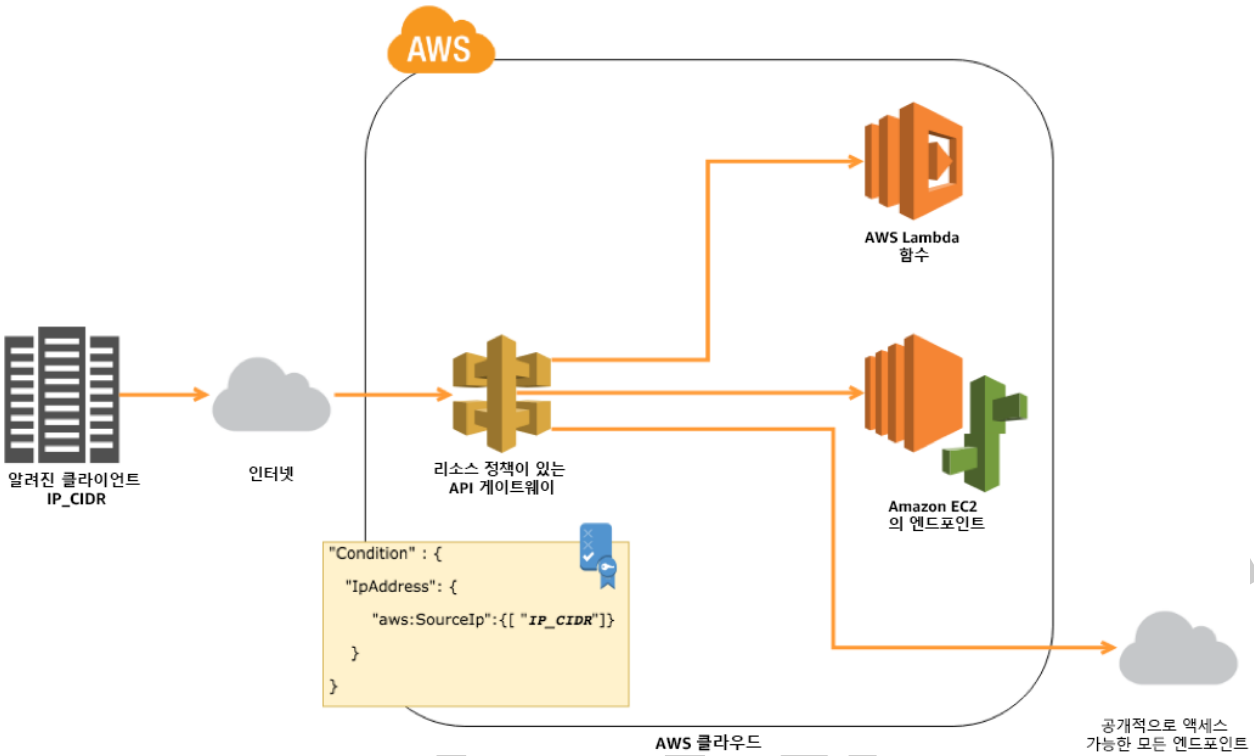

그림 15: IP CIDR 을 기반으로 하는 Amazon API Gateway 리소스 정책

프라이빗 엔드포인트를 사용하면 API Gateway 가 VPC 내 서비스 및 리소스 또는 Direct Connect 를 통해 자체 데이터 센터에 연결된 서비스 및 리소스에 대한 액세스를 제한합니다.

프라이빗 엔드포인트와 리소스 정책을 함께 사용하면 API 를 특정 프라이빗 IP 범위 내의 특정 리소스 호출로 제한할 수 있습니다. 이 조합은 같은 계정 또는 다른 계정에 있을 수 있는 내부 마이크로서비스에 주로 사용됩니다.

대규모 배포와 여러 AWS 계정을 사용하는 조직은 API Gateway 에서 교차 계정 Lambda 권한 부여자를 활용하여 유지 관리를 줄이고 보안 사례를 중앙 집중화할 수 있습니다. 예를 들어 API Gateway 에는 개별 계정에서 Amazon Cognito 사용자 풀을 사용할 수 있는 기능이 있습니다. 또한 Lambda 권한 부여자를 개별 계정에서 생성하고 관리한 다음 API Gateway 로 관리되는 여러 API 에서 재사용할 수 있습니다. 두 시나리오는 API 전체의 권한 부여 사례를 표준화해야 하는 다수의 마이크로서비스가 있는 배포에서 일반적인 시나리오입니다.

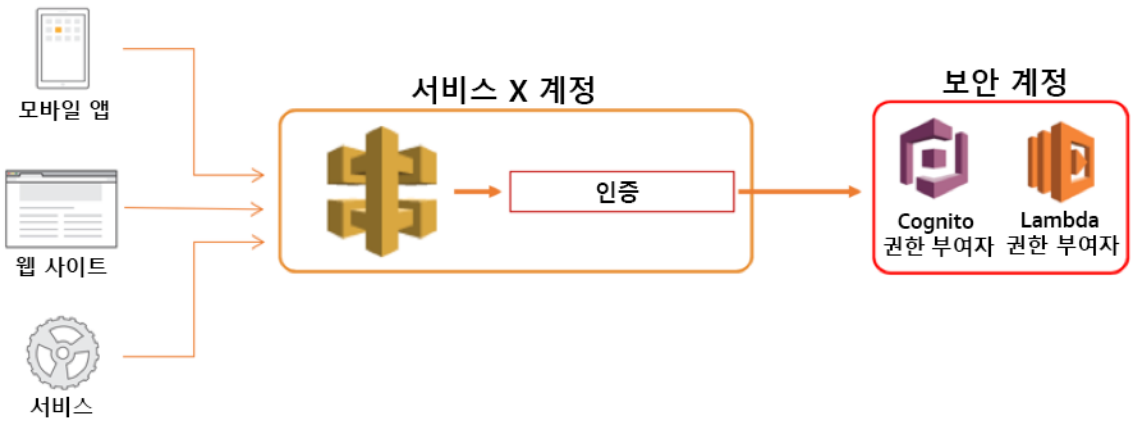

그림 16: API Gateway 교차 계정 권한 부여자

SEC 2: 서버리스 애플리케이션의 보안 경계를 어떻게 관리하고 있습니까?

Lambda 함수를 사용하는 경우 최소 권한 액세스를 따르고 지정된 작업을 수행하는 데 필요한 액세스만 허용하는 것이 좋습니다. 필요한 것보다 많은 권한이 있는 역할을 연결하면 시스템이 침해를 받을 수 있습니다.

보안 컨텍스트에서는 범위가 지정된 활동을 수행하는 더 작은 함수를 사용하는 것이 서버리스 애플리케이션의 올바른 설계에 도움이 됩니다. IAM 역할과 관련하여, 둘 이상의 Lambda 함수 내에서 IAM 역할을 공유하면 최소 권한 액세스가 위반될 수 있습니다.

탐지 제어

올바른 아키텍처를 갖춘 설계에서 로그 관리는 보안/과학 수사부터 규제 또는 법적 요구 사항에 이르는 다양한 이유로 중요합니다.

애플리케이션 종속성의 취약성을 추적하는 것도 중요한데 공격자는 사용되는 프로그래밍 언어에 관계없이 종속성에 포함된 알려진 취약성을 악용할 수 있기 때문입니다.

애플리케이션 종속성 취약성 검사의 경우 OWASP 종속성 검사를 비롯한 여러 상용 및 오픈소스 솔루션을 CI/CD 파이프라인에 통합하여 사용할 수 있습니다. AWS SDK 를 포함한 모든 종속성을 버전 관리 소프트웨어 리포지토리의 일부로 포함하는 것이 중요합니다.

인프라 보호

서버리스 애플리케이션에서 Virtual Private Cloud(VPC)에 배포된 다른 구성 요소 또는 온프레미스에 상주하는 애플리케이션과 상호 작용해야 하는 시나리오에서는 네트워킹 경계를 고려하는 것이 중요합니다.

Lambda 함수를 구성하여 VPC 내의 리소스에 액세스할 수 있습니다. AWS Well-Architected 프레임워크에 설명된 대로 모든 계층에서 트래픽을 제어하십시오. 규정 준수 이유로 아웃바운드 트래픽 필터링이 필요한 워크로드의 경우 비 서버리스 아키텍처에 적용되는 것과 동일한 방식으로 프록시를 사용할 수 있습니다.

네트워크 경계를 애플리케이션 코드 수준에서만 적용하고, 액세스할 수 있는 리소스에 대한 지침을 제공하는 것은 관심의 분리 측면에서 권장되지 않습니다.

서비스 간 통신의 경우 정적 키보다 AWS IAM 을 사용한 임시 자격 증명과 같은 동적 인증을 사용하는 것이 좋습니다. API Gateway 와 AWS AppSync 는 모두 AWS 서비스와의 통신을 보호하는 데 이상적인 IAM 권한 부여를 지원합니다.

데이터 보호

서버리스 애플리케이션 설계에 따라 중요한 데이터가 로그에 포함될 수 있으므로 API Gateway 액세스 로그를 활성화하고 필요한 것만 선택적으로 선택하는 것을 고려하십시오. 또한 이러한 이유로, 서버리스 애플리케이션을 통과하는 중요한 데이터를 암호화하는 것이 좋습니다.

API Gateway 와 AWS AppSync 는 모든 통신, 클라이언트 및 통합에 TLS 를 사용합니다. HTTP 페이로드는 전송 중에 암호화되지만 URL 의 일부인 요청 경로 및 쿼리 문자열은 암호화되지 않을 수 있습니다. 따라서 중요한 데이터가 표준 출력으로 전송되는 경우 CloudWatch Logs 를 통해 실수로 노출될 수 있습니다.

또한 입력 형식 오류 또는 입력 가로채기가 발생할 경우 시스템 액세스 권한을 확보하기 위한 공격 벡터로 사용되거나 오작동을 야기할 수 있습니다. AWS Well-Architected 프레임워크에 자세히 설명된 대로 중요한 데이터는 가능한 모든 계층에서 항상 보호되어야 합니다. 해당 백서의 권장 사항은 여기에도 적용됩니다.

API Gateway 와 관련하여, 중요한 데이터는 HTTP 요청의 일부로 전송하기 전에 클라이언트 측에서 암호화되거나 HTTP POST 요청의 일부인 페이로드로 전송되어야 합니다. 여기에는 지정된 요청을 제출하기 전에, 중요한 데이터가 포함될 수 있는 헤더를 암호화하는 것도 포함됩니다.

Lambda 함수 또는 API Gateway 에 구성될 수 있는 통합의 경우 처리 또는 데이터 조작 전에 중요한 데이터를 암호화해야 합니다. 이렇게 하면 이러한 데이터가 영구 스토리지에서 노출되거나 CloudWatch Logs 에 의해 스트리밍 및 유지되는 표준 출력에 의해 노출되는 경우 데이터 유출을 방지할 수 있습니다.

이 문서의 앞에서 설명한 시나리오에서 Lambda 함수는 저장 중 암호화와 함께 DynamoDB, Amazon ES 또는 Amazon S3 에 암호화된 데이터를 유지합니다. 암호화되지 않은 중요한 데이터를 HTTP 요청 경로/쿼리 문자열의 일부로 또는 Lambda 함수의 표준 출력으로 전송, 로깅 및 저장하지 않는 것이 좋습니다.

중요한 데이터가 암호화되지 않은 경우에는 API Gateway 에서 로깅을 활성화하지 않는 것이 좋습니다. 탐지 제어 하위 섹션에서 언급한 바와 같이 이러한 경우 API Gateway 로깅을 활성화하기 전에 규정 준수 팀에 문의해야 합니다.

SEC 3: 워크로드에서 애플리케이션 보안을 구현할 때는 어떤 방법을 사용합니까

AWS Well-Architected 프레임워크에서 다루는 AWS 보안 공지 및 업계 위협 인텔리전스를 검토하십시오. 애플리케이션 보안에 대한 OWASP 지침도 적용됩니다.

인바운드 이벤트를 검증 및 삭제하고 비 서버리스 애플리케이션에 대해 일반적으로 수행하는 보안 코드 검토를 수행하십시오. API Gateway 의 경우 기본 요청 검증을 첫 번째 단계로 설정하여 요청이 구성된 JSON 스키마 요청 모델과 URI, 쿼리 문자열 또는 헤더의 필수 파라미터를 준수하는지 확인합니다. 별도의 Lambda 함수, 라이브러리, 프레임워크 또는 서비스 등에 대해 애플리케이션별 심층 검증을 구현해야 합니다.

데이터베이스 암호 또는 API 키와 같은 보안 암호를 교체, 보안 및 감사 액세스를 허용하는 Secrets Manager 에 저장합니다. Secrets Manager 를 사용하면 감사를 비롯한 세분화된 정책을 암호에 적용할 수 있습니다.

주요 AWS 서비스

보안을 위한 주요 AWS 서비스로는 Amazon Cognito, IAM, Lambda, CloudWatch Logs, AWS CloudTrail, AWS CodePipeline, Amazon S3, Amazon ES, DynamoDB 및 Amazon Virtual Private Cloud(Amazon VPC)가 있습니다.

리소스

보안 관련 AWS 모범 사례에 대해 자세히 알아보려면 다음 리소스를 참조하십시오.

설명서 및 블로그

- Amazon S3 를 사용한 Lambda 함수의 IAM 역할 예제[23]
- API Gateway 요청 검증[24]
- API Gateway Lambda 권한 부여자[25]
- Amazon Cognito 연동 자격 증명, Amazon Cognito 사용자 풀 및 Amazon API Gateway 로 API 액세스 보호[26]
- AWS Lambda 에 대한 VPC 액세스 구성[27]
- Squid 프록시를 사용하여 VPC 아웃바운드 트래픽 필터링[28]
- Lambda 에서 AWS Secrets Manager 사용
- AWS Secrets Manager 를 사용하여 암호 감사
- OWASP 입력 검증 치트 시트
- AWS 서버리스 보안 워크숍

백서

- OWASP 보안 코딩 모범 사례[29]
- AWS 보안 모범 사례[30]

파트너 솔루션

- PureSec 서버리스 보안
- Twistlock 서버리스 보안[31]
- Protego 서버리스 보안
- Snyk – 상용 취약성 DB 및 종속성 검사[32]
- Lambda 및 API Gateway 에서 Hashicorp 볼트 사용

타사 도구

- OWASP 취약성 종속성 검사[33]

안정성 기반

안정성 기반에는 인프라 또는 서비스 중단으로부터 시스템을 복구하고, 수요를 충족할 컴퓨팅 리소스를 동적으로 확보하며, 잘못된 구성 또는 일시적 네트워크 문제와 같은 중단을 완화하는 기능이 포함됩니다.

정의

클라우드의 안정성에는 3 가지 모범 사례 영역이 있습니다.

- 기반
- 변경 관리
- 오류 관리

안정성을 달성하려면 잘 계획된 기반과 모니터링 기능을 갖춘 시스템과 수요 변경 또는 요구 사항을 처리하거나 잠재적으로 무단 서비스 거부 공격을 방어할 수 있는 메커니즘이 필요합니다. 또한 오류를 탐지하고 자동으로 해결할 수 있도록 시스템을 설계해야 합니다.

모범 사례

기반

REL 1: 인바운드 요청 속도를 어떻게 규제하고 있습니까?

조절

마이크로서비스 아키텍처에서 API 소비자는 별도의 팀에 속하거나 조직 외부에 있을 수 있습니다. 따라서 액세스 패턴을 알 수 없고 소비자 자격 증명이 손상될 수 있는 위험으로 인한 취약성이 발생합니다. 요청 수가 처리 로직/백엔드에서 처리할 수 있는 수를 초과할 경우 서비스 API 가 잠재적으로 영향을 받을 수 있습니다.

또한 새 트랜잭션을 트리거하는 이벤트(예: 데이터베이스 행의 업데이트 또는 API 의 일부로 S3 버킷에 추가되는 새 객체)가 발생하면 서버리스 애플리케이션 전체에서 추가 실행이 트리거됩니다.

서비스 계약을 통해 설정된 액세스 패턴을 적용하려면 API 수준에서 조절을 활성화해야 합니다. 리소스 수준 또는 글로벌 수준에서 소비자가 서비스를 사용하는 방법을 설정하려면 요청 액세스 패턴 전략을 정의하는 것이 기본입니다.

API 내에서 적절한 HTTP 상태 코드(예: 조절의 경우 429)를 반환하면 소비자가 백오프 및 재시도를 적절히 구현하여 제한된 액세스를 계획할 수 있습니다.

더 세분화된 사용량 조절 및 측정을 원한다면 글로벌 조절에 더해 사용량 계획을 통해 소비자에게 API 키를 발급할 수 있습니다. 그러면 API Gateway 가 예기치 않은 동작에서 할당량과 액세스 패턴을 적용할 수 있습니다. 또한 API 키를 사용하면 개별 소비자가 의심스러운 요청을 할 때 관리자가 간편하게 액세스를 차단할 수 있습니다.

API 키를 캡처하는 일반적인 방법은 개발자 포털을 사용하는 것입니다. 개발자 포털에서 서비스 공급자는 소비자 및 요청과 관련된 추가 메타데이터를 사용할 수 있습니다. 애플리케이션, 연락처 정보 및 비즈니스 영역/목적을 캡처하고 DynamoDB 와 같은 내구력 있는 데이터 스토어에 이 데이터를 저장할 수 있습니다. 이렇게 하면 소비자를 추가로 검증하고 자격 증명을 사용한 로깅 추적 기능을 활용하여 소비자와 연락함으로써 변경 업그레이드/문제를 해결할 수 있습니다.

보안 기반에서 설명한 대로 API 키는 요청을 승인하는 보안 메커니즘이 아닙니다. 따라서 API Gateway 내에서 사용 가능한 권한 부여 옵션 중 하나와 함께 사용해야 합니다.

특정 워크로드는 Lambda 와 같은 속도로 확장되지 않을 수 있으므로 서비스 장애로부터 특정 워크로드를 보호하는 동시성 제어가 필요할 수 있습니다. 동시성 제어를 사용하면 개별 Lambda 함수 수준에서 특정 Lambda 함수의 동시 호출 수 할당 설정을 제어할 수 있습니다.

개별 함수의 동시성 세트를 초과하는 Lambda 호출은 AWS Lambda 서비스를 통해 조절되며 결과는 이벤트 소스에 따라 달라집니다. 동기식 호출의 경우 HTTP 429 오류를 반환하고, 비동기식 호출의 경우 대기열에 추가한 후 재시도하며, 스트림 기반 이벤트 소스의 경우 레코드 만료 시간까지 재시도합니다.

그림 17: AWS Lambda 동시성 제어

동시성 제어는 다음 시나리오에서 특히 유용합니다.

- 확장이 제한될 수 있는 중요한 백엔드 또는 통합 시스템
- 동시성 제한을 적용할 수 있는 관계형 데이터베이스와 같은 데이터베이스 연결 풀 제한
- 중요한 경로 서비스: 동일한 계정의 제한을 기준으로 우선 순위가 높은 Lambda 함수(예: 권한 부여 함수와 백오피스 같은 우선 순위가 낮은 함수 비교)
- 이상 발생 시 Lambda 함수(동시성 = 0)를 비활성화하는 기능.
- 원하는 실행 동시성을 제한하여 DDoS(분산 서비스 거부) 공격 차단

Lambda 함수에 대한 동시성을 제어하면 동시성 세트 이상으로 확장하고 계정의 예약된 동시성 풀에서 가져올 수 있는 기능도 제한됩니다. 비동기식 처리의 경우 Kinesis Data Streams 를 사용하여 Lambda 함수의 동시성 제어와 달리 단일 샤드를 통해 효과적으로 동시성을 제어할 수

있습니다. 따라서 샤드 수 또는 병렬화 인수를 늘려 Lambda 함수의 동시성을 유연하게 확장할 수 있습니다.

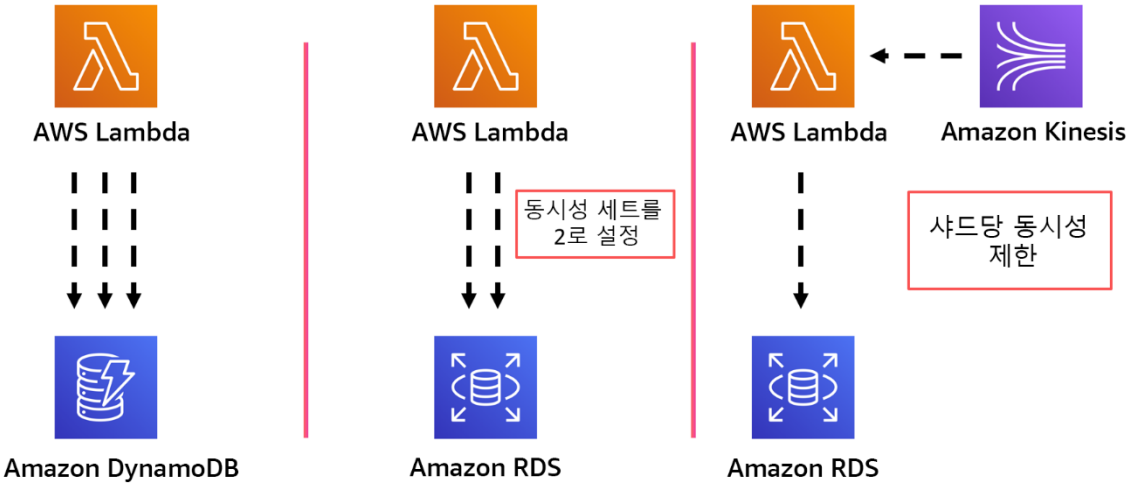

그림 18: 동기식 및 비동기식 요청에 대한 동시성 제어

REL 2: 서버리스 애플리케이션 안에 복원력을 구축할 때 어떤 방법을 사용하고 있습니까?

비동기식 호출 및 이벤트

비동기식 호출은 HTTP 응답의 지연 시간을 줄입니다. 다수의 동기식 호출과 장기 실행되는 대기 주기는 제한 시간을 초과하고 "잠긴" 코드를 야기하여 재시도 로직을 차단할 수 있습니다.

이벤트 중심 아키텍처를 사용하면 비동기식 코드 실행을 간소화하여 소비자 대기 주기를 제한할 수 있습니다. 이러한 아키텍처는 일반적으로 비즈니스 기능을 수행하는 여러 구성 요소에 걸쳐 대기열, 스트림, 게시/구독, 웹후크, 상태 시스템 및 이벤트 규칙 관리자를 사용하여 비동기식으로 구현됩니다.

사용자 경험은 비동기식 호출과 분리됩니다. 전체 실행이 완료될 때까지 전체 환경을 차단하는 대신 프런트엔드 시스템에서 초기 요청의 일부로 참조/작업 ID 를 수신하고 실시간 변경을 구독하거나 레거시 시스템에서는 추가 API 를 사용하여 상태를 폴링합니다. 이러한 분리를 활용하면 이벤트 루프, 병렬 또는 동시성 기술을 사용하여 프런트엔드의 효율성을 개선하는 동시에 응답이 부분 또는 전체적으로 제공될 때 애플리케이션의 일부를 느리게 로드하면서 요청을 수행할 수 있습니다.

사용자 지정 재시도 및 캐싱을 통해 프런트엔드를 강화하면 비동기식 호출의 주요 요소로 프런트엔드를 사용할 수 있습니다. 비정상 동작, 일시적인 조건, 네트워킹 또는 환경의 성능 저하로 인해 허용 가능한 SLA 내에 응답이 수신되지 않은 경우 전송 중인 요청을 중지할 수 있습니다.

또는 동기식 호출이 필요한 경우에는 최소한 총 실행 시간이 API Gateway 또는 AWS AppSync 최대 제한 시간을 초과하지 않는지 확인하는 것이 좋습니다. 외부 서비스(예: AWS Step Functions)를 사용하여 여러 서비스의 비즈니스 트랜잭션을 조정하고 상태를 제어하고 요청 수명 주기에 따라 발생하는 오류 처리를 처리합니다.

변경 관리

변경 관리는 AWS Well-Architected 프레임워크에서 다루며, 운영 우수성 기반에서 서버리스에 대한 특정 정보를 찾을 수 있습니다.

오류 관리

서버리스 애플리케이션의 특정 부분은 다양한 구성 요소에 대한 이벤트 기반 방식의 비동기식 호출(예: 게시/구독 및 기타 패턴)로 설명됩니다. 비동기식 호출이 실패하면 가능할 때마다 이를 캡처하고 재시도해야 합니다. 그렇지 않으면 데이터 손실이 발생하여 고객 경험이 저하될 수 있습니다.

Lambda 함수의 경우 Lambda 쿼리에 재시도 로직을 구축하여 워크로드 급증 시 백엔드가 영향을 받지 않도록 해야 합니다. 운영 우수성 기반에서 다룬 것과 같이 구조화된 로깅을 사용하여 재시도를 기록합니다. 여기에는 사용자 지정 지표로 캡처될 수 있는 오류에 대한 컨텍스트 정보가 포함됩니다. Lambda 대상을 사용하여 오류, 스택 추적 및 재시도에 대한 컨텍스트 정보를 SNS 주제 및 SQS 대기열과 같은 전용 DLQ(배달 못한 편지 대기열)로 전송합니다. 또한 실패한 이벤트를 의도한 서비스로 다시 보내는 별도의 메커니즘으로 폴링하는 계획을 개발하는 것이 좋습니다.

AWS SDK 는 다른 AWS 서비스와 통신할 때 대부분의 경우 충분한 백오프 및 재시도 메커니즘을 기본적으로 제공합니다. 그러나 요구 사항 특히 HTTP keepalive, 연결 및 소켓 제한 시간에 적합하게 이러한 메커니즘을 검토하고 튜닝하십시오.

가능하면 Step Functions 를 사용하여 서버리스 애플리케이션 내의 사용자 지정 시도/가져오기, 백오프 및 재시도 로직의 양을 최소화합니다. 자세한 내용은 비용 최적화 기반 섹션을 참조하십시오. Step Functions 통합을 사용하여 실패한 상태 실행과 그 상태를 DLQ 에 저장합니다.

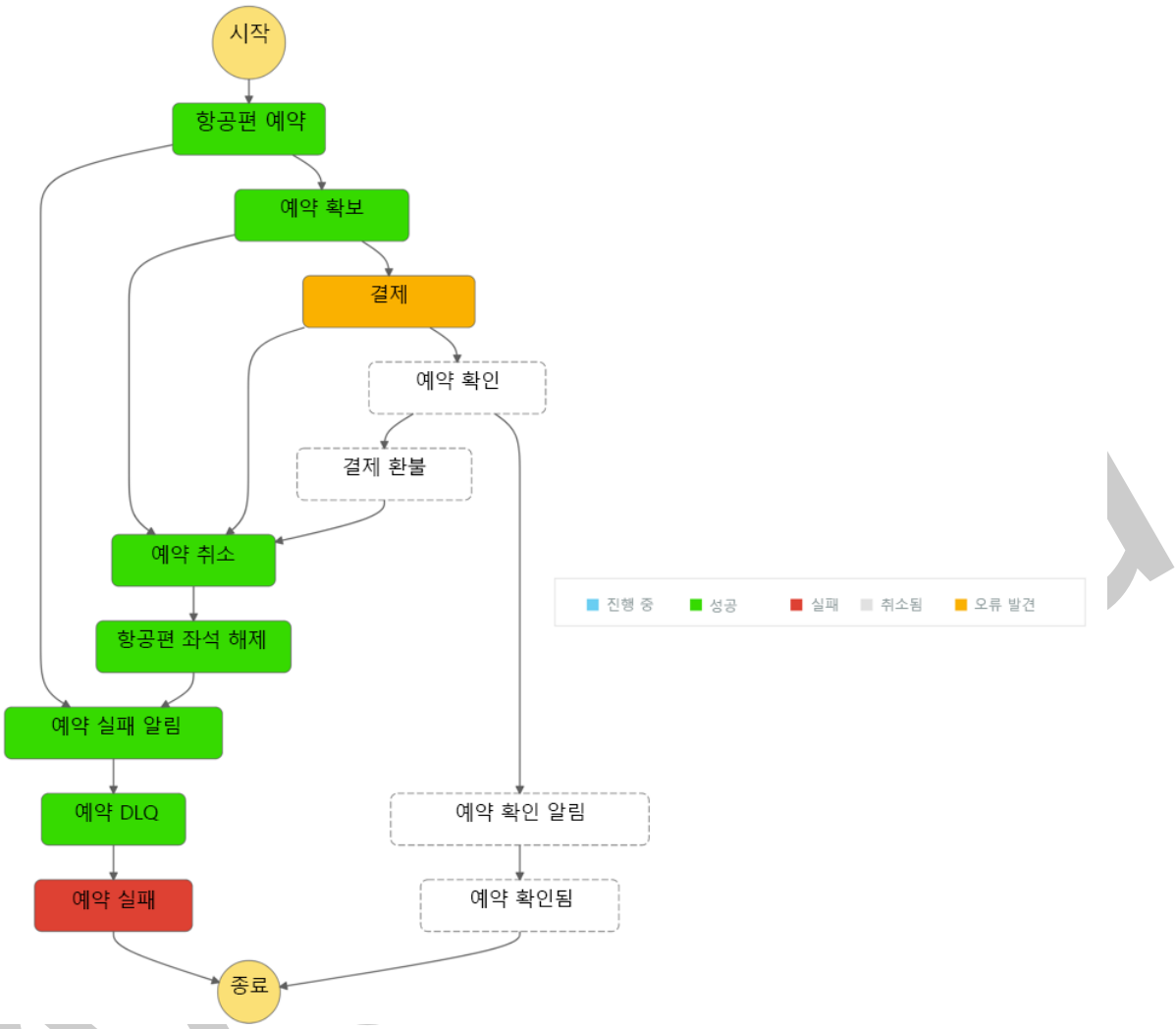

그림 19: DLQ 단계가 포함된 Step Functions 상태 시스템

PutRecords(Kinesis)및 BatchWriteItem(DynamoDB)과 같은 비원자성 작업에서는 하나 이상의 레코드가 성공적으로 수집된 경우 성공을 반환하기 때문에 부분적인 실패가 발생할 수 있습니다. 이러한 작업을 사용할 때는 항상 응답을 검사하고 부분적 실패를 프로그래밍 방식으로 처리합니다.

Kinesis 또는 DynamoDB 스트림에서 사용할 경우 최대 레코드 수명, 최대 재시도 횟수, 오류 시 DLQ 및 함수 오류 시 이등분 배치와 같은 Lambda 오류 처리 제어를 사용하여 애플리케이션의 복원력을 추가로 구축합니다.

트랜잭션 기반이고 특정 보장 및 요구 사항이 따르는 동기식 파트의 경우 Saga 패턴[34]에 설명된 대로 애플리케이션의 로직을 분리하고 간소화하는 Step Functions 상태 시스템을 사용하여 실패한 트랜잭션을 롤백할 수도 있습니다.

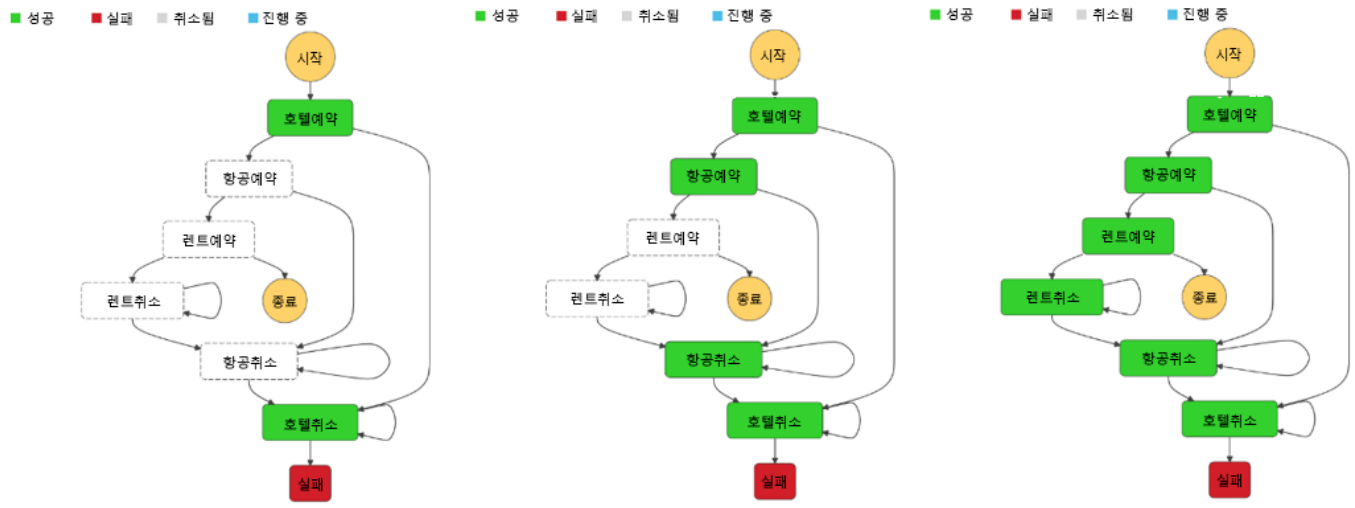

그림 20: Yan Cui 를 사용한 Step Functions 의 Saga 패턴

제한

Well-Architected 프레임워크에서 다루는 내용에 더해 버스트 및 스파이크 사용 사례에 대한 제한을 검토하십시오. 예를 들어 API Gateway 와 Lambda 는 안정적인 요청 빈도와 버스트 요청 빈도에 대한 제한이 서로 다릅니다. 가능한 경우 조정 계층과 비동기식 패턴을 사용하고 로드 테스트를 수행하여 현재 계정 한도로 실제 고객 수요를 유지할 수 있는지 확인합니다.

주요 AWS 서비스

안정성을 위한 주요 AWS 서비스로는 AWS Marketplace, Trusted Advisor, CloudWatch Logs, CloudWatch, API Gateway, Lambda, X-Ray, Step Functions, Amazon SQS 및 Amazon SNS 가 있습니다.

리소스

안정성 관련 AWS 모범 사례에 대해 자세히 알아보려면 다음 리소스를 참조하십시오.

설명서 및 블로그

- Lambda 한도[35]
- API Gateway 제한[36]
- Kinesis Streams 제한[37]
- DynamoDB 제한[38]
- Step Functions 제한[39]
- 오류 처리 패턴[40]
- Lambda 를 사용한 서버리스 테스트[41]
- Lambda 함수 로그 모니터링[42]
- Lambda 버전 관리[43]
- API Gateway 의 단계[44]
- AWS 의 API 재시도[45]
- Step Functions 오류 처리[46]
- X-Ray[47]
- Lambda DLQ[48]
- API Gateway 및 Lambda 를 사용한 오류 처리 패턴[49]
- Step Functions Wait 상태[50]
- Saga 패턴[51]
- Step Functions 를 통해 Saga 패턴 적용[52]
- 서버리스 애플리케이션 리포지토리 앱 – DLQ Redriver
- AWS SDK 관련 재시도 및 시간 초과 문제 해결
- 스트림 처리에 대한 Lambda 복원력 제어
- Lambda 대상
- 서버리스 애플리케이션 리포지토리 앱 – 이벤트 재생
- 서버리스 애플리케이션 리포지토리 앱 – 이벤트 스토리지 및 백업

백서

- AWS 기반 마이크로서비스[53]

성능 효율성 기반

성능 효율성 기반에서는 컴퓨팅 리소스를 효율적으로 사용하여 요구 사항을 충족하고 수요 변경 및 기술 변화에 따라 이러한 효율성을 유지하는 데 중점을 둡니다.

정의

클라우드의 성능 효율성은 다음 4 가지 영역으로 구성됩니다.

- 선택
- 검토
- 모니터링
- 절충

고성능 아키텍처를 선택할 때는 데이터 기반 접근 방식을 취합니다. 상위 수준 설계에서 리소스 유형의 선택 및 구성에 이르기까지 아키텍처의 모든 측면에 대한 데이터를 수집합니다. 주기적으로 선택 사항을 검토함으로써 지속적으로 변화하는 AWS 클라우드를 최대한 활용할 수 있습니다.

모니터링을 활용하면 예상 성능과의 차이를 인지하고 관련 조치를 취할 수 있습니다. 마지막으로, 압축 또는 캐싱을 사용하거나 일관성 요구 사항을 완화하는 등 아키텍처의 각 부분을 적절하게 절충하여 성능을 개선할 수 있습니다.

PER 1: 서버리스 애플리케이션의 성능을 최적화할 때 어떤 방법을 사용했습니까?

선택

안정적인 비율 및 버스트 비율을 사용하여 서버리스 애플리케이션에서 성능 테스트를 실행합니다. 결과를 사용하여 최적의 구성을 선택하는 데 도움이 되는 용량 단위 튜닝 및 변경 후 로드 테스트를 시행합니다.

- **Lambda:** CPU, 네트워크 및 스토리지 IOPS 는 비례하여 할당되므로 서로 다른 메모리 설정을 테스트합니다.
- **API Gateway:** 지리적으로 분산된 고객에게는 엣지 엔드포인트를 사용합니다. 각 리전의 고객과 동일한 리전에서 다른 AWS 서비스를 사용하는 고객에게는 리전 엔드포인트를 사용합니다.
- **DynamoDB:** 예측할 수 없는 애플리케이션 트래픽에는 온디맨드를 사용하고, 일관된 트래픽에는 프로비저닝된 모드를 사용합니다.
- **Kinesis:** 여러 소비자 시나리오에서 소비자당 전용 입력/출력 채널에 향상된 팬아웃을 사용합니다. Lambda 를 사용하는 적은 볼륨의 트랜잭션에는 확장된 배치 기간을 사용합니다.

Lambda 함수에 대한 VPC 액세스는 필요한 경우에만 구성합니다. VPC 지원 Lambda 함수에서 인터넷에 액세스해야 하는 경우 NAT 게이트웨이를 설정합니다. Well-Architected 프레임워크에 설명되어 있듯이 여러 가용 영역에 걸쳐 NAT 게이트웨이를 구성하여 고가용성과 성능을 보장합니다.

API Gateway 엣지 최적화 API 는 지리적으로 분산된 소비자의 액세스를 최적화하기 위해 완전관리형 CloudFront 배포를 제공합니다. 가장 가까운 CloudFront PoP(Point of Presence)로 API 요청이 라우팅되므로 일반적으로 연결 시간이 개선됩니다.

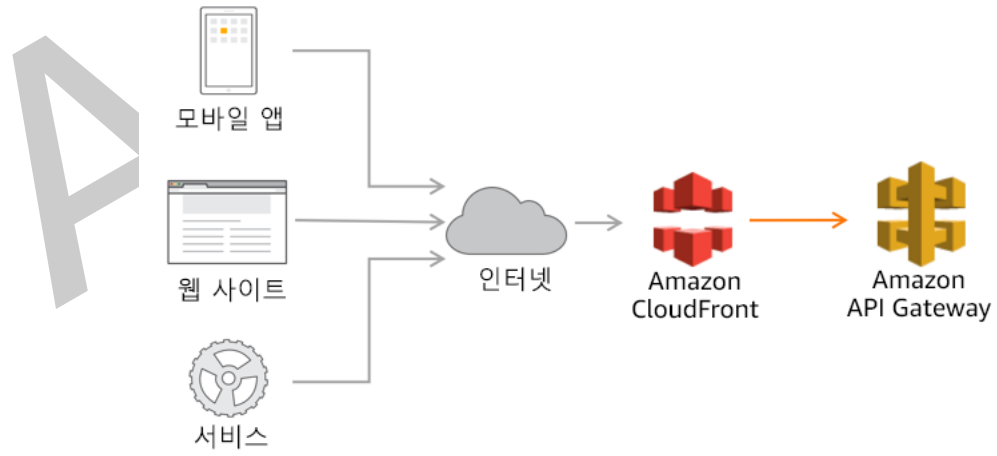

그림 21: 엣지 최적화 API Gateway 배포

API Gateway 리전 엔드포인트는 CloudFront 배포를 제공하지 않고 기본적으로 HTTP2 를 활성화하므로 동일한 리전에서 요청이 시작되는 경우 전체 지연 시간을 줄일 수 있습니다. 리전 엔드포인트를 사용하는 경우 자체 Amazon CloudFront 배포 또는 기존 CDN 을 연결할 수도 있습니다.

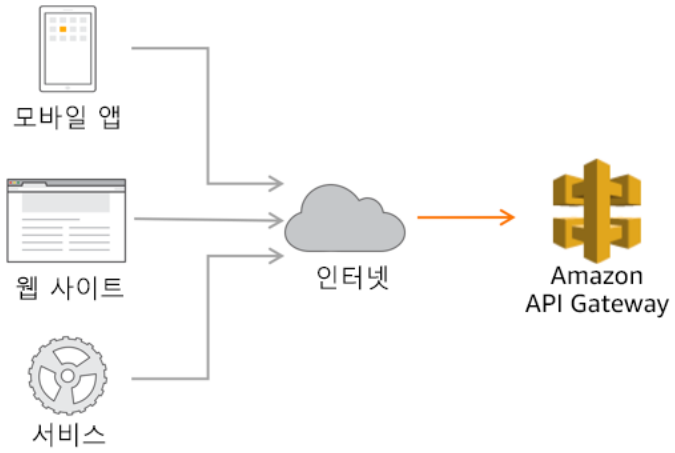

그림 22: 리전 엔드포인트 API Gateway 배포

다음 표는 엣지 최적화 API 를 배포할지 리전 API 엔드포인트를 배포할지를 결정하는 데 도움이 될 수 있습니다.

	엣지 최적화 API	리전 API 엔드포인트
리전 전체에서 API 에 액세스합니다. API Gateway 관리형 CloudFront 배포가 포함됩니다.	X	
동일한 리전 내에서 API 에 액세스합니다. API 가 배포된 리전과 동일한 리전에서 API 에 액세스하는 경우 요청 지연 시간이 최소화됩니다.		X
자체 CloudFront 배포를 연결할 수 있습니다.		X

다음 의사 결정 트리는 Lambda 함수를 VPC 에 배포해야 하는 경우를 결정하는 데 도움이 될 수 있습니다.

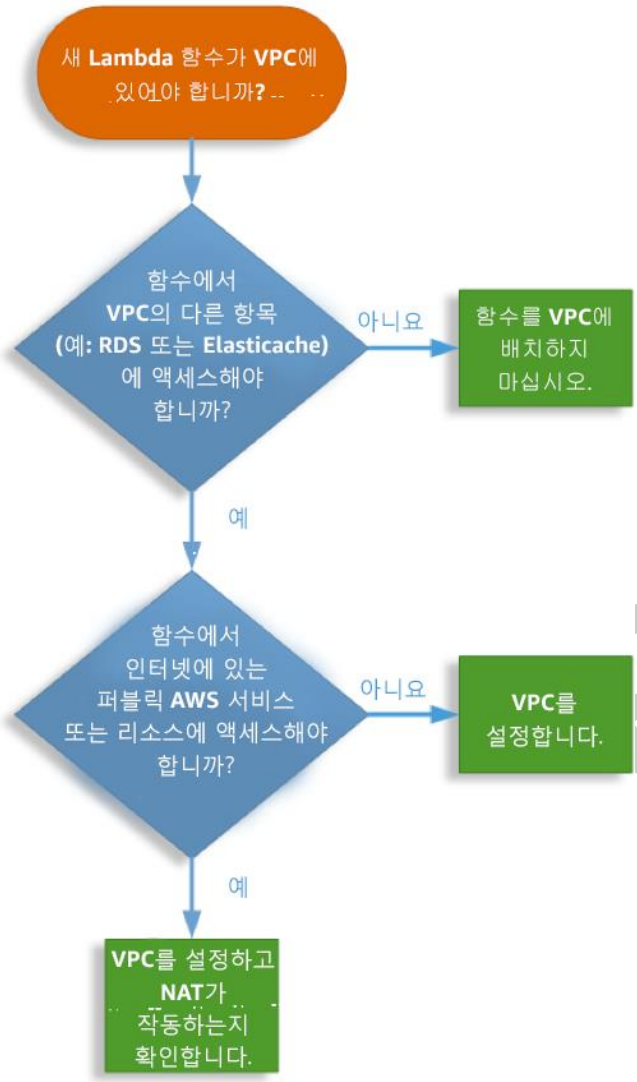

그림 23: Lambda 함수를 VPC 에 배포해야 하는 경우에 대한 의사 결정 트리

최적화

서버리스 아키텍처가 유기적으로 확장되면 일반적으로 다양한 워크로드 프로필에서 특정 메커니즘이 사용됩니다. 애플리케이션의 성능을 개선하려면 설계 테스트를 수행하는 것 외에도 SLA 및 요구 사항을 기반으로 설계 절충을 고려해야 합니다.

API Gateway 및 AWS AppSync 캐싱을 활성화하면 해당하는 작업의 성능을 개선할 수 있습니다. DAX 를 사용하면 읽기 응답이 크게 개선되며 글로벌 및 로컬 보조 인덱스가 개선되므로 DynamoDB 전체 테이블을 스캔하지 않아도 됩니다. 이러한 세부 정보 및 리소스는 모바일 백엔드 시나리오에 설명되어 있습니다.

API Gateway 콘텐츠 인코딩을 사용하면 API 클라이언트에서 API 요청에 대한 응답으로 다시 전송하기 전에 페이로드 압축을 요청할 수 있습니다. 이렇게 하면 API Gateway 에서 API 클라이언트로 전송되는 바이트 수가 줄어들고 데이터 전송에 걸리는 시간이 단축됩니다. API 정의에서 콘텐츠 인코딩을 활성화하고 압축을 트리거하는 최소 응답 크기를 설정할 수도 있습니다. 기본적으로 API 는 콘텐츠 인코딩 지원을 활성화하지 않습니다.

통신 경로에서 사용되는 다운스트림 서비스의 일시적 문제를 고려하여 함수 제한 시간을 평균 실행보다 몇 초 높게 설정합니다. 이는 Step Functions 활동, 작업 및 SQS 메시지 가시성 관련 작업에도 적용됩니다.

AWS Lambda 에서 기본 메모리 설정 및 제한 시간을 선택하면 성능, 비용 및 운영 절차에 원치 않는 영향을 줄 수 있습니다.

평균 실행보다 제한 시간을 훨씬 높게 설정하면 코드 오작동 시 함수가 더 오래 실행되어 비용이 높아지고 함수가 호출되는 방식에 따라 동시성 한도에 도달할 수 있습니다.

함수를 1 번 실행하는 것과 동일한 제한 시간을 설정하면 일시적인 네트워킹 문제 또는 다운스트림 서비스의 이상이 발생할 경우 서버리스 애플리케이션이 실행을 갑자기 중지할 수 있습니다.

로드 테스트를 수행하지 않고, 특히 업스트림 서비스를 고려하지 않고 제한 시간을 설정하면 파트가 제한 시간에 먼저 도달할 때마다 오류가 발생할 수 있습니다.

컨테이너를 재사용하고, 배포 패키지 크기를 필요한 실행 시간으로 최소화하며, 빠른 시작에 최적화되지 않을 수 있는 프레임워크를 포함한 종속성의 복잡성을 최소화하는 등 Lambda 함수 작업에[54] 대한 모범 사례를 따르십시오. 지연 시간의 99 번째 백분위수(P99)를 항상 고려하여 다른 팀과 협의된 애플리케이션 SLA 에 영향을 미치는 일이 없도록 합니다.

VPC 에서 Lambda 함수를 사용하는 경우 VPC 에 있는 기본 리소스의 퍼블릭 호스트 이름을 DNS 확인에 사용하지 마십시오. 예를 들어 Lambda 함수가 VPC 의 Amazon RDS DB 인스턴스에 액세스하는 경우 공개적으로 액세스할 수 없는 옵션을 사용하여 인스턴스를 시작합니다.

Lambda 함수가 실행된 후 AWS Lambda 는 다른 Lambda 함수 호출을 예상하여 일정 시간 동안 실행 컨텍스트를 유지합니다. 따라서 비용이 높은 일회성 작업(예: 데이터베이스 연결 또는 모든 초기화 로직 설정)의 글로벌 범위를 사용할 수 있습니다. 후속 호출에서는 이 컨텍스트가 여전히 유효한지 확인하고 기존 연결을 재사용할 수 있습니다.

비동기식 트랜잭션

고객은 최신의 대화형 사용자 인터페이스를 기대합니다. 따라서 동기식 트랜잭션을 사용해서는 더 이상 복잡한 워크플로를 지속할 수 없습니다. 필요한 서비스 상호 작용의 수가 많을수록 호출 체이닝 수가 늘어나므로 서비스 안정성 및 응답 시간에 대한 위험이 증가할 수 있습니다.

Angular.js, VueJS 및 React 같은 최신 UI 프레임워크, 비동기식 트랜잭션 및 클라우드 네이티브 워크플로는 고객 수요를 충족하는 데 적합한 접근 방식을 제공하며 구성 요소를 분리하고 프로세스 및 비즈니스 도메인에 집중하는 데 도움이 됩니다.

이러한 비동기식 트랜잭션(또는 종종 이벤트 중심 아키텍처로 설명됨)은 응답에 대한 잠금 및 대기(I/O 차단) 상태로 클라이언트를 제한하지 않고 이후에 발생하는 다운스트림 이벤트를 클라우드에서 시작합니다. 비동기식 워크플로는 데이터 수집, ETL 작업, 주문/요청 이행 등(이에 국한되지 않음)의 다양한 사용 사례를 처리합니다.

이러한 사용 사례에서 데이터는 도착하는 대로 처리되고 변경되는 즉시 검색됩니다. 통합 및 비동기식 처리를 위한 몇 가지 최적화 패턴을 알아볼 수 있는 2 가지 일반적인 비동기식 워크플로에 대한 모범 사례를 간략히 설명하겠습니다.

서버리스 데이터 처리

서버리스 데이터 처리 워크플로에서 데이터는 클라이언트에서 Kinesis 로 수집(Kinesis 에이전트, SDK 또는 API 사용)되어 Amazon S3 에 도착합니다.

새 객체에서는 자동으로 실행되는 Lambda 함수가 시작됩니다. 일반적으로 이 함수는 추가 처리를 위해 데이터를 변환하거나 분할하는 데 사용되며 DynamoDB 와 같은 다른 대상 또는 다른 S3 버킷에 데이터의 최종 형식으로 저장될 수 있습니다.

데이터 유형에 따라 사용되는 변환이 다를 수 있으므로 최적의 성능을 위해 변환을 여러 Lambda 함수로 세분화하는 것이 좋습니다. 이 접근 방식을 사용하면 데이터 변환을 병렬로 실행하고 속도와 비용을 유연하게 개선할 수 있습니다.

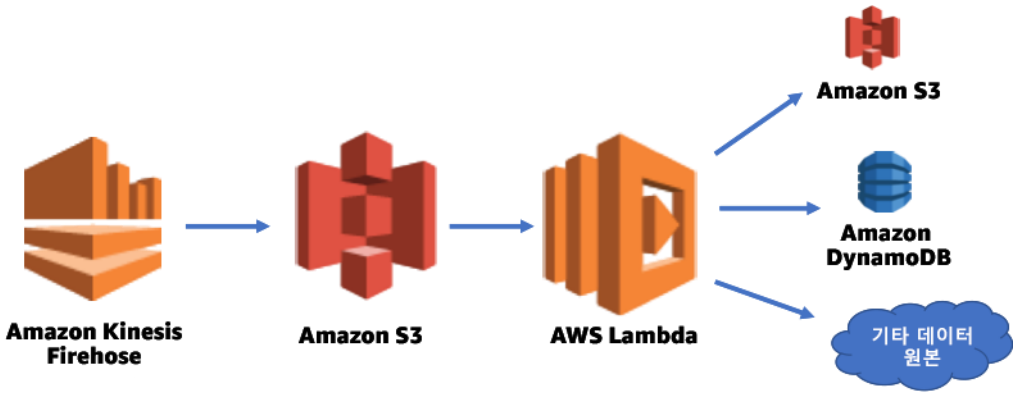

그림 24: 비동기식 데이터 수집

Kinesis Data Firehose 는 Lambda 의 대안으로 사용할 수 있는 기본 데이터 변환을 제공합니다. 이 변환을 사용하는 경우 추가 로직 없이 Apache Log/System 로그의 레코드를 CSV, JSON 으로, JSON 을 Parquet 또는 ORC 로 변환할 수 있습니다.

상태 업데이트를 통한 서버리스 이벤트 제출

예를 들어 전자 상거래 사이트에서 고객이 주문을 제출하여 재고 차감 및 배송 프로세스가 시작되거나, 엔터프라이즈 애플리케이션에서 응답까지 몇 분이 소요될 수 있는 다량의 쿼리를 제출한다고 가정합시다.

이 일반적인 트랜잭션을 완료하는 데 필요한 프로세스에는 완료까지 몇 분이 소요될 수 있는 여러 번의 서비스 호출이 필요할 수 있습니다. 이러한 호출 내에서 재시도 및 지수 백오프를 추가하여 잠재적 장애를 방지하려고 한다면 트랜잭션 완료를 기다리는 모든 사용자에게 최적의 사용자 경험을 제공하지 못할 수 있습니다.

이와 유사한 길고 복잡한 워크플로의 경우 API Gateway 또는 AWS AppSync 를 Step Functions 와 통합할 수 있습니다. 권한이 부여된 새 요청이 수신되면 이 비즈니스 워크플로가 시작되고 Step Functions 는 호출자(모바일 앱, SDK, 웹 서비스 등)에게 실행 ID 로 즉시 응답합니다.

레거시 시스템의 경우 실행 ID 를 사용하여 다른 REST API 를 통해 비즈니스 워크플로 상태에 대한 Step Functions 를 폴링할 수 있습니다. WebSocket 을 사용하면 REST 를 사용하든 GraphQL 을 사용하든, 워크플로의 모든 단계에서 업데이트를 제공하여 비즈니스 워크플로 상태를 실시간으로 수신할 수 있습니다.

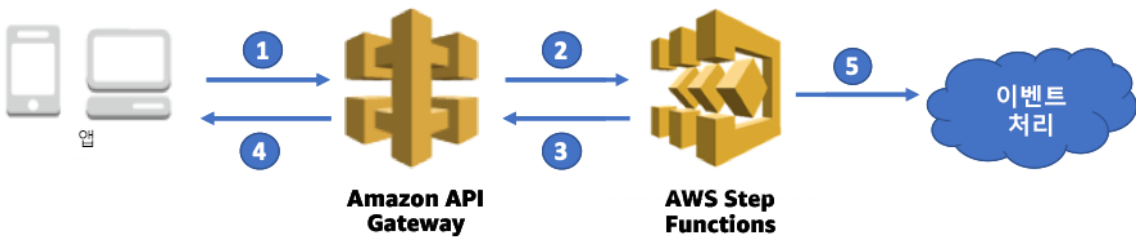

그림 25: Step Functions 상태 시스템을 사용하는 비동기식 워크플로

또 다른 일반적인 시나리오는 API Gateway 를 조정 계층으로 SQS 또는 Kinesis 와 직접 통합하는 것입니다. Lambda 함수는 호출자로부터 추가 비즈니스 정보 또는 사용자 지정 요청 ID 형식을 받아야 하는 경우에만 필요합니다.

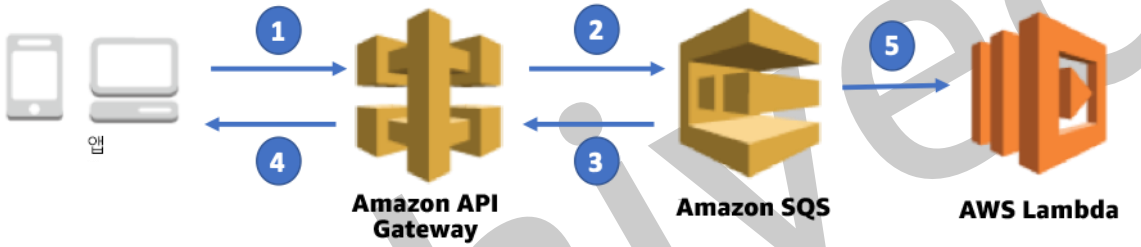

그림 26: 대기열을 조정 계층으로 사용하는 비동기식 워크플로

이 두 번째 예에서는 SQS 가 여러 용도로 사용됩니다.

1. 요청 레코드를 내구력 있는 위치에 저장하면 클라이언트가 요청이 결국 처리될 것임을 확신할 수 있으므로 워크플로 처리를 계속할 수 있습니다.
2. 일시적으로 백엔드 과부하를 야기할 수 있는 이벤트 버스트가 발생하면 요청을 폴링하여 리소스를 사용할 수 있을 때 처리할 수 있습니다.

대기열이 없는 첫 번째 예제와 비교하여 Step Functions 는 대기열이나 상태 추적 데이터 원본 없이도 데이터를 안정적으로 저장합니다. 두 예제 모두에서 모범 사례는 클라이언트가 요청을 제출한 후 비동기식 워크플로를 실행하도록 하고 완료에 몇 분이 소요될 수 있는 경우 결과 응답을 차단 코드로 반환하지 않는 것입니다.

WebSocket 을 사용하는 경우 AWS AppSync 는 GraphQL 구독을 통해 즉시 이 기능을 제공합니다. 구독을 사용하면 권한 있는 클라이언트가 관심 데이터 변형의 수신을 대기할 수 있습니다. 이는 스트리밍 중인 데이터 또는 둘 이상의 응답을 반환할 수 있는 데이터에 적합합니다.

AWS AppSync 를 사용하는 경우 클라이언트는 DynamoDB 의 상태 업데이트가 변경될 때 자동으로 업데이트를 구독하고 수신할 수 있으며 이 패턴은 데이터 기반 사용자 인터페이스에 아주 적합합니다.

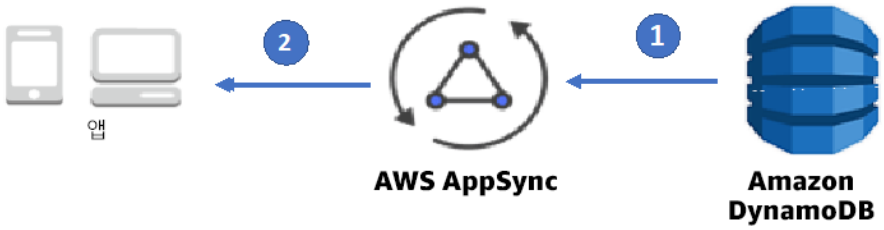

그림 27: AWS AppSync 및 GraphQL 에서 WebSocket 을 통한 비동기식 업데이트

웹후크를 구현하려면 SNS 주제 HTTP 구독을 사용합니다. 소비자는 이벤트(예: Amazon S3 에 데이터 파일 도착) 발생 시 SNS 가 POST 메서드를 통해 콜백하는 HTTP 엔드포인트를 호스팅할 수 있습니다. 이 패턴은 엔드포인트를 호스팅할 수 있는 다른 마이크로서비스로 클라이언트를 구성할 수 있는 경우에 적합합니다. 또는 지정된 작업에 대한 응답을 수신할 때까지 상태 시스템을 차단하는 콜백을 Step Functions 로 사용할 수도 있습니다.

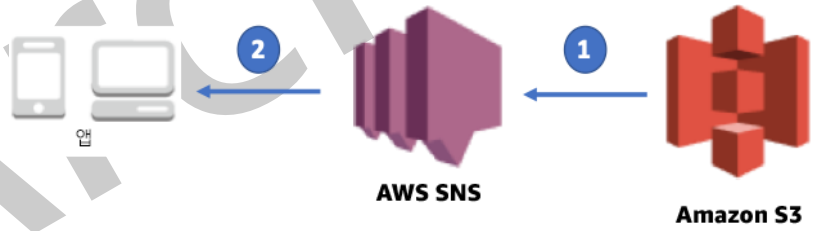

그림 28: SNS 에서 웹후크를 통한 비동기식 알림

마지막으로, 폴링은 여러 클라이언트가 상태 확인을 위해 API 를 지속적으로 폴링하므로 비용과 리소스 측면에서 모두 많은 비용을 야기할 수 있습니다. 환경 제약으로 인해 폴링이 유일한 옵션인 경우 클라이언트에서 SLA 를 설정하여 "빈 폴링" 수를 제한하는 것이 모범 사례입니다.

그림 29: 최근 트랜잭션 업데이트에 대한 클라이언트 폴링

예를 들어 대용량 데이터 웨어하우스 쿼리의 응답에 평균 2 분이 소요된다면, 클라이언트는 데이터를 사용할 수 없을 때 2 분을 기다린 후 지수 백오프를 사용하여 API 를 폴링해야 합니다. 클라이언트의 폴링 빈도가 예상 빈도를 초과하지 않도록 하려면 조절과 타임스탬프라는 2 가지 일반적인 패턴을 사용하여 다시 폴링해도 괜찮은 시기를 지정할 수 있습니다.

타임스탬프를 사용하는 경우 폴링되는 시스템에서는 소비자가 다시 폴링해도 괜찮은 시기에 관한 타임스탬프 또는 기간이 포함된 추가 필드를 반환할 수 있습니다. 이 접근 방식은 소비자가 이 접근 방식을 준수하고 현명하게 사용할 것이라는 낙관적인 시나리오를 따르며, 남용이 발생할 경우 조절을 활용하여 구현의 완성도를 높일 수 있습니다.

검토

서버리스 애플리케이션에 적용되는 성능 효율성에 대한 **검토** 영역의 모범 사례는 AWS Well-Architected 프레임워크 백서를 참조하십시오.

모니터링

서버리스 애플리케이션에 적용되는 성능 효율성에 대한 **모니터링** 영역의 모범 사례는 AWS Well-Architected 프레임워크 백서를 참조하십시오.

절충

서버리스 애플리케이션에 적용되는 성능 효율성에 대한 **절충** 영역의 모범 사례는 AWS Well-Architected 프레임워크 백서를 참조하십시오.

주요 AWS 서비스

성능 효율성을 위한 주요 AWS 서비스로는 DynamoDB Accelerator, API Gateway, Step Functions, NAT 게이트웨이, Amazon VPC 및 Lambda 가 있습니다.

리소스

성능 효율성 관련 AWS 모범 사례에 대해 자세히 알아보려면 다음 리소스를 참조하십시오.

설명서 및 블로그

- AWS Lambda FAQ[55]
- AWS Lambda 함수 작업의 모범 사례[56]
- AWS Lambda: 작동 방식[57]
- AWS Lambda 의 컨테이너 재사용 이해[58]
- Amazon VPC 에서 리소스에 액세스하도록 Lambda 함수 구성[59]
- 응답 개선을 위해 API 캐싱 활성화[60]
- DynamoDB: 글로벌 보조 인덱스[61]
- Amazon DynamoDB Accelerator(DAX)[62]
- 개발자 안내서: Kinesis Streams[63]
- Java SDK: 성능 개선 구성
- Node.js SDK: HTTP Keep Alive 활성화
- Node.js SDK: 가져오기 개선
- Amazon SQS 대기열 및 AWS Lambda 를 사용하여 처리량 개선
- 향상된 팬아웃으로 스트림 처리 성능 개선
- Lambda 파워 튜닝
- Amazon DynamoDB 온디맨드 모드 및 프로비저닝 모드의 사용 시기
- Amazon CloudWatch Logs Insights 를 사용하여 로그 데이터 분석

- 여러 데이터 원본을 AWS AppSync 와 통합
- Step Functions 서비스 통합
- 캐싱 패턴
- 서버리스 애플리케이션 캐싱
- Amazon Athena 및 AWS Glue 모범 사례

비용 최적화 기반

비용 최적화 기반에는 전체 수명 주기 동안 시스템을 개선하고 구체화하는 지속적인 프로세스가 포함됩니다. 최초 개념 증명의 초기 설계부터 지속적인 프로덕션 워크로드 운영에 이르는 전체 과정에 이 문서의 사례를 도입하면 비용을 최소화하면서 원하는 비즈니스 결과를 달성할 수 있는 비용 인식 시스템을 구축하고 운영하여 기업의 투자 수익을 최대화할 수 있습니다.

정의

클라우드의 비용 최적화에는 4 가지 모범 사례 영역이 있습니다.

- 비용 효율적인 리소스
- 수요와 공급 일치
- 지출 인식
- 시간별 최적화

다른 기반과 마찬가지로 비용 최적화 기반에서도 절충을 고려해야 합니다. 예를 들어, 출시 시간 또는 비용을 최적화해야 한다면, 경우에 따라 선결제 비용 최적화에 투자하는 것보다 출시 시간을 단축하거나 새로운 기능을 배포하거나 단순히 납기를 준수하는 등 속도에 최적화하는 것이 가장 좋습니다.

가장 최적화된 비용의 배포에 대한 벤치마킹에 시간을 쓰기보다 "만약"을 대비해 과잉 보상을 하는 것이 더 쉽기 때문에 경험적인 데이터에 기반을 둔 결정이 아니라 서둘러 결정을 내리게 되는 경우도 있습니다.

이는 오버프로비저닝 및 최적화되지 않은 배포로 연결됩니다. 다음 섹션에는 초기 배포와 진행 중인 배포의 비용 최적화에 대한 기술 및 전략적 지침이 나와 있습니다.

일반적으로 서버리스 아키텍처에서는 일부 서비스(예: AWS Lambda)가 유휴 상태일 때 비용이 발생하지 않으므로 비용이 절감되는 경향이 있습니다. 하지만 특정 모범 사례를 따르고 절충을 통해 이러한 솔루션의 비용을 더욱 절감할 수 있습니다.

모범 사례

COST 1: 비용을 최적화하려면 어떻게 합니까?

비용 효율적인 리소스

올바른 리소스를 할당하는 측면에서, 서버리스 아키텍처는 관리가 더 쉽습니다. 서버리스는 가치당 지불 요금 모델을 따르고 수요에 따라 조정되므로 실제로 용량 계획 작업이 줄어듭니다.

운영 우수성과 성능 기반에서 설명한 바와 같이, 서버리스 애플리케이션의 최적화는 애플리케이션이 생성하는 가치와 비용에 직접적인 영향을 미칩니다.

Lambda 는 메모리를 기반으로 CPU, 네트워크 및 스토리지 IOPS 를 비례적으로 할당합니다. 100 밀리초 단위로 증분하므로 실행 속도가 빠를수록 비용이 줄어들고 함수가 창출하는 가치가 늘어납니다.

수요와 공급 일치

AWS 서버리스 아키텍처는 수요에 따라 조정되도록 설계되었기 때문에 따라야 할 사례가 없습니다.

지출 인식

AWS Well-Architected 프레임워크에 설명된 대로 클라우드에서는 향상된 유연성과 민첩성을 바탕으로 혁신을 주도하고 개발 및 배포를 가속화할 수 있습니다. 클라우드에서는 하드웨어 사양을 식별하고, 가격 견적을 협상하고, 주문 번호를 관리하고, 배송을 예약한 후 리소스 배포하기와 같은 온프레미스 인프라 프로비저닝과 관련된 수동 프로세스와 시간이 발생하지 않습니다.

서버리스 아키텍처가 확장되면 Lambda 함수, API, 단계 및 기타 자산의 수가 몇 배로 증가합니다. 비용 및 리소스 관리 측면에서 이러한 아키텍처의 대부분에 대한 예산을 책정하고 예측해야 하는데 여기서 태깅이 도움이 될 수 있습니다. AWS 청구서의 비용을 개별 함수 및 API 에 할당하고 AWS Cost Explorer 에서 프로젝트당 비용을 세부적으로 확인할 수 있습니다.

좋은 구현은 프로그래밍 방식으로 프로젝트에 속하는 자산에 대해 동일한 키-값 태그를 공유하고, 생성한 태그를 기반으로 사용자 지정 보고서를 생성하는 것입니다. 이 기능은 비용을 할당할 때는 물론이고 리소스가 속하는 프로젝트를 식별할 때에도 도움이 됩니다.

시간별 최적화

서버리스 애플리케이션에 적용되는 비용 최적화에 대한 **시간별 최적화** 영역의 모범 사례는 AWS Well-Architected 프레임워크 백서를 참조하십시오.

로깅 수집 및 저장

AWS Lambda 는 CloudWatch Logs 에 실행 출력을 저장하여 실행 문제를 식별하고 해결하고 서버리스 애플리케이션을 모니터링합니다. 이는 수집 및 저장이라는 두 가지 차원에서 CloudWatch Logs 서비스의 비용에 영향을 미칩니다.

그러므로 적절한 로깅 수준을 설정하고 불필요한 로깅 정보를 제거하여 로그 수집을 최적화해야 합니다. 환경 변수를 사용하면 DEBUG 모드에서 애플리케이션 로깅 수준 및 샘플 로깅을 제어하여 필요할 때 추가적인 통찰력을 확보할 수 있습니다.

신규 및 기존 CloudWatch Logs 그룹에 대한 로그 보존 기간을 설정하십시오. 로그 아카이브의 경우 요구 사항에 가장 적합한 비용 효율적인 스토리지 클래스를 내보내고 설정합니다.

직접 통합

Lambda 함수가 다른 AWS 서비스와 통합되는 동안 사용자 지정 로직을 수행하지 않는 경우 해당 함수가 불필요할 가능성이 있습니다.

API Gateway, AWS AppSync, Step Functions, EventBridge 및 Lambda 대상을 여러 서비스와 직접 통합할 경우 가치가 늘어나고 운영 오버헤드가 줄어들 수 있습니다.

대부분의 퍼블릭 서버리스 애플리케이션은 마이크로 서비스 시나리오에 설명된 것과 같이 제공된 계약의 구현과 관계없이 사용할 수 있는 API 를 제공합니다.

직접 통합이 더 적합한 예제 시나리오에는 REST API 를 통해 클릭스트림 데이터를 수집하는 시나리오가 있습니다.

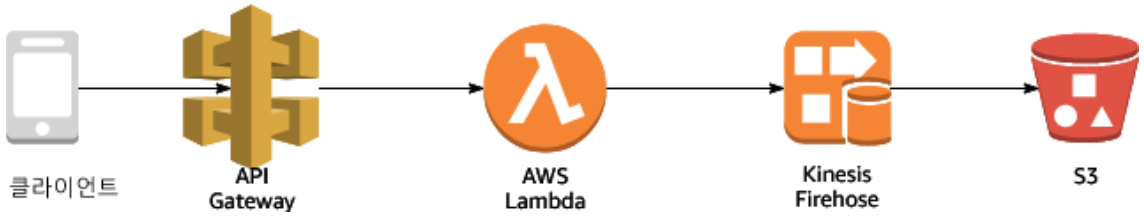

그림 30: Kinesis Data Firehose 를 사용하여 Amazon S3 로 데이터 전송

이 시나리오에서 API Gateway 는 수신되는 레코드를 Kinesis Data Firehose 로 수집한 후 S3 버킷에 저장하기 전에 레코드를 배치 처리하는 Lambda 함수를 실행합니다. 이 예제의 경우 추가 로직이 필요하지 않으므로 API Gateway 서비스 프록시를 사용하여 Kinesis Data Firehose 와 직접 통합할 수 있습니다.

그림 31: AWS 서비스 프록시를 구현하여 Amazon S3 로 전송되는 데이터의 전송 비용 절감

이 접근 방식을 사용하면 API Gateway 내에 AWS 서비스 프록시를 구현하여 Lambda 사용 비용과 불필요한 호출을 없앨 수 있습니다. 그러나 수집 속도를 충족하기 위해 여러 샤드를 사용해야 하는 경우 이로 인해 약간의 복잡성이 발생할 수 있습니다.

지연 시간에 민감한 경우 추상화, 계약 및 API 기능을 포기하는 대신 올바른 자격 증명을 마련하여 Kinesis Data Firehose 로 데이터를 직접 스트리밍할 수 있습니다.

그림 32: Kinesis Data Firehose SDK 를 사용한 직접 스트리밍을 통해 Amazon S3 로의 데이터 전송 비용 절감

VPC 또는 온프레미스 내에서 내부 리소스와 연결해야 하고 사용자 지정 로직이 필요하지 않은 시나리오의 경우 API Gateway 프라이빗 통합을 사용합니다.

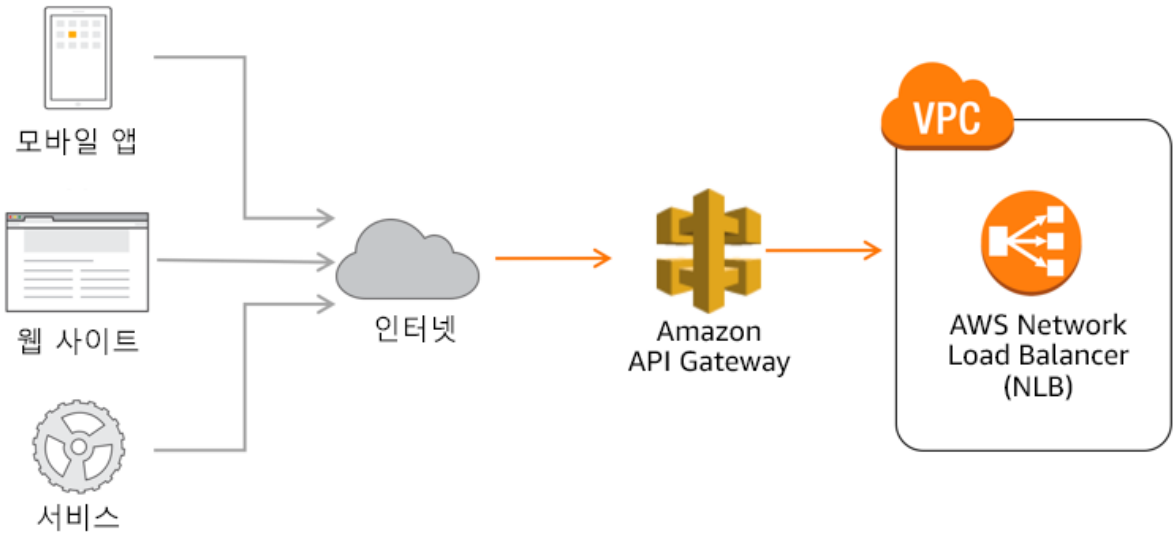

그림 33: VPC 의 Lambda 를 통해 Amazon API Gateway 프라이빗을 통합하여 프라이빗 리소스에 대한 액세스 제공

이 접근 방식을 사용하면 API Gateway 가 들어오는 각 요청을 VPC 에 있는 내부 Network Load Balancer 로 전송합니다. 이 내부 Network Load Balancer 는 IP 주소를 통해 동일한 VPC 또는 온프레미스에 있는 모든 백엔드로 트래픽을 전달할 수 있습니다.

이 접근 방식을 사용하는 경우 프라이빗 백엔드로 요청을 보내기 위한 추가 홉이 필요하지 않으므로 권한 부여, 조절 및 캐싱 메커니즘이라는 추가 이점을 제공하는 동시에 비용과 성능 측면에서도 유리합니다.

또 다른 시나리오는 Amazon SNS 가 모든 구독자에게 메시지를 브로드캐스트하는 팬아웃 패턴입니다. 이 접근 방식에서는 추가 애플리케이션 로직을 사용하여 불필요한 Lambda 호출을 필터링하고 방지해야 합니다.

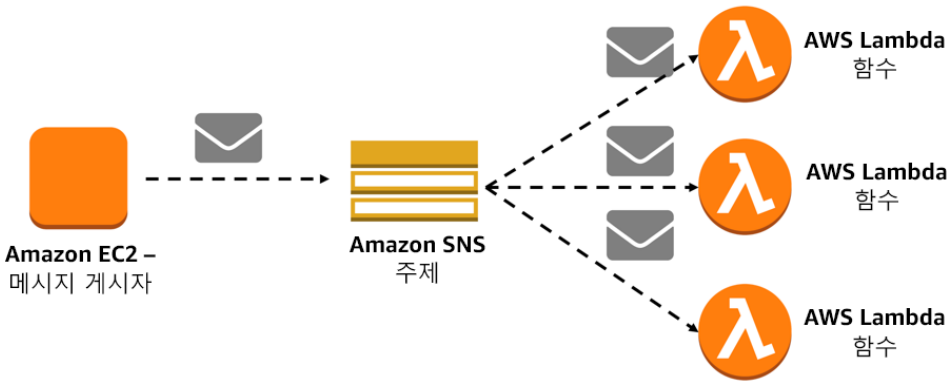

그림 34: 메시지 속성 필터링을 사용하지 않는 Amazon SNS

SNS 는 메시지 속성을 기반으로 이벤트를 필터링하고 올바른 구독자에게 메시지를 보다 효율적으로 전송할 수 있습니다.

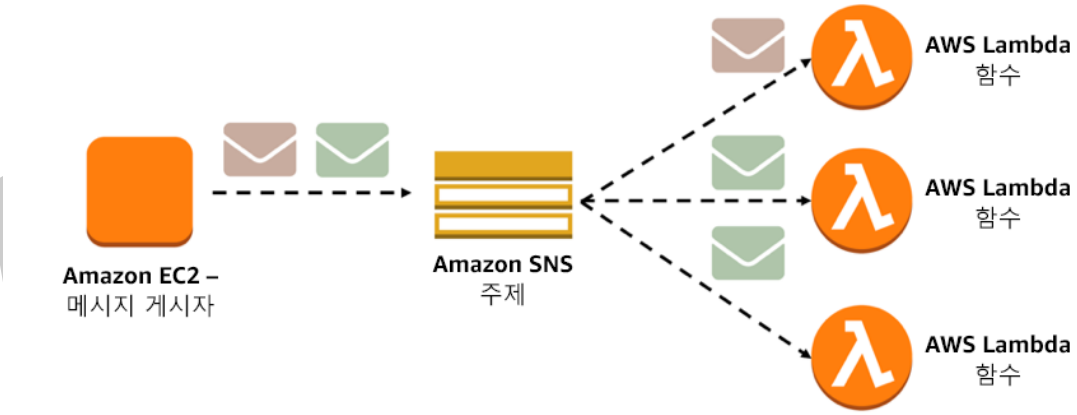

그림 35: 메시지 속성 필터링을 사용하는 Amazon SNS

또 다른 예로는 장기 실행 처리 작업이 있습니다. 이러한 작업의 경우 다음 단계로 이동하기 전에 작업이 완료될 때까지 기다려야 합니다. 이 대기 상태는 Lambda 코드 내에서 구현될 수 있지만, 이벤트를 사용하여 비동기 처리로 변환하거나 Step Functions 를 사용하여 대기 상태를 구현하는 것이 훨씬 효율적입니다.

예를 들어 다음 이미지는 AWS Batch 작업을 폴링하고 30 초마다 상태를 검토하여 작업이 완료되었는지 확인하는 작업을 보여줍니다. Lambda 함수 내에서 이 대기를 코딩하는 대신 폴링(*GetJobStatus)* + 대기(*Wait30Seconds)* + 결정자(*CheckJobStatus)*를 구현합니다.

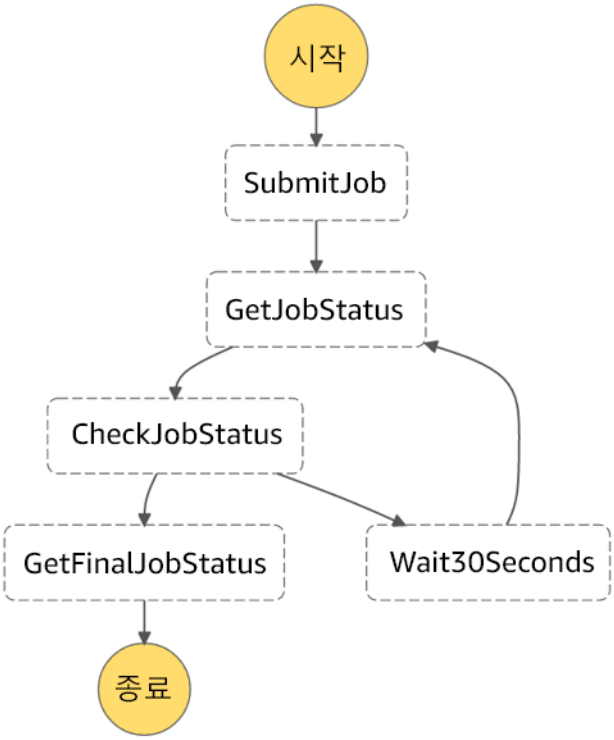

그림 36: AWS Step Functions 를 사용하여 대기 상태 구현

Step Functions 로 대기 상태를 구현하는 경우 Step Functions 의 요금 모델은 상태 내에서 소요된 시간이 아니라 상태 간 전환을 기반으로 요금을 부과하므로 추가 비용이 발생하지 않습니다.

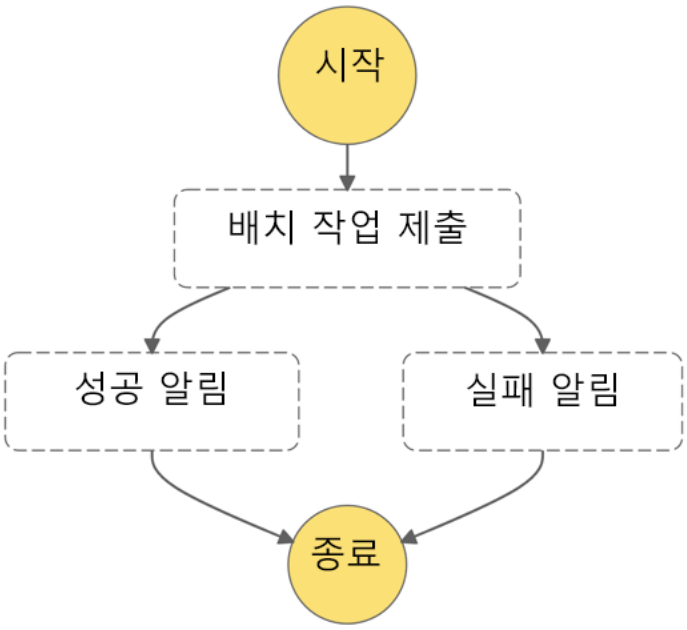

그림 37: Step Functions 서비스 통합 동기식 대기

대기해야 하는 통합에 따라 Step Functions 에서 동기식으로 대기한 후 다음 작업으로 이동하면 추가 전환을 줄일 수 있습니다.

비용 최적화

성능 기반에서 다룬 바와 같이 서버리스 애플리케이션을 최적화하면 각 실행에서 생성되는 가치를 효과적으로 개선할 수 있습니다.

글로벌 변수를 사용하여 데이터 스토어 또는 기타 서비스 및 리소스에 대한 연결을 유지하면 성능이 향상되고 실행 시간이 단축되어 비용도 절감됩니다. 자세한 내용은 성능 기반 섹션을 참조하십시오.

관리형 서비스 기능을 사용하여 각 실행의 가치를 개선할 수 있는 예로는 Amazon S3 에서 객체를 검색하고 필터링하는 경우입니다. Amazon S3 에서 대용량 객체를 가져오려면 Lambda 함수에 더 많은 메모리가 필요하기 때문입니다.

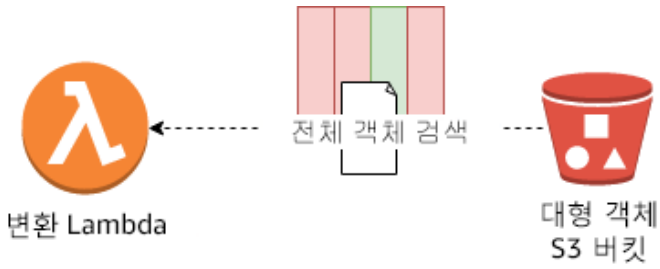

그림 38: 전체 S3 객체를 검색하는 Lambda 함수

이전 다이어그램을 보면 Amazon S3 에서 대용량 객체를 검색할 때 함수가 필요한 데이터를 변환하거나 반복하거나 수집해야 하므로 Lambda 의 메모리 소비와 실행 시간이 늘어날 수 있고, 때로는 이 정보의 일부만 필요하다는 것을 알 수 있습니다.

빨간색 열 3 개(필요하지 않은 데이터)와 녹색 열 1 개(필요한 데이터)가 이를 보여줍니다. Athena SQL 쿼리를 사용하여 실행에 필요한 세분화된 정보를 수집하면 실행의 검색 시간과 변환이 수행되는 객체 크기가 줄어듭니다.

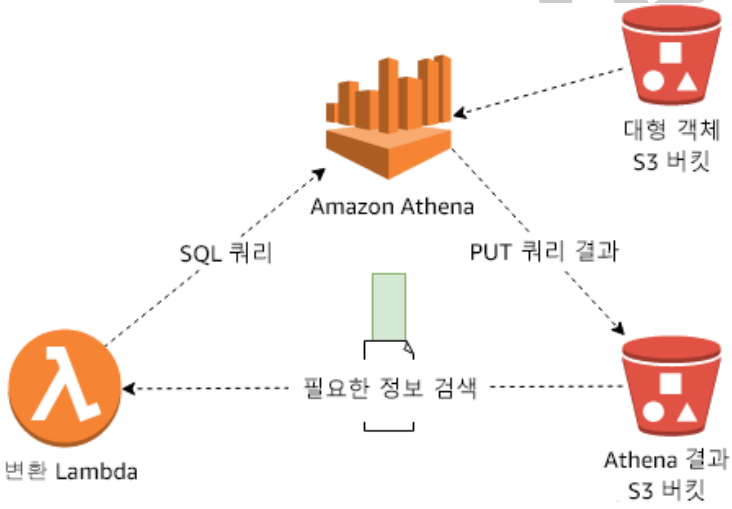

그림 39: Athena 객체 검색과 Lambda

다음 다이어그램을 보면 Athena 를 쿼리하여 특정 데이터를 가져올 때 검색된 객체의 크기가 줄어들며 추가 이점으로 Athena 가 쿼리 결과를 S3 버킷에 저장하고 결과가 Amazon S3 에 도달할 때 비동기식으로 Lambda 호출을 호출하므로 콘텐츠를 재사용할 수 있다는 것을 알 수 있습니다.

S3 Select 에서도 이와 유사한 접근 방식을 사용할 수 있습니다. S3 Select 를 사용하면 애플리케이션이 단순한 SQL 표현식을 사용하여 객체에서 데이터의 하위 집합만 검색할 수

있습니다. Athena 를 사용한 이전 예제와 같이 Amazon S3 에서 더 작은 객체를 검색하면 실행 시간과 Lambda 함수가 사용하는 메모리가 줄어듭니다.

200 초

```
# Download and process all keys
for key in src_keys:
  response =
s3_client.get_object(Bucket=src_bucket,
Key=key)
  contents = response['Body'].read()
  for line in contents.split('\n')[:-1]:
      line_count +=1
      try:
          data = line.split(',')
           srcIp = data[0][:8]
```

....

95 초

```
# Select IP Address and Keys
for key in src_keys:
  response =
s3_client.select_object_content
      (Bucket=src_bucket, Key=key,
expression =
       SELECT SUBSTR(obj._1, 1, 8),
obj._2 FROM s3object as obj)
  contents = response['Body'].read()
  for line in contents:
      line_count +=1
      try:
```

....

그림 40: Amazon S3 와 S3 Select 를 사용한 Lambda 성능 통계

리소스

비용 최적화 관련 AWS 모범 사례에 대해 자세히 알아보려면 다음 리소스를 참조하십시오.

설명서 및 블로그

- CloudWatch 로그 보존[64]
- Amazon S3 로 CloudWatch Logs 내보내기[65]
- Amazon ES 로 CloudWatch Logs 스트리밍[66]
- Step Functions 상태 시스템에서 대기 상태 정의[67]
- Step Functions 로 구동되는 Coca-Cola Vending Pass 상태 시스템[68]
- AWS 에서 처리량이 높은 게놈 배치 워크플로 구축[69]
- Amazon SNS 메시지 필터링으로 게시/구독 메시징 간소화
- S3 Select 및 Glacier Select

- MapReduce 의 Lambda 참조 아키텍처
- 서버리스 애플리케이션 리포지토리 앱 – CloudWatch Logs 그룹 보존 자동 설정
- 모든 서버리스 아키텍트가 알아야 할 10 가지 리소스

백서

- 서버리스 아키텍처로 엔터프라이즈 경제 최적화[70]

결론

서버리스 애플리케이션을 사용하면 차별화되지 않은 힘든 개발 관련 작업이 줄어들지만 여기에도 중요한 원칙은 여전히 적용됩니다.

안정성을 위해 정기적으로 장애 경로를 테스트하면 프로덕션에 도달하기 전에 오류를 포착할 가능성이 더 높아집니다. 성능 측면에서는 고객 기대치를 먼저 고려하여 역방향 설계를 수행하면 최적의 경험을 제공하도록 설계할 수 있습니다. 성능을 최적화하는 데 도움이 되는 여러 AWS 도구를 활용할 수도 있습니다.

비용 최적화를 위해 트래픽 수요에 따라 리소스의 크기를 조정하면 서버리스 애플리케이션 내에서 불필요한 낭비를 줄이고 애플리케이션을 최적화하여 가치를 높일 수 있습니다. 운영의 경우 아키텍처는 이벤트에 자동으로 대응하는 방향을 지향해야 합니다.

마지막으로 안전한 애플리케이션은 조직의 중요한 정보 자산을 보호하고 모든 계층에서 규정 준수 요구 사항을 충족합니다.

서버리스 애플리케이션 환경은 도구 및 프로세스의 확장 및 성숙을 통해 지속적으로 진화하고 있습니다. 이에 AWS 는 서버리스 애플리케이션의 올바른 아키텍처 설계를 위해 이 백서를 계속해서 업데이트할 것입니다.

기고자

다음은 본 문서 작성에 도움을 준 개인 및 조직입니다.

- Adam Westrich: 선임 솔루션스 아키텍트, Amazon Web Services
- Mark Bunch: 엔터프라이즈 솔루션스 아키텍트, Amazon Web Services

- Ignacio Garcia Alonso: 솔루션스 아키텍트, Amazon Web Services
- Heitor Lessa: Well-Architected 부문 수석 서버리스 리드, Amazon Web Services
- Philip Fitzsimons: Well-Architected 부문 선임 관리자, Amazon Web Services
- Dave Walker: 수석 전문 솔루션스 아키텍트, Amazon Web Services
- Richard Threlkeld: 모바일 부문 선임 제품 관리자, Amazon Web Services
- Julian Hambleton-Jones: 선임 솔루션스 아키텍트, Amazon Web Services

추가 자료

추가 정보는 다음을 참조하십시오.

- AWS Well-Architected 프레임워크[71]

문서 개정

날짜	설명
2019 년 12 월	새로운 기능 및 모범 사례 변경에 대한 전체 업데이트.
2018 년 11 월	Alexa 및 모바일에 대한 새로운 시나리오와 새로운 기능 및 모범 사례 변경을 반영하는 전체 업데이트.
2017 년 11 월	최초 게시.

Notes

[1] https://aws.amazon.com/well-architected

[2] http://d0.awsstatic.com/whitepapers/architecture/AWS_Well-Architected_Framework.pdf

[3] https://github.com/alexcasalboni/aws-lambda-power-tuning

[4] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/BestPractices.html

[5] http://docs.aws.amazon.com/elasticsearch-service/latest/developerguide/es-managedomains.html

[6] https://www.elastic.co/guide/en/elasticsearch/guide/current/scale.html

[7] http://docs.aws.amazon.com/streams/latest/dev/kinesis-record-processor-scaling.html

[8] https://d0.awsstatic.com/whitepapers/whitepaper-streaming-data-solutions-on-aws-with-amazon-kinesis.pdf

[9] http://docs.aws.amazon.com/kinesis/latest/APIReference/API_PutRecords.html

[10] http://docs.aws.amazon.com/streams/latest/dev/kinesis-record-processor-duplicates.html

[11] http://docs.aws.amazon.com/lambda/latest/dg/best-practices.html#stream-events

[12] http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-api-usage-plans.html

[13] http://docs.aws.amazon.com/apigateway/latest/developerguide/stage-variables.html

[14] http://docs.aws.amazon.com/lambda/latest/dg/env_variables.html

[15] https://github.com/awslabs/serverless-application-model

[16] https://aws.amazon.com/blogs/aws/latency-distribution-graph-in-aws-x-ray/

[17] http://docs.aws.amazon.com/lambda/latest/dg/lambda-x-ray.html

[18] http://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-paramstore.html

[19] https://aws.amazon.com/blogs/compute/continuous-deployment-for-serverless-applications/

[20] https://github.com/awslabs/aws-serverless-samfarm

[21] https://d0.awsstatic.com/whitepapers/DevOps/practicing-continuous-integration-continuous-delivery-on-AWS.pdf

[22] https://aws.amazon.com/serverless/developer-tools/

[23] http://docs.aws.amazon.com/lambda/latest/dg/with-s3-example-create-iam-role.html

[24] http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-method-request-validation.html

[25] http://docs.aws.amazon.com/apigateway/latest/developerguide/use-custom-authorizer.html

[26] https://aws.amazon.com/blogs/compute/secure-api-access-with-amazon-cognito-federated-identities-amazon-cognito-user-pools-and-amazon-api-gateway/

[27] http://docs.aws.amazon.com/lambda/latest/dg/vpc.html

[28] https://aws.amazon.com/pt/articles/using-squid-proxy-instances-for-web-service-access-in-amazon-vpc-another-example-with-aws-codedeploy-and-amazon-cloudwatch/

[29] https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf

[30] https://d0.awsstatic.com/whitepapers/Security/AWS_Security_Best_Practices.pdf

[31] https://www.twistlock.com/products/serverless-security/

[32] https://snyk.io/

[33] https://www.owasp.org/index.php/OWASP_Dependency_Check

[34] http://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions/

[35] http://docs.aws.amazon.com/lambda/latest/dg/limits.html

[36] http://docs.aws.amazon.com/apigateway/latest/developerguide/limits.html#api-gateway-limits

[37] http://docs.aws.amazon.com/streams/latest/dev/service-sizes-and-limits.html

[38] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html

[39] http://docs.aws.amazon.com/step-functions/latest/dg/limits.html

[40] https://aws.amazon.com/blogs/compute/error-handling-patterns-in-amazon-api-gateway-and-aws-lambda/

[41] https://aws.amazon.com/blogs/compute/serverless-testing-with-aws-lambda/

[42] http://docs.aws.amazon.com/lambda/latest/dg/monitoring-functions-logs.html

[43] http://docs.aws.amazon.com/lambda/latest/dg/versioning-aliases.html

[44] http://docs.aws.amazon.com/apigateway/latest/developerguide/stages.html

[45] http://docs.aws.amazon.com/general/latest/gr/api-retries.html

[46] http://docs.aws.amazon.com/step-functions/latest/dg/tutorial-handling-error-conditions.html#using-state-machine-error-conditions-step-4

[47] http://docs.aws.amazon.com/xray/latest/devguide/xray-services-lambda.html

[48] http://docs.aws.amazon.com/lambda/latest/dg/dlq.html

[49] https://aws.amazon.com/blogs/compute/error-handling-patterns-in-amazon-api-gateway-and-aws-lambda/

[50] http://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-wait-state.html

[51] http://microservices.io/patterns/data/saga.html

[52] http://theburningmonk.com/2017/07/applying-the-saga-pattern-with-aws-lambda-and-step-functions/

[53] https://d0.awsstatic.com/whitepapers/microservices-on-aws.pdf

[54] http://docs.aws.amazon.com/lambda/latest/dg/best-practices.html

[55] https://aws.amazon.com/lambda/faqs/

[56] http://docs.aws.amazon.com/lambda/latest/dg/best-practices.html

[57] http://docs.aws.amazon.com/lambda/latest/dg/lambda-introduction.html

[58] https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/

[59] http://docs.aws.amazon.com/lambda/latest/dg/vpc.html

[60] http://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-caching.html

[61] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html

[62] https://aws.amazon.com/dynamodb/dax/

[63] http://docs.aws.amazon.com/streams/latest/dev/amazon-kinesis-streams.html

[64] http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/SettingLogRetention.html

[65] http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/S3ExportTasksConsole.html

[66] http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/CWL_ES_Stream.html

[67] http://docs.aws.amazon.com/step-functions/latest/dg/amazon-states-language-wait-state.html

[68] https://aws.amazon.com/blogs/aws/things-go-better-with-step-functions/

[69] https://aws.amazon.com/blogs/compute/building-high-throughput-genomics-batch-workflows-on-aws-workflow-layer-part-4-of-4/

[70] https://d0.awsstatic.com/whitepapers/optimizing-enterprise-economics-serverless-architectures.pdf

[71] https://aws.amazon.com/well-architected