Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Microservices and Serveless Computing

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2022/23

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

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** and **Web services**
  - 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 memory persistence tools (e.g., relational databases and NoSQL data stores)

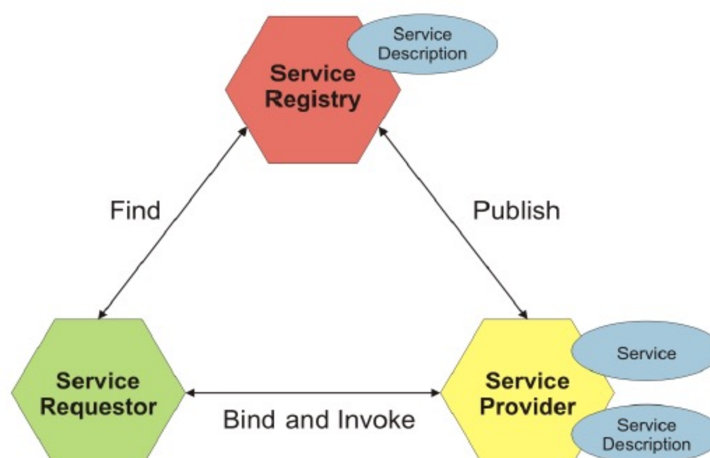
The ancestors: Service Oriented Architecture

- **Service Oriented Architecture** (**SOA**): architectural paradigm for designing loosely coupled distributed sw systems
- Definition (by OASIS docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf)

  SOA is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations
- Properties of SOA (by W3C www.w3.org/TR/ws-arch)
  - Logical view
  - Message orientation and description orientation
  - Service granularity, network orientation
  - Platform neutral

Service Oriented Architecture

- 3 interacting entities
  1. *Service requestor* or *consumer*: requests service execution
  2. *Service provider*: provides service and makes it available
  3. *Service registry*: offers publication and search tools to discover the services offered by providers

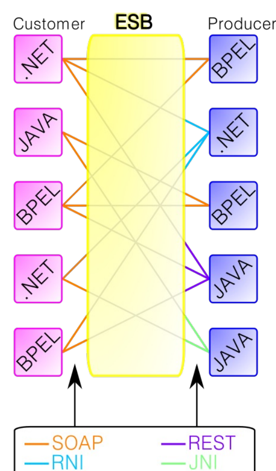


Web services

- Web services: implementation of SOA
- Definition (by W3C www.w3.org/TR/ws-arch)
  - Web service: software system designed to support interoperable machine-to-machine (M2M) interaction over a network
  - Web service interface described in a machine-processable format
  - Other systems interact with web service in a manner prescribed by its description using SOA messages, typically conveyed using HTTP

Web services

- More than 60 standards and specifications, most used:
  - To describe: **WSDL** (Web Service Description Language)
  - To communicate: **SOAP** (Simple Object Access Protocol)
  - To register: **UDDI** (Universal Description, Discovery and Integration)
  - To define business processes: **BPEL** (Business Process Execution Language), **BPMN** (Business Process Model and Notation)
  - To define SLA: **WSLA**
- A variety of technologies
  - Including **ESB** (Enterprise Service Bus), integration platform that provides fundamental interaction and communication services for complex applications



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

Microservices and containers

- Microservices as ideal complementation of container-based virtualization
  - "Microservice instance per container": package each microservice as container image and deploy each microservice instance as container
  - Manage each container at runtime scaling and/or migrating it
- Pros and cons:
  - ✓ Scaling out/in microservice instance by changing number of container replicas
  - ✓ Scaling up/down microservice instance assigning more/less resources
  - ✓ Isolate microservice instance
  - ✓ Apply resource limits on microservice instance
  - ✓ Build and start rapidly
  - ✗ Require container orchestration to manage multi-container app

Microservices: benefits

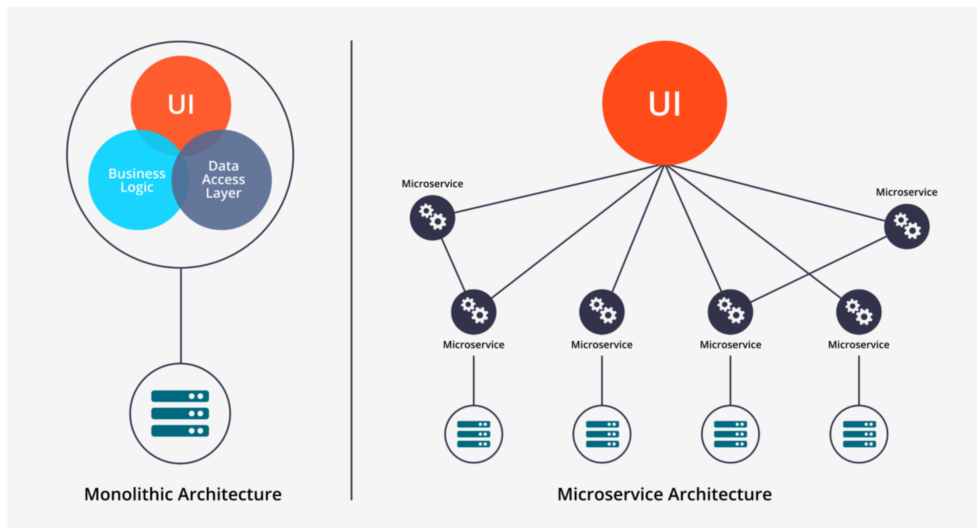
- Increased software agility
  - Microservice: independent unit of development, deployment, operations, versioning, and scaling
  - All interactions with a microservice happen via its API, which encapsulates its implementation details
  - Exploit container-based virtualization
- Improved scalability and fault isolation
- Increased reusability across different areas of business
- Improved data security
- Faster development and delivery
- Greater autonomy
  - Teams can be be more autonomous

Microservices: concerns

- Increased network traffic
  - Inter-service calls over a network cost more in terms of network latency
- Higher complexity
  - Increased operational complexity (e.g., deployment)
  - Global testing and debugging is more complicated

How to decompose the application

- How to decompose a complex app (implemented as *monolithic* application) into microservices?
- Mostly an art, no winner strategy but rather a number of strategies microservices.io/patterns



How to decompose the application

- Main decomposition patterns
  - Let's consider an e-commerce application that takes orders from customers, verifies inventory and available credit, and ships them

1. Decompose by business capability and define services corresponding to business capabilities
   - Business capability: something that a business does in order to generate value
   - E.g., *Order Management* is responsible for orders
2. Decompose by domain-driven design (DDD) subdomain
   - A domain consists of multiple subdomains; each subdomain corresponds to a different part of the business
   - E.g., *Order Management*, *Inventory*, *Product Catalogue*, *Delivery*

How to decompose the application

- Other decomposition patterns
  3. Decompose by use case and define services that are responsible for particular actions
     - E.g., *Shipping Service* is responsible for shipping complete orders
  4. Decompose by nouns or resources and define a service that is responsible for all operations on entities/resources of a given type
     - E.g., *Account Service* is responsible for managing user accounts

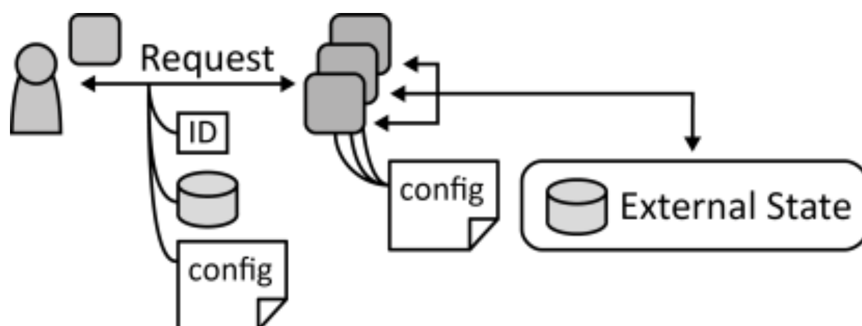
Microservices and scalability

- How to achieve microservice scalability?
  – Use multiple instances of same microservice and load balance requests across multiple instances

- How to improve microservice scalability?
  – State is complex to manage and scale
  – Prefer stateless services: scale better and faster than stateful services

Stateless service

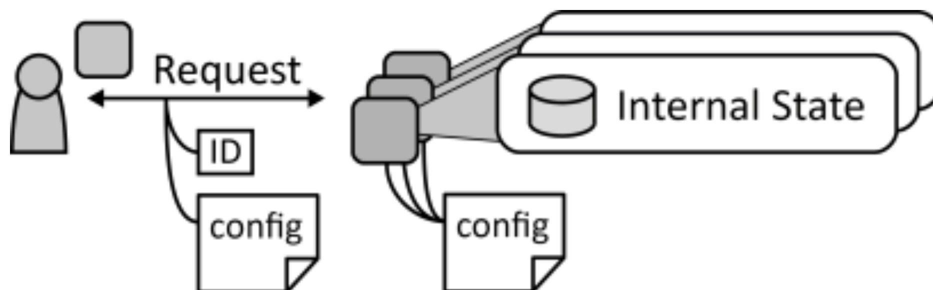
- **Stateless** service: state is handled external of service to ease its scaling out and to make application more tolerant to service failures



Cloud Computing Patterns,
www.cloudcomputingpatterns.org/stateless_component/

Stateful service

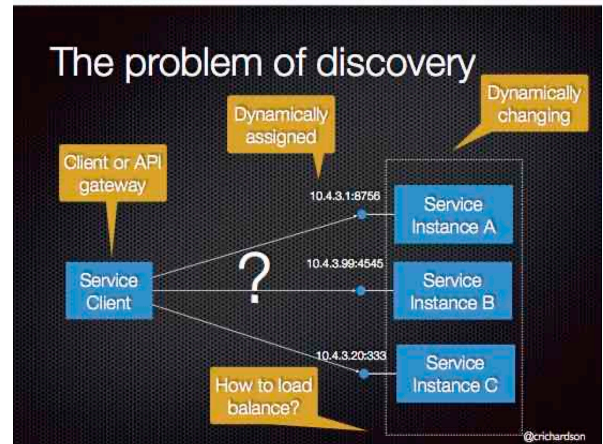
- **Stateful** service: multiple instances of scaled-out service need to synchronize their internal state to provide a unified behavior
- Issue: how can a scaled-out stateful service maintain a synchronized internal state?



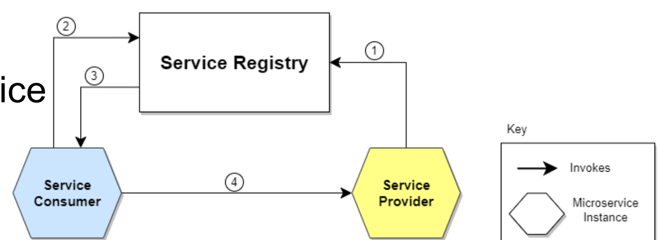
Cloud Computing Patterns,
www.cloudcomputingpatterns.org/stateful_component/

Service discovery

- We also need service discovery
  - Microservice instances have dynamically assigned network locations (IP address and port) and their set changes dynamically because of auto-scaling, failures, and upgrades



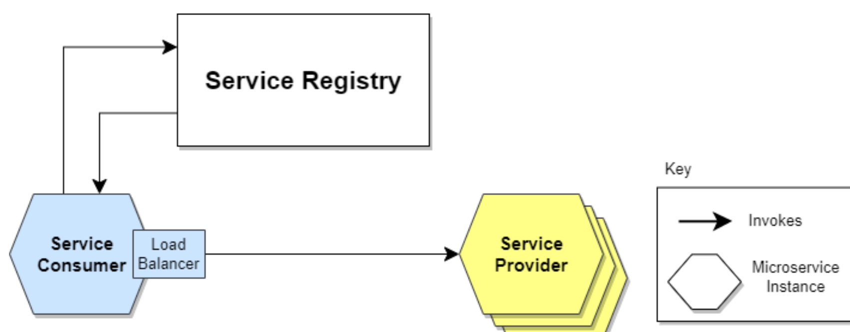
- Service discovery provides
  - a mechanism for a microservice instance to register
  - and a way to find the service once it has registered



Patterns for service discovery

1. Client-side service discovery
   - Client of service is responsible for determining network locations of available service instances and load balancing requests between them
   - Client queries the Service Register, then it uses a load-balancing algorithm to choose one of the available service instances and performs a request

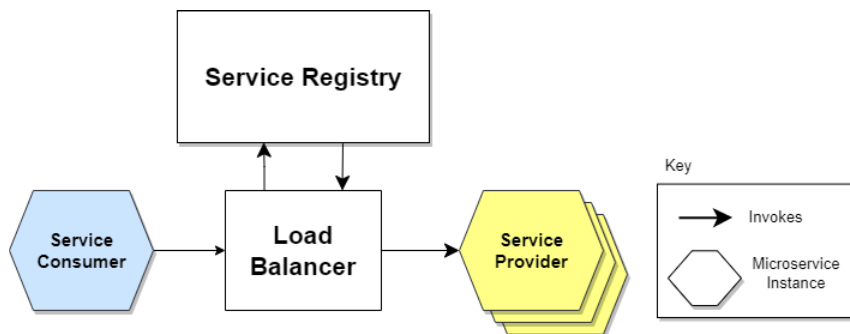
   microservices.io/patterns/server-side-discovery.html



Patterns for service discovery

2. Server-side service discovery
   – Client uses an intermediary that acts as *Load Balancer* and runs at a well known location
   – 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

   microservices.io/patterns/server-side-discovery.html



Integration of microservices

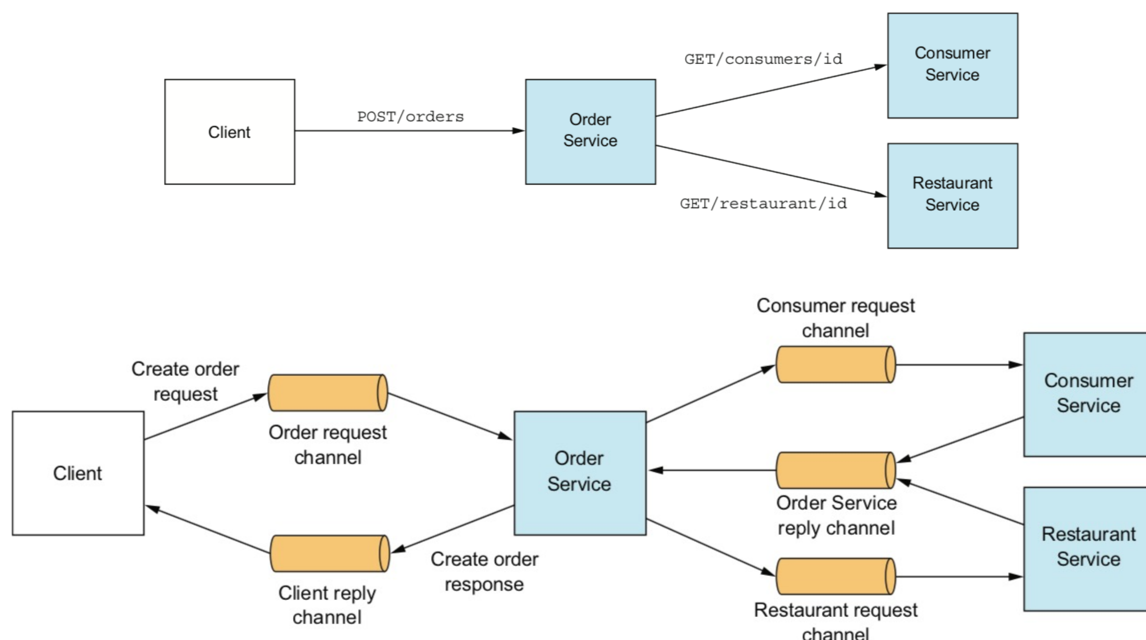
- Let’s consider two issues related to integration of microservices
  – Synchronous vs. asynchronous communication
  – Orchestration vs. choreography

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 communication may reduce availability

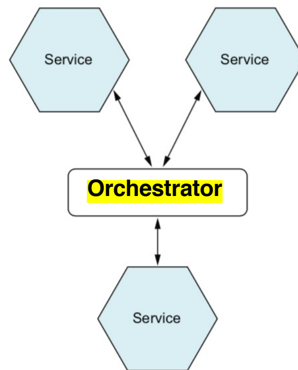
Synchronous vs. asynchronous

- Example of synchronous communication vs. asynchronous communication



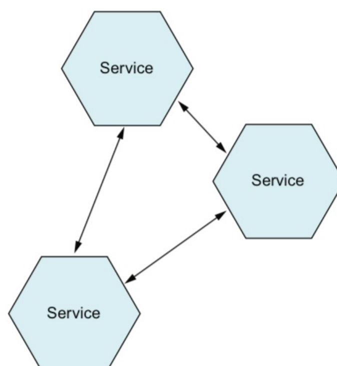
Orchestration and choreography

- Microservices can interact among them following two patterns:
  - Orchestration
  - Choreography
- **Orchestration**: centralized approach
  - A single centralized process (*orchestrator*, *conductor* or *message broker*) coordinates interaction
  - Orchestrator is responsible for invoking and combining services, which can be unaware of composition



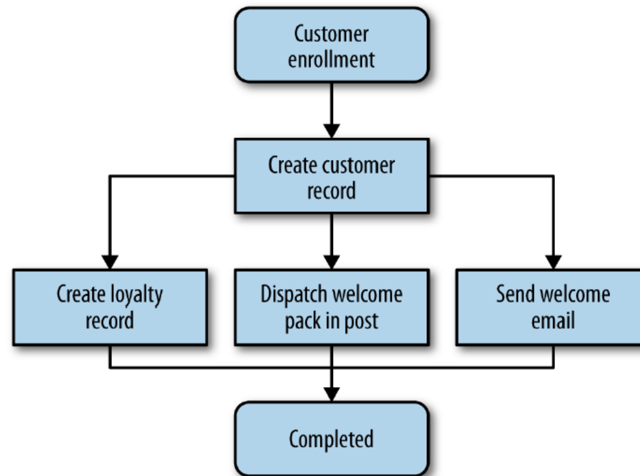
Orchestration and choreography

- **Choreography**: decentralized approach
  - A global description of participating services, which is defined by exchange of messages, rules of interaction and agreements between two or more endpoints
  - Services can exchange messages directly



Example: orchestration and choreography

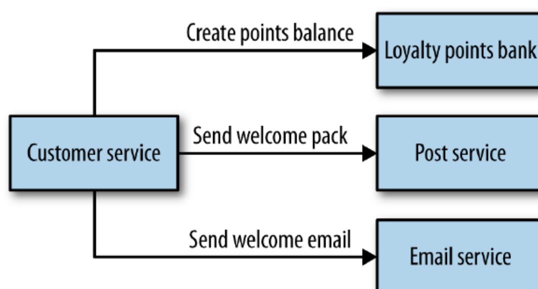
- Example: workflow for customer creation, i.e., process for creating a new customer



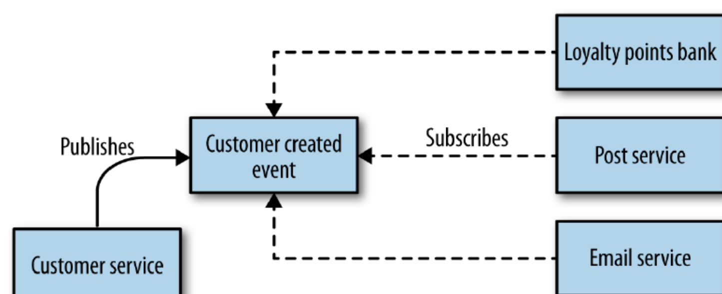
From: S. Newman, “Building Microservices”, O’Really, 2015.

Example: orchestration and choreography

Orchestration



Choreography



Orchestration vs choreography

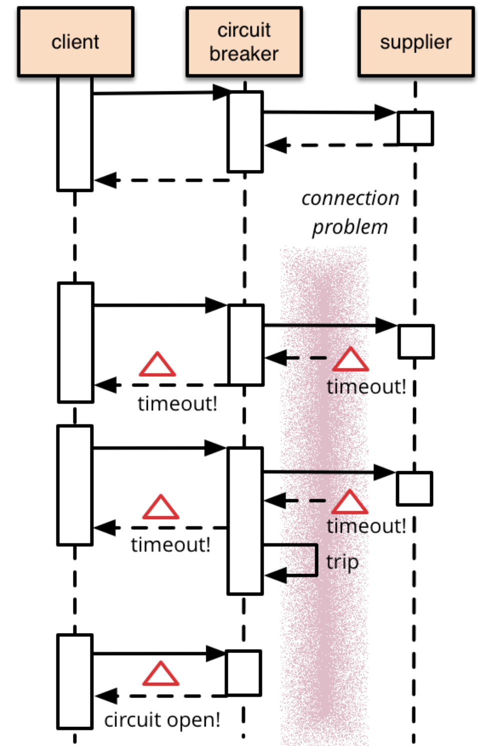
- Orchestration:
  - ✓ Simpler and more popular
  - ✗ SPoF and performance bottleneck
  - ✗ Tight coupling
  - ✗ Higher network traffic and latency
- Choreography
  - ✓ Lower coupling, less operational complexity, and increased flexibility and ease of changing
  - ✗ Services need to know about each other's locations
  - ✗ Extra work to monitor and track services
  - ✗ Implementing mechanisms such as guaranteed delivery is more challenging

Design patterns for microservice-based applications

- Let's examine some design patterns
  1. Circuit breaker
  2. Database per service
  3. Saga (and event sourcing)
  4. CQRS
  5. Log aggregation
  6. Distributed request tracing

Patterns: Circuit breaker

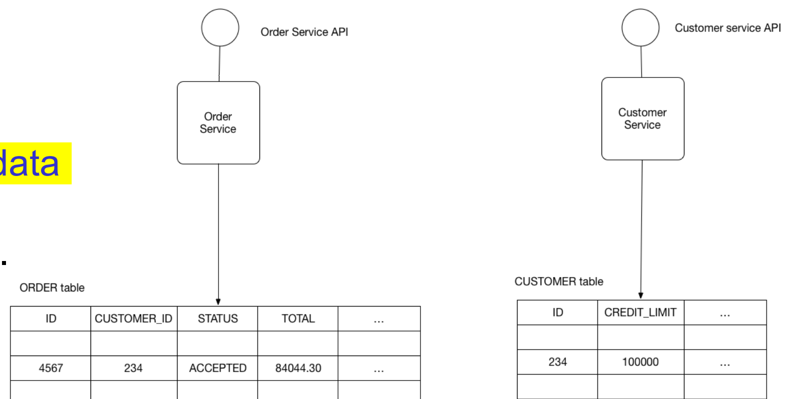
- Problem: How to prevent a network or service failure from cascading to other services?
- Solution: A service client invokes a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker
  - When the *number of consecutive failures* crosses a threshold, the circuit 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



microservices.io/patterns/reliability/circuit-breaker.html

Patterns: Database per service

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

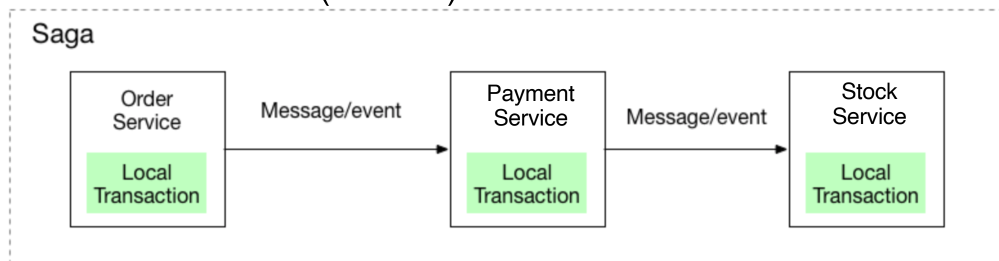


- Pros and cons
  - ✓ Helps ensure that services are loosely coupled
  - ✓ Each service can use the most convenient database type (e.g., NoSQL)
  - ✗ More complex to implement transactions that span multiple services
  - ✗ Complexity of managing multiple databases
    - But do not need to provision a database server for each service
    - Options: private-tables-per-service, schema-per-service, database-server-per-service

microservices.io/patterns/data/database-per-service.html

Patterns: Saga

- Problem: each service has its own database, however some transactions span multiple services: how to maintain data consistency across services without using distributed transactions (2PC protocol)?
- Solution: implement each transaction that spans multiple services as a saga
- **Saga:** sequence of local transactions
  - Each local transaction updates its database and publishes a message or event to trigger the next local transaction in the saga
  - If local transaction fails, then saga executes a series of compensating transactions that undo changes made by preceding local transactions (rollback)

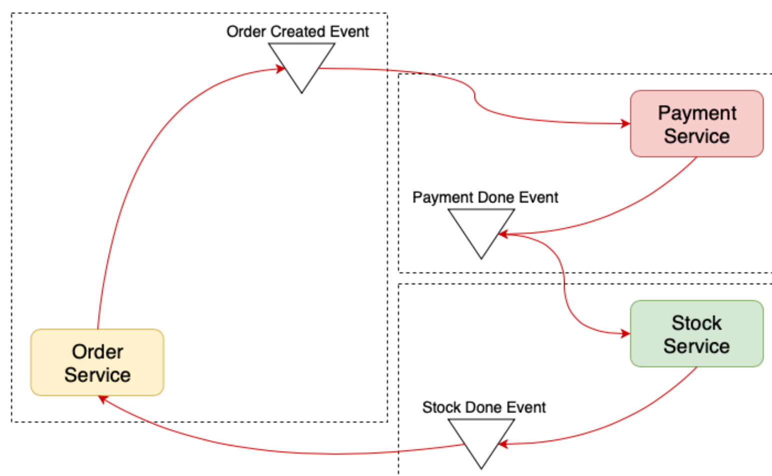


microservices.io/patterns/data/saga.html

Patterns: Saga

- 2 ways to coordinate saga:
  - Choreography: each local transaction publishes events that trigger local transactions in other services
  - Orchestration: an orchestrator tells participants what local transactions to execute

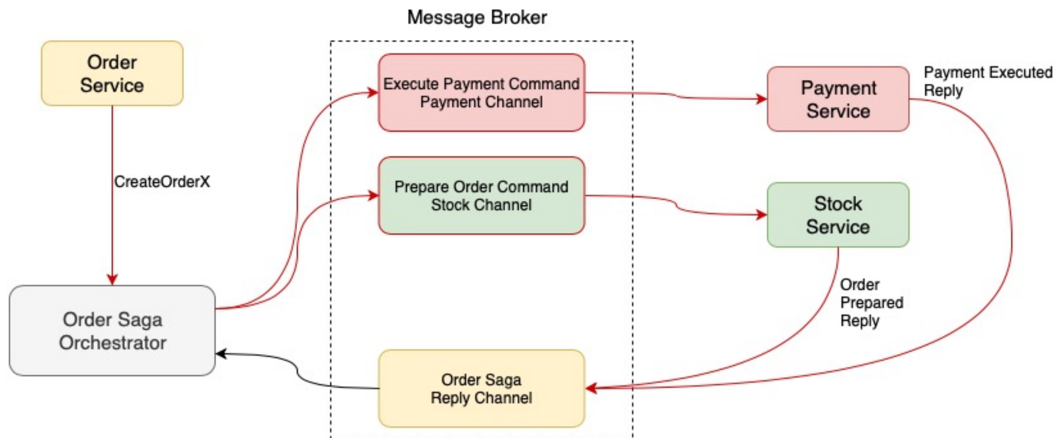
Choreography



Patterns: Saga

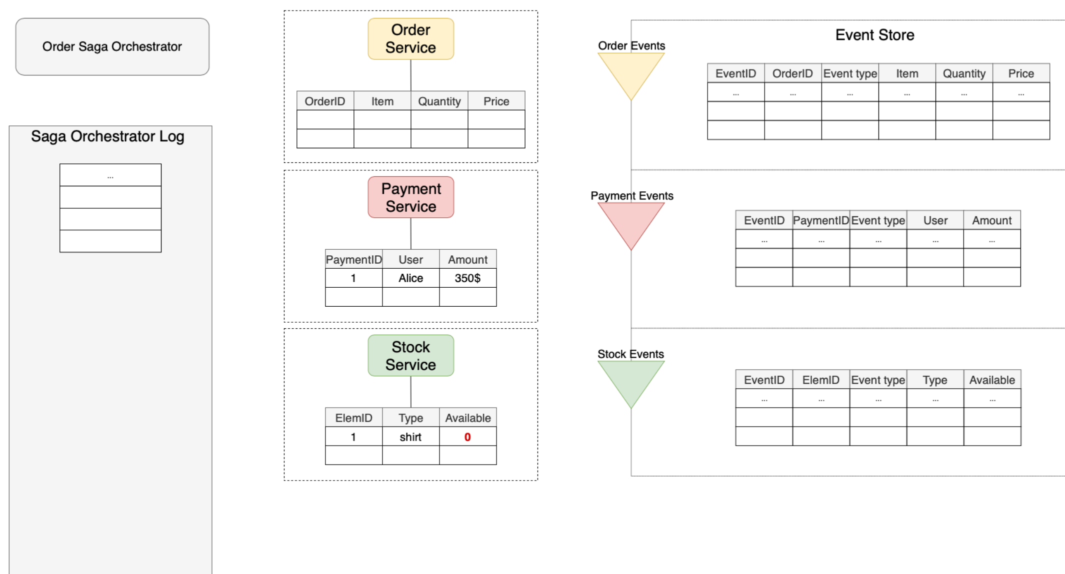
- 2 ways to coordinate saga:
  - Choreography: each local transaction publishes events that trigger local transactions in other services
  - Orchestration: orchestrator tells participants what local transactions to execute

Orchestration



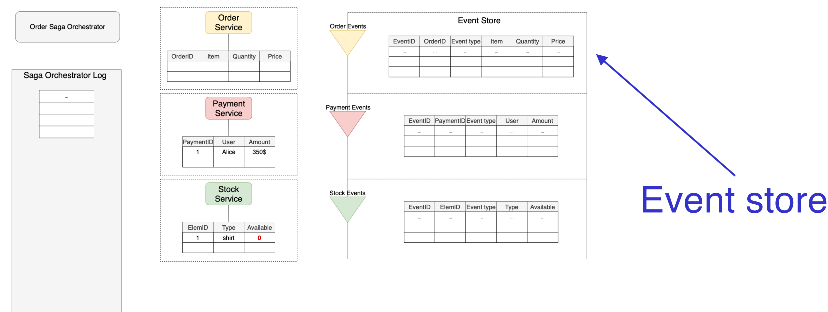
Patterns: Saga

- Example: orchestration-based saga
  - Source: MSc thesis by Andrea Cifola, see Microservice_SAGAexample.pdf



Patterns: Saga

- Example: orchestration-based saga



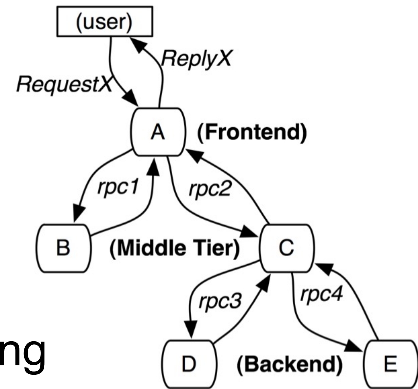
- We also use another pattern: event sourcing
  microservices.io/patterns/data/event-sourcing.html
  – Problem: a service that participates in a saga needs to atomically update the database and sends messages/events in order to avoid data inconsistencies
  – Solution: persist a sequence of domain events that represent state changes; each event in the sequence is stored in an *append-only* event store (a database of events)

Patterns: CQRS

- Problem: How to implement a query that retrieves data from multiple services in a microservice architecture? How to separate read and write load allowing you to scale each independently?
- Solution: define a view database, which is a read-only replica that is designed to support that query
  – Application keeps replica updated by subscribing to Domain events published by the service that own data
- Called Command Query Responsibility Segregation (CQRS), separates read and update operations for a data store
  microservices.io/patterns/data/cqrs.html

Monitoring microservices

- Service distribution, even at large scale: more difficult to monitor microservices apps, e.g., need to capture causal and temporal relationships among services
  - Performance and latency optimization
  - Root cause analysis
  - Service dependency analysis
  - Distributed context propagation
  - Distributed transaction monitoring



- Let's examine 2 patterns for monitoring microservices
  - Log aggregation
  - Distributed request tracing

Patterns: Log aggregation

- Problem: How to understand application behavior and troubleshoot problems?
- Solution: Use a centralized logging service that aggregates logs from each service instance. Users can search and analyze logs and configure alerts that are triggered when certain messages appear in logs
  - E.g., AWS Cloud Watch

✗ Centralized (if physical, not only logical)

✗ Handling large volume of logs requires substantial infrastructure

microservices.io/patterns/observability/application-logging.html

Patterns: Distributed tracing

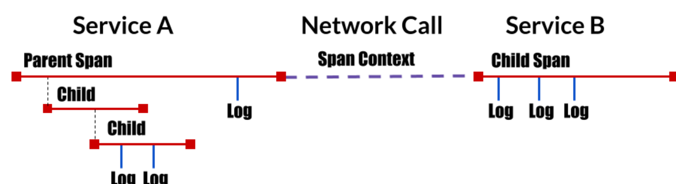
- Problem: How to understand complex app behavior and troubleshoot problems?
- Solution: Instrument services with code that
  – Assigns each external request a unique external request id
  – Passes id to all services that are involved in request handling
  – Includes id in all log messages
  – Records information (e.g., start time, end time) about the requests and operations performed in a centralized service

✗ Aggregating and storing traces can require significant infrastructure

microservices.io/patterns/observability/distributed-tracing.html

Monitoring microservices: tools

- Tools for distributed tracing
  – Most existing microservices frameworks have built-in application-layer tracing capabilities
  – Dapper:
    - Google's production distributed systems tracing infrastructure (paper)

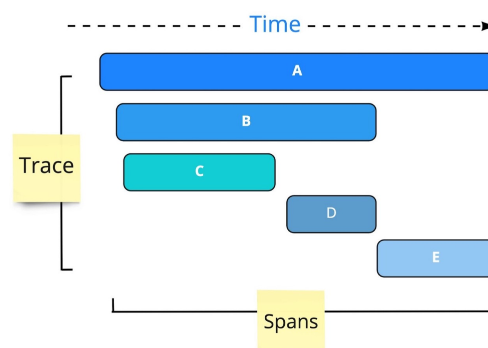


Monitoring microservices: tools

- Tools for distributed tracing
  - Most existing microservices frameworks have built-in application-layer tracing capabilities
  - Dapper:
    - Google's production distributed systems tracing infrastructure (paper)
  - Jaeger
    - Uses Spark/Flink for aggregate trace analysis
  - Zipkin
  - OpenTelemetry
    - Broad language support
    - Integrated with popular frameworks and libraries

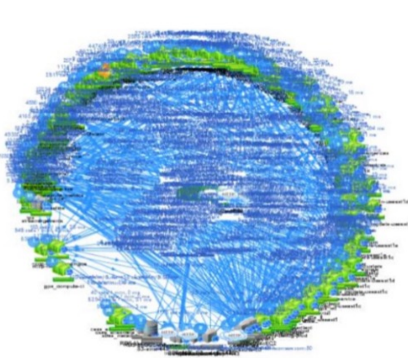
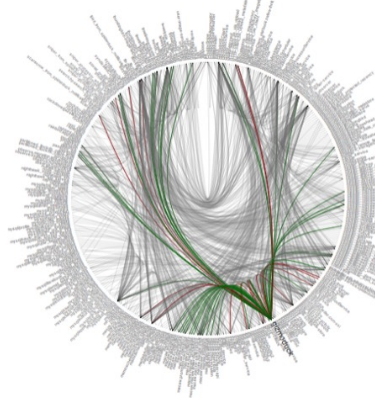
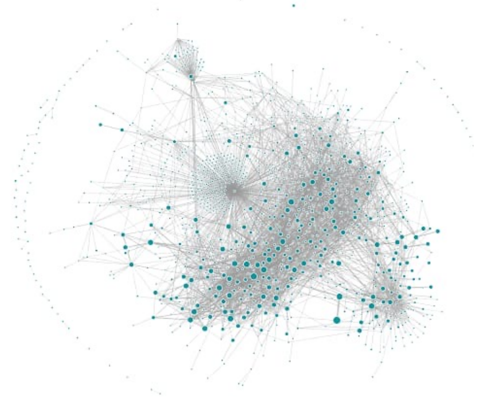
Monitoring microservices: spans and traces

- Distributed tracing systems works with spans and traces
  - *Span*: unit of work in an application (e.g., HTTP request, call to DB); must have an operation name, start time, and duration
  - *Trace*: collection/list of spans connected in a parent/child relationship (can also be thought of as directed acyclic graph of spans); traces specify how requests are propagated through services and other components



Some large-scale examples

- Netflix, Twitter, Uber: 500+ microservices

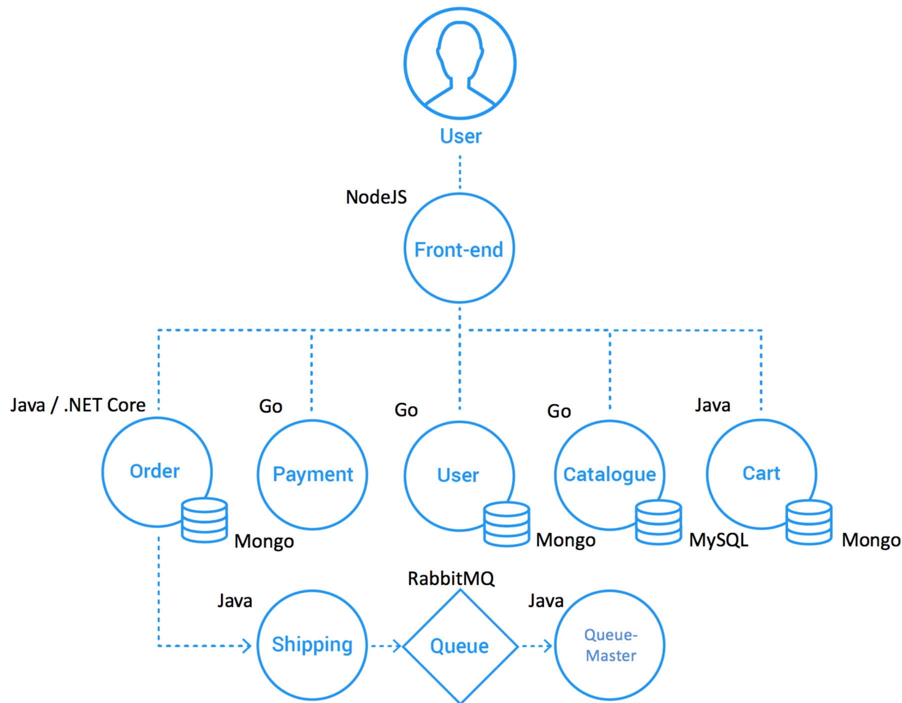
NETFLIX

Examples of microservices app

- Let's examine two microservices apps
  1. Sock Shop
  2. Online Boutique
- Both apps present microservices built in different programming languages
  - Programming languages are silos, but in the last 15 years renaissance in programming language diversity: need for polyglot programmers
- How to realize a polyglot application whose microservices are written in different programming languages?
  1. REST and JSON (as message interchange format): see example 1
  2. gRPC and protocol buffers (as IDL and message interchange format): see example 2

Example 1: Sock shop

- Online shop microservices-demo.github.io/
  github.com/microservices-demo/microservices-demo

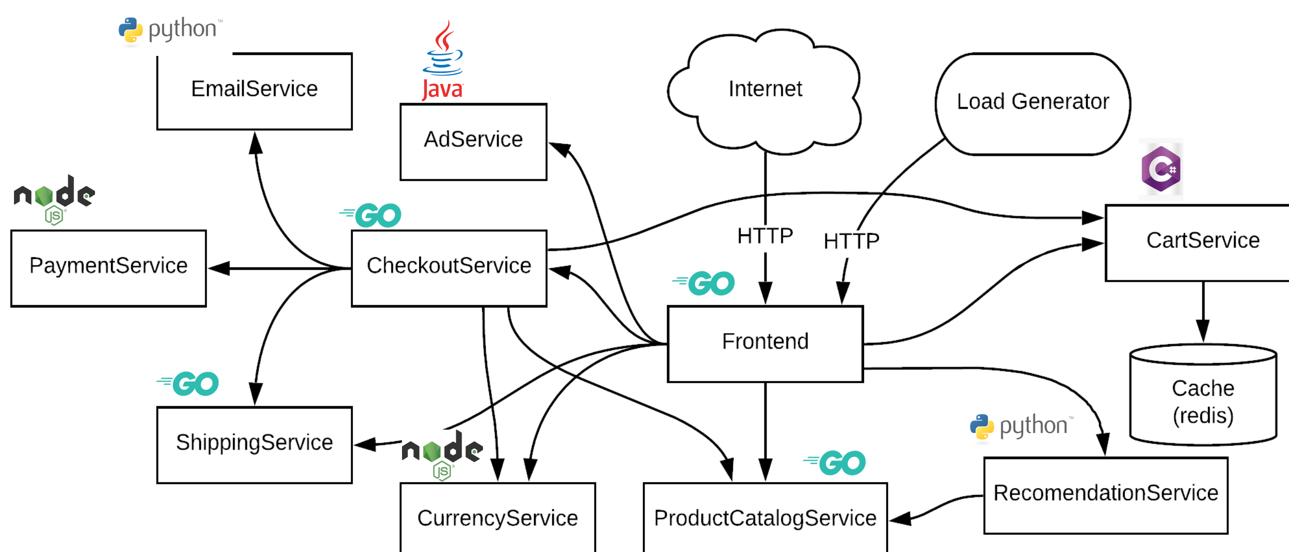


Example 1: Sock shop

- Built using SpringBoot, Go and Node.js and packaged in Docker containers
- Cross-platform: deployment to different orchestrators
  - Docker Compose, Docker Swarm, Kubernetes, …
- Services communicate using REST over HTTP
- Also polyglot data stores (MongoDB, MySQL)
- Includes load test using Locust to simulate user traffic to Sock Shop
- Optional: uses Weave Scope to automatically detect containers and hosts

Example 2: Online boutique

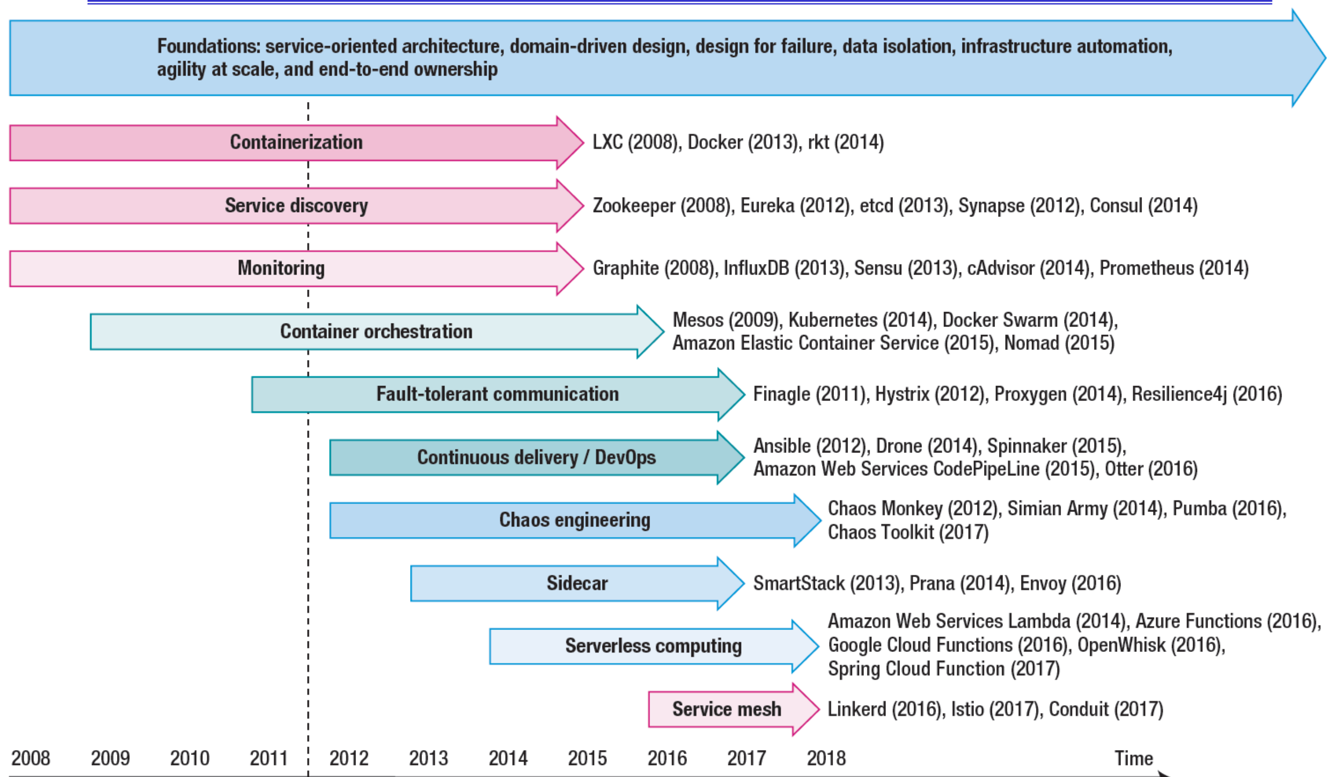
- Online store developed by Google
  github.com/GoogleCloudPlatform/microservices-demo



Example 2: Online boutique

- Composed of 11 microservices written in different languages that communicate using gRPC
- Used by Google to demonstrate use of many technologies:
  - Kubernetes and Google Kubernetes Engine (GKE)
  - gRPC: already examined
  - Istio: service mesh
  - Cloud Operations: integrated monitoring, logging, and trace managed services for apps and systems running on Google Cloud
  - OpenCensus: to collect application metrics and distributed traces, then transfer data to a backend
  - Skaffold: command line tool that facilitates continuous development for Kubernetes applications

Microservice technologies timeline



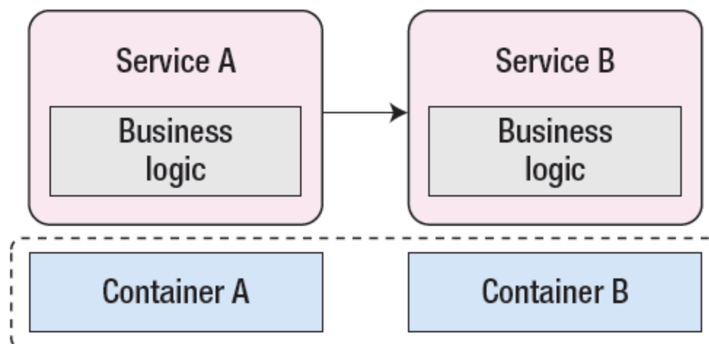
From "Microservices: The Journey So Far and Challenges Ahead".

Generations: at the beginning

- 4 generations of microservice architectures
- 1st generation based on:
  - **Container-based virtualization**
  - **Service discovery** (e.g., Eureka, etcd, ZooKeeper)
    - etcd: distributed reliable key-value store (e.g., used by Kubernetes as primary data store)
    - Eureka: REST based service developed by Netflix; used in AWS cloud for locating services for load balancing and failover of middle-tier servers
  - **Monitoring** (e.g., Graphite, InfluxDB and Prometheus)
    - Enable runtime monitoring and analysis of microservice resources behavior at different levels of detail

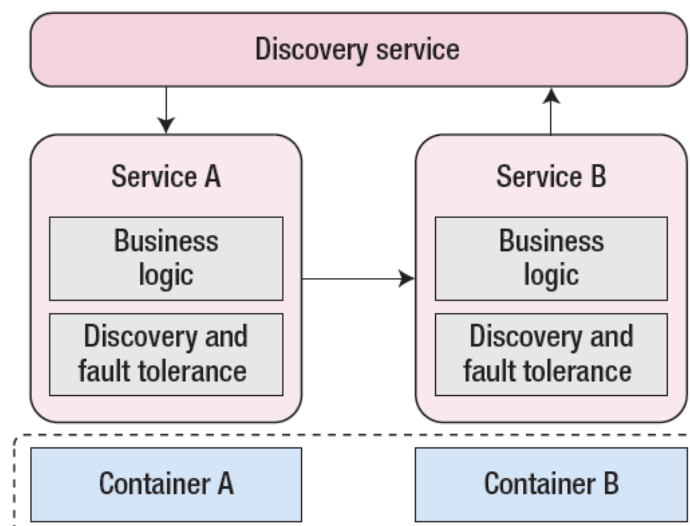
Generations: container orchestration

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



Generations: service discovery and fault tolerance

- 2nd generation based on **discovery services** and **fault-tolerant communication libraries**
  - Let services communicate more efficiently and reliably

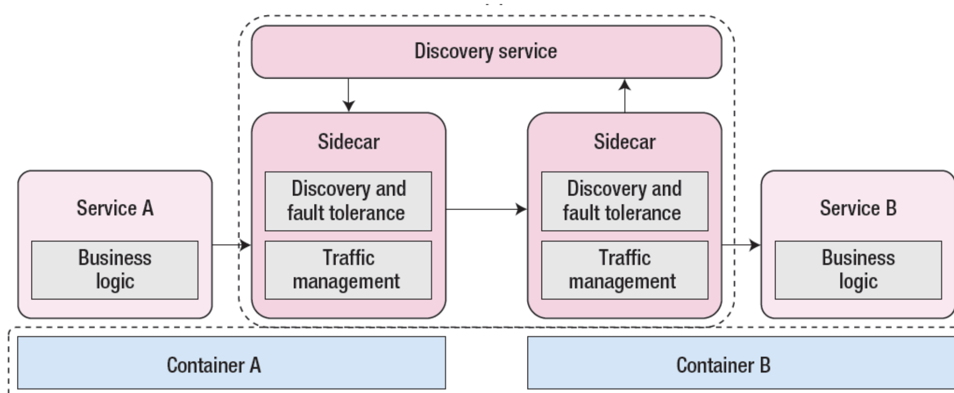


Generations: service discovery and fault tolerance

- Examples of discovery services and fault-tolerant communication libraries
  - **Consul**: service discovery (now a more powerful service mesh)
  - **Finagle**: fault tolerant, protocol-agnostic RPC system designed and used in production by Twitter
  - **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 DS
    - Only in maintenance mode

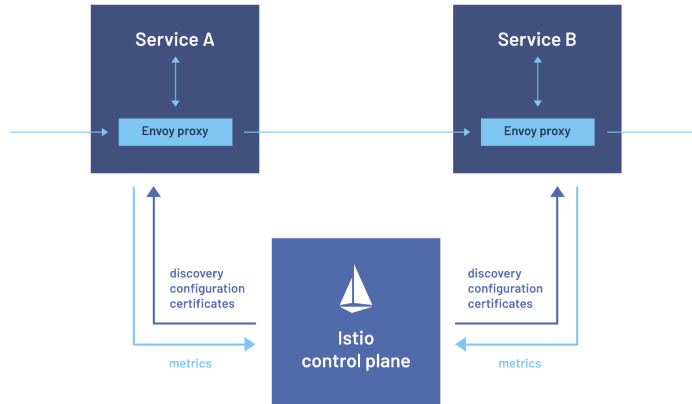
Generations: sidecar and service mesh

- 3rd generation based on **sidecar** (or service proxy) and **service mesh** technologies (e.g., Envoy and Istio)
  - 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



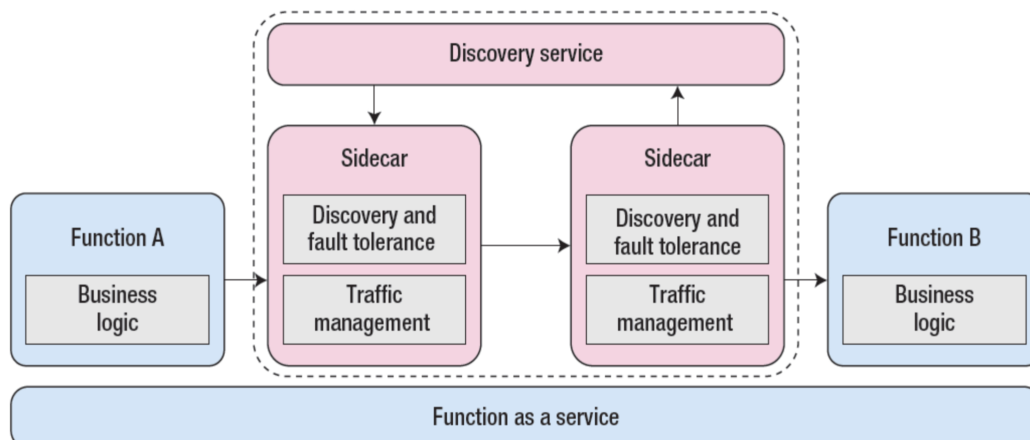
Service mesh

- Dedicated infrastructure layer added to microservice app for facilitating service-to-service communications between microservices using a proxy
- Allows you to transparently add capabilities like observability, traffic management (including load balancing) and security, without adding them to code



Generations: serverless

- 4th generation based on **Function as a Service (FaaS)** and **serverless computing** to further simplify microservice development and delivery



Serverless computing

- Cloud computing model which aims to abstract server management and low-level infrastructure decisions away from users
- Users can develop, run and manage application code (i.e., **functions**), without no worry about provisioning, managing and scaling computing resources
- Runtime environment is fully managed by Cloud provider
- Serverless: functions still run on "servers" somewhere but we don't care
- Function as a Service (**FaaS**) often as synonym
  - Still some discussion

Serverless computing: characteristics

- Ephemeral compute resources
  - May only last for one function invocation
  - ✗ *Cold start*: when a request arrives and no container is ready to serve it, function execution must be delayed until a new container is launched
- Automated (i.e., zero configuration) elasticity
  - Compute resources auto-scale transparently from zero to peak load and back in response to workload shifts
- True pay-per-use
  - Pay only for consumed time, rather than on pre- purchased units of capacity, e.g., AWS Lambda pricing

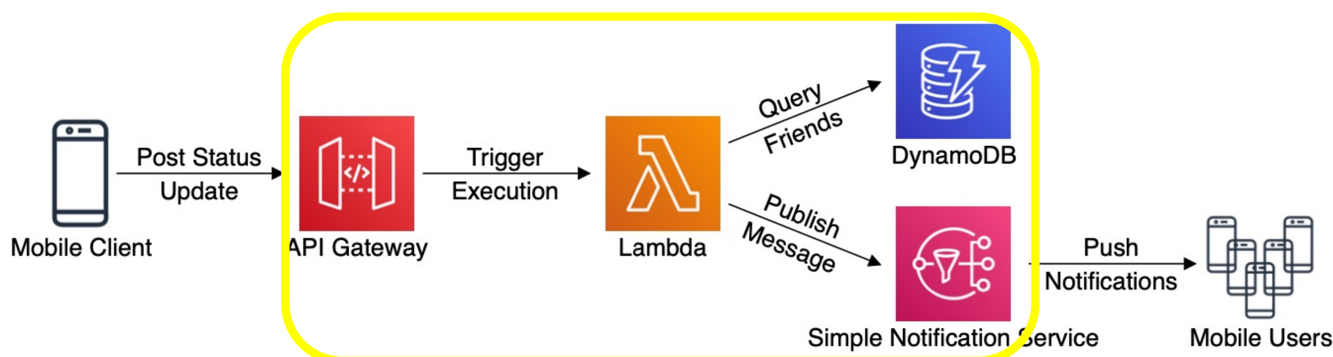
Architecture	Duration	Requests
x86 Price		
First 6 Billion GB-seconds / month	$0.0000166667 for every GB-second	$0.20 per 1M requests
Next 9 Billion GB-seconds / month	$0.000015 for every GB-second	$0.20 per 1M requests
Over 15 Billion GB-seconds / month	$0.0000133334 for every GB-second	$0.20 per 1M requests

Serverless computing: characteristics

- Event-driven
  – When an event is triggered (e.g., file uploaded to storage, message to be consumed in queue), a piece of infrastructure is allocated dynamically to execute the function code
- Can simplify the process of deploying code into production
  – Scaling, capacity planning and maintenance operations are hidden from developers or operators

Example of serverless application

- Mobile backend for a social media app
  1. Users compose status update and send it using mobile clients
  2. Platform orchestrates ops needed to propagate update inside the social media platform and to user's friends using serverless technology (AWS Lambda)
  3. Each friend receives updates on their social media app



Serverless Cloud services

- Several Cloud providers offer serverless computing on their public clouds as fully managed service
  - AWS Lambda
    - See hands-on course
    - Includes functions at the edge (Lambda@Edge)
  - Azure Functions
  - Google Cloud Functions
  - IBM Cloud Functions
- Limited knobs to control performance of functions
  - Developers can configure only amount of memory allocated to function: amount of virtual CPU is proportional to amount of memory
- Cloud platforms also offer other supporting services (e.g., event notification, storage, message queue, DB) that are necessary for operating a serverless ecosystem

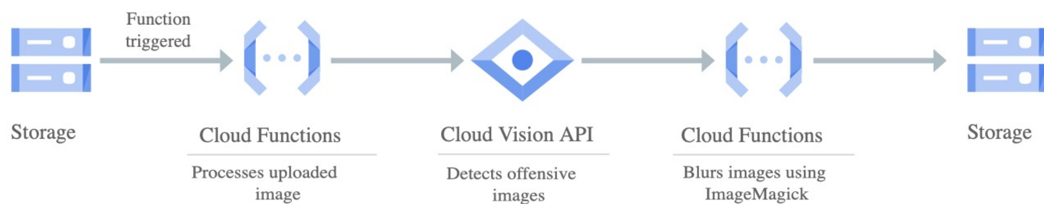
Example: Google Cloud Functions

- "Hello World" FaaS example from Google using Go
  - HTTP response that displays "Hello, World!"

```
// helloGet is an HTTP Cloud Function.
func helloGet(w http.ResponseWriter, r *http.Request) {
        fmt.Fprint(w, "Hello, World!")
}
```

  - Plus some initialization code, see full example cloud.google.com/functions/docs/tutorials

Example: Google Cloud Functions



- A more complex example
  - Function execution is triggered from storage when an image is uploaded to a Cloud Storage bucket
  - Function uses Cloud Vision API to detect violent or adult content
  - When violent or adult content is detected in an uploaded image, a second function is called to download the offensive image: it uses ImageMagick to blur the image, and then uploads the blurred image to the output bucket

  Code: cloud.google.com/functions/docs/tutorials/imagemagick

Serverless computing: state

- Stateless functions are easy to manage (horizontal scalability, fast recovery, …)
  - But stateless functions are not enough for a broad range of applications and algorithms
- How to handle stateful processing?
  - State can be external (e.g., handed over to external DB)
  - Issues to address:
    - Efficient access to shared state, so to keep auto-scaling benefits
    - Programming support, e.g., Azure Durable Functions
- How to handle transactions?

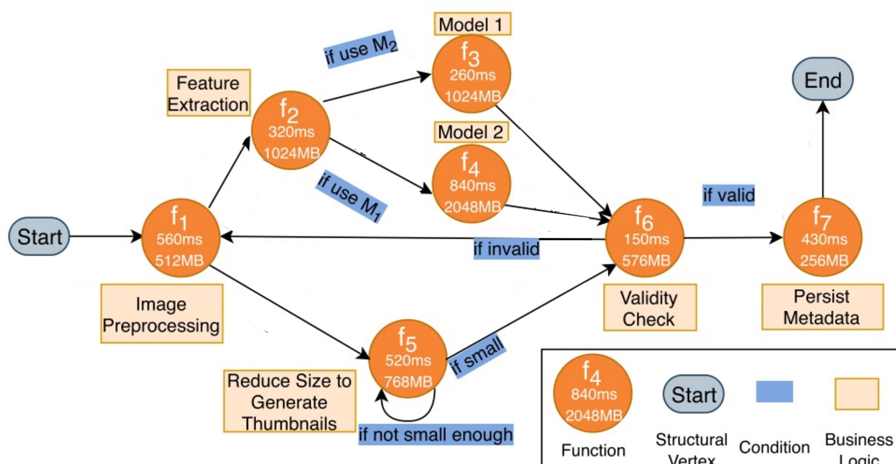
Serverless computing: challenges and limitations

- Performance
  - Cold starts
    - "The first time you deploy a function it may take several minutes as we need to provision the underlying infrastructure to support your functions. Subsequent deployments will be much faster." (Google Cloud Functions)
- Runtime and programming language support
  - E.g., on Google: Node.js, Python, Go, Java, C#, Ruby, PHP
  - Language runtime impacts on performance and cost of serverless functions
- Resource limits
  - E.g., on AWS memory between 128 and 10240 MB per function
- Lack of standards
- Risk of vendor lock-in

Lower flexibility

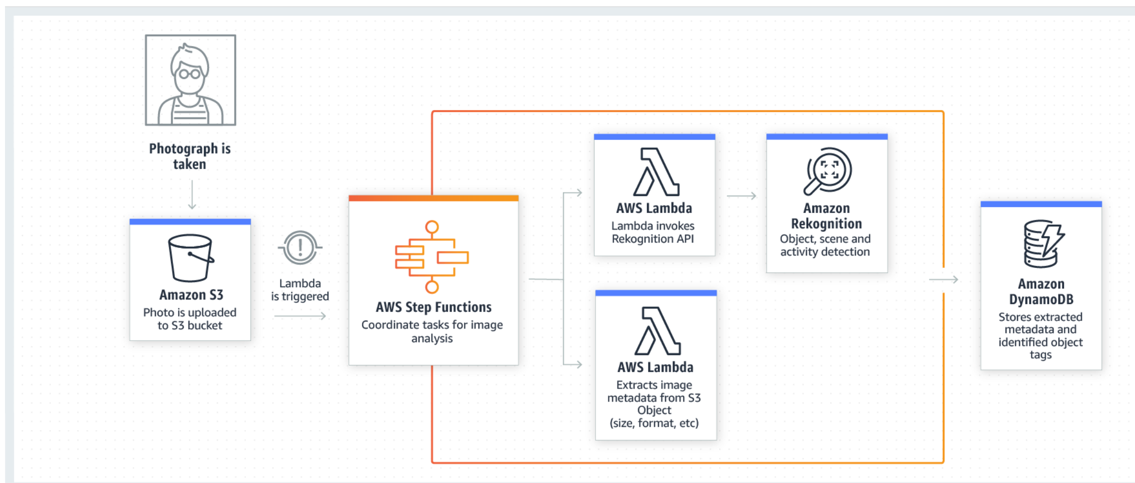
Composition of serverless functions

- Write small, simple, stateless functions
  - Complex functions are hard to understand, debug, and maintain
  - Separate code from data structures
- Then compose them in a workflow



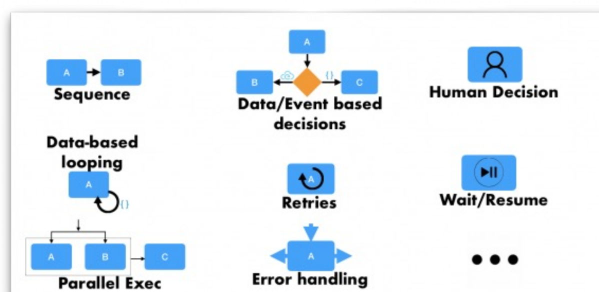
Example: AWS Step Functions

- AWS Step Functions: serverless orchestration service that allows developers to coordinate multiple Lambda functions into workflows
- Example: process photo after its upload in S3



Composition of serverless functions

- Not yet a standard solution to define workflows
- An attempt: CNCF [Serverless Workflow]
  - Vendor-neutral, open-source, and community-driven ecosystem for defining and running domain-specific language (DSL) based workflows
  - Developer can use YAML and JSON to describe workflows
  - Supports invocation of both RESTful and event-triggered functions
  - Supports many control-flow logic constructs



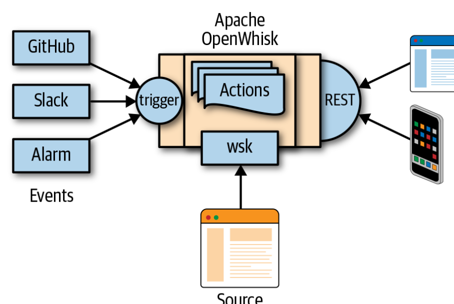
Open-source FaaS platforms

- Can run on commodity hardware
- Most platforms rely on Kubernetes for orchestration and management of serverless functions
  - Configuration management of containers
  - Container scheduling and service discovery
  - Elasticity management

- Prominent platforms
  - Apache OpenWhisk
  - OpenFaaS
  - Fission
  - Knative
  - Nuclio

OpenWhisk

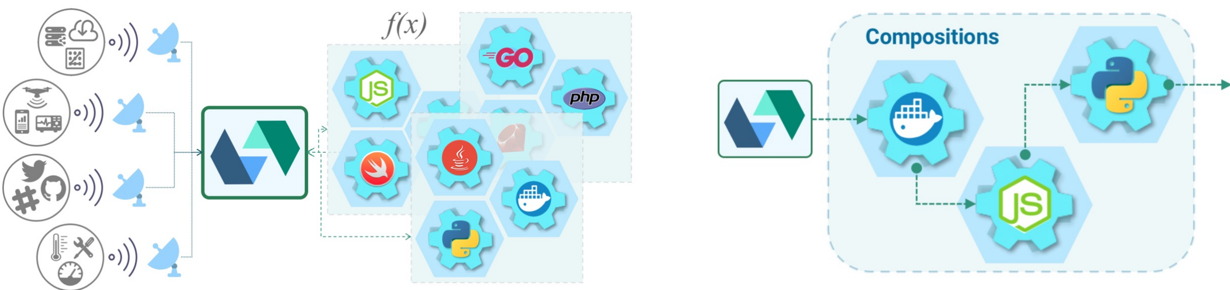
- Open-source, distribute serverless platform that executes functions in response to events at any scale
openwhisk.apache.org



- Based on Docker containers
- Multiple container orchestration frameworks

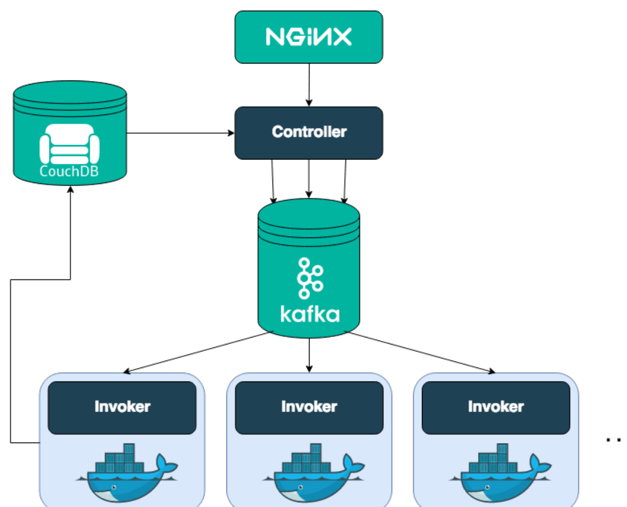
OpenWhisk

- Developers write functional logic (called actions)
  - In any supported programming language
  - Dynamically scheduled and run in response to associated events (via triggers) from external sources (feeds) or from HTTP requests
- Functions can be combined into compositions



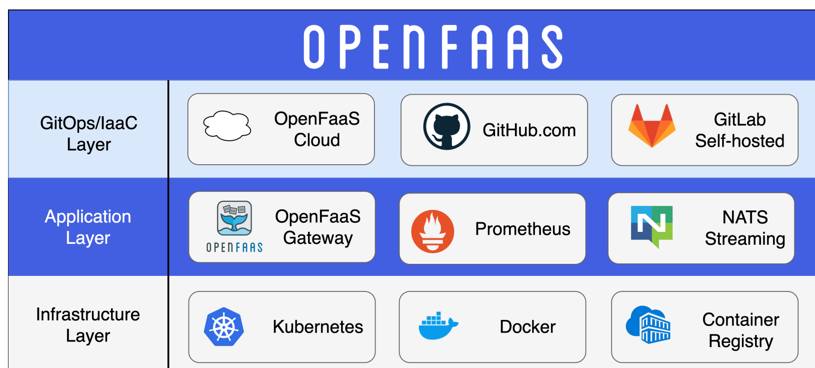
OpenWhisk architecture

- Internal architecture powered by multiple distributed frameworks: Apache Kafka, CouchDB, Docker and NGINX



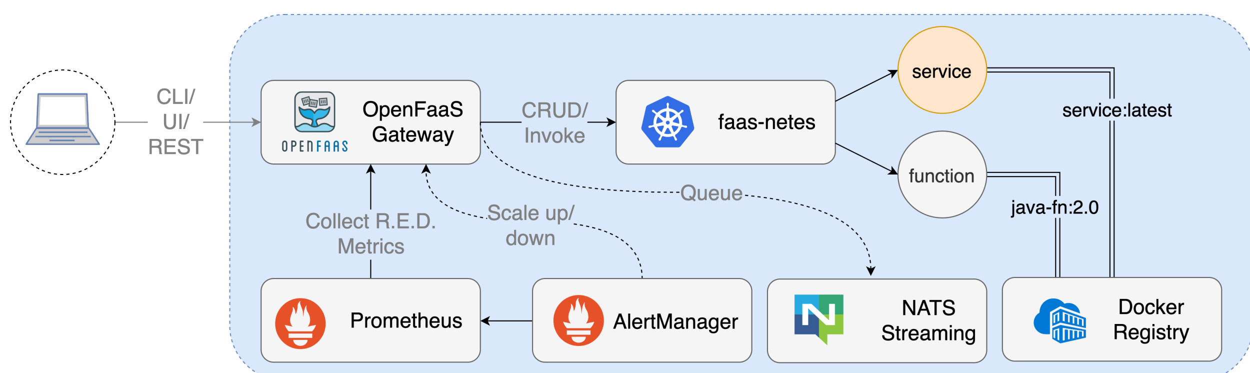
OpenFaaS

- Open-source FaaS framework for building functions on top of Docker and Kubernetes www.openfaas.com
- OpenFaaS stack
  - Gateway provides an external route into functions, collects metrics and scale functions
  - Prometheus provides metrics and enables auto-scaling
  - NATS provides asynchronous execution and queuing



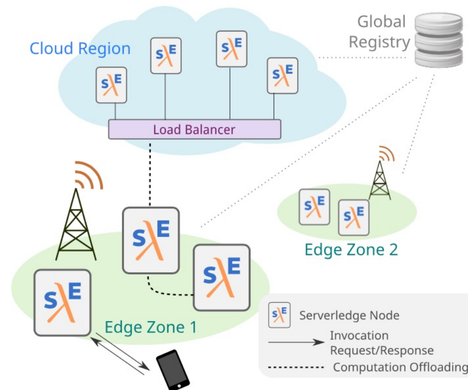
OpenFaaS

- Conceptual workflow
  - OpenFaaS Gateway can be accessed through its REST API, via CLI or through UI
  - Prometheus collects metrics made available via Gateway's API and used for auto-scaling
  - NATS Streaming enables long-running tasks or function invocations to run in background



Serverless for the Edge-Cloud Continuum

- Open-source FaaS frameworks have centralized functionalities (e.g., scheduling): not suitable for Edge-Cloud continuum
- We are working on a decentralized FaaS framework called Serverledge: thesis opportunities!



Microservices: References and resources

- Lewis and Fowler, Microservices
- Lewis and Fowler, Microservice Guides
- Richardson, Microservice Architecture
- Jamshidi et al., Microservices: The Journey So Far and Challenges Ahead, *IEEE Software*, 2018

Serverless: References and resources

- Roberts, Serverless Architectures
- Castro et al., The Rise of Serverless Computing, ACM Comm., 2019
- Schleier-Smith et al., What serverless computing is and should become: the next phase of cloud computing, ACM Comm., 2021