

Microservices Patterns

Architectural Patterns for Microservices: A Systematic Mapping Study

Davide Taibi¹ and Valentina Lenarduzzi¹ and Claus Pahl²

¹Tampere University of Technology, Tampere, Finland

²Free University of Bozen-Bolzano, Bozen-Bolzano, Italy

{davide.taibi, valentina.lenarduzzi}@tut.fi, claus.pahl@unibz.it

Keywords: Microservices, Architectural Style, Architecture Pattern, Cloud Native, Cloud Migration, DevOps.

Abstract: Microservices is an architectural style increasing in popularity. However, there is still a lack of understanding how to adopt a microservice-based architectural style. We aim at characterizing different microservice architectural styles and their principles. We present a systematic mapping study that can be used as a reference for future research in order to identify reported usage of microservices, and based on these, we extract common patterns and principles. We present two key contributions. Firstly, we identified several agreed microservice architecture patterns that seem widely adopted and reported in the case studies identified. Secondly, we presented these as a catalogue in a common template format including a summary of the advantages, disadvantages, and lessons learned for each pattern from the case studies. We can conclude that different architecture patterns emerge for different migration, orchestration, storage and deployment settings for a set of agreed principles.

1 Introduction

Microservices are increasing their popularity in industry, being adopted by several big players such as Netflix, Spotify, Amazon and many others and several companies are now following the trend, migrating their monolithic systems to microservices. However, still have the issues of selecting the most appropriate architectural patterns, mainly because of the lack of knowledge about the available patterns (Taibi et al., 2017).

Microservices are small autonomous services deployed independently, with a single and clearly defined purpose (Lewis and Fowler, 2014). Their independent deployability is advantageous for continuous delivery. They can scale independently from other services, and they can be deployed on the hardware that best suits their needs. Moreover, because of their size, they are easier to maintain and more fault-tolerant since the failure of one service will not break the whole system, which could happen in a monolithic system (Lewis and Fowler, 2014). Microservices have emerged as a variant of service-oriented architecture (SOA). Their aim is to structure software systems as sets of small services that are deployable on a different platform and running in their own process while communicating with each other through lightweight mechanisms without a need for centralized control (Pahl et al., 2018). We consider an archi-

tectural style here as a set of *principles* and coarse-grained *patterns* that provide an abstract framework for a family of systems. An architectural style consists of a set of architectural principles and patterns that are aligned with each other to make designs recognizable and design activities repeatable; principles express architectural design intent; patterns adhere to the principles and are commonly occurring (proven) in practice (Zimmermann, 2009).

Microservices enable continuous development and delivery (Pahl et al., 2018). DevOps is the integration of Development and Operations that include a set of continuous delivery practices aimed at decrease the delivery time, increasing the delivery efficiency and reducing time among releases while maintaining software quality. It combines software development, quality assurance, and operations (Bass et al., 2015).

Despite both microservices and DevOps being widely used, there are still challenges in understanding how to construct such kinds of architectures (Balalaie et al., 2016), (Taibi and Lenarduzzi, 2018) and developers often adopt the patterns where they can find more documentation online (Taibi et al., 2017), even if they are not the most appropriate ones. In order to help developers to identify the most appropriate patterns, we aim here to identify and characterize different microservice architecture patterns reported in the literature, be that as proposals and/or as case studies with implementations. The identifi-

CLOSER 2018

INFORTE - Tampere 02.09.2019

Paper Selection

Selection Process	#considered papers	#rejected papers	Validation
Paper extracted from the bibliographic sources	2754		10 random papers independently classified by three researchers
Sift based on title and abstract		2669	Good inter-rater agreement on first sift (K-statistic test)
Primary papers identified	85		
Secondary papers inclusion	858	855	Systematic snowballing (Wohlin, 2014) including all the citations reported in the 85 primary papers and sifting them based on title and abstract
Full papers considered for review	88		Each paper has been read completely by two researchers and 858 secondary papers were identified from references
Sift based on full reading		46	Papers rejected based on inclusion and exclusion criteria
Relevant papers included	42		

Paper Selection

Searched for

(microservice* OR micro-service*) AND (architect* OR migrat* OR modern* OR reengineer* OR re-engineer* OR refactor* OR re-factor* OR rearchitect* OR re-architect* OR evol*)

Included

non peer-reviewed contributions if their number of citations was higher than those of average peer-reviewed papers

Summary

65% of these papers were published at conferences

23% were accepted at workshops

only 7% of the papers were published as journal articles

nearly 5% (2 papers) are non peer-reviewed websites

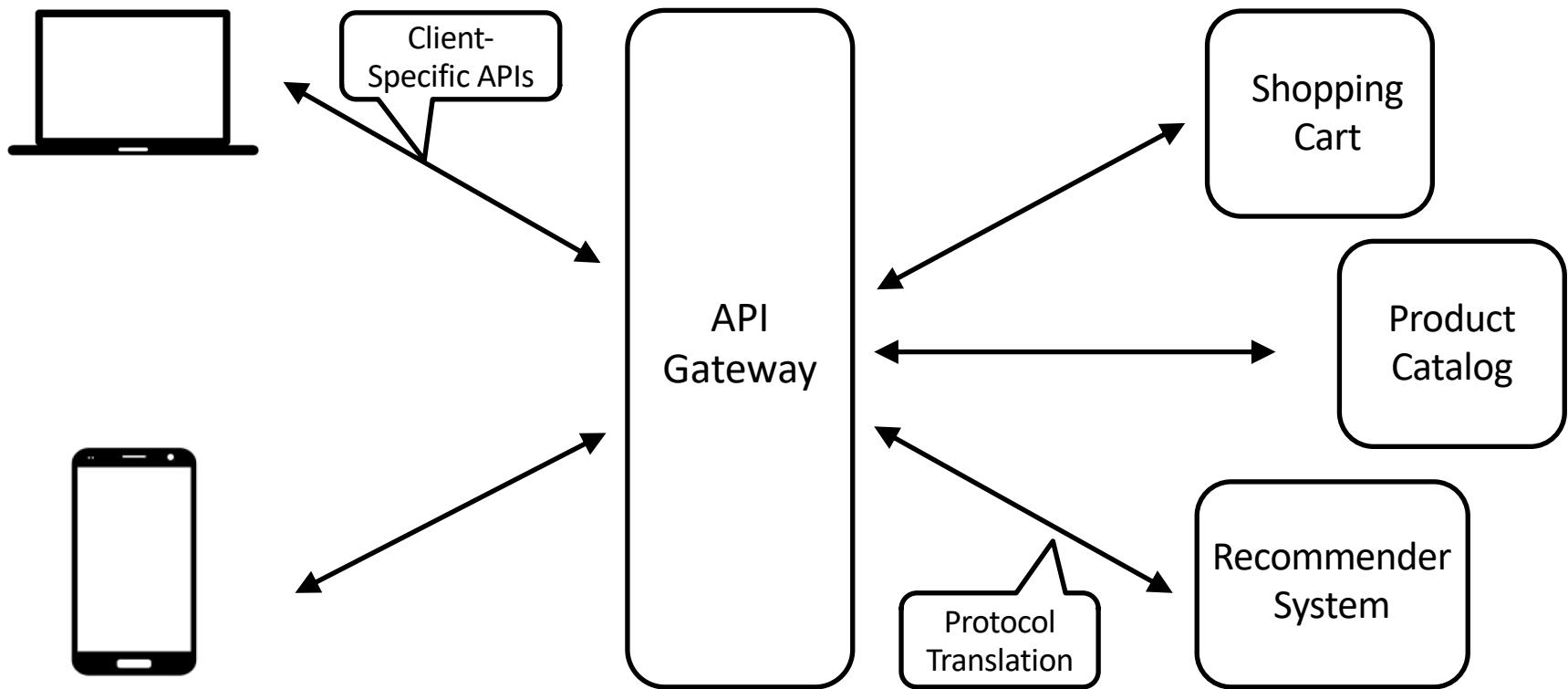
Pattern Categories

Categorize the emerging architecture patterns:

Orchestration and Coordination-oriented architecture patterns that capture communication and coordination from a logical perspective

Patterns reflecting physical **Deployment** strategies for microservices on hosts through containers or virtual machines

Patterns that reflect on data management, specifically **Data Storage** options in addition to the orchestration and coordination patterns oriented at inter-component communication.



API Gateway Pattern

The main goal is:

to increase system performance and simplify interactions,
thus reducing the number of requests per client.

It acts as an entry point for the clients, routing their requests to the connected services, aggregating the required contents, and serving them to the clients

Properties:

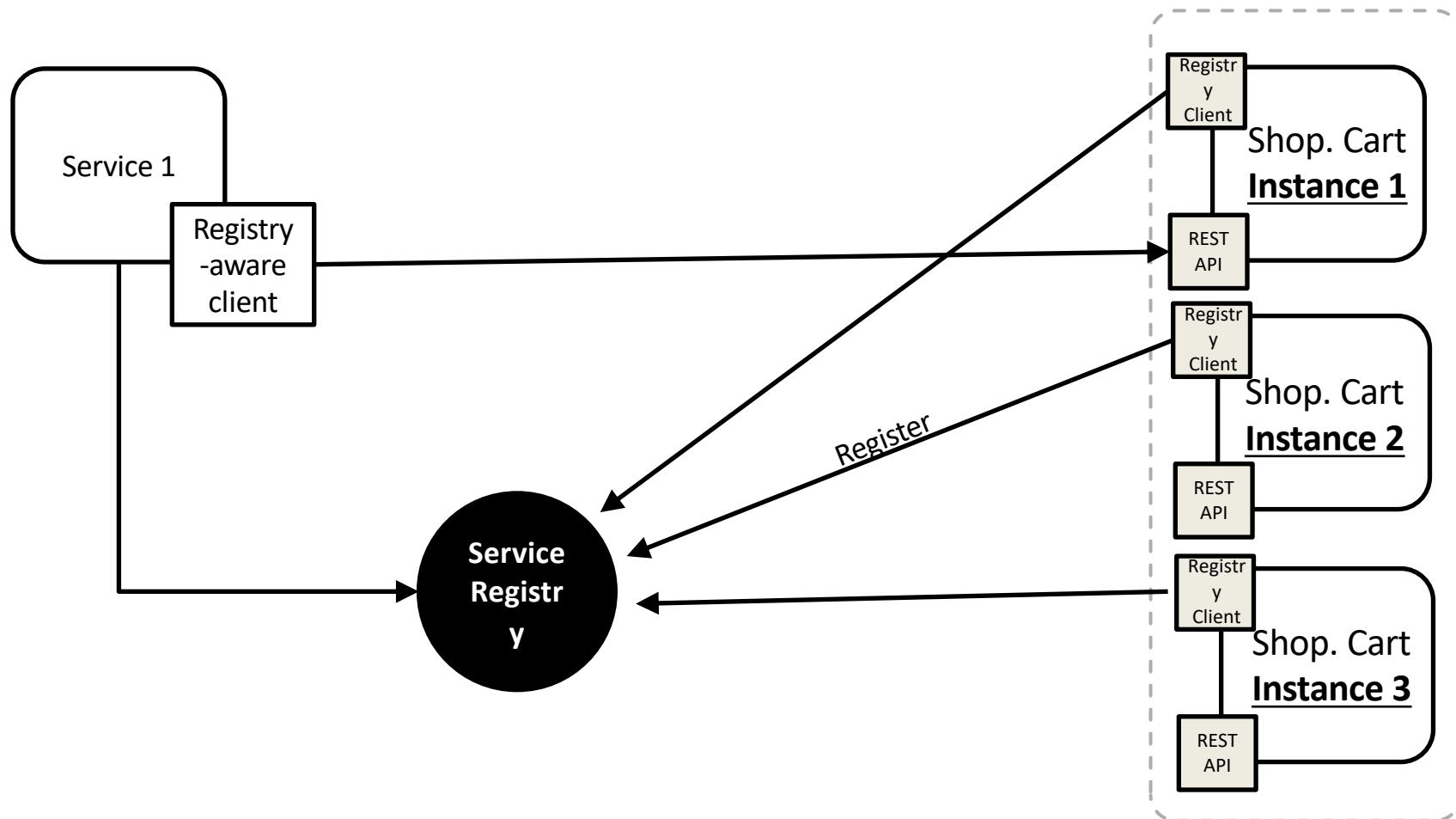
The API-Gateway does not provide support for publishing, promoting, or administering services at any significant level.

However, it is responsible for

the generation of customized APIs for each platform and
optimizing communications between the clients and the application,
encapsulating microservices details.

It allows microservices to evolve without influencing the clients.

Client-Side Discovery Pattern



Client-side Service Registry Pattern

Goal:

the client is responsible for mapping the available services instances to their network locations locations.

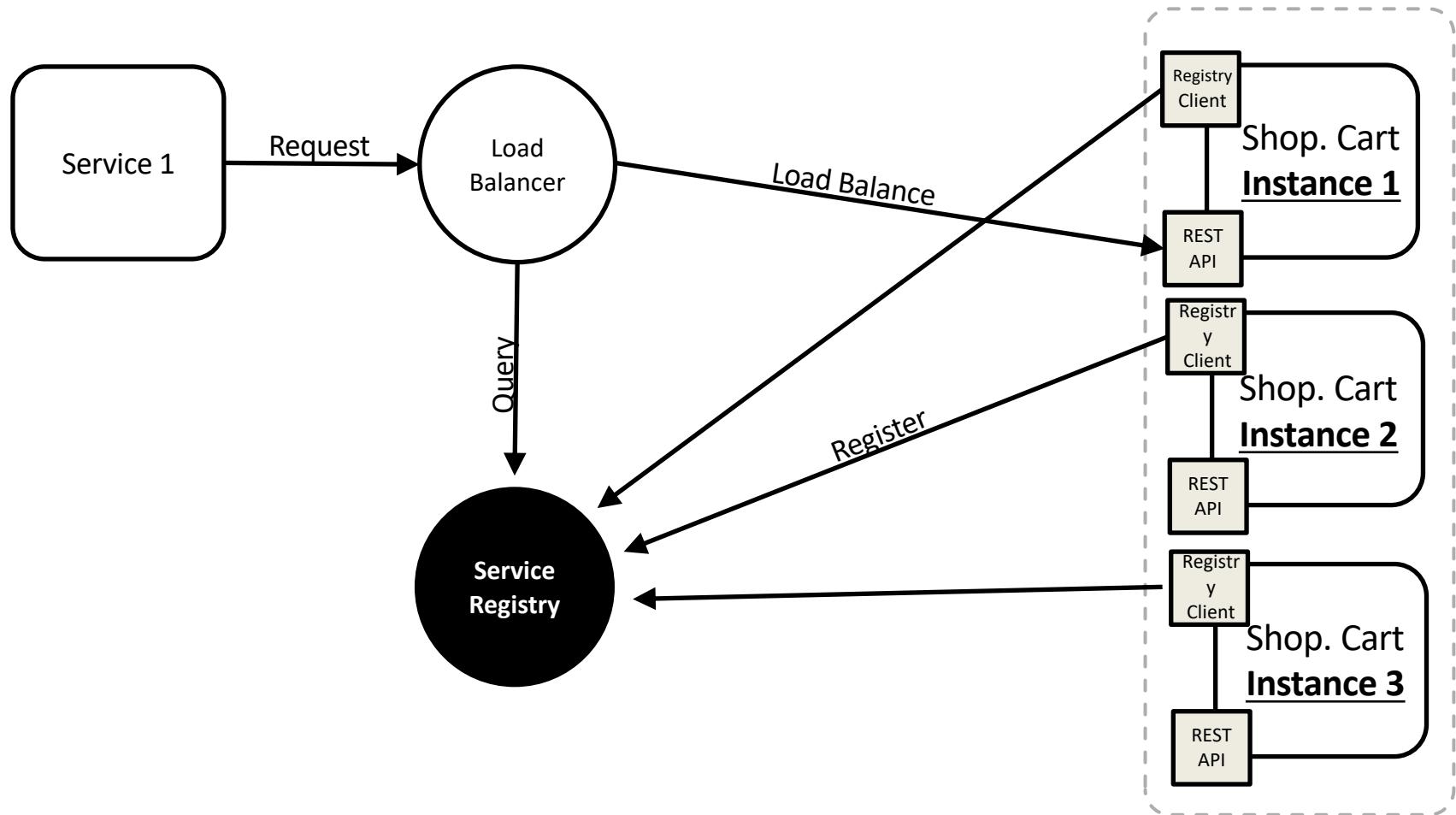
Moreover, the service discovery can also serve as load balancer of the requests. The client then uses a load-balancing algorithm to select one of the available service instances and makes a request.

Properties:

the main advantage of this pattern is connected to the ease of development. The clients are aware of service instance locations and therefore can connect directly to them without adding the development complexity of the server-side discovery.

The main reported disadvantage is the high coupling between the client and the service registry

Server-Side Discovery Pattern



Server-Side Discovery Pattern

Goal:

Unlike the API-Gateway pattern, this pattern allows clients and microservices to talk to each other directly.

It relies on a Service Registry, acting in a similar manner as a DNS server.

Properties:

The Service Registry knows the dynamic location of each microservice instance.

When a client requests access to a specific service, it first asks the registry for the service location;

The registry contacts the microservice to ensure its availability and forwards the location to the calling client.

Finally, unlike in the API-Gateway, clients communicate directly with the required services and access all available APIs exposed by the service, without any filter or service interface translation.

Classification of Advantages and Disadvantages of the Identified Patterns

	Pattern	Advantages	Disadvantages
Orchestration & Coordination	General	<ul style="list-style-type: none"> -Increased maintainability -Can use different languages -Flexibility -Reuse -Physical isolation -Self-healing 	<ul style="list-style-type: none"> -Development/testing complexity -Implementation effort -Network-related issue
	API Gateway	<ul style="list-style-type: none"> -Extension easiness -Market-centric Architecture -Backward compatibility 	<ul style="list-style-type: none"> -Potential bottleneck -Development complexity -Scalability
	Service Registry	<ul style="list-style-type: none"> -Increased maintainability -Communic., developm., migration -Software understandability -Failure safety 	<ul style="list-style-type: none"> -Interface design must be fixed -Service registry complexity -Reuse -Distributed system complexity
	Hybrid	<ul style="list-style-type: none"> -Migration easiness -Learning curve 	-SOA/ESB integration issues
Deployment	Multiple service per host	<ul style="list-style-type: none"> -Scalability -Performance 	
	Single service per host	<ul style="list-style-type: none"> -Service isolation 	<ul style="list-style-type: none"> -Scalability -Performance
Data Storage	DB per Service	<ul style="list-style-type: none"> -Scalability -Independent development -Security mechanism 	<ul style="list-style-type: none"> -Data needs to be splitted -Data consistency
	DB Cluster	<ul style="list-style-type: none"> -Scalability -Implementation easiness 	<ul style="list-style-type: none"> -Increase complexity -Failure risks
	Shared DB server	<ul style="list-style-type: none"> -Migration easiness -Data consistency 	<ul style="list-style-type: none"> -Lack of data isolation -Scalability

Guiding Principles of a Microservices Architectural Style (Advantages)

Increased maintainability: the key characteristic of microservices-based implementations reported by all the papers.

Possibility to write code in different languages. Highlights the benefit of using different languages, in contrast to monolithic applications

Flexibility. teams can choose their own technology based on needs

Reuse. The maintenance of a single microservice will reflect on any connected project, reducing the effort overhead by applying the same changes to the same component used in different projects

Ease of Deployment. Each microservice can be deployed independently, without need to recompile and redeploy whole application

Physical Isolation. This is the key for scaling, thanks to the microservices architectural style

Guiding Principles of a Microservices Architectural Style (Advantages)

Self-Healing. Failed services can be easily restarted or replaced by previous safer versions

Application Complexity. Since the application is decomposed into several components, these are less complex and easier to manage

Design for Failure. Better support for continuous delivery of many small changes can help developers in changing one thing at a time

Observability. A microservices architecture helps to visualize the "health status" of every services in the system in order to quickly locate and respond to any problem that occurs.

Unlimited Application size. In monolithic applications the application size is limited by the hardware and by web container specifications, while with microservices we could build a system with, in theory, no size limits.

Disadvantages

Testing Complexity. More components and collaborations among them increases testing complexity

Implementation Effort. It is reported that implementing microservices requires more effort than monolithic systems

Network related issues. Since endpoints are connected via a network, the network should be reliable

Latency: network latency can increase the communication time among microservices

Bandwidth: as service communication often relies on a network, bandwidth during normal and high peak operation needs to be considered.

In cloud environments and other distributed systems settings, unreliability is given and failure is always possible.

Microservices can also fail independently of each other, rather than together on a single node.

User Authorization. The API exposed by the microservices need to be protected with a shared user-authentication mechanism, which is often much more complex to implement than monolithic solutions

Disadvantages

Complexity. In the case of applications with a relatively small number of users (hundreds or thousands), the monolith could be a faster approach to start and could possibly be refactored into a microservices-based architectural style once the user-base grows

Automation Requirement. The explosion of the number of services and relationships among them requires a way to automate things. DevOps could be a good solution for this issue

Increased Dependence. Microservices are meant to be decoupled, making it critical to preserve independence and independent deployability.

Development Complexity. The learning curve is not very steep, but requires an experienced developer, at least for setting-up the basic architecture when compared to monolithic systems

Applications

	Research Prototype / System	Validation-specific Implementations	Industrial Implementations
Websites	[S11], [S39]	[S15], [S24], [S26], [S31]	[S13], [S32]
Services & APIs	IOT Integration [S33]	[S9], [S10], [S14], [S16], [S23], [S37], [S36]	[S21], [S34]
Others	Enterprise Measurement [S4] IP Multimedia [S25]	Benchmark/Test [S35], [S41], [S42] Business Process Modeling [S12]	Mobile Dev Platform [S38] Deployment Platform [S30]

Emerging Issues

Comparison of SOA and Microservices. The differences have not been thoroughly investigated. There is a lack of comparison from different points of view (e.g., performance, development effort, maintenance).

Microservices Explosion. What happens once a growing system has thousands, or millions of microservices is still not clear. Will all the mentioned microservices qualities degrade?

Negative Results. In which contexts do microservices turn out to be counterproductive? Are there anti-patterns?

DevOps

Taking into account the continuous delivery process, the DevOps pipeline is **only partially covered by research work.**

Considering the idea of continuous architecting, there is a number of implementations that report success stories regarding how to architect, build, and code microservice-based systems, but there are no reports on how to continuously deliver and how to continuously re-architect existing systems.

DevOps techniques: the **operation side, monitoring, deployment, and testing techniques are the most investigated steps** of the DevOps pipeline.

Only few papers propose specific techniques, and apply them to small example projects.

Release-specific techniques have not been investigated in our selected works. No empirical validation have been carried out in the selected works.

Summary

Overall, a catalogue of patterns emerges with patterns
for **orchestration/coordination** and **storage**

as classical structure-oriented architectural pattern} groups that address component-level interaction and data management concerns, resp., as traditional software architecture concerns,

for **deployment** alternatives

that link microservices to their deployment in the form of containers or VMs, linking microservices inherently to their deployment strategies in host environments.

Microservice Antipatterns and Bad Smells

FOCUS: MICROSERVICES

On the Definition of Microservice Bad Smells

Davide Taibi and Valentina Lenarduzzi, Tampere University of Technology

// To identify microservice-specific bad smells, researchers collected evidence of bad practices by interviewing developers experienced with microservice-based systems. They then classified the bad practices into 11 microservice bad smells frequently considered harmful by practitioners. //



MICROSERVICES ARE CURRENTLY enjoying increasing popularity and independence from monolithic architectures, being adopted by several big players such as Amazon, LinkedIn, Netflix, and SoundCloud. Microservices are relatively new, and while the microservices that work together are modeled around a business capability, and have a single and clearly defined purpose, they are often deployed independently, allowing small teams to work on separated and focused services by using the most suitable technologies for their job that can be deployed and scaled independently.

Several patterns and platforms such as nginx (www.nginx.org) and Kubernetes (kubernetes.io) exist on top of microservices. During the migration process, practitioners often come across problems, which are due mainly to their lack of knowledge regarding bad practices.

In this article, we provide a catalog of bad smells that are specific to systems developed using a microservice architectural style, together with possible solutions to overcome these smells. In practice this catalog was collected by interviewing 72 experienced developers over the course of two years, focusing on bad practices they found during the development of microservice-based systems and on how they overcame them. We identified a catalog of 11 microservice-specific bad smells by applying an open and selective process to derive the smell catalog from the practitioners' feedback.

The goal of this work is to help practitioners avoid these bad practices altogether or deal with them more efficiently when developing or migrating monolithic systems to microservice-based systems.

As with code and architectural smells, which are patterns commonly found in systems of bad design,^{1,2} we define microservice-specific bad smells (hereafter "microservice smells") as instances of smells such as undesired patterns, antipatterns, or bad practices—that negatively affect software quality attributes such as understandability, reusability, maintainability, and scalability of the system under development.

Background

Several generic architectural-smell detection tools and practices have been defined in the past years.^{3–7} Moreover, several microservice-specific architectural patterns have been defined.^{8–10} However, to the best of our knowledge, no previous work and, in particular, no empirical studies have proposed bad practices, antipatterns, or smells specifically concerning microservices.

However, some practitioners have started to discuss bad practices in microservices. In his ebook *Microservices AntiPatterns and Pitfalls*,

56 IEEE SOFTWARE | PUBLISHED BY THE IEEE COMPUTER SOCIETY 0460-7857/18/\$33.00 © 2018 IEEE

IEEE Software May/June 2018

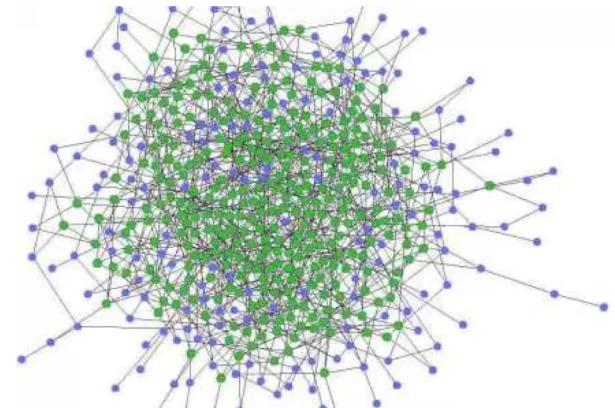
INFORTE - Tampere 02.09.2019

Microservices Architectural Smells

Exploratory Survey (72 practitioners) [2]

11 Architectural Smells identified

- Cyclic Dependencies
- Problem of Microservices Explosion
 - More than 50 connected microservices become unmanageable



Microservice Bad Smells

Microservice smell	Description (Desc.) / Detection (Det.)	Problem it may cause (P) / Adopted solutions (S)
API Versioning	Desc.: APIs are not semantically versioned. Det.: A lack of semantically consistent versions of APIs (e.g., v1.1, 1.2, etc.) Also proposed as Static Contract Pitfall. ^{11,12}	P: In the case of new versions of non-semantically-versioned APIs, API consumers may face connection issues. For example, the returning data might be different or might need to be called differently. S: APIs need to be semantically versioned to allow services to know whether they are communicating with the right version of the service or whether they need to adapt their communication to a new contract.
Cyclic Dependency	Desc.: A cyclic chain of calls between microservices exists. Det.: The existence of cycles of calls between microservices; e.g., A calls B, B calls C, and C calls back A.	P: Microservices involved in a cyclic dependency can be hard to maintain or reuse in isolation. S: Refine the cycles according to their shape, ⁴ and apply the API Gateway pattern. ²
ESB Usage	Desc./Det.: The microservices communicate via an enterprise service bus (ESB). An ESB is used for connecting microservices.	P: An ESB adds complexities for registering and deregistering services on it. S: Adopt a lightweight message bus instead of the ESB.
Hard-Coded Endpoints	Desc./Det.: Hardcoded IP addresses and ports of the services between connected microservices exist. Also proposed as Hardcoded IPs and Ports. ¹²	P: Microservices connected with hardcoded endpoints lead to problems when their locations need to be changed. S: Adopt a service discovery approach.
Inappropriate Service Intimacy	Desc.: The microservice keeps on connecting to private data from other services instead of dealing with its own data. Det.: A request for private data of other microservices. A direct connection to other microservices' databases.	P: Connecting to private data of other microservices increases coupling between microservices. The problem could be related to a mistake made while modeling the data. S: Consider merging the microservices.

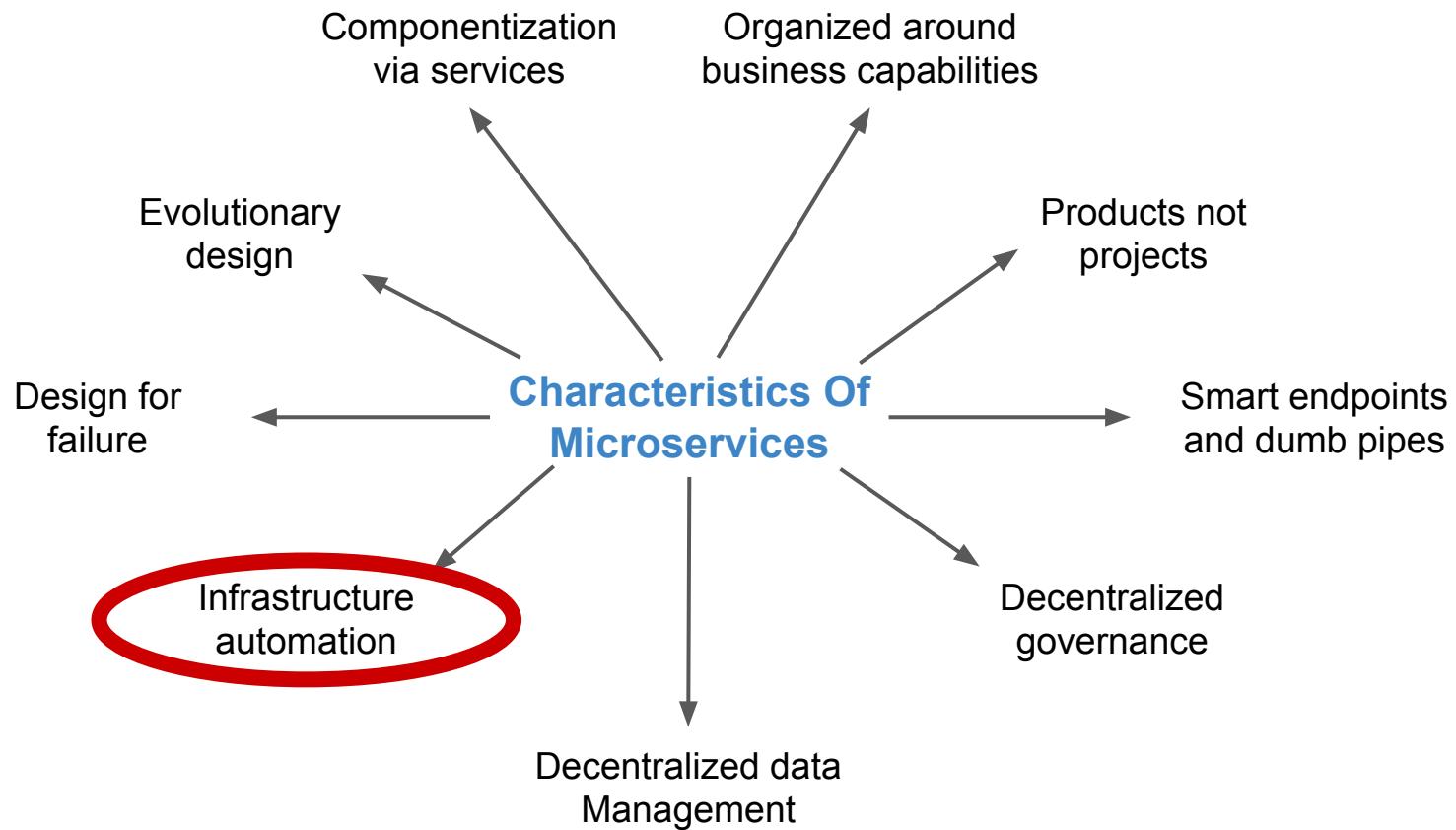
Microservice Bad Smells

Microservice Greedy	<p>Desc.: Teams tend to create new microservices for each feature, even when they are not needed. Common examples are microservices created to serve only one or two static HTML pages.</p> <p>Det.: Microservices with very limited functionalities (e.g., a microservice serving only one static HTML page).</p>	<p>P: This smell can generate an explosion of the number of microservices composing a system, resulting in a useless huge system that will easily become unmaintainable because of its size.</p> <p>S: Carefully consider whether the new microservice is needed.</p>
Not Having an API Gateway	<p>Desc.: Microservices communicate directly with each other. In the worst case, the service consumers also communicate directly with each microservice, increasing the complexity of the system and decreasing its ease of maintenance.</p> <p>Det.: Direct communication between microservices.</p>	<p>P: Our interviewees reported being able to work with systems consisting of 50 interconnected microservices. However, if the number was higher, they started facing communication and maintenance issues.</p> <p>S: Apply the API Gateway pattern² to reduce the communication complexity between microservices.</p>
Shared Libraries	<p>Desc./Det.: Shared libraries between different microservices are used.</p>	<p>P: Microservices are tightly coupled together, leading to a loss of independence between them. Moreover, teams need to coordinate with each other when they need to modify the shared library.</p> <p>S: Two possible solutions are to</p> <ol style="list-style-type: none">1. accept the redundancy to increase dependency among teams, or2. extract the library to a new shared service that can be deployed and developed independently by the connected microservices.
Shared Persistency	<p>Desc./Det.: Different microservices access the same relational database. In the worst case, different services access the same entities of the same relational database. Also proposed as Data Ownership.¹⁴</p>	<p>P: This smell highly couples the microservices connected to the same data, reducing team and service independence.</p> <p>S: Three possible solutions are to</p> <ol style="list-style-type: none">1. use independent databases for each service,2. use a shared database with a set of private tables for each service that can be accessed by only that service, or3. use a private database schema for each service.

Microservice Bad Smells

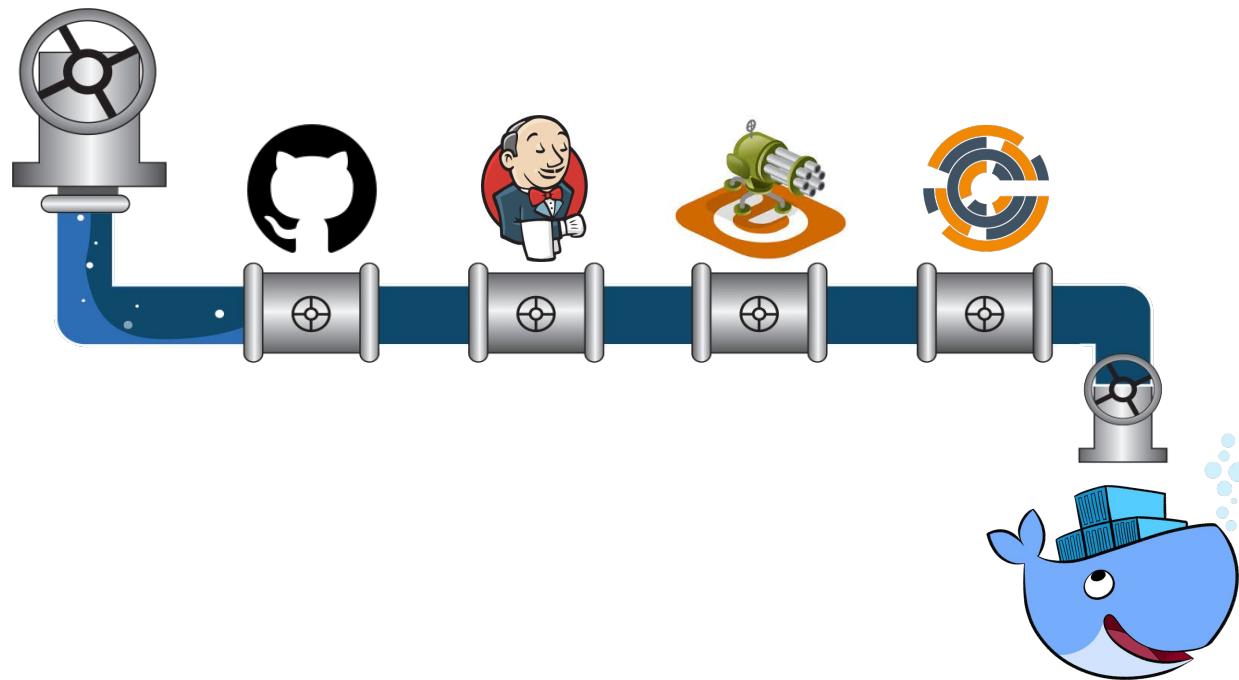
Too Many Standards	<p>Desc./Det.: Different development languages, protocols, frameworks, etc. are used. Also proposed as the Lust and Gluttony bad practices.¹⁶</p>	<p>P: Although microservices allow the use of different technologies, adopting too many different technologies can be a problem in companies, especially in the event of developer turnover.</p> <p>S: Carefully consider the adoption of different standards for different microservices, without following the latest hype.</p>
Wrong Cuts	<p>Desc.: Microservices are split on the basis of technical layers (presentation, business, and data layers) instead of business capabilities.</p>	<p>P: The wrong separation of concerns and increased data-splitting complexity can occur.</p> <p>S: Perform a clear analysis of business processes and the need for resources.</p>

Microservices Characteristics



Infrastructure Automation

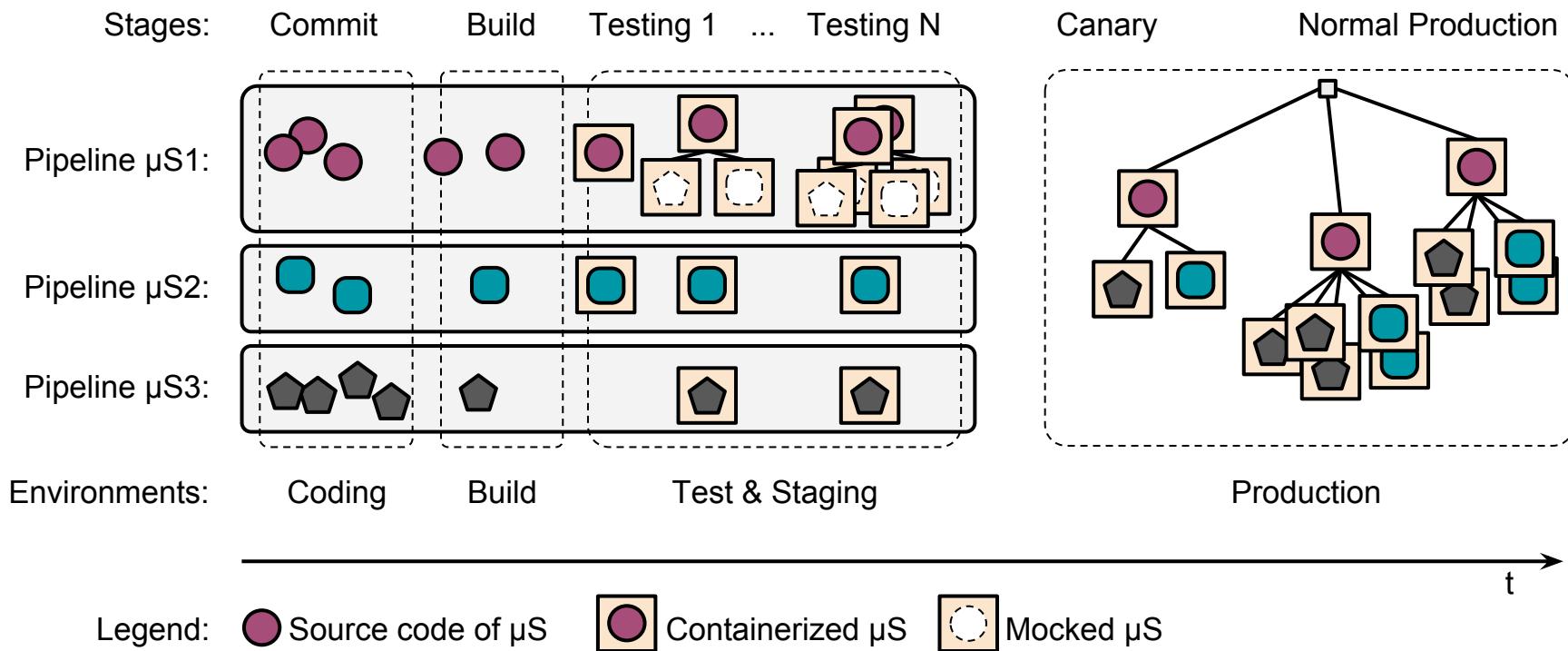
Enables the creation of **continuous delivery** (CD) microservices pipelines based on **DevOps** and other **agile practices**



source: <https://www.neudescic.com/services/continuous-delivery-devops/> (adapted)

Infrastructure Automation (2)

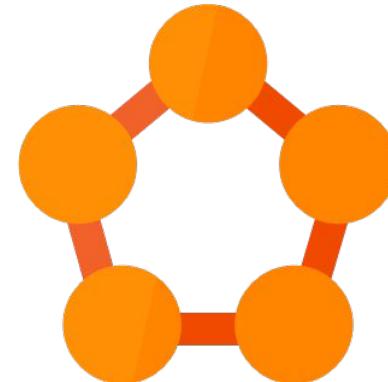
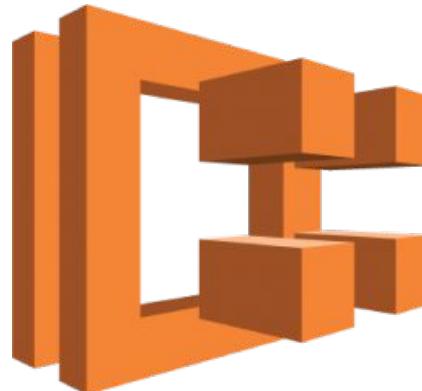
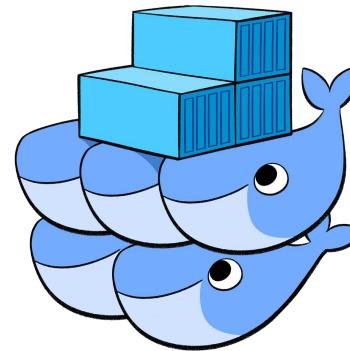
A typical CD strategy for microservices would look like this:



source: Heinrich et al., *Performance Engineering for Microservices: Research Challenges and Directions*. In ICPE'17 Companion, 2017

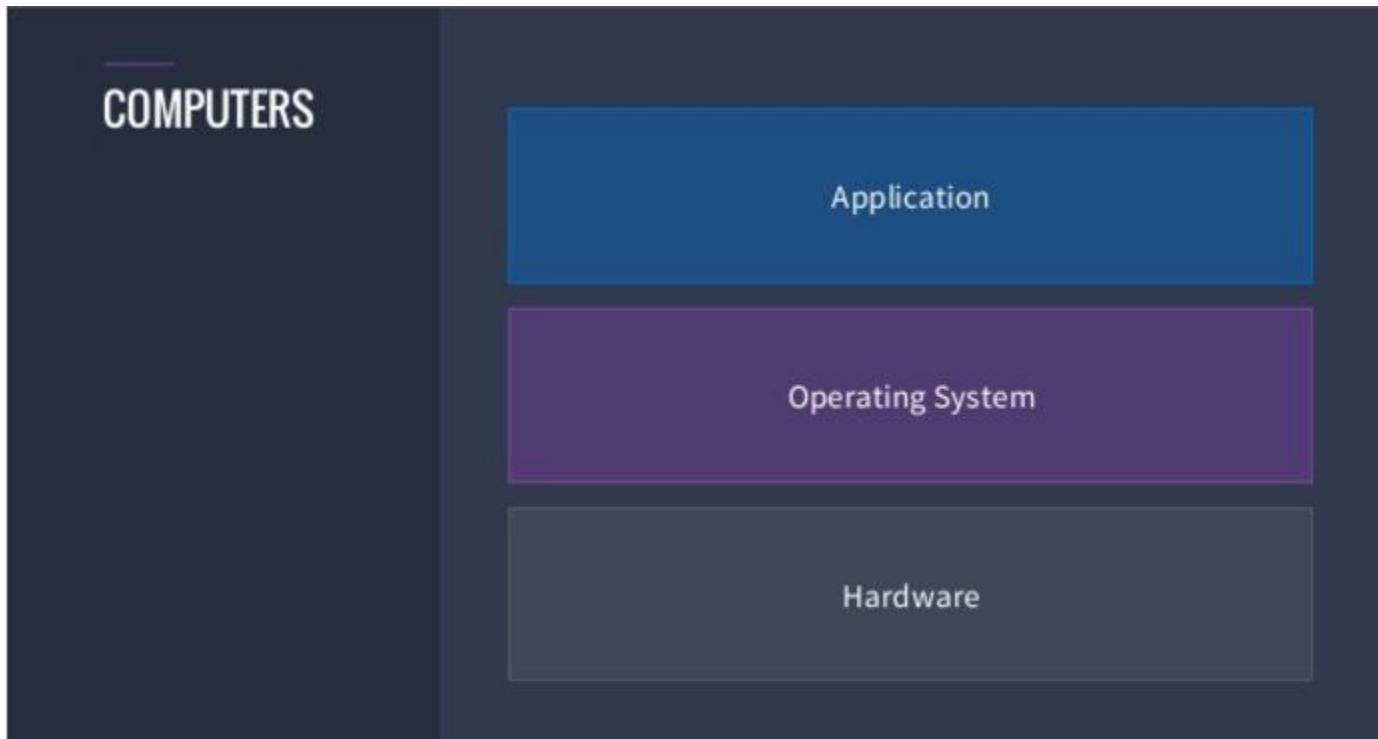
Infrastructure Automation (3)

CD automation often involves the use of **lightweight containers** (e.g., Docker) and **container orchestration tools** (e.g, Kubernetes, Docker Swarm, Mesos/Marathon, AWS ECS, Azure Service Fabric)



Container Orchestration

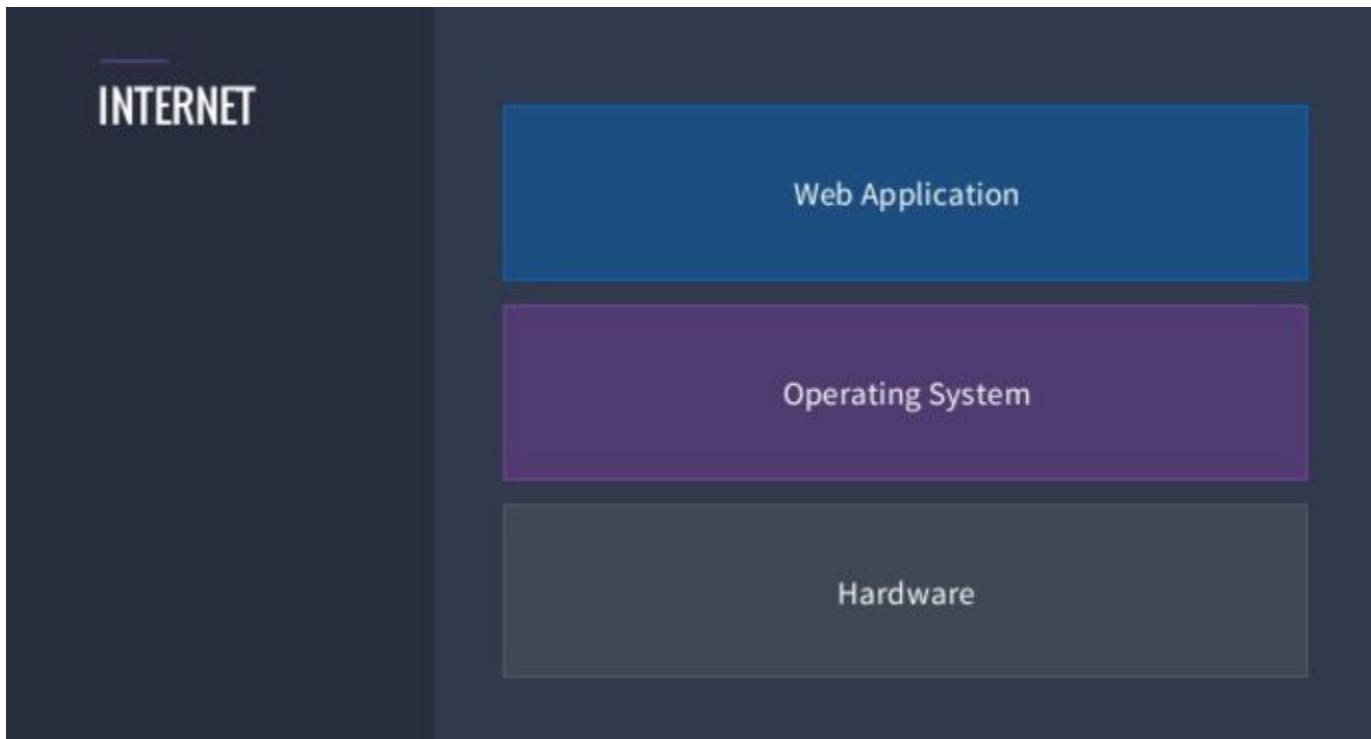
Historical perspective



source: <https://www.slideshare.net/Karllsenberg/container-orchestration-wars>

Container Orchestration (2)

Historical perspective



source: <https://www.slideshare.net/Karllsenberg/container-orchestration-wars>

Container Orchestration (3)

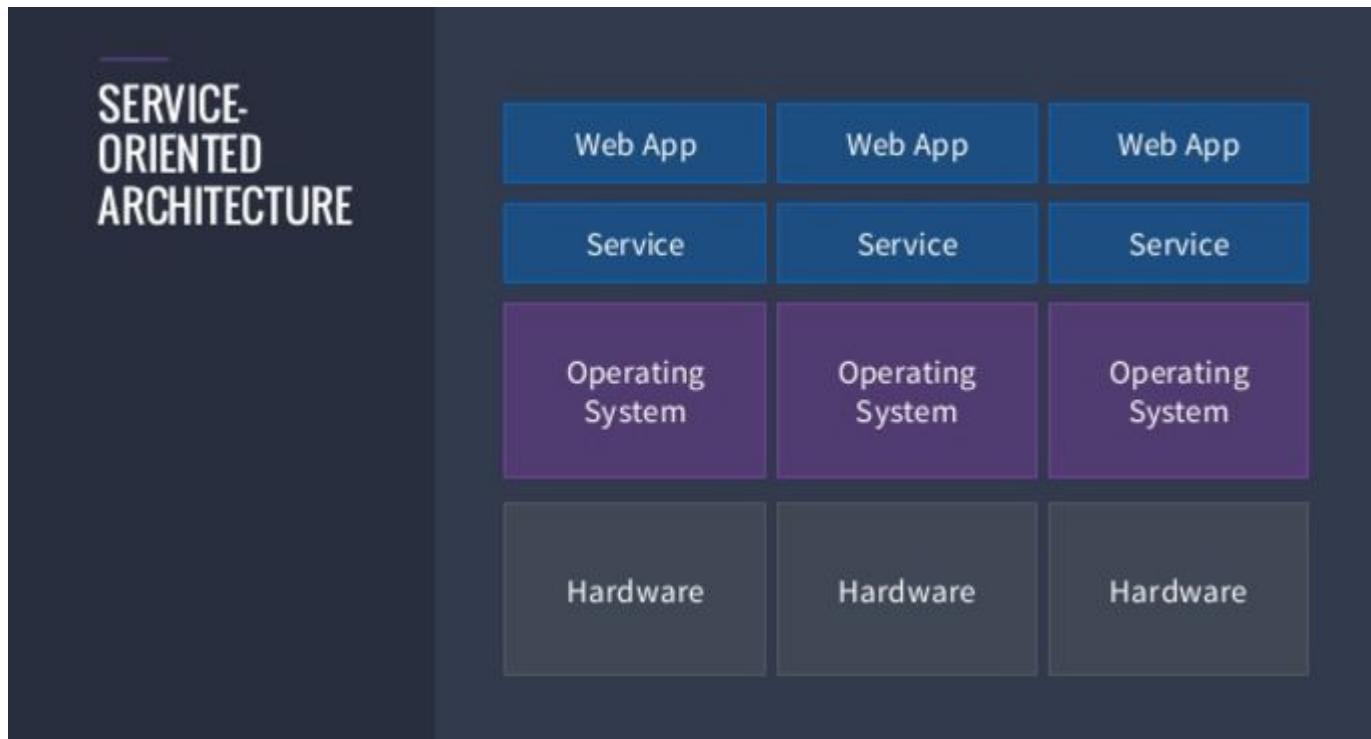
Historical perspective



source: <https://www.slideshare.net/Karllsenberg/container-orchestration-wars>

Container Orchestration (4)

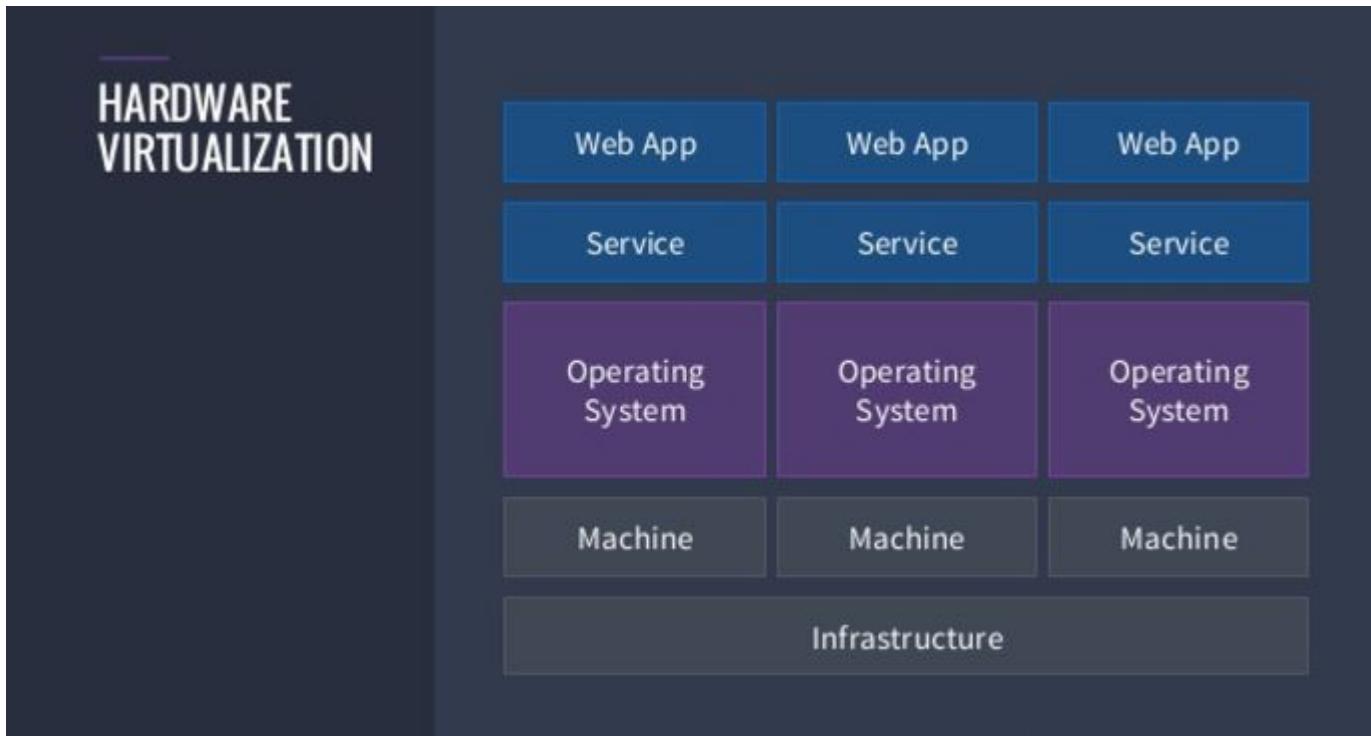
Historical perspective



source: <https://www.slideshare.net/Karllsenberg/container-orchestration-wars>

Container Orchestration (5)

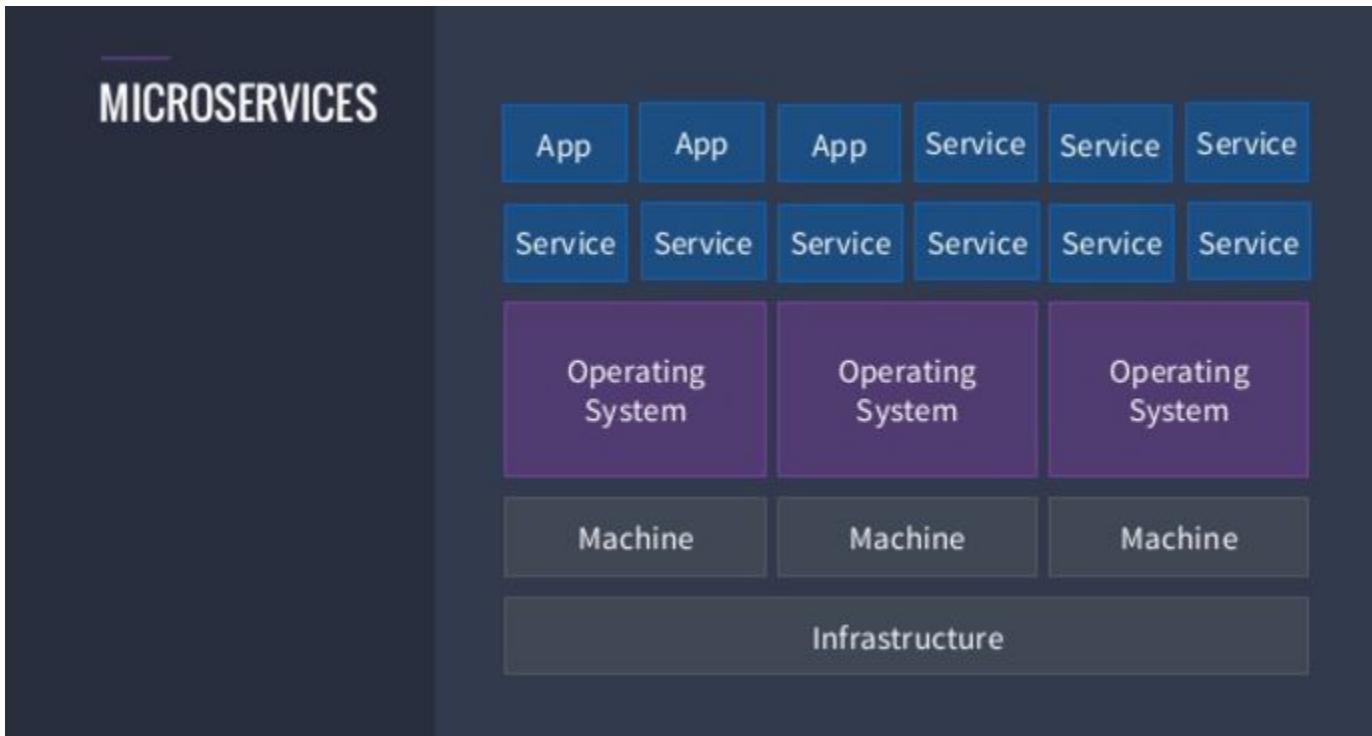
Historical perspective



source: <https://www.slideshare.net/Karllsenberg/container-orchestration-wars>

Container Orchestration (6)

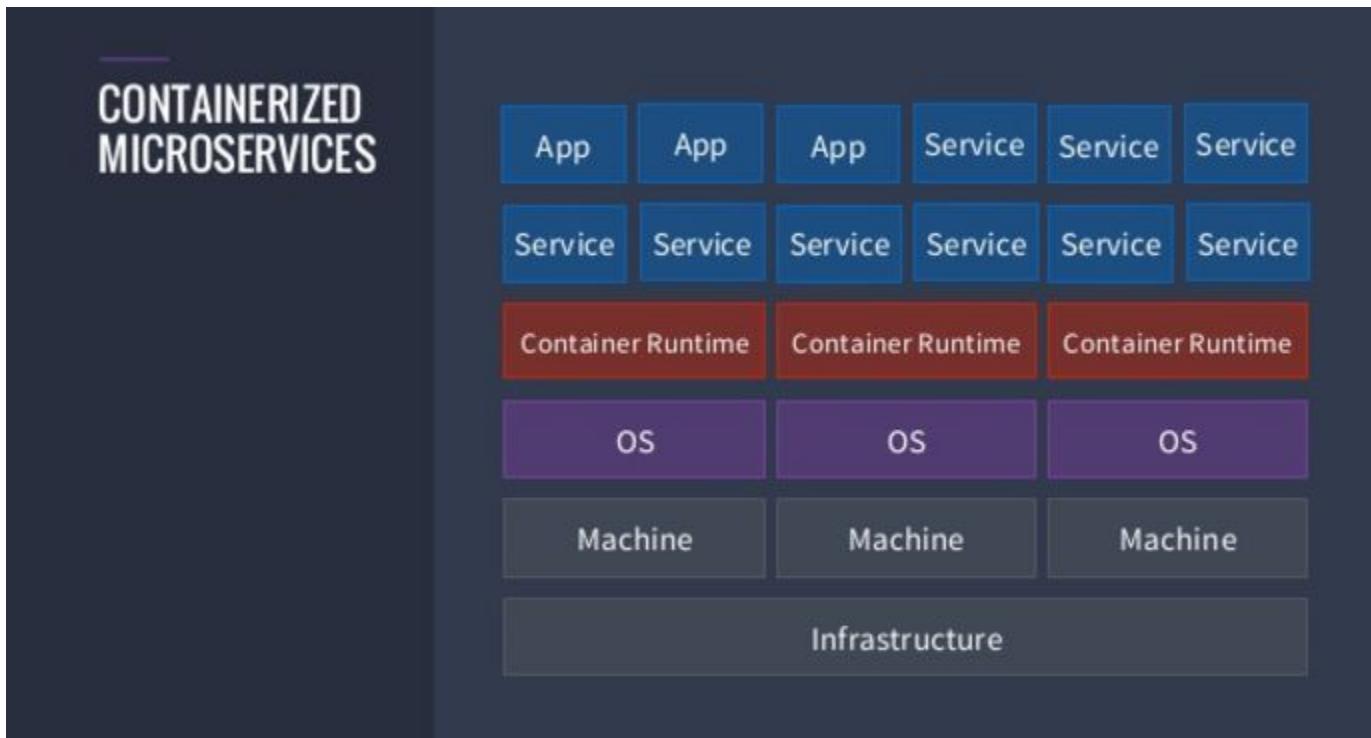
Historical perspective



source: <https://www.slideshare.net/Karllsenberg/container-orchestration-wars>

Container Orchestration (7)

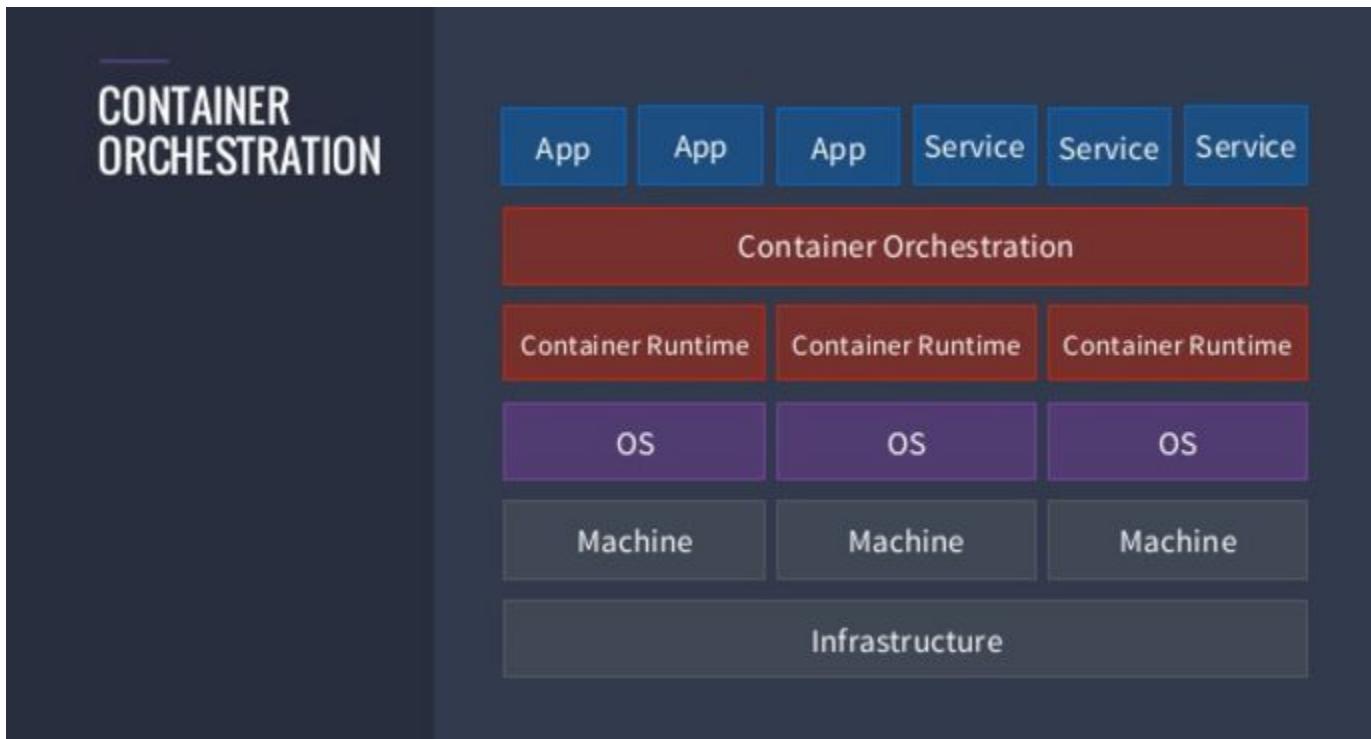
Historical perspective



source: <https://www.slideshare.net/Karllsenberg/container-orchestration-wars>

Container Orchestration (8)

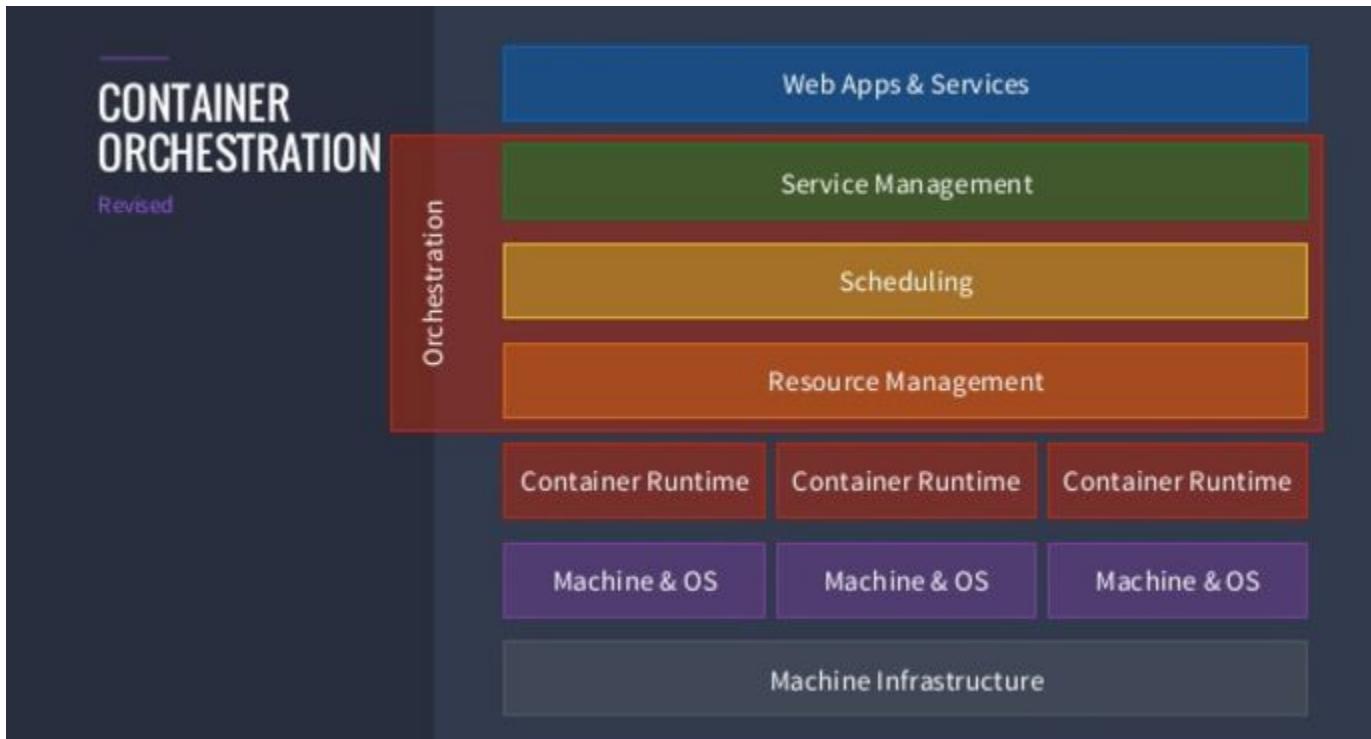
Historical perspective



source: <https://www.slideshare.net/Karllsenberg/container-orchestration-wars>

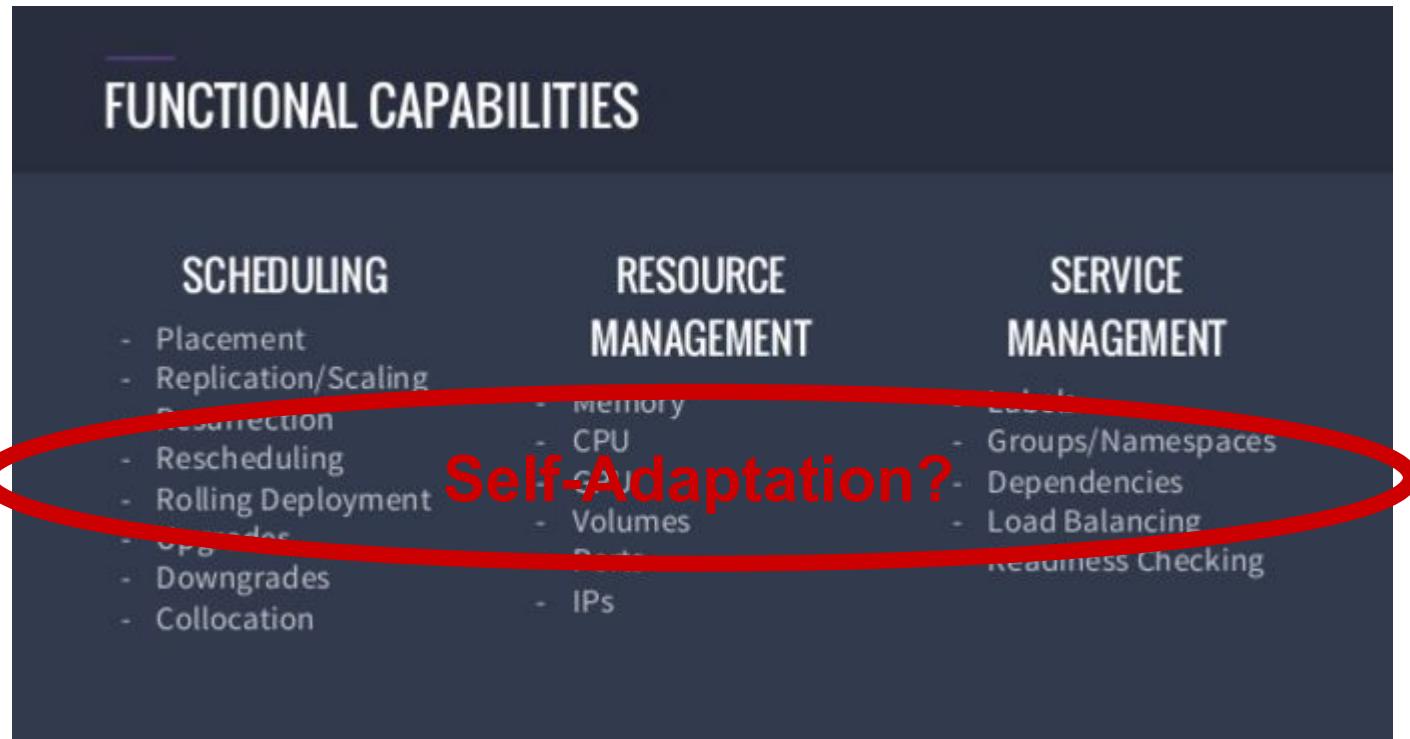
Container Orchestration (9)

Historical perspective



source: <https://www.slideshare.net/Karllsenberg/container-orchestration-wars>

Container Orchestration (10)



source: <https://www.slideshare.net/Karllsenberg/container-orchestration-wars>

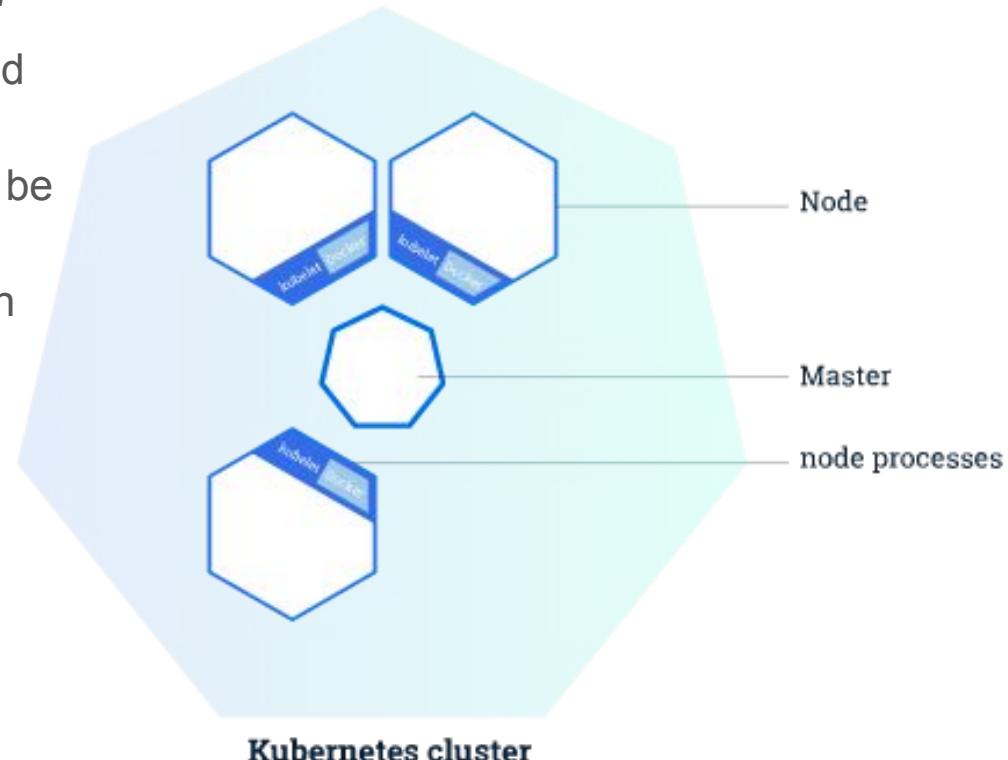
Container Orchestration (11)



source: <https://www.slideshare.net/Karllsenberg/container-orchestration-wars>

Example: Kubernetes

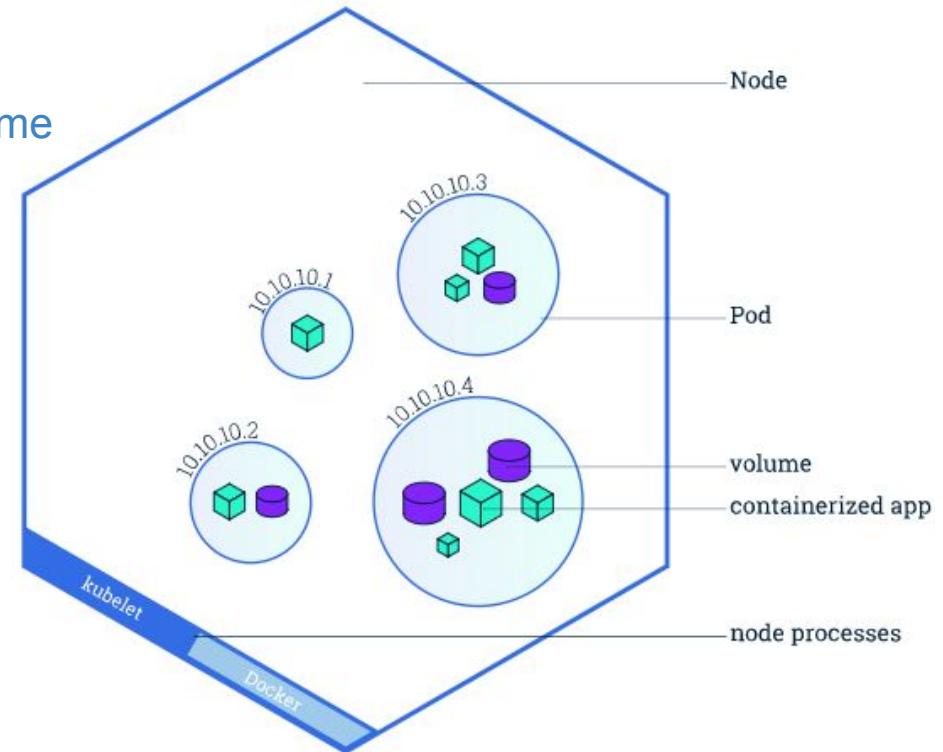
- A Kubernetes **cluster** is composed of a (replicated) **master** and multiple **nodes**
- The **master** coordinates all activities in the cluster, including scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates
- A **node** is a worker machine and may be either a **virtual** or a **physical machine**, depending on the cluster configuration



source: <https://kubernetes.io/docs/tutorials/kubernetes-basics/>

Example: Kubernetes (2)

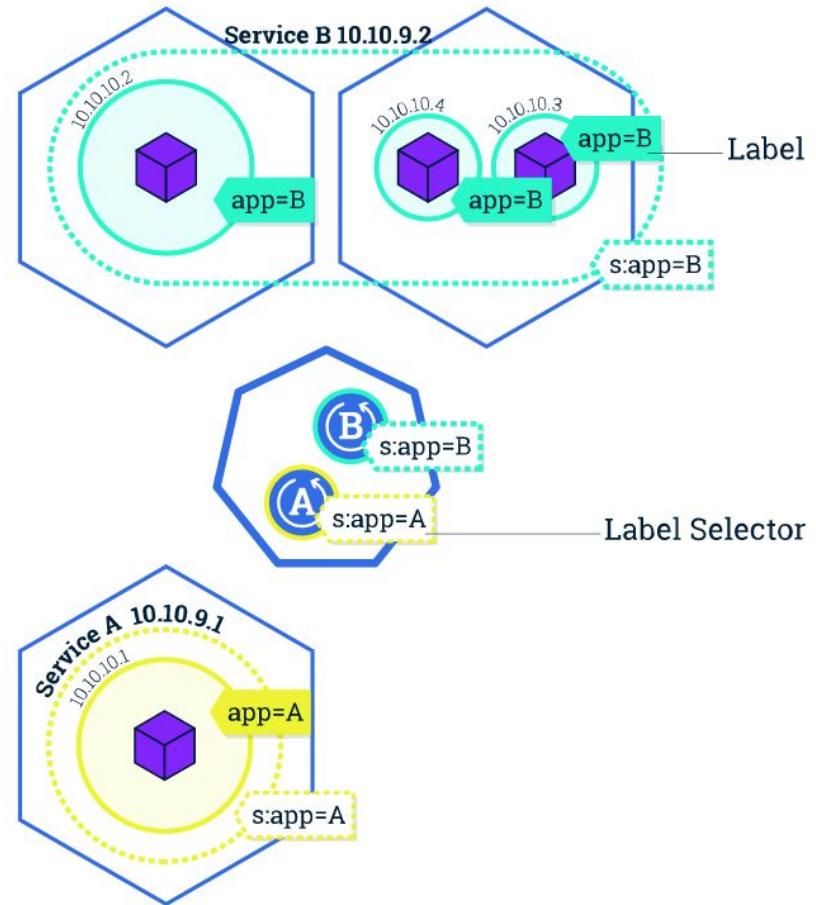
- A node can have multiple **pods**, which are automatically scheduled across the nodes in the cluster by the master
- A **pod** is a Kubernetes abstraction that represents a group of one or more **application containers** that are guaranteed to be **co-located on the same node** and can share resources (e.g, IP address, disk volumes, container image)



source: <https://kubernetes.io/docs/tutorials/kubernetes-basics/>

Example: Kubernetes (3)

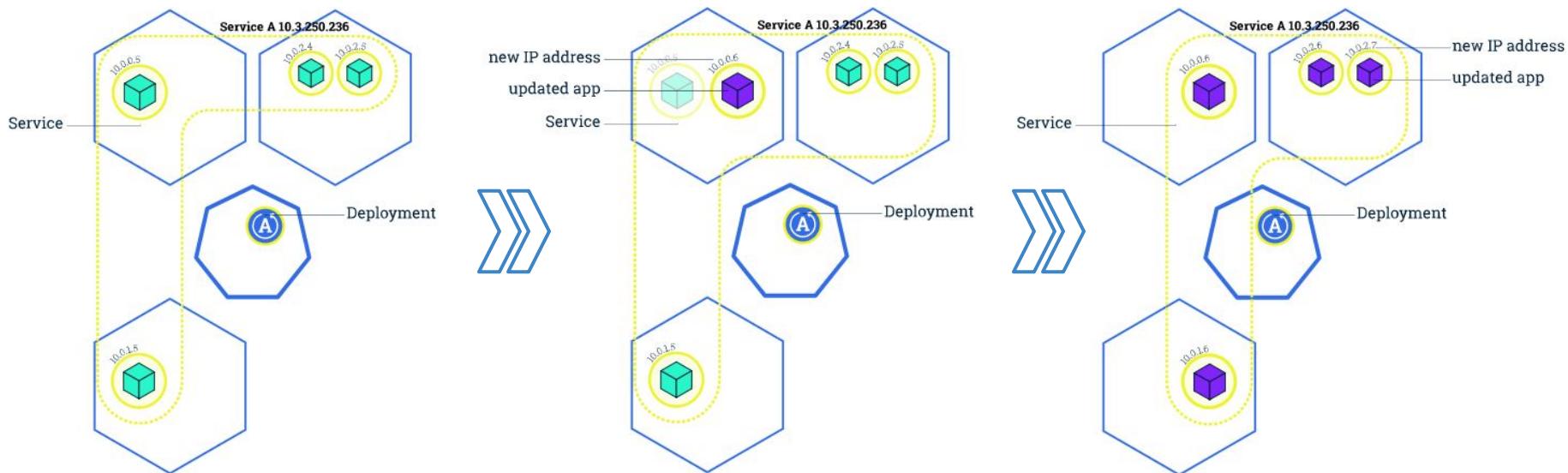
- A **service** is a set of pods that work together, such as one tier of a multi-tier application
- A **deployment** is special type of controller used to maintain a desired state for a service or a given **set of pods**
- Nodes and pods can be logically organized by assigning one or more **labels** to them. This allows Kubernetes users to map their own organizational structures (e.g., versions, environments, regions) onto systems objects in a loosely coupled fashion, without explicitly referring to their individual properties (e.g., IP address, Docker image)
- Labels can also be used to specify **scheduling constraints** (e.g., a pod should/should not be allocated to a node with a specific label)



source: <https://kubernetes.io/docs/tutorials/kubernetes-basics/>

Example: Kubernetes (4)

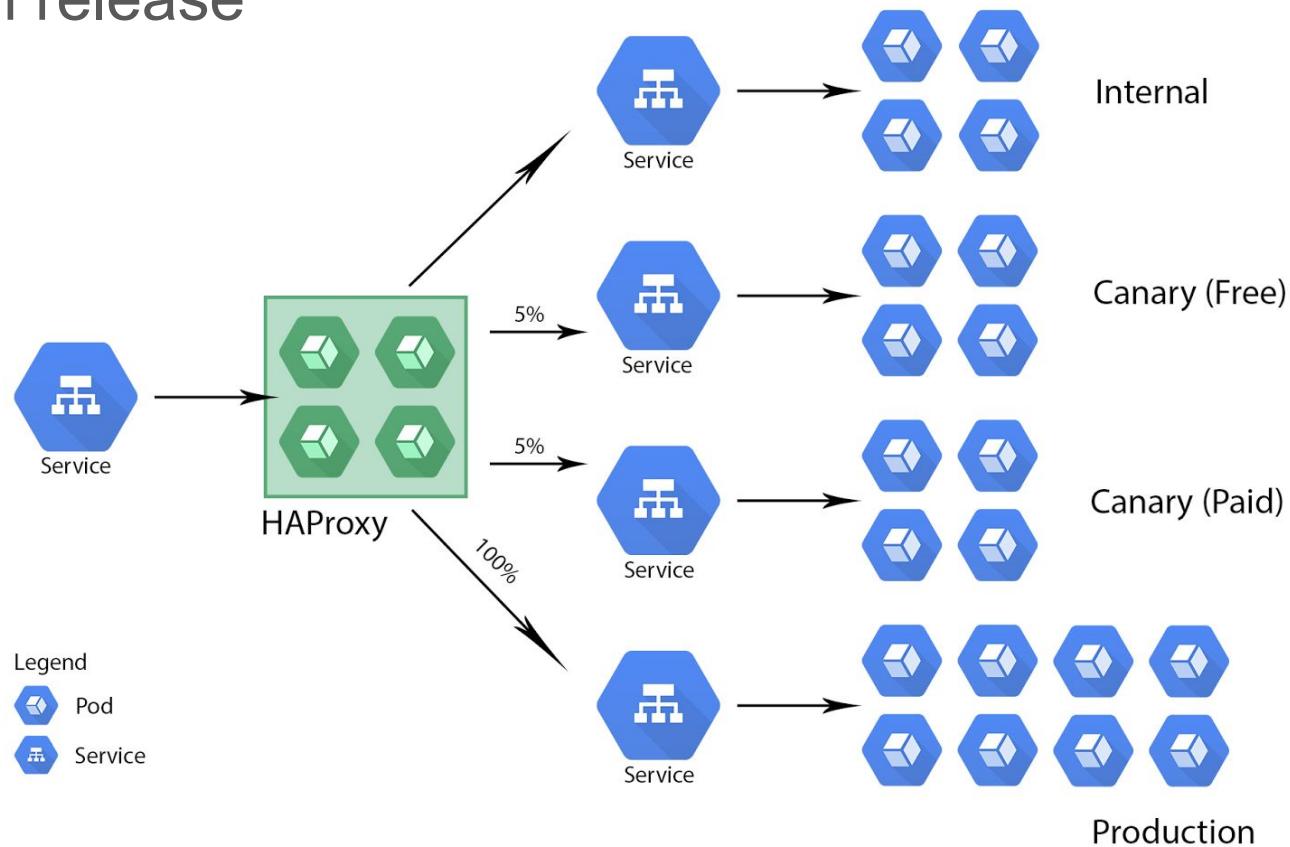
- A **rolling update** (or **rollout**) allow a deployment update to take place with zero downtime by incrementally replacing pod instances with new (updated) ones
- Kubernetes's default **rollout algorithm** looks like this:
 1. Create a new replication controller with the updated configuration
 2. Increase/decrease the replica count on the new and old controllers until the correct number of replicas is reached
 3. Delete the original replication controller
- Kubernetes keeps the **rollout history** of every deployment, so that they can be **rolled back** to any previous known configuration at **anytime**



source: <https://kubernetes.io/docs/tutorials/kubernetes-basics/>

Example: Kubernetes (5)

One can create a **canary deployment** with Kubernetes by creating multiple deployments for the same application, one for each release



Example: Kubernetes (6)

The Kubernetes API can be extended in two ways:

- By defining [custom resources](#), which are resources not originally available in every Kubernetes installation (e.g., customized controllers)
- By defining [API server aggregations](#), which are specialized (standalone) server implementations to handle specific custom resources. The main API server delegates requests for custom resources to the customized servers, making them available to all of its clients

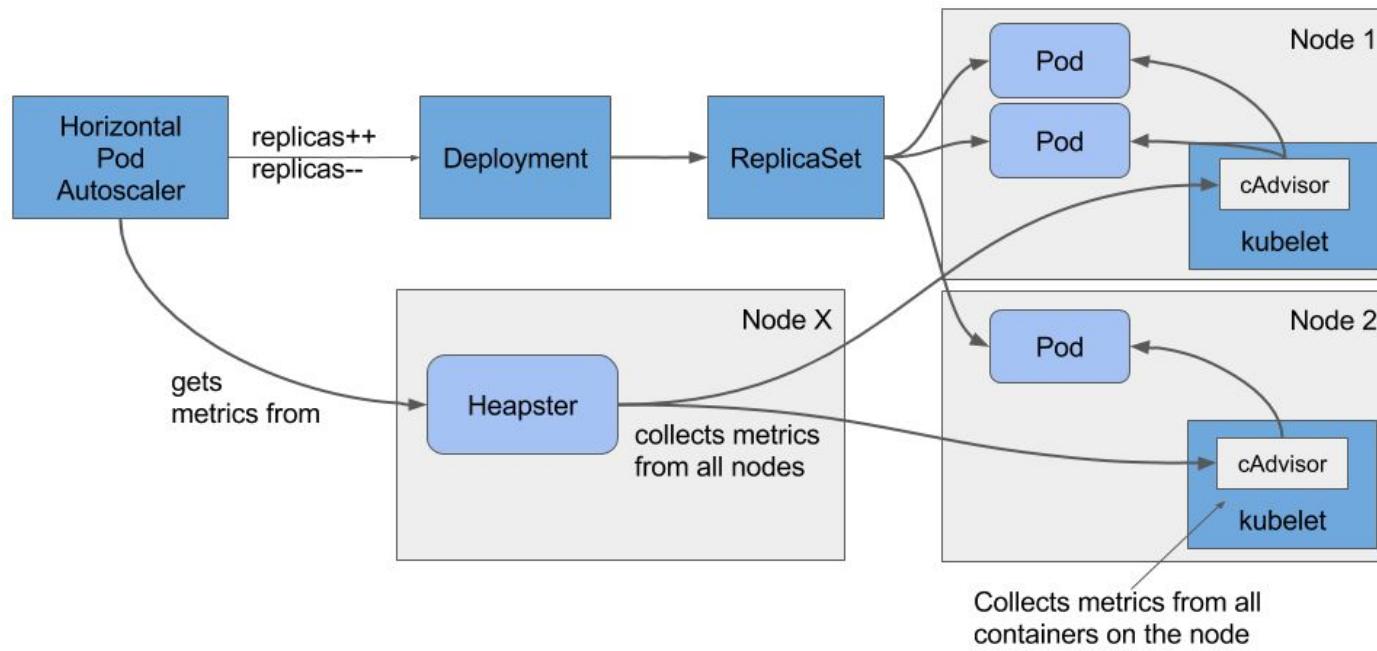
Some controller extensions available in Kubernetes:

- **etcd** – provides a reliable, distributed [key-value store](#), used as the primary configuration datastore of Kubernetes itself
- **Prometheus** – creates, configures, and manages Prometheus [monitoring](#) instances
- **Horizontal Pod Autoscaler (HPA)** – provides an [auto-scaling service](#) for Kubernetes pods

Example: Kubernetes (7)

Horizontal Pod Autoscaler (HPA)

- Native (reactive) auto-scaling solution implemented as a Kubernetes customer controller
 - Uses [Heapster](#) and [cAdvisor](#) for compute resource usage analysis and monitoring
 - The HPA controller periodically adjusts the number of replicas in a deployment to match the observed per-pod resource metrics (customer metrics also supported) to the target specified by the HPA user

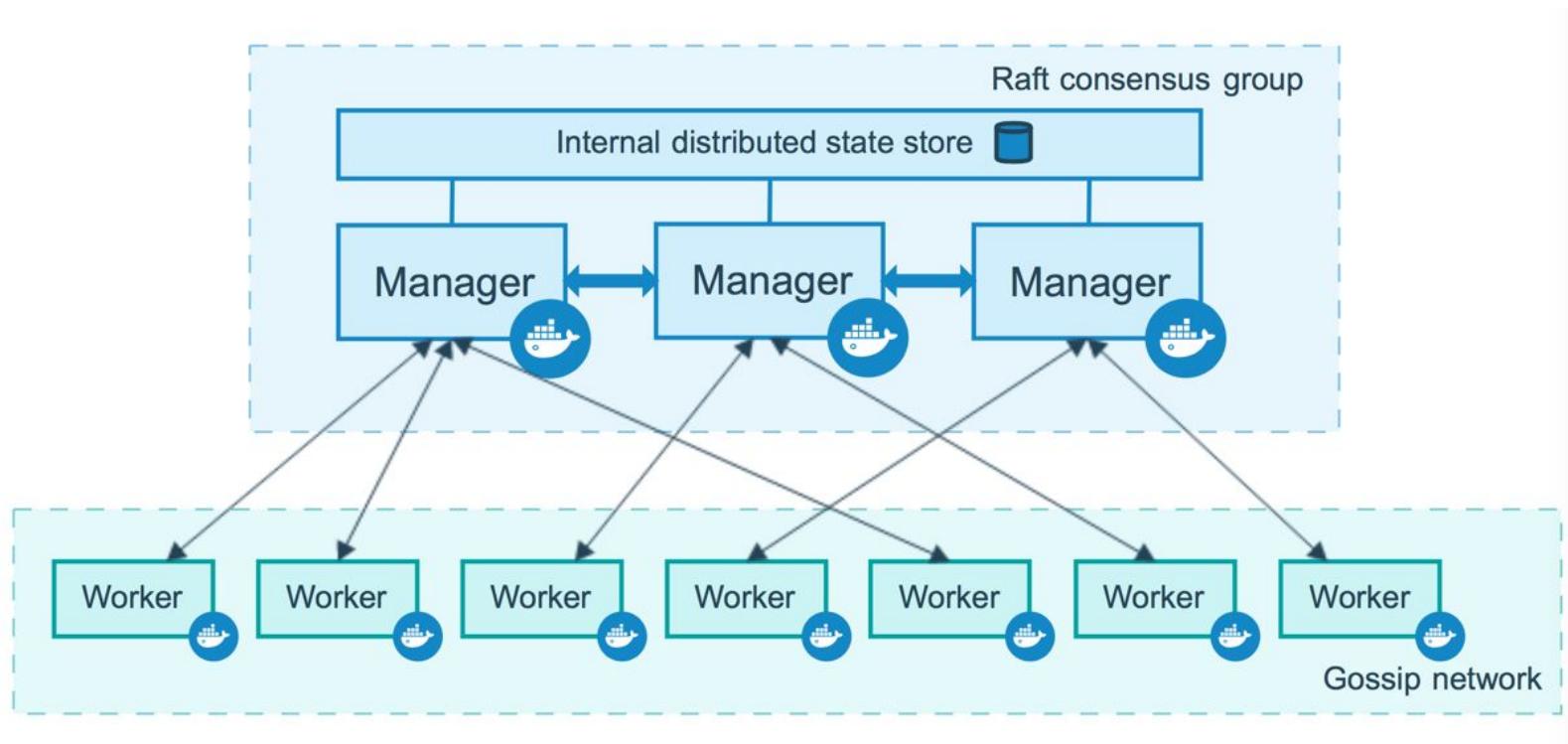


source: <https://medium.com/@marko.luksa/>

Example: Docker Swarm

Native container orchestration solution for Docker

- Embedded in the Docker engine ([swarm mode](#)) since version 1.12.0 (2016-07-28)

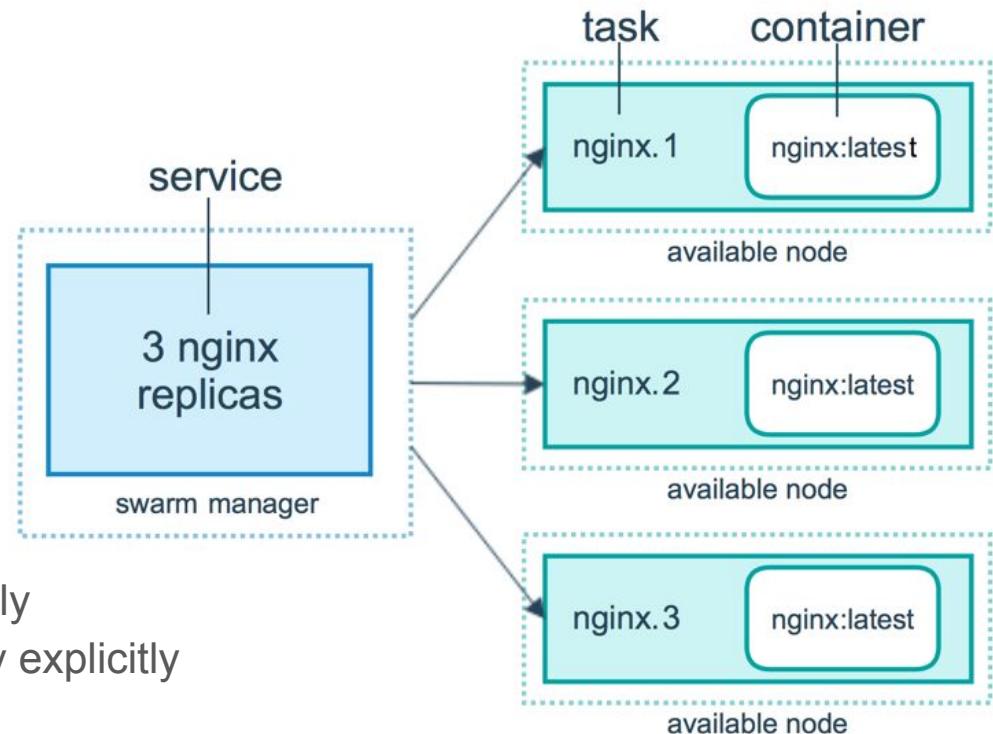


Source: <https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/>

Example: Docker Swarm (2)

A **service** is defined in terms of its Docker image, number of **tasks**, rolling update policy, overlay network, etc.

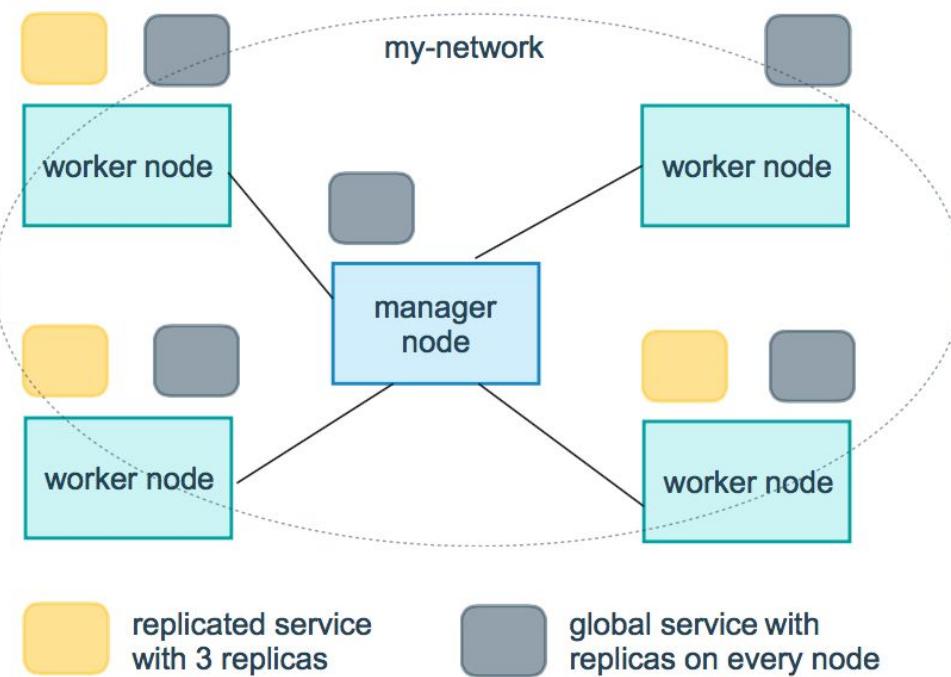
- The Swarm **manager** schedules each deployed service on **worker nodes** as one or more (**replicated**) **tasks**
- If a task fails, the manager automatically schedules a new task of the same Docker image to the same or to a different node
- As with Kubernetes, Docker Swarm also allows one to logically group and manage containers by explicitly assigning labels to them



Example: Docker Swarm (3)

A service can be defined as **global**, which means that it will have a **replicated task** on **every node**

- Each time a new node is added to the swarm, the manager creates a new task for each global service and schedules those tasks to the new node
- Commonly used for deploying **monitoring agents** and **security scanners**



Example: Docker Swarm (4)

Docker Swarm also supports **rolling updates** of services

Docker's default **rolling update algorithm** looks like this:

1. Stop the first task
2. Update the stopped task
3. Start the container for the updated task
4. If a task update is successful, wait for the specified delay period then update the next task
5. If, at any time during the update, a task fails, pause the update

The user can customize this algorithm by defining **rolling update policies**, which include the following options:

- the minimum time interval between updates
- the number of tasks that can be updated in parallel
- the action to be performed when a task update fails

Infrastructure Automation Summary

Current container orchestration tools offer some **useful yet limited** support for **self-adaptation**

- Restart in case of failures
- Rolling updates
- Metrics-based reactive auto-scaling

Possibility of customizing their native controllers to implement more sophisticated (e.g., **MAPE-K** like) adaptations

Best strategy for deploying self-adaptive microservices?

Credits

Some slides may include content from the following external sources:

- James Lewis. Microservices. XP2016 Workshops
- Nabor C Mendoca. Self-Adaptation Challenges for Cloud-Native Applications. 2017

davide.taibi@tut.fi

www.taibi.it

Microservices vs SOA

	Traditional SOA	Microservices
Messaging type	Smart, but dependency-laden ESB	Dumb, fast messaging (as with Apache Kafka)
Programming style	Imperative model	Reactive actor programming model that echoes agent-based systems
Lines of code per service	Hundreds or thousands of lines of code	100 or fewer lines of code
State	Stateful	Stateless
Messaging type	Synchronous: wait to connect	Asynchronous: publish and subscribe
Databases	Large relational databases	NoSQL or micro-SQL databases blended with conventional databases

[2] PWA Research

Microservices vs SOA

	Traditional SOA	Microservices
Code type	Procedural	Functional
Means of evolution	Each big service evolves	Each small service is immutable and can be abandoned or ignored
Means of systemic change	Modify the monolith	Create a new service
Means of scaling	Optimize the monolith	Add more powerful services and cluster by activity
System-level awareness	Less aware and event driven	More aware and event driven

[2] PWA Research