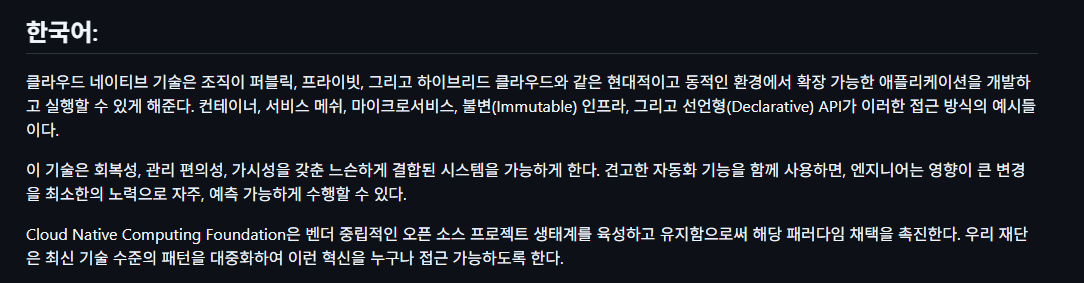
# 클라우드 네이티브란?

* 클라우드의 장점을 최대한 활용하여 정보 시스템을 구축 및 실행하는 환경
* 클라우드 네이티브 기술, 애플리케이션, 아키텍처, 개발방법론, 조직, 프로세스 등 다양한 용어와 결합하여 다양한 의미로 사용
* 쿠버네티스는 클라우드 네이티브 구성요소를 완전히 수행할 수 있는 최고의 플랫폼



출처: [toc/DEFINITION.md at main · cncf/toc · GitHub](https://github.com/cncf/toc/blob/main/DEFINITION.md)

# 기존 애플리케이션과 클라우드 네이티브 애플리케이션의 차이점

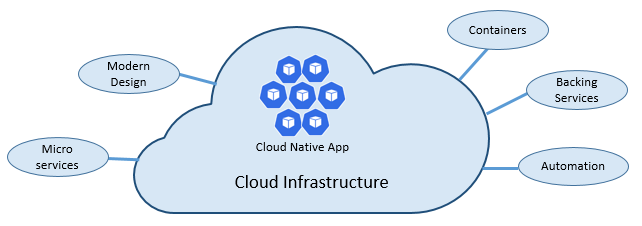
* 기존 애플리케이션은 장기간에 걸쳐 긴말하게 결합된 모놀리식 기반 위에서 동작
* 클라우드 네이티브 애플리케이션은 소규모 서비스 단위의 마이크로서비스로 구성되며 가상 컨테이너 환경에서 동작되도록 설계되고 구현



# 클라우드 네이티브 구성 요소



* 마이크로 서비스: 서비스를 각각 고유한 논리, 상태 및 데이터가 있는 독립적인 서비스로 기능을 분리
* 컨테이너: 코드, 해당 의존성 및 런타임은 컨테이너 이미지라는 이진 파일로 패키지
* 최신 디자인: 클라우드 기반 애플리케이션을 생성하기 위해 널리 허용되는 방법 12단계 애플리케이션(코드 베이스, 의존성, 구성, 지원서비스, 빌드/릴리즈/실행, 프로제스, 동시성, 일회성, Dev/Prod 패리티, 로깅, 관리 프로세스)
* 서비스 지원: 데이터 저장소, 메시지 브로커, 모니터링 및 id 서비스와 같은 다양한 보조 리소스 구성
* 자동화: 스크립트를 활용한 인프라, CD/CD 배포 자동화

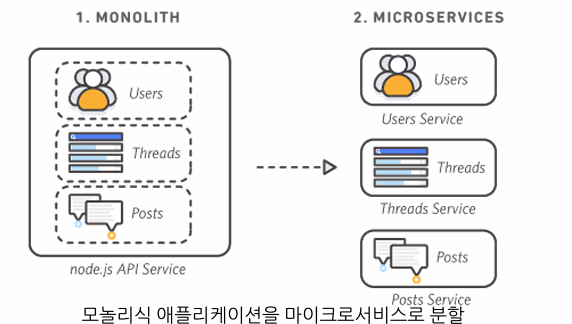


출처: [클라우드 네이티브란? - .NET | Microsoft Learn](https://learn.microsoft.com/ko-kr/dotnet/architecture/cloud-native/definition)

# 모놀리식 아키텍처 & 마이크로서비스 아키텍처

## 모놀리식 아키텍처(Monolithic architecture)

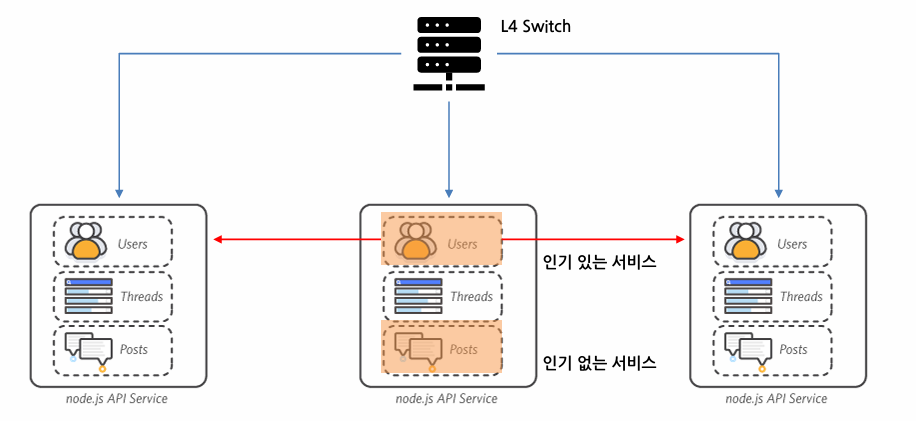
* 전통적인 아키텍처, 기존에 사용하던 서비스 방법
* 서비스가 하나의 애플리케이션으로 돌아가는 구조
* 기존의 개발 방식을 사용해 개발하여 간단히 배포
* 하나의 서비스 또는 어플리케이션이 하나의 거대한 아키텍처
* 다양한 기능을 동작하는 서비스를 서버에서 실행하여 서비스



출처: [마이크로서비스란 무엇입니까? | AWS](https://aws.amazon.com/ko/microservices/)

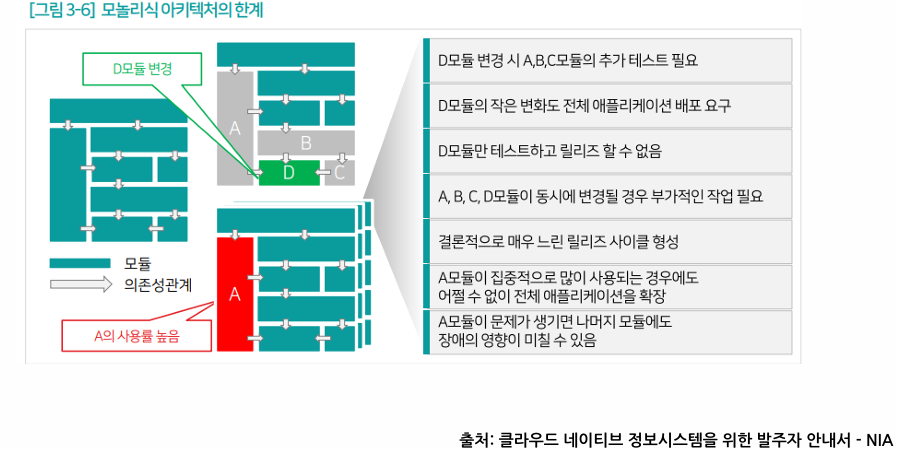
### 단점

* 모놀리식 서비스 아키텍처를 스케일아웃
* 기존의 애플리케이션을 그대로 복제하여 로드밸런싱
* 불필요한 서비스까지 모두 복제



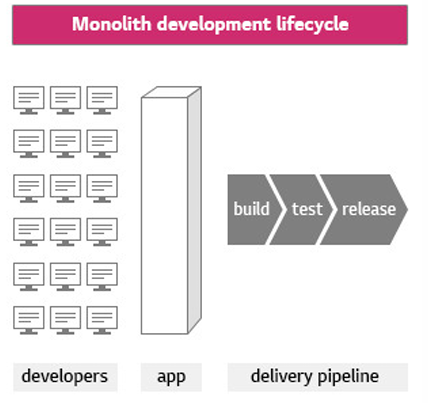
1. 종속적인 라이브러리의 충돌

* 각각의 기능들은 서로 다른 기능을 제공하여 버전의 종속성을 필요한 경우가 존재
* 각 기능의 따른 라이브러리를 매 업데이트마다 관리하기 매우 어려움



1. 조금만 수정해도 전체 빌드 및 배포 필요

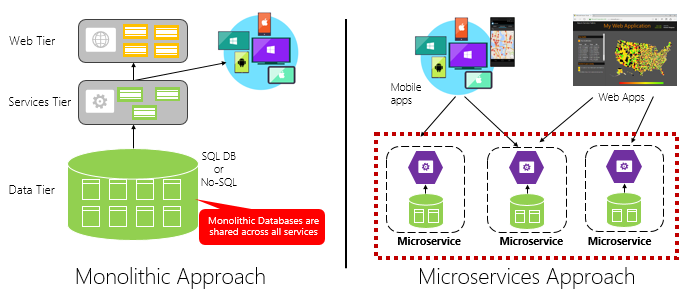
* 소스코드 전체가 하나로써 동작하기 때문에 작은 수정만 있더라도 전체를 빌드하여 다시 배포해야 함
* 프로그램의 크기가 어느정도 커짐면 한 번만 컴파일해서 전체 테스트를 돌려도 30분 내지 수 시간 소모
* 하루에 버그가 여러 개 순차적으로 발견된다면? 막막함



출처: [TISTORY](https://blog.lgcns.com/1278)

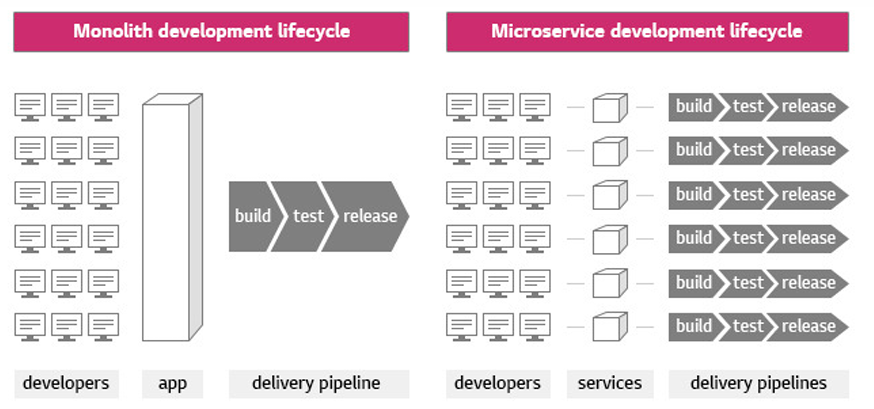
## 마이크로서비스 아키텍처

* 모놀리식 아키텍처의 대안으로 반대되는 개념
* 애플리케이션의 각각의 기능을 분리하여 개발 및 관리
* 마이크로 서비스 장점
  + 개발자가 특정 비즈니스 로직에 대해서만 집중하여 개발 가능해 빠른 개발
  + 개별 서비스 단위로 개발, 패키징, 빌드, 테스트, 배포로 각 서비스마다 유연한 스케줄
  + 서비스 단위로 스케일아웃이 가능하여 불필요한 서비스는 줄이고 더 많은 자원이 필요한 서비스는 확장가능



출처: [클라우드 네이티브란? - .NET | Microsoft Learn](https://learn.microsoft.com/ko-kr/dotnet/architecture/cloud-native/definition)

### 모놀리식 라이프 사이클과 마이크로 서비스 라이프 사이클 비교



출처: [TISTORY](https://blog.lgcns.com/1278)

### 장점

* 서비스 단위 고효율 저비용 Scale-Out 구조
  + 서비스 단위로 스케일아웃이 가능하여 불필요한 서비스는 줄이고 더 많은 자원이 필요한 서비스는 확장 가능
* 라이브러리 종속성 고민 X



### 단점

* 분산 시스템 환경에서 Transaction 보장, 테스트, 배포, 관리 복잡

# DevOps 모델 개념

* 데브옵스는 소프트웨어 개발과 IT 운영을 결합한 합성어
* 기존의 분리된 소프트웨어 개발팀과 IT 운영팀의 협업으로 전체 라이프사이클을 함께 관리할 수 있자는 일종의 철학 또는 운동
* 소프트웨어 개발팀과 IT팀이 더 빠르고 안정적으로 소프트웨어를 빌드, 릴리즈할 수 있도록 두 팀 간의 프로세스를 자동화하는 일련의 과정

## 이점

* 속도: 서로의 업무에 대해 더 잘 이해하며 좀더 신속하게 사용자에게 필요한 업데이트를 수행 가능. 개발자는 소비자가 무엇을 원하는지, 운영자는 애플리케이션을 제공하는데 해결할 문제를 인지할 수 있다.
* 신속한 제공: 컨테이너와 마이크로서비스를 사용하면 더 자주 빠르게 릴리즈하는 것이 가능. 더 빠르게 좋은 기능을 제공할 수 있으며 개발자가 운영에 필요한 인프라와 하드웨어에 대해 잘 몰라도 릴리즈가 가능하다.
* 개발과 릴리즈가 편해지므로 안정성이 확보, 협업 강화

## 개발자와 시스템 관리자의 역할 분담

### 개발자의 관점

* 새로운 기능을 만들어 사용자 경험을 개선하는 것을 좋아함
* 기본 운영체제의 보안 패치나 이와 관련된 모든 것이 최선인지 확인하는 것을 시스템 관리자에게 맡기려 함

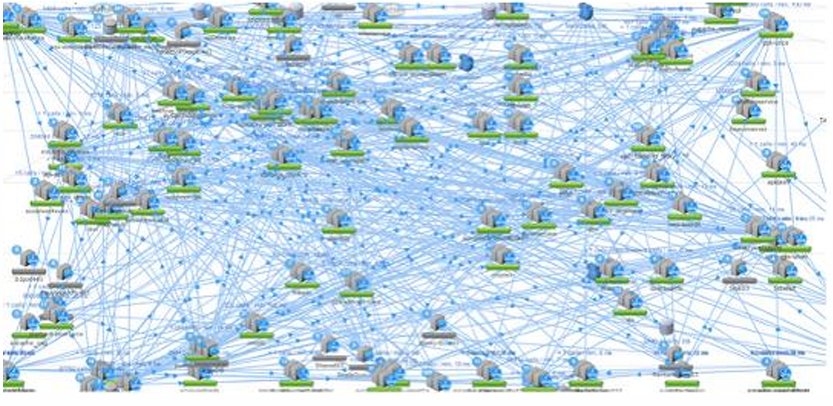
### 시스템 관리자의 관점

* 제품 배포와 운영하는 하드웨어 인프라를 담당하며 시스템보안, 활용, 개발자의 우선순위가 높지 않은 측면에 신경을 씀
* 운영 담당자는 모든 애플리케이션 구성 요소의 암묵적 상호 의존성에 대처하기를 원하지 않음
* 기본 운영체제나 인프라를 변경했을 때 애플리케이션 전체 동작에 어떤 영향을 미칠지는 생각치 못함

# 마이크로서비스 성공 사례

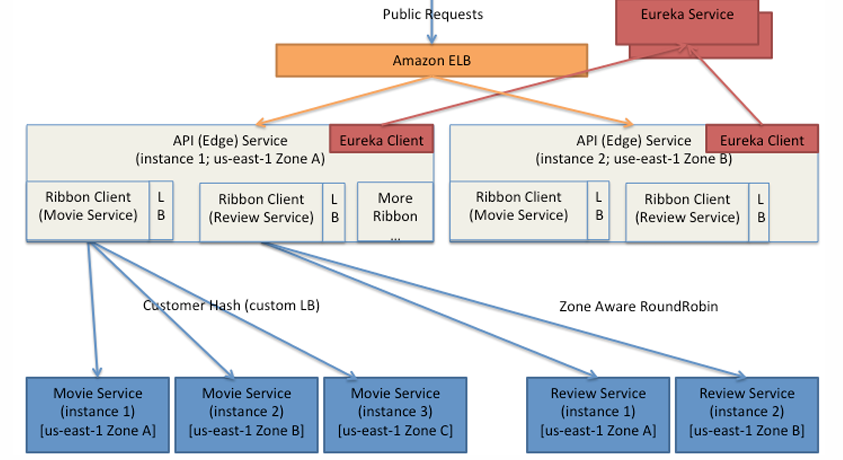
## Neflix 기술 블로그 (2013년)

* 최초의 마이크로서비스 사례
* 수백 개가 넘는 작은 단위의 서비스 운영
* API 기반의 마이크로서비스를 활용한 넷플릭스 아키텍처 의존성 도면  
  (컨테이너가 아닌 VM(AWS EC2)를 이용해 구성)



출처: Neflix 기술 블로그

* Service Oriented Architecture (Netflix의 대표적인 배포 아키텍처)



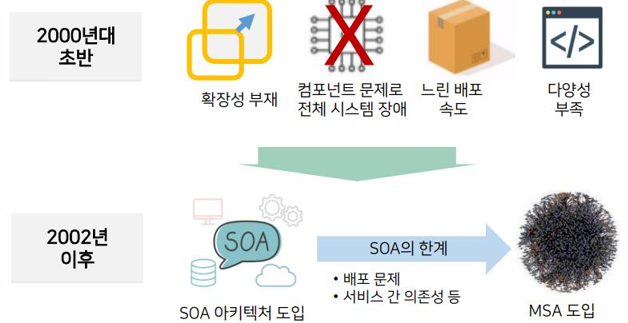
* Netflix 도입 배경과 내용



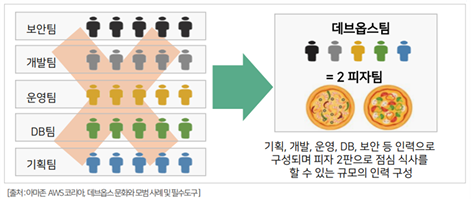
출처: 클라우드 네이티브 정보시스템을 위한 발주자안내서-NIA

## I Love APIs 2015: Microservices at Amazon

* 모놀리식 서비스에서 마이크로 서비스로 변경
* 빌드 타임 감소
* 복잡도 감소
* 수 천개의 DevOps 팀이 자체적으로 배포
* 연간 5천만회(2014년 기준)

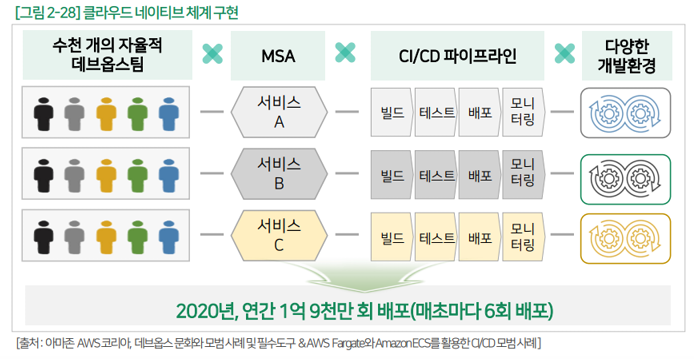


출처: 클라우드 네이티브 정보시스템을 위한 발주자안내서-NIA



### Amazon

* 수천 개의 자율적 데브옵스팀 구성, MSA 적용, 팀별 CI/CD 파이프라인 자동화를 통한 지속적인 통합/배포
* 다양한 개발환경 구성 등을 통해 신속한 서비스 배포 체계를 구축
* 2020년 기준 연간 1억 9천만 회(매초 6회) 배포



# 하이퍼바이저와 가상 머신의 이해

# 컨테이너의 개념과 기술 이해

# 도커와 쿠버네티스 이해