AWS

클라우드 개발 가이드

클라우드 환경 아키텍처 목표

클라우드 환경 특징

어플리케이션 서버 분산 : 저용량 서버로 부하 분산 후, 필요 시 scale-out하는 구성

DB 서버 분산 : write/read db 분리, 업무별 db 분리 구성

어플리케이션 분산 : 비즈니스 요구사항을 신속하게 구현/배포하기 위해 잘게 나누어 독립적으로 배포/실행

모든 자원은 필요에 따라 쉽게 만들고 폐기하고 모니터링할 수 있어야 하며 이게 자동화되어야 한다. ex) EC2에 oracle을 설치하는 게 아니라 RDS 오라클을 사용해야 managed service임

간단 구조

Elastic load balancing – web werver(s) – elastic load balancing – 업무x was server(s) – 업무 x DB

클라우드 환경 아키텍처 유형

1) 모노리틱 : 어플리케이션/디비를 하나로 구성. Was에 오토스케일링은 적용.

Lift & shift 유형

인프라만 전환하고 어플리케이션 및 소스 변경 최소화

오토스케일링을 위해 웹/와스 모두 elb로 부하 분산 구성

2) Read 분산 : read 부하 분산용 read-replica db 분리, 어플리케이션도 read 프로그램 구분

Refactoring 유형

쓰기 처리용 마스터 디비와 읽기 처리용 read-replica로 구성

was에서 멀티 데이터 소스 설정

쓰기용 읽기용 프로그램 분리로 어플리케이션 수정 일부 발생

3) 업무별 분산 : 디비와 어플리케이션을 업무별로 분산

Refactoring 유형

오토스케일링은 업무별로 그룹화된 단위로 가능

업무 그룹별로 어플리케이션/와스/디비 분리

서비스간 호출은 리모트 콜이나 큐 방식 연계 등으로 변경

4) 샤딩 : 한 업무에 대한 write 부하 분산을 위해 디비 분산

메인 디비와 별개로 쓰기 부하 분산을 위해 샤딩 디비 구성, 샤딩 키에 따라 데이터 분산

샤딩 키에 따라 디비 접속하므로 멀티 데이터 소스 설정 및 라우팅 필요

샤딩 키에 따라 디비에 분산된 데이터를 읽어서 어그리게이션 하는 기능 구현 필요할 수 있음

아키텍처 설계 및 구현

서비스 분산

클라우드 기반 분산 어플리케이션은 여러 서버에 나누어진 서비스를 어떻게 호출하고 요청을 배분할 것인가에 따라 3가지 방안으로 구분됨

Centralized : 클라우드 환경에 부적합

Decentralized : ELB/HA proxy 방식에 대응

Distributed : API G/W 또는 오픈소스 서비스 디스커버리 방식에 대응

1) 서비스 디스커버리 방식

여러 서버에 나누어진 서비스를 유레카나 아마존 api gateway 같은 서비스를 통해 찾아내서 호출하는 방식

2) 서비스 분배

Elb나 ha proxy를 활용하여 port나 uri를 이용하여 분배하는 방식

Front end에서 rest api로 elb를 통해 서비스 호출 - elb에서는 uri로 구분하여 타겟 노드로 부하분산

서비스 연계

서비스와 디비를 분산하면 피호출 서비스가 다른 노드에 있으므로 분산된 서비스 간 연계방법이 필요, aws는 rest api로 호출 또는 message queue를 이용한 비동기 방식 권고

1) 분산 서비스 호출 : 개별 서비스 업그레이드 독립적, 업무적으로 상태 관리 필요, rest/soa 같은 표준화된 프로토콜 활용

Rest 연계 - Gateway 서버가 front/end point를 구성하고 요청이 들어오면 backend service에 배분

2) 메시지 큐 : 개별 서비스 가용성과 상관없는 loosely-coupled 형태, 메시지 큐에 대한 모니터링 및 관리 필요

세션 처리

분산 시스템을 위해서 어플리케이션은 stateless하게 설계해야 함

세션 정보 사용을 최소화하고 필요 정보는 요청시마다 담아 보내도록 설계

Aws는 세션 공유가 필요할 경우 서비스 dynamodb나 elasticache 사용을 권장

외부 저장소를 통해 세션 정보를 공유할 경우 2가지 방법

. 서블릿 스펙에 httpsession을 override한 기능 사용 – dynamodb, redis spring session

. 외부 저장소에 세션 정보를 저장/조회 기능 구현 : redis 클라이언트 활용

정적 컨텐츠 처리 방식

아마존 스토리지 서비스 s3를 활용하여 사용자가 웹 서버가 아닌 s3에서 정적 컨텐츠 다운

멀티 인스턴스간 파일 공유에도 사용

웹 서버 / ELB 연동

Ec2인스턴스의 오토스케일링이 가능하며 이를 위해 ELB 서비스를 이용

WEB server(EC2) ⇔ ELB ⇔ WAS(EC2)와 같이 구성

빌드/배포

Codecommit, codebuild, codedeploy, codepipeline 서비스를 이용하여 빌드/배포 자동화 구성

개발자가 소스를 codecommit에 push

소스 push로 codepipeline이 trigger되어 소스를 가져와 빌드하고 결과물을 s3 bucket에 저장

Lambda로 s3 bucket의 빌드 결과물을 서울 리전 s3 bucket에 복사

Lambda로 codedeploy 수행, codedeploy가 ec2 인스턴스에 배포

Codecommit : 프라이빗 git repository 기능 제공

Codebuild : 소스 코드를 컴파일하고 테스트 실행, 배포 준비가 된 소프트웨어 패키지를 생성하는 서비스

Codedeploy : 코드 배포를 자동화하는 서비스, 배포 대상 인스턴스에 에이전트 설치/실행 필요

Codepipeline : 빌드 자동화 및 CI/CD 프로세스 구성

클라우드 아키텍처 가이드

클라우드 아키텍처 특징

모든 리소스는 프로그래밍 가능한 형태로 존재 : 서버/디비/스토리지 등이 논리적 형태로 존재하고, 이를 이용하여 환경 구성

글로벌/가용/무제한 : 유연하게 확장 가능한 형태로 구성, 자원의 수평/수직 확장/축소 가능

높은 수준의 관리 서비스 : 모든 리소스는 클라우드 솔루션 업체의 관리 하에 존재, 사용자는 리소스 모니터링/관리 등을 위한 솔루션과 인력 비용 절감 가능

보안 기능 내장 : 한 물리 서버에 다수가 가상 환경을 구성, 사용하므로 보안 관련 기능이 강화되어 있고 기본 서비스로 제공

클라우드 아키텍처 구조

4개 레이어로 구분되고, 어디까지 클라우드 업체에서 서비스를 받느냐에 따라 클라우드 서비스 모델이 결정됨

Physical layer – 물리적 서버/스토리지/네트워크 등 시스템 인프라가 존재하는 데이터 센터

Logical layer – 물리적 시스템 인프라를 사용자에게 할당할 수 있게 논리적 분리 구성한 상태

Platform layer – 논리적 분리된 시스템 인프라 위에 was/dbms 같은 사용자 어플리케이션을 실행할 수 있는 주요 엔진 포함, 클라우드 업체 서비스 이용 시 개발에 필요한 sdk/runtime 등 포함

Service layer – 클라우드 업체가 제공하는 ready-made된 어플리케이션, 사용자가 직접 설정을 통해 사용

클라우드 서비스 모델

클라우드 제공자의 관리 영역에 따라 IaaS, PaaS, SaaS로 구분

IaaS – 시스템 인프라를 서비스로 제공, 구축/유지비용 없고 사용한 만큼 지불

PaaS – IaaS의 시스템 인프라 위에 sw 개발을 위한 개발 환경, 이를 이용한 서비스 환경 제공. 사용자는 os/dbms/was 및 배포 환경을 이용한 자신의 어플리케이션 개발하고 서비스 가능

SaaS – 사용자가 필요 서비스를 바로 선택하여 사용할 수 있는 ready-made 어플리케이션 제공 (전체 layer)

클라우드 아키텍처 전환 모델

Rehost – 기존 환경 중 하드웨어만 교체, 기존 어플리케이션에 변경 거의 없음

Replace (repurchase) – 사용자 데이터를 제외하고 클라우드 환경, 솔루션으로 대체

Refactor – rehost, replace의 중간 개념, 필요한 솔루션을 적극 활용, 클라우드 솔루션에 맞게 코드 수정 및 사용자 데이터의 migration/conversion 등이 필수 (코드 변경 정도에 따라 revise – 일부 개정, rebuild – 전면 재개발로 구분)

Aws 설계 원칙

필요 용량에 대한 추측 중단

운영환경 규모의 테스트 시스템

아키텍처 변경에 따르는 위험 감소

아키텍처 실험을 간편하게 할 수 있는 자동화

아키텍처의 지속적인 혁신

클라우드 아키텍처 설계 시 고려사항

서비스 설계 우선 – 개별 서비스 연계 방안, 디비 분산 설계 등 우선 수행

데이터 흐름 기반 – 데이터 흐름을 기반으로 고가용성/분산 고려하여 아키텍처 구성요소 설계

분산 데이터베이스 – 디비 서버, 디비 인스턴스 설계

업무 간 인터페이스 – 업무 서버와 인터페이스 대상 서버가 증가/축소될 수 있으므로 큐 또는 api 방식을 이용하고 루즈하게 연결 설계

제도/법 – 사용자가 보유한 센터/서버에 환경 구성 필요

솔루션 업체 서비스 이용 – 요구사항 중 솔루션 업체 서비스를 이용 가능한 범위 명확화

보안 설계 – 사용자를 역할로 구분하고 역할별 수행 가능 행위 정의, 아키텍처 설계

리소스 용량 산정 – 온프리미즈와 동일하게 수행 후 테스트 시 용량 검증 및 비용 산정

수평적 용량 확장 – 서버 수의 확장/축소가 가능하므로 이를 기반으로 서버의 용량 산정 및 부하 분산 고려, 설계. 수직적 확장은 서비스 정지 후 가능하므로 가급적 지양

고정된 사용량 기반 – 업무 증감에 민감하지 않은 메모리 같은 리소스 사용량을 기반으로 서버 설계 (cpu, network 는 업무 증감에 따라 변경이 잦음)

통합 아키텍처 설계

확장에 대한 설계

서버 인스턴스 그룹화 (업무별) / 로드 밸런서 구성 / stateless 구성 / scale-in 구성 (인스턴스 제거 시 종료 처리)

느슨한 결합 구조 아키텍처 고려사항

수평 확장 가능하도록 설계, 상태 정보를 어플리케이션에 개별적으로 저장하지 않음 – 어플리케이션 서버 노드 수가 변해도 기능에 영향 없음

상태 정보는 별도 상태 정보 저장소에 구성

메시지 순서 보장, 메시지 중복 처리 대응, 메시지 반환, 일괄 처리, 대기열 구성

자동화 설계 고려사항

어플리케이션 오류에 대한 처리, 자동화 구성에 대한 검증

장애 발생에 대한 설계 고려사항

어플리케이션 상태 정보는 외부에 저장,

비동기식 분산 처리로 특정 노드의 장애가 전체 노드로 전이되는 것 방지,

로그는 항상 중앙 집중된 장소에 저장,

로그에 오류 추적에 필요한 정보 추가 저장,

SPoF 제거 (서버 노드를 여러 리전에 분산 구성, failover 설계 수행하여 업무 중단 가능성 최소화)

다중처리 설계 고려사항

병렬 처리 구성 – 빅데이터 같이 긴 시간 필요한 작업은 다수 노드에 분산하여 수행하도록 구성

성능 관려 설계 고려사항

데이터 읽기/쓰기 분리 구성

메모리 기반 아키텍처 구성

정적 데이터 처리 – html, image 등은 별도 서비스로 네트워크 입구 가까이에 구성하여 이 데이터를 위해 서버 인스턴스까지 요청이 도달하지 않도록 구성

정합성에 대한 설계 고려사항

데이터 파티셔닝 – 한 데이터스키마에 모두 저장하지 않게 하여, 한 디비에서 수행되는 데이터 저장이 다른 데이터 스키마에 미치는 영향 최소화

어플리케이션을 이용한 일관성 유지 – 별도 트랜잭션 실패 시 이를 논리적으로 롤백하기 위한 보상 트랜잭션 패턴 설계, 구성

참조 아키텍처 패턴

1) N-Tier

온프리미스와 같이 프레젠테이션 계층, 비즈니스 계층, 데이터 계층으로 구분 또는 그 이상

각 계층은 별도 시스템에서 실행되고 계층 간 호출은 직접 수행 또는 비동기 메시징 사용

한 계층에 하나 이상의 기능 구성 가능 – 대기 시간은 감소하나 확장성, 유연성은 낮아짐

2) Queue-worker

요청처리 front-end와 업무 처리 worker로 구성하고 이들 간 통신은 큐를 이용하여 비동기 방식으로 통신 수행.

3) microservice

전체 비즈니스를 단독 실행 가능하고 독립적 배치가 가능한 작은 단위로 기능을 분해하여 서비스로 구성

한 업무 처리가 다수의 서비스들 조합으로 수행

Api 호출을 통하여 상호간 연계 수행, service discovery와 api gateway에 의해 모든 서비스들의 엔드포인트를 단외하여 묶어주고 api에 대한 인증 및 인가 기능, 메시지에 따른 라우팅 기능을 수행

개별 서비스는 팀간 조정 없이 잦은 업데이트 수행 가능

n-tier, queue기반보다 구축 관리가 복잡하고, 업무 요건을 서비스 형태로 개발하기 위한 개발 능력 필요

업무 요건에 대한 개발, 테스트, 실적용이 신속함

4) CQRS

Command and query responsibility segregation

디비에 대한 읽기 쓰기 작업을 별도의 작업으로 구분

쿼리에 대한 독립적인 최적화 수행 가능

대규모 사용자, 업무가 집중되는 데이터 계층에 적절

Azure

서비스 유형

Virtual machines – os, 미들웨어 등을 자유롭게 설치 사용하는 유형

Cloud services – vm에 원하는 소프트웨어를 설치하여 실행 가능, 편의성은 낮음

App service – 다양한 언어로 어플리케이션 작성 가능,

설계 원칙

확장성 (업무 증가 대응) / 가용성(서비스 정상 수행) / 탄력성(장애 시 신속 복구) / 관리(어플리케이션 유지/관리) / 보안 (전체 어플리케이션 lifecycle에 대한 보안)

장애에 대한 자동 조치

모든 구성요소의 다중화

시스템 확장을 고려한 어플리케이션 설계

수평적 확장 고려

작은 단위의 파티셔닝

운영 고려 (모니터링, 감사/추척 가능)

Managed service 사용

업무 성격에 따른 데이터 저장소

업무 요건의 신속한 반영

업무 요구사항을 반영한 아키텍처

DMA (Data Migration Assistant)

데이터 이행 방안 – 동종 DBMS 간 이관만 지원

ADF (Azure Data Factory)

클라우드에서 데이터 이동 및 변환을 오케스트레이션 하고 자동화 하기 위해 파이프라인을 구성하여 데이터 기반 워크플로우를 만들 수 있는 데이터 통합 서비스

서로 다른 데이터 저장소에서 데이터를 수집하는 워크플로우(파이프라인)를 만들고, Hadoop/spark/ machine learning 같은 서비스로 처리/번환 후 데이터 저장소에 출력 데이터를 저장

온프레미스, 클라우드에 둘 다 접근하는 시나리오에서 지속적 데이터 마이그레이션 필요한 경우

데이터를 트랜잭션 처리/수정할 때 또는 마이그레이션 중 비즈니스 논리를 추가할 때

스토리지 서비스

Blob : object storage, 구조화되지 않은 데이터 (block / page / append로 구분)

File : 표준 SMB 프로토콜 사용, rest api로 온프레미스 환경에서도 접근 가능

Queue : 프로그램 간 통신을 위한 비동기 메시징 제공

Table : 개발 및 대량 데이터에 빠른 접근이 가능한 NoSQL key-value 저장소

GCP

리소스 계층 구조 : 조직 -> 폴더 -> 프로젝트 -> 리소스

접근 방법 : 웹 콘솔, CLI (shell, sdk), 모바일 앱, rest api

EKS (Elastic Kubernetes Service)

개요

쿠버네티스를 사용하여 컨테이너식 어플리케이션을 쉽게 배포, 관리, 확장할 수 있는 서비스

아키텍처는 CI/CD pipeline, Control plane, data plane으로 구성

Control plane – 컨테이너 클러스터 마스터 역할을 수행하는 곳, ECS/EKS가 있음

Data plane – 실제 컨테이너가 배포되는 곳, fargate/EC가 있음

CI/CD pipeline – codepipeline 사용하여 구성 가능

Registry – ecr 사용 가능

컨테이너를 실행할 수 있는 서비스 비교

EKS	ECS	Beanstalk
쿠버네티스를 사용하는 컨테이너식 어플리케이션 관리형 환경	클러스터에서 도커 컨테이너를 실행, 중지, 관리해주는 컨테이너 관리 서비스	Aws에서 가장 쉽게 웹 어플리케이션, 웹 서비스를 배포, 확장, 관리할 수 있는 완전 관리형 서비스
쿠버네티스 마스터를 여러 가용 영역에 자동배포하여 가용성이 높은 아키텍처 제공	MSA 모델이 정교한 어플리케이션 아키텍처 구축 가능	어플리케이션 업로드만 하면 용량 프로비저닝, 부하분산, 조정, 모니터링, 배포 등을 자동으로 처리
모든 언어 지원	모든 언어 지원	언어 제약 있음
모든 버전 지원	모든 버전 지원	언어별 지원 버전 고정
Docker container 실행	Docker container 실행	소스 빌드 후 사전 정의된 실행환경에 배포, 실행
Ec2 node 확장 방식으로 클러스터 자동 확장	Ecs 콘솔에서 cloudwatch metric으로 정책 설정	Ec2 node 확장 방식으로 클러스터 자동 확장
배포 환경 – ec2, fargate 예정	Ec2, fargate	Ec2
작업자 노드 클러스터를 프로비저닝, 쿠버네티스 제어 플레인의 프로비저닝, 확장 및 관리	서비스 스케일링, 애플리케이션 로드 밸런서, 서비스 디스커버리, aws batch, fargate 지원	Ec2 인스턴스, 용량 프로비저닝, 로드 밸런싱, 오토스케일링, 모니터링 등 자동 처리

Amazon codepipeline

빠르고 안정적인 어플리케이션 및 인프라 업데이트를 위해 릴리즈 파이프라인을 자동화하는 완전 관리형 지속적 전달 서비스

코드 변경 발생 시 사용자가 정의한 릴리즈 모델을 기반으로 릴리즈 프로세스의 빌드, 테스트, 뱊포를 자동화

깃허브, 사용자 지정 플러그인과 손쉽게 통합 가능

Codecommit, codebuild, codedeploy 서비스와 결합하여 사용 가능

eks는 codecommit, codebuild와 함께 구성 가능

eks 모니터링

eks 서비스는 cloudwath를 사용하여 일부 항목을 제외하고는 못하므로 Kubernetes native dashboard를 설치하여 모니터링 권장

AKS (Azure Kubernetes Service)

애저 클라우드 플랫폼 기반에 컨테이너식 어플리케이션 배포를 위한 관리형 환경

쿠버네티스를 사용하여 컨테이너 배포를 오케스트레이션 하는 방식으로 컨테이너 복제 및 관리 가능한 클러스터 구성

애저의 컨테이너 서비스 종류 5가지

AKS

완벽한 관리형 쿠버네티스 환경 제공

컨테이너 오케스트레이션 지식 없이 컨테이너화된 응용 프로그램을 쉽게 배포 관리하는 클러스터 환경 제공

온라인으로 클러스터 리소스 프로비전, 업그레이드, 조정 가능

ACI (Azure Container Instances)

클라우드 응용 프로그램을 패키지, 배포, 관리하는 기본 환경 제공

가상 머신을 직접 관리하지 않고 애저에서 컨테이너를 실행하는 가장 빠르고 간단한 방법 제공

간단하게 격리된 컨테이너 실행 환경이 필요한 경우에 적합

ACS (Azure Container Services)

애저의 포털, cli, api 기능이 포함된 sla 기반 서비스

표준 컨테이너 오케스트레이션 도구(swram, k8s, dc/os)를 선택, 클러스터를 신속 구현 및 관리

모든 수준의 클러스터 구성을 사용자가 변경, 사용할 수 있는 프로젝트, 따라서 sla는 제공되지 않으며 운영자가 직접 모든 서비스 관리 가능 이 서비스는 종료 예정

Service Fabric Clusters

마이크로 서비스 및 컨테이너를 관리하도록 배포된 시스템 플랫폼

클라우드 네이티브 어플리케이션 개발 및 인프라 관리를 간편하게 제공

컨테이너에서 실행되는 엔터프라이즈급 레벨1 클라우드 규모의 응용 프로그램 빌드 관리를 위한 차세대 플랫폼 서비스

App Services

Windows/linux에서 사용 가능한 azure web app을 통해 개발자가 인프라 걱정 없이 엔터프라이즈

급 웹 응용 프로그램을 쉽게 배포 확장 가능

Web app for containers를 통해 개발자는 docker 형식의 컨테이너 이미지를 가져와 azure에서 대규모로 쉽게 배포, 실행할 수 있음.

Azure functions는 기존 azure app service 플랫폼을 확장하는, 서버를 사용하지 않는 이벤트 구동 환경

GKE (Google Kubernetes Engine)

개요

GCP 기반에 컨테이너식 어플리케이션 배포를 위한 관리형 환경

K8s 명령 및 리소스를 이용하여 응용 프로그램 배포 관리하고 자동 배포를 위한 정책 설정과 어플리케이션 상태 모니터링 서비스를 제공

CI/CD 구성 – spinnaker

넷플릭스에서 오픈소스화한 멀티 클라우드를 지원하는 배포 자동화 솔루션

오케스트레이션 파이프라인 구조 지원

배포 단계를 여러 스텝으로 나눠 수행하고 각 워크플로우에 대한 관리 필요

State / steps / tasks / operation이라는 개념으로 하위 태스크 구현

개발자가 repository에 commit -> container builder가 trigger -> 해당 이미지 빌드 및 테스트 완료되면 push -> spinnaker가 deploy 수행, 테스트 수행, 사용자에게 배포 승인 받음 -> 최종 프로덕션 서버에 배포 완료

MSA 구성을 위한 Istio 활용

Istio는 envoy proxy를 제어하는 서비스 (envoy – L7 layer proxy, service discovery, health check 기능 수행)

Service a가 proxy a를 통해 proxy b 호출 (config를 통해 proxy a는 proxy b를 안다 – service discovery?)

Proxy b는 policy 확인 후 service b에게 전달

Service b는 proxy b에 결과 전달하면 proxy b는 proxy a에게 전달하고 모니터링 정보 남김

Proxy a는 service a에게 응답 전달하고 모니터링 정보 남김

컨테이너를 실행할 수 있는 서비스 비교

GKE	GAE (App Engine) - flexible	GAE – standard
K8s 기반 컨테이너식 어플리케이션 관리 환경	완전 관리형 서버리스 어플리케이션 플랫폼	

컨테이너 복제, 모니터링, 복구로 가용성을 높임 리소스 확장/축소 가능	개발자는 서버 관리와 구성 배포를 고민하지 않고 어플리케이션을 개발하고 사용한 리소스 비용만 지불 어플리케이션 확장, 축소, 패치 등 인프라 작업은 구글이 함	
모든 언어	모든 언어	제약 있음
모든 버전	모든 버전	언어별 지원 버전 고정
컨테이너 실행	컨테이너 실행	Vm 형태
사용자가 만든 도커 이미지 실행	플랫폼이 소스 기반으로 생성한 도커 이미지, 또는 사용자가 만든 도커 이미지	컨테이너 기본 설정은 제공, 사용자가 환경변수를 통해 제한적 변경 가능
미지원	웹 취약점 점검 자동	웹 취약점 점검 자동
수동 클러스터 확장	부하에 따라 자동 서버 확장	부하에 따라 자동 서버 확장
컨테이너 오케스트레이션에 대해 통제 가능	로깅, 스케일링, 트래픽 분산 등을 자동 처리	빠른 스케일 업/다운 가능

Serverless computing

아키텍처 패러다임 변화

서버 -> 가상화 -> 컨테이너

모놀리틱 -> 마이크로 서비스

사용자 요청의 응답 대기 구조 -> 이벤트 드리븐 방식

어플리케이션 개발 및 운영의 단순화 => 서버리스 컴퓨팅

서버리스 컴퓨팅은 프로그래밍 언어 런타임 가상화 기술로, 서버 인프라와 운영체제 설정 등의 관리 요소는 클라우드 서비스 업체가 하고, 사용자는 코드에만 집중하여 생산성 증가, 시스템 관리 비용 최소화

초당 수 천 개 요청까지 자동 확장, 사용한 시간에 대해서만 비용 발생

항목	On-premise	Virtual machine	container	Serverless computing
배포 시간	주	분	초	밀리초
관리	모두	OS 미들웨어 소스	미들웨어 소스	소스
확장	매뉴얼	매뉴얼	매뉴얼	자동
비용	One-time	시간 또는 분	시간 또는 분	초, 실행 시간
고가용성	이중화 필요	이중화 필요	이중화 필요	자동
추상화	N/A	하드웨어	OS	프로그래밍 언어 런타임

정의

서버가 존재하지 않는다는 게 아니라, 사용자가 서버 설정 없이 함수 코드만으로 어플리케이션과 서비스를 구축, 실행할 수 있음을 의미

Aws lambda 서비스를 선보이며 용어 사용

함수 서비스라 FaaS 또는 function PaaS로 불리기도 함

AWS Lambda / Azure Functions / Google Cloud Functions / IBM Cloud Functions

공통 특성 :

1) 서버리스 : csp가 서버 프로비저닝, 설정, 구성 등 서버 관리. 내부적으로 컨테이너 기반이나 사용자는 알 필요 없음

2) 이벤트 드리븐 : 함수(코드) 등록 후 트리거로 호출하면 동작하는 방식. 호출 방식은 주로 http, queue, 게시/구독, 데이터 스트림 또는 폴링 이벤트 등

3) 개발 생산성 향상 : 서버 관리 필요없어 비즈니스 로직에 집중. 함수별로 언어가 달라도 영향 없음

4) 자동 확장 : 클라우드 환경이라 시스템 확장 편리. 서버리스 컴퓨팅은 설정없이 요청 수에 맞춰 자동 확장

5) 사용한 만큼 비용 지불 : 실행 시간, 요청 횟수로 과금하여 자주 사용하지 않는 시스템이면 가상머신보다 저렴

동작 방식

1) 이벤트 트리거에 의해 서버리스 컴퓨팅 호출

2) 서버리스 컴퓨팅이 설치된 서버에서 함수를 처리하기 위한 컨테이너 할당

3) 특정 스토리지에서 함수 코드 무결성 체크 후 할당된 컨테이너에 등록, 이벤트 트리거에서 전달한 이벤트 소스와 함께 함수 호출

4) 함수 코드 수행

5) 결과값 전달

6) 일정 시간 후 할당된 컨테이너 회수

도입 고려사항

1) 공급업체 종속성 :

다른 클라우드 업체나 온프레미스로 이전 시 막대한 비용 발생

업체별 사용 방법이 다르고 트리거 서비스가 연계되지 않아 멀티 클라우드 전략이 어려움

공급 업체 종속성을 피하려면 오픈소스 서버리스 컴퓨팅 플랫폼을 이용해야 함

2) 업체마다 다른 SLA

서버를 관리하는 공급업체의 실수로 장애가 발생할 수 있는데, 이에 대한 대비와 보상을 위해 SLA 확인 필요 (AWS Lambda 같은 경우 SLA 없음)

이를 참고하여 중요한 업무에는 서버리스 컴퓨팅을 사용하지 않거나 예외처리를 해야 한다

3) 복잡한 운영 관리

별도 서비스에 로그가 쌓이고 보관 비용 발생

수많은 함수가 있으면 원인 파악이 쉽지 않아 배포 시 단위/통합/연계테스트가 중요하고 롤백 방안 등도 필요함

트리거와 연계 서비스가 필요하므로 보안 사고가 없도록 적절히 필요한 권한만 사용하는 것이 중요

4) 기타 제약 사항

최대 실행 시간 : 실행 제한시간이 존재하며, 함수별로 설정 가능, 잘못된 로직으로 인한 과다 비용 발생 방지 가능. 제한 시간 초과 시 바로 종료됨(9~15분). 긴 시간이 필요하면 함수를 쪼개거나 가상머신 같은 serverfull한 서비스를 이용. 최대 실행 시간이 초과된 함수의 재시도는 호출 단에서 처리하는 것이 안전

최대 동시 실행 수 : 플랫폼별로 제한 기준은 다음. 동시 실행 수 오류가 발생하면 로직을 추가해야 안정적 서비스 유지 가능

콜드 스타트 지연시간 : 이벤트 호출로 시작될 때 컨테이너를 할당받는 준비과정이 수행되는 시간을 뜻함. 수행 종료 후 컨테이너 반납 없이 일정 시간 대기하다 재호출이 일어나서 컨테이너를 재사용하는 시간은 웜스타트. 콜드스타트는 웜스타트보다 100ms~10s의 지연시간이 발생하기 때문에 이를 고려하여 비기능 설계를 해야하며, 콜드스타트 지연시간을 허용할 수 없는 서비스는 서버리스 컴퓨팅 외의 방안을 사용해야 함

아키텍처 고려사항

적용 영역 선정

1) 신규 소규모 서비스 : 빠른 출시가 필요한 소규모 웹사이트, 모바일 어플리케이션 등

대규모 아키텍처는 함수들 관리가 복잡하고 모니터링이 어려움

2) Add on : 기존 시스템에 기능을 추가할 때 트리거가 될 서비스만 연결하면 기능이 변경되어도 패치 없이 함수만 변경하면 되므로 빠른 대응 가능.

그러나 기존 시스템의 전체 마이그레이션은 추천하지 않음. 하더라도 점진적 교체

3) 이벤트 드리븐 마이그레이션 : 기존 아키텍처를 이벤트 드리븐 아키텍처로 마이그레이션 할 때 사용. Ex) Web socket은 클라이언트가 많아지면 많은 서버 필요 -> 이벤트 드리븐으로 변경

4) proxy API 서버 : 폐쇄망에서 외부 데이터 가져올 때 proxy api 서버로 사용

5) 배치 처리 : 타이머 + 서버리스 컴퓨팅. 배치는 가끔 실행하거나 일정 간격으로 실행하므로 효율적. 오래 걸리는 작업은 비동기 호출이나 함수를 쪼갠다.

6) 스트림 데이터 실시간 처리 : 대량 스트림 데이터 수집 후, 분석할 때만 사용. 동일 비용으로 빨리 분석하려면 스트림 데이터를 병렬로 처리

적용 아키텍처 선정

1) 파일 처리 : 파일 이벤트를 감지하여 이미지나 동영상 같은 컨텐츠성 파일 변환 또는 데이터 파일 분석 용도

S3 / EC2+SNS -> Lambda (api gateway 연결) -> S3/RDS/EC2 DB/Dynamo DB 등에 저장

2) 실시간 스트림 데이터 처리 : pipe-filter 패턴으로 연속적인 인풋 데이터에 변형을 주거나 필요한 부분만 추출. ETL, IoT, 비디오 스트리밍, 클릭 이벤트 같은 스트림 데이터를 수집하여 실시간 처리

Kinesis / Dynamo DB -> Lambda (여러 기능으로 분산) -> S3 / Dynamo DB / RDS / ElasticSearch 등

3) 서버리스 백엔드 : restful api를 구성하여 웹사이트/모바일/IoT/Proxy API 등 다양한 백엔드로 활용

API gateway (api 서비스로 http 호출) -> lambda (api gateway와 연결, 각 rest api 처리) – Dymano DB …

4) 타이머 기반 : 일정 시간마다 수행해야 하는 경우 – 운영 자동화/배치/주기적 CI, CD 작업 등

Timer -> lambda -> EC2/메일서비스 등

아키텍처 설계

1) lambda 사이징 : 메모리 설정 시 cpu 자동 적용됨, 높은 메모리로 짧은 시간 실행이 저렴

2) 인터페이스 : 비즈니스 요구사항에 맞는 이벤트 트리거 사용 필요

3) 스토리지 : 휘발성 스토리지이므로 처리 결과는 별도 저장소(기존 서비스가 사용하던)에 보관 필요

4) 네트워크 : 내부망에 함수가 위치하면 다른 시스템 데이터를 가져오거나 저장할 때 제약이 많으므로 꼭 필요한 경우가 아니면 외부망에 위치

5) 보안 : 이벤트 트리거가 함수 실행 시, 함수가 다른 서비스 연결 시 인증 절차 필요, 필요한 권한만 적용

6) 서비스별 제한 : 한 예로 api gateway는 초당 처리 요청 수 제한이 있어 lambda를 확장해도 앞단에서 처리 불가능한 경우 발생. 시스템의 모든 제한 값을 파악하여 확장 또는 별도 방안 수립

7) 동시 실행 수 : 동시 실행 수도 자동확장되나 시간 지연 시 원활한 서비스 제공이 안될 수 있으니 미리 늘리는 방안 검토

8) 콜드스타트 지연시간 : 회피 방안 고려

아키텍처 구성

1) 이벤트 트리거 계층 : 함수를 호출하는 매개체. HTTP, Storage 변화, 메시지 큐 (순차 처리 보장은 안하지만 안정성 높음), 타이머, 데이터 스트림 수집 시, NoSQL에 데이터 입/출력 시, 직접 호출

2) 이벤트 처리 계층 : 함수가 위치한 곳, 단독/순차 수행 가능, 순차 수행 시 워크플로우 서비스 활용 권장

하나의 함수는 하나의 기능만, 동작 시간 3초 미만, 콜드스타트 포함 최대 10초

하나의 함수는 여러 파일로 구성 가능하고 zip으로 압축하여 업로드

워크플로우는 직접 구현하거나 CSP의 서비스 이용 가능

3) 이벤트 저장 계층 : stateless 하므로 이를 보완하여 처리과정이나 결과를 별도 저장하는 계층

Storage, NoSQL, In-memory cache, Database, Datawarehouse