CHAPTER 9

MICROSERVICES

Chapter 9: MICROSERVICES

OUTLINE

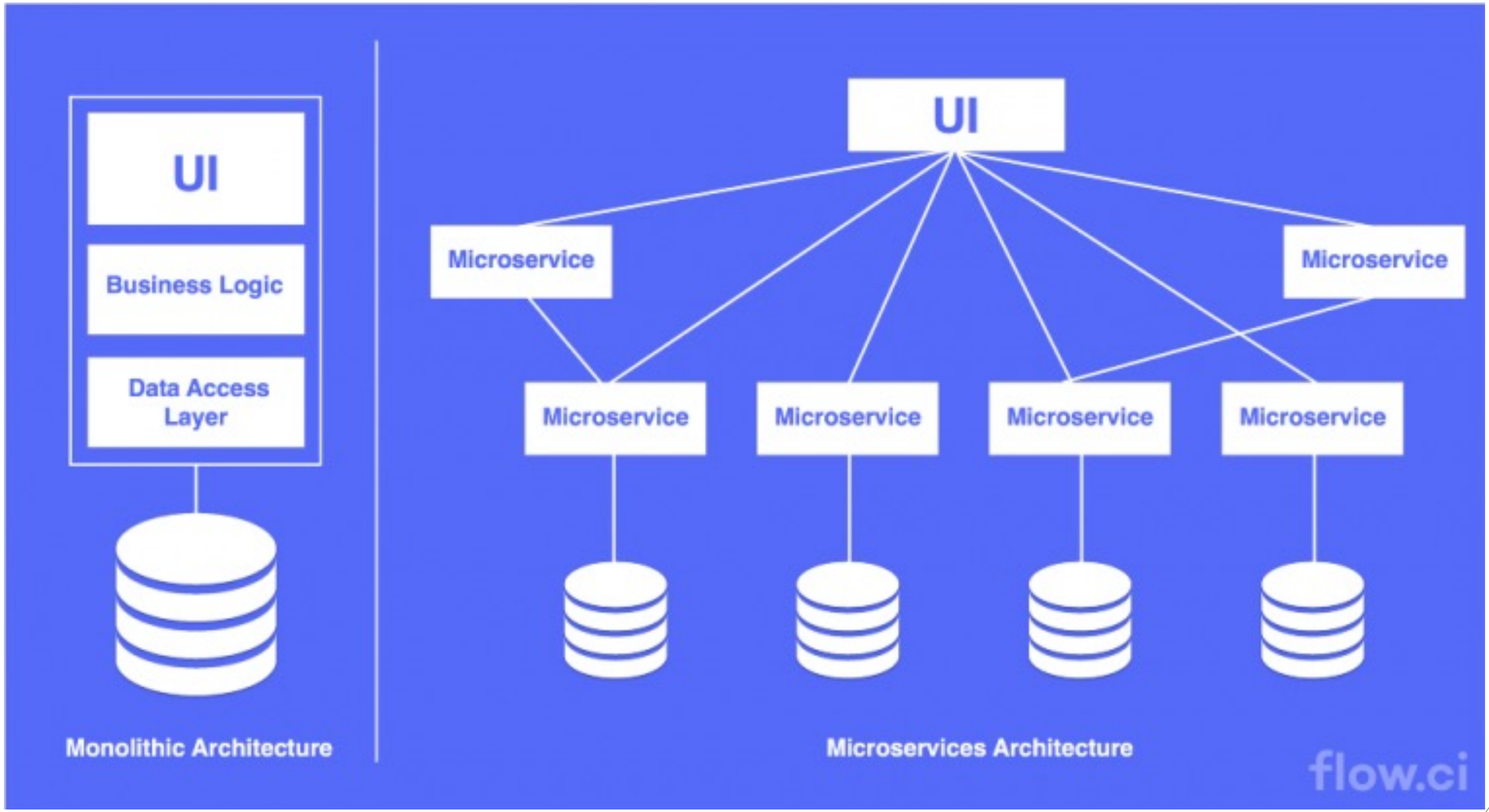
What are Microservices?

The microservice architecture style is an approach to developing:

- A single application as a **suite of small services**, each **running in its own process** and communicating with **lightweight mechanisms**, often an HTTP resource API.
- These services are **build around business capabilities** and **independently deployable** by fully automated deployment machinery.
- There is a **bare minimum of centralized management** of these services, which may be written in different programming languages and use different **data storage technologies**.

- James Lewis and Martin Fowler

What are Microservices?



SaaS & 12-Factor Apps

- Software –As-A-Service (SaaS)
  - Subscribe to software – periodic contract
  - Scalability – horizontal
  - Multi-tenancy
- 12-Factor apps:
  - Methodology for building SaaS apps
- Majority of Microservices & distributed system concerns are defined by 12-factor methodology

12-Factors

1. Codebase
   - Single codebase for one app – tracked in revision control
   - If more than 1 app then each has it’s own codebase – distributed system
2. Dependencies
   - Declared explicitly through dependency manifest – gradle, maven, npm, …
   - Dependency isolation – no implicit dependencies, gradle dependencyManagement for exact version of dependency.
   - Simplify developer setup
   - Project dependency – build system (gradle, maven, npm, …)
   - Runtime dependencies – containers (docker, ….)

12-Factors (cont.)

3. Processes
   - Stateless – share nothing
   - Persistent data needs to be stored in stateful backing service like database
4. Port binding
   - Export HTTP as a service by binding to a port
   - In build web server like Tomcat for Spring boot
5. Concurrency
   - Different process models for different tasks – WebServer for HTTP, workers for background
   - Multiple processes on multiple machines – easy to scale out

12-Factors (cont.)

6. Config
   - Strict separation of config from code
   - Hosted externally or setup as environment variables
7. Backing services
   - Databases, message queue system, S3, …
   - Treat as attached resources accessed over a URL
   - No code changes needed for resource swap – local mySQL with RDS
8. Build, release run
   - Strictly separate build (gradle, maven), release (Github action) and run (docker)
   - Ability to **rollback** to previous release

12-Factors (cont.)

9. Disposability
   - Started or stopped at moments notice
10. Dev/prod parity
    - DevOps, quick deploys
    - Use Docker and external configuration
11. Logs
    - Treat logs as event streams routed to common destination
    - ELK stack, Fluent, ..
12. Admin processes
    - One off administrative maintenance tasks like database migration, running one-time scripts

Companies that use microservices

Netflix

Amazon

eBay

Uber

Twitter

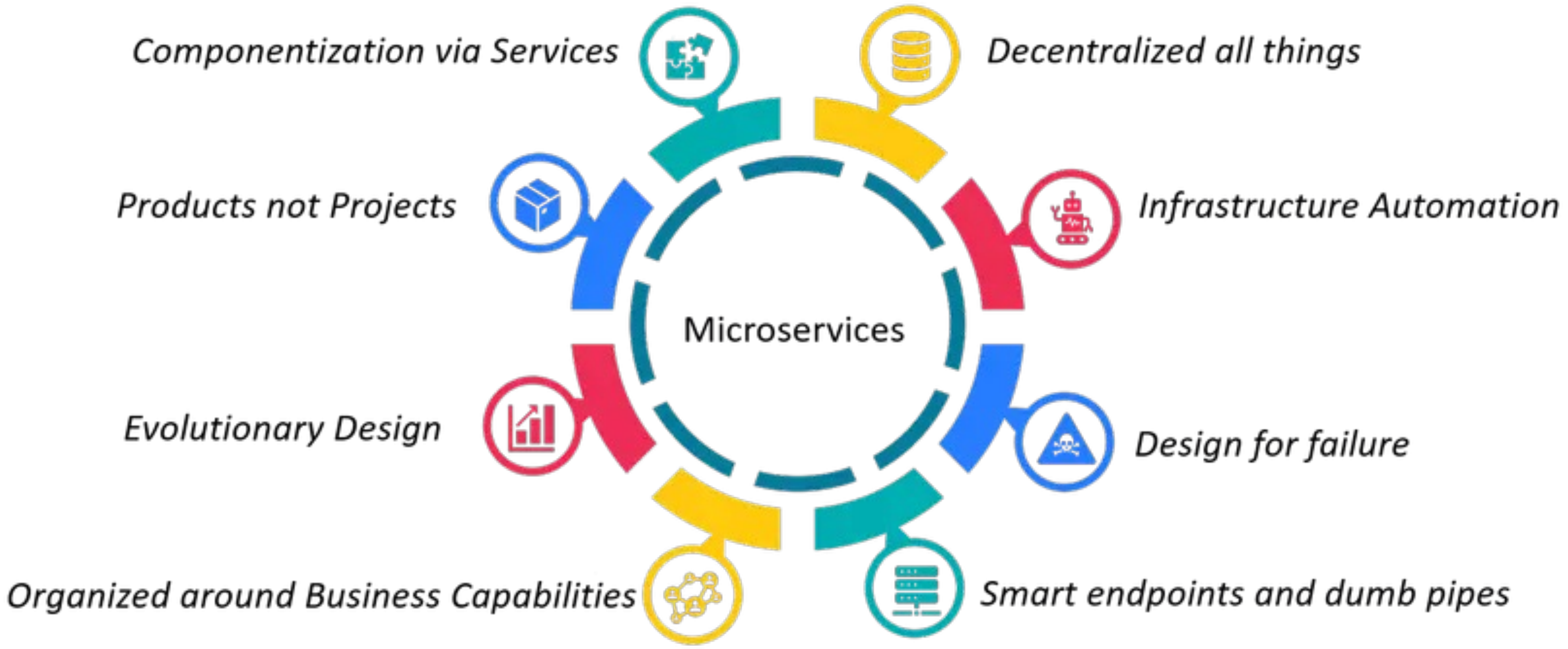
PayPal

Spotify

The Guardian

LinkedIn

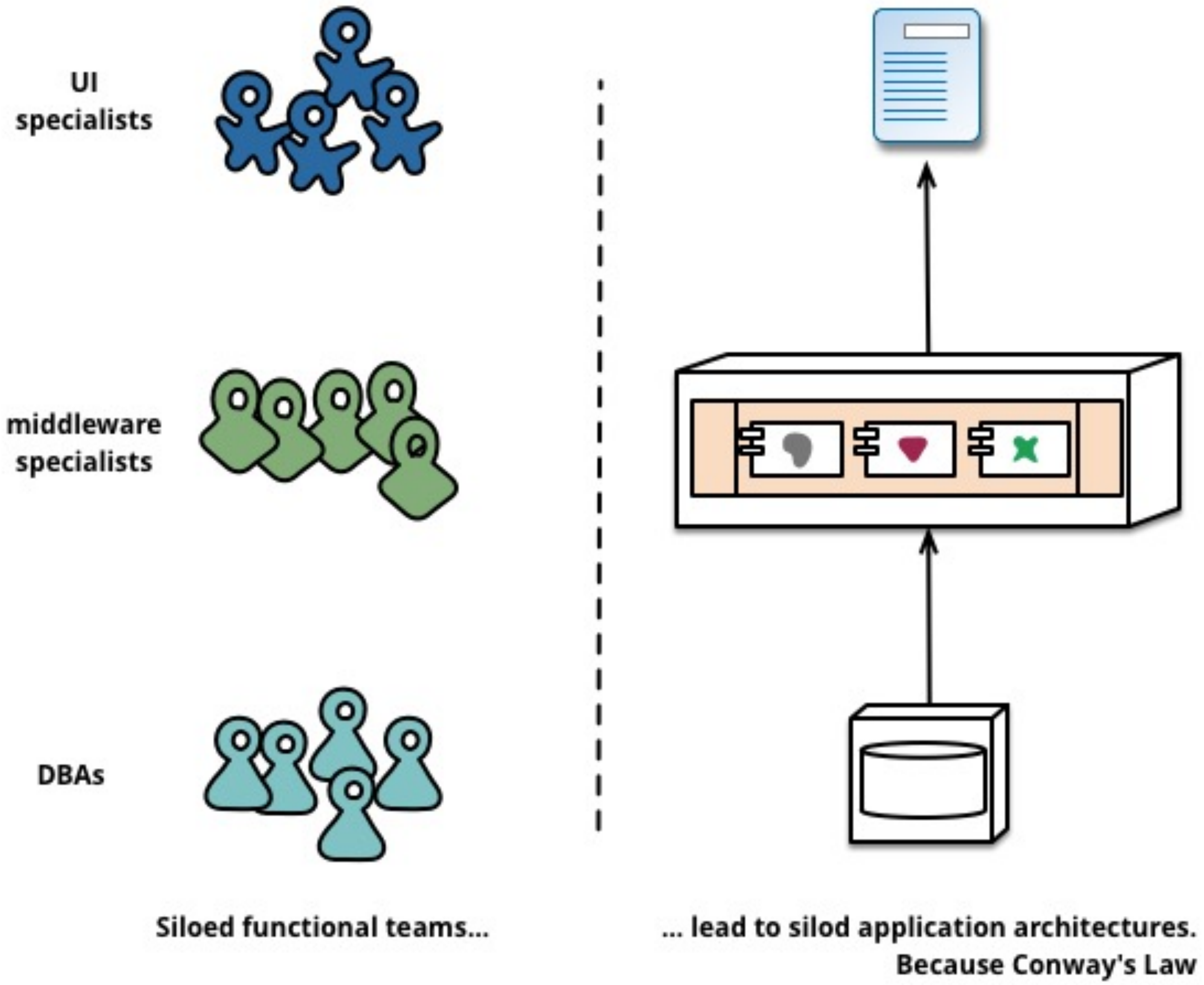
Characteristics of a Microservice Architecture


Componentization via Services
Decentralized all things
Products not Projects
Infrastructure Automation
Microservices
Evolutionary Design
Design for failure
Organized around Business Capabilities
Smart endpoints and dumb pipes

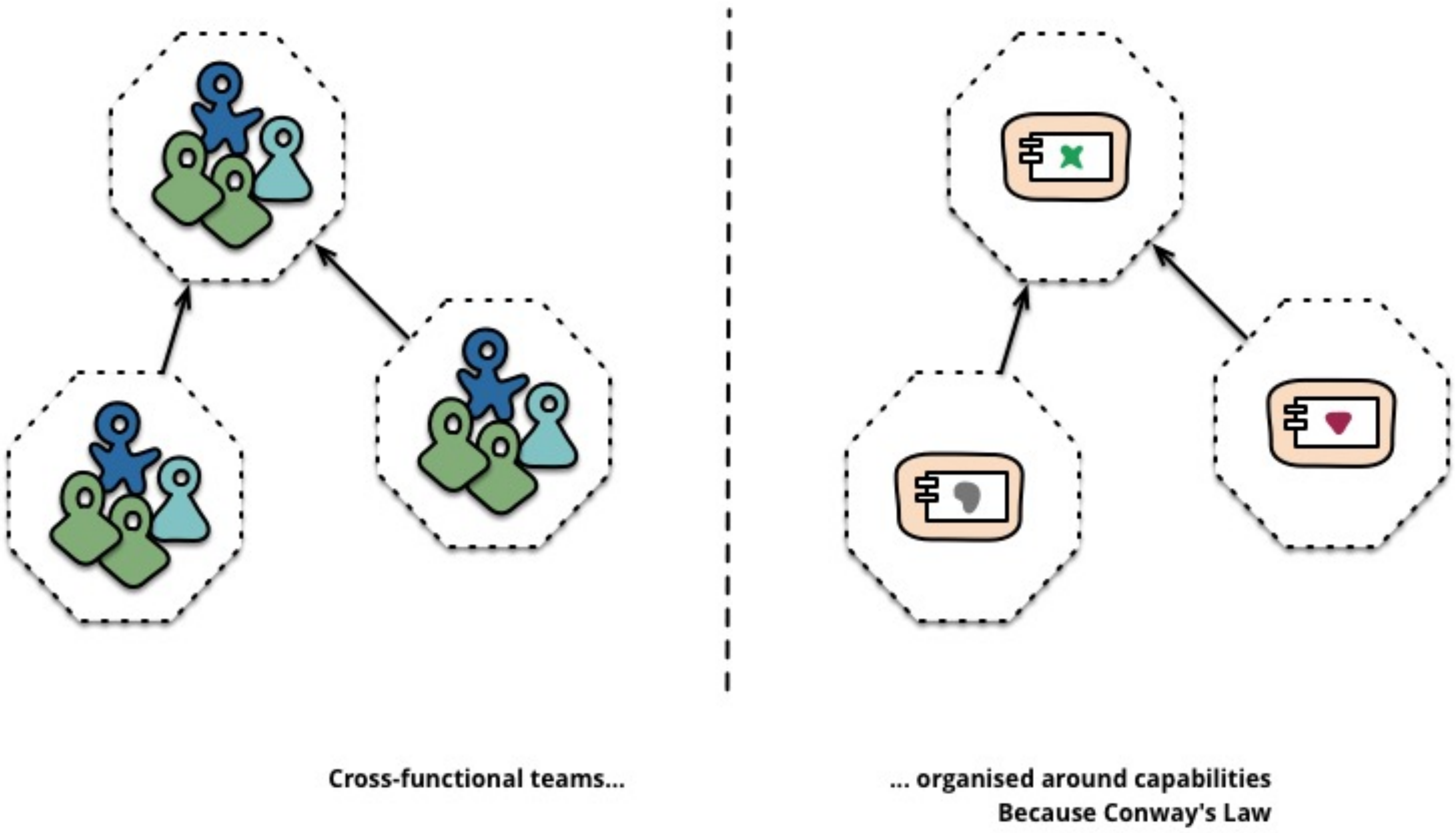
Componentization via Services

- Component is a unit of software that is independently replaceable and upgradeable
- One main reason for using services as components (rathe than libraries) is that services are independently deployable.
- Each Service talks to other services using well defined (public) API

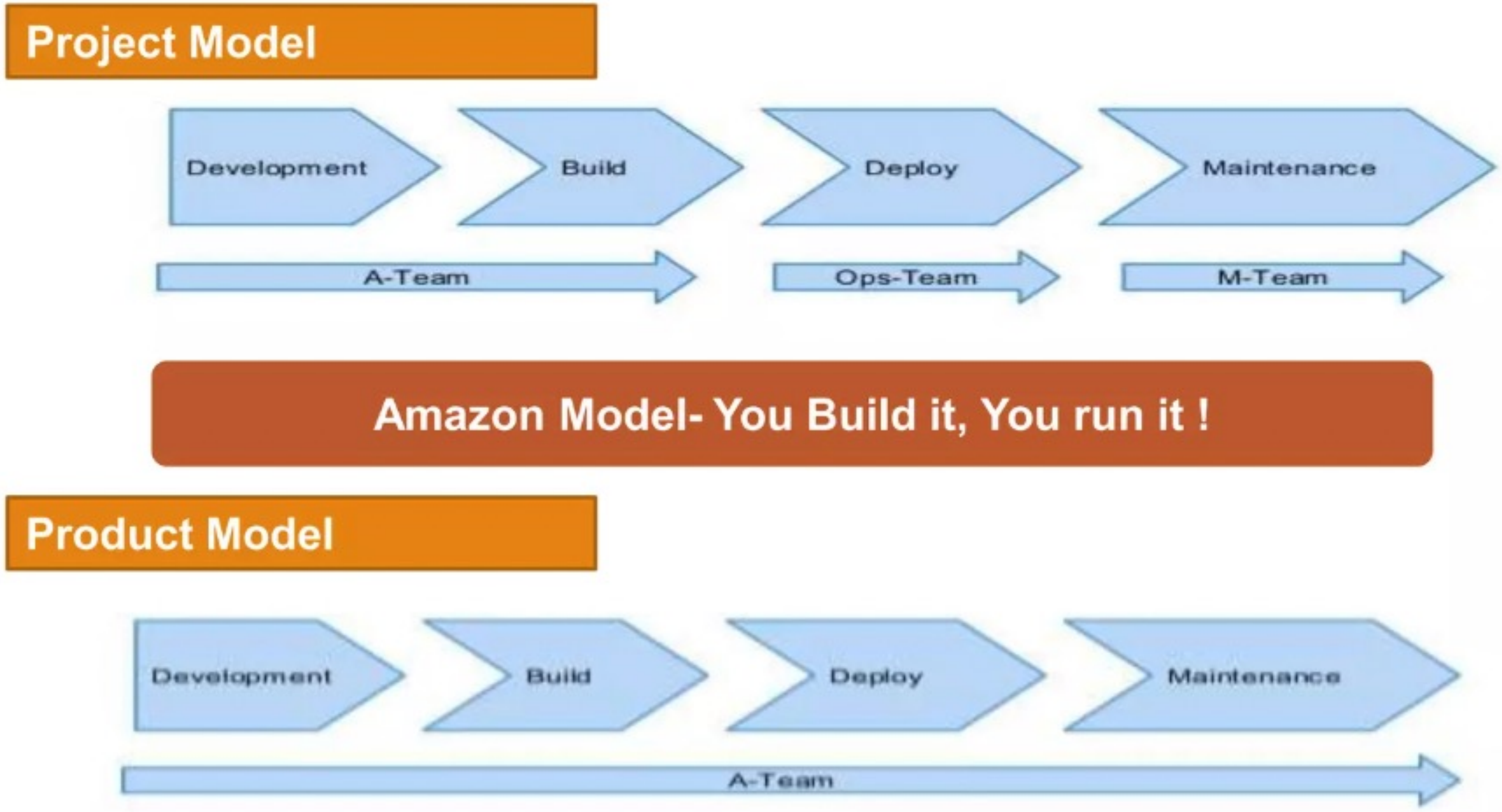
Organized around Business Capabilities


UI specialists
middleware specialists
DBAs
Siloed functional teams...
... lead to silod application architectures.
Because Conway's Law

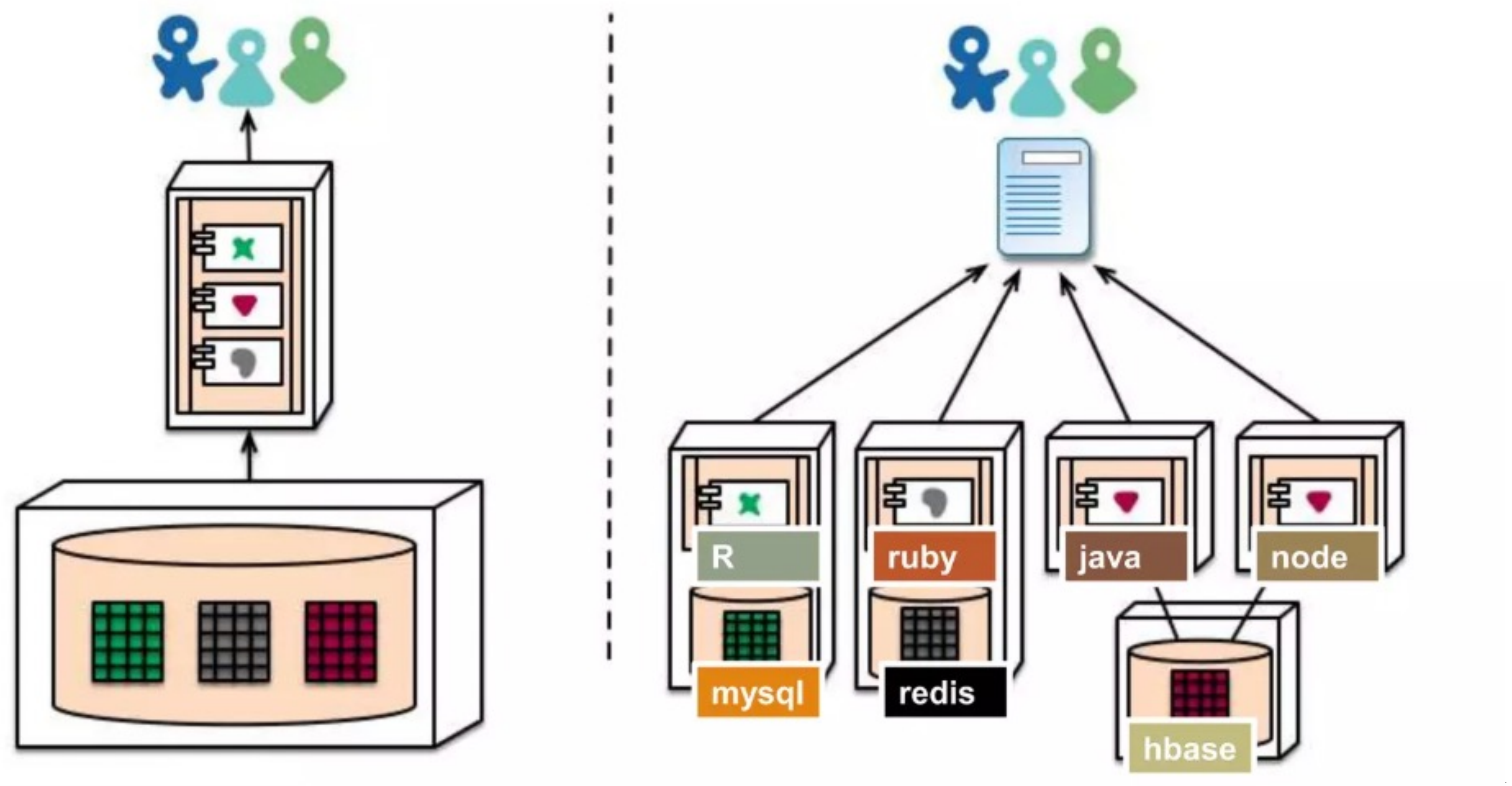
Organized around Business Capabilities (cont.)



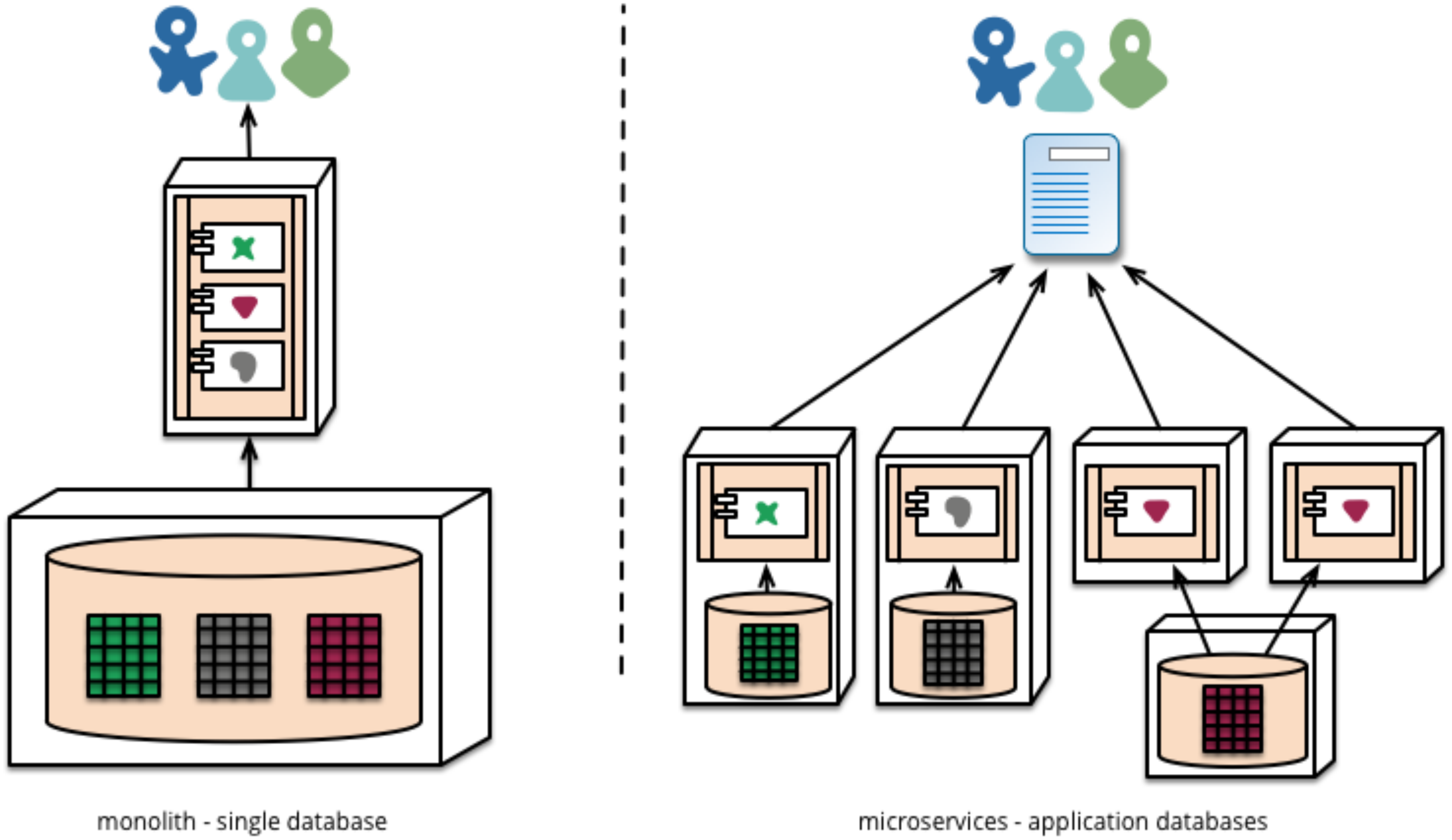
Products not Projects



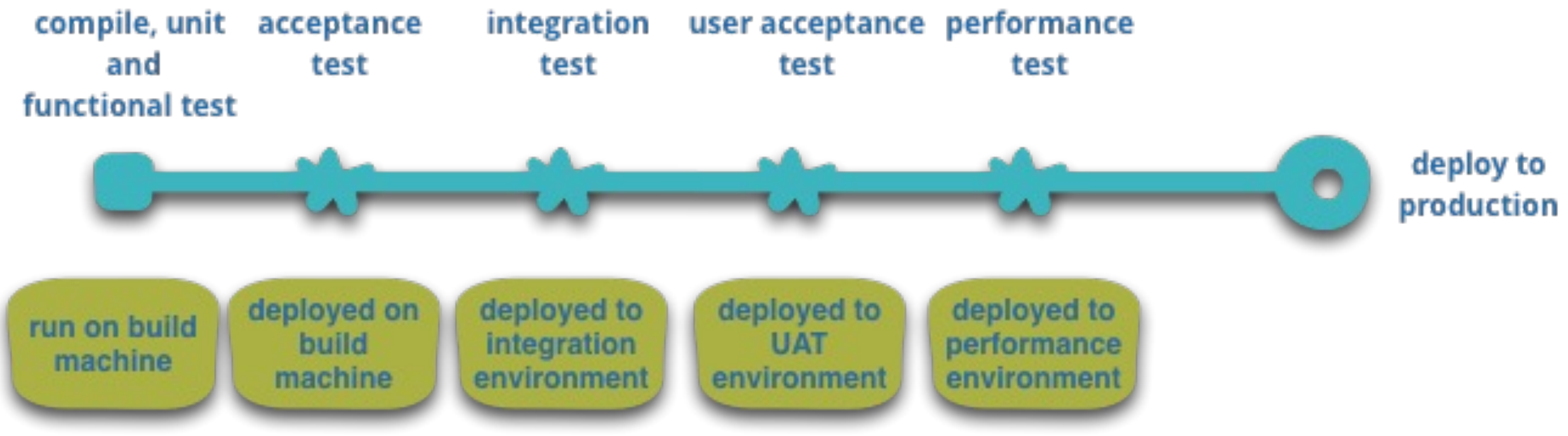
Decentralized Governance



Decentralized Data Management


monolith - single database
microservices - application databases

Infrastructure Automation


compile, unit and functional test
acceptance test
integration test
user acceptance test
performance test
deploy to production
run on build machine
deployed on build machine
deployed to integration environment
deployed to UAT environment
deployed to performance environment

Design for failure

- Applications need to be designed so that the can tolerate the failure of services
- Detect the failures quickly and, if possible, automatically restore service

Evolutionary Design

- Drive modularity through the pattern of change
- Ongoing service decomposition – independent replacement and upgradeability

Microservice Ecosystem

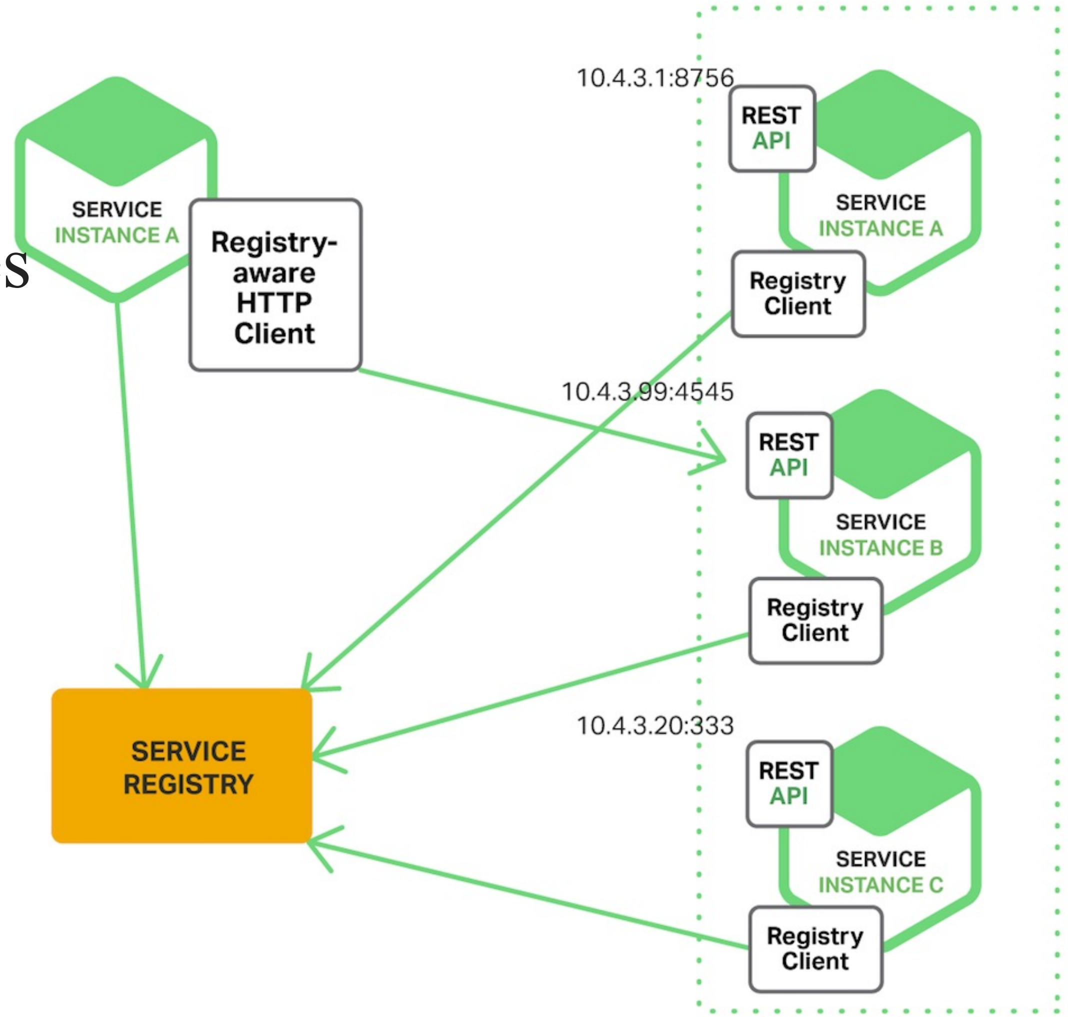
1. Load balancer: distribute the incoming load among many instances of Microservices
   - **Client-Side Discovery Pattern:** the client is responsible for determining the network locations of available service instances and load balancing requests across them. The client queries a service registry, which is a database of available service instances. The client then uses a load-balancing algorithm to select one of the available service instances and makes a request.
   - **Server-Side Discovery Pattern:** The client makes a request to a service via a load balancer. The load balancer queries the service registry and routes each request to an available service instance. As with client-side discovery, service instances are registered and deregistered with the service registry.

Microservice Ecosystem

1. **Load balancer:** distribute the incoming load among many instances of Microservices

Client-Side Discovery Pattern:

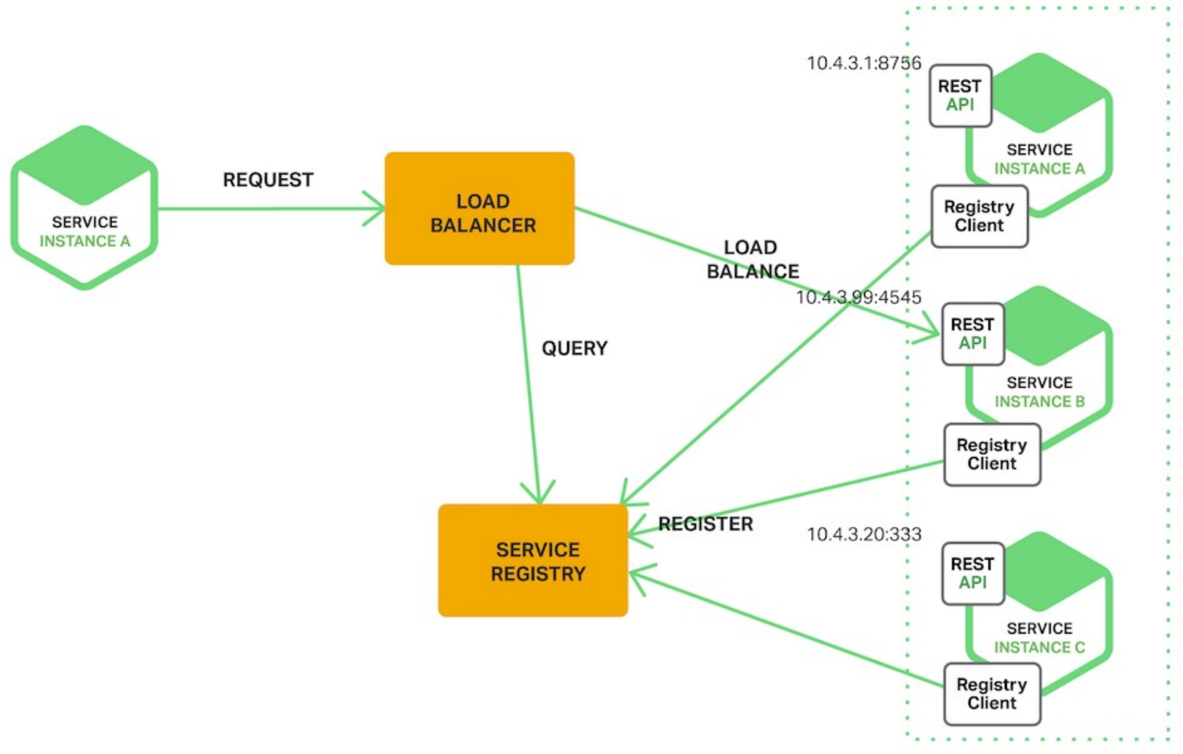
- Client talks to Service registry and does load balancing
- Client service needs to be Service registry aware
- Ex: Netflix OSS



Microservice Ecosystem (cont.)

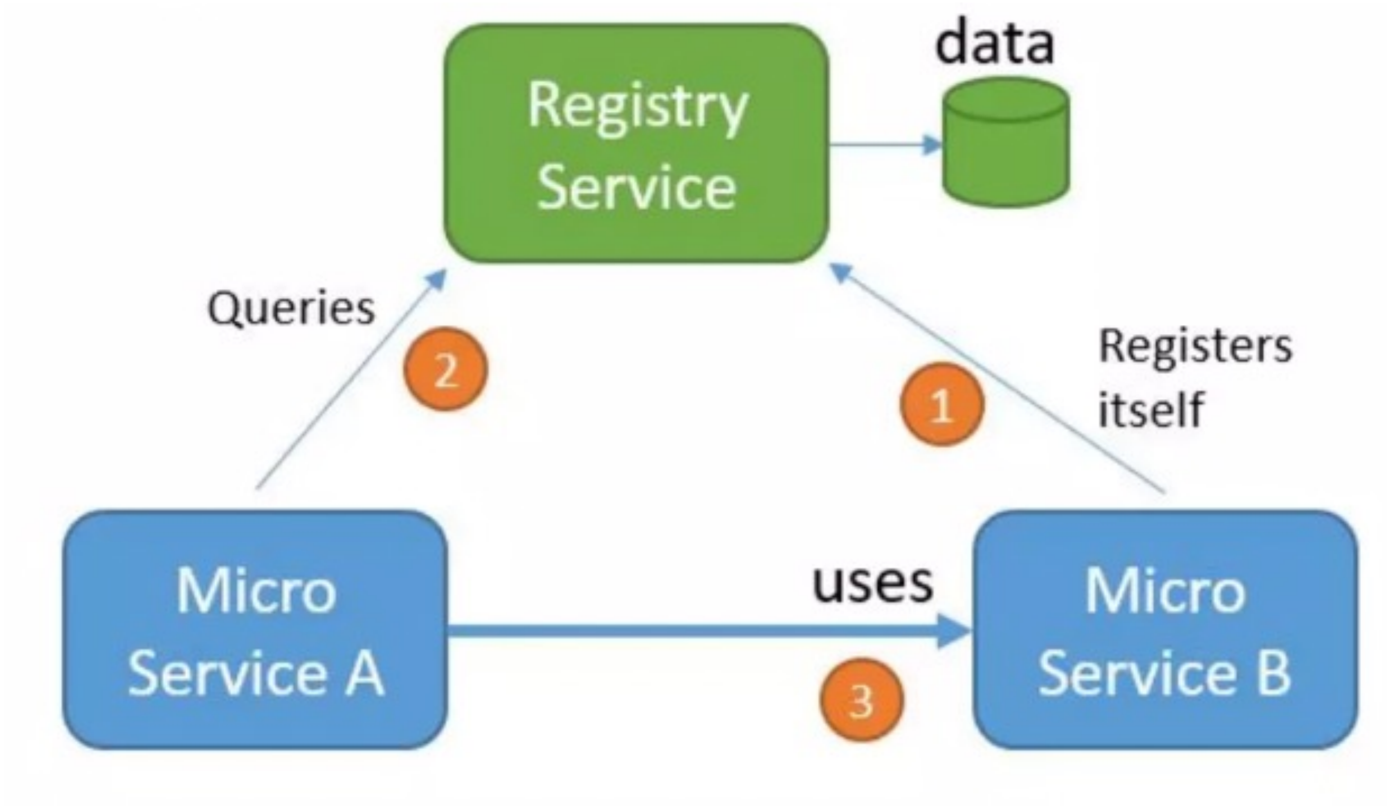
Server-Side Discovery Pattern

- Client talks to load balancer and load balancer talks to Service registry
- Client service needs not be Service registry aware
- Ex: Consul, AWS ELB, k8s, Docker



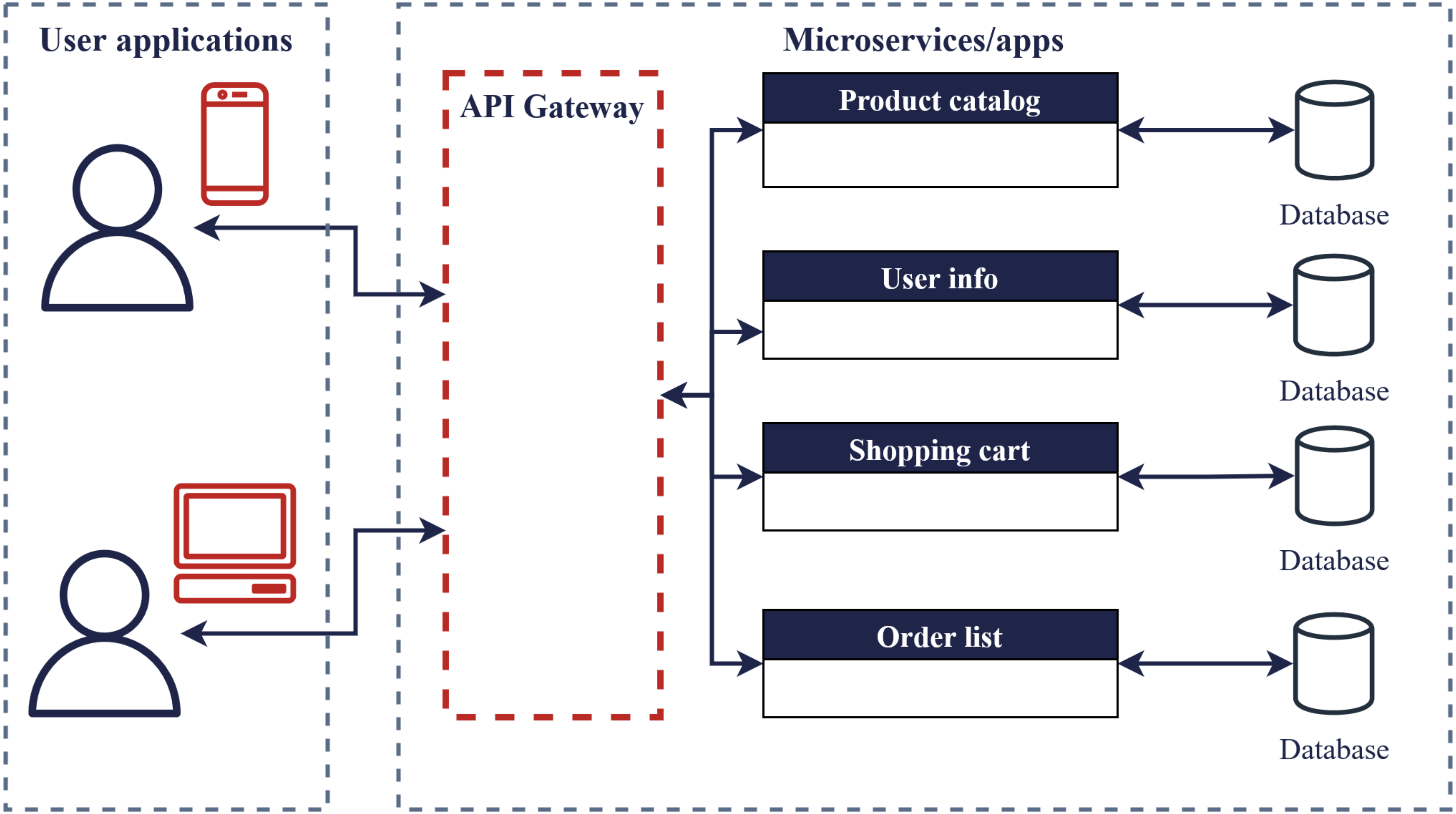
Microservice Ecosystem (cont.)

2. **Service Discovery server**: instead of manually keeping track of what microservices that are deployed currently and on what hosts and ports we need service discovery functionality that allows, through an API, microservices to self-register at startup



Microservice Ecosystem (cont.)

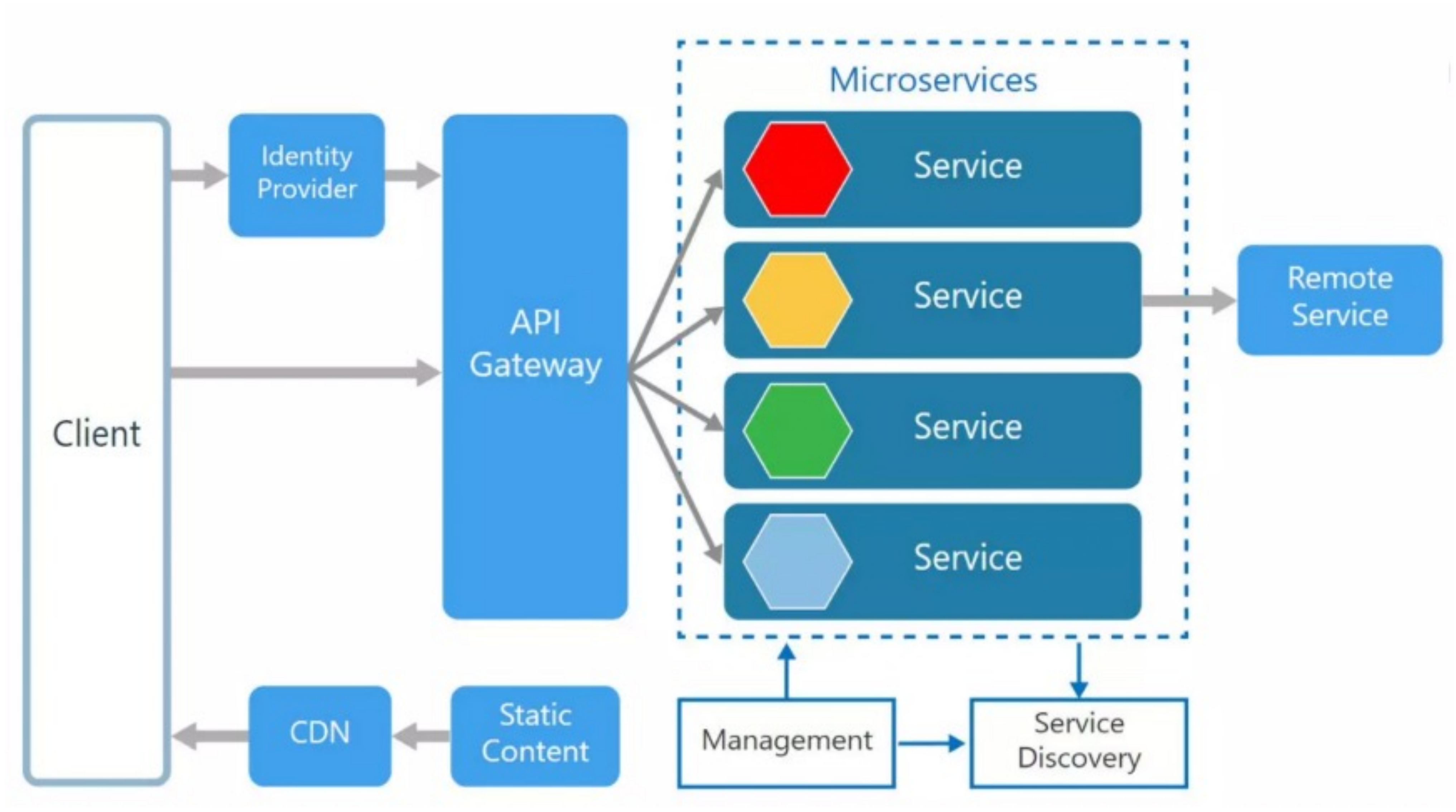
3. **API Gateway**:
   - Insulates the client from how the application is partitioned into microservices
   - Insulates the client from the problem of determining the locations of service instances
   - Provides the optimal API for each client


User applications
Microservices/apps
API Gateway
Product catalog
User info
Shopping cart
Order list
Database
Database
Database
Database

Microservice Ecosystem (cont.)

4. Central Configuration server
5. Monitoring
6. Containerization
7. Centralized log analysis
8. Circuit Breaker: to avoid the chain of failures problem we need to apply the Circuit Breaker patterns

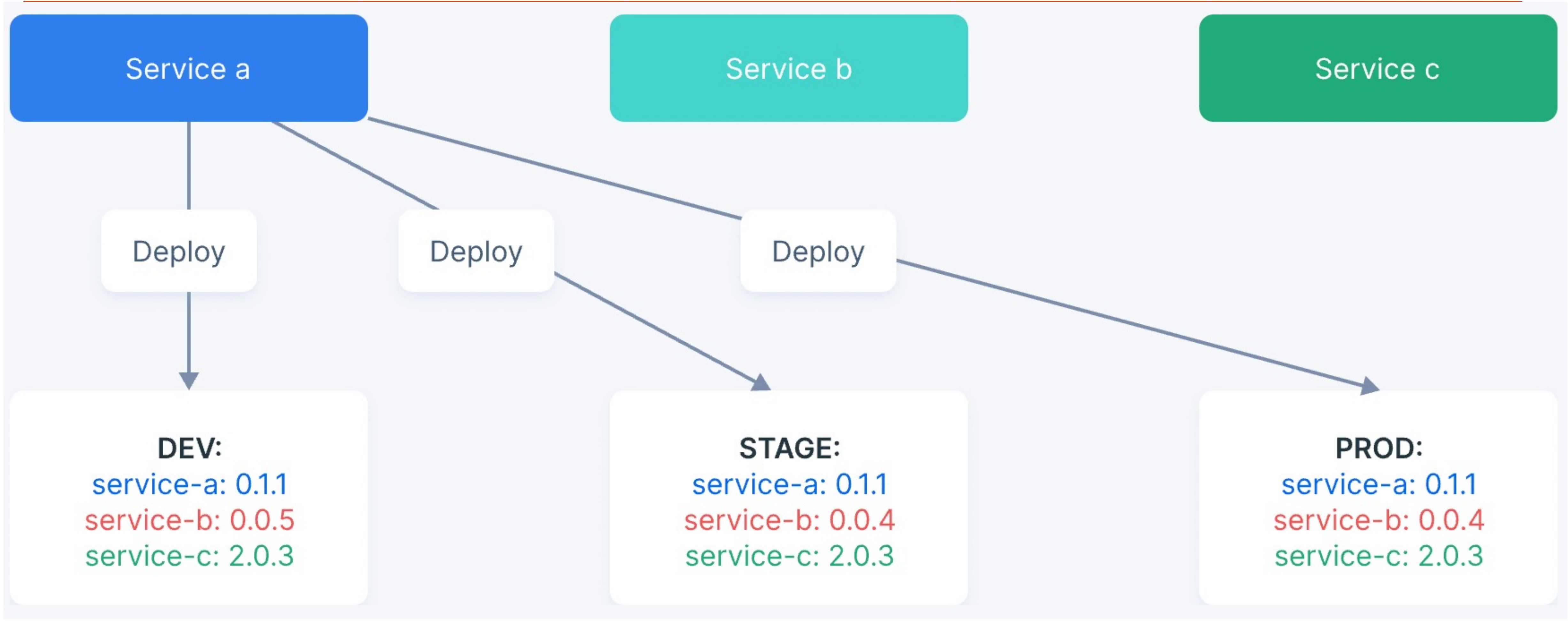
Architecture View



Challenges

- Inter-process communication (via network)
- Distributed transactions
- Large number of services
- Require more automation

Deployment



Service a
Service b
Service c
Deploy
Deploy
Deploy
DEV:
service-a: 0.1.1
service-b: 0.0.5
service-c: 2.0.3
STAGE:
service-a: 0.1.1
service-b: 0.0.4
service-c: 2.0.3
PROD:
service-a: 0.1.1
service-b: 0.0.4
service-c: 2.0.3

Ref

1. https://martinfowler.com/articles/microservices.html

Q&A