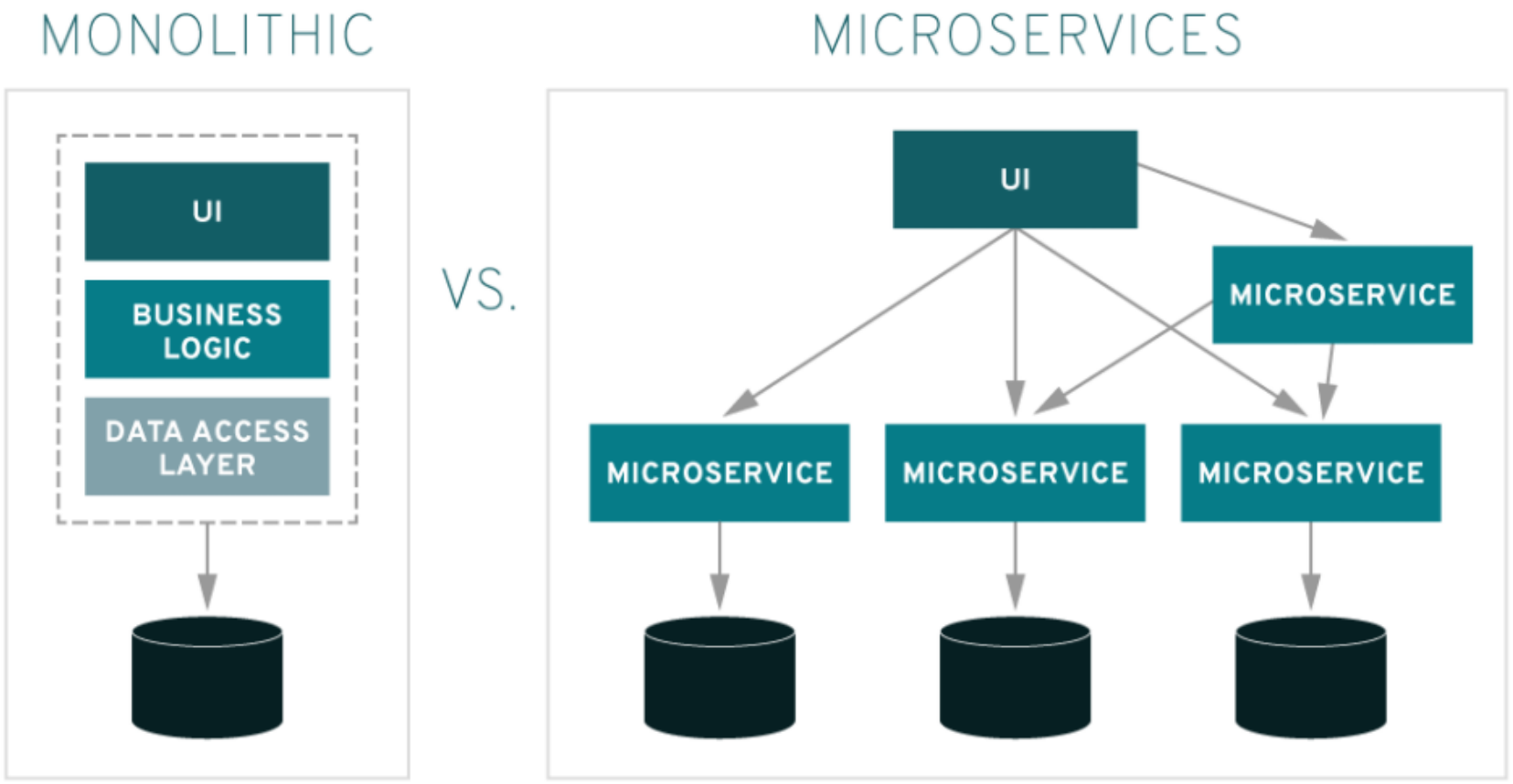
1. Monolithic vs MSA

- **MSA는 대형 어플리케이션의 경계를 허물고 시스템 내부에 논리적으로 독립적인 소형 시스템들을 구축하는데 도움을 준다.**



2. MSA 목적

- **MSA는 여러 서비스들을 상호 독립적으로 구축 및 운영될 수 있기 때문에 시스템에 대한 개발 및 운영 복잡성을 효율적으로 낮출 수 있다.**
- **특정 서비스만 집중할 수 있고, 코드 규모가 작아 효율적인 유지보수가 가능하다.**
- **Restful API와 같이 lightweight 한 통신을 통해 효과적인 유지보수가 가능하다.**
- **독립적인 서비스 단위 확장(scale-out)및 배포가 가능하기 때문에 효율적인 시스템 자원 활용이 가능하다.**

결국, 전체 서비스를 관련된 기능별로 분리해서 모듈화하고 각 모듈(마이크로서비스)을 독립적으로 설계, 개발, 테스트, 배포할 수 있게 만들어 느슨하게 결합시킨 서비스 형태로서, 모듈화 특성 때문에 빠른 구현과 단계적으로 완성도를 높이는 애자일 방법론과 기획부터 개발운영까지 아우르는 데브옵스 문화를 수용하는 소규모팀에 적합하다. 또한 리소스의 유연한 확장과 축소가 가능한 클라우드 아키텍쳐를 적용하는데 최적의 서비스 형태라고 할 수 있다.

3. MSA 아키텍쳐 특징

- **각각의 MSA는 자체 비즈니스 계층과 데이터베이스를 가지고 있다.**
  따라서 하나의 MSA를 수정한다고 해서 다른 MSA에 영향을 주지 않는다.
  (모듈성 강제)
- **일반적으로 MSA는 HTTP/REST 같은 널리 채택된 lightweight 프로토콜 또는 AMQP와 같은 비동기 메시징 프로토콜을 사용하여 서로 통신한다.**
- **어플리케이션 로직은 책임이 명확한 작은 컴포넌트들로 분해하고 이들을 조합해서 솔루션을 제공하며 완전히 상호 독립적으로 배포된다.**
- **MSA는 항상 기술 중립적인 프로토콜을 사용해 통신하므로 서비스 구현 기술과는 독립적이다.(프로그램언어, DB, 하드웨어등)**
  따라서 가장 중요한 기술적 선택은 서로 통신(동기, 비동기, UI연동)하고 프로토콜이 통신을 위해 사용되는 방식(REST, 메시징) 이다.

4. MSA 장단점 비교

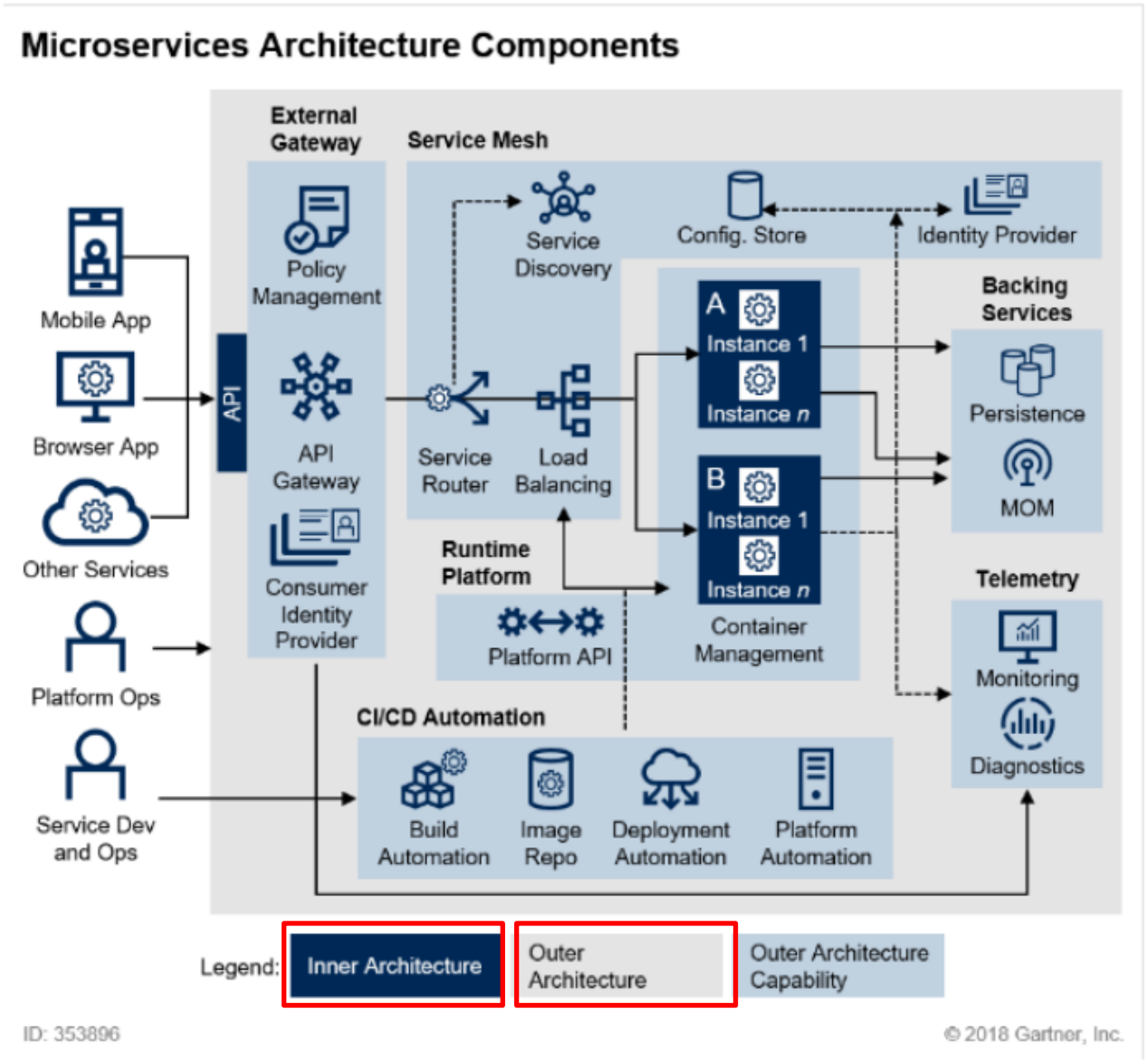
- **많은 서비스가 API 방식으로 상호 통신하기 때문에 전체 서비스 복잡도가 높아진다. 따라서 각 서비스 API에 대한 가시성, 접근성, 접근 제어, 모니터링등을 확보하기 위해 단일(모놀리식) 아키텍쳐보다 더 많은 노력이 필요하다.**

- **서비스가 분리되어 있기 때문에 테스트와 트랜잭션의 복잡도가 증가한다.**

MSA 장단점

장점	단점
다른 개발 기술 스택 사용가능	각 서비스별로 발생하는 장애처리
단일 사업 영역에 집중	각 서비스간 통신으로 인한 지연시간
소규모의 작은 배포 가능	각 서비스 및 구성 요소간 연동을 위한 설정 증가
소프트웨어의 수시 업데이트 가능	각 서비스간 통신양의 증가로 인한 관리 어려움
각 서비스에 대한 일관된 보안 적용	각 데이터베이스간 데이터 트랙킹이 어려움
각 서비스별 개발 및 배포 가능	(다른 언어를 쓸 경우) 코드 재사용 불가

5. MSA 아키텍쳐


Microservices Architecture Components
External Gateway
Service Mesh
Mobile App
Browser App
Other Services
Platform Ops
Service Dev and Ops
API
Policy Management
API Gateway
Consumer Identity Provider
Service Discovery
Service Router
Load Balancing
Config. Store
Identity Provider
A
Instance 1
Instance n
B
Instance 1
Instance n
Container Management
Runtime Platform
Platform API
Backing Services
Persistence
MOM
Telemetry
Monitoring
Diagnostics
CI/CD Automation
Build Automation
Image Repo
Deployment Automation
Platform Automation
Legend:
Inner Architecture
Outer Architecture
Outer Architecture Capability
ID: 353896
© 2018 Gartner, Inc.

6. Inner architecture

Inner architecture 개요

내부 서비스와 관련된 architecture로서 내부의 서비스를 어떻게 잘 분리하고
여러 서비스에 걸쳐진 트랜잭션을 어떻게 처리할 지에 대한 설계이다.
Inner architecture는 비즈니스마다, 서비스마다, 시스템마다 각각의 특성이 있기
때문에 정해진 표준이 없다.

Inner architecture 고려사항

- 서비스를 어떻게 정의할 것인가?
  쇼핑몰에서 주문과 카트 담기를 같은 서비스로 넣을 것인지, 다른
  서비스로 분리할 것인지는 그 비즈니스 시스템의 특성에 따라 정의되어야 한다.

- DB Access 구조를 어떻게 설계할 것인가?
  MSA의 데이터는 일반적으로 일관된 API를 통해서 접근하고 자체의 DB를 가질 수
  있는데, 일부의 비즈니스 트랜잭션은 여러 MSA에 걸쳐 있기 때문에, 각 서비스에
  연결된 데이터베이스의 트랜잭션을 보장해 줄 수 있는 방안이 필요하다.

- 논리적인 컴포넌트들의 layer를 어떤 방식으로 설계할 것인가?

7. Outer architecture

Outer architecture 개요

MSA를 적용한 시스템이 커지면 마이크로서비스의 인스턴스 수가 증가함에 따라 시스템 런타임 복잡성 문제가 발생된다. 보안, 로드 밸런싱, 모니터링 등 동적으로 수많은 마이크로서비스의 인스턴스간 통신이 유발하는 관심사들을 내부 네트워크에서 안정적으로 다루기 위해 새로운 기능들이 필요하고 이러한 서비스를 관리하기 위한 목적으로 Outer architecture 가 필요하다.

Outer architecture 분류

Gartner에서는 MSA의 Outer architecture을 총 6개의 영역으로 분류하고 있다.

1. **External Gateway**
2. **Service Mesh (*)**
3. **Container Management**
4. **Backing Services**
5. **Telemetry**
6. **CI/CD Automation**

8. Service Mesh 적용 방안

2) Spring Boot + Spring Cloud 기반 직접 구축

Spring Cloud는 어플리케이션 스택의 일부로서 모든 MSA 관심사를 해결하도록 잘 통합된 다양한 자바 라이브러리들의 묶음이다.
개발자가 분산 시스템의 공통 패턴인 Configuration Management, Service Discovery, Circuit Breaker, Routing, Proxy, Distribute Session, Cluster 상태 등을 신속하게 구축할 수 있는 도구를 제공한다.
개발자의 PC 뿐만 아니라 Cloud Foundry 같은 클라우드 플랫폼에서도 원활하게 동작된다.

Spring Cloud 제공기능

- **분산 및 버전으로 구분된 설정 관리 (Config)**
- **서비스 등록 및 조회 (Service Registry)**
- **라우팅 및 상태 조회 (API Gateway)**
- **서비스 대 서비스 호출 (RestTemplate, Feign Client)**
- **서비스 분산 로딩 (Load balancing)**
- **서비스 간 호출 분리 (Circuit breaker)**
- **클러스터링 환경 관리**
- **분산 메시징 (Kafka, RabbitMQ)**