# Internet of Things and Services

## Service-oriented architectures

# Microservices

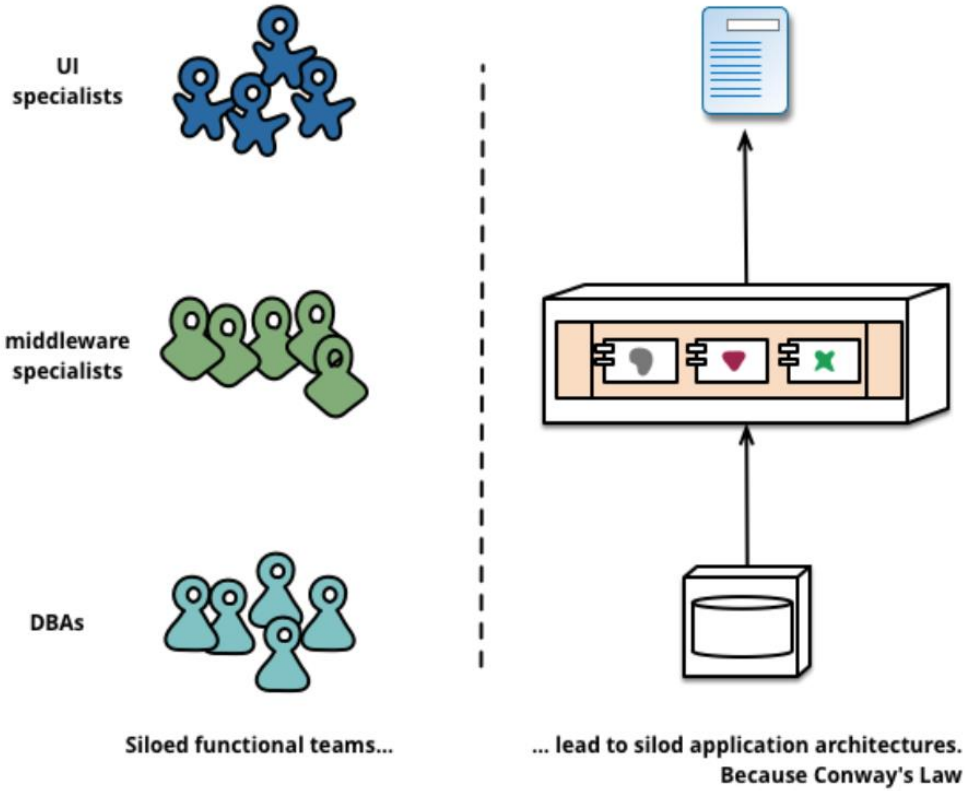
Department of Computer Science
Faculty of Electronic Engineering, University of Niš

# Distributed systems & IoT

- Distributed architecture
  - Multi tier architecture with Web front-ends
  - Built collaboratively by several development teams
  - With traffic load that requires horizontal scaling (i.e. load balancing across multiple copies of the system)
- Problem: Such systems are often built as a **software monolith** with coarse-grained services within SOA
  - One build and deployment unit
  - One code base
  - One technology stack (Linux, JVM, Tomcat, Java Spring,…)
- Benefits
  - Development: One unit of access for coding, building, and deploying
  - Scaling: Just run multiple copies behind a load balancer

# Software monolith

- "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure" - Melvyn Conway

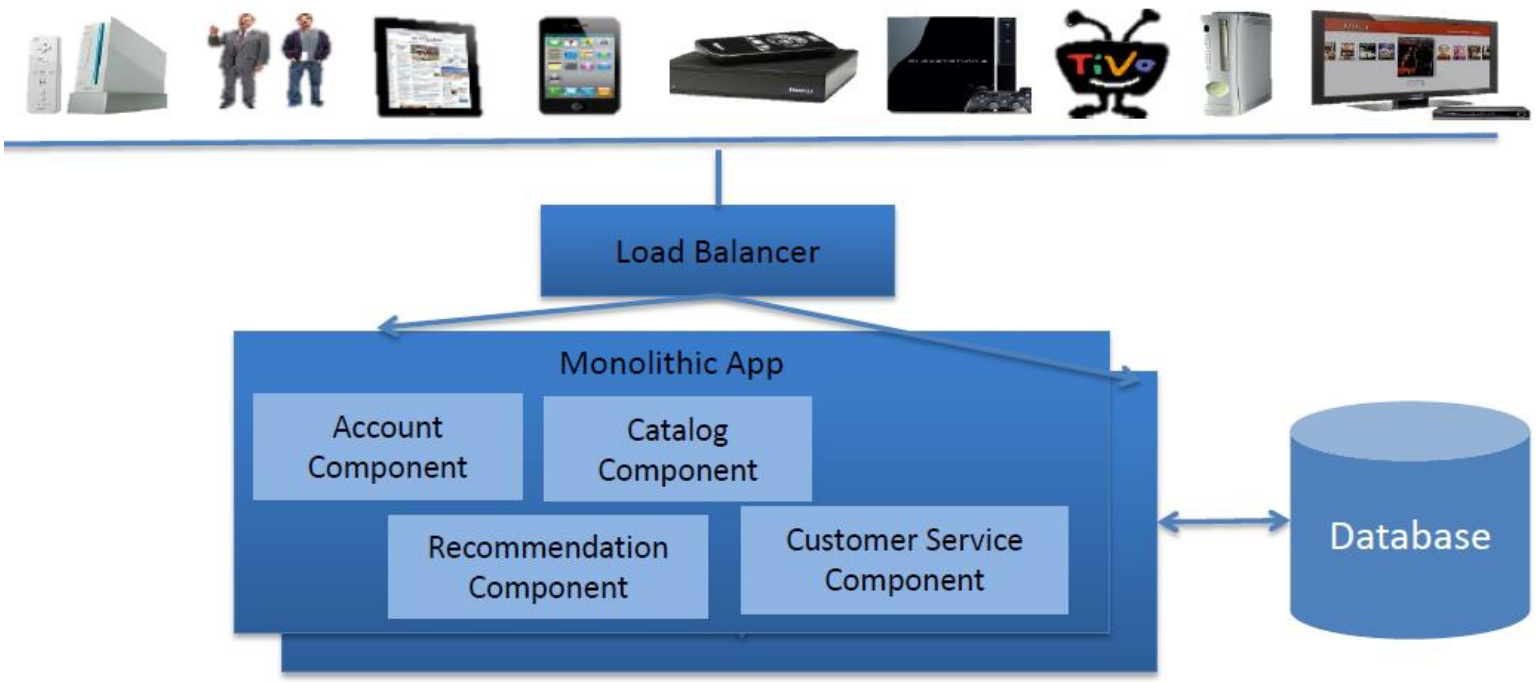


# Software monolith problems

- Huge and intimidating code base for developers
- Development tools get overburdened
  - Refactorings take minutes
  - Builds take hours
  - Testing in continuous integration takes days
- Scaling is limited
  - Running a copy of the whole system is resource intense
  - It doesn't scale with the data volume out of the box
- Deployment frequency is limited
  - Redeploying means halting the whole system
  - Redeployments will fail and increase the perceived risk of deployment

# Monolith architecture

- Monolithic applications can be successful, but increasingly people are feeling frustrations with them, especially as more applications are being deployed to the cloud



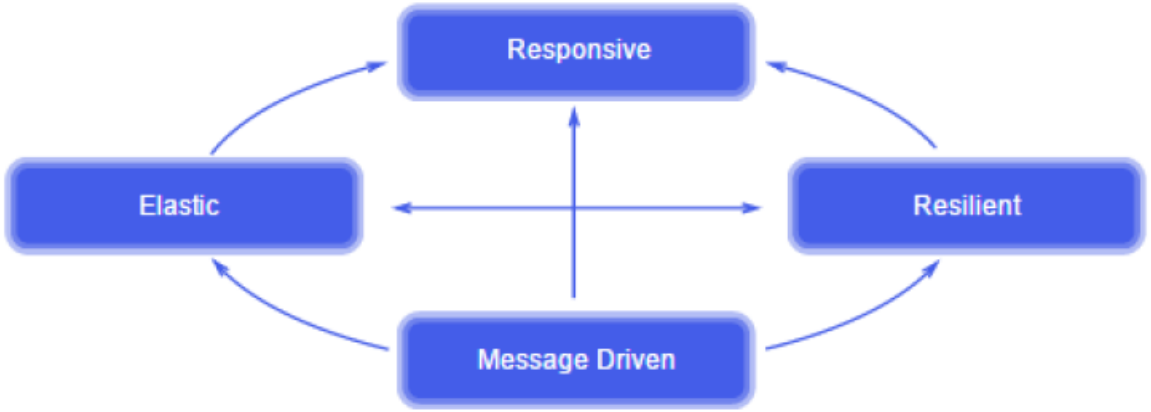
# Trends in Software Development

Platform as a Service

Autonomous teams

Continuous Delivery

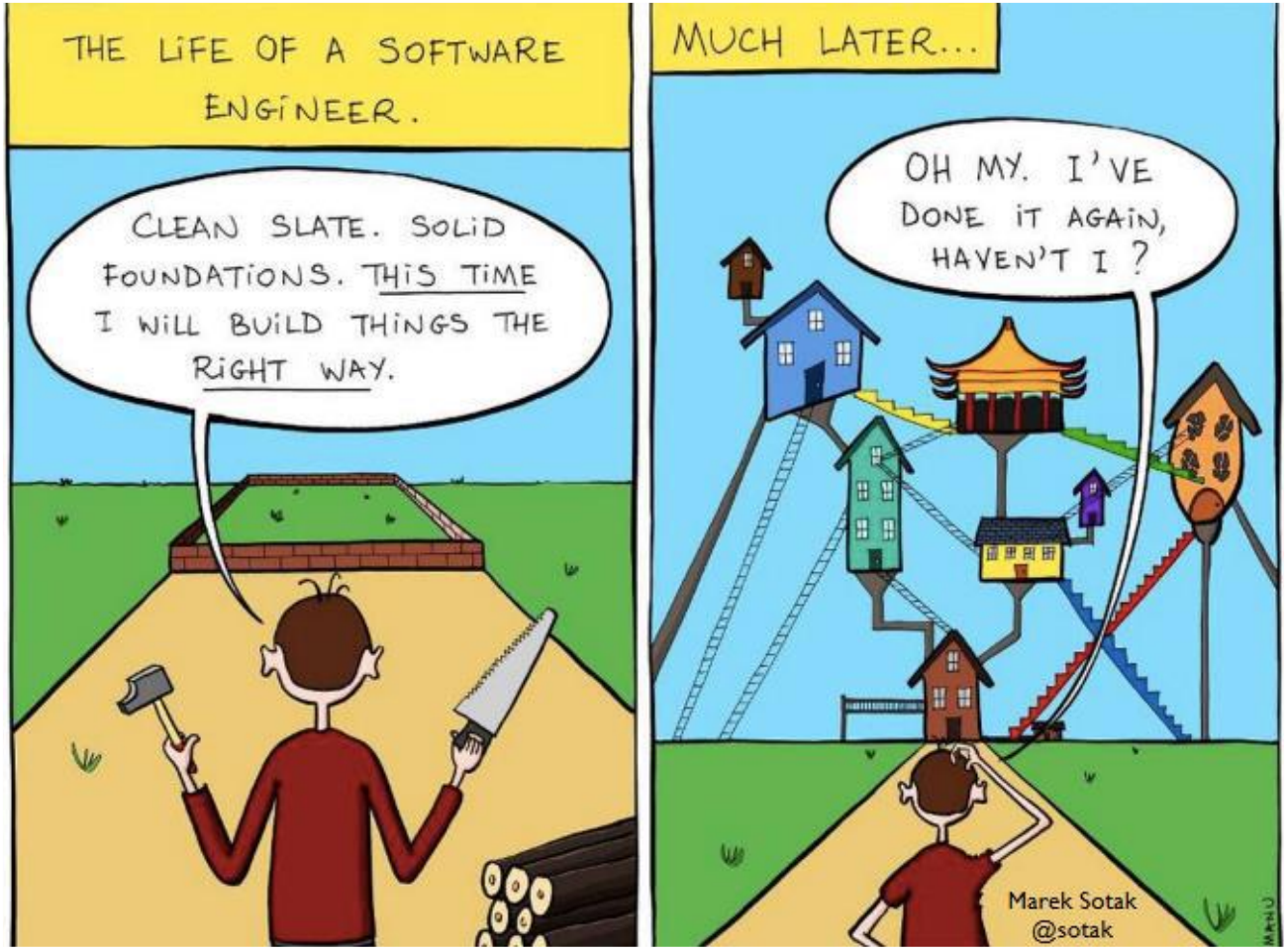
Agile Organization




Reactive manifesto

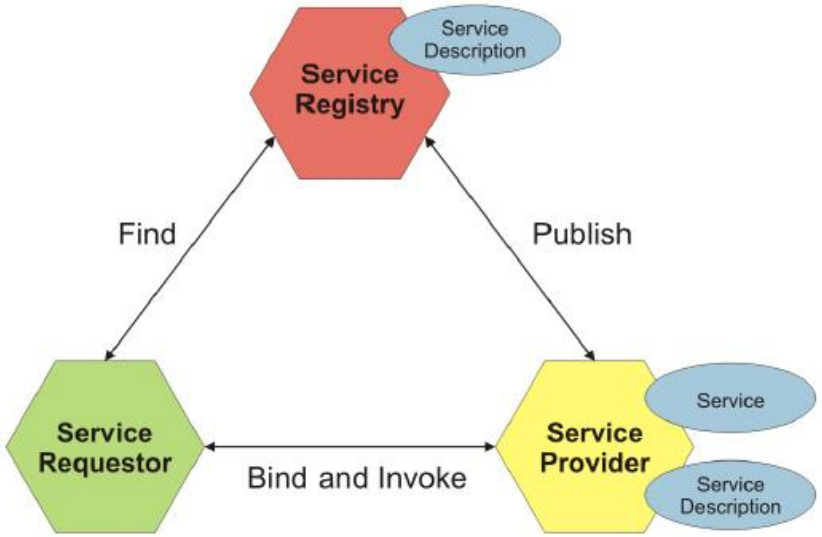
# The new architectural pattern

- Yesterday's best practice is tomorrow's anti-pattern.

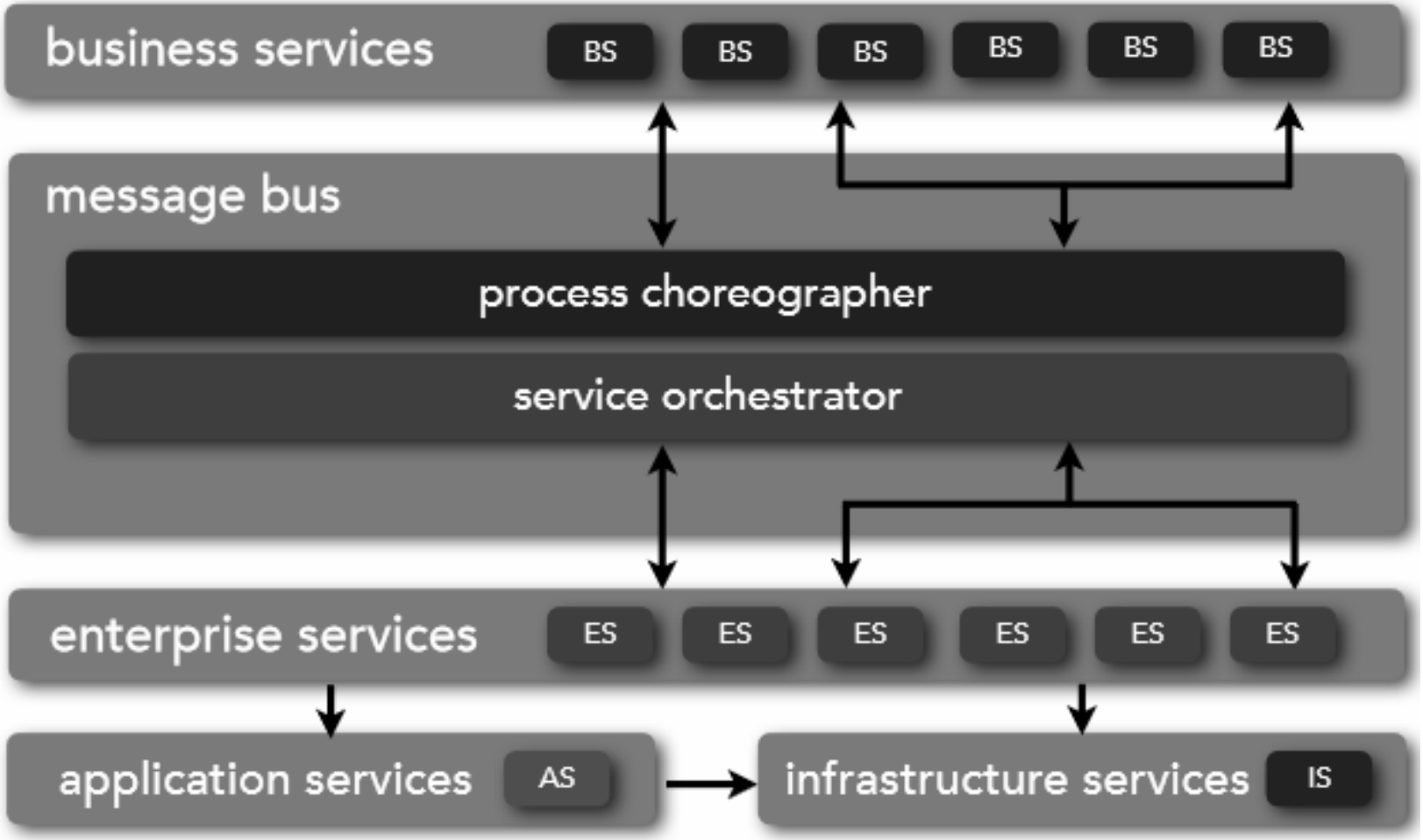


# Service Oriented Architecture

- Service-Oriented Architecture (SOA) is a software architecture pattern where services are provided to the other components by application components, through a communication protocol over a network.
- There are three roles in SOA: service provider, service service registry (broker, repository), and service requestor/consumer.



# Service Oriented Architecture



# From SOA to microservices

- Applications and teams need to grow beyond the limits imposed by monoliths and layered SOA systems, and they do in an uncontrolled way.
- Large companies end up with landscapes of layered systems that often interoperate in undocumented ways.
- These landscapes then often break in unexpected ways.
- How can a company grow and still have a working IT architecture and vision?
- Observing and documenting successful companies (e.g. Amazon, Netflix) lead to the definition of microservice architecture principles.

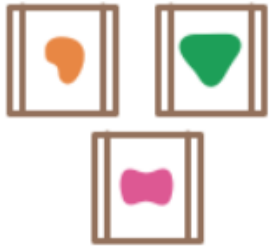
# SOA vs. microservices

- Heavyweight vs. lightweight technologies
- SOA tends to rely strongly on heavyweight middleware (e.g., ESB), while microservices rely on lightweight technologies
- Protocol families
  - SOA is often associated with web services protocols
    - SOAP, WSDL, and WS-* family of standards
  - Microservices typically rely on REST and HTTP
- Views
  - SOA mostly viewed as integration solution
  - Microservices are typically applied to build individual software applications

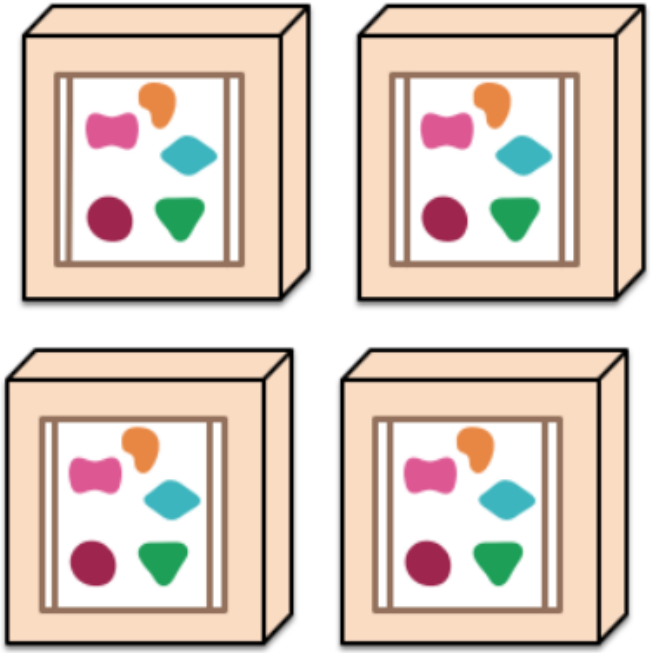
# Monolith vs Microservices

*A monolithic application puts all its functionality into a single process...*

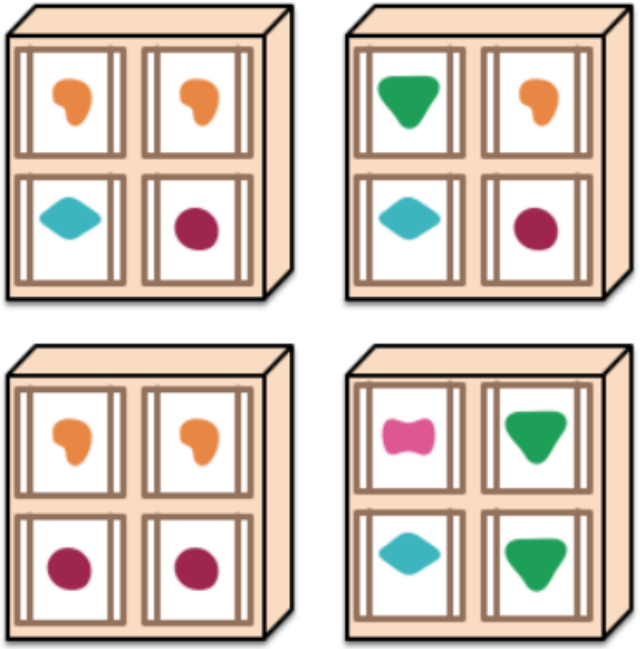
*A microservices architecture puts each element of functionality into a separate service...*



*... and scales by replicating the monolith on multiple servers*

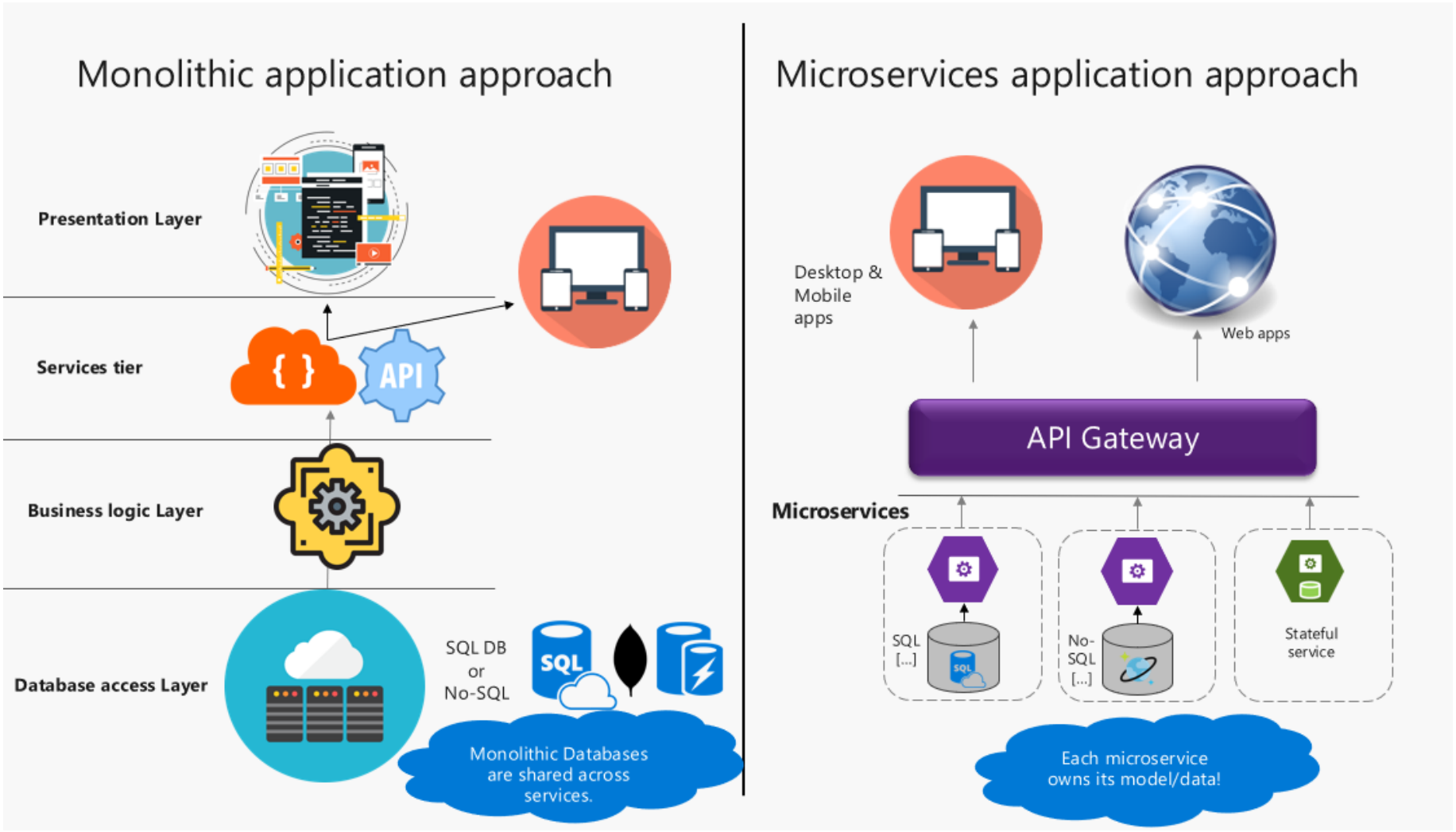


*... and scales by distributing these services across servers, replicating as needed.*



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

# Monolith vs Microservices



# Microservice architecture

- Microservice architectural 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 built 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 (2014) https://martinfowler.com/microservices/

# Microservices

- A "new" emerging architectural style for distributed applications that structures an application as a collection of loosely coupled services
- Not so new: deriving from **SOA**
  - But with some significant differences
- Address how to build, manage, and evolve architectures out of small, self-contained units
  - Modularization: decompose app into a set of independently deployable services, that are loosely coupled and cooperating and can be rapidly deployed and scaled
  - Services equipped with dedicated memory persistence tools (e.g., databases and NoSQL data stores)

# Microservices

- The term "micro" refers to the sizing: a microservice must be manageable by a single development team (5-9 developers).
- Functional system decomposition means vertical slicing (in contrast to horizontal slicing through layers).
- Independent deployability implies no shared state and inter-process communication (often via HTTP RESTish interfaces).
- Autonomous, small and focused on doing one thing well.
- "Loosely coupled service-oriented architecture with bounded contexts" - Adrian Cockcroft (Netflix)
- "SOA done right" - Anonymous

# Microservice characteristics

- Each microservice is functionally complete with:
  - Resource representation
  - Data management
- Each microservice handles one resource (or verb), e.g. Clients, Shop Items, Carts, Checkout,…
- Microservices are fun-sized services, as in "still fun to develop and deploy"
- Independent deployability is a key
- It enables separation and independent evolution of
  - code base
  - technology stacks
  - scaling
  - and features, too

# Independent Codebase

- Each service has its own software repository.
- Codebase is maintainable for developers – it fits into their brain.
- Tools work fast – building, testing, refactoring code takes seconds.
- Service startup only takes seconds.
- No accidental cross-dependencies between code bases.

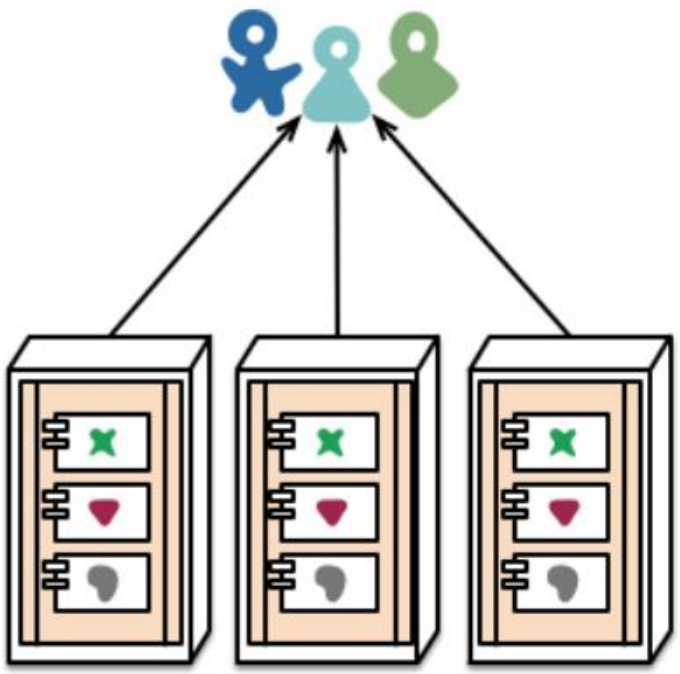
# Independent technology stack

- Each service is implemented on its own technology stacks.
- The technology stack can be selected to fit the task best.
- Teams can also experiment with new technologies within a single microservice.
- No system-wide standardized technology stack also means
  - No struggle to get your technology introduced to the canon
  - No piggy-pack dependencies to unnecessary technologies or libraries
  - It's only your own dependency hell you need to struggle with
- Selected technology stacks are often very lightweight
  - A microservice is often just a single process that is started via command line, and not code and configuration that is deployed to a container.

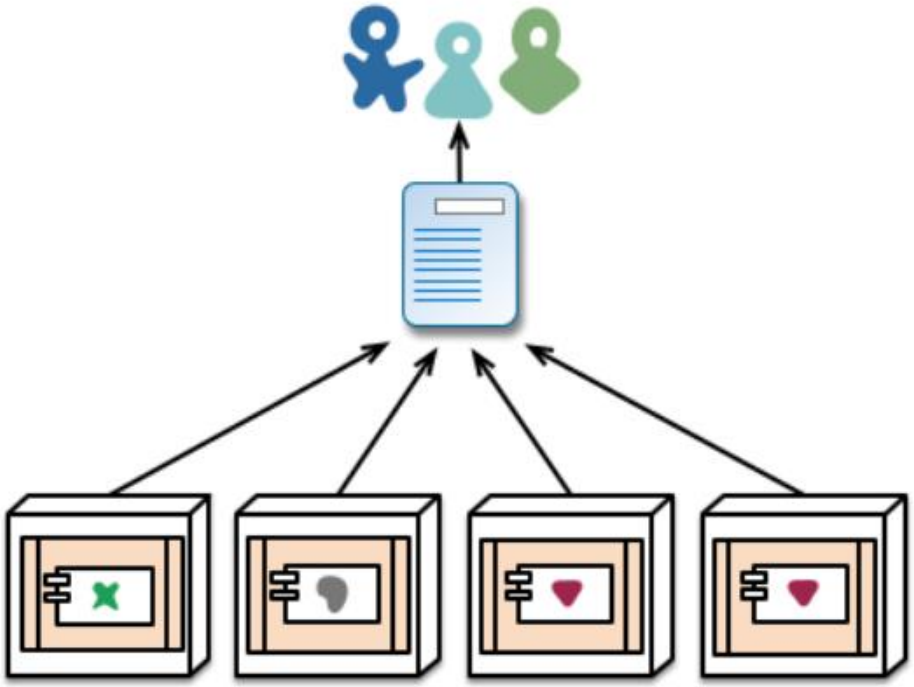
# Independent Scaling

- Each microservice can be scaled independently
- Identified bottlenecks can be addressed directly
- Data sharding can be applied to microservices as needed
- Parts of the system that do not represent bottlenecks can remain simple and un-scaled
- Microservices can be extended without affecting other services
  - For example, you can deploy a new version of (a part of) the UI without re-deploying the whole system
  - You can also go so far as to replace the service by a complete rewrite
  - But you have to ensure that the service interface remains stable

# Independent Processes



monolith - multiple modules in the same process



microservices - modules running in different processes

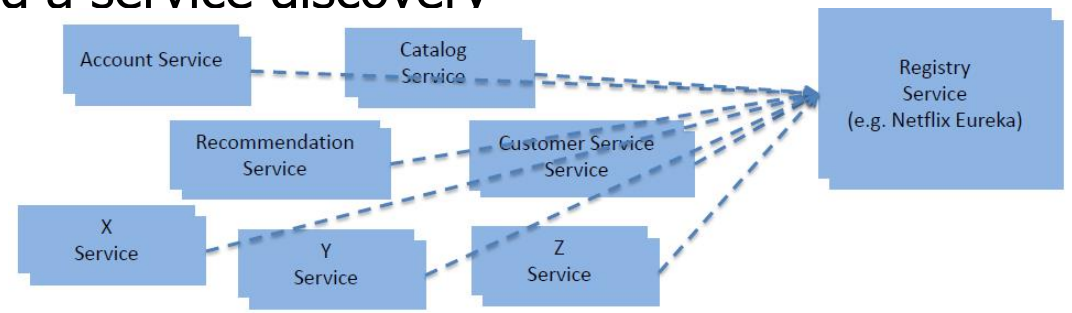
# Language Agnostic APIs

- "Language agnostic" describes a software development paradigm  where a particular language is chosen because of its appropriateness for a particular task (taking into consideration all factors, including ecosystem, developer skill-sets, performance, etc.), and not purely because of the skill-set available within a development team.


**Microservices**

# Microservices and scalability

- How to achieve service scalability?
  - Use multiple instances of the same service and load balance request across multiple instances
- How to improve service scalability?
  - State is complex to manage and scale
  - Stateless services scale better and faster than stateful services
- We also need service discovery
  - Service instances have dynamically assigned network locations and their set changes dynamically because of autoscaling, failures, and upgrades: we need a service discoverv
  - Service Registry



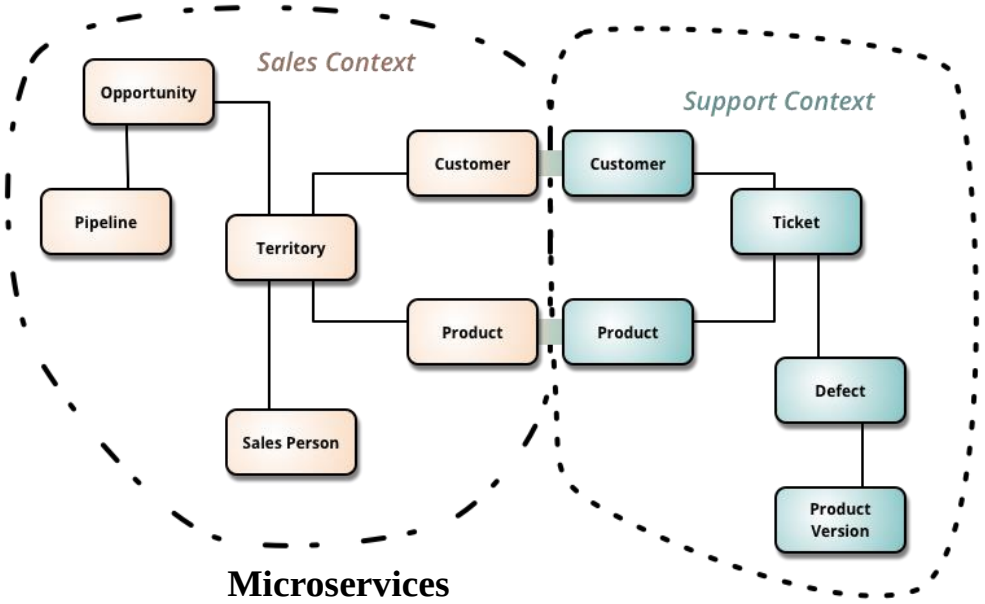

**Microservices**

# Microservices and decomposition

- How to decompose the application into microservices?
- Mostly an art, no winner strategy but rather a number of strategies:
  - Decompose by business capability and define services corresponding to business capabilities
  - Decompose by domain-driven design (DDD) subdomain
  - Decompose by use case and define services that are responsible for particular actions
  - Decompose by nouns or resources and define a service that is responsible for all operations on entities/resources of a given type

# Bounded Context

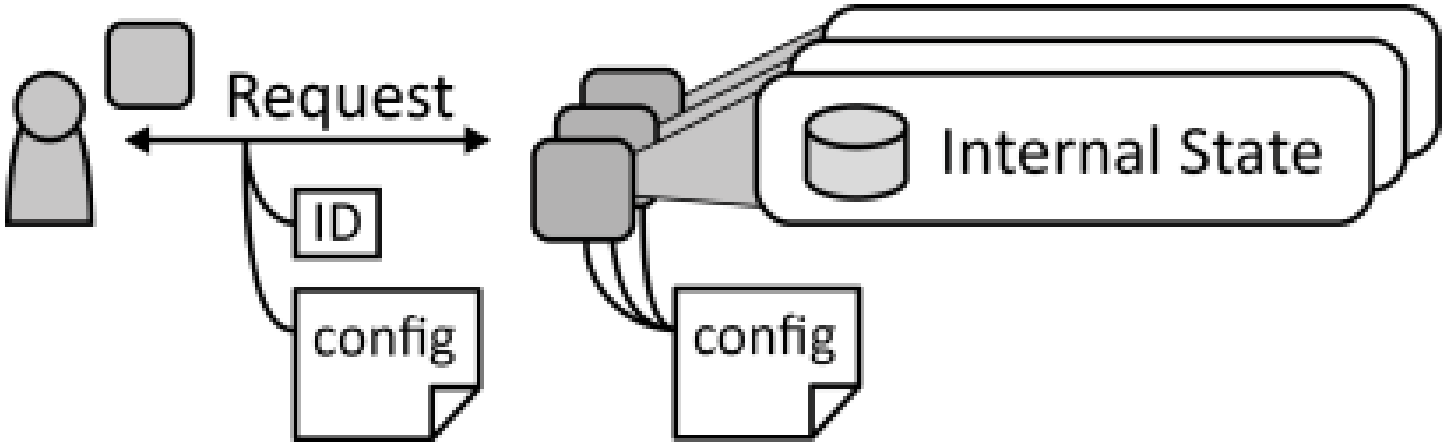
- Bounded Context is a central pattern in Domain-Driven Design.
- It is the focus of DDD's strategic design section which is all about dealing with large models and teams.
- DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.




**Microservices**

# Stateless vs. stateful service

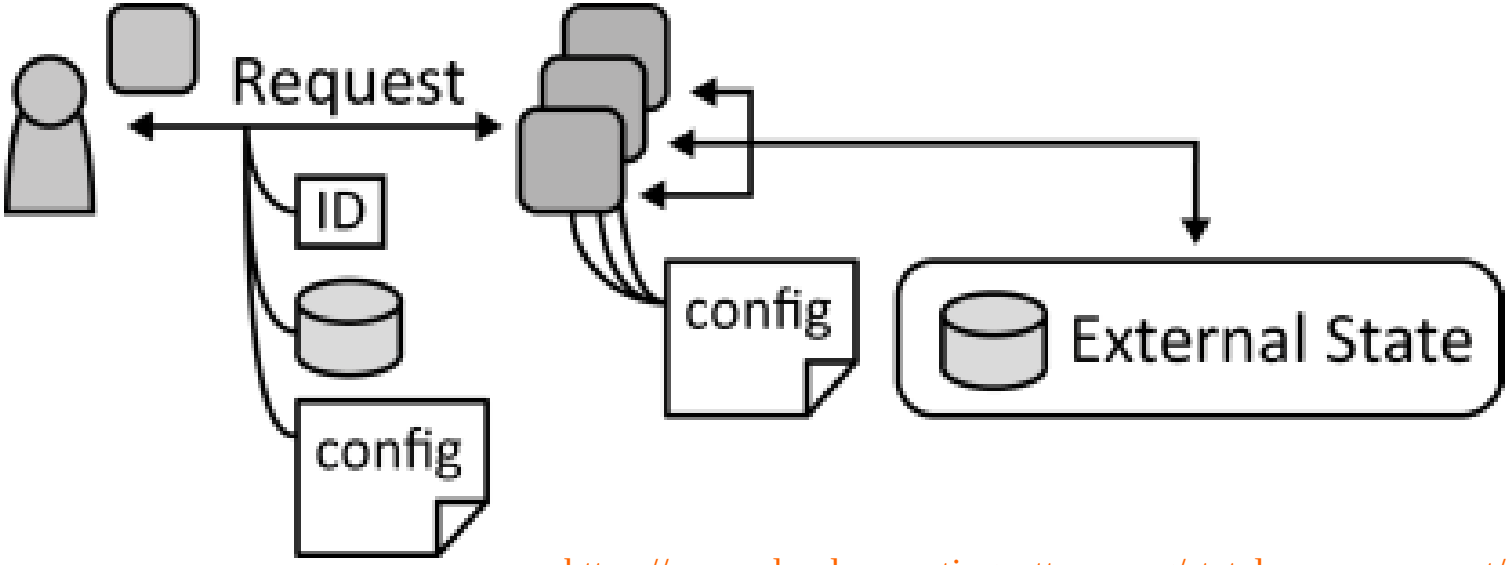
- Stateful service: multiple instances of a scaled-out service need to synchronize their internal state to provide a unified behavior
- Issue: how can a stateful service that is scaled-out maintain a synchronized internal state?




https://www.cloudcomputingpatterns.org/stateful_component/

# Stateless vs. stateful service

- Stateless service: state is handled external of service to ease its scaling out and to make the application more tolerant to service failures

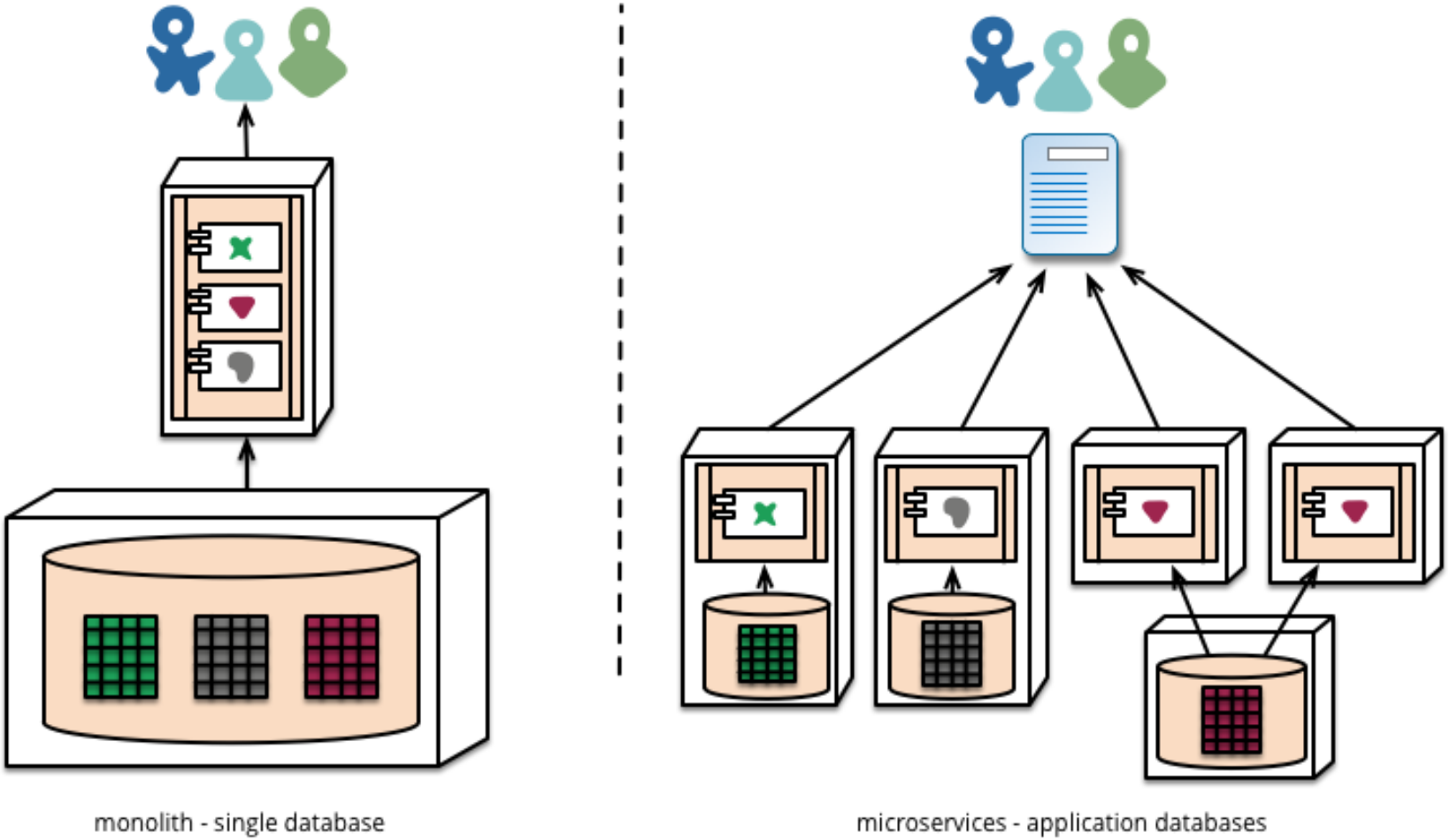



https://www.cloudcomputingpatterns.org/stateless_component/

# Decentralized data management

- While monolithic applications prefer a single logical database for persistent data, enterprises often prefer a single database across a range of applications
- Each service can choose the persistence solution that fits best its
  - Data access patterns
  - Scaling and data sharding requirements
- Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems -an approach called Polyglot Persistence.
- You can use polyglot persistence in a monolith, but it appears more frequently with microservices.

# Polyglot Persistence



monolith - single database

microservices - application databases

# ACID vs BASE

**ACID**

- **A**tomic
- **C**onsistent
- **I**solated
- **D**urable

**BASE**

- **B**asic **A**vailability
- **S**oft-state
- **E**ventual Consistency

**As your system becomes more distributed, prefer BASE to ACID...**

...because CAP Theorem

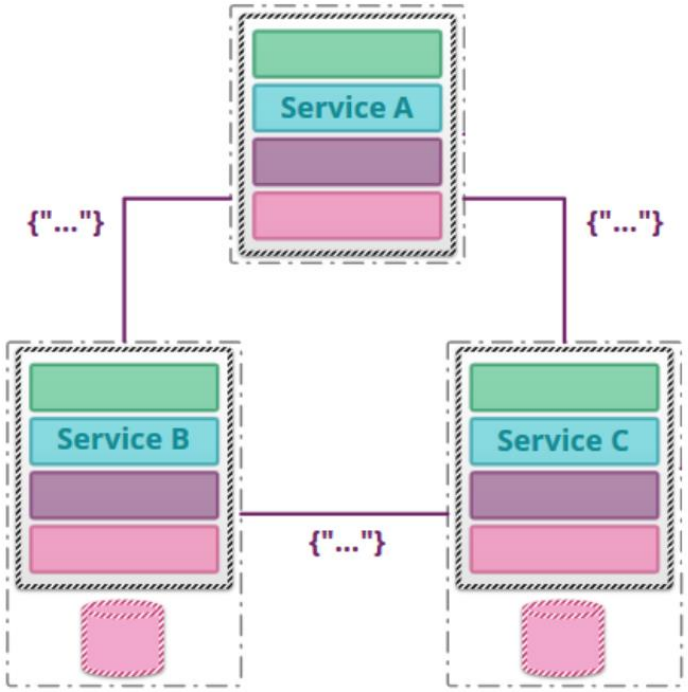
# Microservice communication

- Communication between microservices is often standardized using
  - HTTP(S) –battle-tested and broadly available transport protocol
  - REST –uniform interfaces on data as resources with known manipulation means
  - JSON –simple data representation format
  - gRPC & ProtoBuf
- REST and JSON are convenient because they simplify interface evolution


**Microservices**

# Microservice integration

- Let's consider two issues related to the integration and collaboration of microservices
  - Synchronous vs. asynchronous communication
  - Orchestration vs. choreography



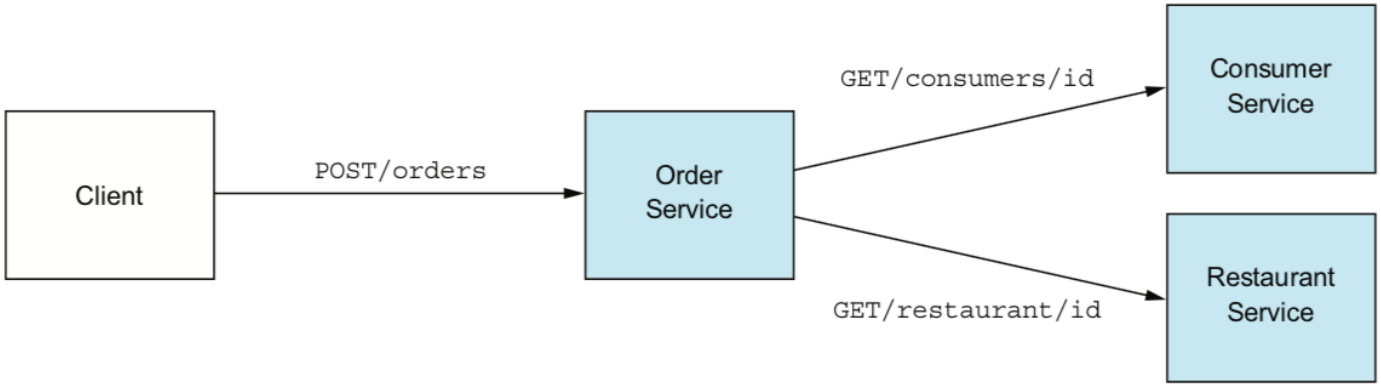

**Microservices**

# Synchronous vs. asynchronous

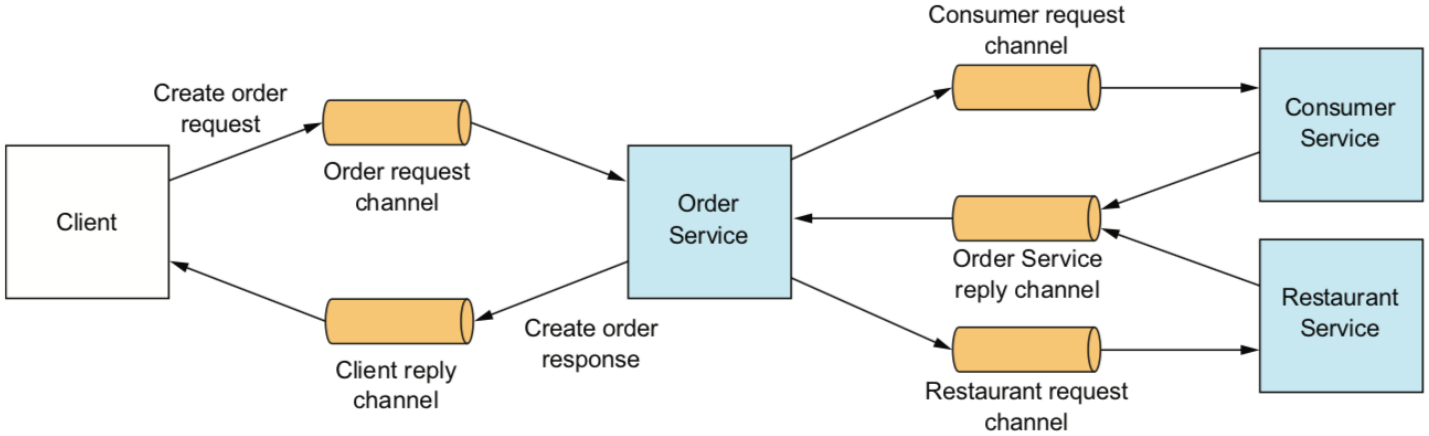
- Should communication be synchronous or asynchronous?
  - Synchronous: request/response style of communication
  - Asynchronous: event-driven style of communication
- Synchronous communication
  - Synchronous request/response-based communication mechanisms, such as HTTP-based REST or gRPC
- Asynchronous communication
  - Asynchronous, message-based communication mechanisms such as pub-sub systems, message queues and related protocols
  - Interaction style can be one-to-one or one-to-many

# Synchronous vs. asynchronous

- Synchronous communication reduces availability




- Asynchronous communication improves performance



# Orchestration and choreography

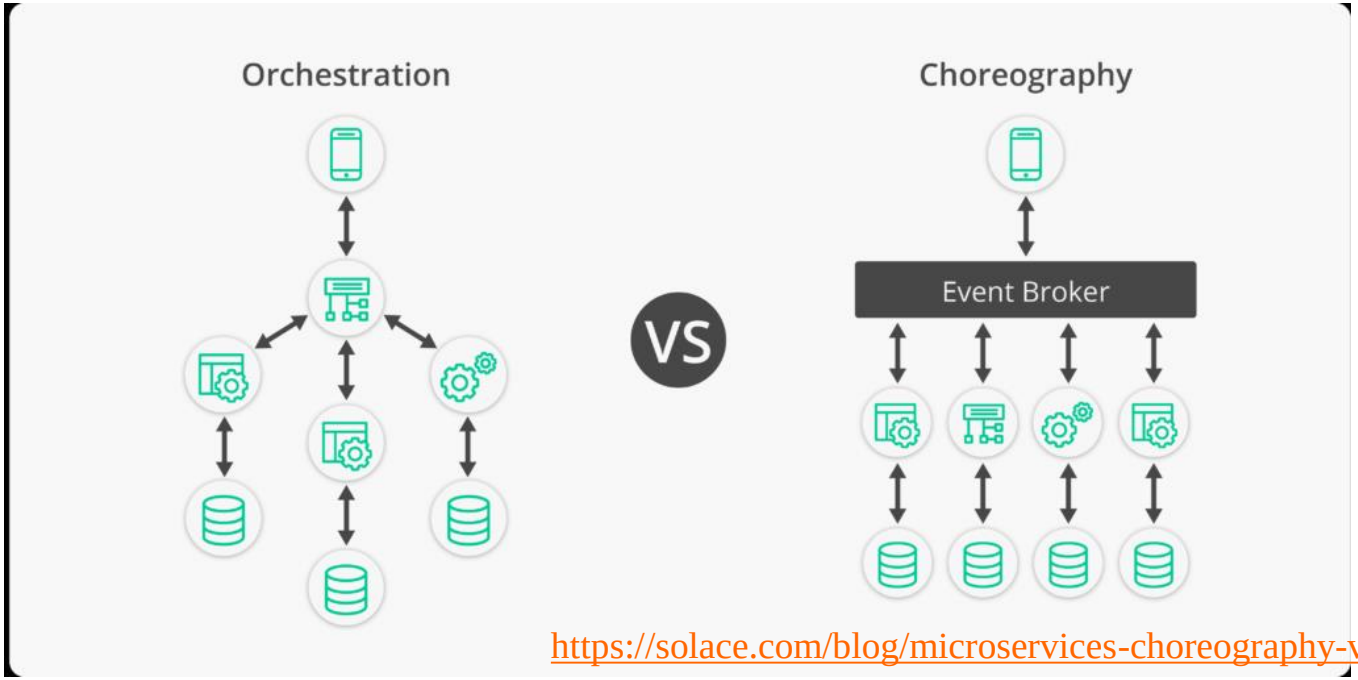
- Microservices can interact among them following two patterns:
  - Orchestration
  - Choreography
- **Orchestration**: centralized approach
  - A single centralized process (*orchestrator*) coordinates the interaction among different services
  - The orchestrator is responsible for invoking and combining the services, which can be unaware of the composition
- **Choreography**: decentralized approach
  - A global description of the participating services, which is defined by exchange of messages, rules of interaction and agreements between two or more endpoints
  - Services can exchange messages directly

# Orchestration vs choreography

- Orchestration is simpler and more popular, but
  - Single point of failure and performance bottleneck
  - Tight coupling
  - Higher network traffic and latency
- Choreography has lower coupling, less operational complexity and increased flexibility and ease of changing, but
  - Services need to know about each other's locations
  - Extra work to monitor and track services
  - Implementing mechanisms, such as guaranteed delivery, is more challenging

# Orchestration and choreography

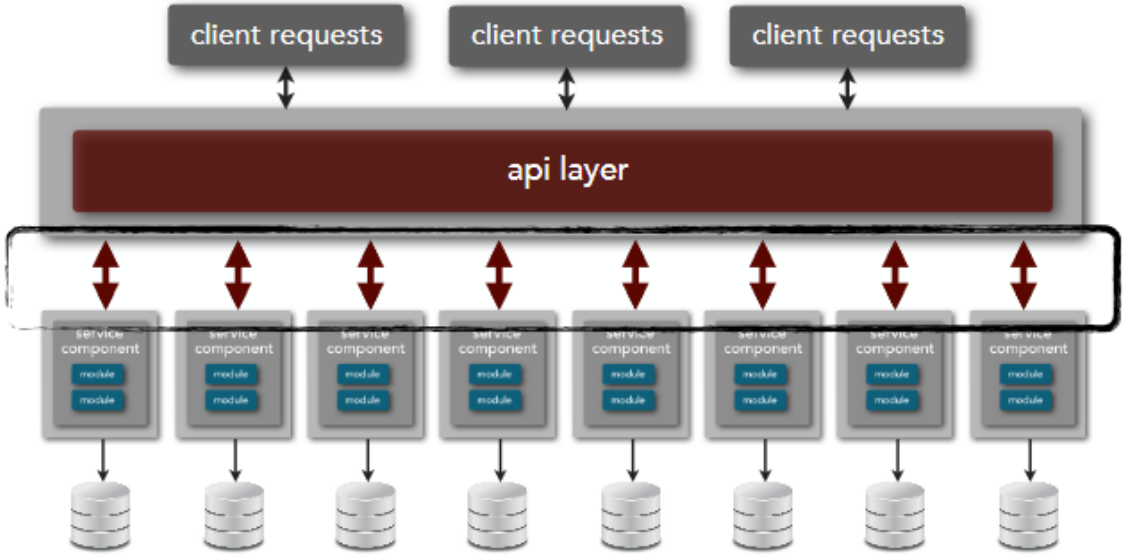
- Choreography vs. orchestration is NOT about choosing the right approach. In real life, you need to balance both, so it is about choosing wisely on a case-by-case basis.
  - Overall choreography and local orchestration




https://solace.com/blog/microservices-choreography-vs-orchestration/

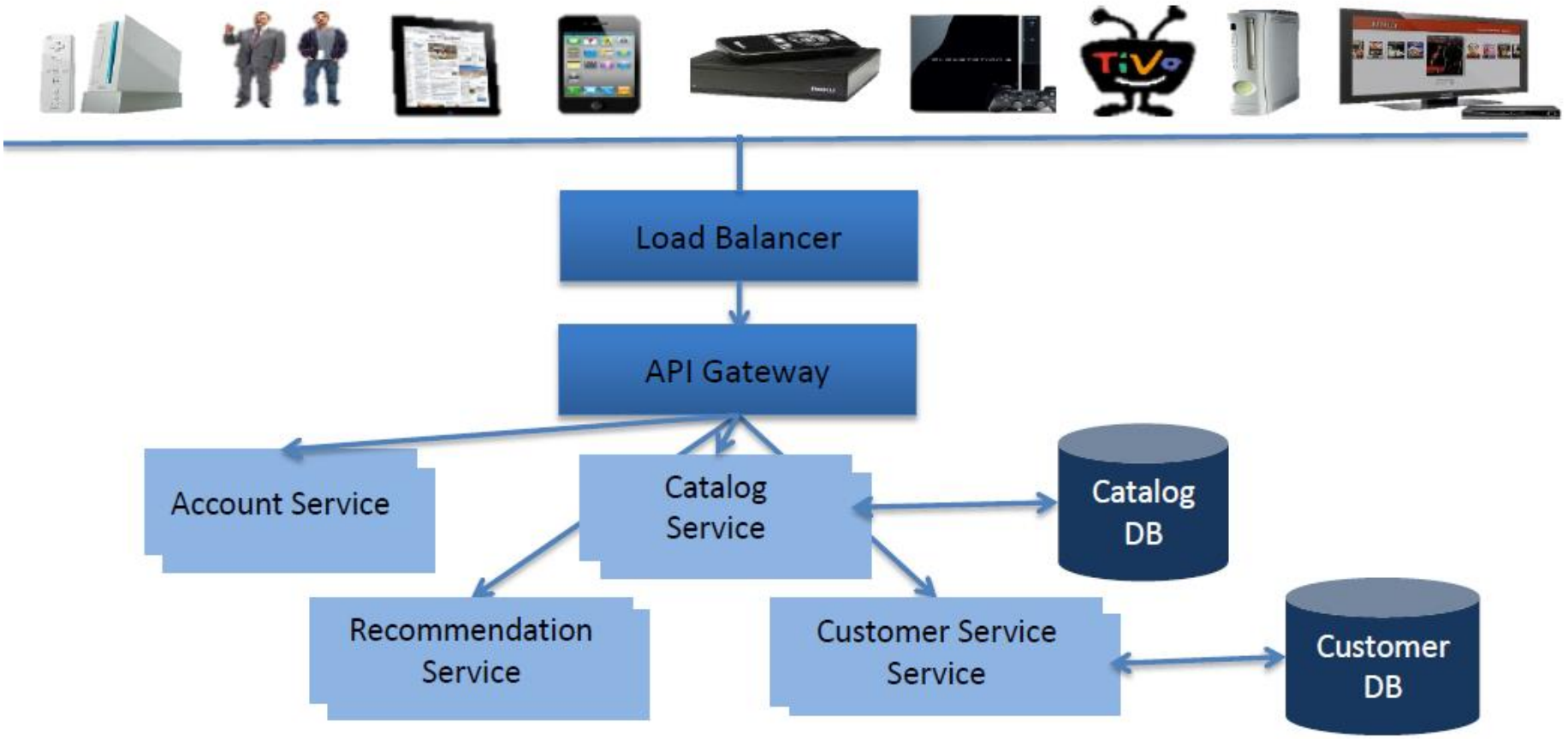
# Microservice architecture

- Features:
  - Distributed architecture
  - Separately deployed components
  - Service components
  - Bounded context
  - Service orchestration




**Microservices**

# Microservice architecture (2)



# Microservice architecture
## Characteristics

- Componentization via Services
- Organized around Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure
- Evolutionary Design

# When to use Microservices?

- Microservices provide **benefits**…
  - Strong Module Boundaries: Microservices reinforce modular structure, which is particularly important for larger teams.
  - Independent Deployment: Simple services are easier to deploy, and since they are autonomous, are less likely to cause system failures when they go wrong.
  - Technology Diversity: With microservices you can mix multiple languages, development frameworks and data-storage technologies.
- …but come with **costs**
  - Distribution: Distributed systems are harder to program, since remote calls are slow and are always at risk of failure.
  - Eventual Consistency: Maintaining strong consistency is extremely difficult for a distributed system, which means everyone has to manage eventual consistency.
  - Operational Complexity: You need a mature operations team to manage lots of services, which are being redeployed regularly.

# Advantages of Microservices

- Each micro service is small and focused on a specific feature/business requirement.
- Microservice can be developed independently by small team of developers (normally 2 to 5 developers).
- Microservice is loosely coupled, means services are independent, in terms of development and deployment both.
- Microservice can be developed using different programming languages.
- Microservice allows easy and flexible way to integrate automatic deployment with Continuous Integration tools (for e.g. Jenkins, Hudson, bamboo etc..).
- The productivity of a new team member will be quick enough.
- Microservice is easy to understand, modify and maintain for a developer because separation of code, small team and focused work.

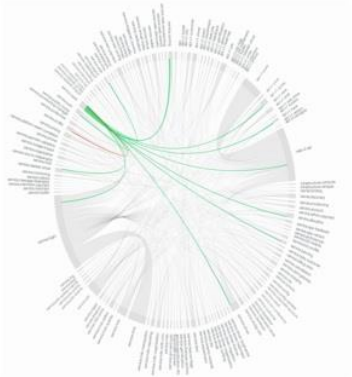
# Disadvantages of Microservices

- Microservice architecture brings a lot of operations overhead.
- DevOps skills required
- Duplication of efforts
- Distributed system is complicated to manage .
- Default to trace problem because of distributed deployment.
- Complicated to manage whole products when number of services increases.

# Who use Microservices?

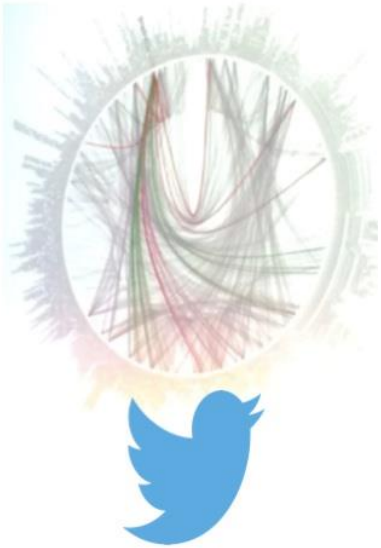
- Most large scale companies including Twitter, Netflix, Amazon and eBay have evolved from a monolithic architecture to a microservice architecture.
- *Death stars*

450 microservices




500+ microservices


500+ microservices



Source:
Netflix: http://www.slideshare.net/BruceWong3/the-case-for-chaos
Twitter: https://twitter.com/adrianco/status/441883572618948608
Hail-o: https://sudo.hailoapp.com/services/2015/03/09/journey-into-a-microservice-world-part-3/

# References


- Martin Fowler & James Lewis
  - *Microservices* https://martinfowler.com/articles/microservices.html
  - *Microservices Guide* http://martinfowler.com/microservices/
- Sam Newman, *Building Microservices: Designing Fine-Grained Systems*, O'Reilly Media; 1st edition, 2015
- Sam Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*, O'Reilly Media Inc. 2020.