Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Microservices and Serveless Computing

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2025/26

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

Microservices

- Architectural style for distributed applications
- Structures an application as a collection of loosely coupled services
- Not entirely new: derives from **SOA** and **Web services**
  - But with some significant differences
- Focuses on how to build, manage, and evolve architectures composed of small, self-contained units
- Key characteristics
  - Modularization: application decomposed into a set of independently deployable services, that are loosely coupled and cooperating
  - Rapid deployment and scalability
  - Data management: services typically own their memory persistence layer (e.g., relational DBs and NoSQL data stores)

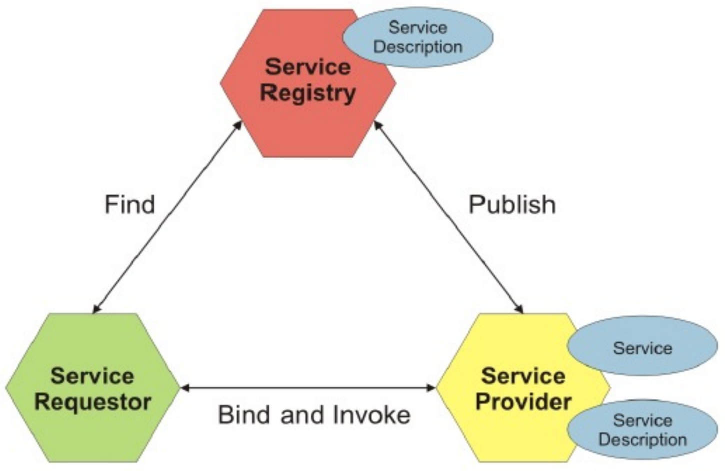
Service Oriented Architecture (SOA)

- 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 of services
  - Message-oriented and description-oriented
  - Service granularity and network orientation
  - Platform neutral

SOA: Core entities

- 3 interacting entities
  1. *Service requestor* (or *consumer*): requests service execution
  2. *Service provider*: implements and exposes the service
  3. *Service registry*: publishes the service and enables discovery

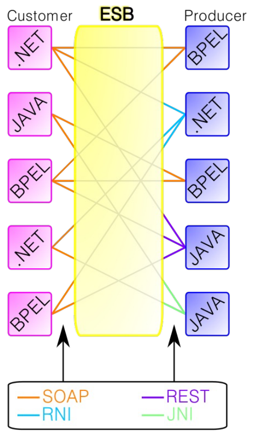


Web services

- 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

- Large ecosystem of standards and specification (more than 60), among which:
  - Service description : **WSDL** (Web Service Description Language)
  - Communication: **SOAP** (Simple Object Access Protocol)
  - Service registry: **UDDI** (Universal Description, Discovery and Integration)
  - Business process definition: **BPEL** (Business Process Execution Language), **BPMN** (Business Process Model and Notation)
  - SLA: **WSLA**
- Wide variety of technologies
  - Including **ESB** (Enterprise Service Bus): integration platform for routing, mediation, transformation, and communication; commonly used in enterprise SOA systems



SOA vs. microservices

- Heavyweight vs. lightweight
  - SOA: heavyweight middleware (e.g., ESB)
  - Microservices: lightweight technologies
- Protocols
  - SOA: web services protocols
  - Microservices: RESTful APIs and HTTP, often JSON-based
- Architectural view
  - SOA: integration solution
  - Microservices: to build complete applications
- Data ownership
  - SOA: often shared databases or centralized data models
  - Microservices: each service owns its own data store
- Deployment and lifecycle
  - SOA: services often deployed together or coordinated
  - Microservices: independent deployment and versioning

Microservices and containers

- Microservices are an ideal complement to container-based virtualization
  - "One microservice instance per container"
    - Each microservice packaged as a container image
    - Each instance deployed as a container
  - Containers enable runtime management (scaling, migration)
- Pros and cons:
  - ✓ Scale out/in by changing the number of container replicas
  - ✓ Scale up/down by adjusting container resources
  - ✓ Isolation of microservice instance
  - ✓ Resource limits per service
  - ✓ Fast build and startup
  - ✗ Require container orchestration to manage multi-container applications

Microservices: benefits

- Increased software agility
  - Each microservice is an independent unit of development, deployment, operation, versioning, and scaling
  - Encapsulation via APIs: interaction occurs through well-defined APIs that hide implementation details
  - Container-based virtualization
- Improved scalability and fault isolation
- Increased reusability across business domains
- Improved data security
- Faster development and delivery
- Greater team autonomy

Microservices: concerns

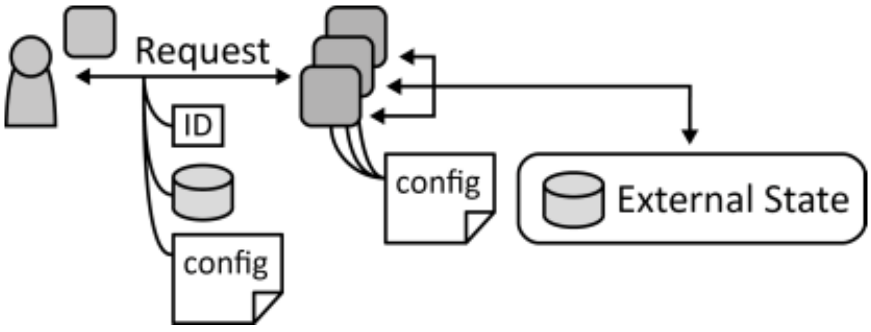
- Increased network traffic
  - Remote service calls introduce network latency
- Higher system complexity
- Increased operational complexity
  - Deployment, monitoring, and management
- More difficult testing and debugging
  - End-to-end and distributed testing are harder

Microservices and scalability

- How to achieve scalability of microservices?
  - Run multiple instances of the same microservice
  - Load balance requests across instances
- Prefer **stateless services**
  - No state stored in the service instance
  - Scale faster and more easily than stateful services

Stateless service

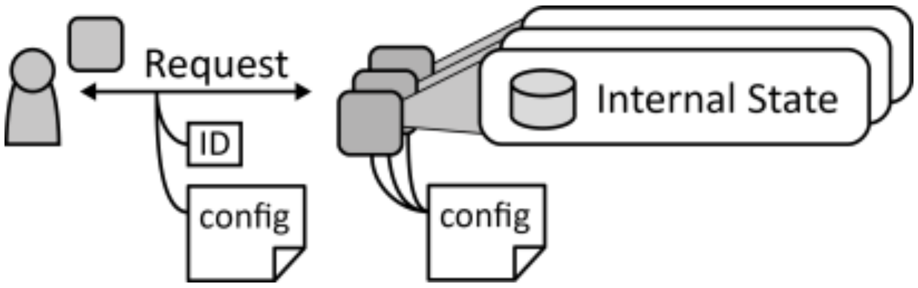
- Does not store client or session state internally
- State handled externally (e.g., database, cache)
- Easier to scale out
- More tolerant to service failures



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

Stateful service

- Stores state internally
- Multiple instances of a scaled-out service must synchronize state
- Required to provide consistent, unified behavior
- State management adds complexity and latency
- How can multiple service instances maintain a synchronized internal state while scaling?

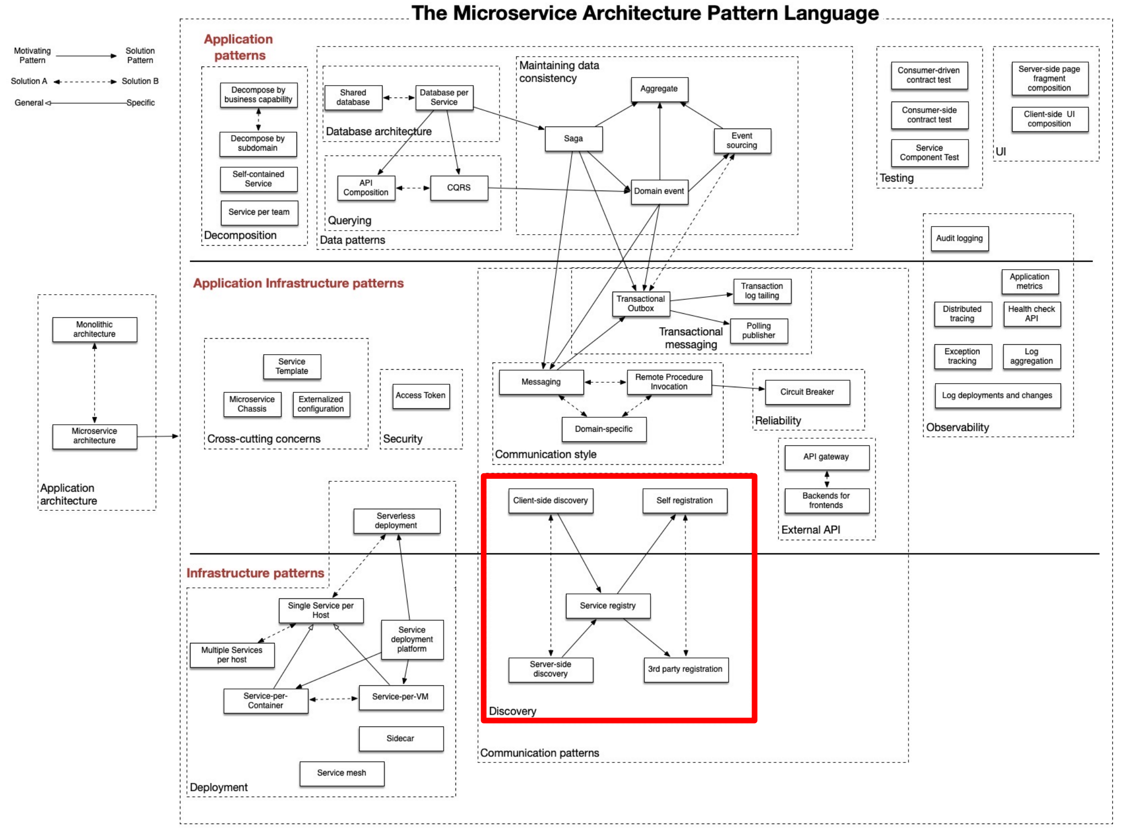


https://www.cloudcomputingpatterns.org/stateful_component/

Stateful service: scaling approaches

- Centralized storage
  – All instances read/write state to a shared database or cache
- State replication
  – Replicate state across instances
- Event sourcing / CQRS patterns
  – Store changes as events; rebuild state per instance
- State partitioning (or sharding)
  – Divide state into independent partitions
  – Each instance handles a subset of the state
- Sticky sessions (less used)
  – Route requests from the same client to the same instance
- Tradeoffs
  – Adds operational complexity
  – Can limit elastic scalability compared to stateless services

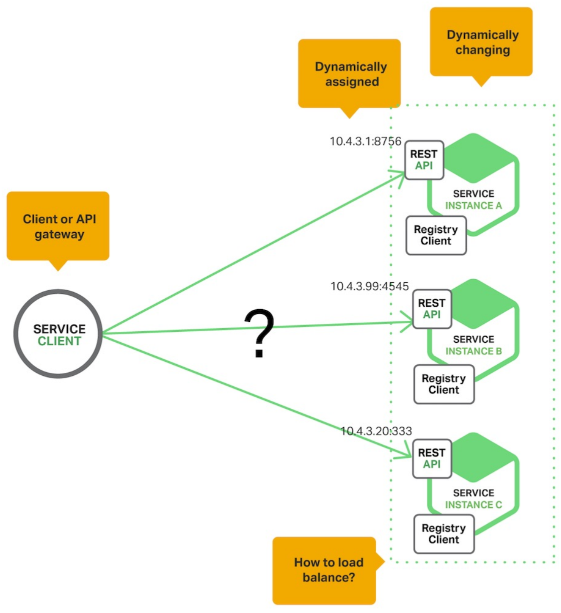
Microservice patterns



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

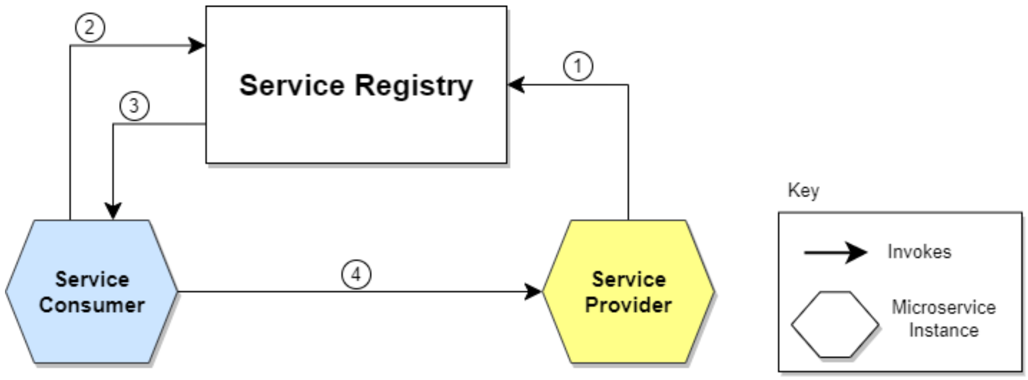
Service discovery

- Microservice clients need to locate service instances
  - Instances have dynamic network locations (IP address, port)
  - The set of instances changes due to auto-scaling, failures, upgrades
- Service discovery provides
  - Registration: microservice instances register themselves when they start
  - Lookup / resolution: clients can find service instances dynamically



Service discovery: patterns

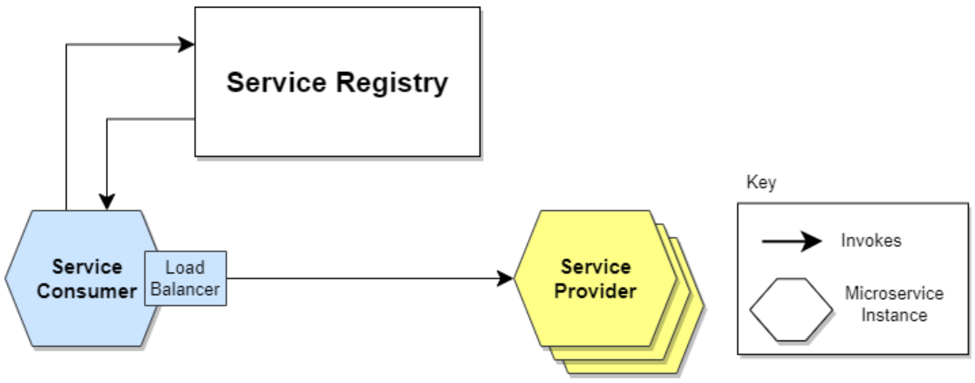
1. Service registry
   – A database of services, instances and network locations
   – Instances register at startup and deregistered at shutdown
   – Clients query the registry to find available instances



Service discovery: patterns

2. Client-side service discovery
   – Client determines service network location and load balances requests among them
   – Client queries Service Registry, uses a load-balancing algorithm to pick an instance, and sends request directly to chosen instance

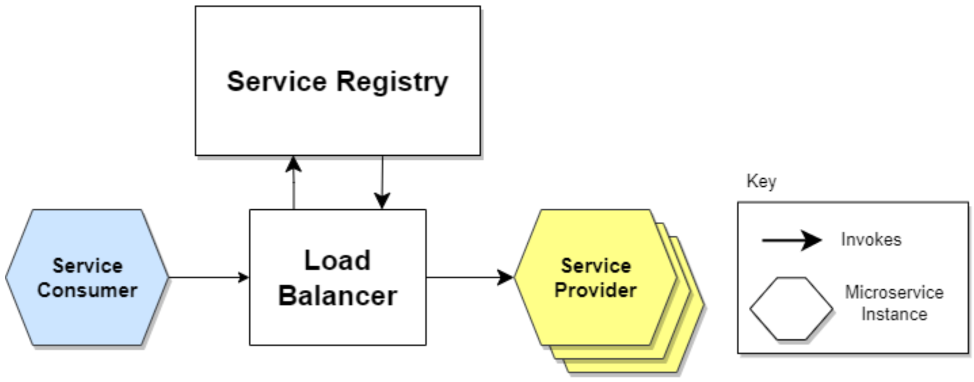
   https://microservices.io/patterns/client-side-discovery.html



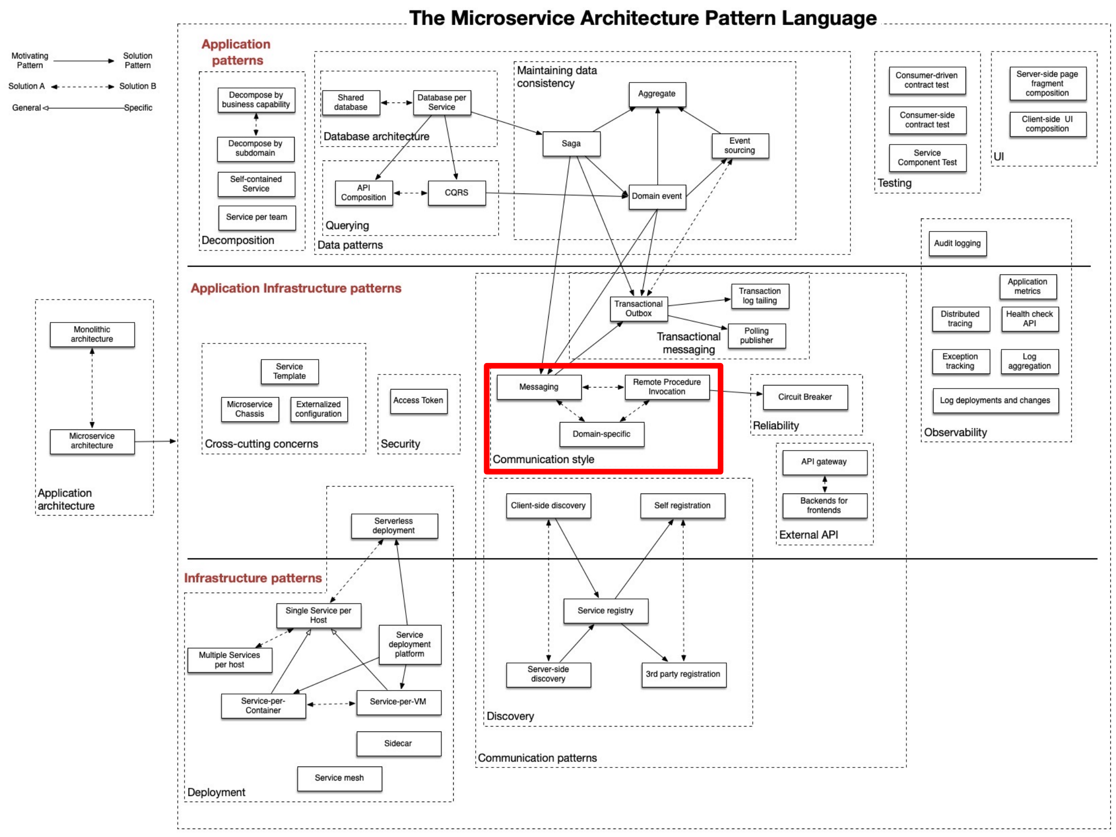
Service discovery: patterns

3. Server-side service discovery
   - Client sends request to load balancer at a known location
   - Load balancer queries Service Registry and routes request to an available instance

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



Microservice patterns



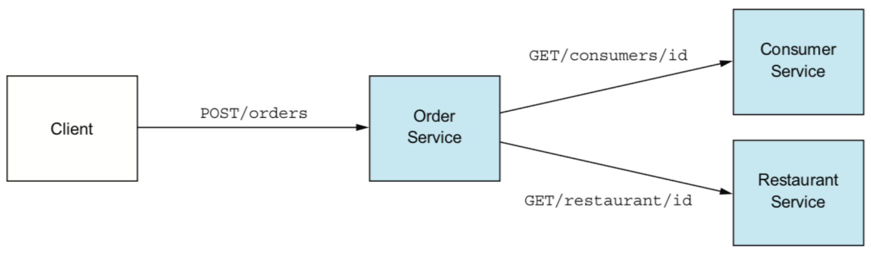
Learn-Build-Assess Microservices http://adopt.microservices.io

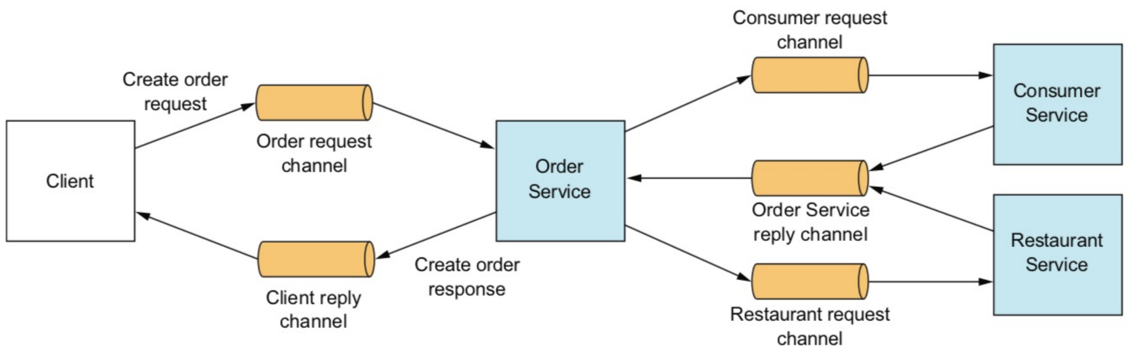
Communication styles

- Synchronous communication
  - Request/response style
  - Mechanisms: HTTP/REST, RPC
  - Typically, one-to-one interaction
  - ✗ Can reduce availability if service is slow or down
- Asynchronous communication
  - Event-driven or message-based
  - Mechanisms: pub/sub systems, message queues, related protocols
  - Supports one-to-one or one-to-many interaction
  - ✓ Improves resilience and decoupling

Communication styles

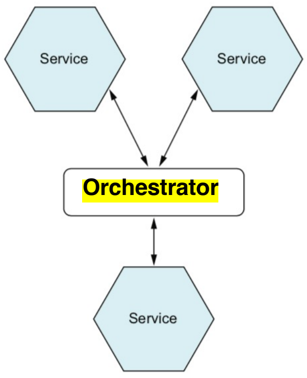
- Example of synchronous vs. asynchronous communication





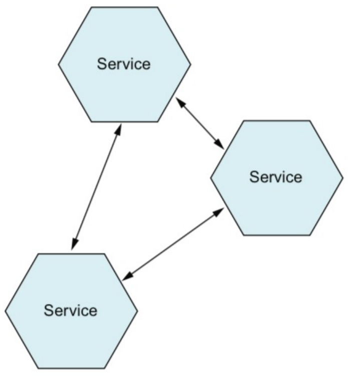
Service interaction

- Microservices can interact according to 2 patterns:
  - Orchestration
  - Choreography
- **Orchestration**: centralized approach
  - A single centralized process (*orchestrator*, *conductor* or *message broker*) coordinates interactions
  - Orchestrator invokes and combines services, which may be unaware of the composition



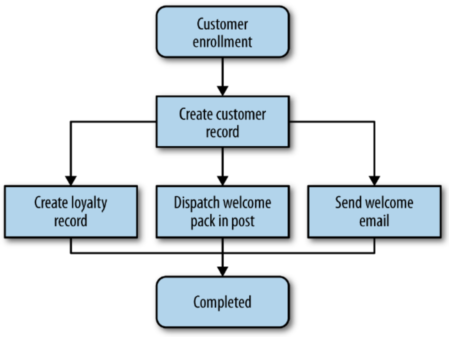
Orchestration and choreography

- **Choreography**: decentralized approach
  - Interaction defined by exchange of messages, rules, and agreements between services
  - Services react to events/messages directly, without a central coordinator

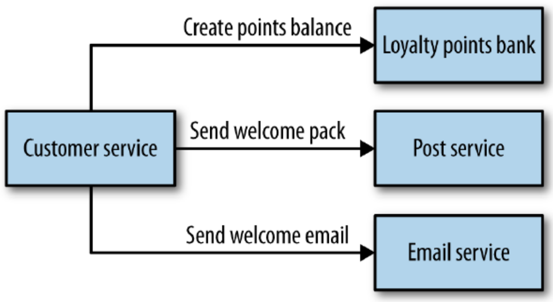


Orchestration and choreography: example

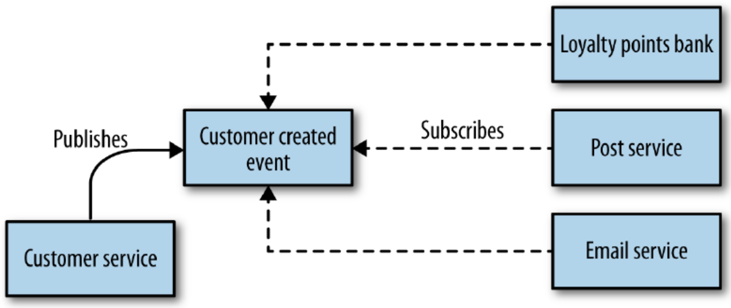
- Workflow for customer creation



Orchestration



Choreography



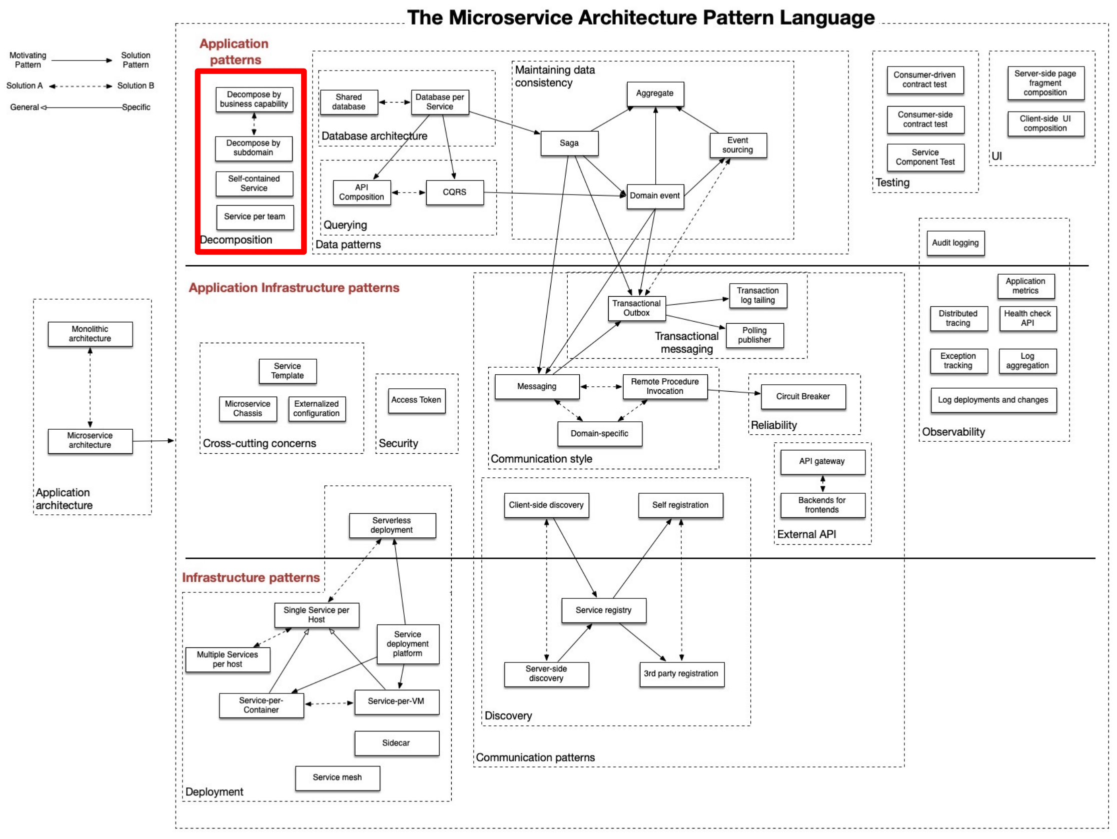
Source: S. Newman, “Building Microservices”, O’Really, 2015

Orchestration vs choreography

- Orchestration:
  - ✓ Simpler and more popular
  - ✗ SPoF and potential performance bottleneck
  - ✗ Tight coupling between orchestrator and services
  - ✗ Higher network traffic and latency

- Choreography
  - ✓ Lower coupling and operational complexity
  - ✓ Increased flexibility and easier to change individual services
  - ✗ Services must be aware of each other’s locations
  - ✗ Harder to observe and debug
  - ✗ Implementing guarantees like reliable delivery is more challenging

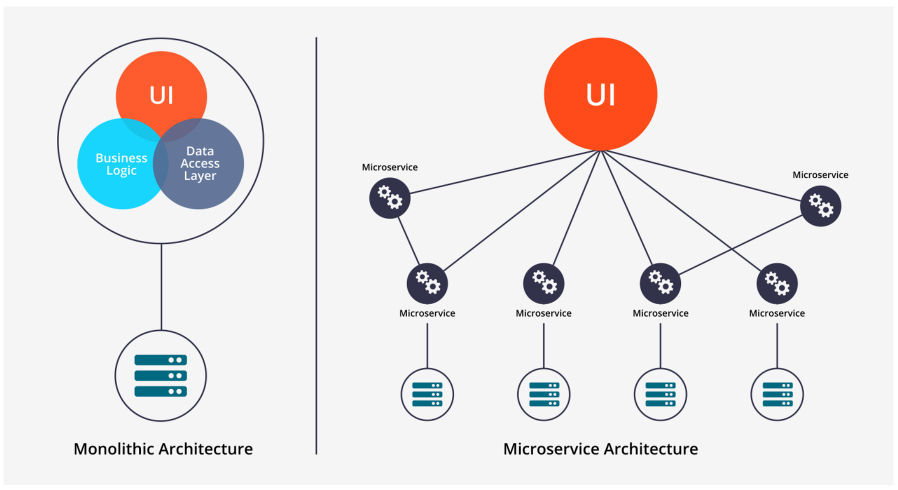
Microservice patterns



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

Decomposition patterns

- Monolithic application: built and deployed as a single unit
- Decomposing into microservices is mostly an art
  - No single "best" strategy
  - Multiple decomposition patterns exist https://microservices.io/patterns



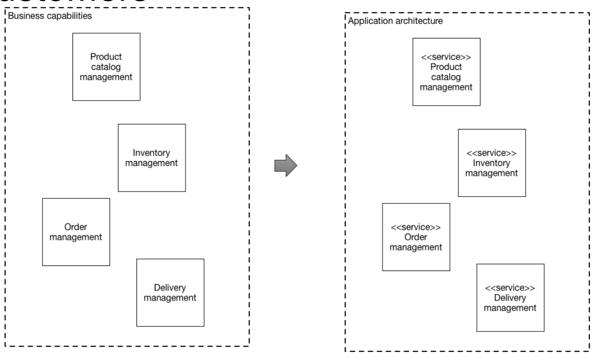
Decomposition patterns

- Design considerations
  - Stable architecture
    - Decomposition boundaries should not change frequently
  - High cohesion
    - Each service implements a small set of strongly related functions
  - Common Closure Principle (CCP)
    - Things that change together should be packaged together
    - A change should ideally affect only one service
  - Loose coupling
    - Each service exposes an API that encapsulates its implementation
    - Internal changes should not affect clients
  - Testability
    - Each service should be independently testable
  - Team size and ownership
    - Service small enough for a "two-pizza team" (≈ 6–10 people)
    - Teams should be autonomous, with minimal coordination

Main decomposition patterns

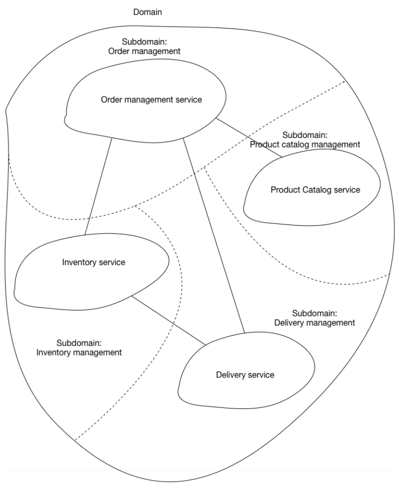
- Example: e-commerce app that takes orders from customers, verifies inventory and available credit, and ships them

1. Decompose services based on business capability
   - Business capability: a function the business performs to generate value
   - Organizational/business view
   - Good for initial decomposition and team alignment
   - E.g., *Order Management* is responsible for orders, *Customer Management* for customers

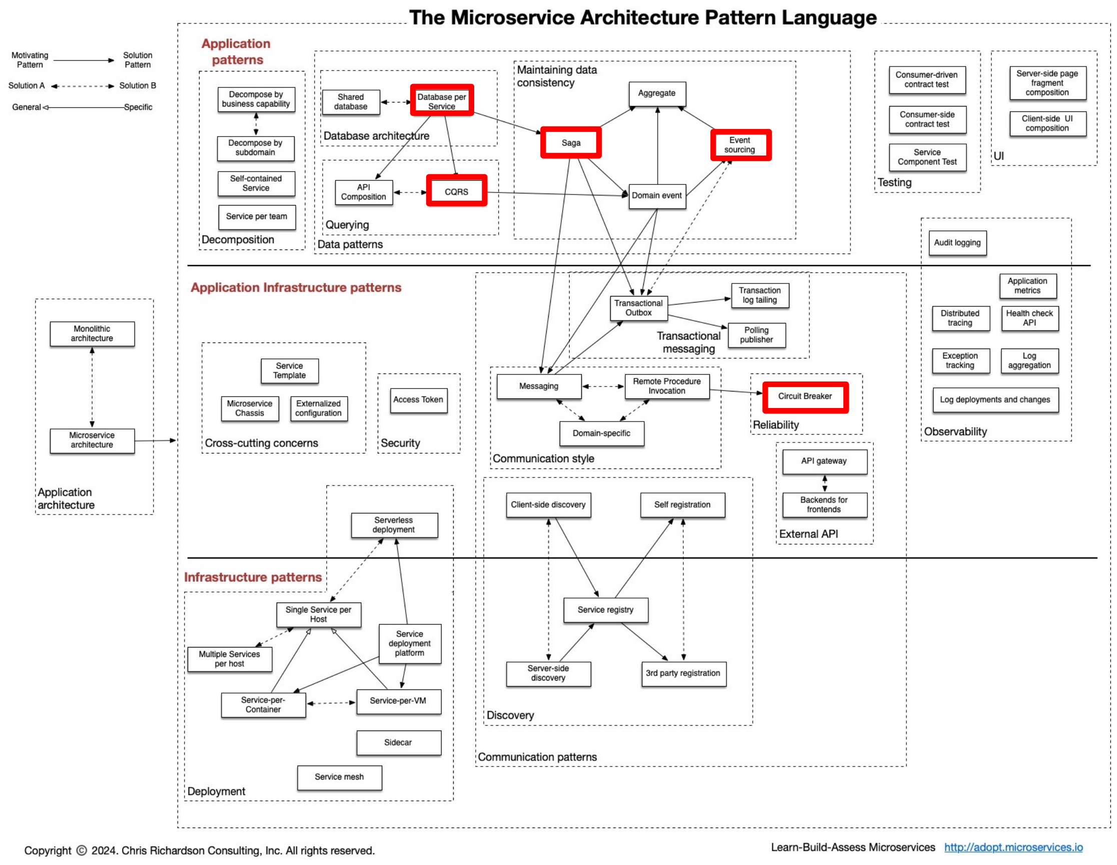

Business capabilities
Product catalog management
Inventory management
Order management
Delivery management
Application architecture
<<service>> Product catalog management
<<service>> Inventory management
<<service>> Order management
<<service>> Delivery management

Main decomposition patterns

2. Decompose by domain-driven design (DDD) subdomain
   - A domain is divided into multiple subdomains, each one corresponding to a different part of the business
   - Domain-modeling view
   - Good for complex domains with rich business rules
   - E.g., *Order Management*, *Inventory*, *Product Catalogue*, *Delivery*

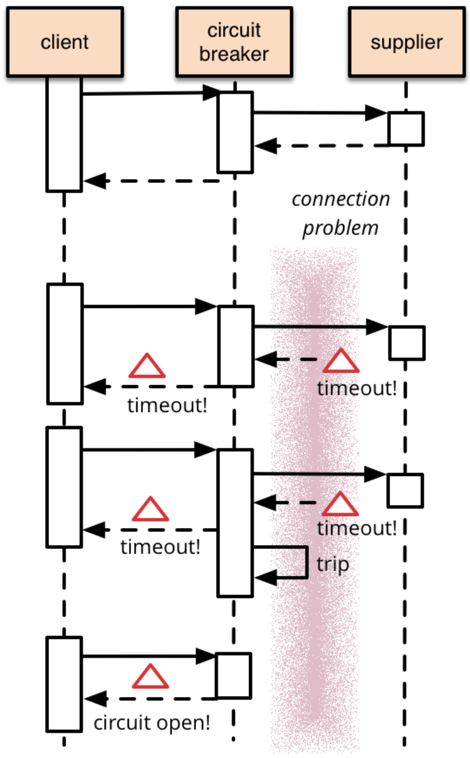

Domain
Subdomain: Order management
Order management service
Subdomain: Product catalog management
Product Catalog service
Inventory service
Subdomain: Delivery management
Subdomain: Inventory management
Delivery service

Microservice patterns



Reliability patterns: Circuit breaker

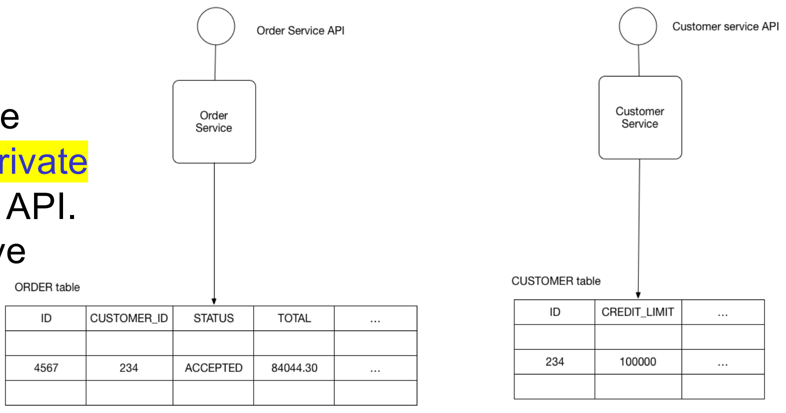
- Problem: how to prevent a network or service failure from cascading to other services?
- Solution: a service client invokes a remote service via a proxy that behaves like an electrical circuit breaker
  - Requests flow normally while the circuit is closed
  - When failures exceed a threshold, the circuit opens and requests fail immediately (optional fallback response)
  - After a timeout, the circuit enters half-open state and allows limited test requests
  - On success, it closes again; on failure, it reopens and restarts the timeout



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

Data patterns: Database per service

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



ORDER table

ID	CUSTOMER_ID	STATUS	TOTAL	...
4567	234	ACCEPTED	84044.30	...

CUSTOMER table

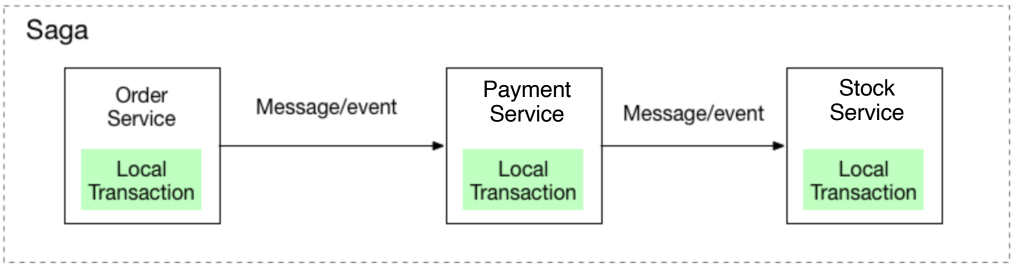
ID	CREDIT_LIMIT	...
234	100000	...

- Pros and cons
  - ✓ Helps ensure loose coupling among services
  - ✓ Each service can use the most appropriate DB type (e.g., KV data store, graph database)
  - ✗ Transactions spanning multiple services are more complex
  - ✗ Managing multiple DBs increases operational complexity
    - A dedicated DB server per service is not required
- 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, but some transactions span multiple services: how to maintain data consistency across services without using distributed transactions (e.g., two-phase commit)?
- Solution: implement each cross-service transaction as a saga
- **Saga:** a sequence of local transactions
  - Each local transaction updates its DB and publishes a message/event to trigger the next transaction
  - If a local transaction fails, the saga executes compensating transactions to undo changes made by preceding transactions (rollback)

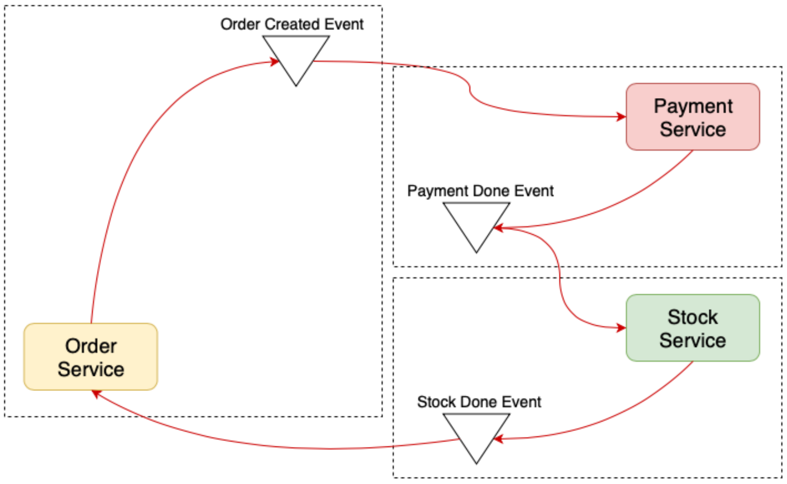


https://microservices.io/patterns/data/saga.html

Data patterns: Saga

- 2 ways to coordinate a saga:
  - Choreography: each local transaction publishes events that trigger local transactions in other services
  - Orchestration

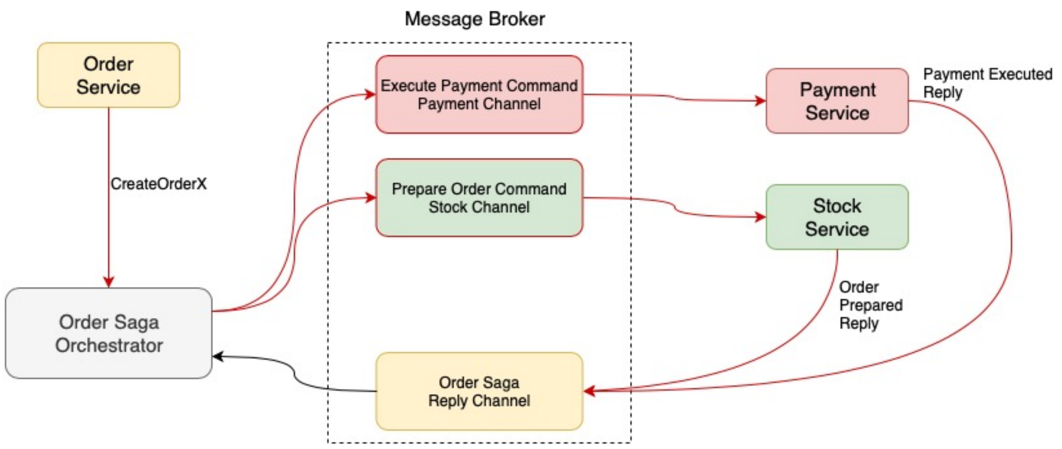
Choreography



Data patterns: Saga

- 2 ways to coordinate a saga:
  - Choreography
  - Orchestration: a central orchestrator tells each service which local transaction to execute

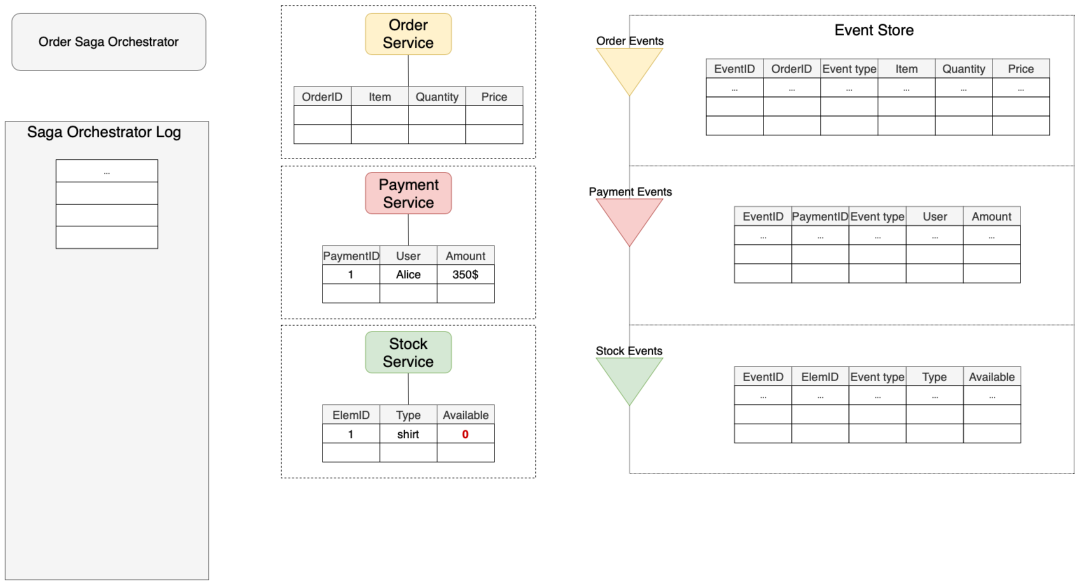
Orchestration



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



OrderID	Item	Quantity	Price

PaymentID	User	Amount
1	Alice	350$

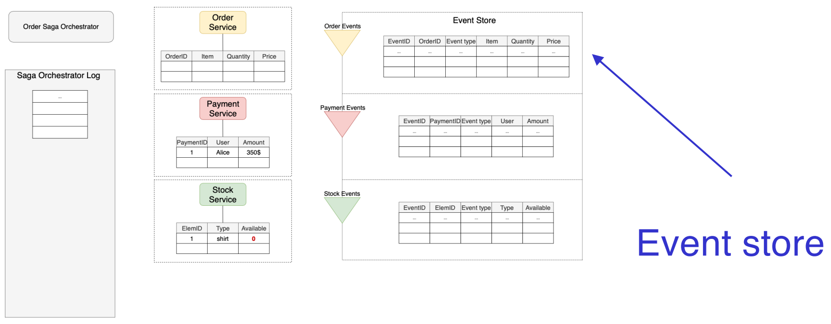
ElemID	Type	Available
1	shirt	0

EventID	OrderID	Event type	Item	Quantity	Price
...	...	...	...	...	...

EventID	PaymentID	Event type	User	Amount
...	...	...	...	...

EventID	ElemID	Event type	Type	Available
...	...	...	...	...

Data patterns: event sourcing



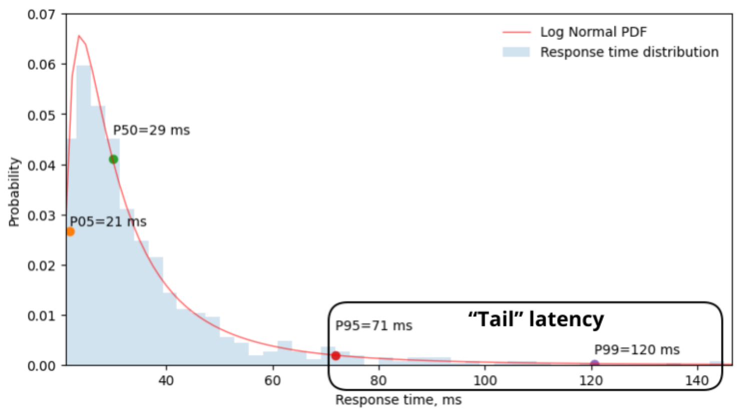
- In the saga, we also use another pattern: event sourcing https://microservices.io/patterns/data/event-sourcing.html
  - Problem: a service participating in a saga must atomically update its DB and publishes messages/events to the orchestrator to avoid data inconsistencies
  - Solution: persist a sequence of domain events representing state changes; store events in an *append-only* event store (a DB of events); service state can be reconstructed by replaying events

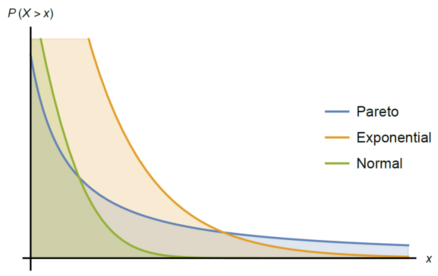
Data patterns: CQRS

- Problem: how to query data from multiple services in a microservice architecture? How to separate read and write load to scale each independently?
- Solution: use a view DB, a read-only replica designed to support queries
  - Application keeps the view updated by subscribing to domain events published by the service that owns the data https://microservices.io/patterns/data/domain-event.html
- Known as Command Query Responsibility Segregation (CQRS), i.e., separate commands (writes) from queries (reads) for a data store https://microservices.io/patterns/data/cqrs.html

Microservices observability challenge

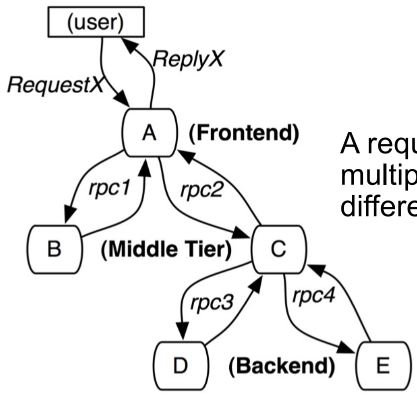
- Service distribution, even at large scale: difficult to monitor and capture causal and temporal relationships among microservices
- Need for monitoring
  - To debug the application
  - To analyze performance and latency, including tail latency





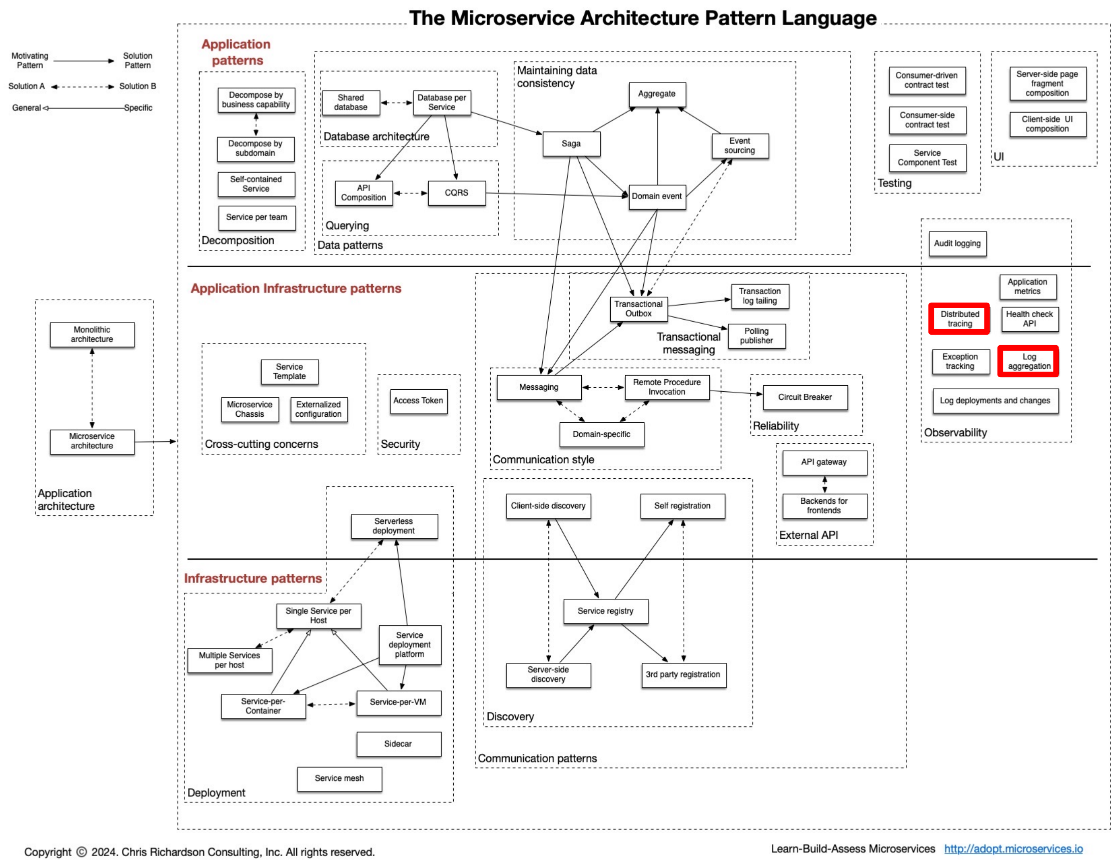
Microservices observability challenge

- Need for monitoring
  - To analyze service dependencies
  - To identify the root cause of anomalies, which requires:
    - Constructing a service dependency graph that shows the sequence of invoked microservices
    - Localizing the root cause microservices using the graph, traces, logs, and KPIs



A request is passed through multiple microservices with different functionalities

Microservice patterns



Observability patterns: Log aggregation

- Problem: how to understand application behavior and troubleshoot problems?
- Solution: use a centralized logging service that aggregates logs from all microservice instances
  - DevOps team can search and analyze logs and configure alerts triggered by specific log messages
  - E.g., AWS CloudWatch, Splunk

✗ Centralized (if physical, not only logical)

✗ Handling large volumes of logs requires substantial infrastructure

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

Observability patterns: Distributed tracing

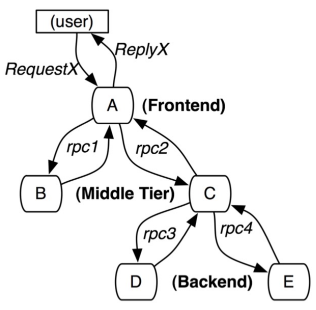
- Problem: how to understand complex app behavior and troubleshoot problems?
- Solution: instrument microservices with code to
  - Assign a unique request ID (*trace ID*) to each user request
  - Pass the trace ID through all microservices involved in handling the request
  - Include the trace ID in log messages
  - Record trace context (e.g., start time, operation, duration) in a distributed store

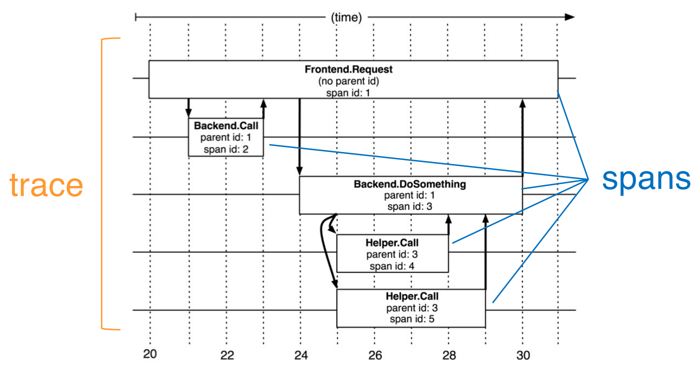
✗ Storing and aggregating traces may require significant infrastructure

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

Monitoring microservices: tools

- Dapper
  - Google's distributed tracing system
  - Based on *spans* and *traces*
    - Span: individual unit of work in an application(e.g., HTTP request, call to DB); includes operation name, start time, and duration
    - Trace: collection of spans in a parent/child relationship (can be seen as a DAG); shows how requests propagate through services and other components




(time)
trace
spans
Frontend.Request (no parent id) span id: 1
Backend.Call parent id: 1 span id: 2
Backend.DoSomething parent id: 1 span id: 3
Helper.Call parent id: 3 span id: 4
Helper.Call parent id: 3 span id: 5
20
22
24
26
28
30

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

Monitoring microservices: tools

- Dapper trace sampling and storage
- Traces are sampled using an adaptive rate
  - Why sample traces?
    - Storing all traces consume excessive storage, generate high network traffic, and introduce significant application overhead
  - Sampling rate adjusts dynamically based on traffic volume and system behavior
- Trace collection and storage
  - Span data is written to local log files by each service
  - Dapper daemons pull the logs and send them through a collection infrastructure
  - Traces are stored in BigTable, with one row per trace ID

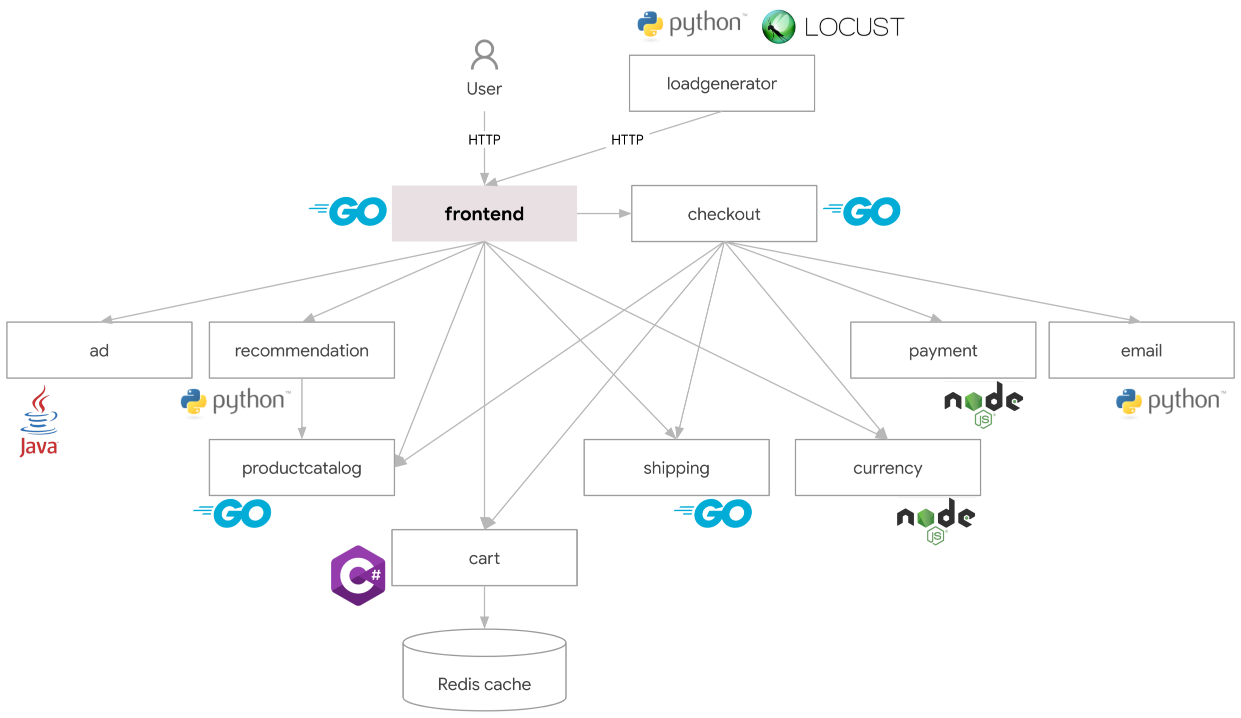
Monitoring microservices: tools

- Open-source distributed tracing tools
  - Jaeger https://www.jaegertracing.io
    - Inspired by Dapper, supports large-scale trace analysis (optionally via Spark/Flink)
  - Zipkin https://zipkin.io
    - Lightweight distributed tracing system
  - OpenTelemetry https://opentelemetry.io
    - Industry-standard instrumentation framework
    - Broad language support
    - Integrated with popular frameworks and libraries
    - Exports traces to multiple backends (e.g., Jaeger, Zipkin)
- Need for standards to support interoperability across tracing tools
  - W3C Trace Context defines a standardized format for propagating tracing data https://www.w3.org/TR/trace-context-2/

Example microservices app

- Google's Online Boutique
  https://github.com/GoogleCloudPlatform/microservices-demo
  - Online store where users can browse items, add them to the cart, and purchase them
- Composed of 11 microservices written in different languages
  - Polyglot microservices: a "renaissance" in programming language diversity
- How to build a polyglot application?
  1. REST and JSON as message interchange format
  2. gRPC and protocol buffers as IDL and message interchange format: chosen approach in Online Boutique

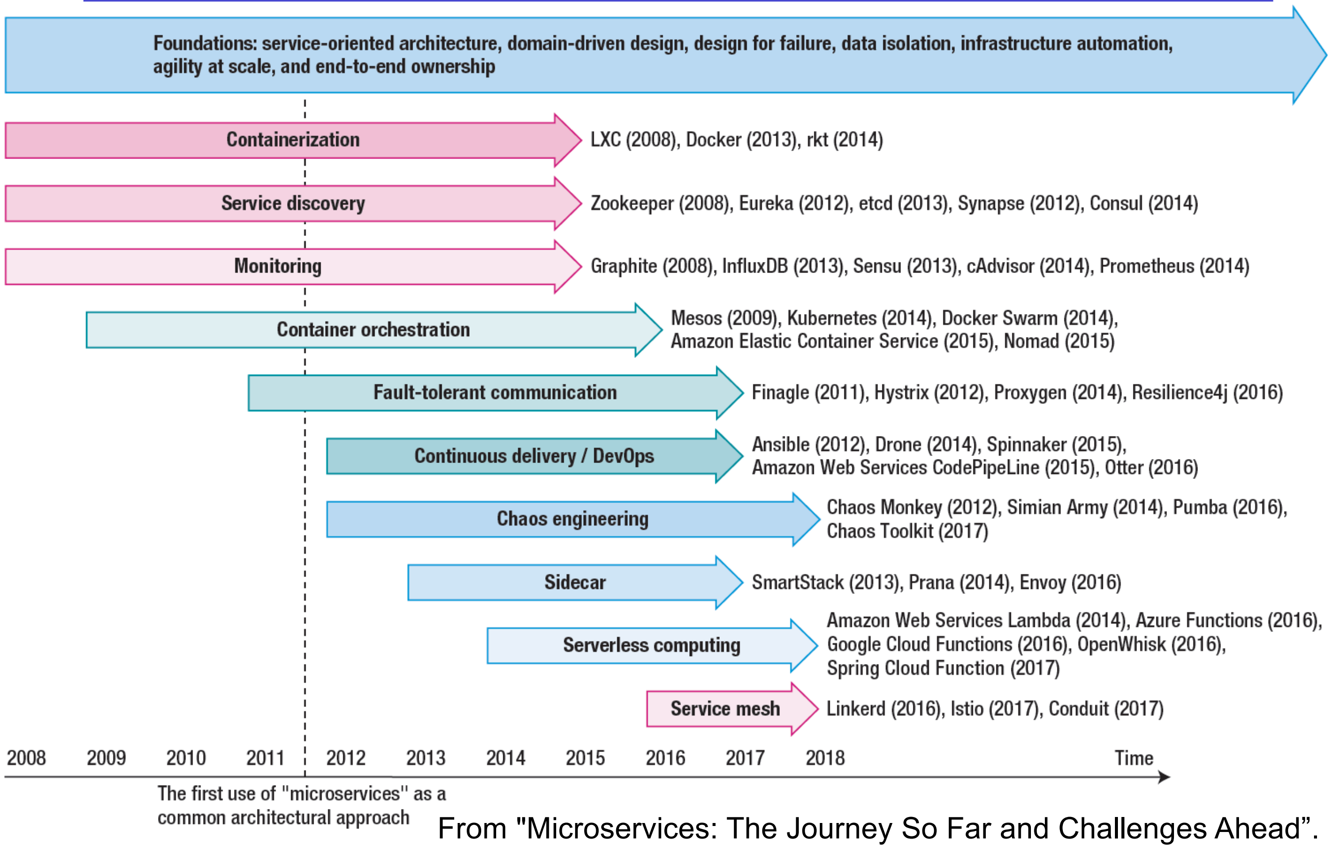
Online Boutique: architecture



Online Boutique: features

- Composed of 10 microservices (plus a load generator), written in different languages and communicating using gRPC
- Technologies demonstrated by Google Online Boutique:
  - Kubernetes and Google Kubernetes Engine (GKE): container orchestration
  - gRPC: we know it ☺
  - Istio / Cloud Service: service mesh for traffic management, security and observability
  - Google Cloud 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 for Kubernetes and containers development https://skaffold.dev

Microservice technologies timeline


Foundations: service-oriented architecture, domain-driven design, design for failure, data isolation, infrastructure automation, agility at scale, and end-to-end ownership
Containerization
LXC (2008), Docker (2013), rkt (2014)
Service discovery
Zookeeper (2008), Eureka (2012), etcd (2013), Synapse (2012), Consul (2014)
Monitoring
Graphite (2008), InfluxDB (2013), Sensu (2013), cAdvisor (2014), Prometheus (2014)
Container orchestration
Mesos (2009), Kubernetes (2014), Docker Swarm (2014), Amazon Elastic Container Service (2015), Nomad (2015)
Fault-tolerant communication
Finagle (2011), Hystrix (2012), Proxygen (2014), Resilience4j (2016)
Continuous delivery / DevOps
Ansible (2012), Drone (2014), Spinnaker (2015), Amazon Web Services CodePipeLine (2015), Otter (2016)
Chaos engineering
Chaos Monkey (2012), Simian Army (2014), Pumba (2016), Chaos Toolkit (2017)
Sidecar
SmartStack (2013), Prana (2014), Envoy (2016)
Serverless computing
Amazon Web Services Lambda (2014), Azure Functions (2016), Google Cloud Functions (2016), OpenWhisk (2016), Spring Cloud Function (2017)
Service mesh
Linkerd (2016), Istio (2017), Conduit (2017)
2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 Time
The first use of "microservices" as a common architectural approach

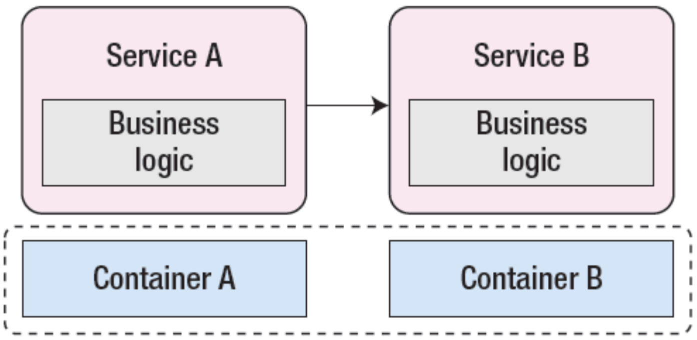
From "Microservices: The Journey So Far and Challenges Ahead".

Generations: at the beginning

- 4 generations of microservice architectures
- 1st generation: containers and orchestration
  - **Container-based virtualization** (e.g., Docker)
  - **Service discovery**, 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/

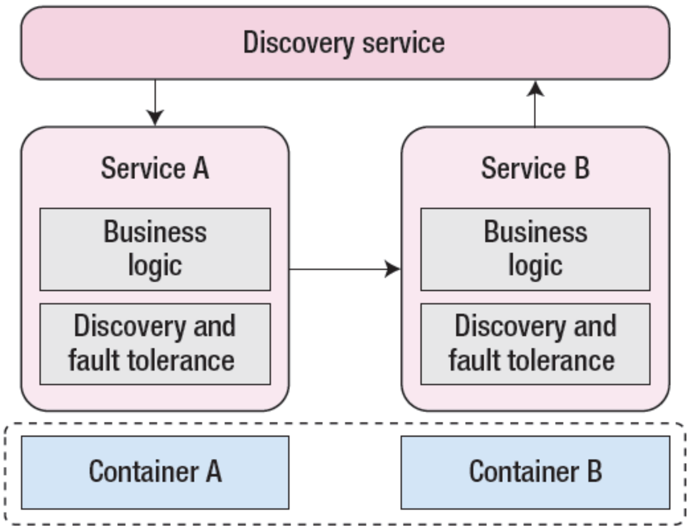
Generations: container orchestration

- **Container orchestration**
  - E.g., Kubernetes, Docker Swarm
  - Automates container deployment, scaling, and management
  - Abstracts physical/virtual infrastructure from developers
- Limitation
  - Application-level fault tolerance implemented inside microservice code



Generations: service discovery and fault tolerance

- 2nd generation service discovery tools and fault-tolerant (FT) communication libraries
  - Goal: enable more efficient and reliable service-to-service communication
  - Use FT communication libraries implementing resiliency patterns: circuit breaker, fallback, retry/timeout

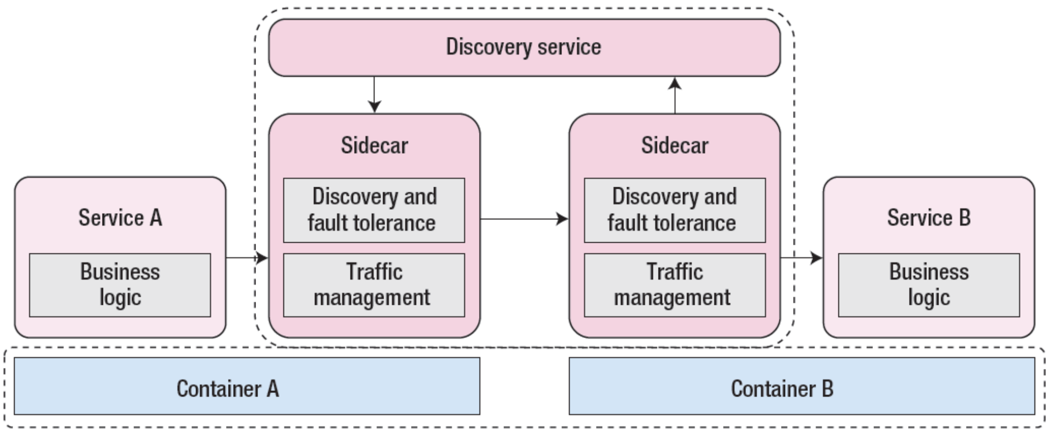


Generations: service discovery and fault tolerance

- Examples:
  - Consul: initially service discovery, now service mesh https://developer.hashicorp.com/consul
  - Finagle: protocol-agnostic FT RPC library https://github.com/twitter/finagle
  - Resilience4j: Java FT library with multiple resiliency patterns https://resilience4j.readme.io
- Limitation
  - Developers must explicitly integrate FT libraries into application code

Generations: service mesh

- 3rd generation: service mesh and sidecar proxies
  - Encapsulate communication concerns: service discovery, load balancing, security, fault tolerance, observability
  - Benefits
    - Abstract communication logic away from application code
    - Improve software reuse
    - Provide a homogeneous communication interface



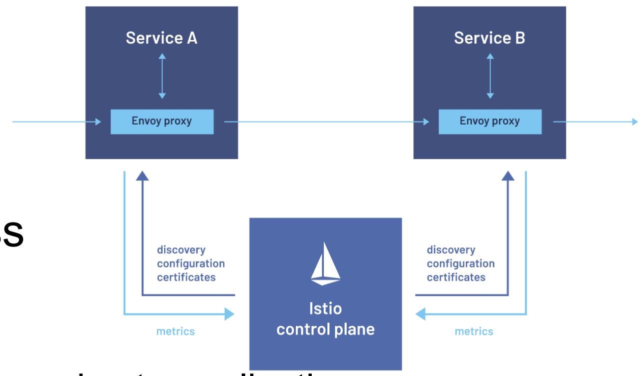
Service mesh

- A dedicated infrastructure layer for microservice apps that facilitates service-to-service communication
- Provided features (no need to embed them into application logic) https://servicemesh.es
  - Traffic management: service discovery, load balancing, routing
  - Security: authentication, encryption, and authorization between services
  - Resilience: circuit breaking, retries, timeouts, and fallbacks
  - Observability: monitoring, logging, and tracing service interactions
- Popular products:
  - Istio https://istio.io
  - Linkerd https://linkerd.io
  - Consul https://www.consul.io

Service mesh: architecture

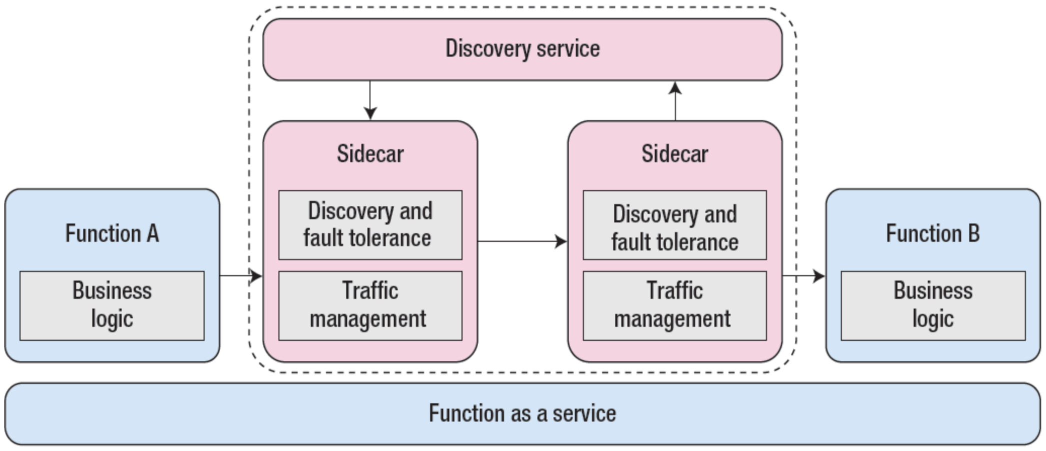
- Composed of data plane and control plane
- Data plane: decentralized
  - Sidecar proxies deployed alongside each microservice
  - Each proxy handles inbound and outbound traffic for the service it is attached to
- Control plane: centralized
  - Centralized configuration management for all proxies
- Recent approach: proxyless service mesh
  - Moves service-to-service communication from sidecar proxies to application or a lightweight, centralized proxy
  - Traffic management, security, and observability are still managed by the mesh

  ✓ Reduces resource overhead and simplifies deployment



Generations: serverless

- 4th generation: Function as a Service (FaaS) and serverless computing
  - Further simplify microservice development and delivery

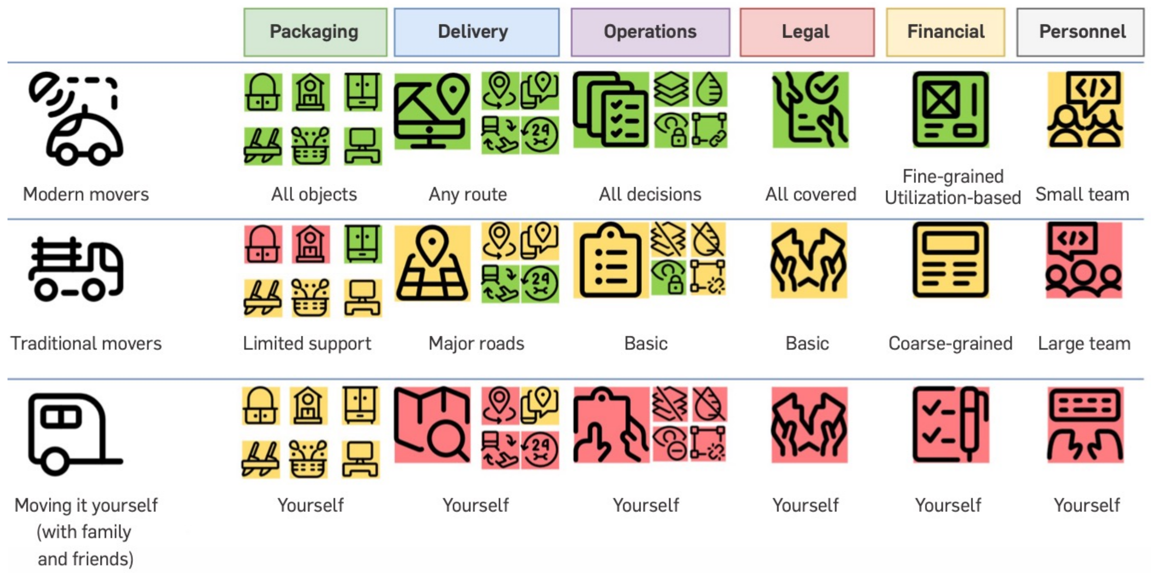


Serverless computing

- Cloud computing model that abstracts server management and low-level infrastructure decisions away from users through full automation
- Key characteristics
  - Users develop, run, and manage application code (**functions**)
  - No need to provision, manage, or scale computing resources
  - The runtime environment is fully managed by the cloud (or private platform) provider
- Note: serverless does not mean no servers
  - Functions still run on servers, we simply do not manage or care about them

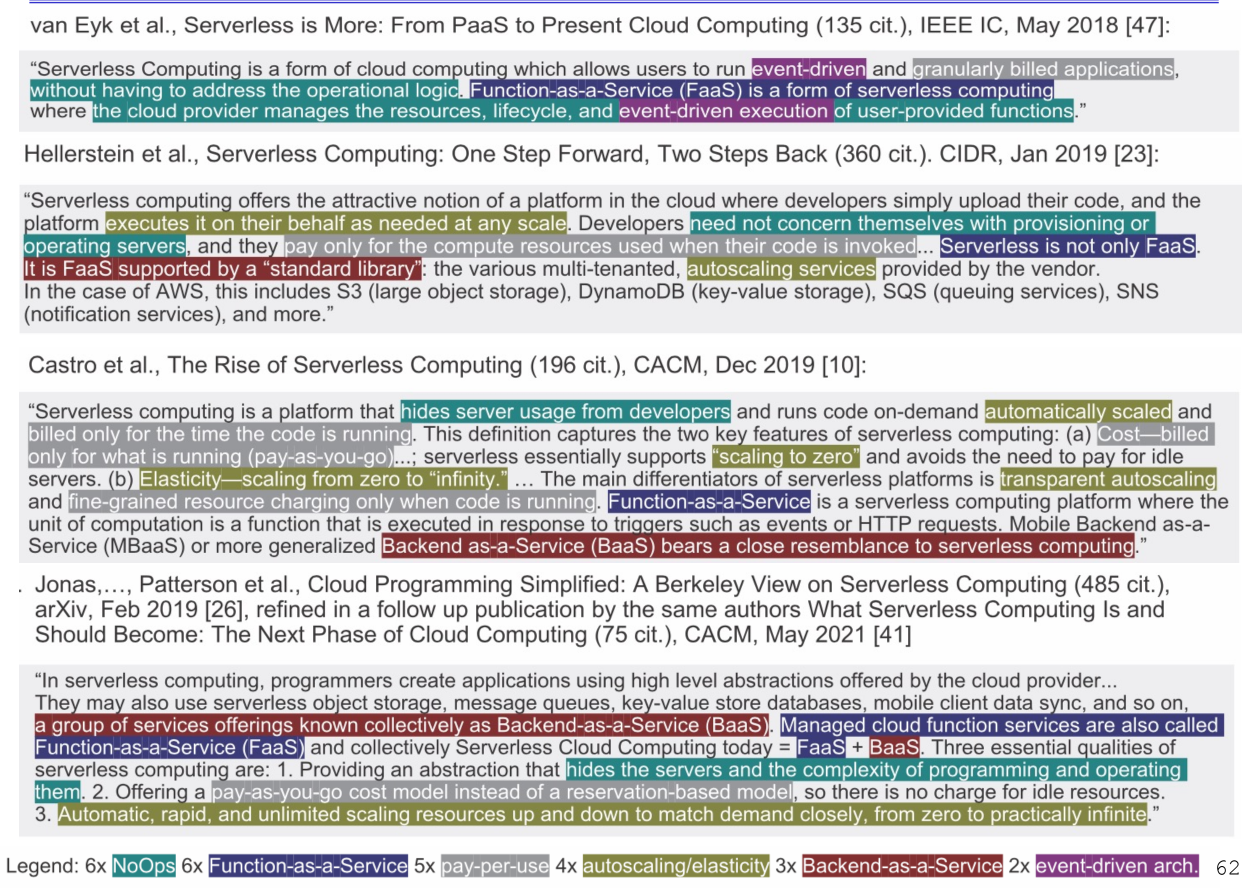
Serverless through an analogy

- Services for moving homes



		Packaging	Delivery	Operations	Legal	Financial	Personnel
serverless	Modern movers	All objects	Any route	All decisions	All covered	Fine-grained Utilization-based	Small team
IaaS/PaaS cloud	Traditional movers	Limited support	Major roads	Basic	Basic	Coarse-grained	Large team
self-hosting	Moving it yourself (with family and friends)	Yourself	Yourself	Yourself	Yourself	Yourself	Yourself

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.

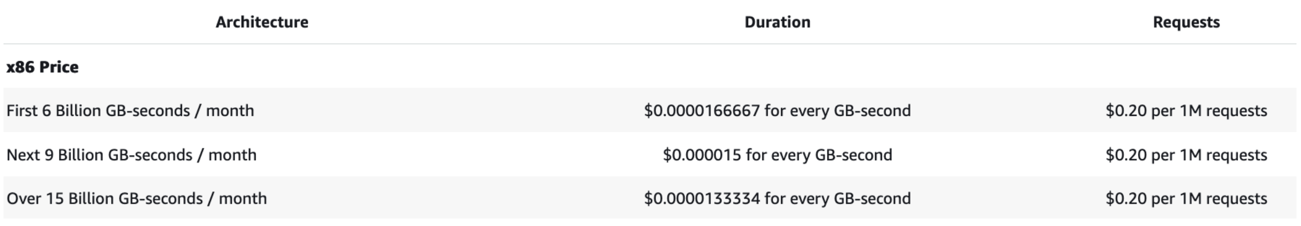
Providers apply utilization-based billing: they charge cloud users with fine granularity, in proportion to the resources that applications actually consume from the cloud infrastructure, such as computing time, memory, and storage space.

Serverless, FaaS and BaaS

- Function as a Service (FaaS) and serverless often used interchangeably, some discussion on difference
- FaaS is the most prominent model 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)
  - Provides developers with backend functions via API, such as authentication, data storage, real-time messaging, and push notifications
  - Examples:
    - AWS DynamoDB (no-demand mode): pay-per-request pricing and automatic scaling
    - Google Cloud Firestore (NoSQL document DB) and Pub/Sub
    - AWS Amplify: serverless development platform that simplifies building, deploying, and managing web and mobile applications

Serverless: features

- Ephemeral compute resources
  - May exist only for the duration of a function invocation
  - ✗ Cold start: if no container or microVM is ready, the function waits until a new instance is launched
- Automated (i.e., zero configuration) elasticity
  - Compute resources auto-scale transparently from zero to peak load and back
- True pay-per-use: fine-grained and utilization-based
  - E.g., AWS Lambda pricing is based on the number of invocation requests and the duration it takes for code to run



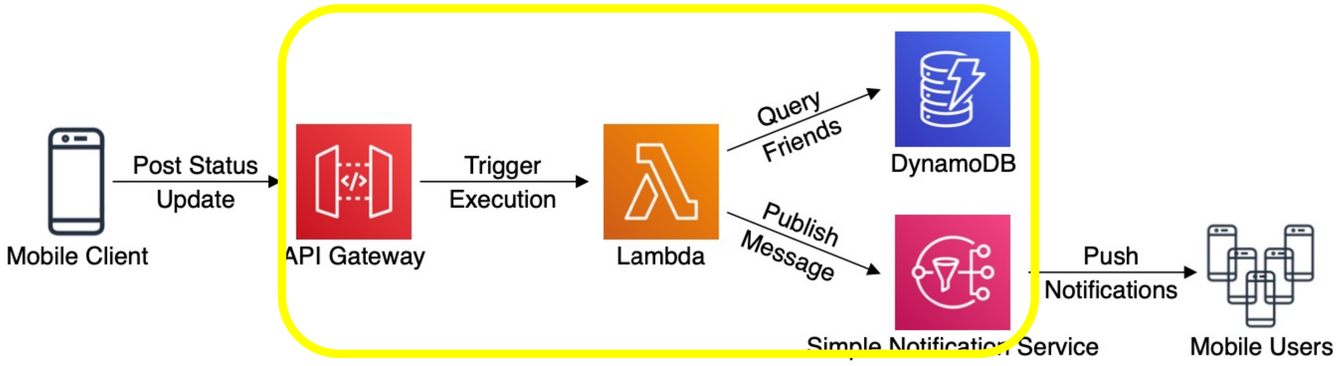
Architecture	Duration	Requests
x86 Price		
First 6 Billion GB-seconds / month	$0.0000166667 for every GB-second	$0.20 per 1M requests
Next 9 Billion GB-seconds / month	$0.000015 for every GB-second	$0.20 per 1M requests
Over 15 Billion GB-seconds / month	$0.0000133334 for every GB-second	$0.20 per 1M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

Serverless: features

- Event-driven execution
  - Functions are triggered by events (e.g., file upload, message queue, HTTP request)
  - Infrastructure is dynamically allocated to execute the function code in response to the event
- NoOps (no operations)
  - Simplifies the deployment process: no need for developers to manage scaling, capacity planning, or infrastructure maintenance
  - Developers focus only on business logic
- Supports diverse applications
  - Use cases range from enterprise automation and real-time data analytics to scientific computing and ML inference

Serverless application: a first example

- Propagating updates in a social media app in a serverless fashion
  1. User creates and sends a status update
  2. Platform orchestrates operations needed to propagate the update inside the social media platform and to user's friends using serverless (AWS Lambda) and other cloud services (AWS DynamoDB and SNS)
  3. Friends receive the update

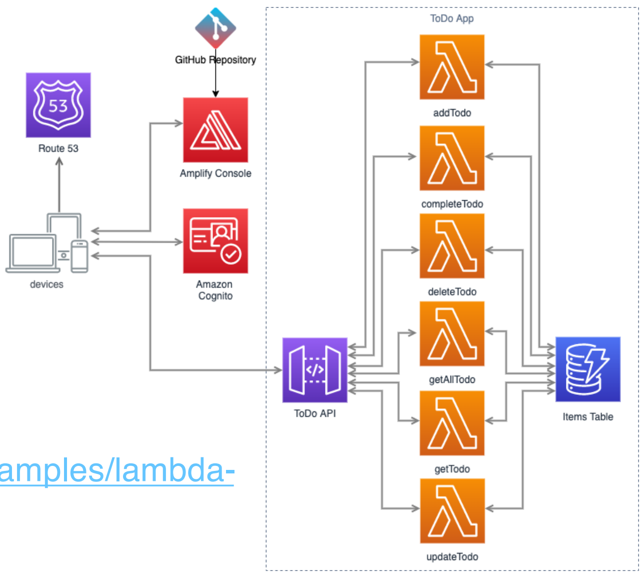


Serverless Cloud services

- Several Cloud providers offer serverless computing as fully managed service on their public clouds
  - 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)
  - Auto-scaling is handled automatically by the cloud provider
- Cloud platforms also offer supporting services to operate serverless ecosystems
  - E.g., event notifications, storage, message queues, DBs

Example: AWS reference Web app

- A simple "to-do list" web app that allows registered users to create, update, view, and delete to-do items
- Event-driven web app uses AWS Lambda and Amazon API Gateway for the business logic, DynamoDB as database, and Amplify Console to host static content

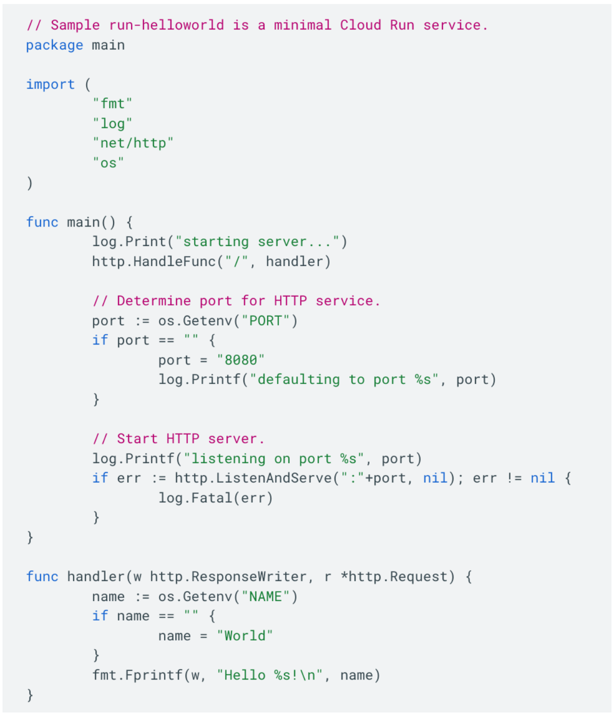


https://github.com/aws-samples/lambda-refarch-webapp

Example: Google Cloud Run Functions

- “Hello World” example from Google using Go
  - Basic web server that replies with “Hello, World!”

https://docs.cloud.google.com/run/docs/quickstarts/build-and-deploy/deploy-go-service



```go
// Sample run-helloworld is a minimal Cloud Run service.
package main

import (
	"fmt"
	"log"
	"net/http"
	"os"
)

func main() {
	log.Print("starting server...")
	http.HandleFunc("/", handler)

	// Determine port for HTTP service.
	port := os.Getenv("PORT")
	if port == "" {
		port = "8080"
		log.Printf("defaulting to port %s", port)
	}

	// Start HTTP server.
	log.Printf("listening on port %s", port)
	if err := http.ListenAndServe(":"+port, nil); err != nil {
		log.Fatal(err)
	}
}

func handler(w http.ResponseWriter, r *http.Request) {
	name := os.Getenv("NAME")
	if name == "" {
		name = "World"
	}
	fmt.Fprintf(w, "Hello %s!\n", name)
}
```

Serverless: state

- Stateless functions are easy to manage (horizontal scalability, fast recovery)
  - However, stateless functions are not enough for some applications (e.g., ML, streaming)
- How to support stateful computation?
  1. Externalize state (e.g., handed over to an external shared storage system), so functions themselves remain stateless
     - Requires efficient access to shared state, so to keep auto-scaling benefits
  2. Embed state into the serverless runtime: the serverless platform manages state transparently and colocates it with computation; the state is automatically persisted, replicated, and recovered
     - Examples: Azure Stateful Functions, Cloudflare Durable Objects
- 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: can introduce performance variability, especially during traffic spikes
- Programming languages
  - Language support varies by cloud provider
  - Choice of language affects cold start times, function performance, and cost
- Security
  - E.g., more entry points, financial exhaustion attacks
- Resources
  - Limits on resource amount (e.g., in AWS Lambda per-function memory between 128 MB and 10 GB)
  - GPU support

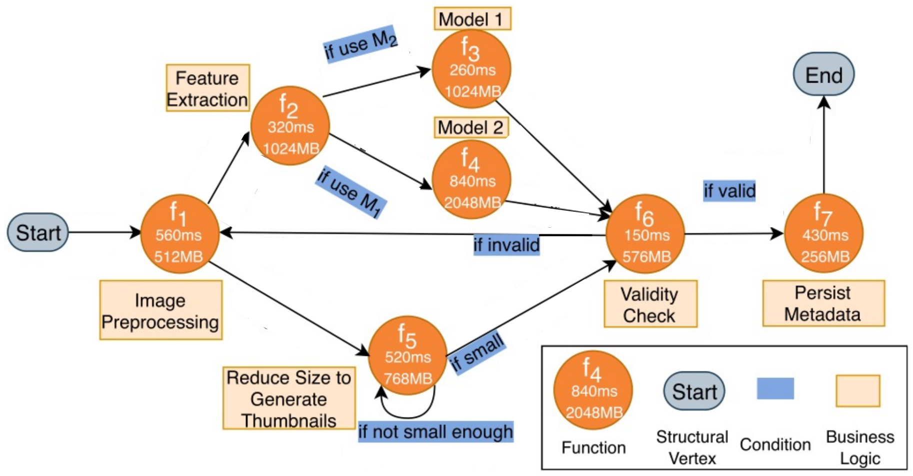
$\Big\}$ Lower flexibility

Serverless: challenges and limitations

- Cost-saving
  - Not always cost-saving for users, possibly leading to expense explosion
  - Not cost-saving: high, predictable traffic, long-running applications, complex applications, high memory/CPU needs
- Vendor lock-in
  - Serverless offerings are tied to specific cloud providers

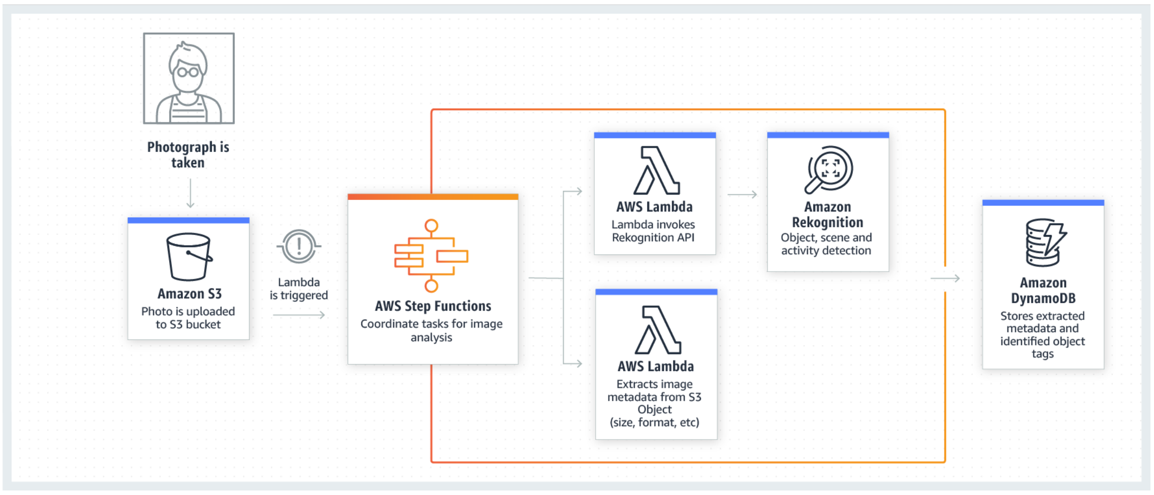
Composition of serverless functions

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



Example: AWS Step Functions

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

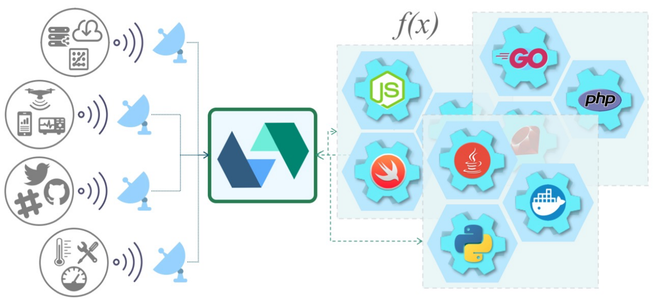

Photograph is taken
Amazon S3
Photo is uploaded to S3 bucket
Lambda is triggered
AWS Step Functions
Coordinate tasks for image analysis
AWS Lambda
Lambda invokes Rekognition API
Amazon Rekognition
Object, scene and activity detection
AWS Lambda
Extracts image metadata from S3 Object (size, format, etc)
Amazon DynamoDB
Stores extracted metadata and identified object tags

Open-source serverless platforms

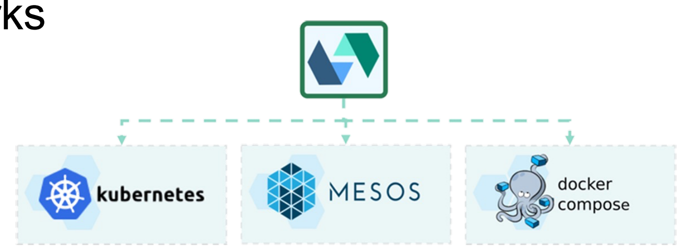
- Can run on commodity hardware
- Popular serverless platforms
  - Apache OpenWhisk https://openwhisk.apache.org
  - OpenFaaS https://www.openfaas.com
  - Fission https://fission.io
  - Knative https://knative.dev
  - Nuclio https://nuclio.io
- Most platforms rely on Kubernetes for orchestration and 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

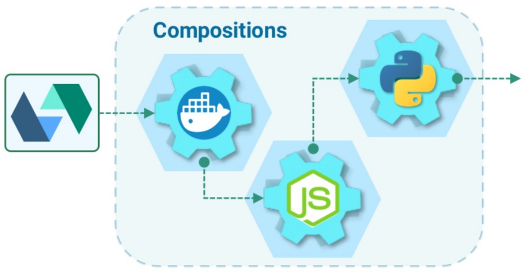


- Functions run inside Docker containers
- Support for multiple container orchestration frameworks



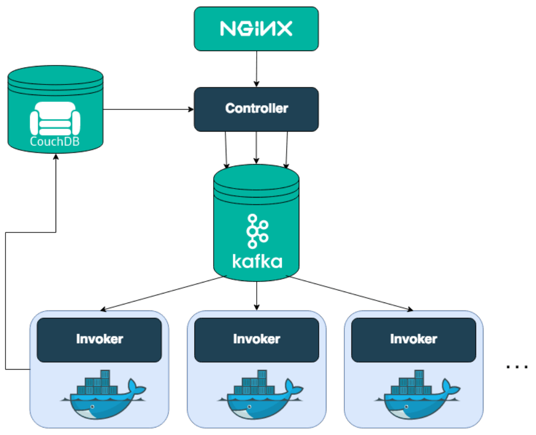
OpenWhisk

- 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



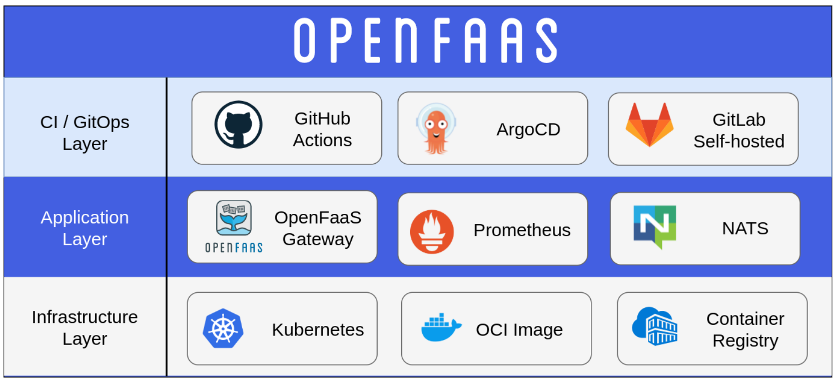
OpenWhisk: architecture

- Powered by multiple frameworks
  - NGINX: entry point that receives HTTP requests and forwards them to Controller
  - Controller: translates HTTP requests into invocations of the appropriate action; manages the lifecycle of actions and coordinates between other components
  - CouchDB (document-oriented NoSQL data store): stores authentication and authorization info, action code, ...
  - Kafka: mediates communication between Controller and Invokers
  - Docker: used by Invokers to execute action code



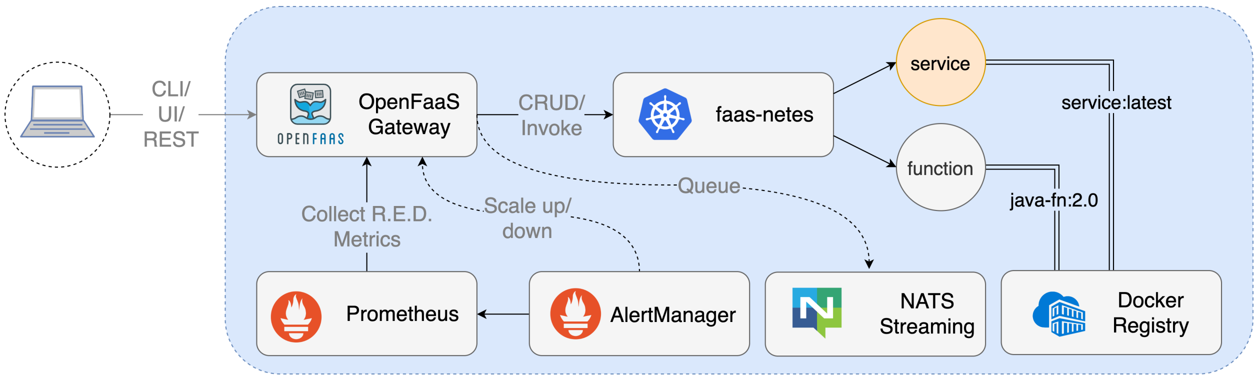
OpenFaaS

- Distributed serverless framework, 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

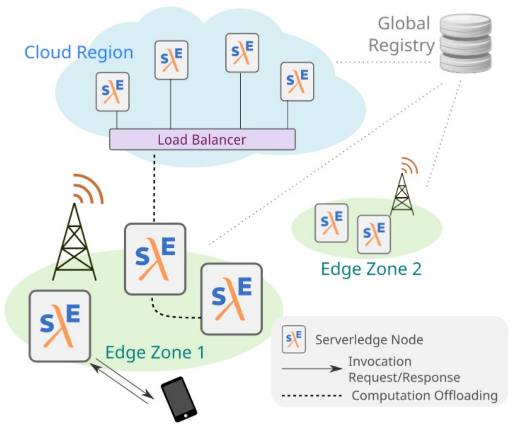
- Conceptual workflow
  – Gateway can be accessed through its REST API, CLI or UI
  – Prometheus collects metrics which are made available via gateway's API and are used for auto-scaling
  – NATS enables function invocations to run asynchronously



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

Serverless in the compute continuum

- Open-source serverless platforms typically rely on centralized components, which make them unsuitable for the compute continuum
- We are developing Serverledge, a decentralized FaaS framework: thesis opportunities! https://github.com/serverledge-faas/serverledge



Russo Russo et al., Decentralized Function-as-a-Service for the Edge-Cloud Continuum, Percom 2023 http://www.ce.uniroma2.it/publications/serverledgePerCom2023.pdf

References

- 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
- Roberts, Serverless architectures, https://martinfowler.com/articles/serverless.html
- 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