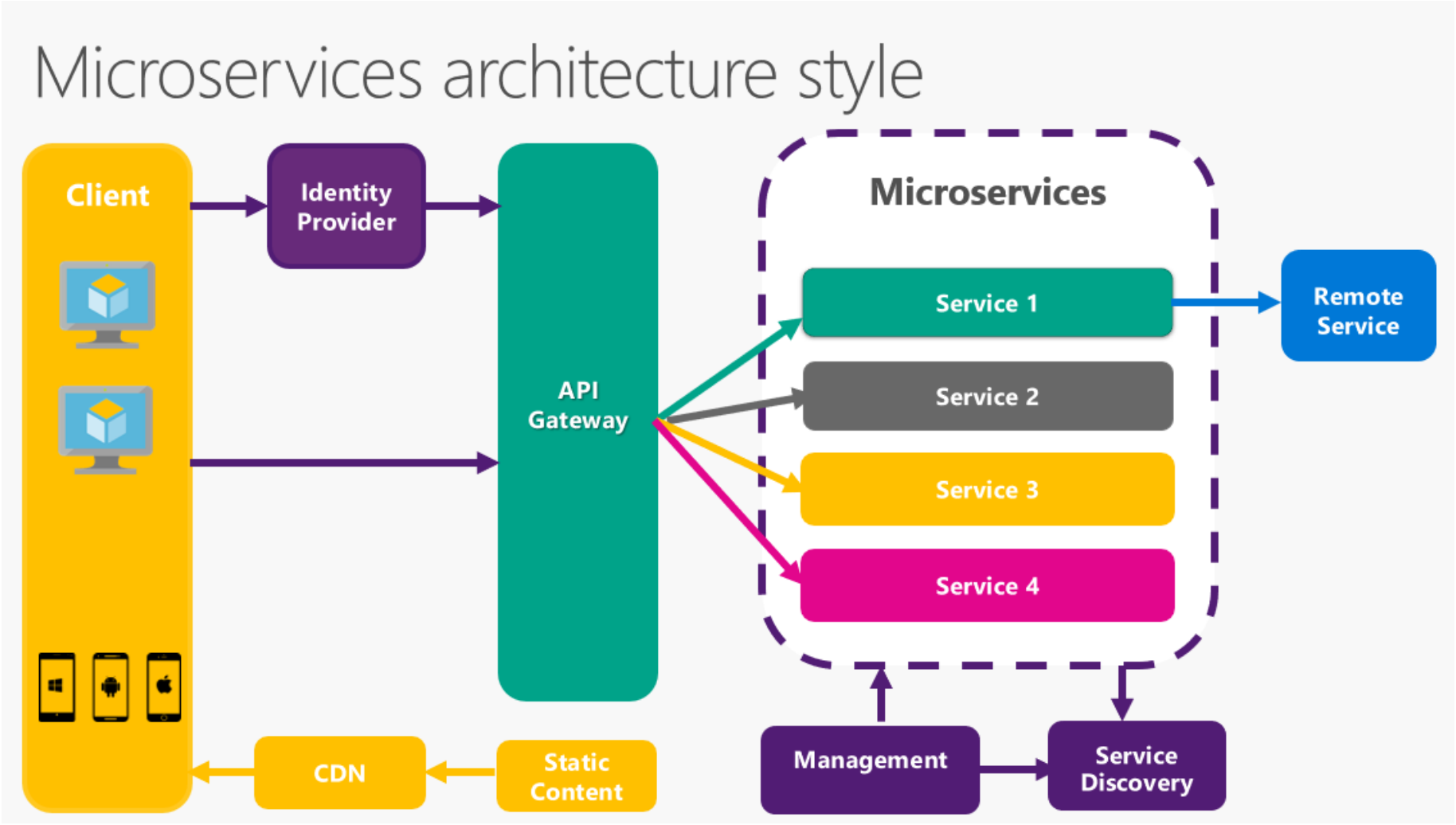
Internet of Things and Services

Service-oriented architectures

Microservice architecture and patterns

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

Microservices architecture style



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

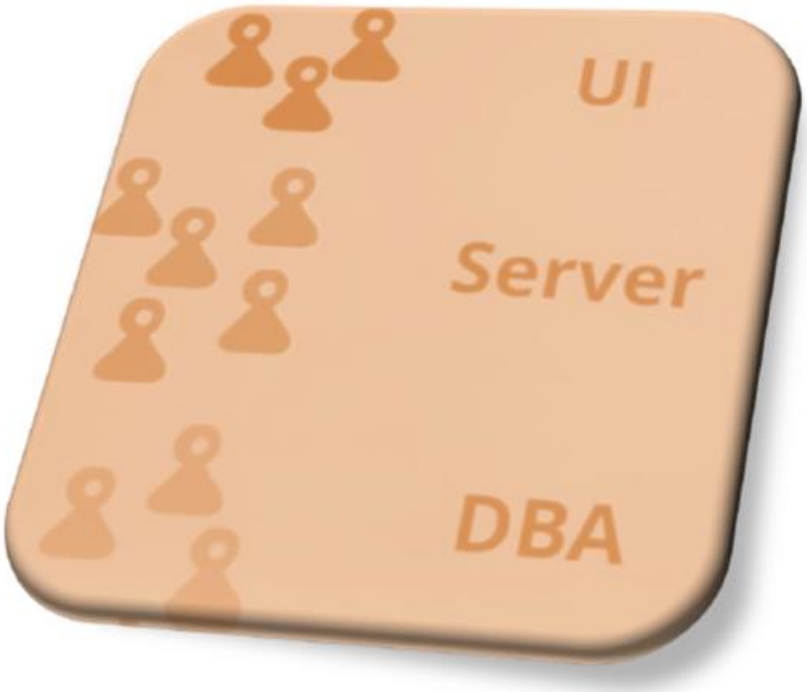
Componentization via Services

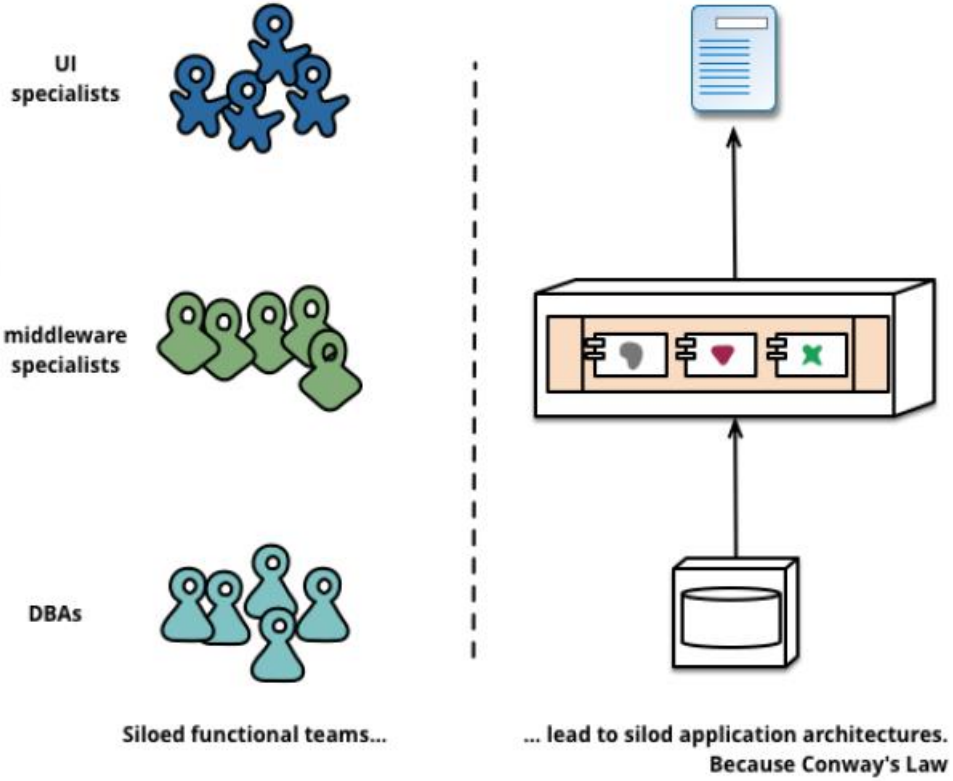
- What is a component? We think about how it relates to software instead of software itself
  - Independently Replaceable, Independently Updatable
- How can we do this: Libraries or Services? → We do with services
- Interaction mode: share-nothing, cross-process communication
- Independently deployable (with all the benefits), Explicit, REST-based public interface
- Sized and designed for replaceability
- Downsides
  - Communication is more expensive than in-process
  - Interfaces need to be coarser-grained
  - Re-allocation of responsibilities between services is harder

Organized around business capabilities (1)

- The microservice approach to division is different, splitting up into services organized around business capability.
- Such services take a broad-stack implementation of software for that business area, including user-interface, persistent storage, and any external collaborations.
- Consequently, the teams are cross-functional, including the full range of skills required for the development: user-experience, database, and project management.
- Line of separation is along functional boundaries, not along tiers.

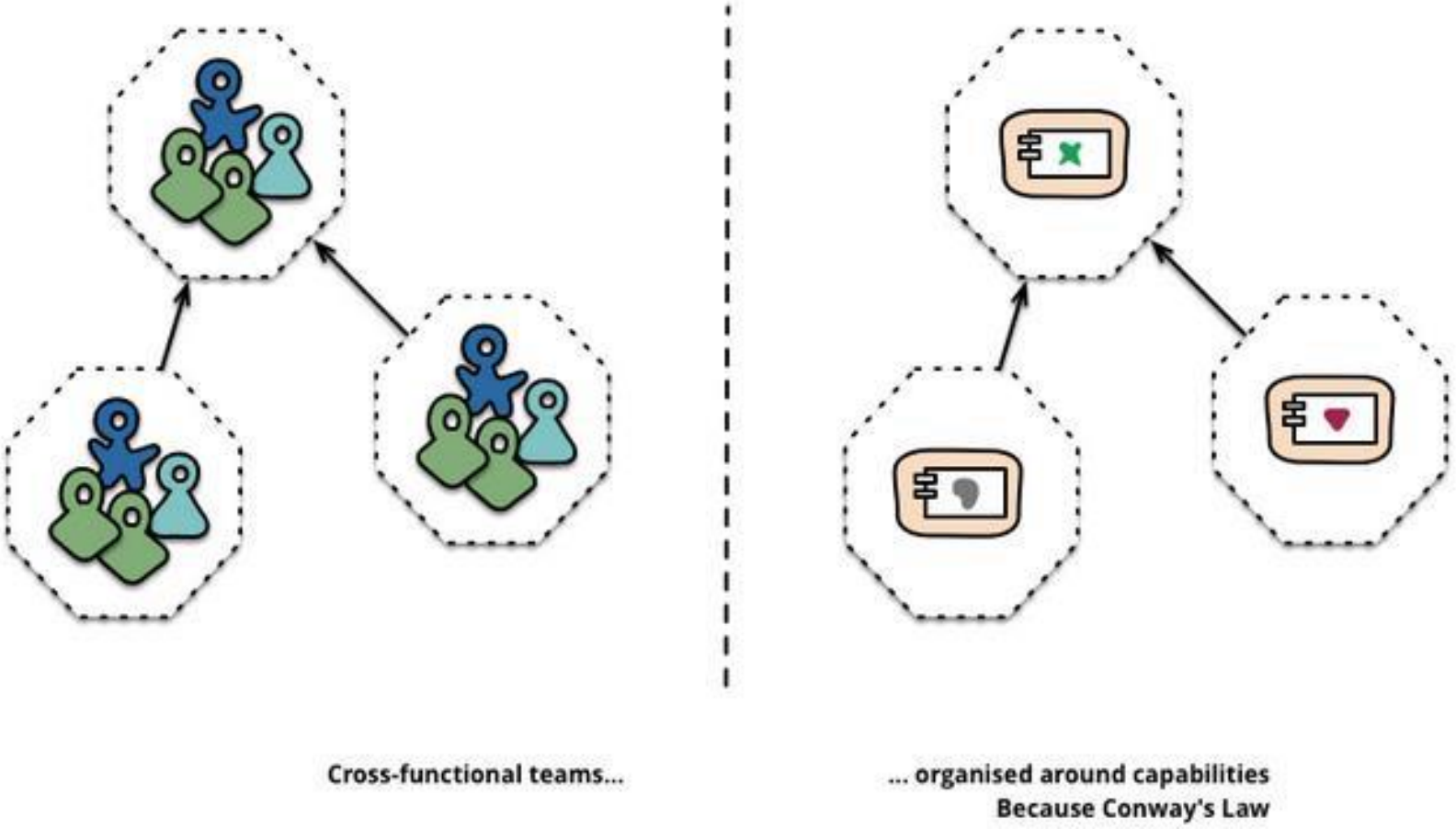
Organized around business capabilities (2)





Will this work?

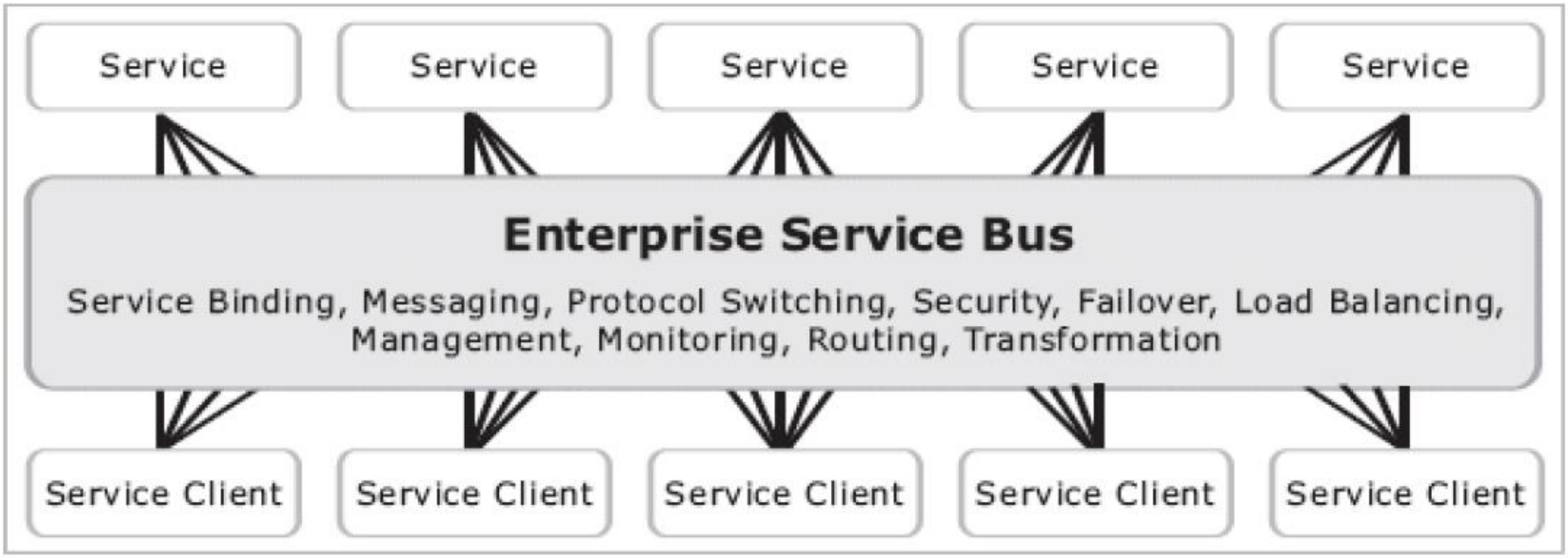
Organized around business capabilities (3)



Products not projects

- Most application development efforts that we see use a project model: where the aim is to deliver some piece of software which is then considered to be completed.
- On completion the software is handed over to a maintenance organization and the project team that built it is disbanded.
- Microservice proponents tend to avoid this model, preferring instead the notion that a team should own a product over its full lifetime.
- A common inspiration for this is Amazon's notion of "you build, you run it" where a development team takes full responsibility for the software in production.
- This brings developers into day-to-day contact with how their software behaves in production and increases contact with their users, as they have to take on at least some of the support burden.

Smart endpoints and dumb pipes (1)



What's the problem here?

Smart endpoints and dumb pipes (2)

- Applications built from microservices aim to be as decoupled and as cohesive as possible - they own their own domain logic and act more as filters in the classical Unix sense - receiving a request, applying logic as appropriate and producing a response.
- These are choreographed using simple RESTish protocols rather than complex protocols such as WS-Choreography or BPEL or orchestration by a central tool.
- Microservice teams use the principles and protocols that the WWW (and to a large extent, Unix) is built on.
- Often used resources can be cached with very little effort on the part of developers or operations folk.
- Uses HTTP request-response with resource API.
- Messaging over a lightweight message bus. The infrastructure chosen is typically dumb (dumb as in acts as a message router only) – simple implementations such as RabbitMQ or ZeroMQ don't do much more than provide a reliable asynchronous fabric - the smarts still live in the end points that are producing and consuming messages; in the services.

Decentralized governance

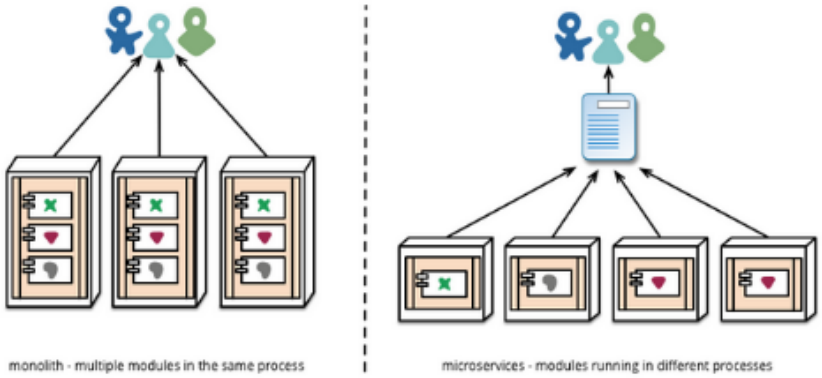
- Principle: focus on standardizing the relevant parts, and leverage battletested standards and infrastructure
- Rather than use a set of defined standards written down somewhere on paper, prefer the idea of producing useful tools that other developers can use to solve similar problems to the ones they are facing.
- These tools are usually harvested from implementations and shared with a wider group, sometimes, but not exclusively using an internal open-source model.
- What needs to be standardized
  - Communication protocol (HTTP)
  - Message format (JSON)
- What should be standardized
  - Communication patterns (REST)
- What doesn't need to be standardized
  - Application technology stack

Decentralized data management

- Each service can choose the persistence solution that fits best its
  - Data access patterns
  - Scaling and data sharding requirements
- Only few services really need enterprise persistence
- Structured programming – shared global state is a bad thing!!!
- 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.

Infrastructure automation

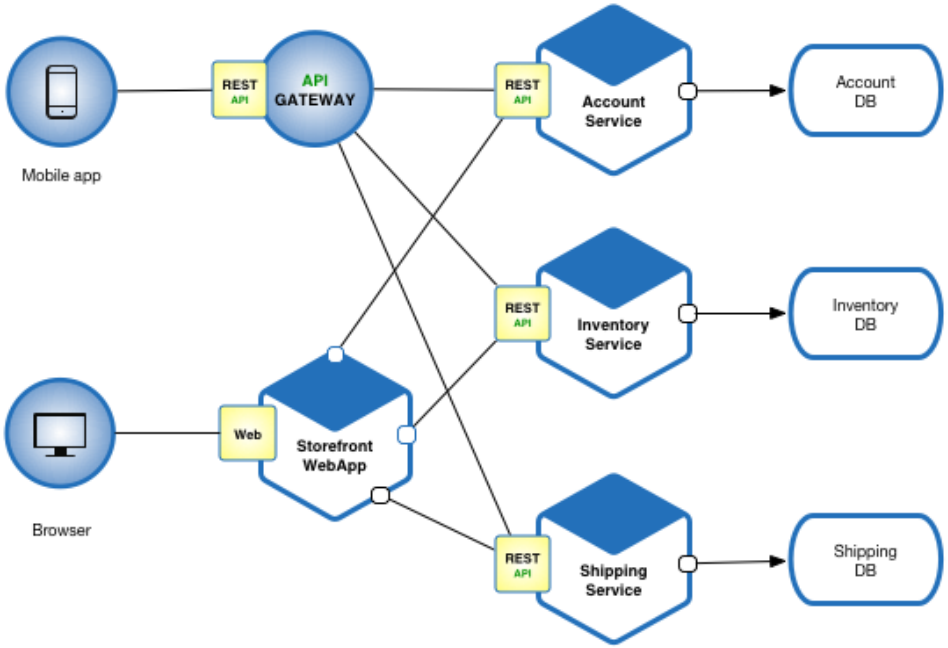
- Having to deploy significant number of services forces operations to automate the infrastructure for
  - Deployment (Continuous Delivery)
  - Monitoring (Automated failure detection)
  - Managing (Automated failure recovery)
- Consider that:
  - Amazon AWS is primarily an internal service
  - Netflix uses Chaos Monkey to further enforce infrastructure resilience



Microservice architecture and patterns

Infrastructure as a code practices

1. Use definition files
2. Self-documented systems and processes
3. Version all things
4. Continuously test systems and processes
5. Small changes rather than batches
6. Keep services available continuously



Design for failure

- A consequence of using services as components, is that applications need to be designed so that they can tolerate the failure of services.
- Since services can fail at any time, it's important to be able to detect the failures quickly and, if possible, automatically restore service.
- Microservice applications put a lot of emphasis on real-time monitoring of the application, checking both architectural elements (how many requests per second is the database getting) and business relevant metrics (such as how many orders per minute are received).
- Semantic monitoring can provide an early warning system of something going wrong that triggers development teams to follow up and investigate.
- Microservice teams would expect to see sophisticated monitoring and logging setups for each individual service such as dashboards showing up/down status and a variety of operational and business relevant metrics

Evolutionary design (1)

- The key property of a component is the notion of independent replacement and upgradeability - which implies we look for points where we can imagine rewriting a component without affecting its collaborators.
- Indeed, many microservice groups take this further by explicitly expecting many services to be scrapped rather than evolved in the longer term.
- The Guardian website is a good example of an application that was designed and built as a monolith but has been evolving in a microservice direction.
- The monolith is still the core of the website, but they prefer to add new features by building microservices that use the monolith's API.

Evolutionary design (2)

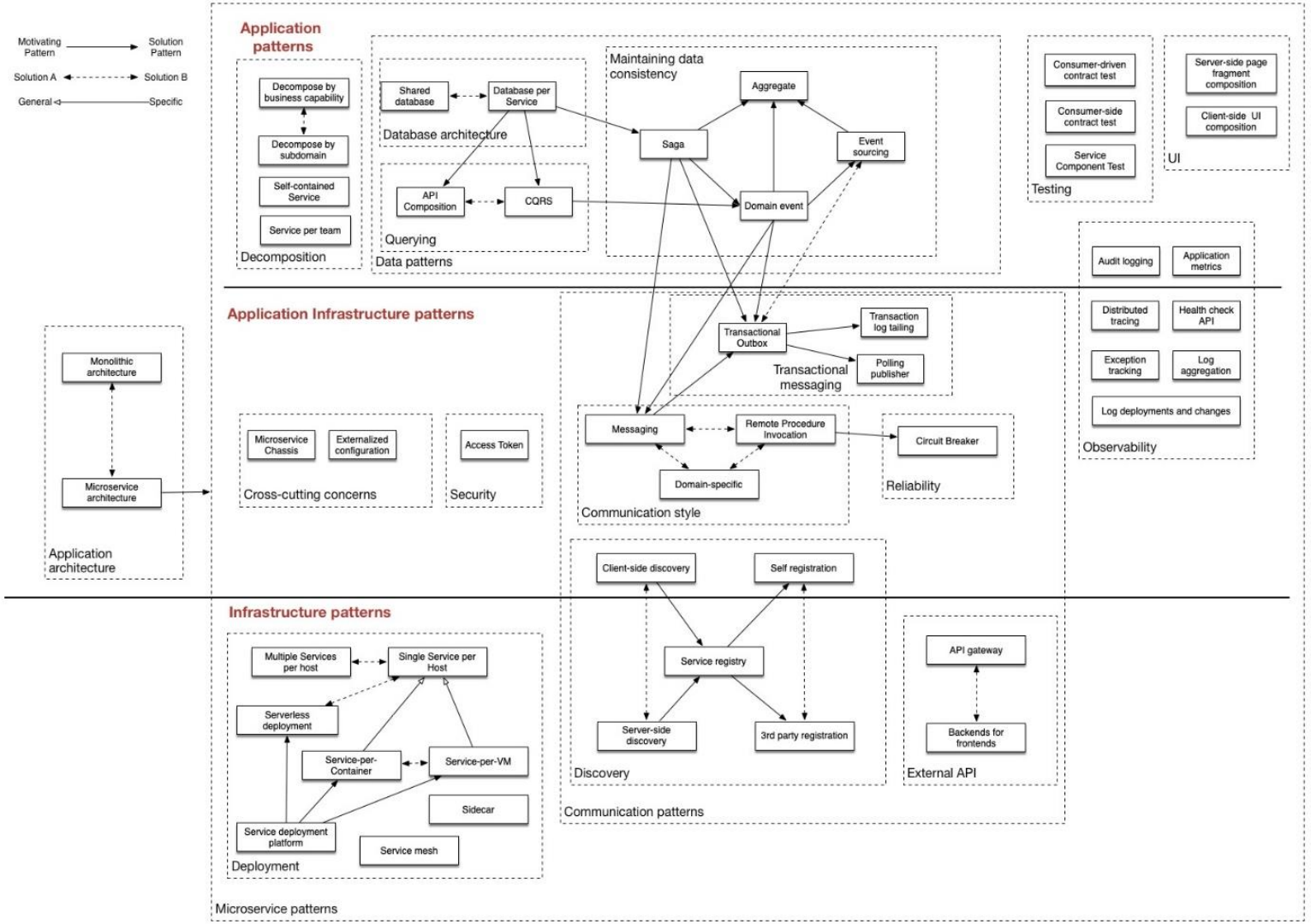
- This approach is particularly handy for features that are inherently temporary, such as specialized pages to handle a sporting event.
- Such a part of the website can quickly be put together using rapid development languages and removed once the event is over.
- Similar approaches at financial institutions where new services are added for a market opportunity and discarded after a few months or even weeks.

Microservice patterns

- Application architecture patterns
- Decomposition
- Refactoring to microservices
- Data management
  - Transactional messaging
- Testing
- Deployment patterns
- Cross cutting concerns
- Communication patterns
  - Style
  - External API
  - Service discovery
- Security
- Observability
- UI patterns

Chris Richardson, *Microservice Patterns*
https://microservices.io/patterns/index.html

Microservice patterns



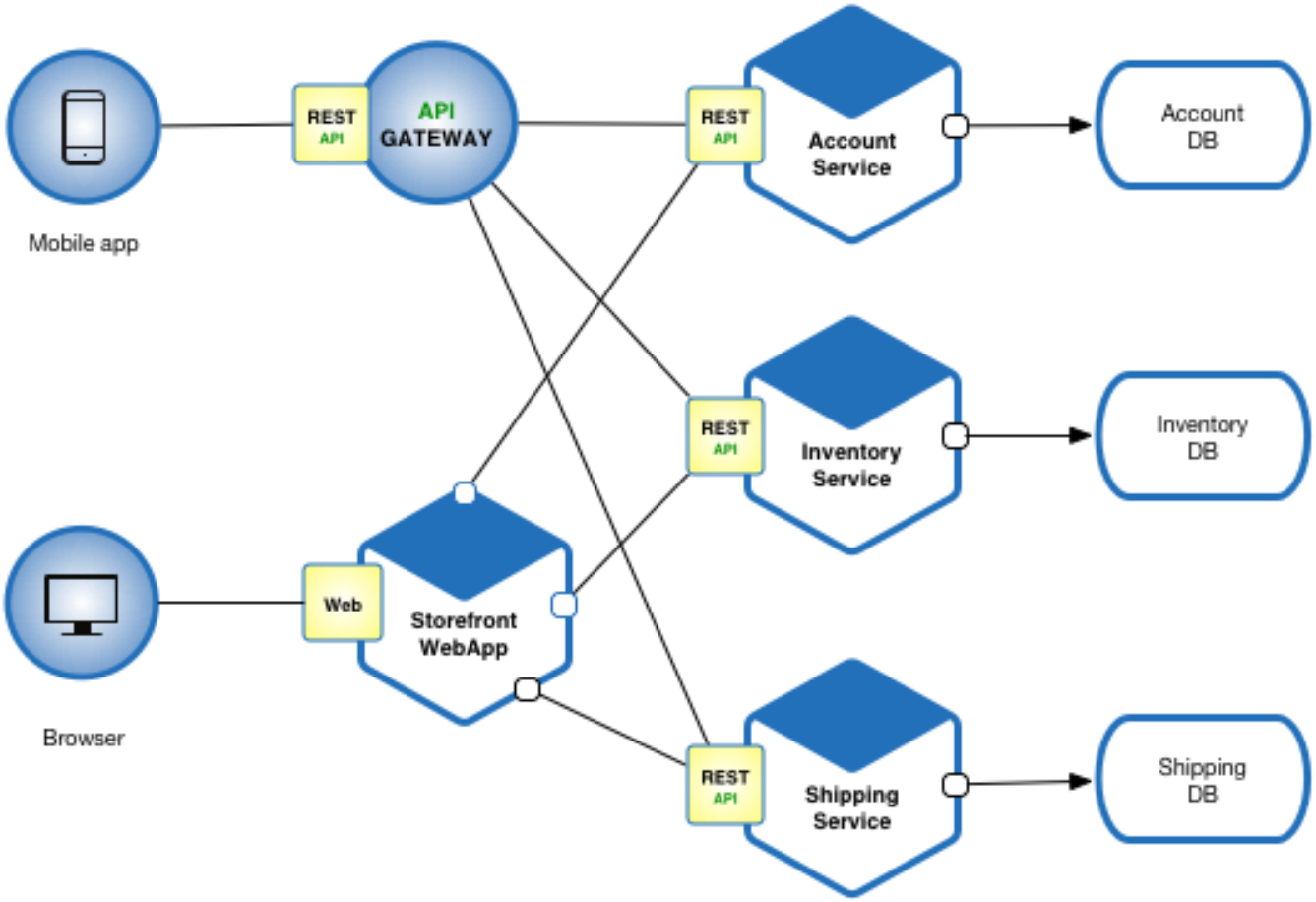
Learn-Build-Assess Microservices http://adopt.microservices.io

Pattern: Microservice architecture

- ***Context**:* You are developing a server-side enterprise application. It must support a variety of different clients including desktop browsers, mobile browsers and native mobile applications. The application might also expose an API for 3rd parties to consume. It might also integrate with other applications via either web services or a message broker.
- ***Problem**:* What's the application's deployment architecture?
- ***Solution**:* Define an architecture that structures the application as a set of loosely coupled, collaborating services. Each service is:
  - Highly maintainable and testable - enables rapid and frequent development and deployment
  - Loosely coupled with other services - enables a team to work independently the majority of time on their service(s) without being impacted by changes to other services and without affecting other services
  - Independently deployable - enables a team to deploy their service without having to coordinate with other teams
  - Capable of being developed by a small team - essential for high productivity by avoiding the high communication head of large teams

Pattern: Microservice architecture

- Microservice e-commerce application

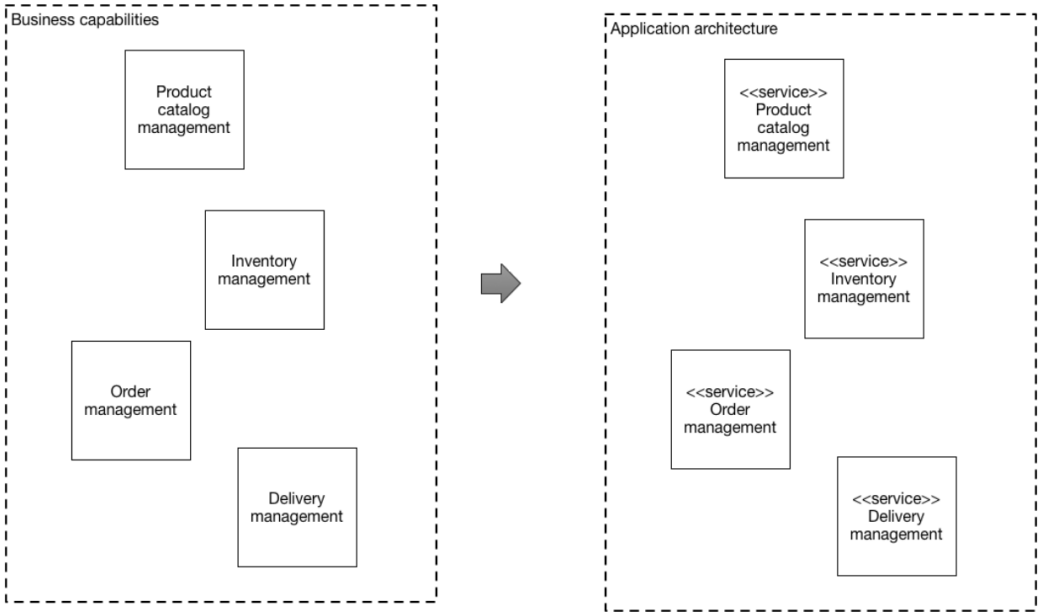


Pattern: Decompose by business capability

- ***Context:*** You are developing a large, complex application and want to use the microservice architecture. The goal of the microservice architecture is to accelerate software development by enabling continuous delivery/deployment. The microservice architecture does this in two ways:
  - Simplifies testing and enables components to deployed independently
  - Structures the engineering organization as a collection of small (6-10 members), autonomous teams, each of which is responsible for one or more services
- ***Problem:*** How to decompose an application into services?
- ***Solution:*** Define services corresponding to business capabilities. A business capability is a concept from business architecture modeling. It is something that a business does in order to generate value.

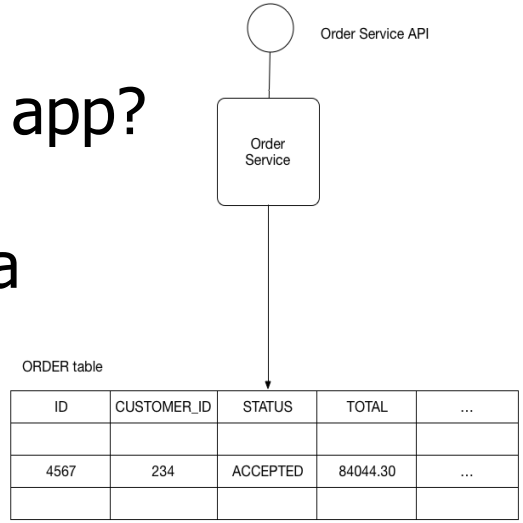
Pattern: Decompose by business capability

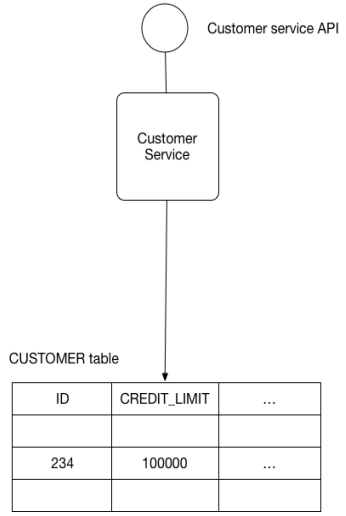
- The business capabilities of an online store include:
  - Product catalog management
  - Inventory management
  - Order management
  - Delivery management
  - ...
- The corresponding microservice architecture would have services corresponding to each of these capabilities.



Pattern: Database per service

- ***Problem:*** which database architecture for microservice app?
- ***Solution:*** keep each microservice's persistent data private to that service and accessible only via its API.





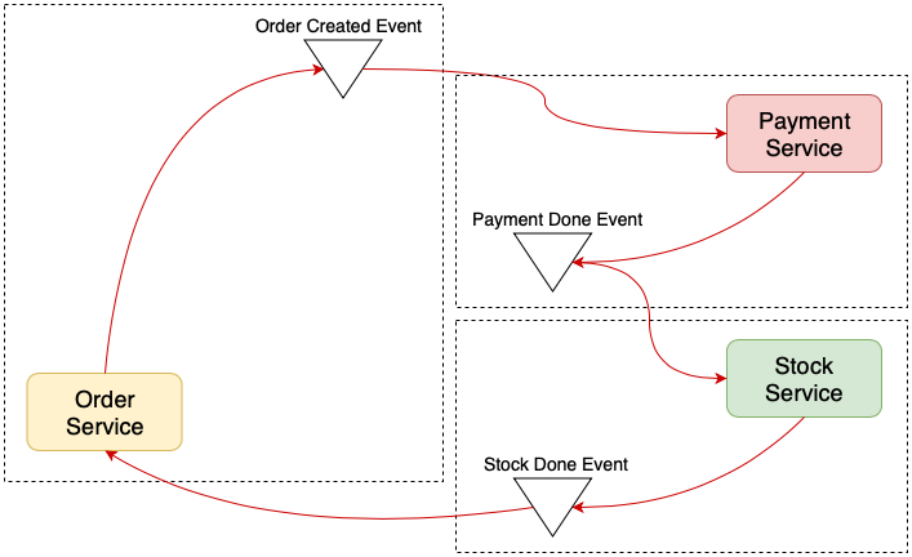
- Service transactions only involve its database
- Pros:
  - Helps ensure that services are loosely coupled
  - Each service can use the type of database that is best suited to its needs (e.g., SQL or NoSQL)
- Cons
  - More complex to implement transactions that span multiple services
  - Complexity of managing multiple databases -> SAGA pattern

Pattern: Saga

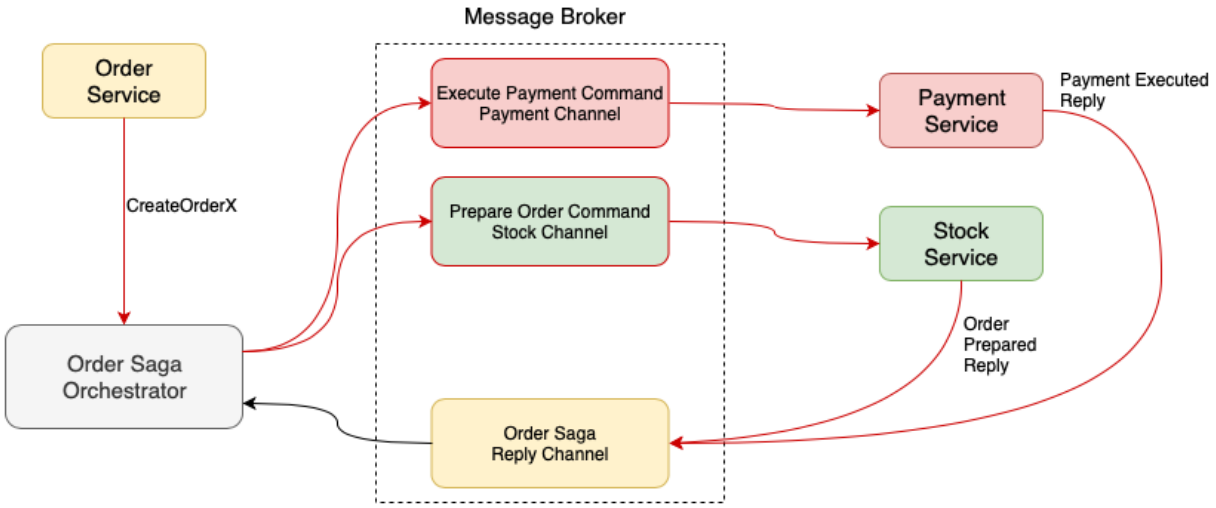
- ***Problem:*** each service has its own database, however some transactions span multiple service: how to ensure data consistency across services?
- ***Solution:*** implement each transaction that spans multiple services as a saga. A Saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction. If a local transaction fails, then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions
- 2 ways of coordination sagas:
  - Choreography: each local transaction publishes events that trigger local transactions in other services
  - Orchestration: an orchestrator tells the participants what local transactions to execute

Pattern: Saga

- Choreography

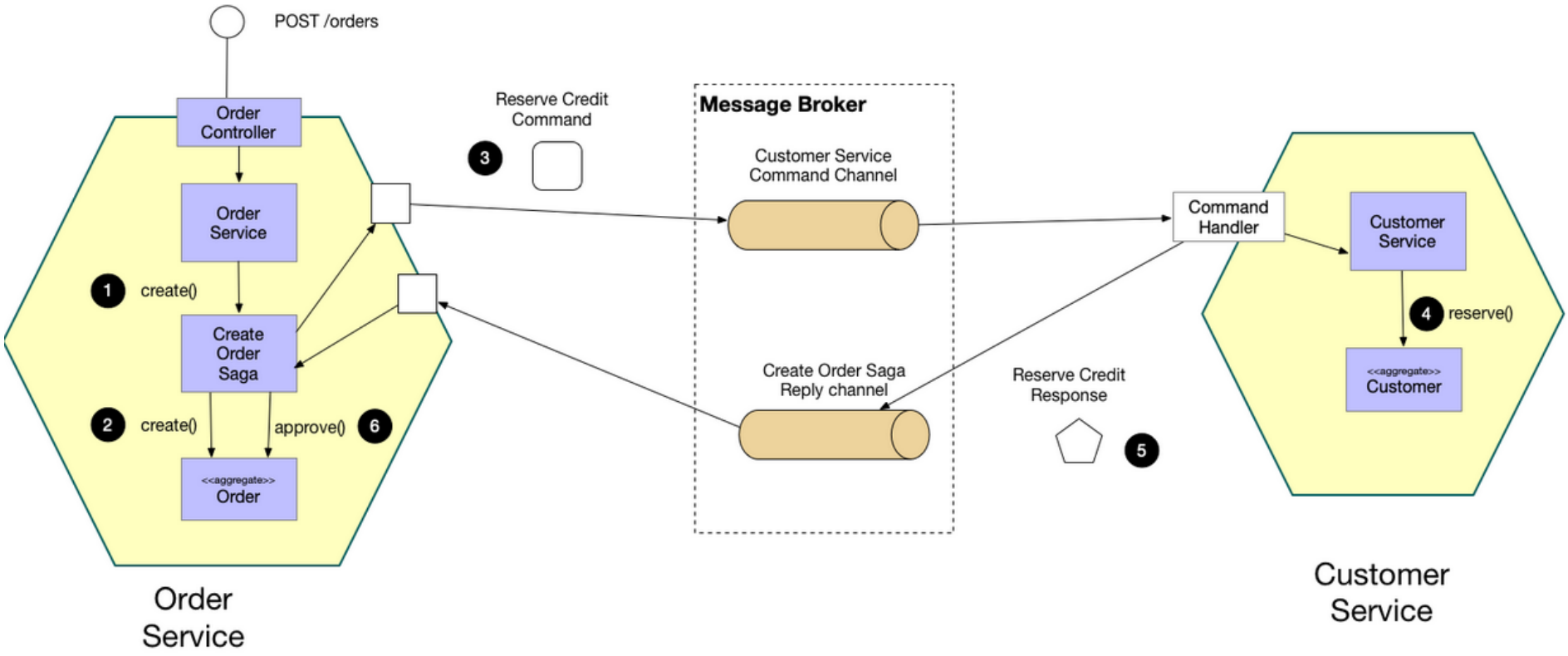


- Orchestration



Pattern: Saga

- Let's consider an example using Saga with orchestration
  - https://microservices.io/patterns/data/saga.html



Pattern: Saga

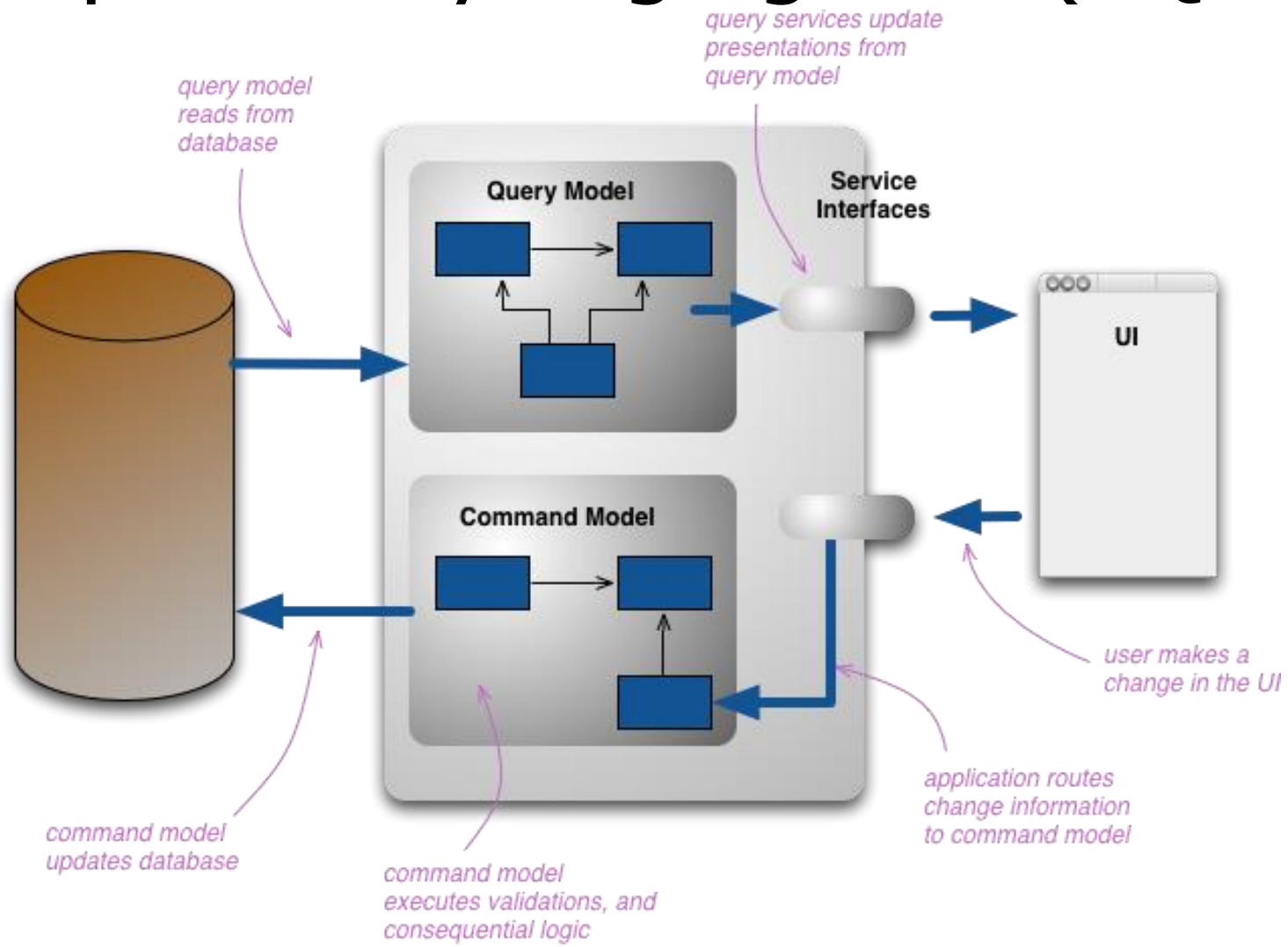
- An e-commerce application that uses this approach would create an order using an orchestration-based saga that consists of the following steps:

1. The Order Service receives the POST /orders request and creates the Create Order saga orchestrator
2. The saga orchestrator creates an Order in the PENDING state
3. It then sends a Reserve Credit command to the Customer Service
4. The Customer Service attempts to reserve credit
5. It then sends back a reply message indicating the outcome
6. The saga orchestrator either approves or rejects the Order

Pattern: Command Query Responsibility Segregation (CQRS)

- ***Context:*** You have applied the Microservices architecture pattern and the Database per service pattern. As a result, it is no longer straightforward to implement queries that join data from multiple services. Also, if you have applied the Event sourcing pattern then the data is no longer easily queried.
- ***Problem:*** How to implement a query that retrieves data from multiple services in a microservice architecture?
- ***Solution:*** Define a view database, which is a read-only replica that is designed to support that query. The application keeps the replica up to data by subscribing to Domain events published by the service that own the data.

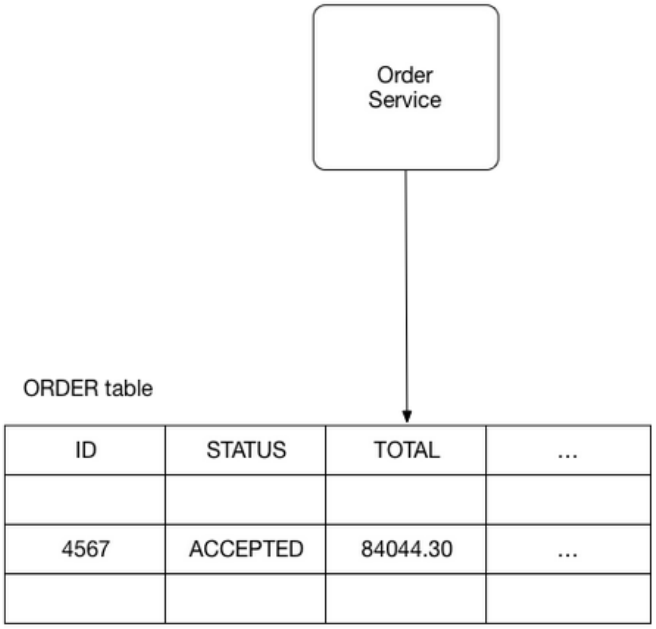
Pattern: Command Query Responsibility Segregation (CQRS)

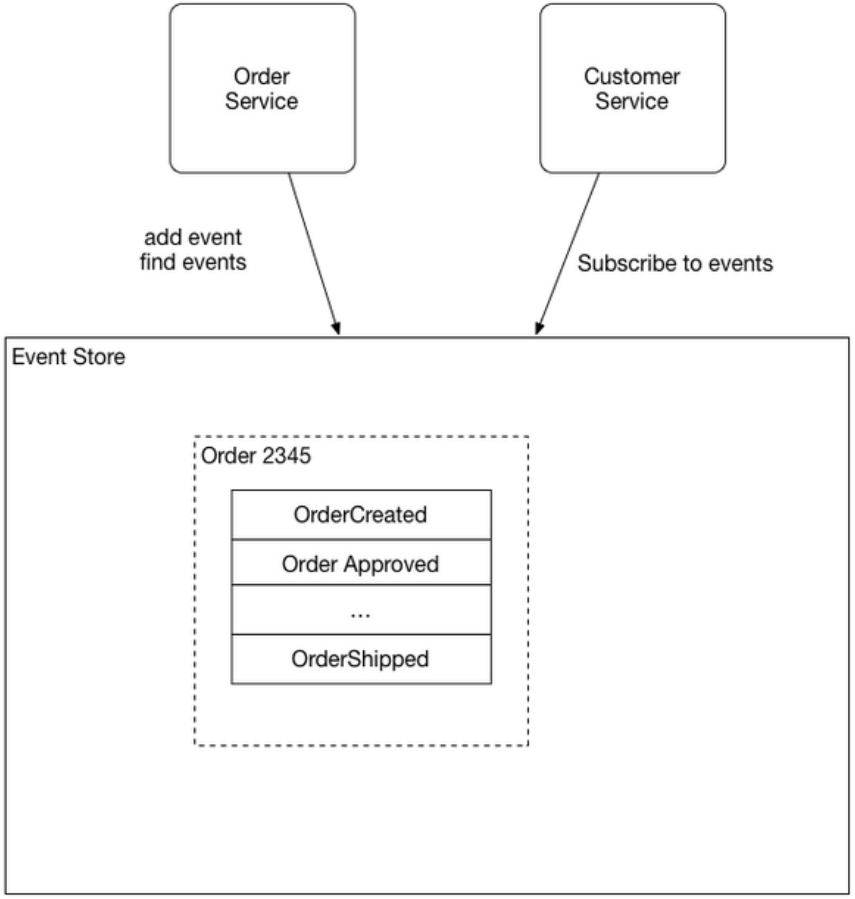


Pattern: Event sourcing

- ***Context**:* A service command typically needs to update the database and send messages/events. For example, a service that participates in a Saga needs to atomically update the database and sends messages/events.
- ***Problem**:* How to reliably/atomically update the database and publish messages/events?
- ***Solution**:* A good solution to this problem is to use event sourcing. Event sourcing persists the state of a business entity such an Order or a Customer as a sequence of state-changing events. Whenever the state of a business entity changes, a new event is appended to the list of events. Since saving an event is a single operation, it is inherently atomic. The application reconstructs an entity's current state by replaying the events.

Pattern: Event sourcing

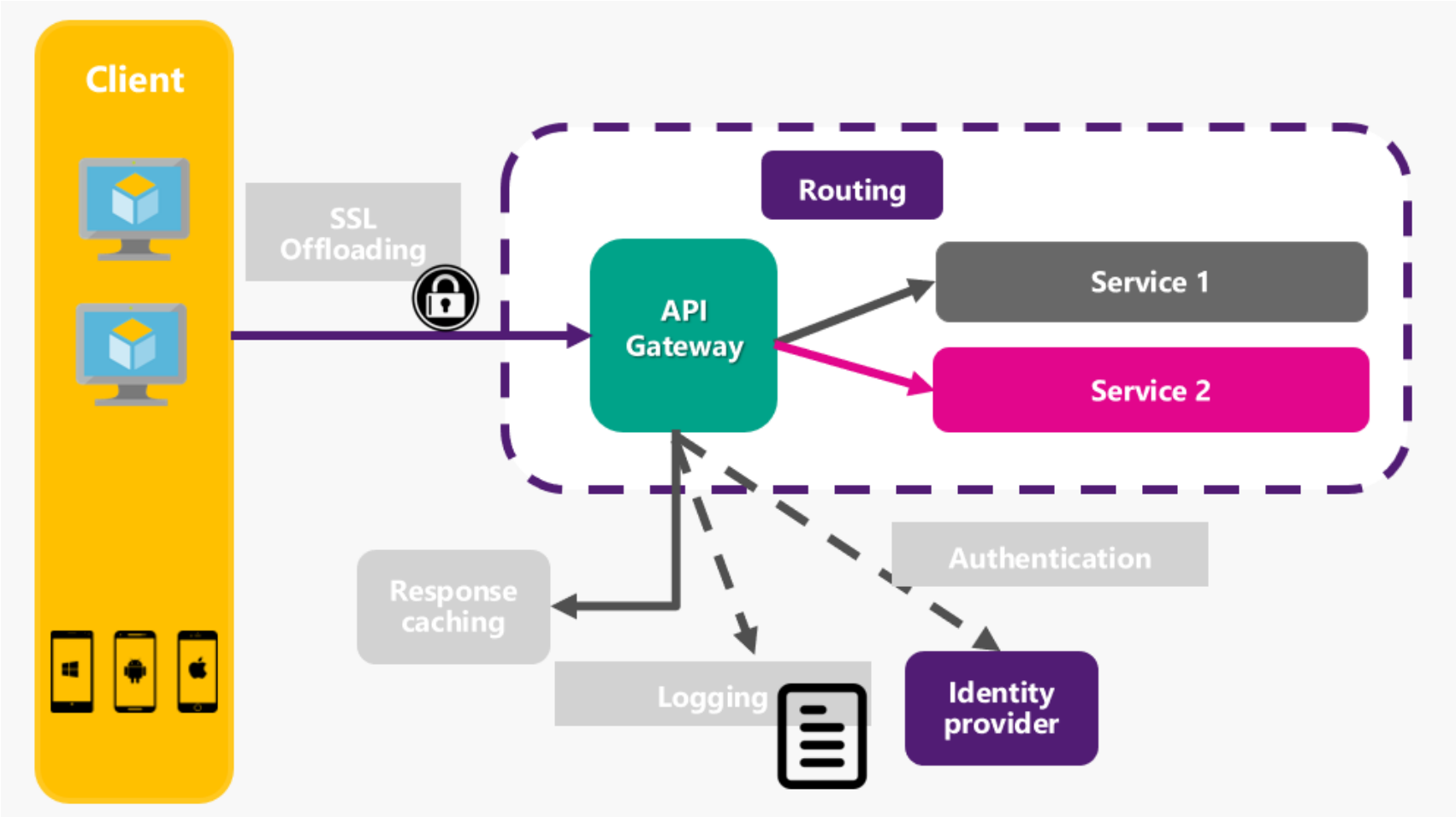




Pattern: API Gateway

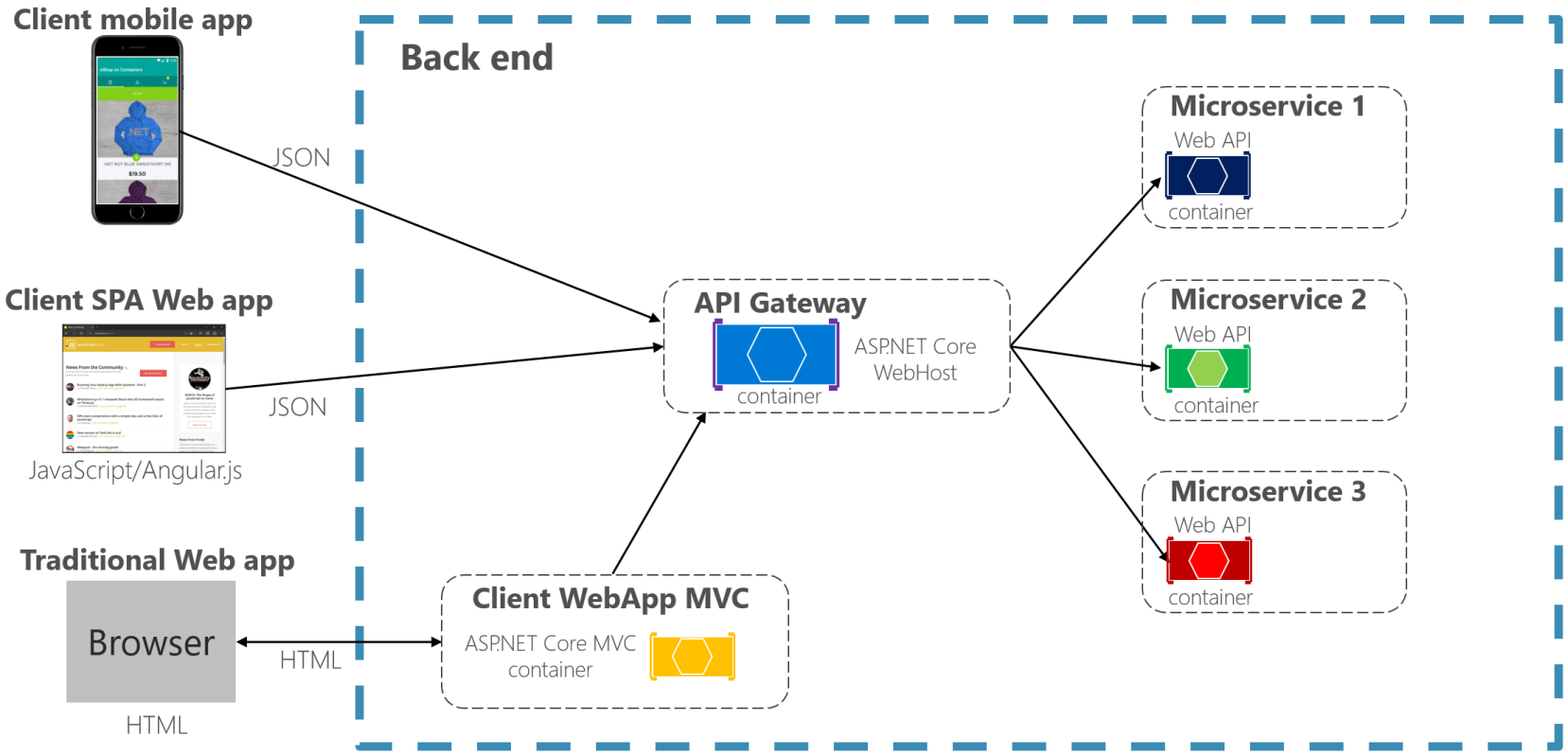
- **Problem:** How do the clients of a Microservices-based application access the individual services?
- **Solution:** Implement an API gateway that is the single entry point for all clients. The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services.
- Rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client.
- The API gateway might also implement security, e.g. verify that the client is authorized to perform the request.
- A variation of this pattern is the Backends for frontends. It defines a separate API gateway for each kind of client.

Pattern: API Gateway



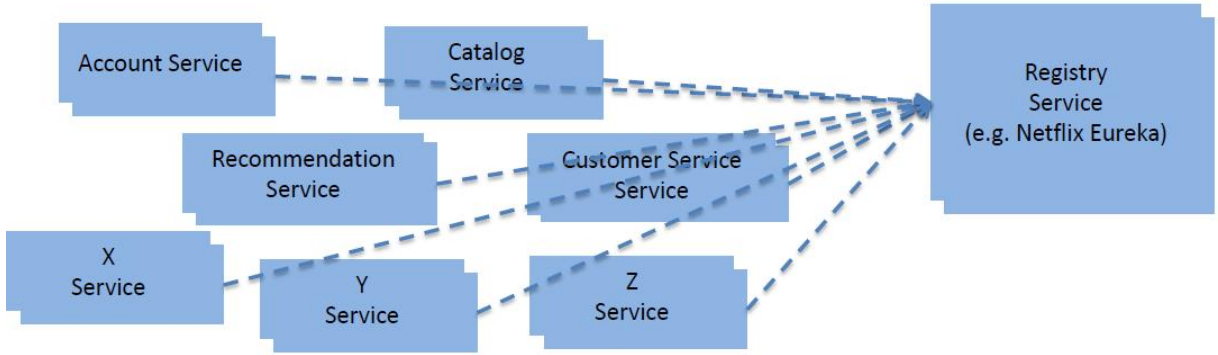
API Gateway in ASP.NET Core

Using a single custom **API Gateway service**



Pattern: Service registry

- ***Problem***: How do clients of a service (in the case of Client-side discovery) and/or routers (in the case of Server-side discovery) know about the available instances of a service?
- ***Solution***: Implement a service registry, which is a database of services, their instances and their locations. Service instances are registered with the service registry on startup and deregistered on shutdown. Client of the service and/or routers query the service registry to find the available instances of a service.



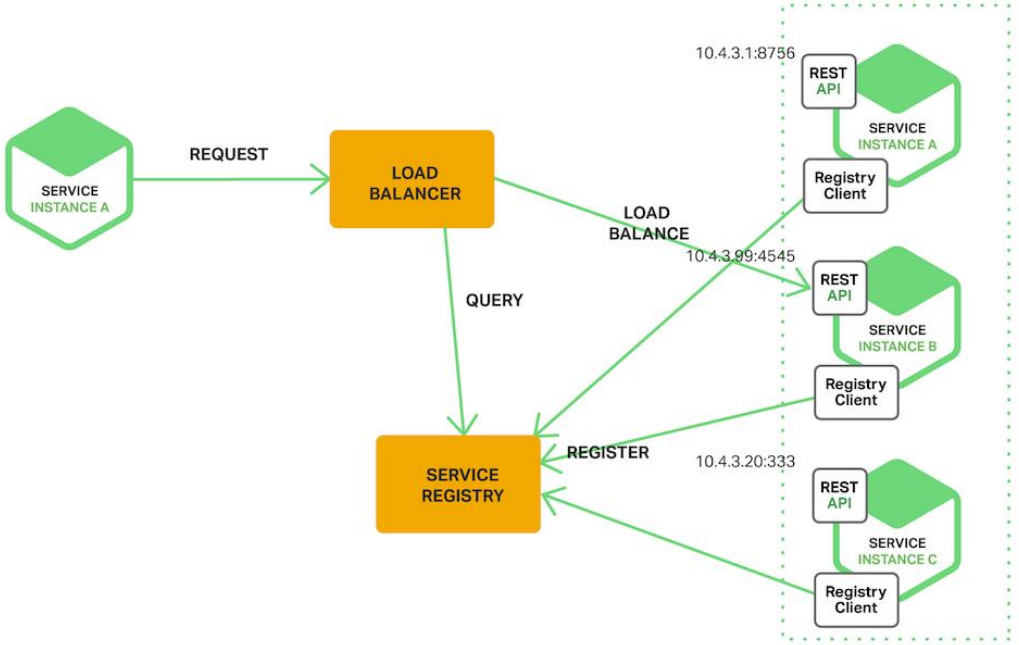
Pattern: Service registry - Tools

- Zookeeper - https://zookeeper.apache.org/
- Netflix Eureka - https://github.com/Netflix/eureka
- Consul - https://www.consul.io/
- etcd – https://coreos.com/etcd
- Synapse - https://github.com/airbnb/synapse

https://consul.io/

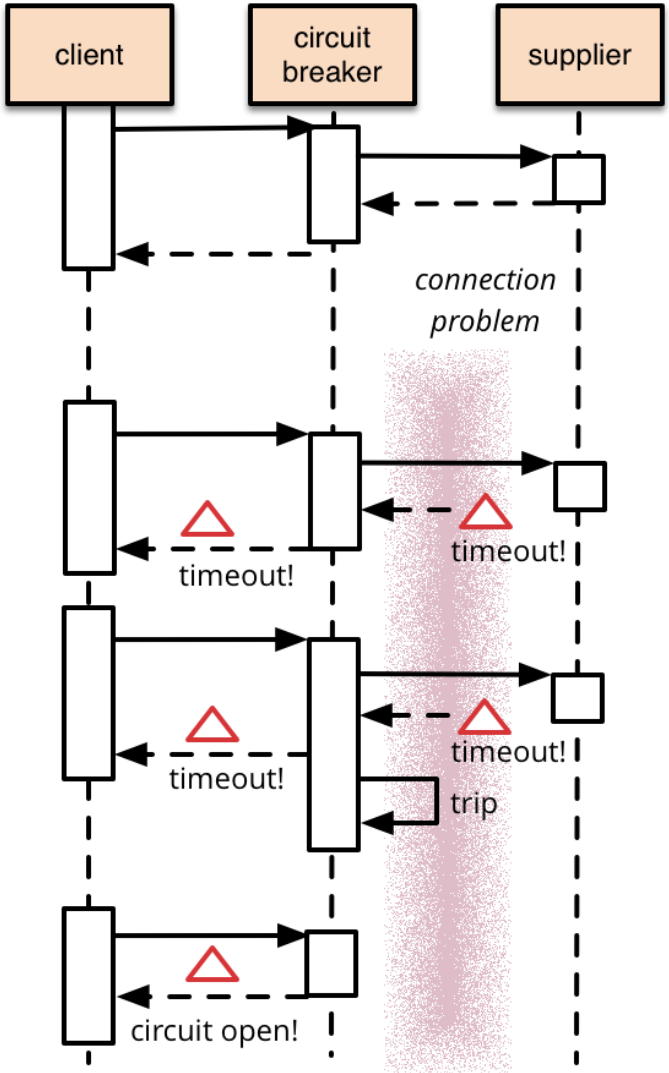
http://zookeeper.apache.org

etcd
https://coreos.com/etcd/



Pattern: Circuit breaker

- ***Problem***: How to prevent a network or service failure from cascading to other services?
- ***Solution***: A service client invokes a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker
- When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately
- After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation.
- Otherwise, if there is a failure the timeout period begins again



Monitoring microservices

- Service distribution (at large scale) → more difficult to monitor microservice app
  - Performance and latency
  - Transaction monitoring
  - Root cause analysis
  - Service dependency analysis
  - Distributed context propagation
- Let's examine two patterns for monitoring microservices (Observability)
  - Log aggregation
  - Distributed request tracing

Pattern: Log aggregation

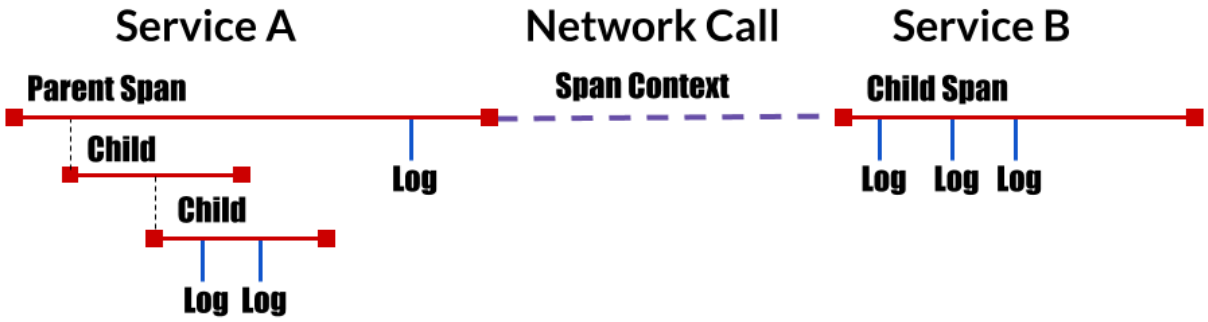
- ***Problem***: How to understand the behavior of an application and troubleshoot problems?
- ***Solution***: Use a centralized logging service that aggregates logs from each service instance. The users can search and analyze the logs. They can configure alerts that are triggered when certain messages appear in the logs
- Cons:
  - Centralized
  - Handling a large volume of logs requires substantial infrastructure
  - https://microservices.io/patterns/observability/application-logging.html

Pattern: Distributed tracing

- ***Problem***: How to understand the behavior of an application and troubleshoot problems?
- ***Solution***: Instrument services with code that
  - Assigns each external request a unique external request id
  - Passes the external request id to all services that are involved in handling the request
  - Includes the external request id in all log messages
  - Records information (e.g. start time, end time) about the requests and operations performed when handling a external request in a centralized service
- Con: aggregating and storing traces can require significant infrastructure
  - https://microservices.io/patterns/observability/distributed-tracing.html

Monitoring microservices - Tools

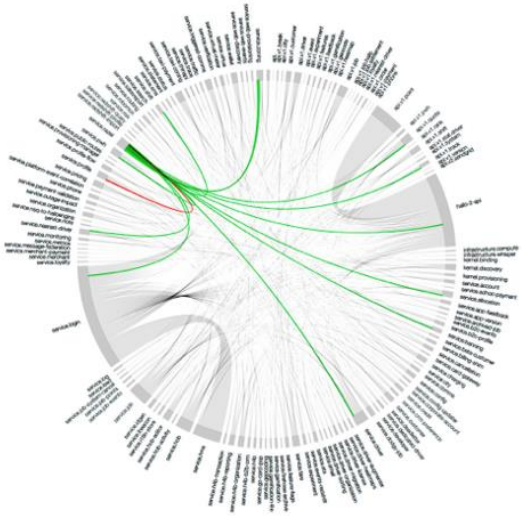
- Tools for microservice monitoring and distributed tracing
  - Prometheus - https://prometheus.io/
  - Sensu - https://sensu.io/
  - OpenTelemetry - https://opentelemetry.io/
  - Zipkin https://zipkin.io/
  - Jaeger https://www.jaegertracing.io
  - cAdvisor – https://github.com/google/cadvisor



Large-scale microservice examples

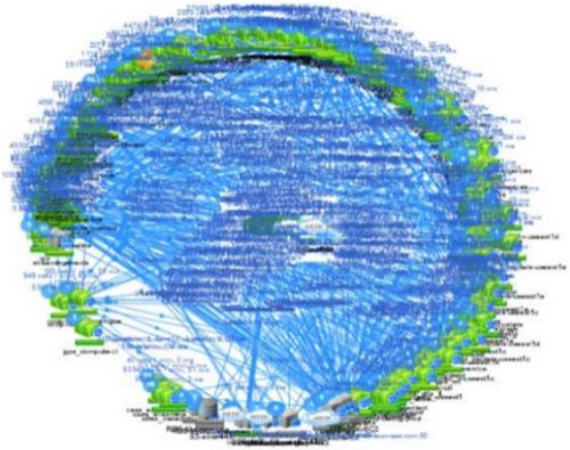
- Netflix and Twitter: 500+ microservices (in 2015)

450+ microservices



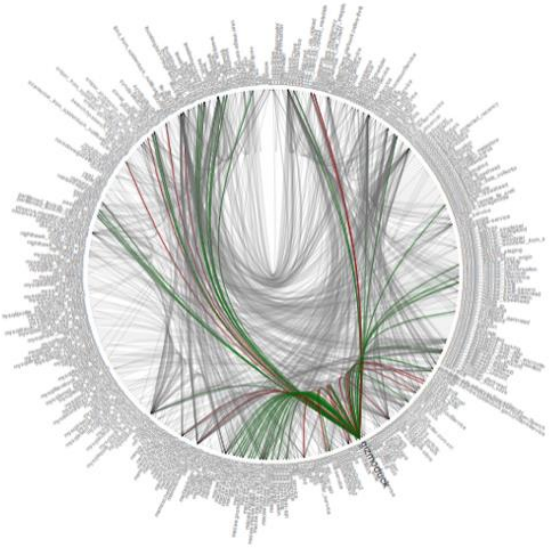
HAIL O

500+ microservices



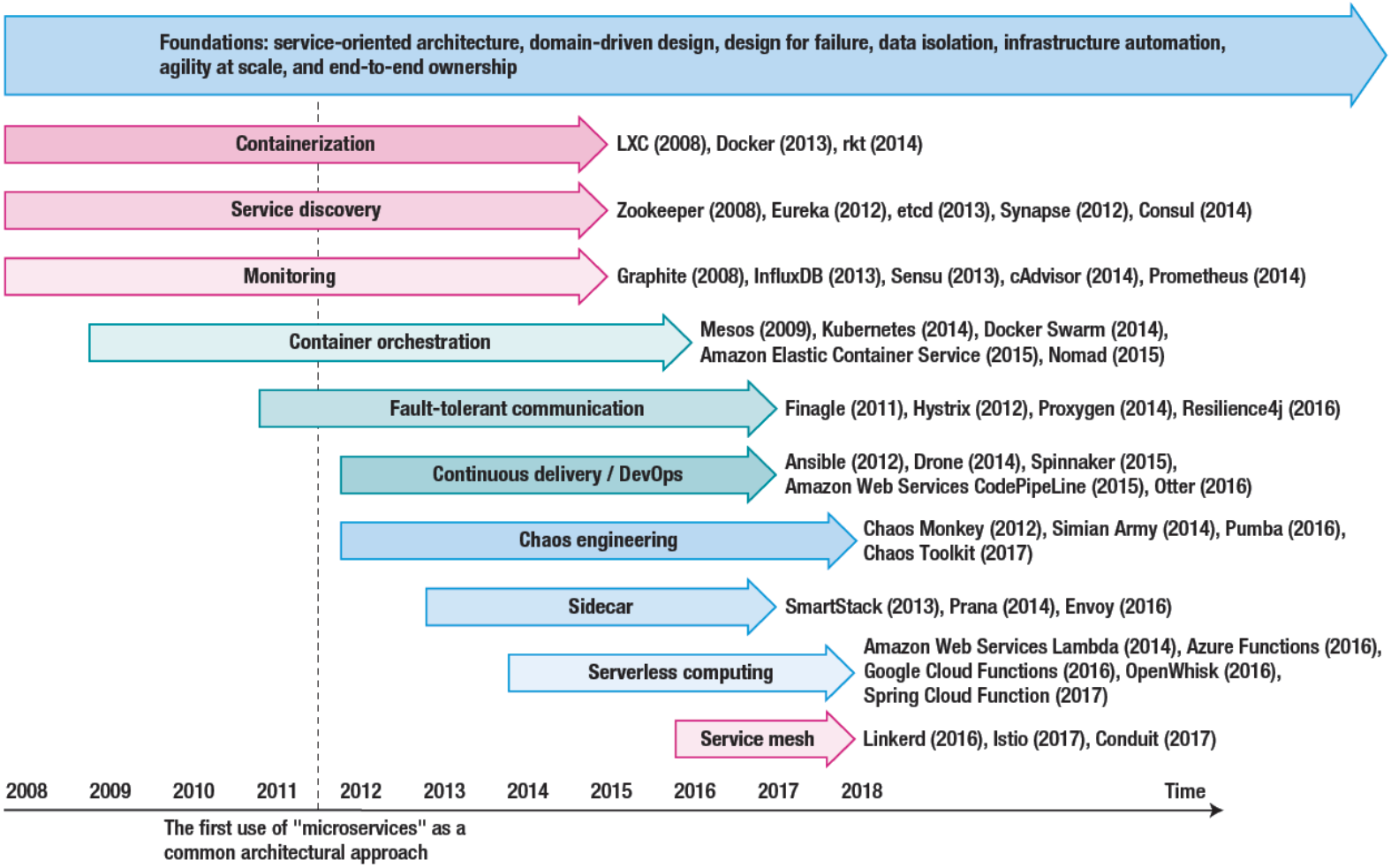
NETFLIX

500+ microservices



Microservice technologies timeline

- Microservices: The Journey So Far and Challenges Ahead

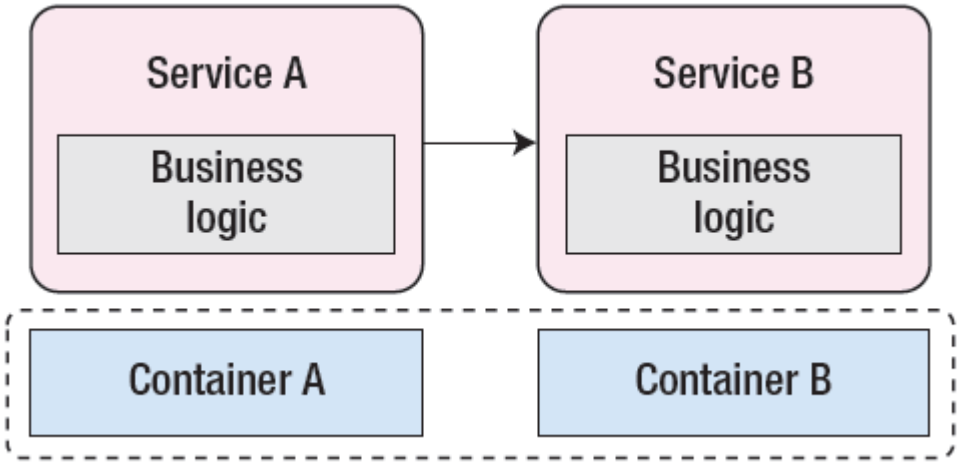


4 generations of microservice architectures - 1st generation

- Container-based virtualization
- Service discovery (e.g., Eureka, ZooKeeper and etcd)
  - Let services communicate with each other without explicitly referring to their network locations
  - Eureka: REST based service developed by Netflix and is primarily used in the AWS cloud for locating services for the purpose of load balancing and failover of middle-tier servers
  - etcd: distributed reliable key-value store for the most critical data of a distributed system (e.g., used by Kubernetes as primary datastore) https://etcd.io
- Monitoring (e.g., Graphite, InfluxDB and Prometheus)
  - Enable runtime monitoring and analysis of the behavior of microservice resources at different levels of detail

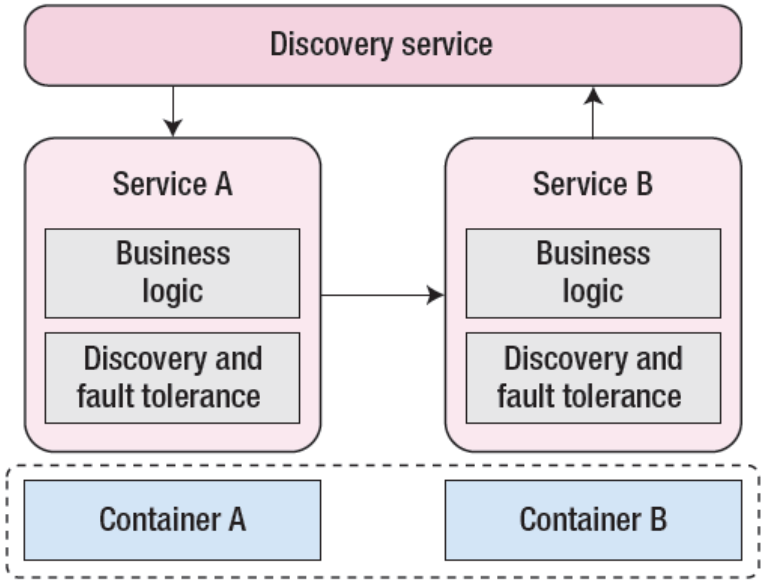
1st gen - Container orchestration

- E.g., Kubernetes, Docker Swarm
- Automate container allocation and management tasks, abstracting away the underlying physical or virtual infrastructure from service developers
- But application-level failure-handling mechanisms still implemented in services source code



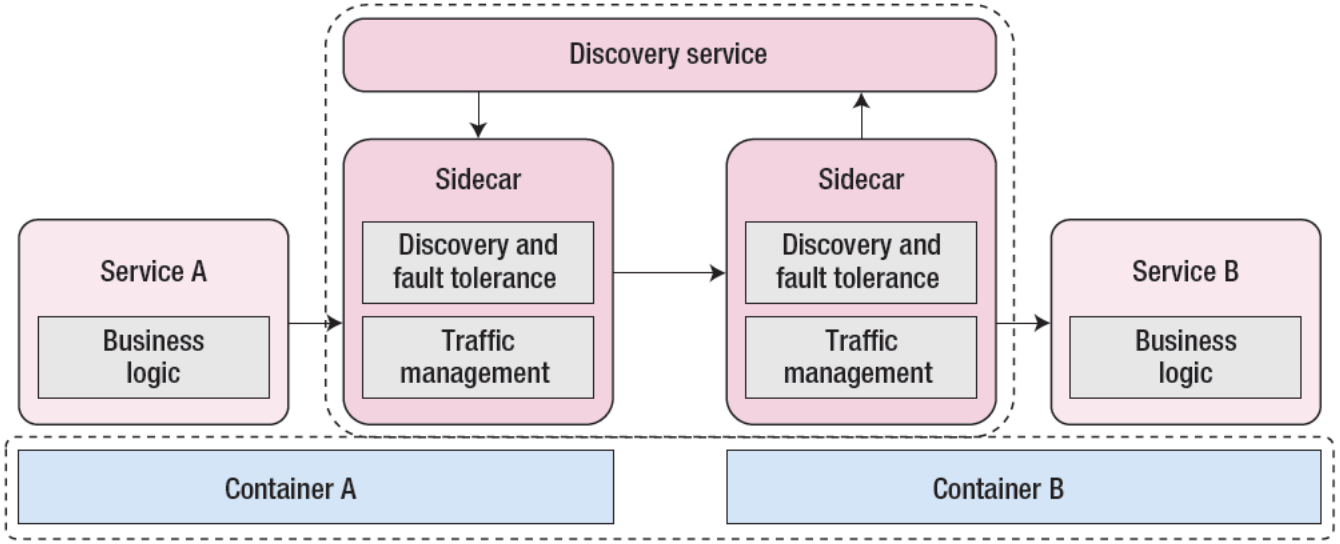
2nd gen - Service discovery and fault tolerance

- Let services communicate more efficiently and reliably based on discovery services and fault-tolerant communication libraries
- **Consul**: service discovery https://www.consul.io
- **Finagle**: fault tolerant, protocol-agnostic RPC system http://twitter.github.io/finagle
- **Hystrix**: latency and fault tolerance library designed by Netflix to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable https://github.com/Netflix/Hystrix



3rd gen – Sidecar proxy

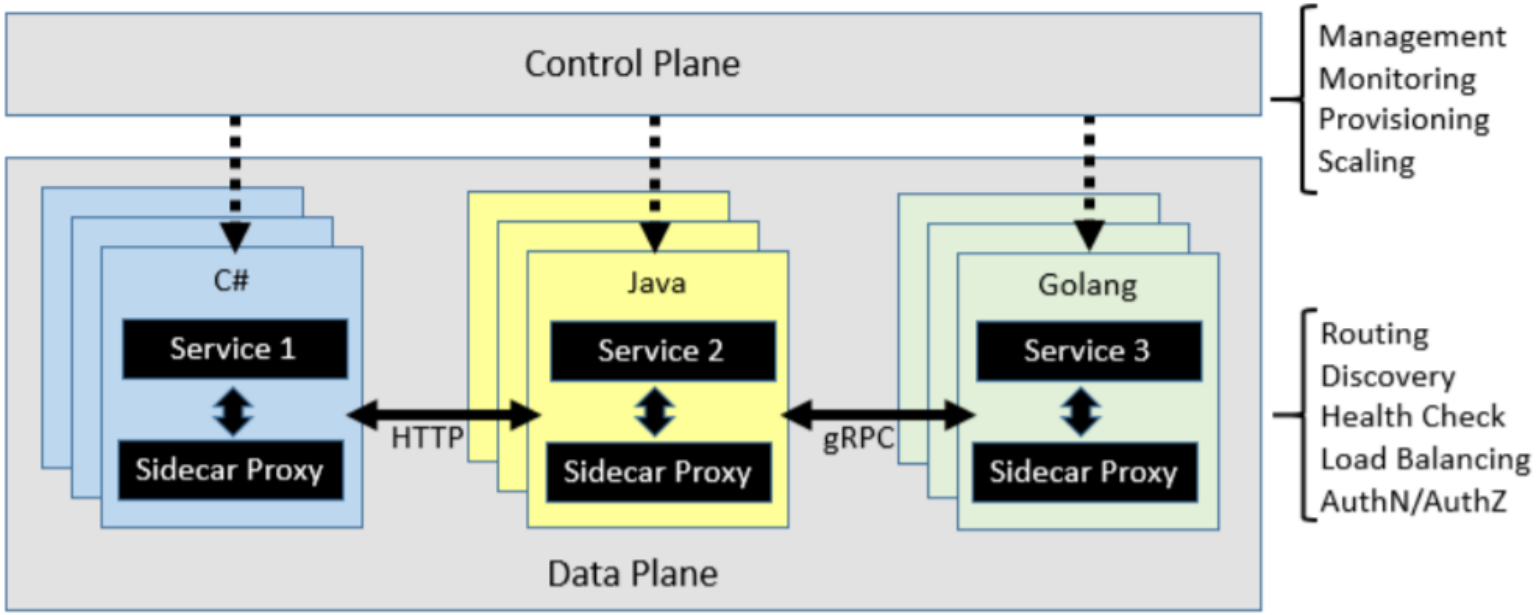
- Based on **sidecar** (or service proxy or ambassador) technologies (e.g., Prana and Envoy)
  - Encapsulate communication-related features such as service discovery and use of protocol-specific and fault-tolerant communication libraries
  - Goal: abstract them from service developers, improve sw reusability and provide homogeneous interface



Microservice architecture and patterns

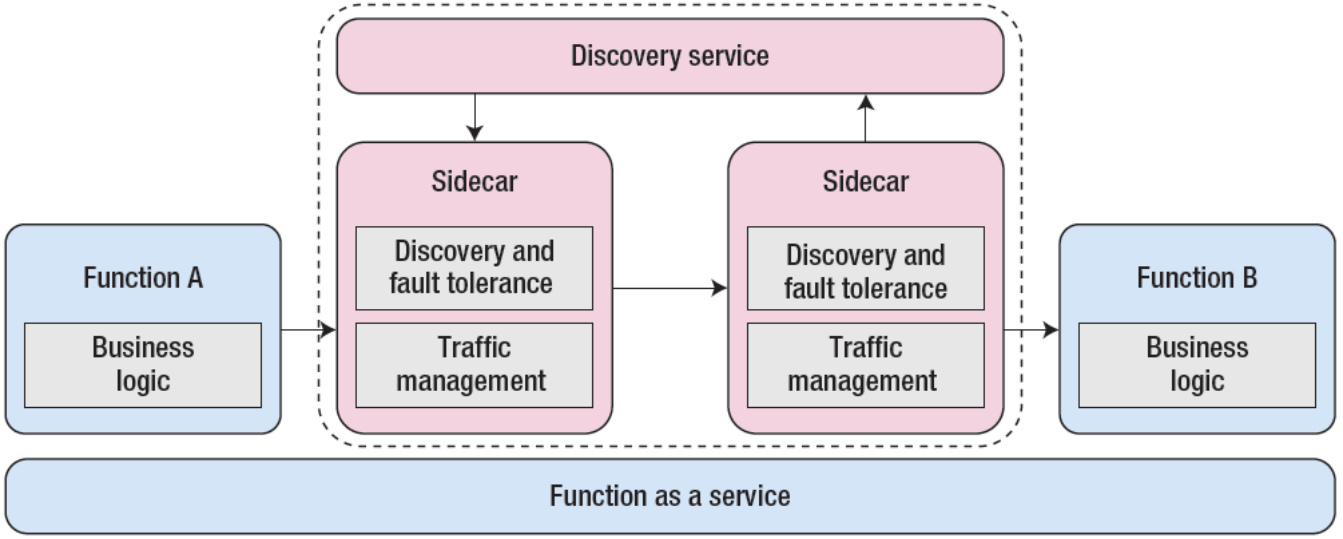
Service mesh

- Service mesh is a configurable infrastructure layer with built-in capabilities to handle service-to-service communication, resiliency, and many cross-cutting concerns.
- It is logically split into two components:
  - Data plane and Control plane



4th gen – Serverless computing

- Based on Function as a Service (FaaS) and serverless computing to further simplify microservice development and delivery
  - AWS Lambda - https://aws.amazon.com/lambda/
  - Azure Functions - https://azure.microsoft.com/services/functions/
  - OpenWhisk - http://openwhisk.apache.org/



Microservice architecture and patterns

References

- Chris Richardson, *Microservice Architecture & Patterns*
  - https://microservices.io
  - https://microservices.io/patterns/index.html
- Chris Richardson, Floyd Smith, *Microservices: From Design to Deployment*, NGINX 2016.
  - https://www.nginx.com/blog/introduction-to-microservices/
- P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis and S. Tilkov, *Microservices: The Journey So Far and Challenges Ahead*, *IEEE Software*, 2018.
  - https://ieeexplore.ieee.org/document/8354433
- *.NET Microservices: Architecture for Containerized .NET Applications*, Microsoft Developer Division, .NET, and Visual Studio product teams 2021,
  - https://docs.microsoft.com/en-us/dotnet/architecture/microservices/