TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Microservices and Serveless Computing

Corso di Sistemi Distribuiti e Cloud Computing

A.A. 2024/25

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)

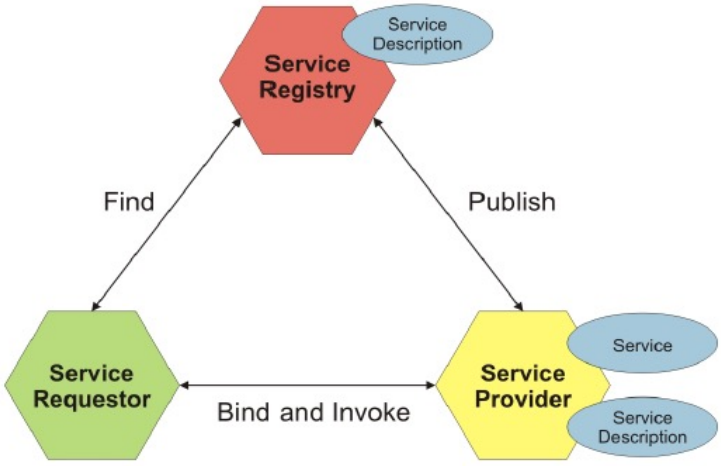
The ancestors: Service Oriented Architecture

- **Service Oriented Architecture** (**SOA**): architectural paradigm for designing loosely coupled distributed sw systems
- Definition https://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 https://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

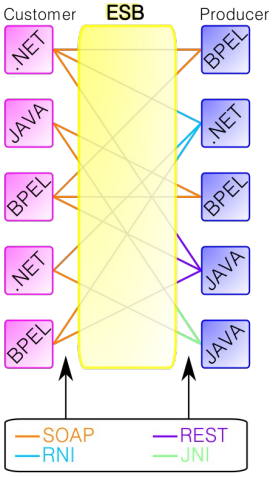


Web services

- Web services: implementation of SOA
- Definition https://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 SOAP 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 complement 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:
  - ✓ Scale out/in microservice instance by changing number of container replicas
  - ✓ Scale up/down microservice instance assigning more/less resources to container
  - ✓ Isolate microservice instance
  - ✓ Set 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
  - Interaction 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 of teams

Microservices: concerns

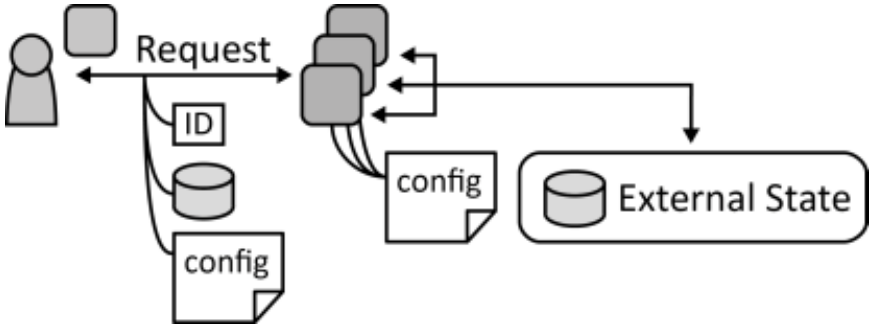
- Increased network traffic
  - Microservice calls over a network cost more in terms of network latency
- Higher complexity
  - Increased operational complexity (e.g., deployment)
  - Global testing and debugging is more complicated

Microservices and scalability

- How to achieve scalability of microservices?
  - Use multiple instances of same microservice and load balance requests across multiple instances
- How to improve scalability of microservices?
  - Microservice state is complex to manage and scale
  - Prefer stateless services: scale better and faster than stateful services

Stateless service

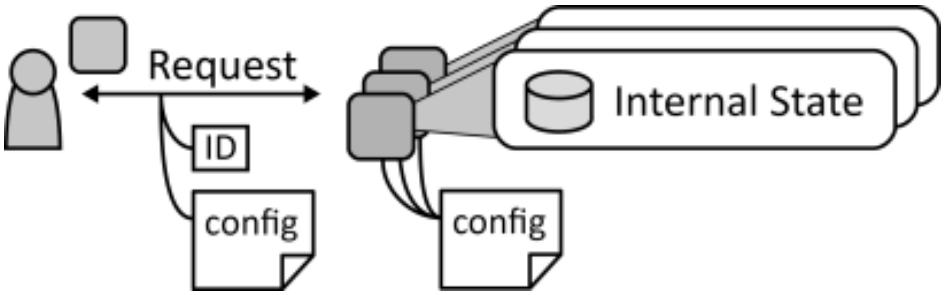
- **Stateless** service: state is handled external of service to ease its scaling out and to make application more tolerant to service failures



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

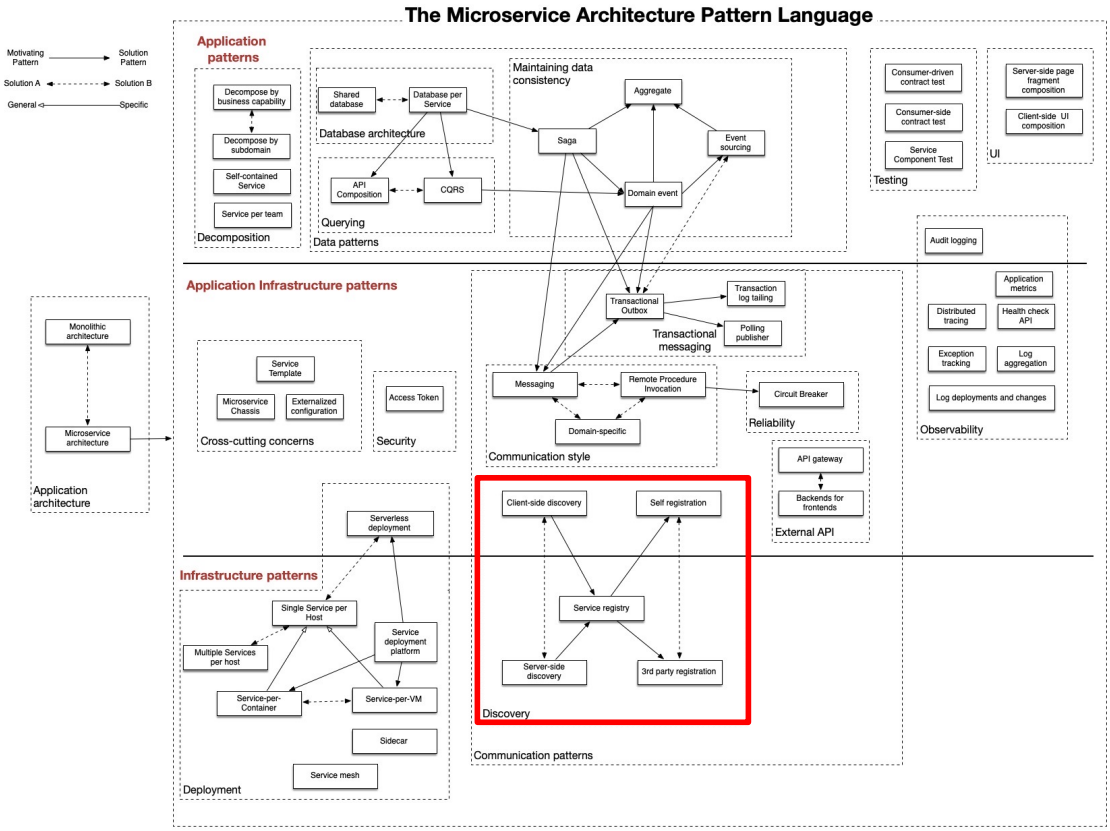
Stateful service

- **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?



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

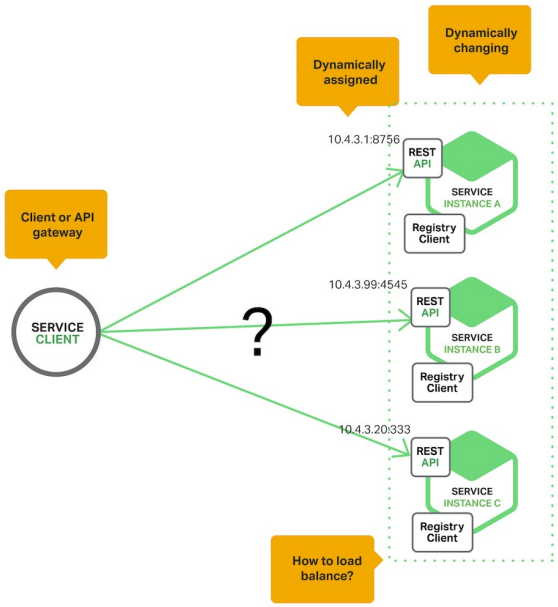
A full view of microservice patterns



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

Service discovery

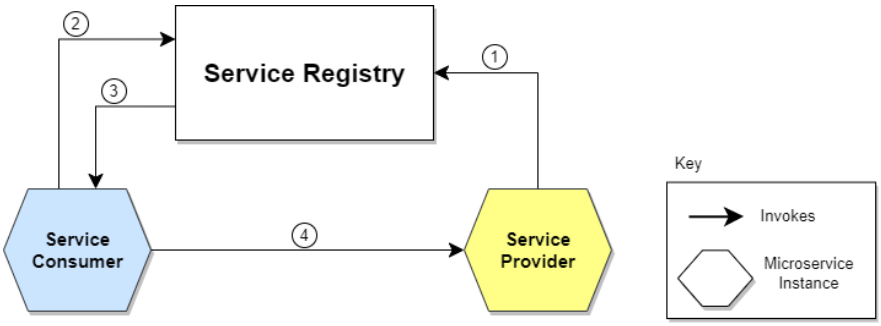
- Why service discovery? The client of a microservice needs to discover the network location of a microservice instance
  - Microservice instances have dynamically assigned network locations (IP address and port) and their set changes dynamically because of auto-scaling, failures, and upgrades



- Service discovery provides
  - Mechanism for a microservice instance to register
  - Way to find the service once it has registered

Service discovery: patterns

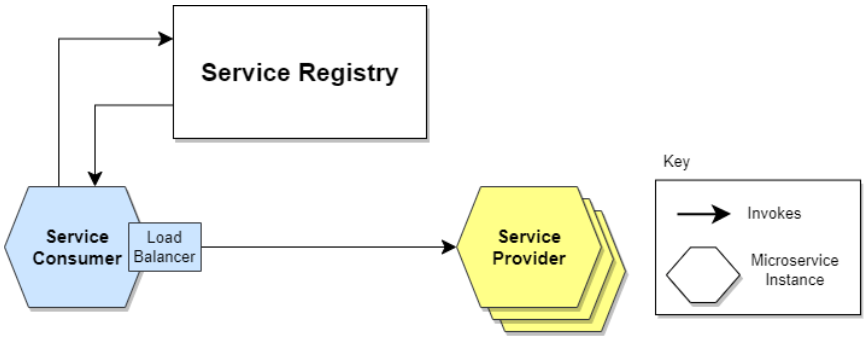
1. Service registry
   - A database of services, their instances and their locations
   - Service instances are registered with service registry on startup and deregistered on shutdown
   - A clients query the service registry to find the available instances of a service



Service discovery: patterns

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

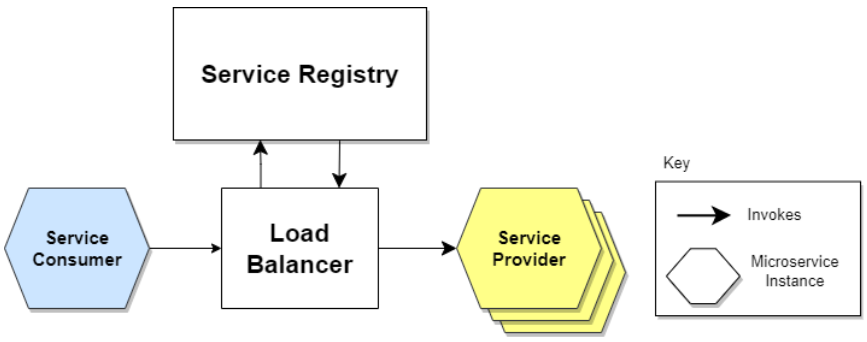
   https://microservices.io/patterns/client-side-discovery.html



Service discovery: patterns

3. 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

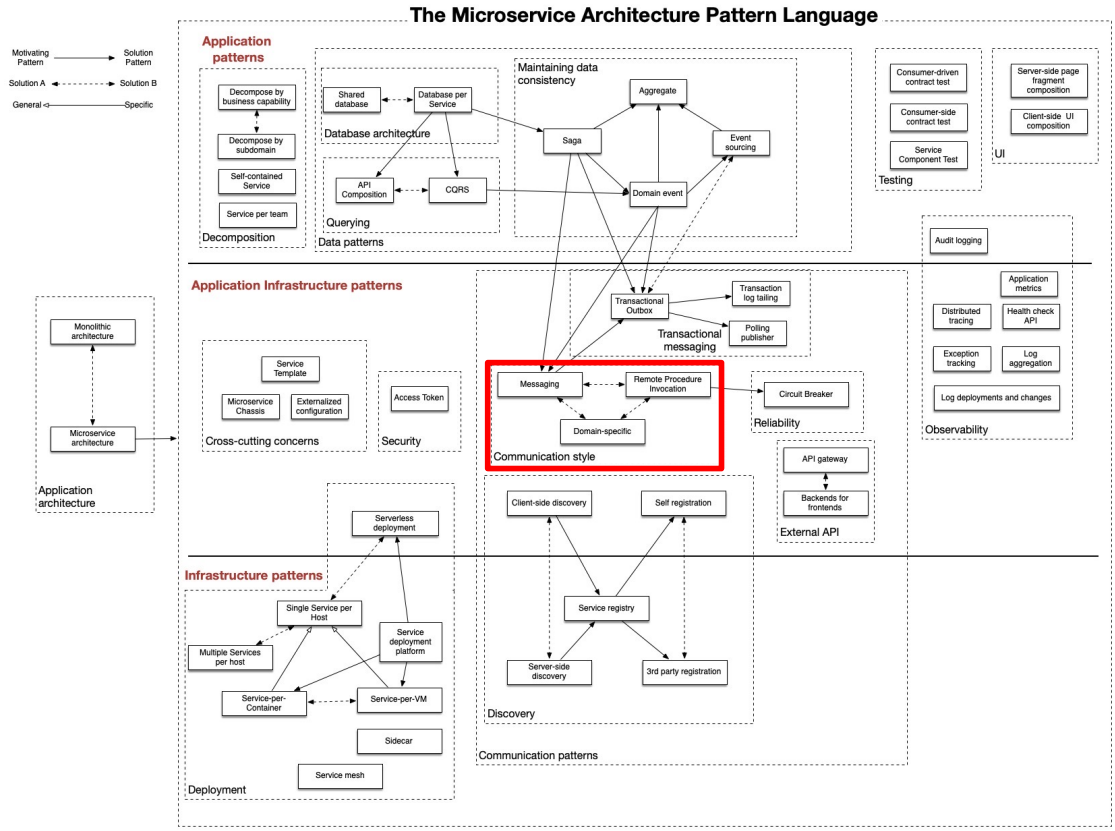
   https://microservices.io/patterns/server-side-discovery.html



Integration of microservices

- Let's consider two issues related to integration of microservices
  - Synchronous vs. asynchronous communication (or RPC vs. messaging)
  - Orchestration vs. choreography

A full view of microservice patterns



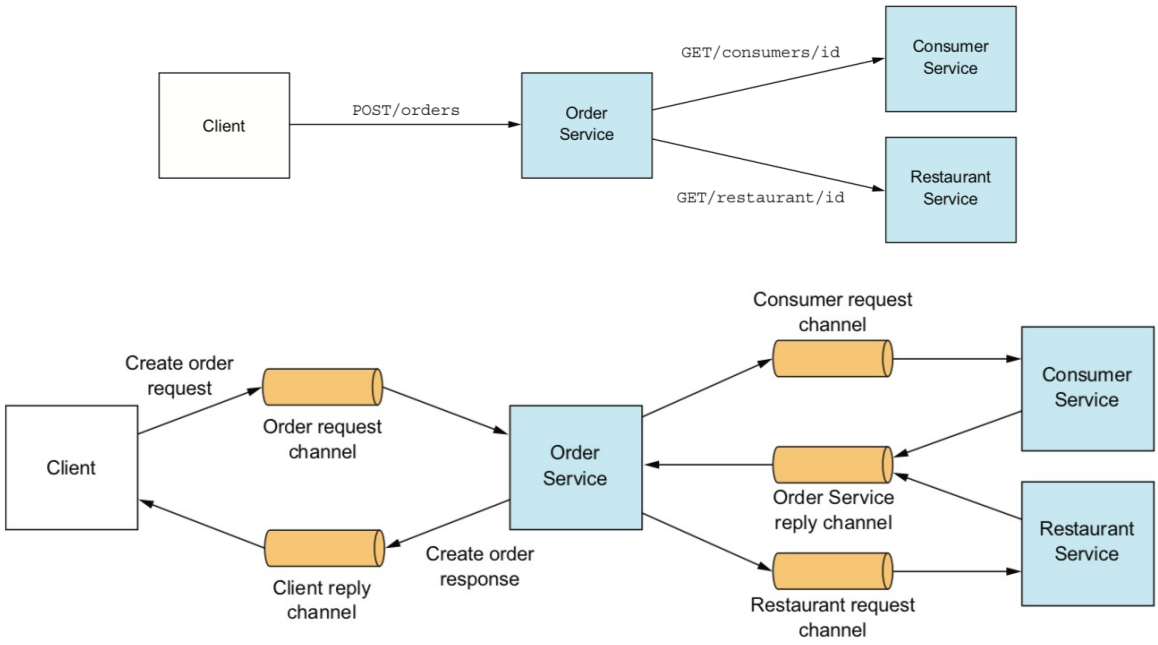
Learn-Build-Assess Microservices http://adopt.microservices.io

Synchronous vs. asynchronous

- Should communication be synchronous or asynchronous (or RPC vs. messaging)?
  - 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

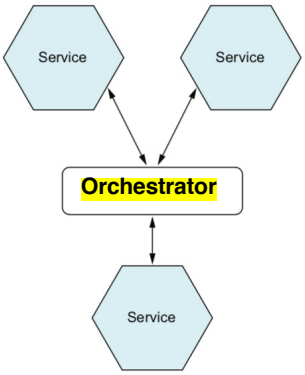
Synchronous vs. asynchronous

- Example of synchronous communication vs. asynchronous communication



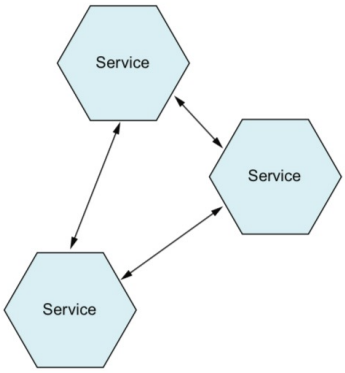
Orchestration and choreography

- Microservices can interact among them according to 2 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



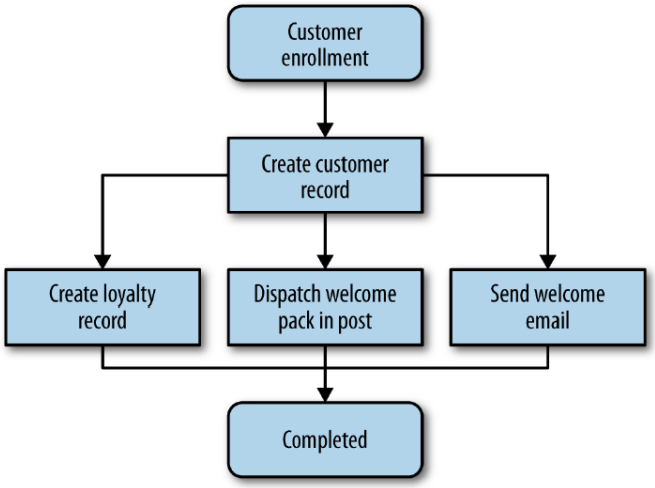
Orchestration and choreography

- **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

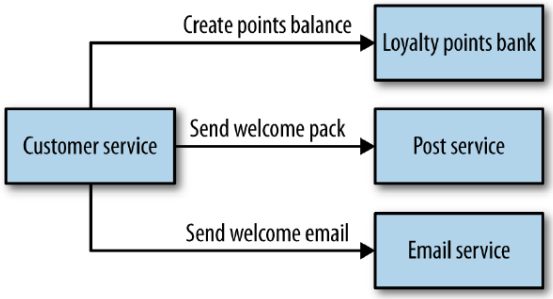
- Example: workflow for customer creation, i.e., process for creating a new customer



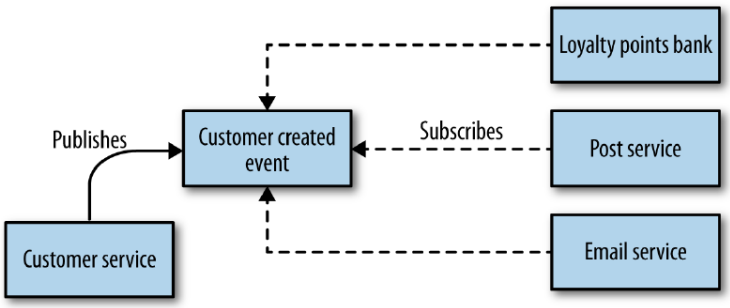
From: S. Newman, “Building Microservices”, O’Really, 2015.

Example: orchestration and choreography

Orchestration



Choreography

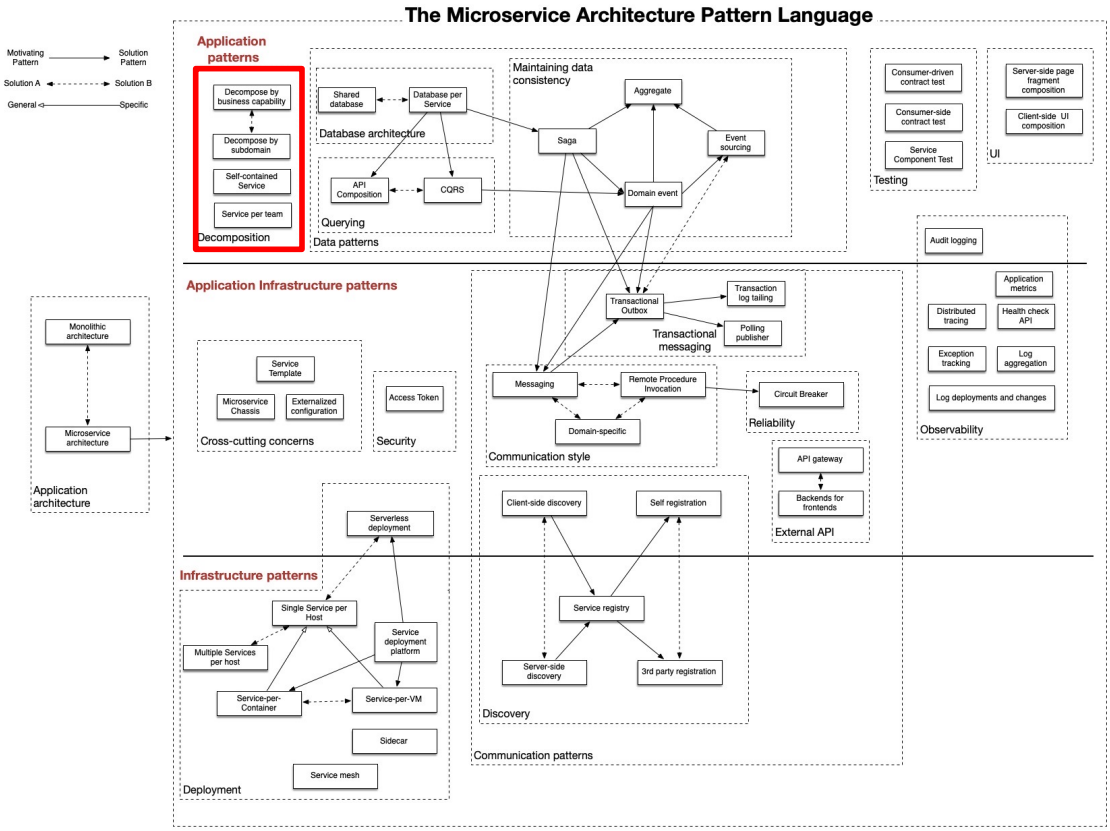


Orchestration vs choreography

- 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

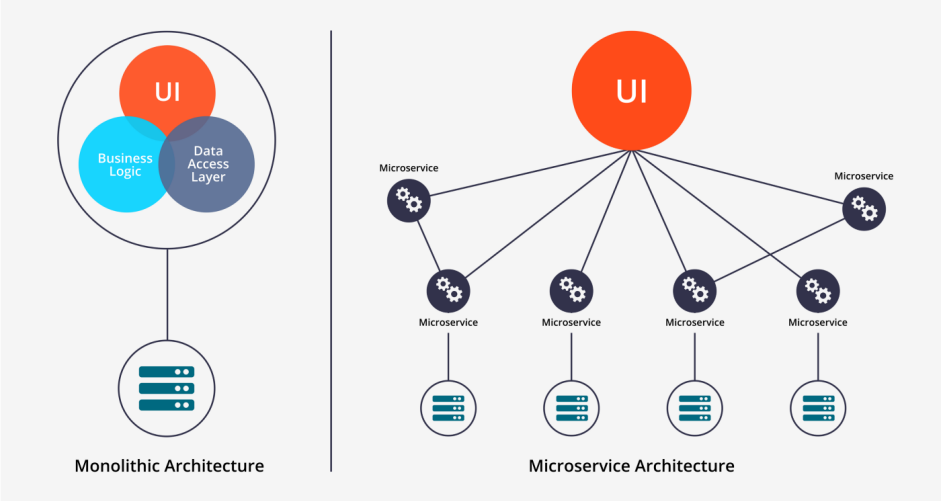
A full view of microservice patterns



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

Decomposition patterns

- Let's consider how to decompose a monolithic application into microservices
  - Monolithic: application as a single deployable unit
- Mostly an art, no winner strategy but rather a number of strategies https://microservices.io/patterns



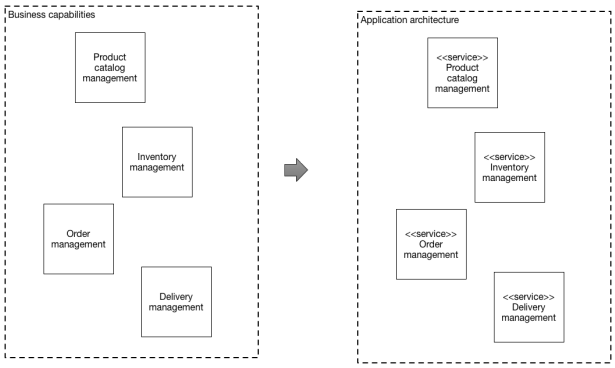
Decomposition patterns

- What to consider
  - Architecture must be stable
  - Each service must be cohesive
    - A service should implement a small set of strongly related functions
  - Services must conform to Common Closure Principle to ensure that each change affect only one service
    - Things that change together should be packaged together
  - Services must be loosely coupled
    - Each service as an API that encapsulates its implementation, which can be changed without affecting clients
  - A service should be testable
  - Each service should be small enough to be developed by a "two pizza" team (6-10 people)
  - Each team that owns one or more services must be autonomous, minimal collaboration with other teams

Main decomposition patterns

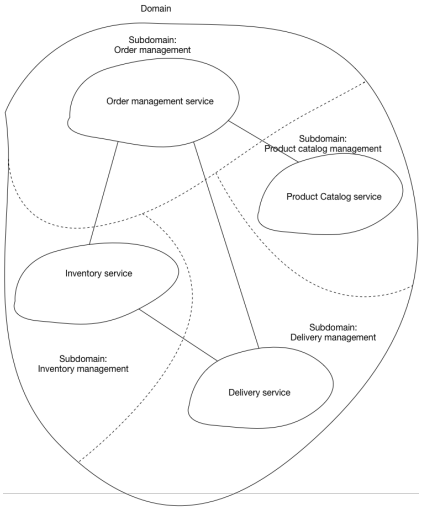
- Let's consider e-commerce app that takes orders from customers, verifies inventory and available credit, and ships them

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, *Customer Management* is responsible for customers



Main decomposition patterns

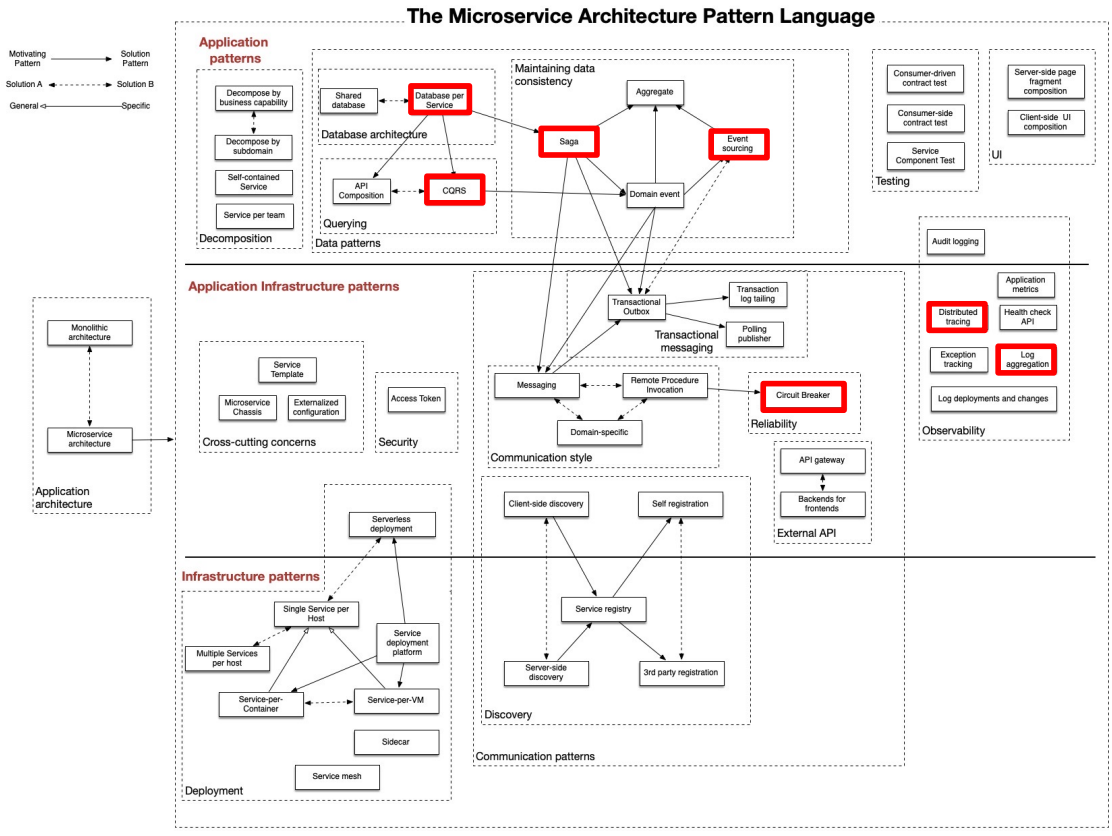
2. Decompose by domain-driven design (DDD) subdomain
   - A domain consists of multiple subdomains; each subdomain corresponds to a different part of the business
   - E.g., *Order Management*, *Inventory*, *Product Catalogue*, *Delivery*



Design patterns

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

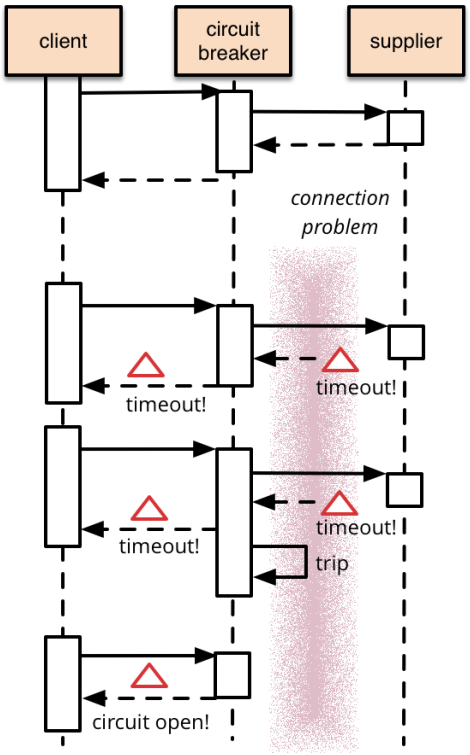
A full view of microservice patterns



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

Reliability patterns: Circuit breaker

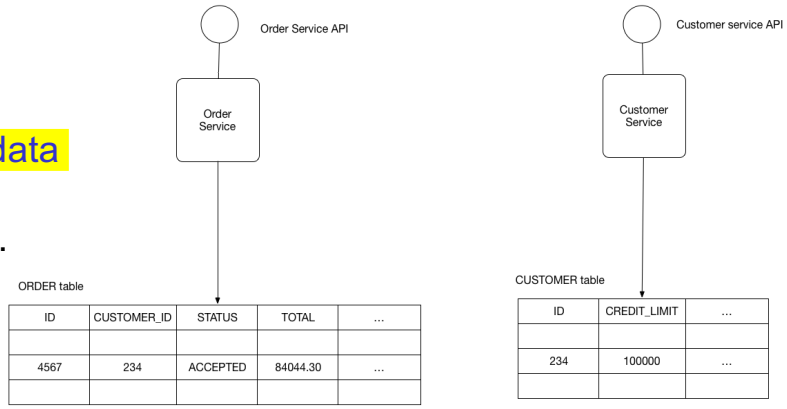
- Problem: How to prevent a network or service failure from cascading to other services?
- Solution: A service client invokes a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker
  - When the *number of consecutive failures* crosses a threshold, the circuit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



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

Data patterns: Database per service

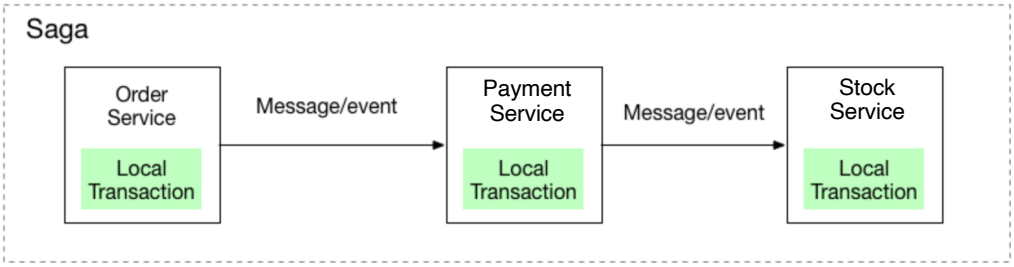
- Problem: which database architecture?
- Solution: keep each microservice's persistent data private to that service and accessible only via its API. Service transactions only involve its DB
- Pros and cons
  - ✓ Helps ensure that services are loosely coupled
  - ✓ Each service can use the most convenient DB type (e.g., KV data store, graph database)
  - ✗ More complex to implement transactions that span multiple services
  - ✗ Complexity of managing multiple DBs
    - But do not need to provision a DB server for each service
    - Options: private-tables-per-service, schema-per-service, database-server-per-service



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

Data patterns: Saga

- Problem: each service has its own DB, however some transactions span multiple services: how to maintain data consistency across services *without* using distributed transactions (e.g., two-phase commit protocol)?
- Solution: implement each transaction that spans multiple services as a saga
- **Saga:** sequence of local transactions
  - Each local transaction updates its DB 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)

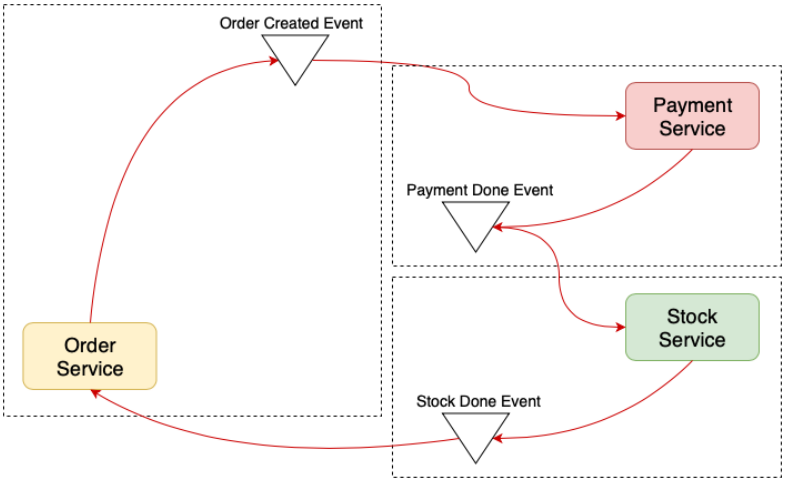


https://microservices.io/patterns/data/saga.html

Data 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

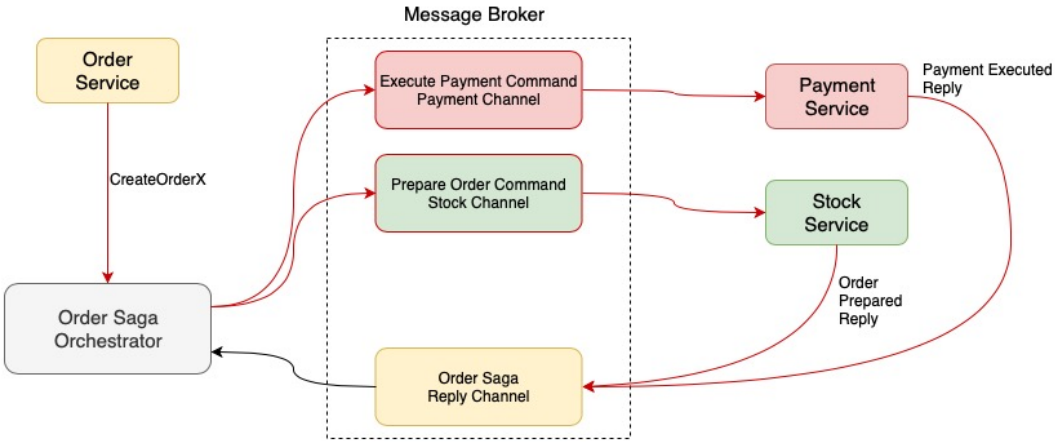
Choreography



Data 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

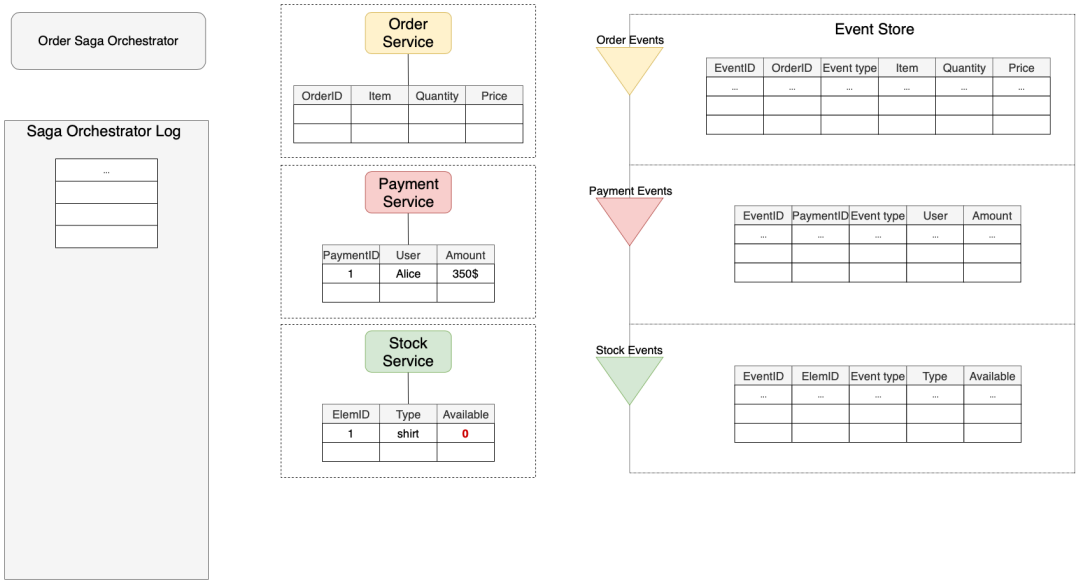
Orchestration



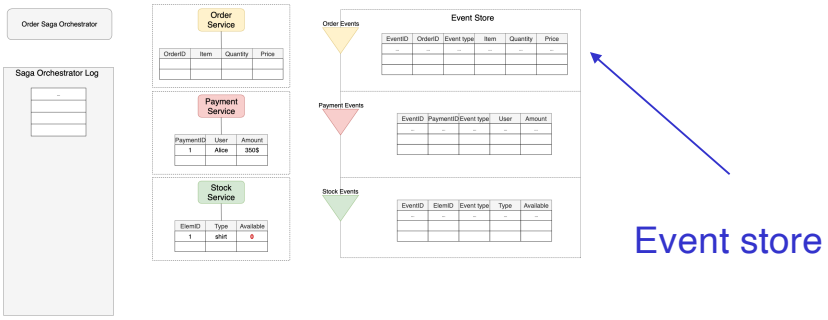
Data patterns: orchestration-based Saga

- Let's consider orchestration-based saga
  - Source: MSc thesis by Andrea Cifola

http://www.ce.uniroma2.it/courses/sdcc2122/slides/Microservice_SAGAexample.pdf



Data patterns: orchestration-based Saga



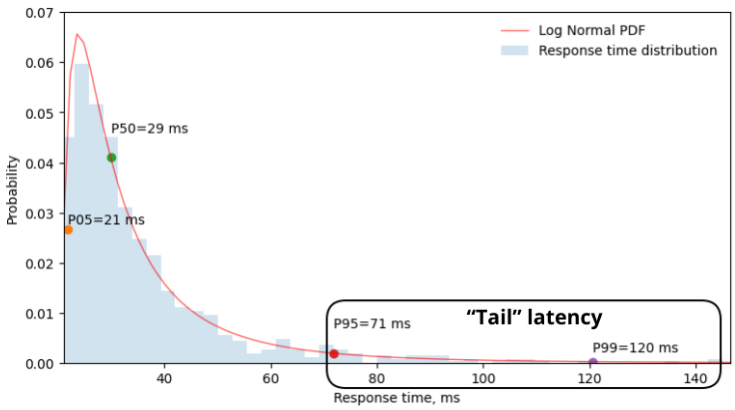
- We also use another pattern: event sourcing
  https://microservices.io/patterns/data/event-sourcing.html
  - Problem: a service that participates in a saga needs to atomically update the DB and sends messages/events in order to avoid data inconsistencies
  - Solution: persist a sequence of domain events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

Data 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 DB, 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 owns data
    https://microservices.io/patterns/data/domain-event.html
- Called Command Query Responsibility Segregation (CQRS), i.e., separate read and update operations for a data store
  https://microservices.io/patterns/data/cqrs.html

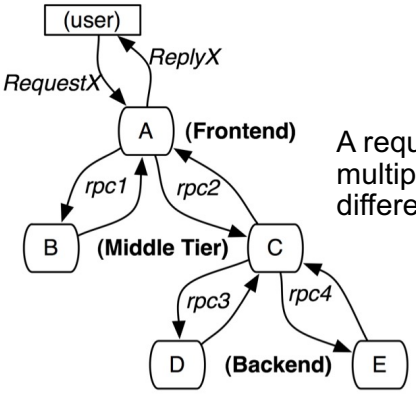
Monitoring microservices

- Service distribution, even at large scale: difficult to monitor microservice app and capture causal and temporal relationships among microservices
  - Aka microservices observability challenge
- We need monitoring
  - To debug the application
  - To analyze performance and latency, including tail latency



Monitoring microservices

- We need monitoring
  - To analyze service dependencies
  - To identify root cause of anomalies, which requires to:
    - Construct a service dependency graph that outlines the sequence of microservices that are invoked
    - Localize the root cause microservices using the graph, traces, logs, and KPIs



A request is passed through multiple microservices with different functionalities

Observability patterns

- Let's consider 2 observability patterns to monitor microservices

1. Log aggregation
2. Distributed request tracing

Observability patterns: Log aggregation

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

✗ Centralized (if physical, not only logical)

✗ Handling large volume of logs requires substantial infrastructure

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

Observability patterns: Distributed tracing

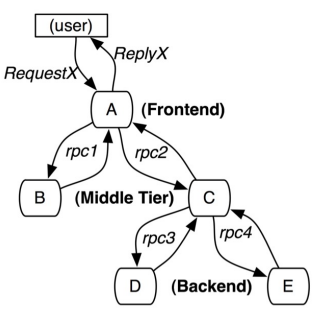
- Problem: How to understand complex app behavior and troubleshoot problems?
- Solution: Instrument microservices with code that
  - Assigns to each user request a unique request id (aka *trace id*), that allows to track that request through the microservices it traverses
  - Passes trace id to each microservice involved in handling the user request
  - Includes trace id in log messages
  - Records *trace context* (e.g., start time, operation, duration) in a (distributed) data store

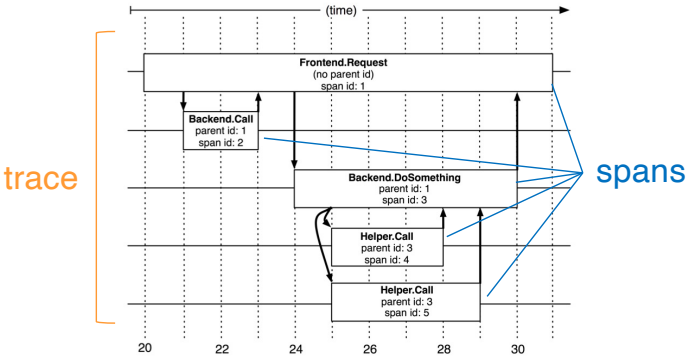
✗ Storing and aggregating traces can require significant infrastructure

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

Monitoring microservices: tools

- Dapper
  - Google's production distributed systems tracing infrastructure
  - Based on *spans* and *traces*
    - Span: individual unit of work (e.g., HTTP request, call to DB) in application; 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 DAG of spans); traces specify how requests are propagated through services and other components





Barroso er al., Dapper, a large-scale distributed systems tracing infrastructure, 2010

Monitoring microservices: tools

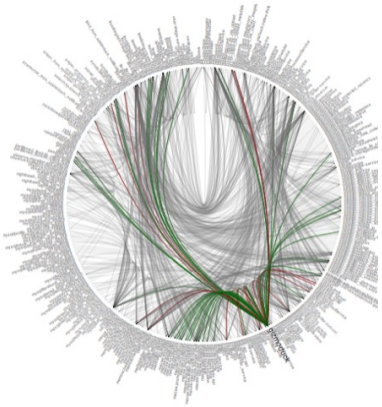
- Dapper
  - Traces are sampled using an adaptive sampling rate, why?
    - Storing everything would require too much storage and network traffic, as well as introducting too much application overhead
  - Span data is written to local log files, then pulled from there by Dapper daemons, sent over a collection infrastructure, and finally traces are stored into BigTable, with one row in a trace table dedicated to each trace id

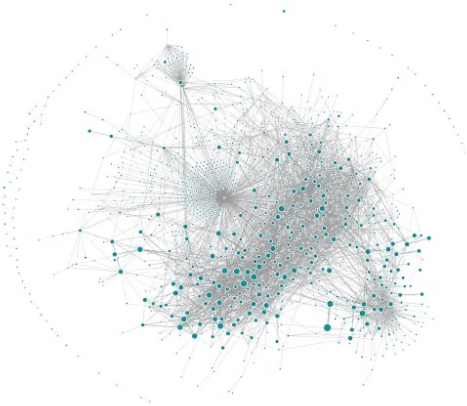
Monitoring microservices: tools

- Open-source tools for distributed tracing
  - Jaeger https://www.jaegertracing.io
    - Uses Spark/Flink for aggregate trace analysis
  - Zipkin https://zipkin.io
  - OpenTelemetry https://opentelemetry.io
    - Broad language support
    - Integrated with popular frameworks and libraries
- Need for standards to support interoperability between different tracing tools
  - W3C defines Trace Contex: standardized format for unifying tracing data https://www.w3.org/TR/trace-context-2/

Some large-scale examples

- Netflix, Twitter, Uber: 500+ microservices





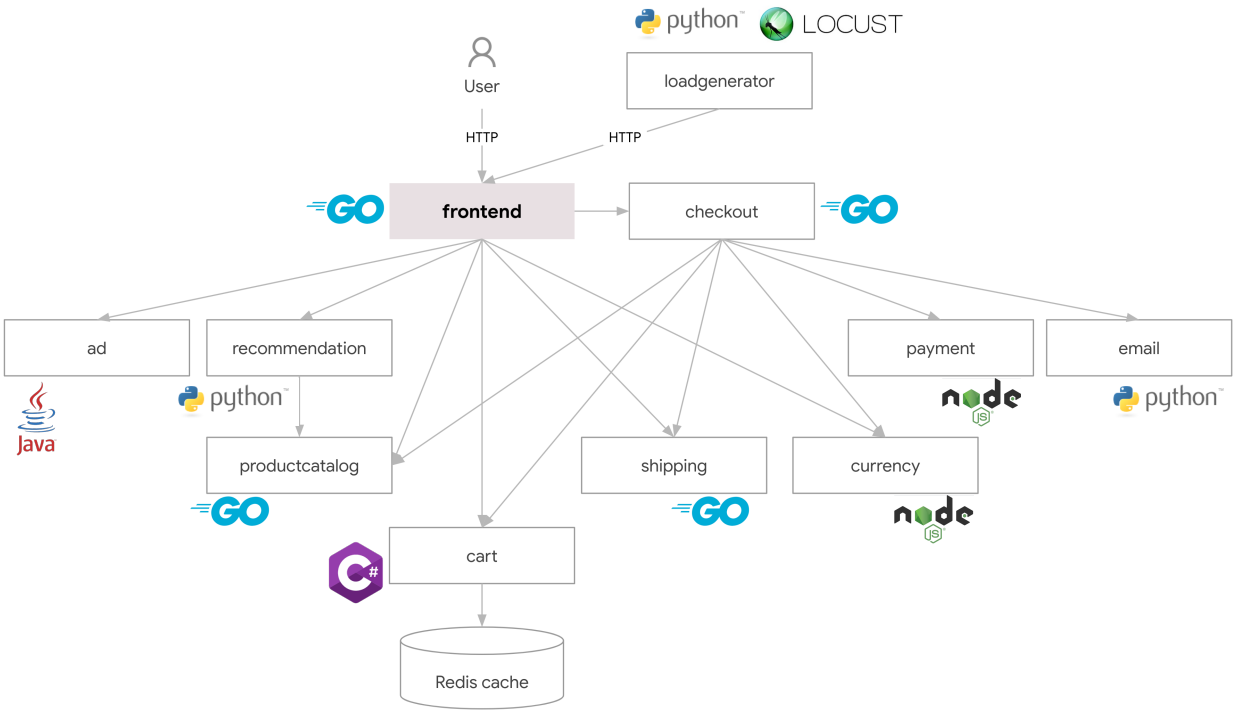
NETFLIX

Uber

Example of microservices app

- Let's examine a microservices app: Google's Online Boutique
  https://github.com/GoogleCloudPlatform/microservices-demo
- Online store composed of 11 microservices written in different languages
  - Renaissance in programming language diversity: need for polyglot programming
- How to realize a polyglot application?
  1. REST and JSON as message interchange format
  2. gRPC and protocol buffers as IDL and message interchange format: approach chosen in Online Boutique

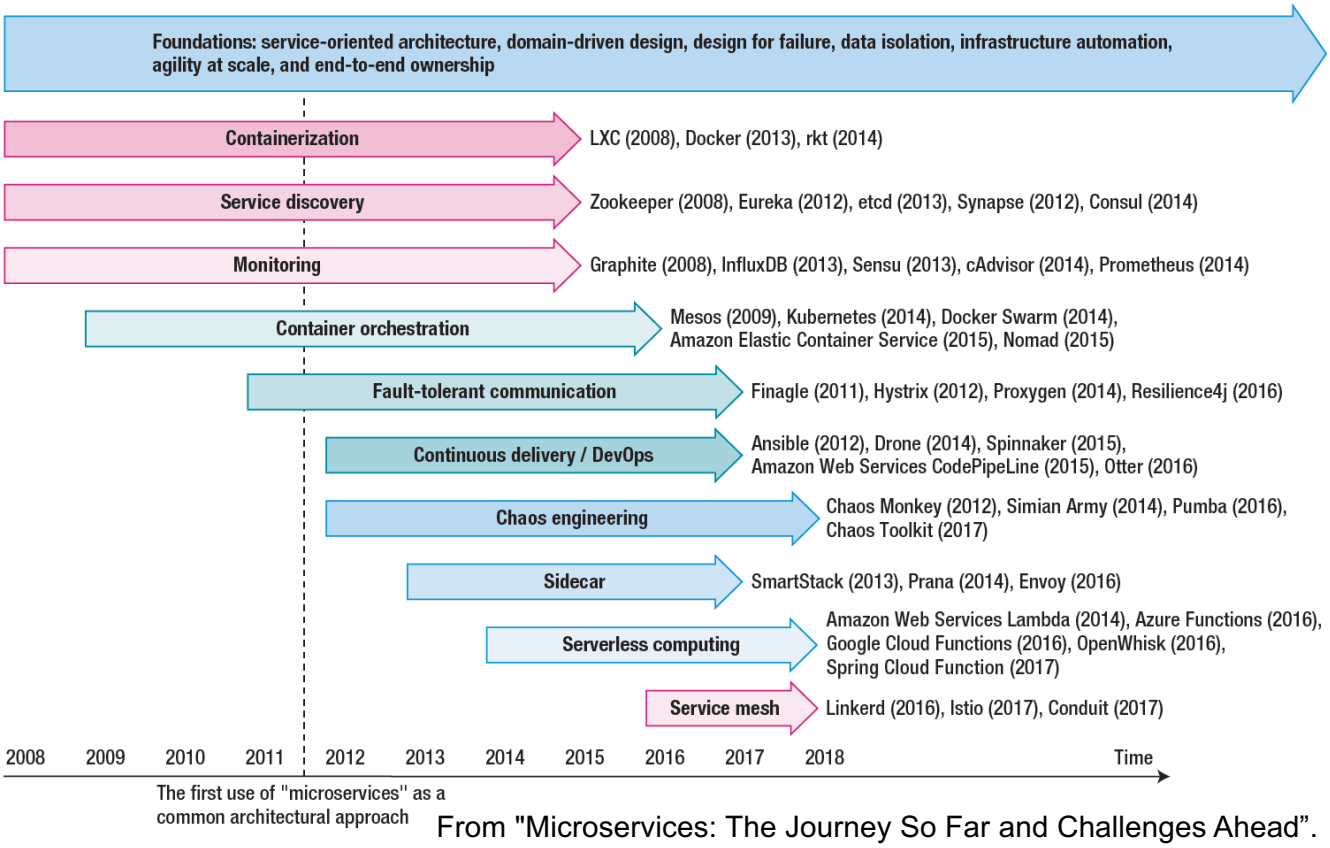
Online Boutique: architecture



Online Boutique: features

- Composed of 10 microservices written in different languages that communicate using gRPC
- Used by Google to demonstrate use of many technologies:
  - Kubernetes and Google Kubernetes Engine (GKE): container orchestration
  - gRPC: we know it ☺
  - Istio / Cloud Service: service mesh
  - Google Cloud's Observability: monitoring, logging, and tracing on Google Cloud https://cloud.google.com/products/observability
  - Locust: load testing tool https://locust.io
  - Skaffold: command line tool that facilitates Kubernetes and containers development https://skaffold.dev

Microservice technologies timeline



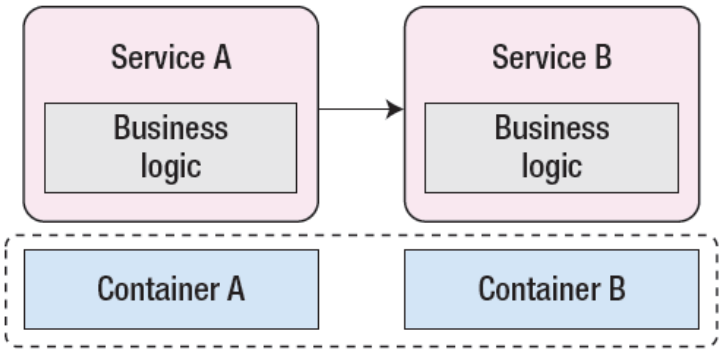
From "Microservices: The Journey So Far and Challenges Ahead".

Generations: at the beginning

- 4 generations of microservice architectures
- 1st generation based on:
  - **Container-based virtualization**, e.g., Docker
  - **Service discovery** tools, e.g.,
    - etcd https://etcd.io: distributed reliable key-value store
    - Zookeeper
  - **Monitoring** tools: enable runtime monitoring and analysis of microservice resources behavior at different levels of detail
    - Graphite https://graphiteapp.org
    - InfluxDB https://www.influxdata.com
    - Prometheus https://prometheus.io/

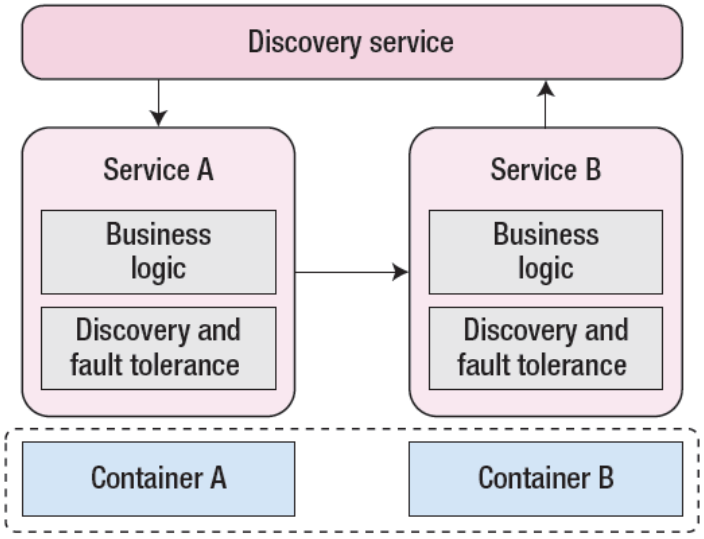
Generations: container orchestration

- Then, **container orchestration**
  - E.g., Kubernetes, Docker Swarm
  - Automate container allocation and management, abstracting away underlying physical or virtual infrastructure from developers
  - But application-level fault-tolerance mechanisms are still implemented inside microservice code



Generations: service discovery and fault tolerance

- 2nd generation based on **service discovery** tools and **fault-tolerant (FT) communication** libraries
  - Goal: let services communicate more efficiently and reliably
  - FT communication libraries implement resiliency patterns (e.g., circuit breaker, fallback, retry/timeout)

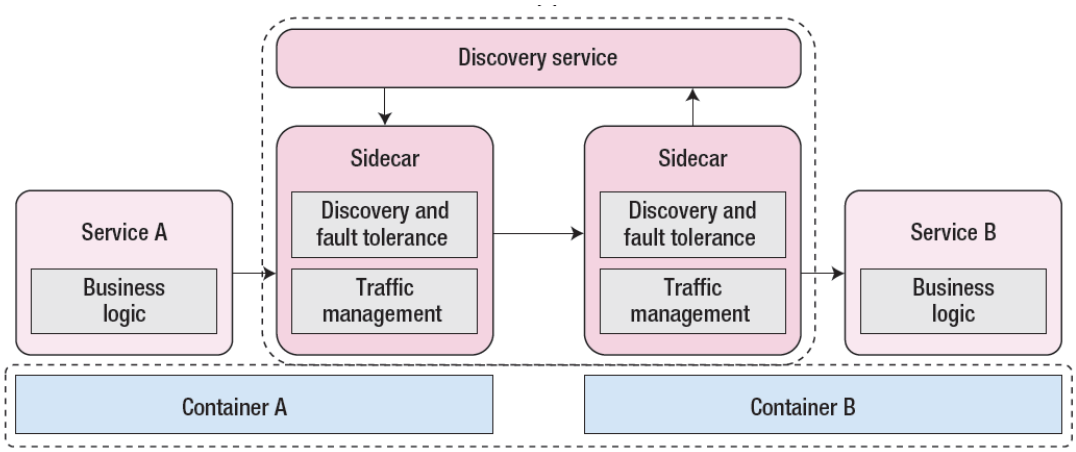


Generations: service discovery and fault tolerance

- Examples:
  - Consul: initially only service discovery, now service mesh
  - Finagle https://github.com/twitter/finagle: FT, protocol-agnostic RPC library
  - Hystrix https://github.com/Netflix/Hystrix: Netflix's FT library (in maintenance mode)
  - Resilience4j https://resilience4j.readme.io: FT library for Java and functional programming, provides circuit breaker and other resiliency patterns

Generations: service mesh

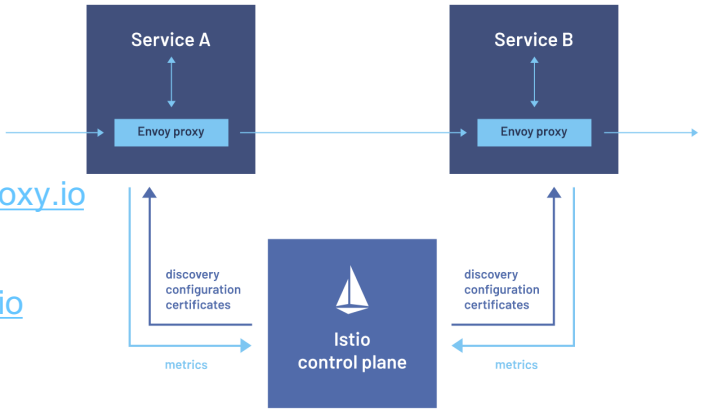
- 3rd generation based on **service mesh** technology and **sidecar proxies**
  - Encapsulate communication-related features and use of protocol-specific and fault-tolerant communication libraries
  - Goal: abstract these functionalities from developers, improve sw reusability and provide homogeneous interface



Service mesh

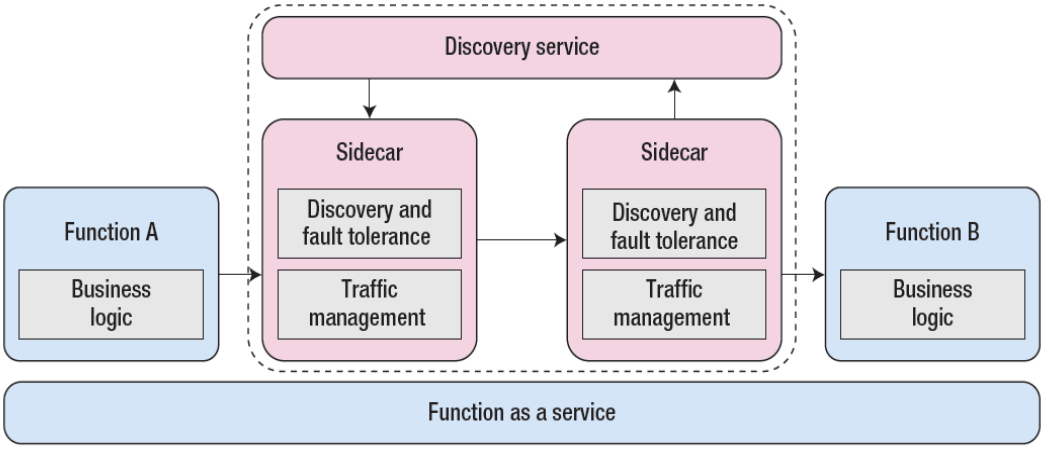
- Dedicated infrastructure layer for microservice apps to facilitate communication among microservices
- Provides a set of features including service discovery, load balancing, authentication, encryption, observability, without adding them to application code
- Typically organized with a centralized control plane and a decentralized data plane (sidecar proxy per microservice)
- Examples:
  - Istio https://istio.io
  - Envoy https://www.envoyproxy.io
  - Linkerd https://linkerd.io
  - Consul https://www.consul.io

See https://servicemesh.es



Generations: serverless

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

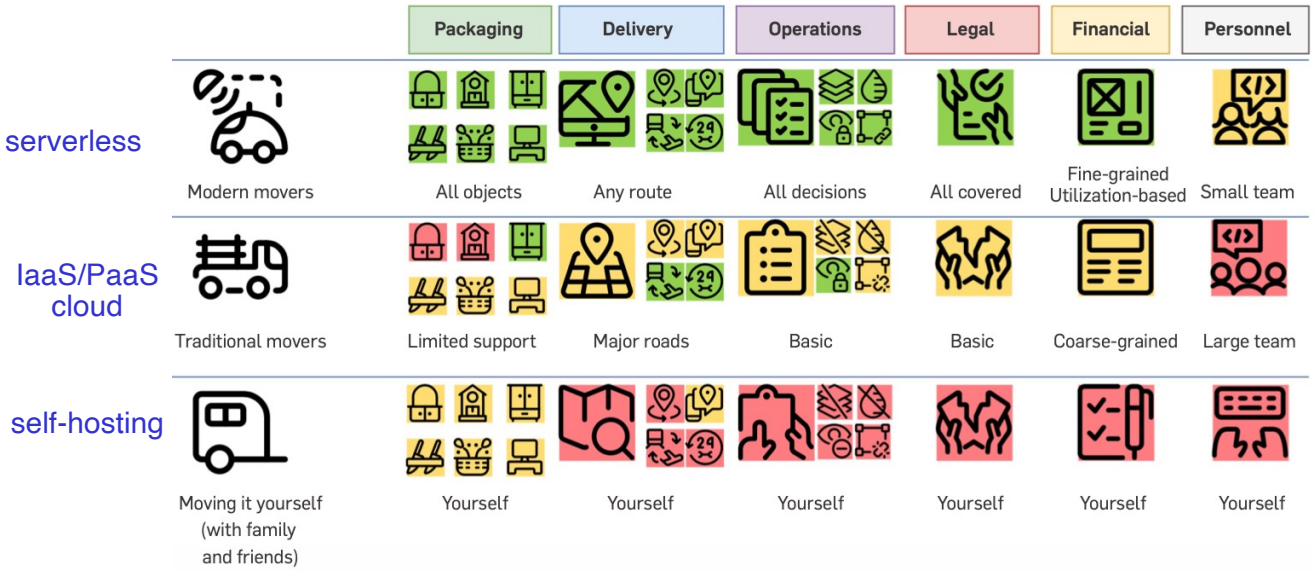


Serverless computing

- Cloud computing model which aims to abstract server management and low-level infrastructure decisions away from users by means of full automation
- Users develop, run and manage application code (i.e., **functions**), without any worry about provisioning, managing and scaling computing resources that run the application code
- Runtime environment is fully managed by Cloud (or private infrastructure and platform) provider
- Serverless: functions still run on “servers” somewhere but we don’t care about them

Serverless through an analogy

- Services for moving homes



Serverless: many definitions

van Eyk et al., Serverless is More: From PaaS to Present Cloud Computing (135 cit.), IEEE IC, May 2018 [47]:

"Serverless Computing is a form of cloud computing which allows users to run event-driven and granularly billed applications, without having to address the operational logic. Function-as-a-Service (FaaS) is a form of serverless computing where the cloud provider manages the resources, lifecycle, and event-driven execution of user-provided functions."

Hellerstein et al., Serverless Computing: One Step Forward, Two Steps Back (360 cit.). CIDR, Jan 2019 [23]:

"Serverless computing offers the attractive notion of a platform in the cloud where developers simply upload their code, and the platform executes it on their behalf as needed at any scale. Developers need not concern themselves with provisioning or operating servers, and they pay only for the compute resources used when their code is invoked... Serverless is not only FaaS. It is FaaS supported by a "standard library": the various multi-tenanted, autoscaling services provided by the vendor. In the case of AWS, this includes S3 (large object storage), DynamoDB (key-value storage), SQS (queuing services), SNS (notification services), and more."

Castro et al., The Rise of Serverless Computing (196 cit.), CACM, Dec 2019 [10]:

"Serverless computing is a platform that hides server usage from developers and runs code on-demand automatically scaled and billed only for the time the code is running. This definition captures the two key features of serverless computing: (a) Cost—billed only for what is running (pay-as-you-go)...; serverless essentially supports "scaling to zero" and avoids the need to pay for idle servers. (b) Elasticity—scaling from zero to "infinity." … The main differentiators of serverless platforms is transparent autoscaling and fine-grained resource charging only when code is running. Function-as-a-Service is a serverless computing platform where the unit of computation is a function that is executed in response to triggers such as events or HTTP requests. Mobile Backend as-a-Service (MBaaS) or more generalized Backend as-a-Service (BaaS) bears a close resemblance to serverless computing."

Jonas,…, Patterson et al., Cloud Programming Simplified: A Berkeley View on Serverless Computing (485 cit.), arXiv, Feb 2019 [26], refined in a follow up publication by the same authors What Serverless Computing Is and Should Become: The Next Phase of Cloud Computing (75 cit.), CACM, May 2021 [41]

"In serverless computing, programmers create applications using high level abstractions offered by the cloud provider... They may also use serverless object storage, message queues, key-value store databases, mobile client data sync, and so on, a group of services offerings known collectively as Backend-as-a-Service (BaaS). Managed cloud function services are also called Function-as-a-Service (FaaS) and collectively Serverless Cloud Computing today = FaaS + BaaS. Three essential qualities of serverless computing are: 1. Providing an abstraction that hides the servers and the complexity of programming and operating them. 2. Offering a pay-as-you-go cost model instead of a reservation-based model, so there is no charge for idle resources. 3. Automatic, rapid, and unlimited scaling resources up and down to match demand closely, from zero to practically infinite."

Legend: 6x NoOps 6x Function-as-a-Service 5x pay-per-use 4x autoscaling/elasticity 3x Backend-as-a-Service 2x event-driven arch.

Serverless: many definitions

Kounev et al., Serverless Computing: What It Is, and What It Is Not?, Comm. ACM, 2023

Serverless computing is a cloud computing paradigm encompassing a class of cloud computing platforms that allow one to develop, deploy, and run applications (or components thereof) in the cloud without allocating and managing virtualized servers and resources or being concerned about other operational aspects.

The responsibility for operational aspects, such as fault tolerance or the elastic scaling of computing, storage, and communication resources to match varying application demands, is offloaded to the cloud provider.

Providers apply utilization-based billing: they charge cloud users with fine granularity, in proportion to the resources that applications actually consume from the cloud infrastructure, such as computing time, memory, and storage space.

Serverless, FaaS and BaaS

- Function as a Service (**FaaS**) and serverless sometimes used interchangeably, some discussion on difference
- FaaS can be seen as the most prominent example of serverless computing
  - Can be defined as "a serverless computing platform where the unit of computation is a function that is executed in response to triggers such as events or HTTP requests" (Kounev et al.)
- Backend as a Service (**BaaS**)
  - BaaS offerings are focused on specialized cloud application components, such as object storage, databases, and messaging
  - Examples: AWS S3 and DynamoDB, Google Cloud Firestore (NoSQL document database) and Pub/Sub

Serverless: features

- Ephemeral compute resources
  - May only last for one function invocation
  - ✗ Cold start: when a request arrives and no container/microVM is ready to serve it, function execution must be delayed until a new container/microVM 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: fine-grained and utilization-based
  - E.g., AWS Lambda price is per 1 ms associated with memory size

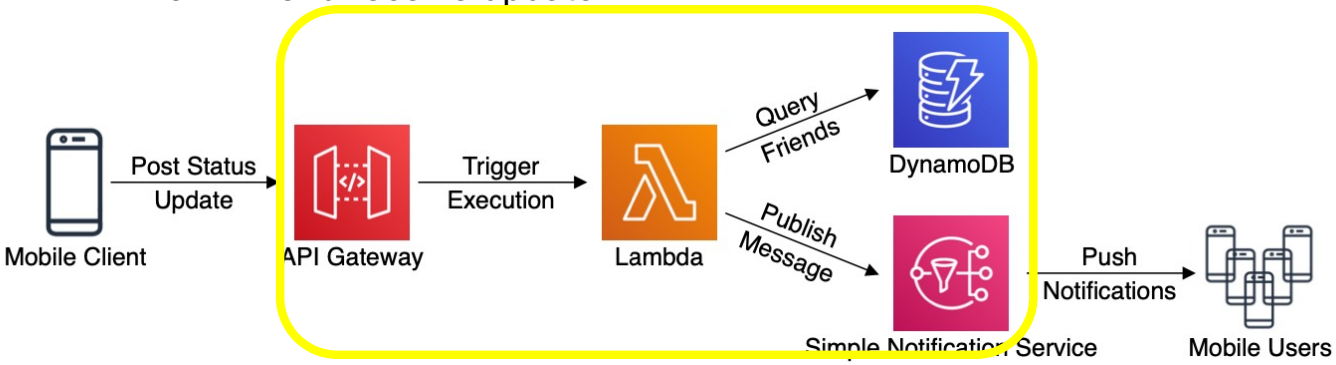
Architecture	Duration	Requests
x86 Price		
First 6 Billion GB-seconds / month	$0.0000166667 for every GB-second	$0.20 per 1M requests
Next 9 Billion GB-seconds / month	$0.000015 for every GB-second	$0.20 per 1M requests
Over 15 Billion GB-seconds / month	$0.0000133334 for every GB-second	$0.20 per 1M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

Serverless: features

- Event-driven
  - When event is triggered (e.g., file uploaded to storage, message ready in queue) or HTTP request arrives, serving infrastructure is allocated dynamically to execute the function code
- NoOps (no operations): simplifies process of deploying code into production
  - Scaling, capacity planning and maintenance operations are hidden from developers
- Supports diverse kinds of applications
  - From enterprise automation to scientific computing to ML inference

Serverless application: a first example

- Propagate updates in social media app in a serverless fashion
  1. User composes status update and sends it using mobile client
  2. Platform orchestrates ops needed to propagate update inside social media platform and to user's friends using serverless (AWS Lambda) and other cloud services (AWS DynamoDB and SNS)
  3. Friend receive update

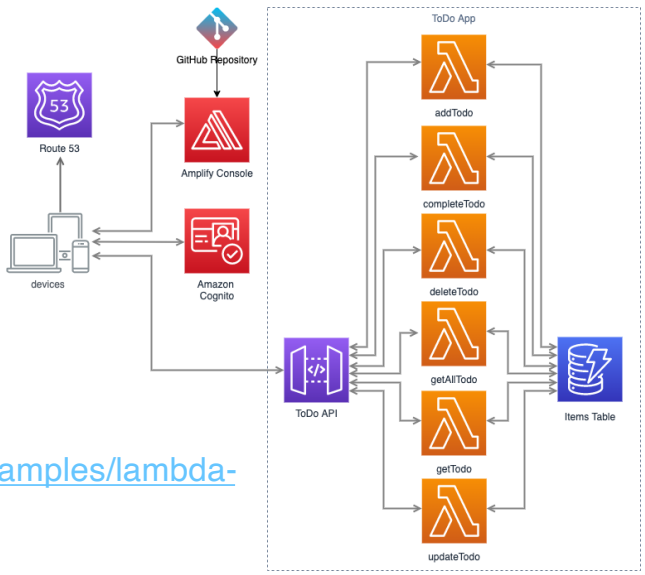


Serverless Cloud services

- Several Cloud providers offer serverless computing on their public clouds as fully managed service
  - AWS Lambda https://aws.amazon.com/lambda/
    - See hands-on course
    - Lambda@Edge: functions at the edge https://aws.amazon.com/lambda/edge/
  - Azure Functions https://azure.microsoft.com/products/functions
  - Google Cloud Run Functions https://cloud.google.com/functions
- User has limited knobs to control performance
  - Amount of memory allocated to function (CPU ~ memory)
- Cloud platforms also offer supporting services to operate a serverless ecosystem
  - E.g., event notification, storage, message queue, DB

Example: AWS's reference Web app

- Simple "to-do list" web app that enables a registered user to create, update, view, and delete items
- Event-driven web app uses AWS Lambda and Amazon API Gateway for its business logic, DynamoDB as its database, and Amplify Console to host all static content



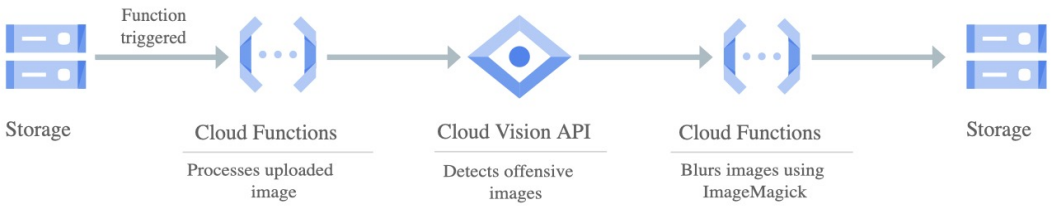
https://github.com/aws-samples/lambda-refarch-webapp

Example: Google Cloud Run Functions

- "Hello World" example from Google using Go
  - HTTP response that displays "Hello, World!"
    https://cloud.google.com/functions/docs/tutorials

```go
// Package helloworld provides a set of Cloud Functions samples.
package helloworld

import (
        "fmt"
        "net/http"

        "github.com/GoogleCloudPlatform/functions-framework-go/functions"
)

func init() {
        functions.HTTP("HelloGet", helloGet)
}

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


Example: Google Cloud Run Functions



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

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

Serverless: state

- Stateless functions are easy to manage (horizontal scalability, fast recovery, …)
  - But stateless functions are not enough for some applications (e.g., ML, streaming)
- How to handle stateful computation?
  - First approach: state is external (e.g., handed over to external shared storage system) so functions are still stateless
  - Issues to address:
    - Efficient access to shared state, so to keep auto-scaling benefits
    - Programming support, e.g., Azure Durable Functions https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview
- What about transactions? https://cacm.acm.org/practice/transactions-and-serverless-are-made-for-each-other/

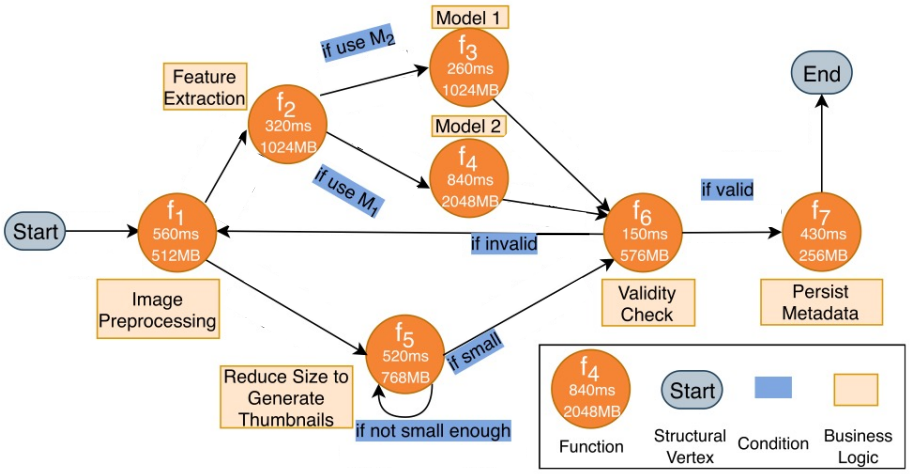
Serverless: challenges and limitations

- Performance
  - Cold starts
    - "Starting a new function instance involves loading the runtime and your code. Requests that include function instance startup, called *cold starts*, can be slower than requests routed to existing function instances." (Google Cloud Run Functions)
  - Autoscaling
- Programming languages
  - Language support depends on Cloud provider
  - Language impacts on performance and cost of functions
- Resource limits
  - E.g., on AWS memory between 128 and 10240 MB per function
- Security
  - E.g., more entry points, financial exhaustion attacks
- Vendor lock-in

Less flexibility

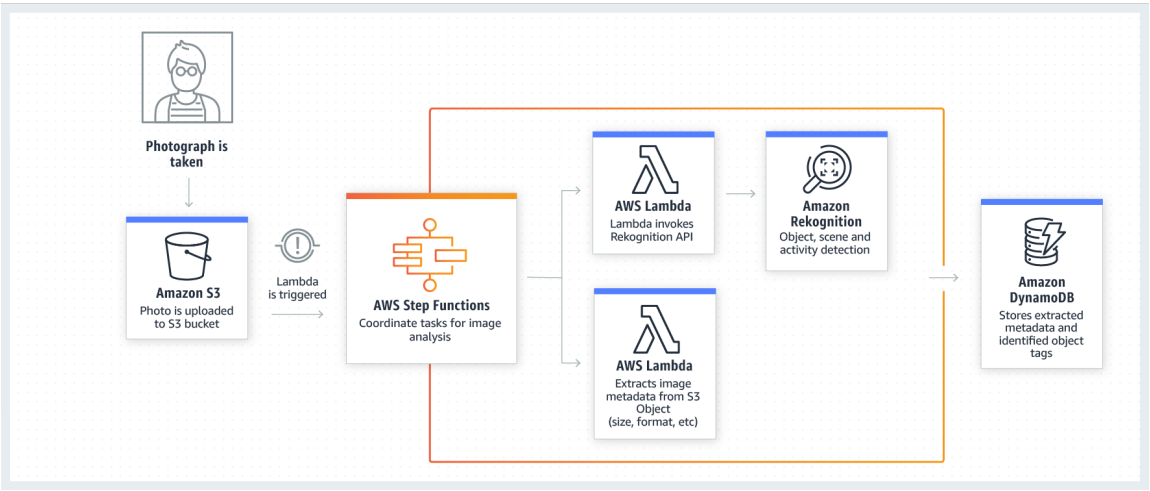
Composition of serverless functions

- 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 a workflow
- Example: process photo after its upload in S3

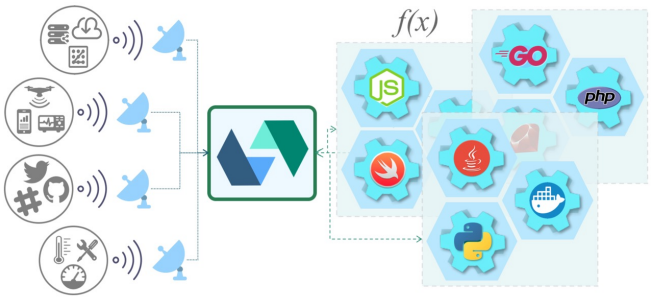


Open-source serverless platforms

- Can run on commodity hardware
- Popular platforms
  - Apache OpenWhisk https://openwhisk.apache.org
  - OpenFaaS https://www.openfaas.com
  - Fission https://fission.io
  - Knative https://knative.dev
  - Nuclio https://nuclio.io
- Most platforms rely on Kubernetes for orchestration and management of serverless functions
  - Configuration and management of containers inside which functions run
  - Container scheduling and service discovery
  - Elasticity management

OpenWhisk

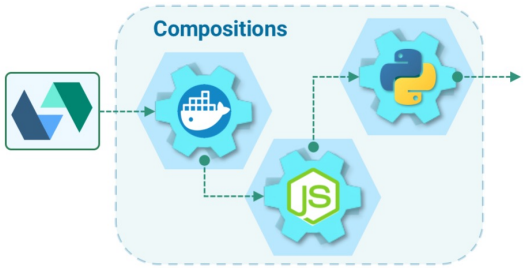
- Distributed serverless platform that executes functions in response to events
  https://openwhisk.apache.org



- Functions run inside Docker containers
- Support for multiple container orchestration frameworks

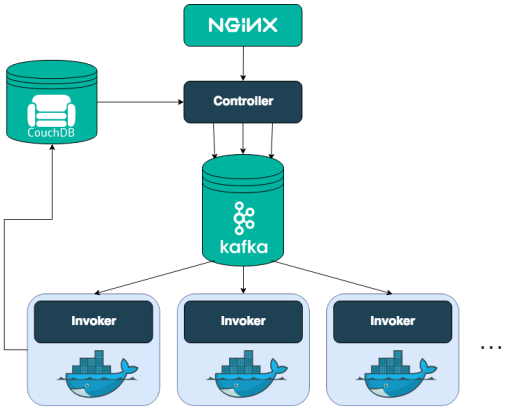
OpenWhisk

- Developers write functions (called actions)
  - In any supported programming language
  - Actions are dynamically deployed, scheduled and run in response to associated events (via triggers) from external sources (feeds) or from HTTP requests
- Functions can be combined into compositions



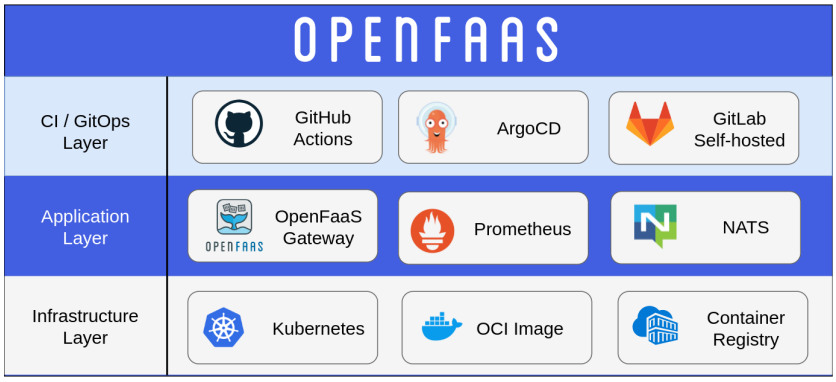
OpenWhisk: architecture

- Architecture powered by multiple frameworks
  - NGINX: OpenWhisk entry point that receives HTTP request and forwards it to Controller, that translate the request into invocation of an action
  - CouchDB (document-oriented NoSQL data store): stores authentication and authorization info, action code, ...
  - Kafka: mediates communication between controller and invokers
  - Docker: used by Invoker to execute action code



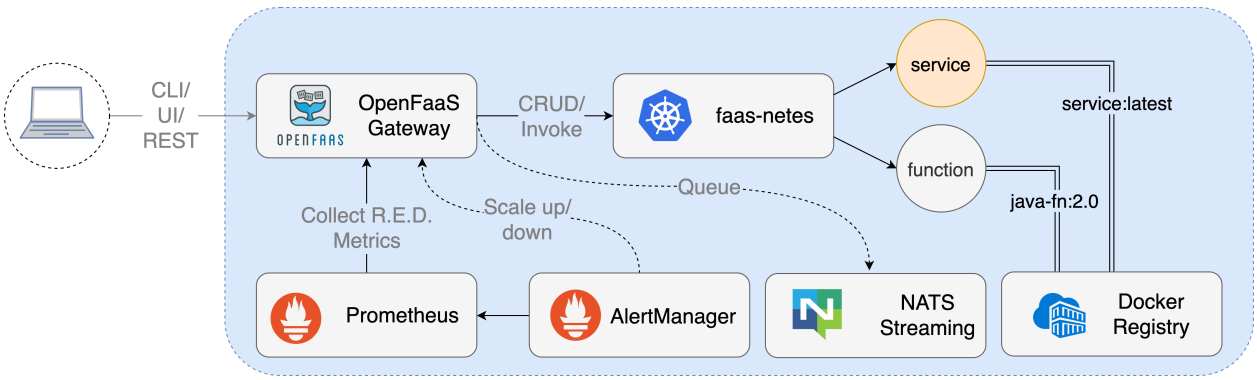
OpenFaaS

- Distributed serverless framework for functions, built on top of Docker and Kubernetes https://www.openfaas.com
- Layered architecture
  - OpenFaaS gateway: provides REST API to manage and scale functions, record metrics
  - NATS: used for asynchronous function execution and queuing https://nats.io
  - Prometheus: provides metrics and enables auto-scaling https://prometheus.io



OpenFaaS

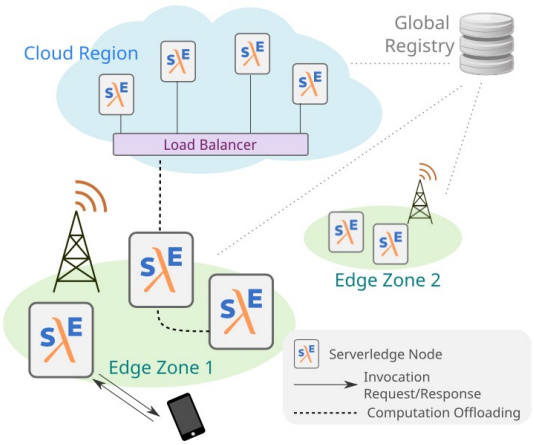
- Conceptual workflow
  - Gateway can be accessed through its REST API, CLI or UI
  - Prometheus collects metrics which are made available via gateway's API and are used for auto-scaling
  - NATS enables function invocations to run asynchronously



https://docs.openfaas.com/architecture/stack/

Serverless in the compute continuum

- Open-source serverless platforms are unsuitable for compute continuum because of centralization components
- We are developing Serverledge, a decentralized FaaS framework: thesis opportunities!



Russo Russo et al., Decentralized Function-as-a-Service for the Edge-Cloud Continuum, Percom 2023 https://github.com/grussorusso/serverledge

References: Microservices

- Lewis and Fowler, Microservice, https://martinfowler.com/articles/microservices.html
- Lewis and Fowler, Microservice Guides, https://martinfowler.com/microservices
- Richardson, Microservice Architecture, https://microservices.io
- Jamshidi et al., Microservices: The Journey So Far and Challenges Ahead, *IEEE Software*, 2018 https://ieeexplore.ieee.org/iel7/52/8354413/08354433.pdf

References: Serverless

- Roberts, Serverless Architectures, https://martinfowler.com/articles/serverless.html
- Schleier-Smith et al., What serverless computing is and should become: the next phase of cloud computing, *Comm. ACM*, 2021 https://cacm.acm.org/research/what-serverless-computing-is-and-should-become
- Kounev et al., Serverless Computing: What It Is, and What It Is Not?, *Comm. ACM*, 2023 https://dl.acm.org/doi/pdf/10.1145/3587249