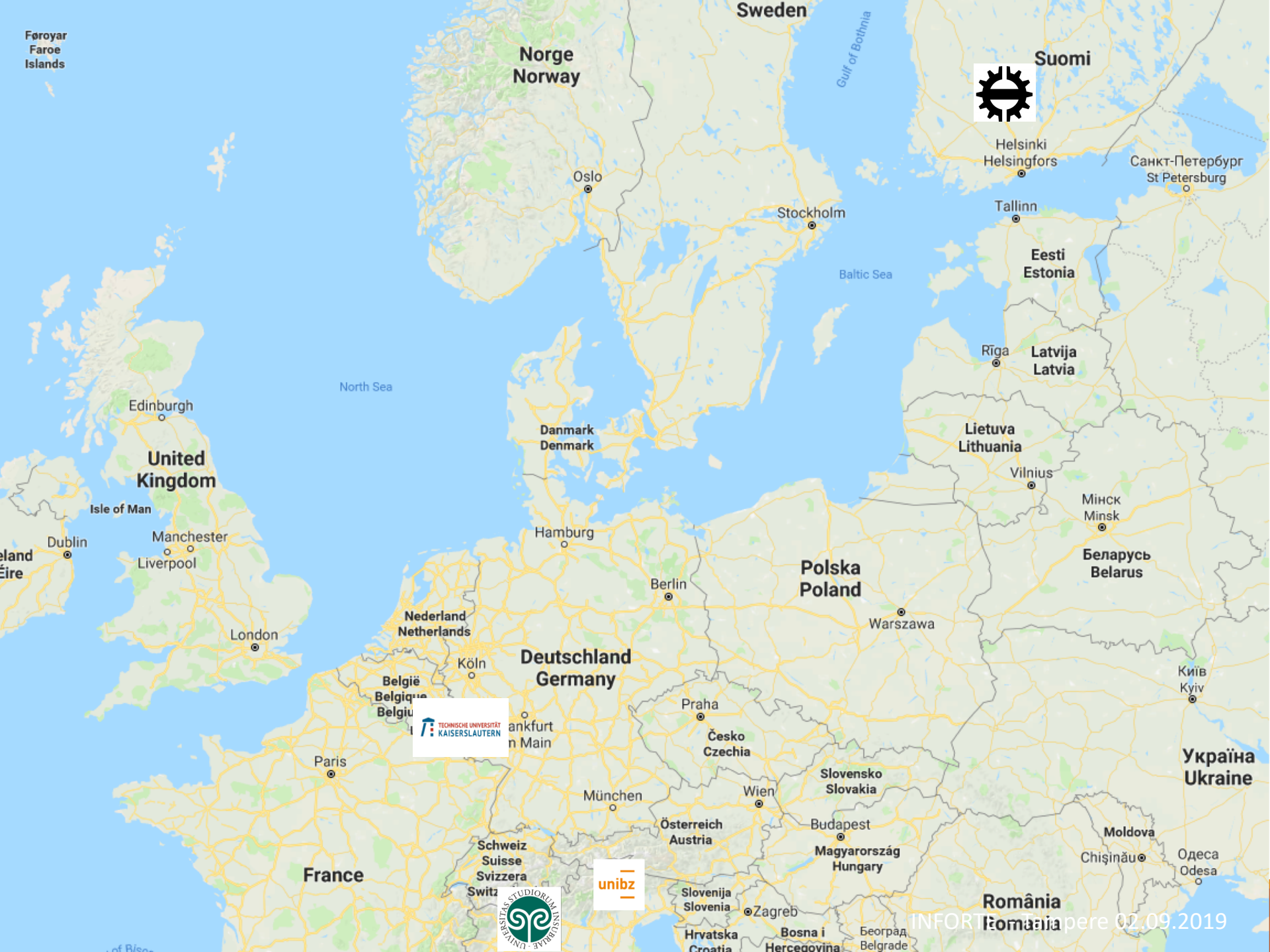
Cloud Native Introduction, and Patterns

INFORTE - Summer School on Software Evolution: From Monolithic to Cloud-Native

Davide Taibi
davide.taibi@tuni.fi
www.taibi.it



INFORTE Tampere 02.09.2019

Main Research Areas

- Cloud and Web software Engineering
  - Microservices

- Microservices Patterns
- Anti Patterns and Bad Smells
- Orchestration
- Kubernetes and Docker Swarm
- Conclusion

- Final Exam

Outline

- Introduction to Microservices
  - Why and how companies migrate to microservices
    - Motivations
    - Migration Process

- Technical Debt

Software Architecture Evolution

1990s and earlier | **2000s** | **2010s**

Coupling

Pre-SOA (monolithic)
Tight coupling

Traditional SOA
Looser coupling

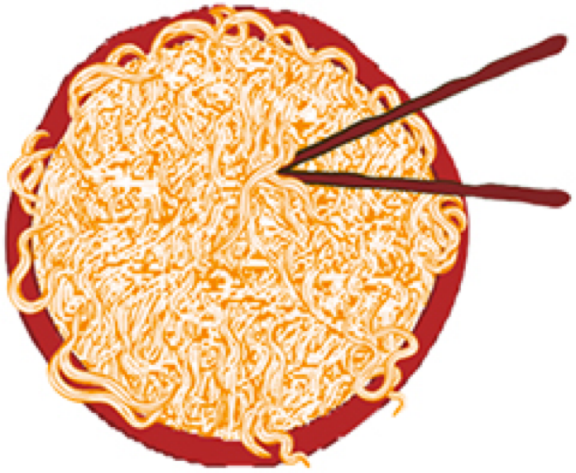
Microservices
Decoupled

D.Taibi, V.Lenarduzzi, C.Pahl, A.Janes. Microservices Architectural Styles: Agile or not Agile? XP 2017

Software Architecture Evolution

1990s and earlier

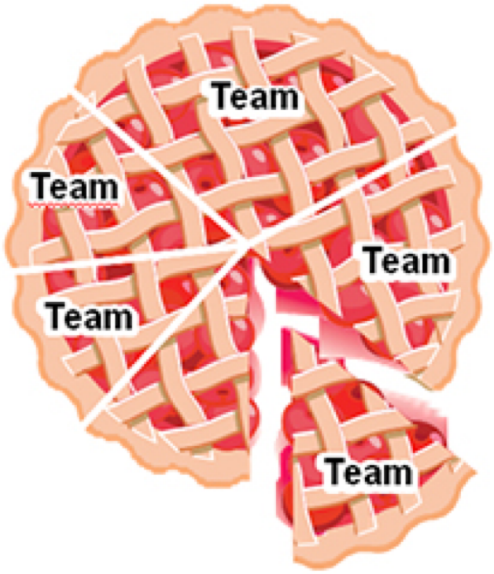
Pre-SOA (monolithic)
Tight coupling



For a monolith to change, all must agree on each change. Each change has unanticipated effects requiring careful testing beforehand.

2000s

Traditional SOA
Looser coupling



Elements in SOA are developed more autonomously but must be coordinated with others to fit into the overall design.

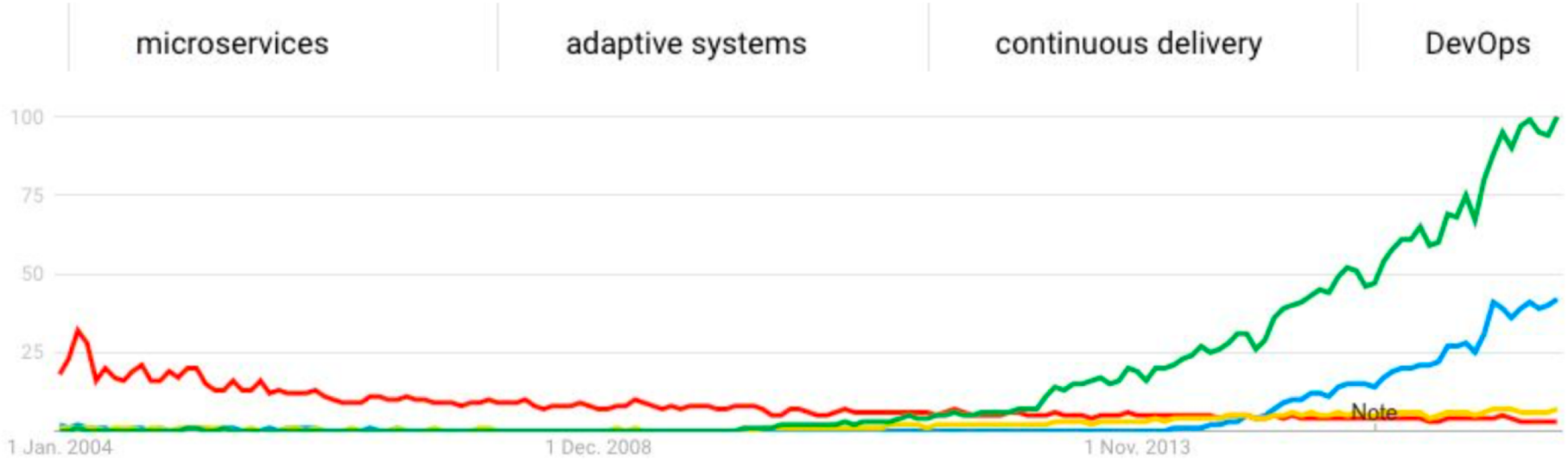
2010s

Microservices
Decoupled

Developers can create and activate new microservices without prior coordination with others. Their adherence to MSA principles makes continuous delivery of new or modified services possible.

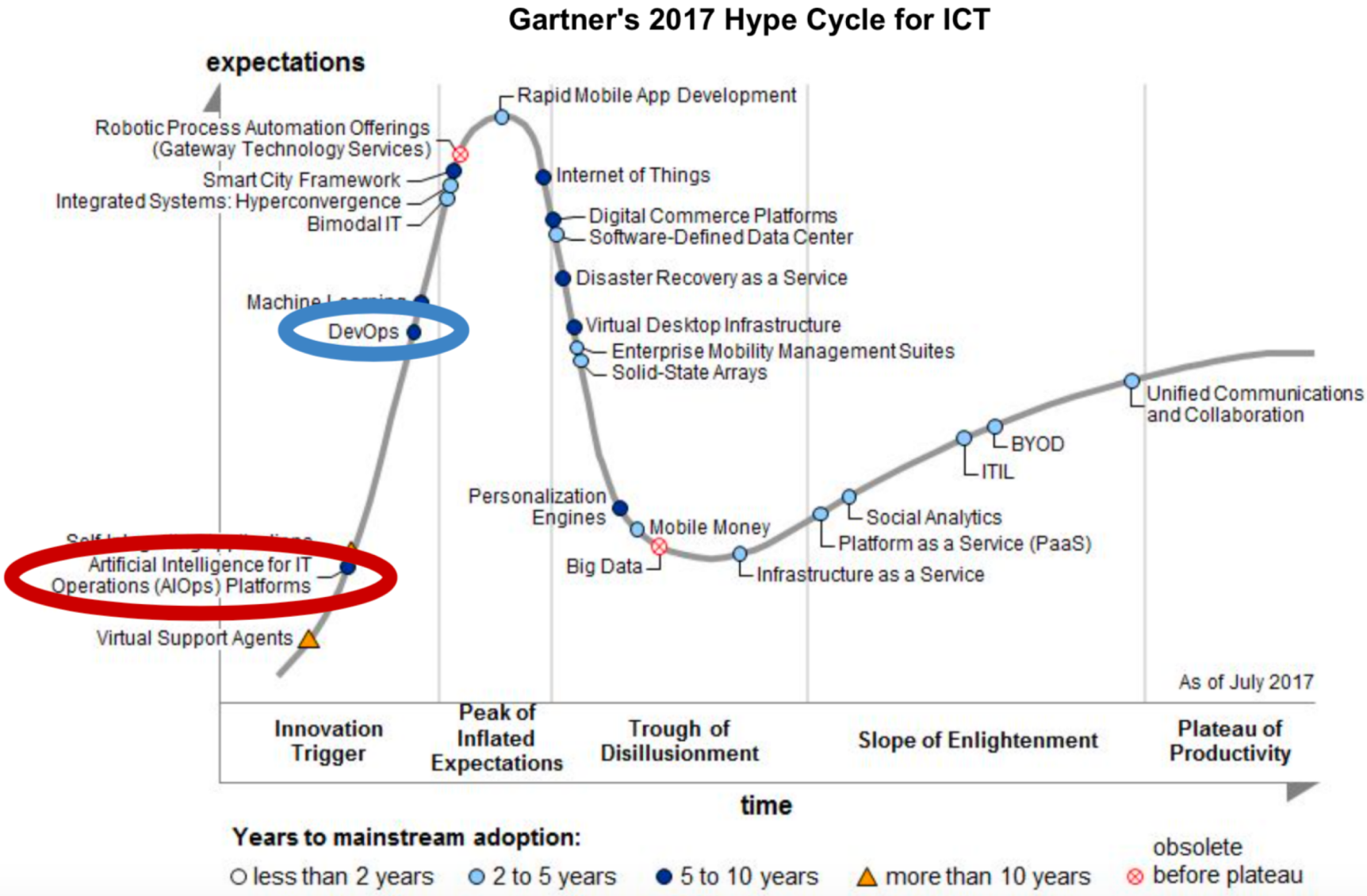
D.Taibi, V.Lenarduzzi, C.Pahl, A.Janes. Microservices Architectural Styles: Agile or not Agile? XP 2017

Why?



source: https://trends.google.co.in/trends/

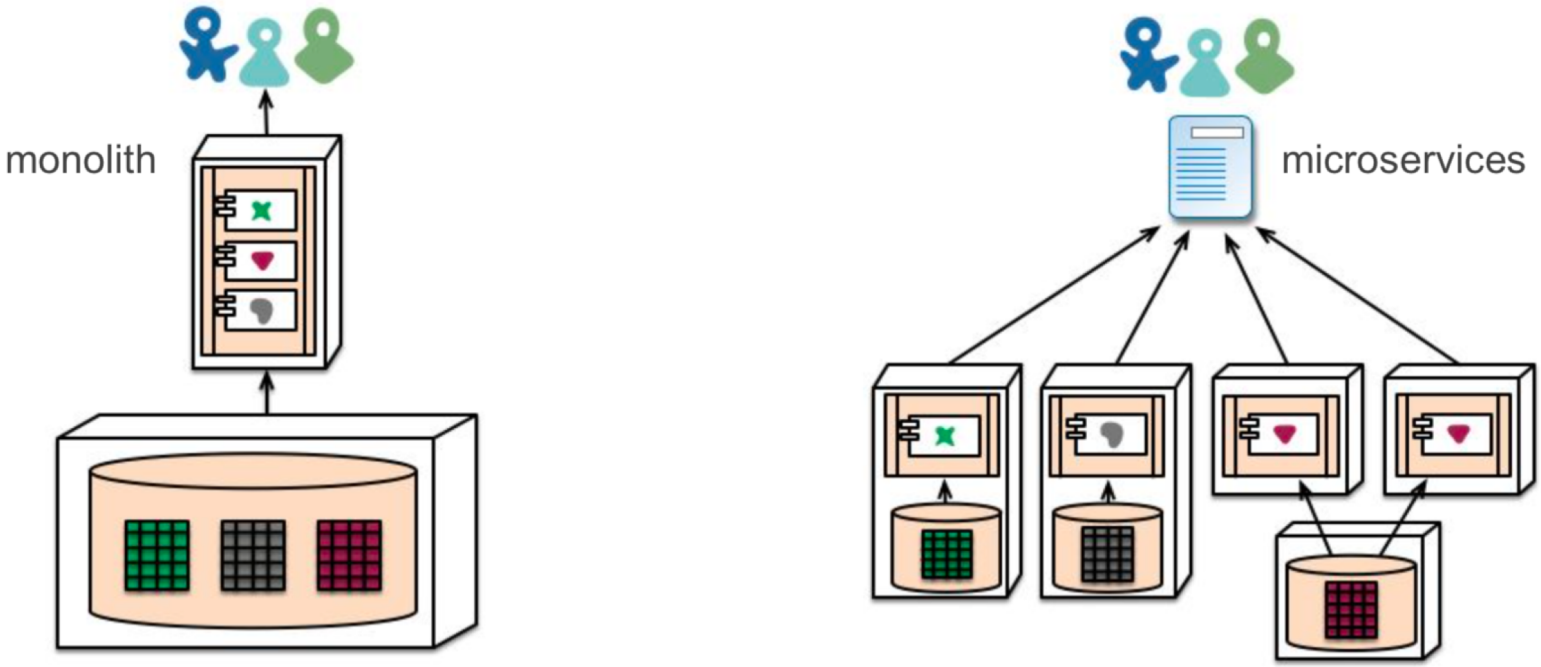
Why?


Gartner's 2017 Hype Cycle for ICT
expectations
Rapid Mobile App Development
Robotic Process Automation Offerings (Gateway Technology Services)
Smart City Framework
Integrated Systems: Hyperconvergence
Bimodal IT
Internet of Things
Digital Commerce Platforms
Software-Defined Data Center
Disaster Recovery as a Service
Machine Learning
DevOps
Virtual Desktop Infrastructure
Enterprise Mobility Management Suites
Solid-State Arrays
Unified Communications and Collaboration
BYOD
ITIL
Personalization Engines
Mobile Money
Social Analytics
Platform as a Service (PaaS)
Big Data
Infrastructure as a Service
Artificial Intelligence for IT Operations (AIOps) Platforms
Virtual Support Agents
As of July 2017
Innovation Trigger
Peak of Inflated Expectations
Trough of Disillusionment
Slope of Enlightenment
Plateau of Productivity
time
Years to mainstream adoption:
less than 2 years
2 to 5 years
5 to 10 years
more than 10 years
obsolete before plateau

"A **suite of small services**, each **running in its own process** and **communicating with lightweight mechanisms**, often an HTTP resource API."

"These services are **built around business capabilities** and **independently deployable** by **fully automated** deployment machinery."

"There is a **bare minimum of centralized management** of these services, which may be written in **different programming languages** and use **different data storage technologies**."
J. Lewis & M. Fowler, ThoughtWorks



source: https://martinfowler.com/articles/microservices.html

"Small **autonomous** services that **work together**, modelled around a **business domain**."

S. Newman, ThoughtWorks, author of "Building Microservices"

"**Loosely coupled** service-oriented architecture with **bounded contexts**."

"Monolithic apps have **invisible internal complexity**. Microservices expose that [complexity] as **explicit micro service dependencies**."

Adrian Cockcroft, AWS (formerly at Netflix)

"We need to move to **managed complexity**. Microservices are about **negotiated interfaces**, **strict boundaries**, **shared nothing**!"

J. Higginbotham, LaunchAny

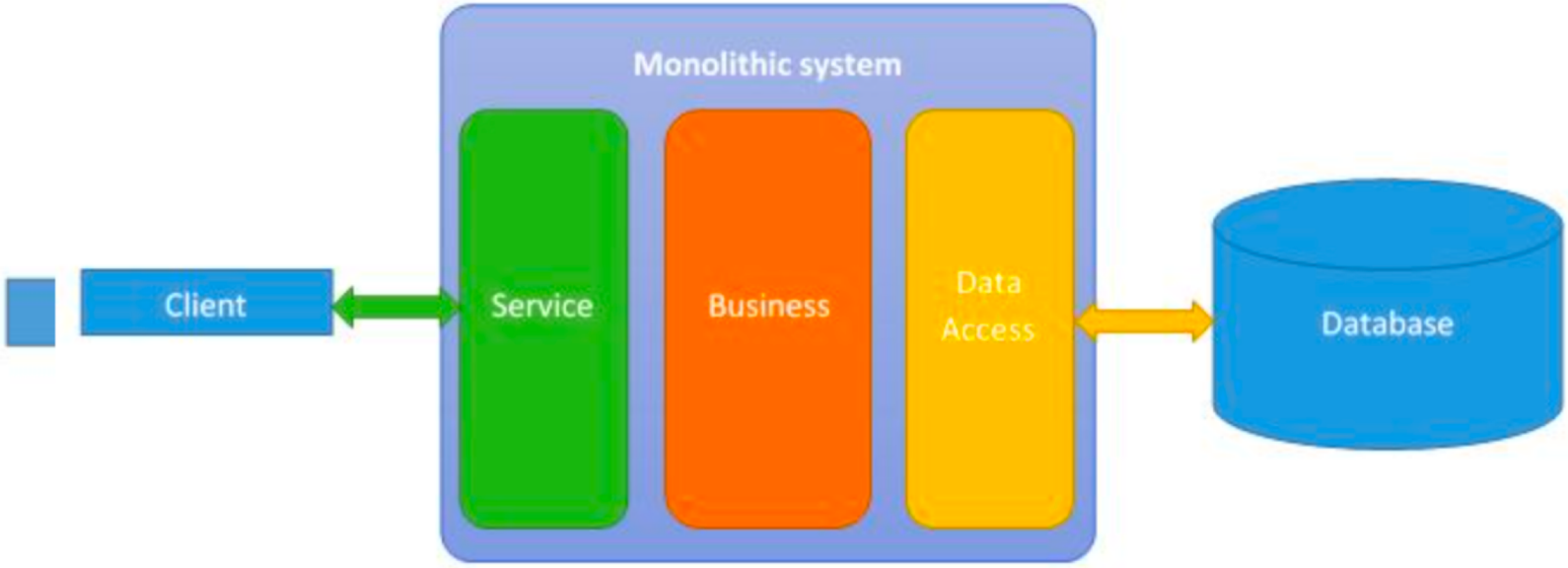
Amazon's now famous migration from the Obidos monolithic application to a service-oriented architecture with encapsulated databases and small, "two-pizza" teams

Amazon's design principles:

- Design for flexibility
- Design for on demand
- Design for automation
- Design for failure
- Be elastic
- Design for utility pricing
- Break transparency
- Decompose to its simplest form
- Design with security in mind
- Don't do It alone
- Focus on what doesn't change
- Let your customers benefit
- Continuously innovate

"For us service orientation means encapsulating the data with the business logic that operates on the data, with the only access through a published service interface. No direct database access is allowed from outside the service, and there's no data sharing among the services."

-- Werner Vogel, Amazon's CTO, 2006



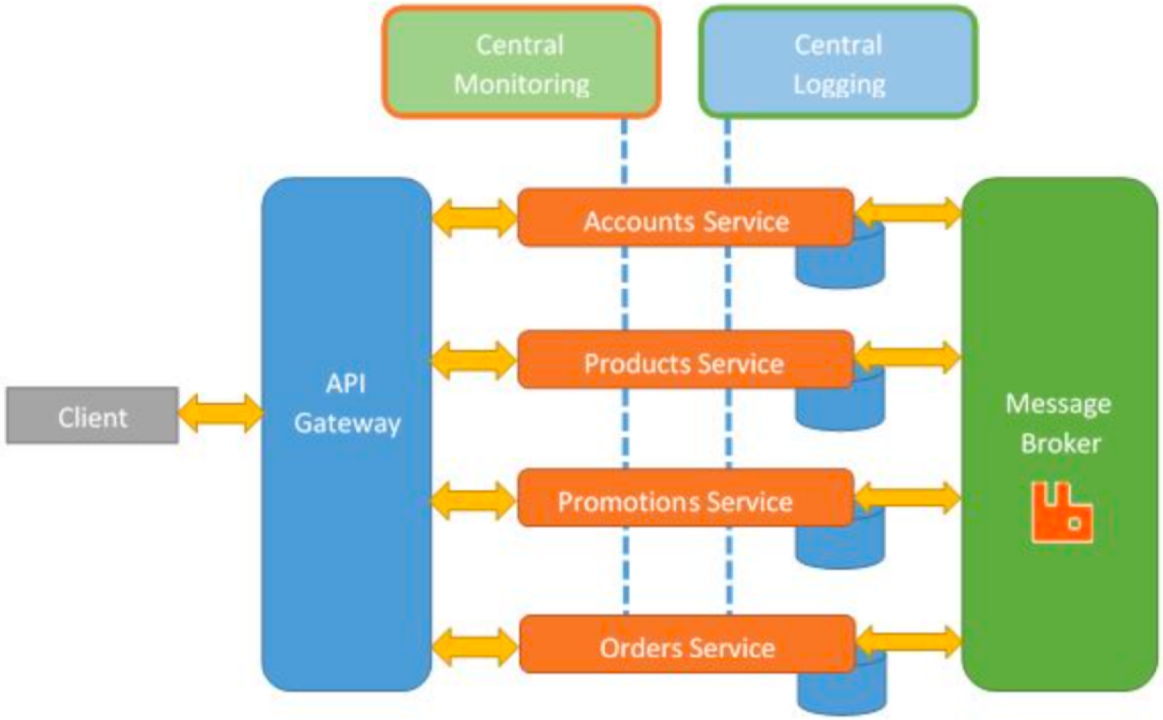
source: http://www.acarlstein.com/

"There's no reason why you can't make a single monolith with well defined module boundaries. At least there's no reason *in theory*. In practice, it seems too easy for module boundaries to be breached and monoliths to get tangled as well as large."

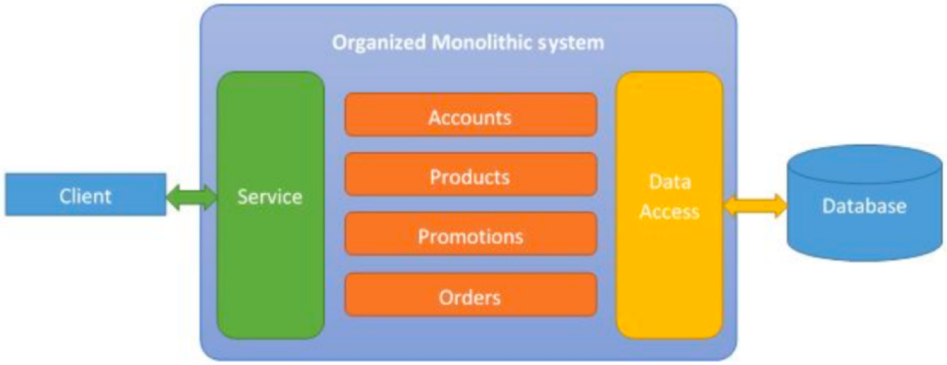
-- Martin Fowler

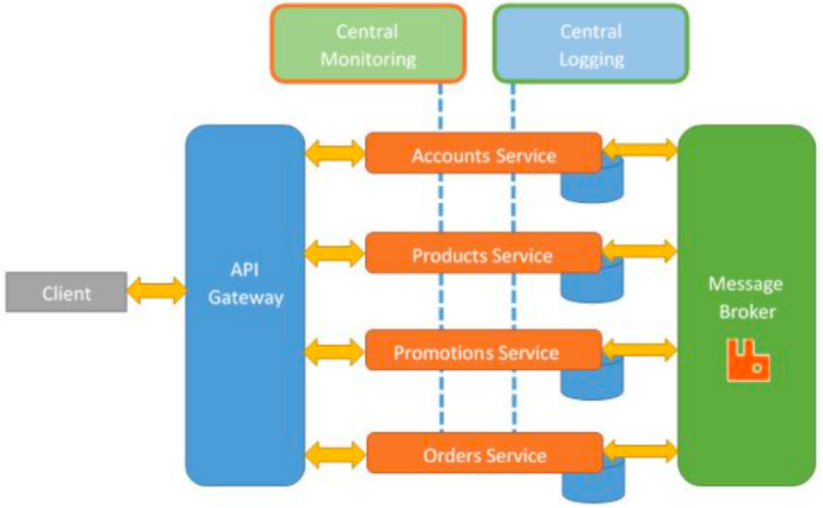
The microservices style is an approach to design systems whose parts are easy to change and replace *at runtime*

It pushes information hiding to new heights by enforcing strict module boundaries and by promoting information isolation



source: http://www.acarlstein.com/



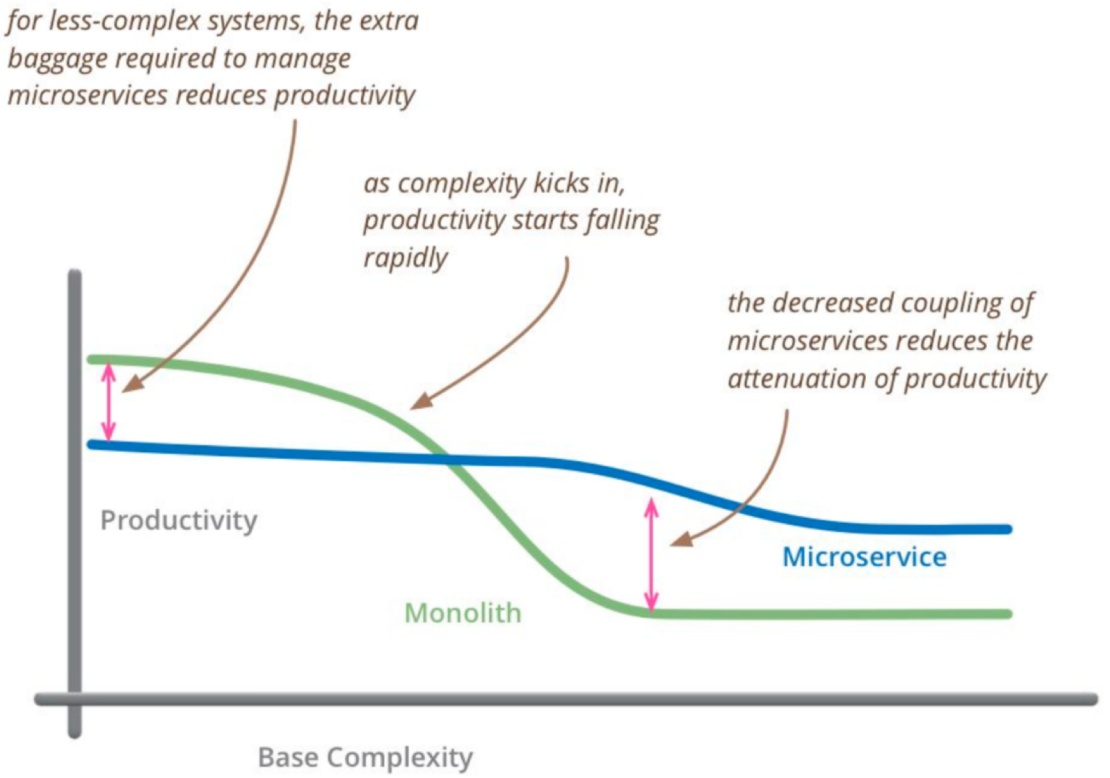


- One large code base
- Long release cycles
- Usually stuck with one dominant technology
- More susceptible to system-wide failures
- Can only monitor system-level properties (difficult to identify which modules are causing problems)
- Scaling requires updating or replicating the whole system stack

- Multiple "small" code bases
- Short release cycles
- Freedom to explore different (i.e., competing) technologies
- Failure in one module may not necessarily affect the other modules
- Modules can be individually monitored
- Modules can be independently scaled

The microservices style introduces its own set of (distributed systems related) complexities:

- automated deployment
- continuously monitoring
- dealing with failure
- eventual consistency
- security

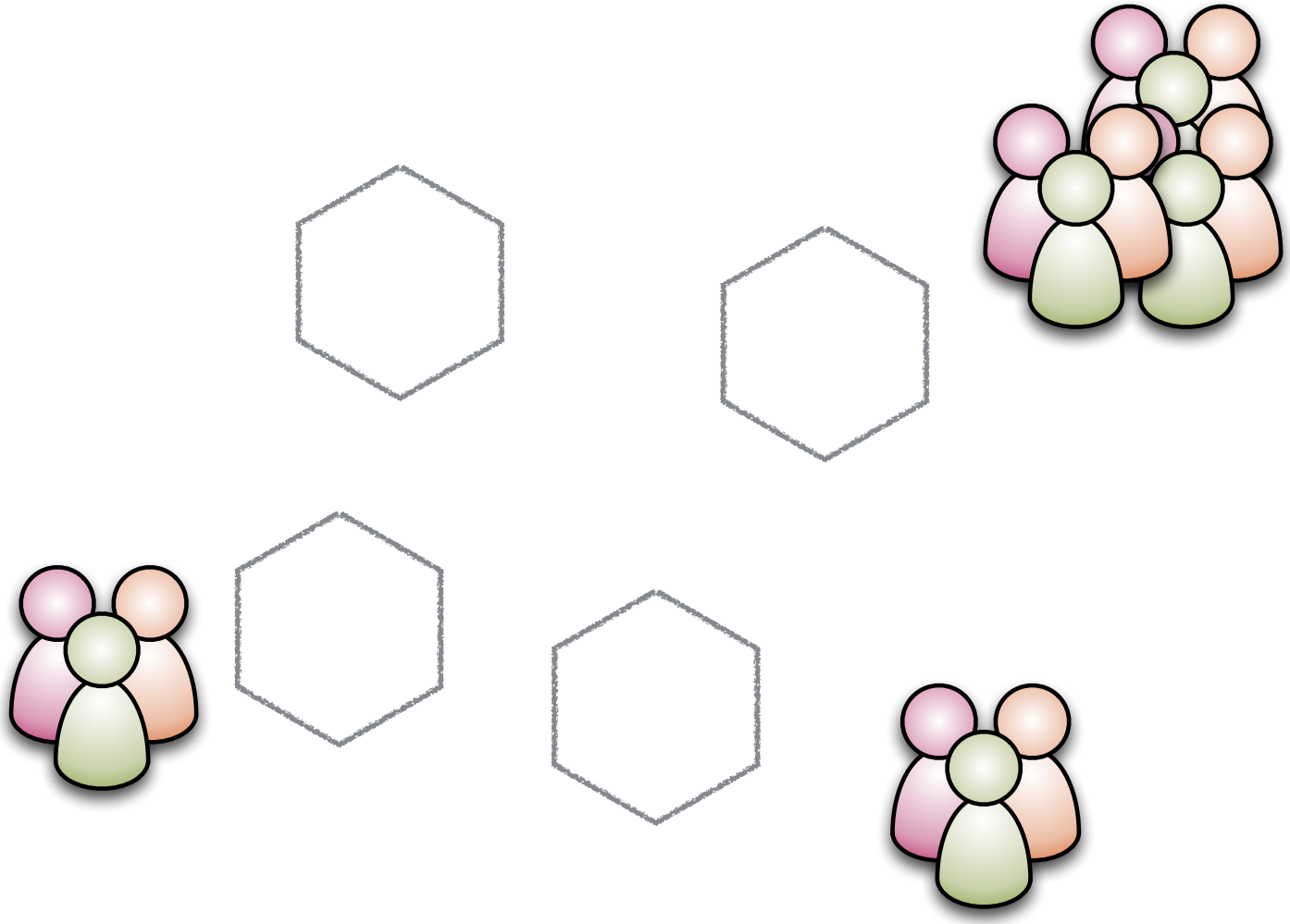


source: https://martinfowler.com/bliki/MicroservicePremium.html

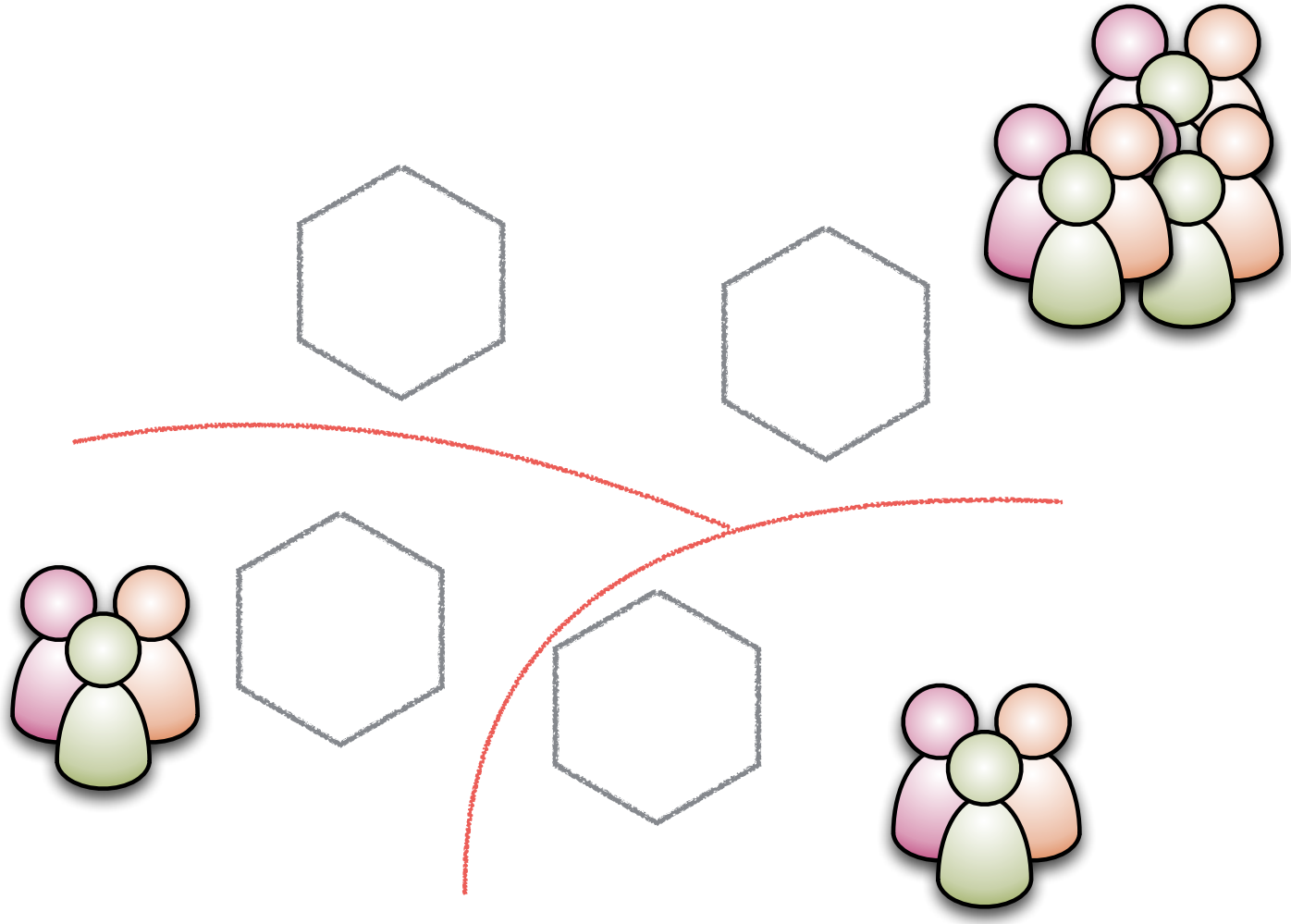
"The majority of software systems should be built as a single monolithic application. Do pay attention to good modularity within that monolith. Don't even consider microservices unless you have a system that's too complex to manage as a monolith."

-- Martin Fowler

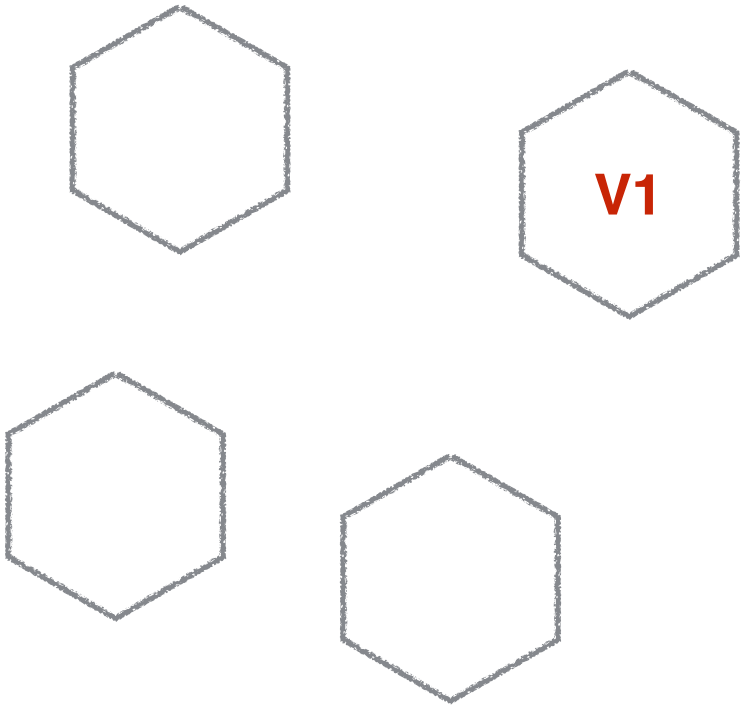
We can organise services along organisational boundaries



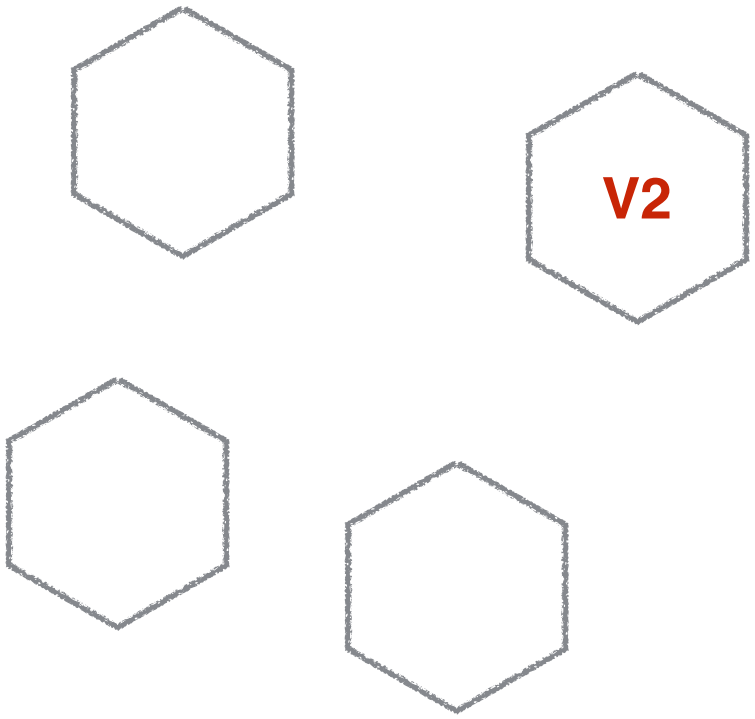
We can organise services along organisational boundaries



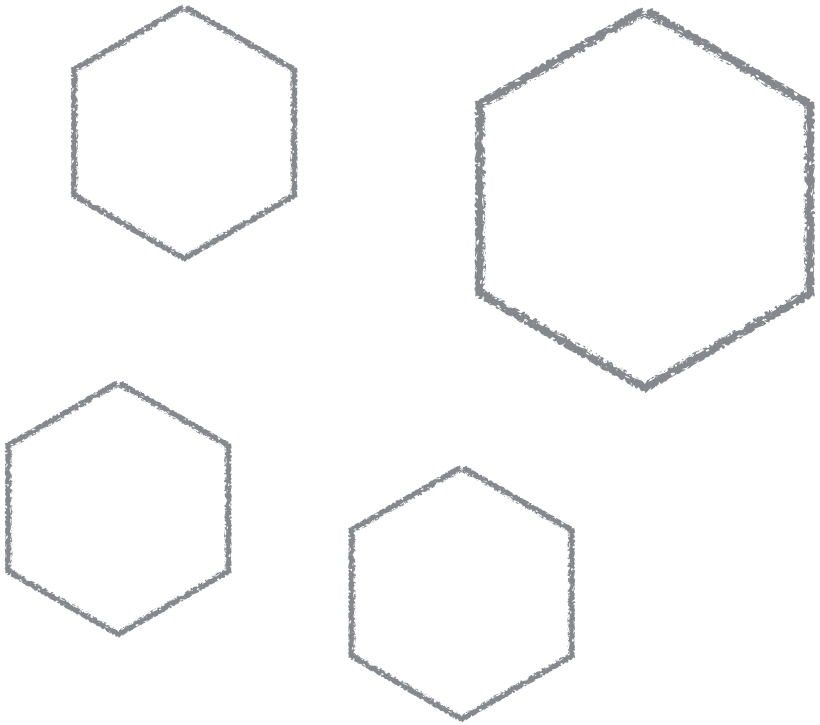
By sub-divinding our systems, we can speed the release of new features



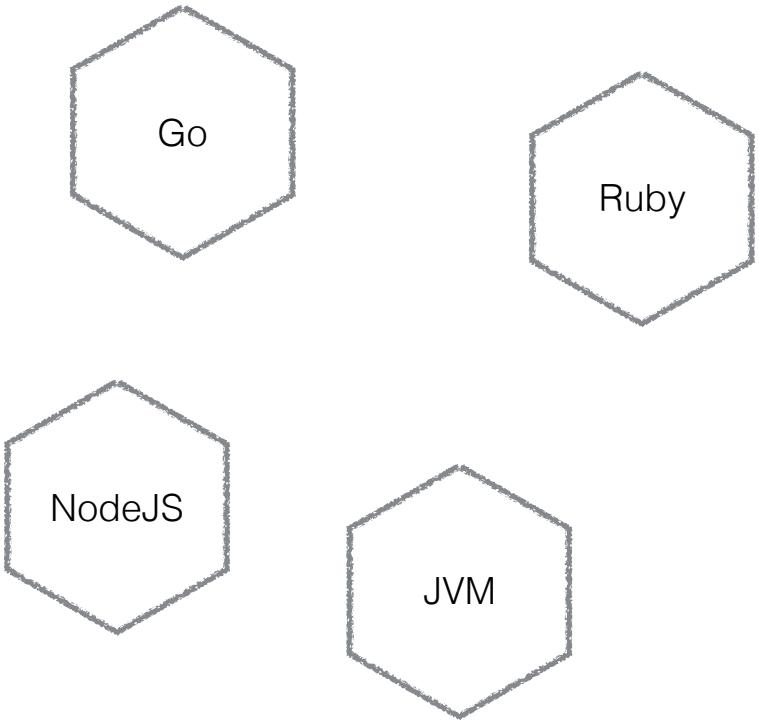
By sub-divinding our systems, we can speed the release of new features



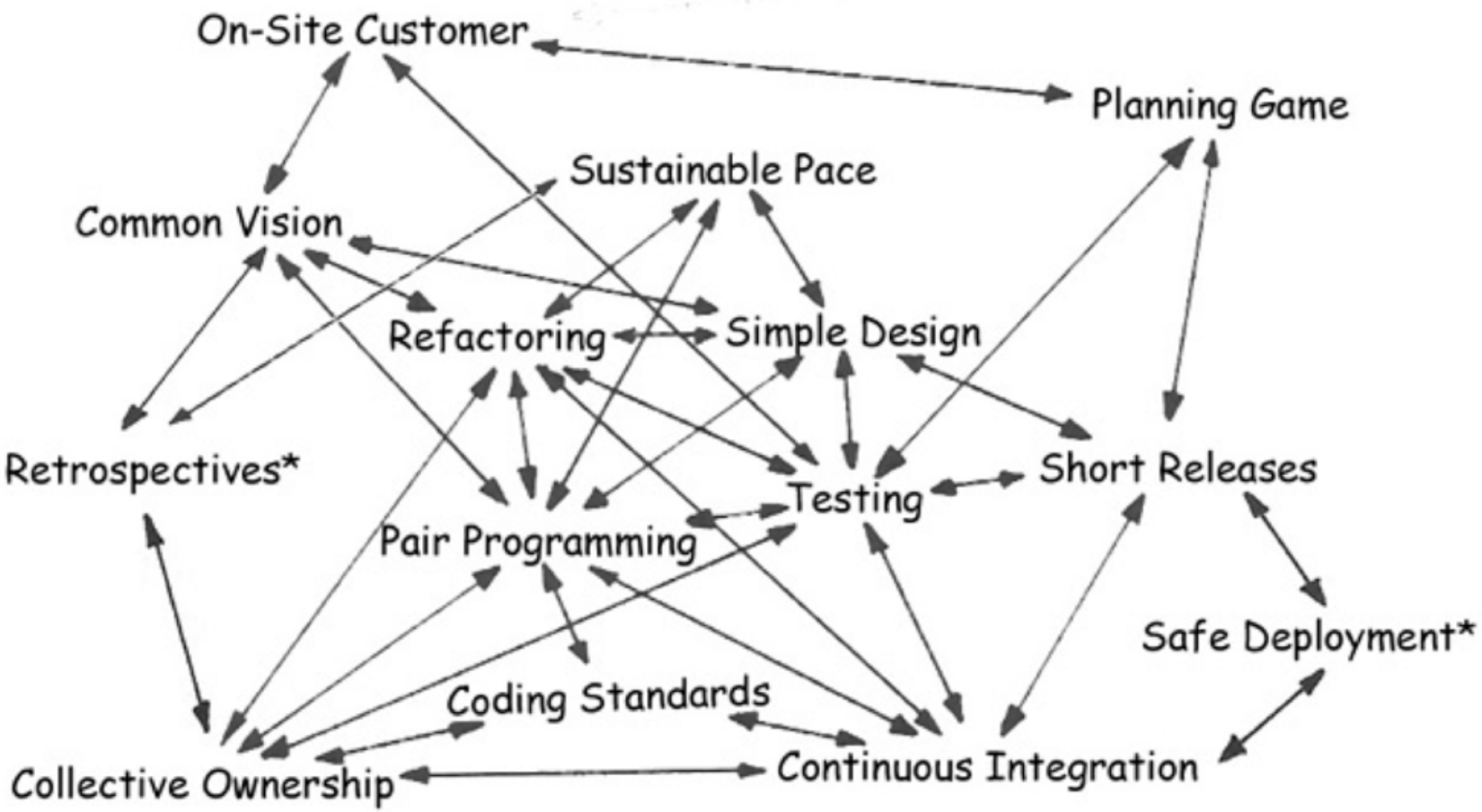
It allows us different options in terms of scaling



and we can use different tools and tech



Why now?


On-Site Customer
Planning Game
Sustainable Pace
Common Vision
Refactoring
Simple Design
Retrospectives*
Testing
Short Releases
Pair Programming
Safe Deployment*
Coding Standards
Collective Ownership
Continuous Integration


standing on the
shoulders of giants


The Pragmatic Programmers
Release It!
Design and Deploy
Production-Ready Software
Michael T. N
The Addison-Wesley Signature Series
A Martin Fowler Signature Book
Continuous Delivery
Reliable Software Releases through Build, Test, and Deployment Automation
Jez Humble
David Farley
Foreword by Martin Fowler
Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler
REST in Practice

Summary

We understand more about building reliable distributed systems

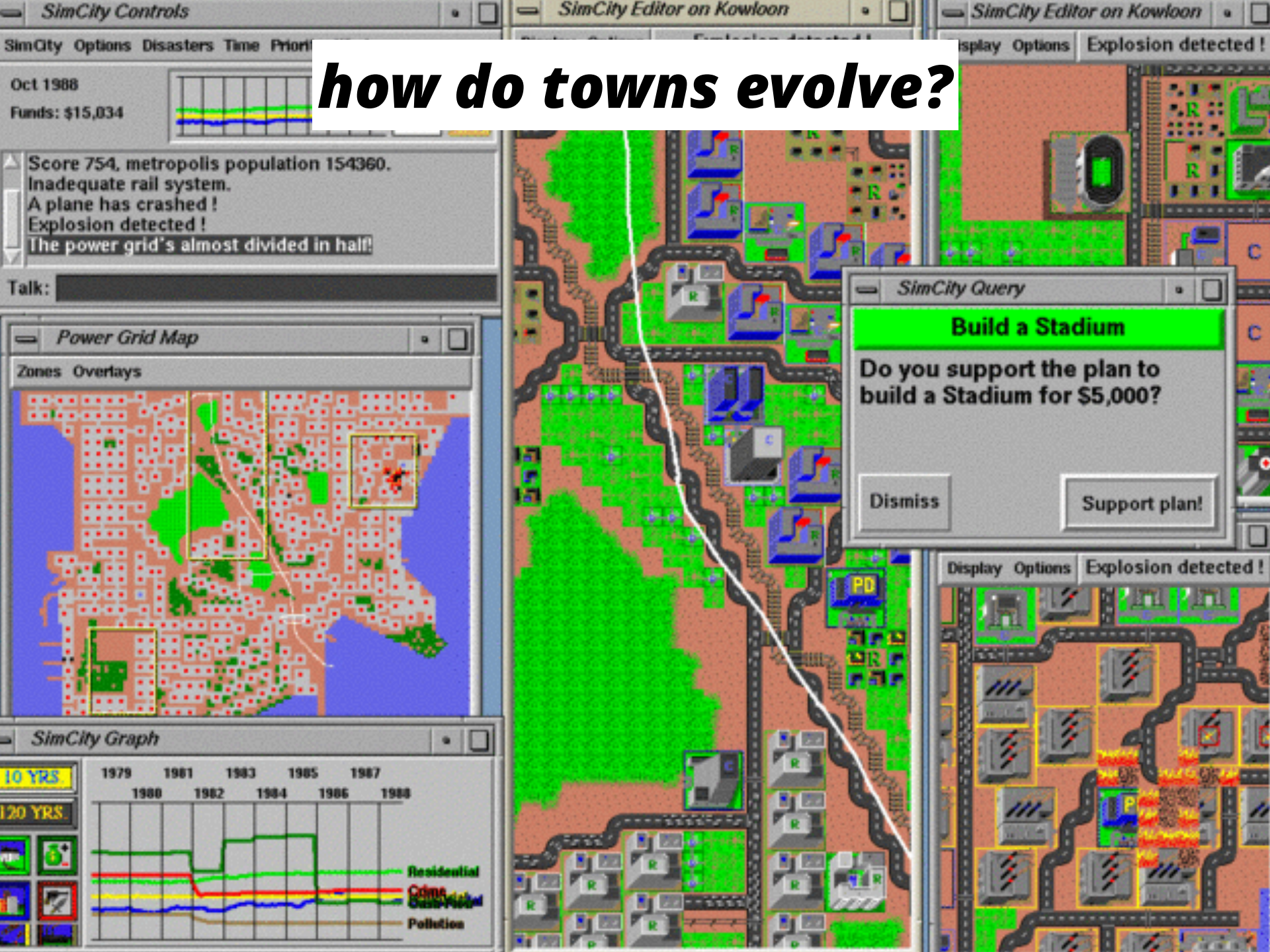
cloud compute and programmable infrastructure has matured

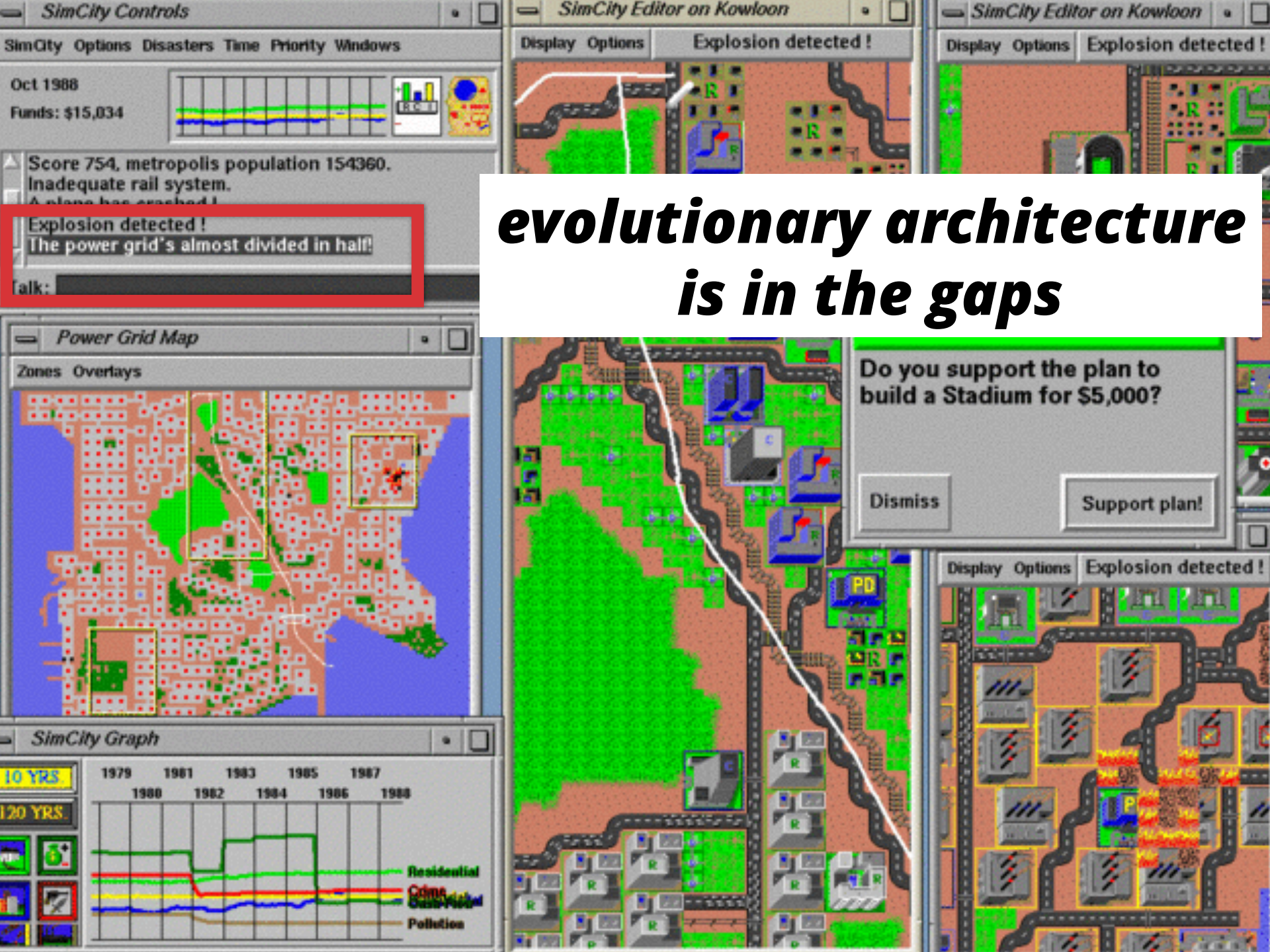
organisations need to adapt and change quickly to survive

we spend **too much money** on building monoliths

Evolutionary Architecture

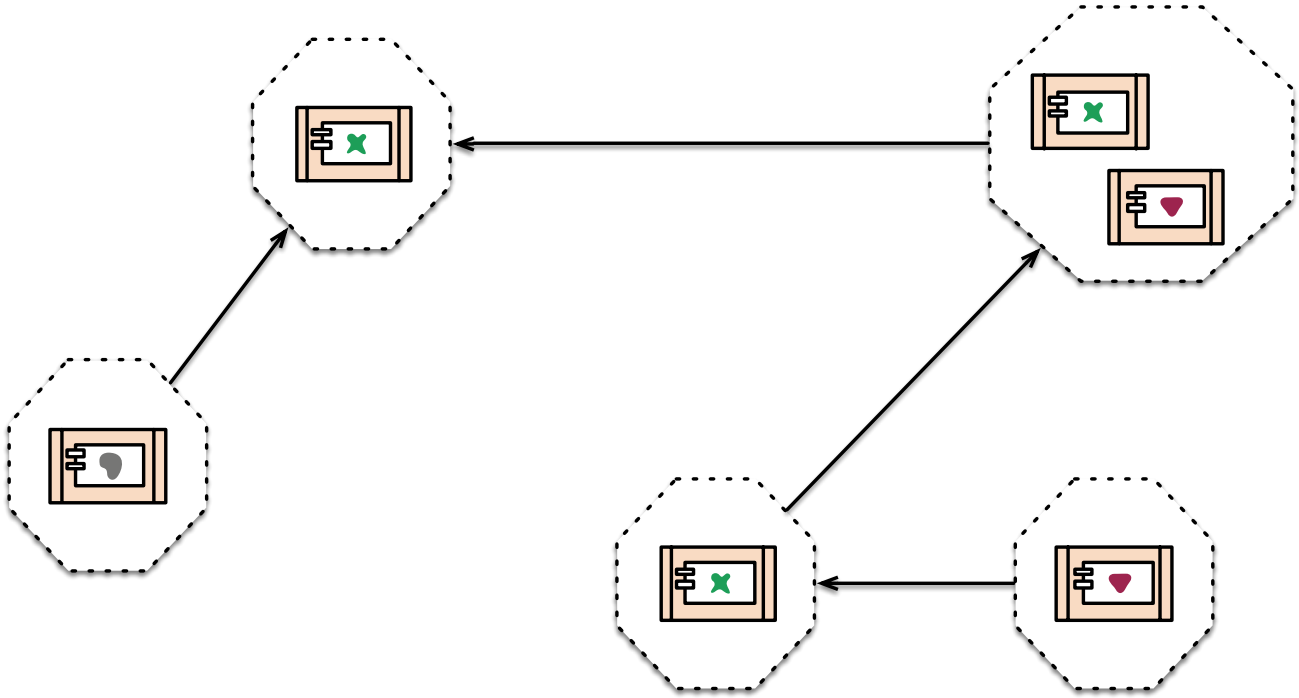
“Just enough architecture”


how do towns evolve?
SimCity Controls
SimCity Options Disasters Time Priorit
Oct 1988
Funds: $15,034
Score 754, metropolis population 154360.
Inadequate rail system.
A plane has crashed !
Explosion detected !
The power grid's almost divided in half!
Talk:
Power Grid Map
Zones Overlays
SimCity Graph
10 YRS
120 YRS
1979 1980 1981 1982 1983 1984 1985 1986 1987 1988
Residential
Pollution
SimCity Editor on Kowloon
SimCity Editor on Kowloon
Display Options
Explosion detected !
SimCity Query
Build a Stadium
Do you support the plan to build a Stadium for $5,000?
Dismiss
Support plan!
Display Options
Explosion detected !


SimCity Controls
SimCity Options Disasters Time Priority Windows
Oct 1988
Funds: $15,034
Score 754, metropolis population 154360.
Inadequate rail system.
Explosion detected !
The power grid's almost divided in half!
Talk:
SimCity Editor on Kowloon
Display Options
Explosion detected !
evolutionary architecture
is in the gaps
Power Grid Map
Zones Overlays
Do you support the plan to
build a Stadium for $5,000?
Dismiss
Support plan!
SimCity Graph
10 YRS
120 YRS
1979 1980 1981 1982 1983 1984 1985 1986 1987 1988
Residential
Pollution

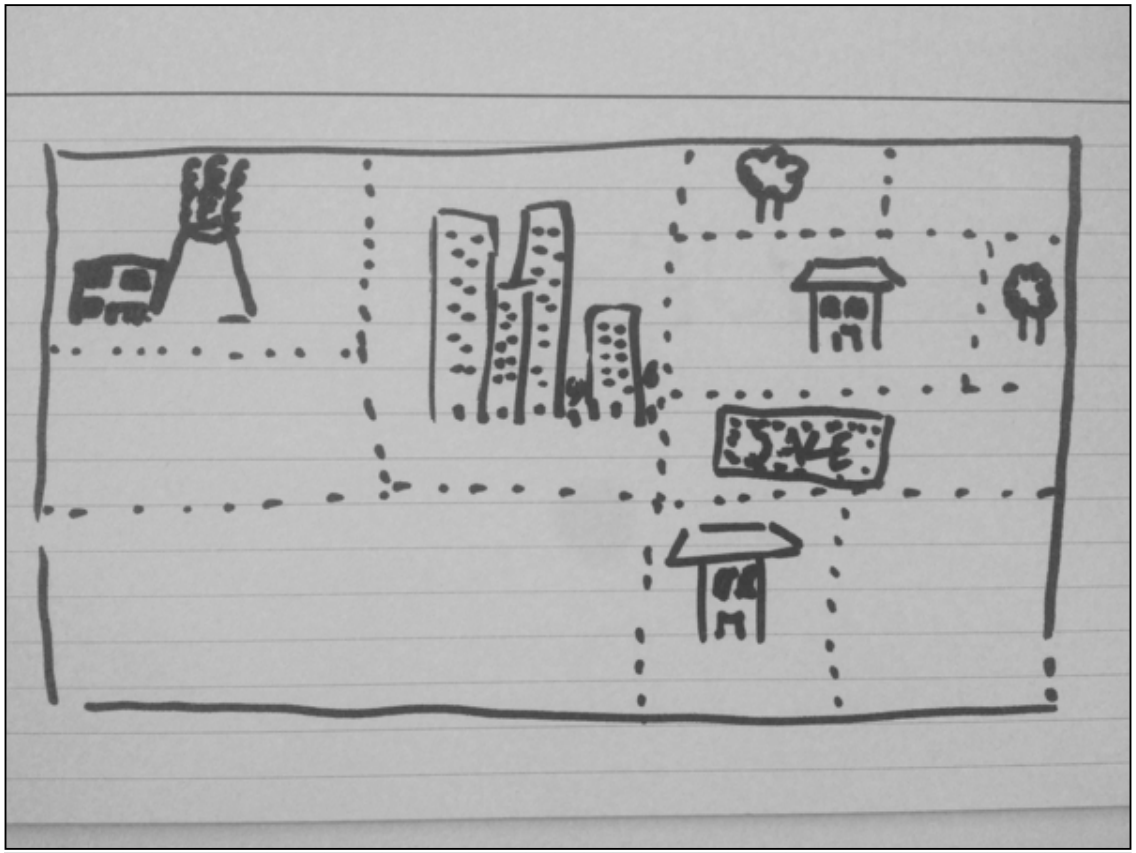

SimCity Controls
SimCity Options Disasters Time Priority Windows
Oct 1988
Funds: $15,034
Score 754, metropolis population 154360.
Inadequate rail system.
A plane has crashed !
Explosion detected !
The power grid's almost divided in half!
Talk:
SimCity Editor on Kowloon
Display Options
Explosion detected !
SimCity Query
Build a Stadium
Do you support the plan to
Power Grid Map
Zones Overlays
SimCity Graph
10 YRS
120 YRS
1979 1980 1981 1982 1983 1984 1985 1986 1987 1988
Residential
Pollution
emergent design is within the zones

ALLOW BUSINESS CAPABILITIES TO EVOLVE



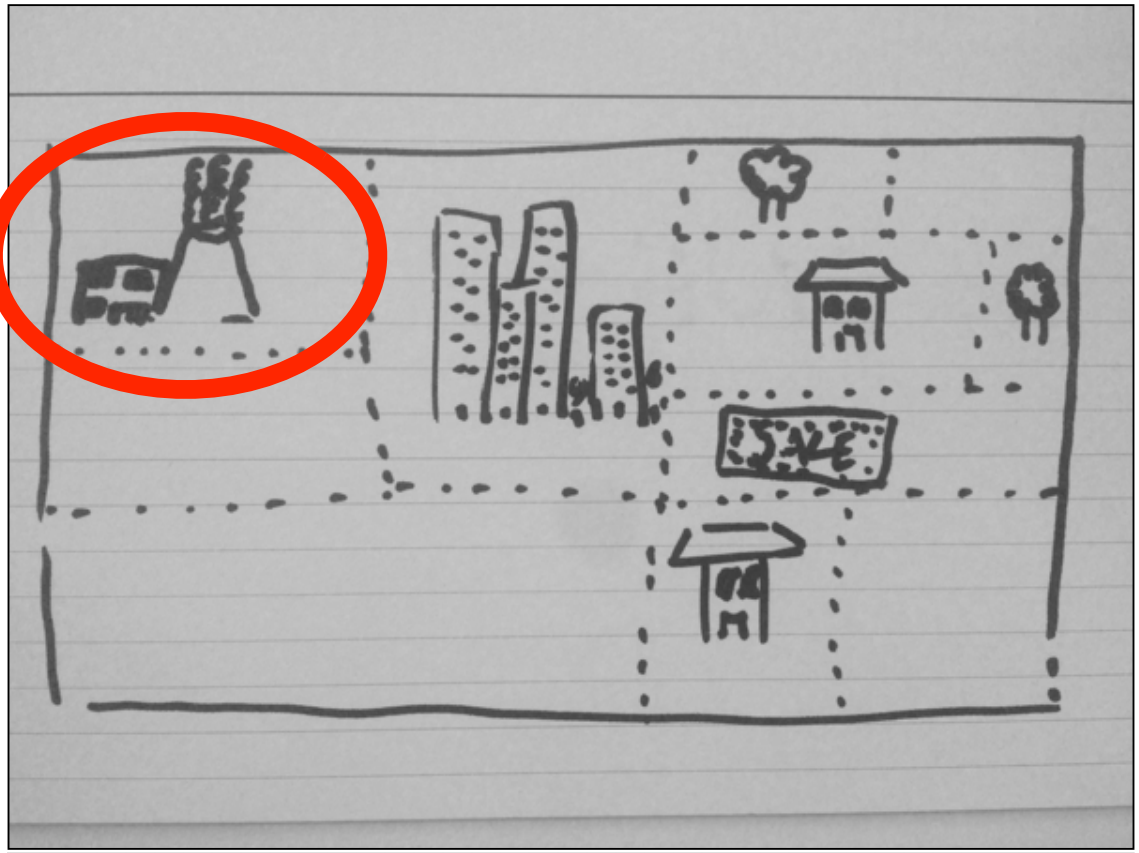
HAVE A ROUGH IDEA ABOUT WHAT YOU WANT TO BUILD, AND DEFER DECISIONS UNTIL YOU KNOW MORE


TOWN
PLAN

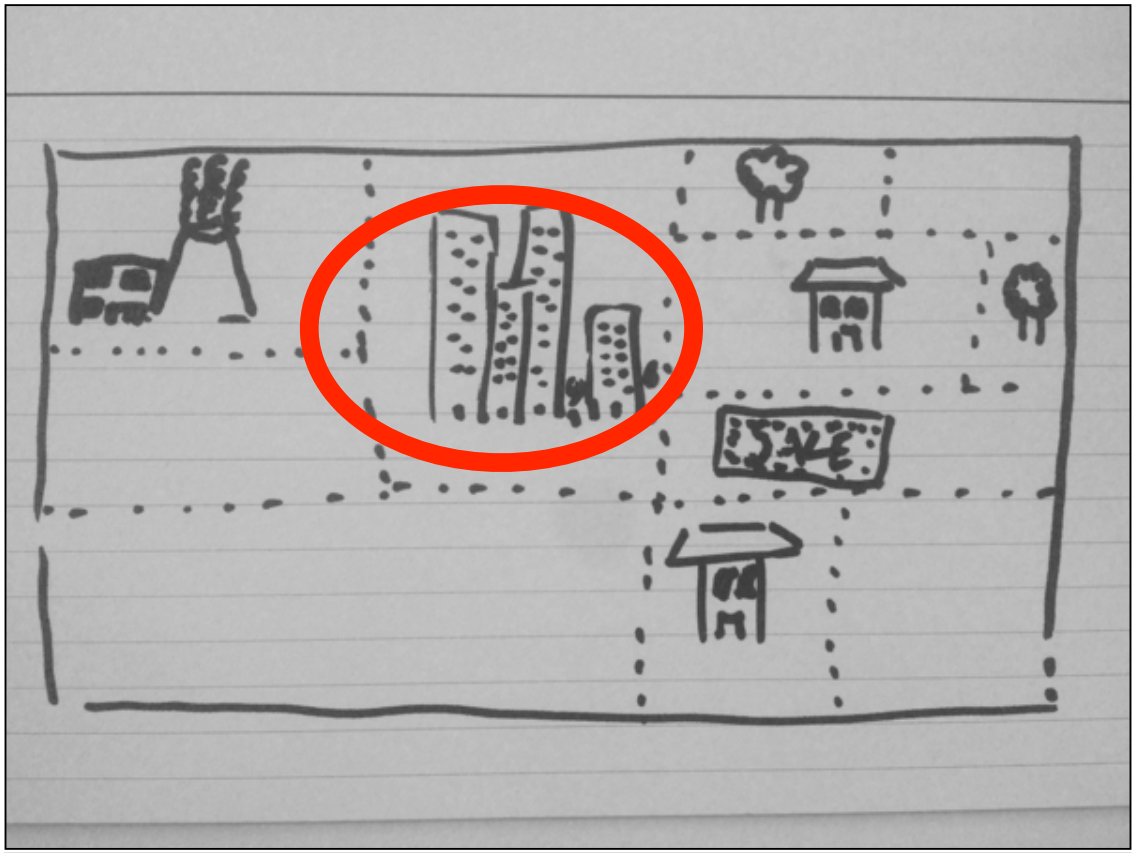


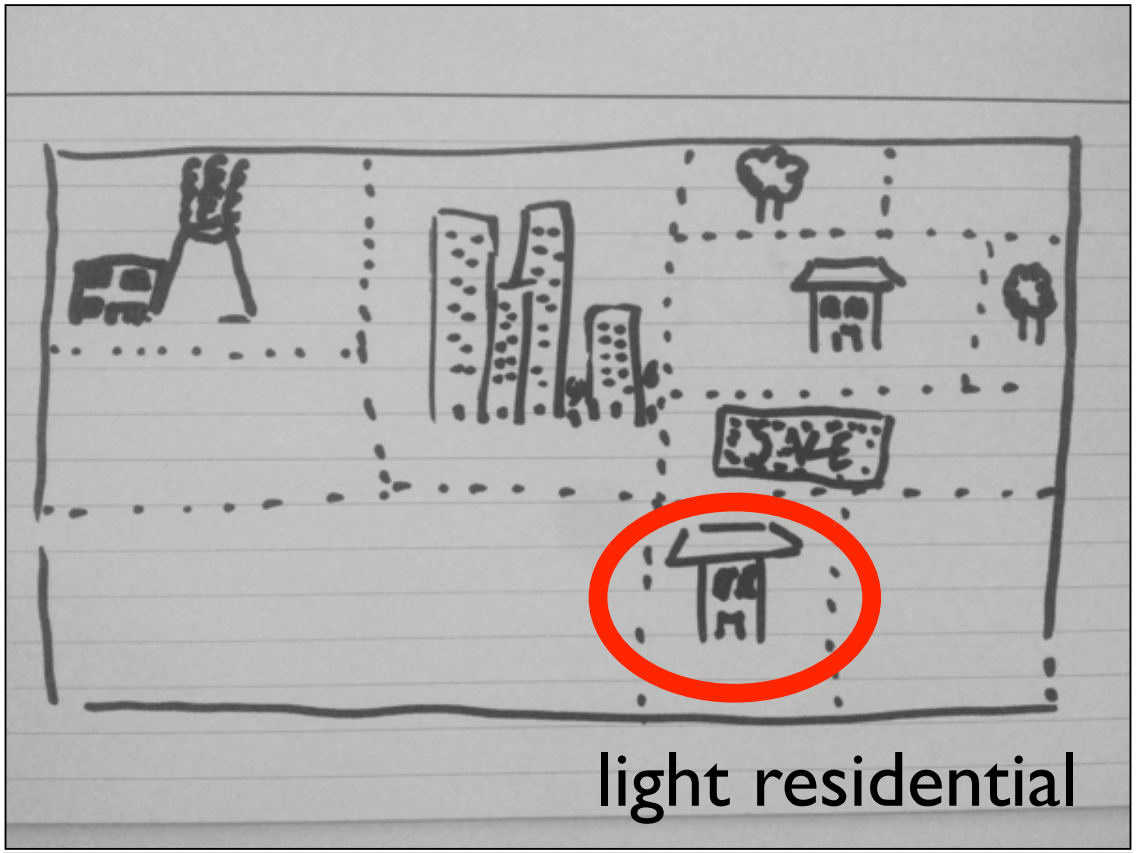
Towns are Zoned

heavy industrial

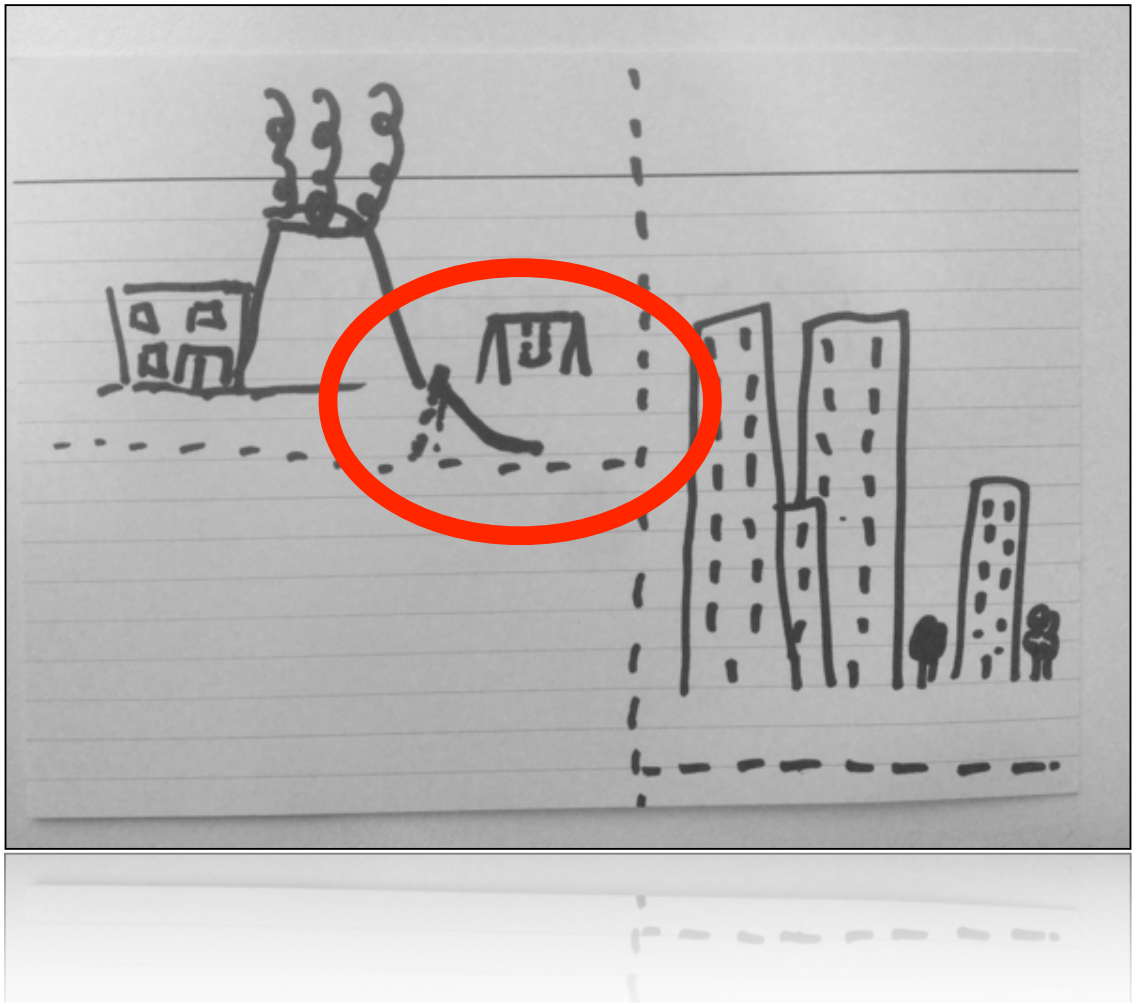


commercial



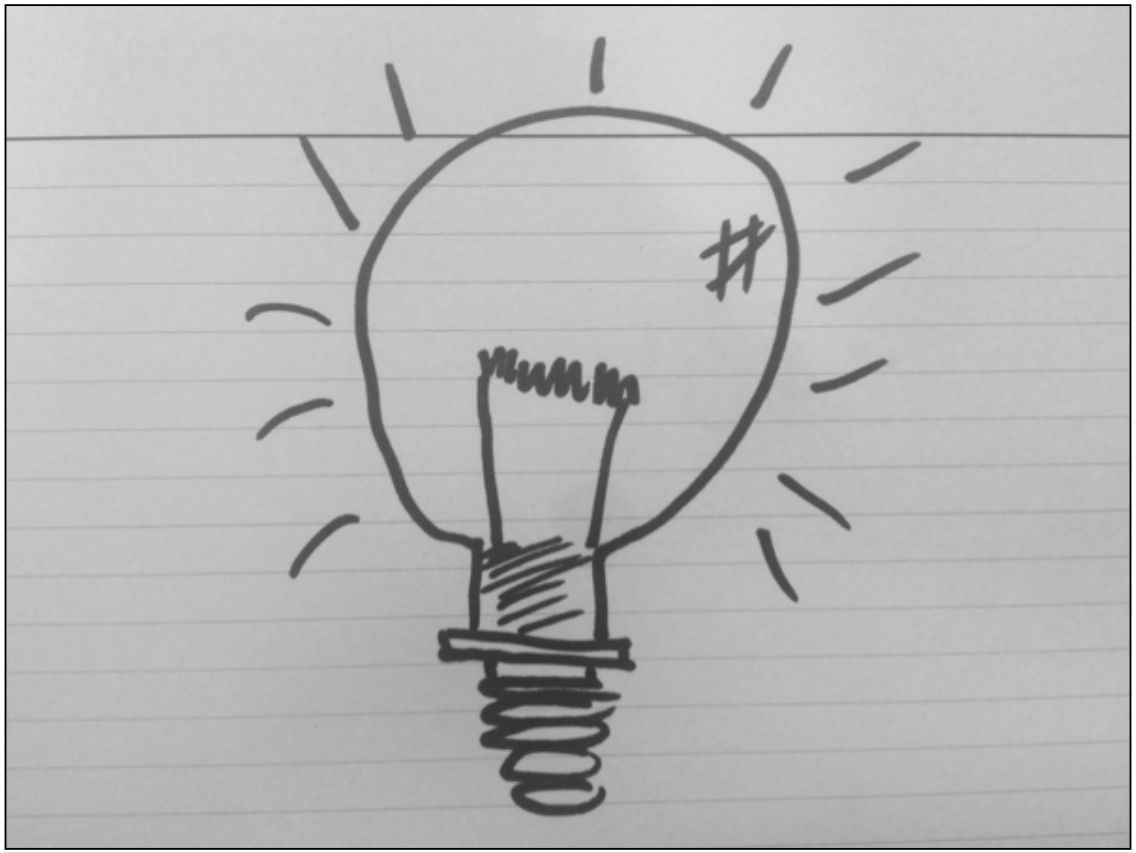

light residential

Would you build a playground next to a power station?

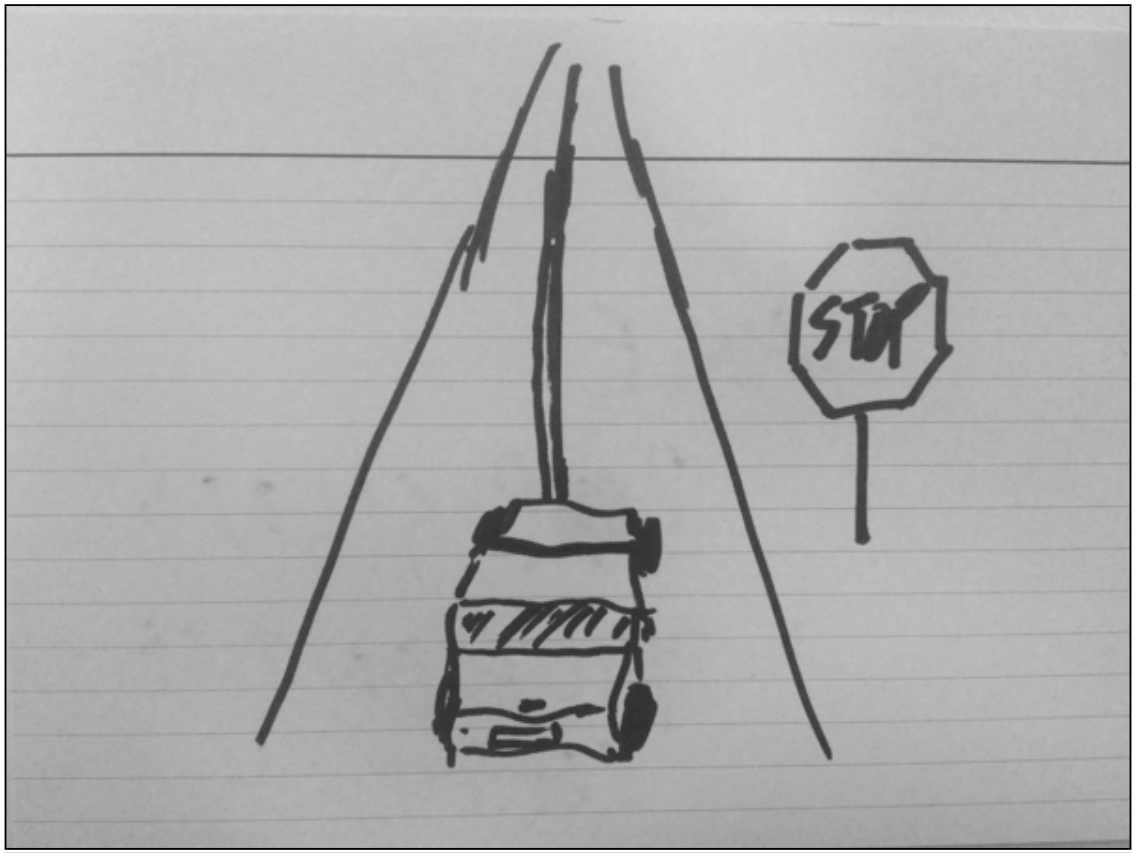


Town share utilities

Everyone uses 240V DC right?

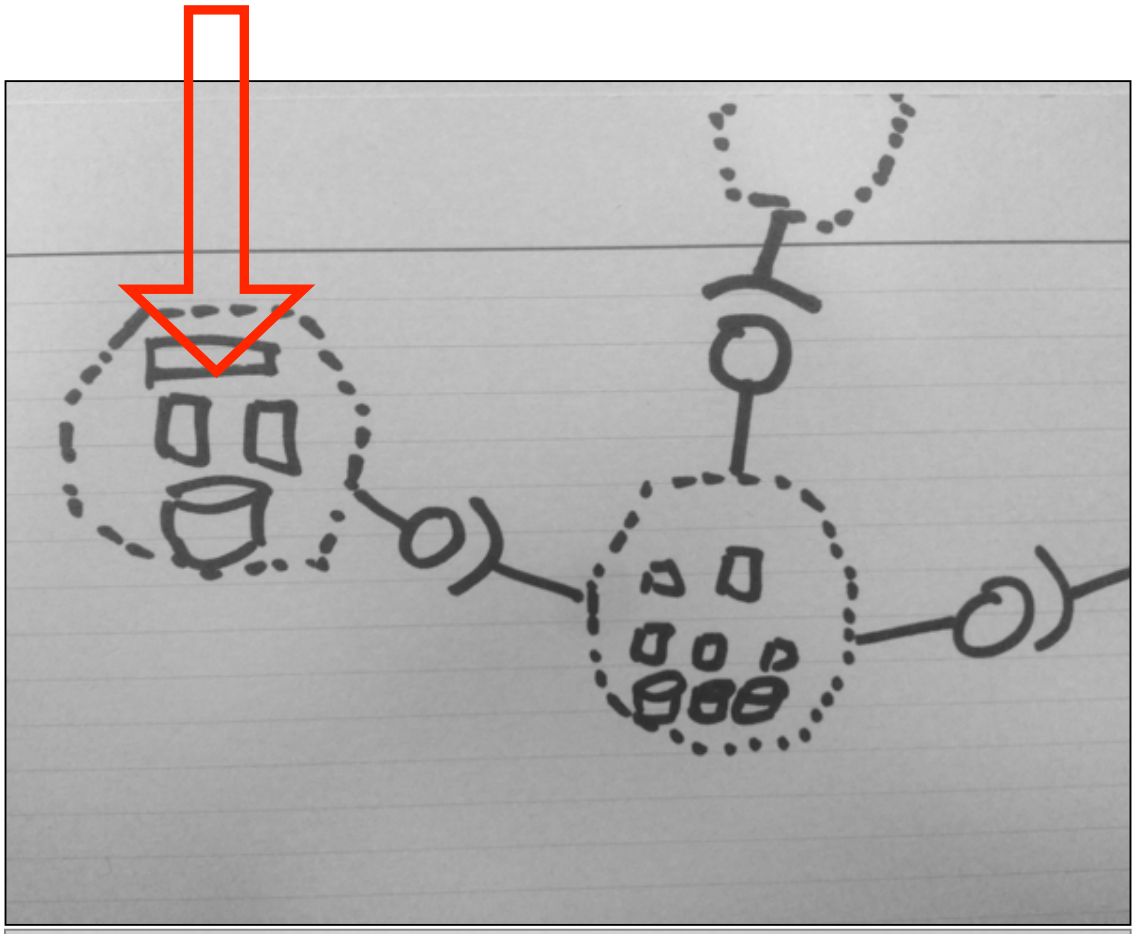


and it would be a bad idea not to use the same language for stop signs...



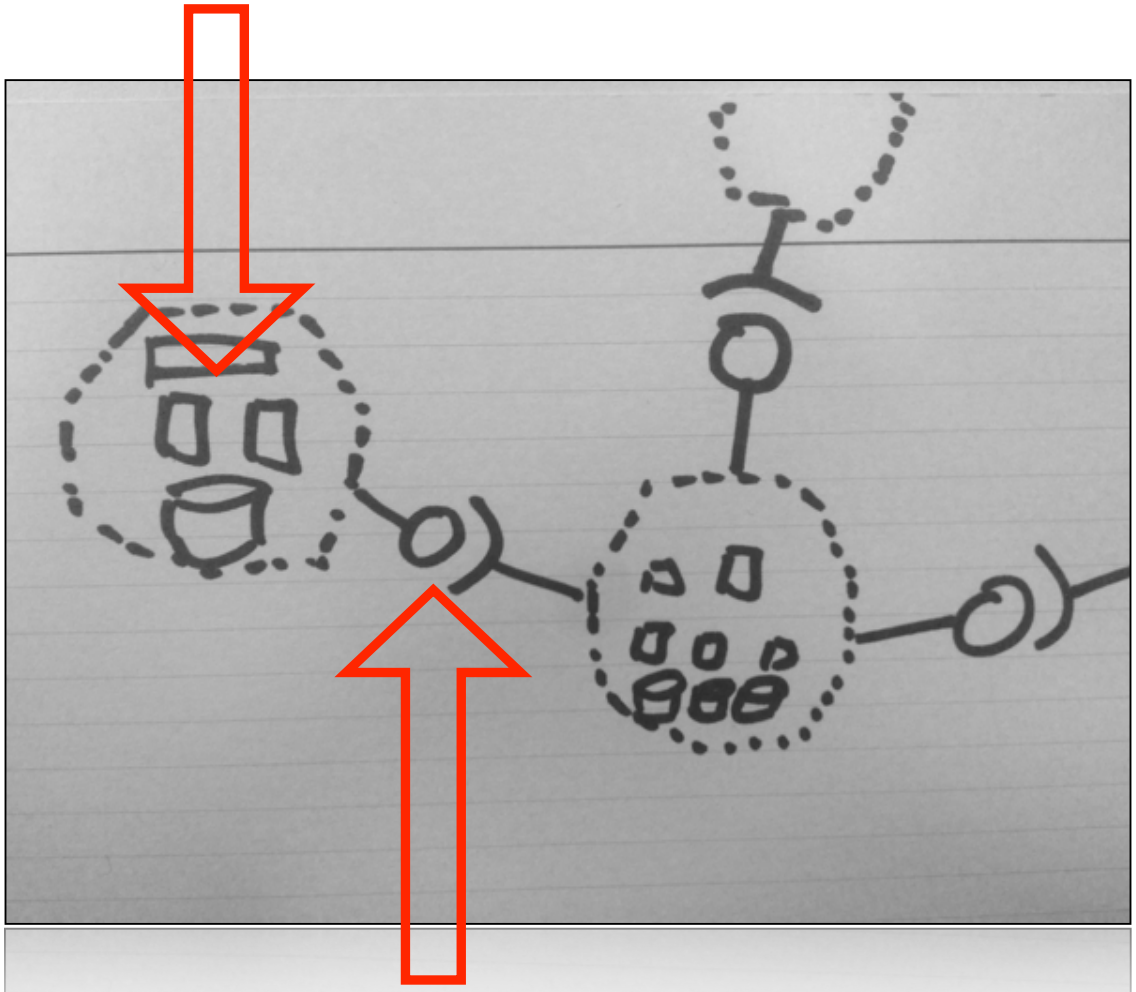

CAPABICITY
MAP

emergent design is within the **zones**



evolutionary architecture is in the **gaps**

emergent design is within the **zones**



evolutionary architecture is in the **gaps**

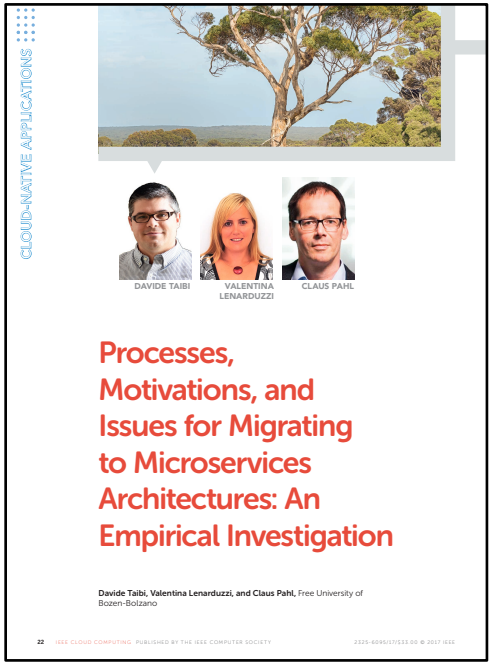
Think about

- Concentrate on the business capabilities
  - technical acronyms make us think the wrong way
- What are the common features? Integration methods?
- What different types of data live where?

Microservices Migration Process

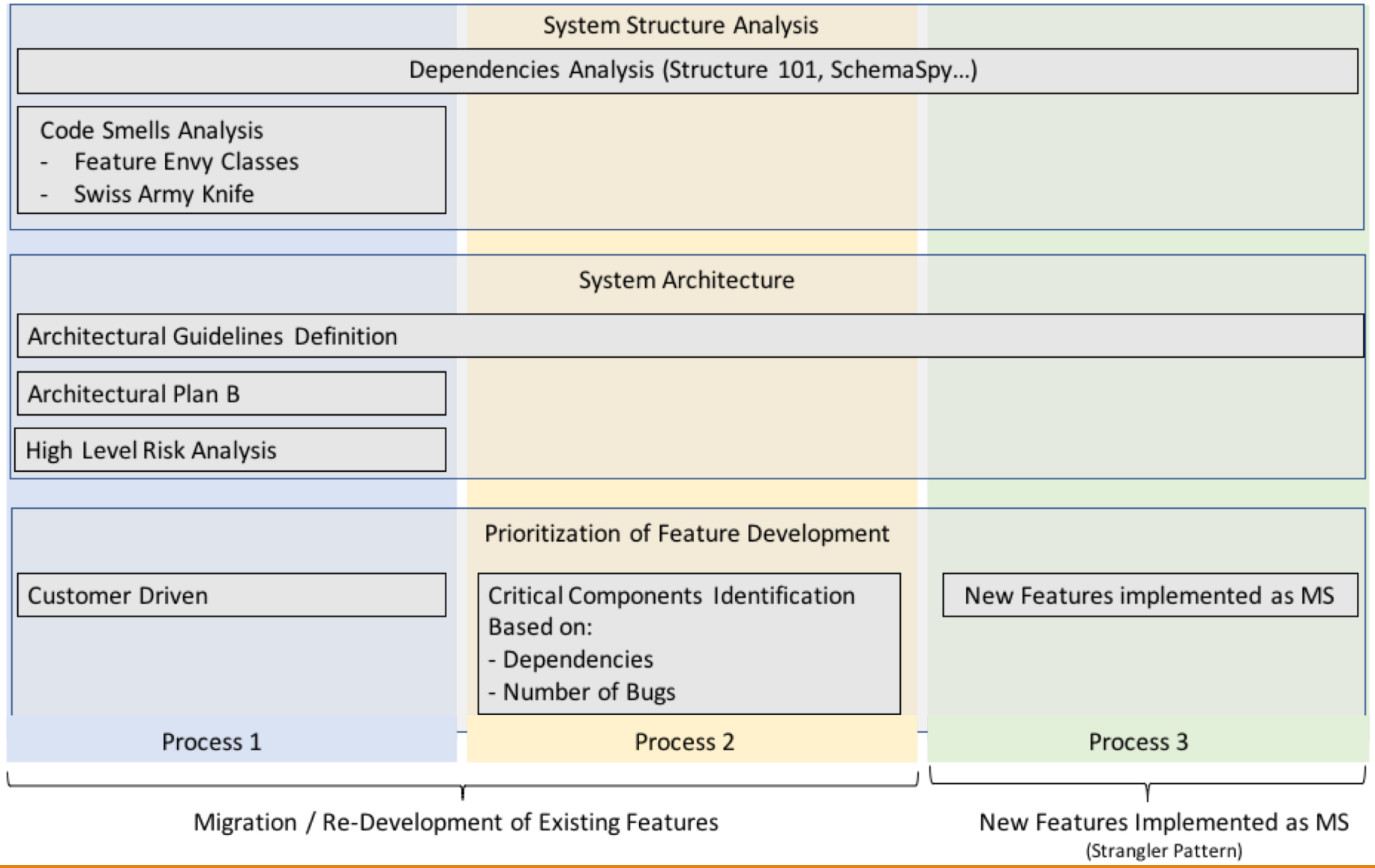
21 companies interviewed

Goal: Understand the migration process

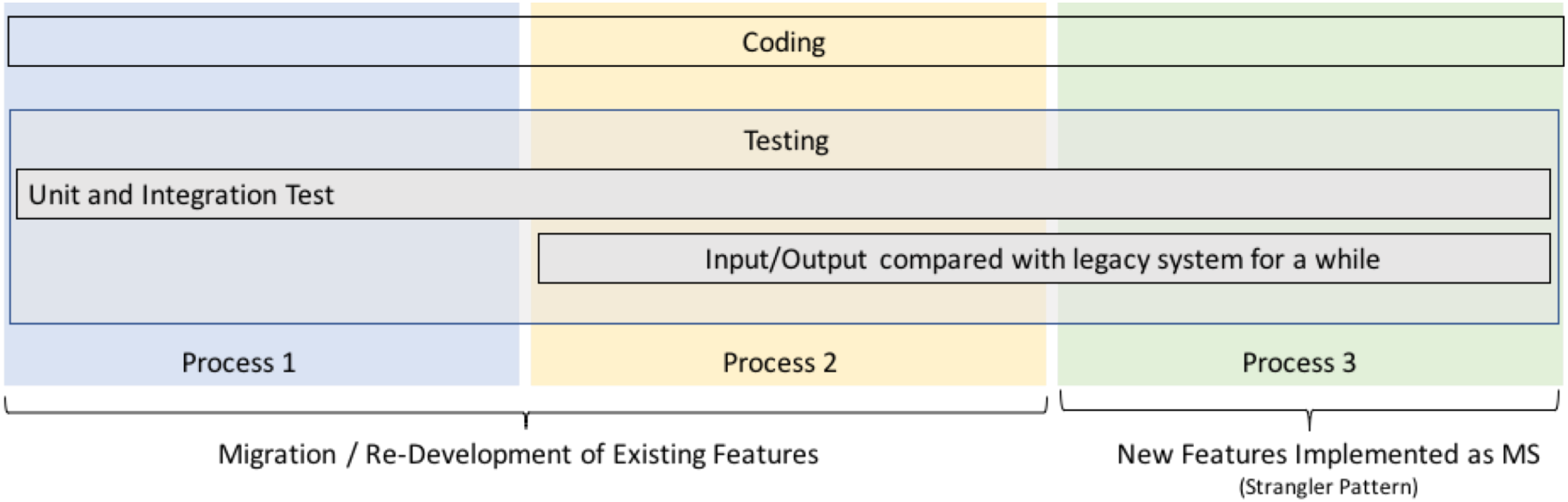


IEEE CLOUD Sept/Oct 2017

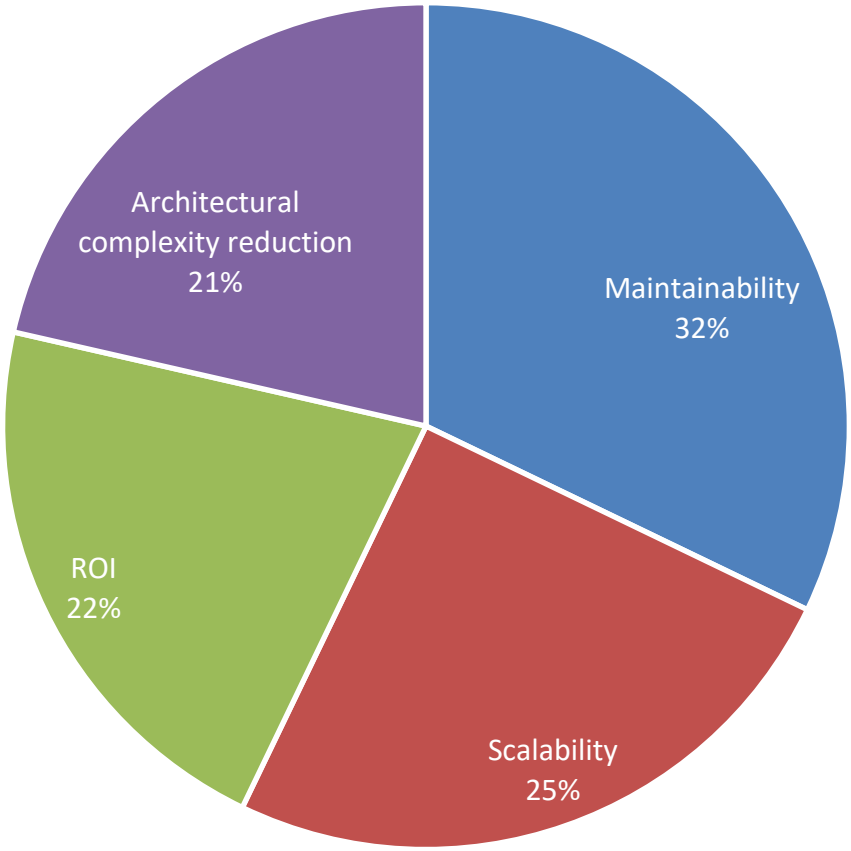
Migration Process (1/2)



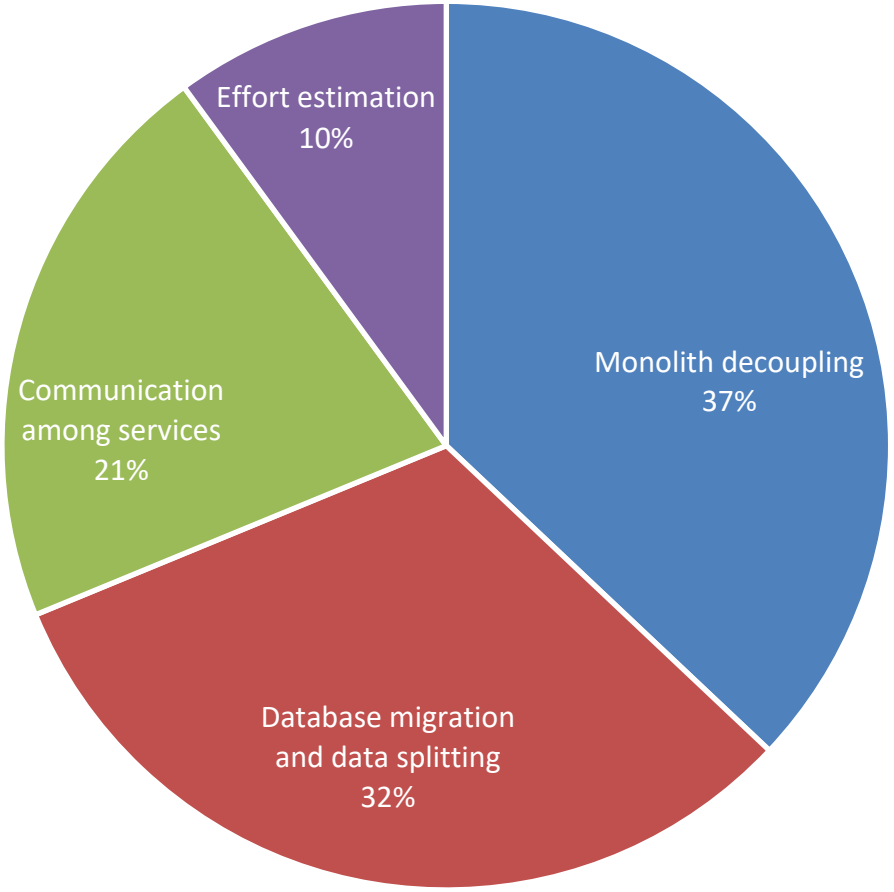
Migration Process (2/2)



Main Benefits



Main Issues


Effort estimation
10%
Monolith decoupling
37%
Communication
among services
21%
Database migration
and data splitting
32%

Main Issues Identified

Technical Issues

- Decoupling from the monolithic system
- Database migration and data splitting
- Communication among services
- Service orchestration complexity

Main Issues Identified

Effort-Related issues

- Effort estimation and overhead
  - effort at least 20% higher
- Effort required for the DevOps infrastructure
- Effort required for library conversion

Main Issues Identified

Other issues

- People's minds
- ROI achieved in longer time (or never) compared to monolithic systems

Migration Issues

Issues	Entire Dataset		Migration Consultant		Others	
	#	Median	#	Median	#	Median
Monolith decoupling	7	3	/	/	7	3
Database migration and data splitting	6	4	/	/	6	4
Communication among services	4	3.5	2	4	2	3
Effort estimation	2	4	2	4	/	/
DevOps infrastructure requires effort	2	4	/	/	2	4
Library conversion effort	2	4	/	/	2	4
People's minds	2	4	1	4	1	4
Expected long-term return on investment (ROI)	2	3	1	3	1	3

Expected Benefits

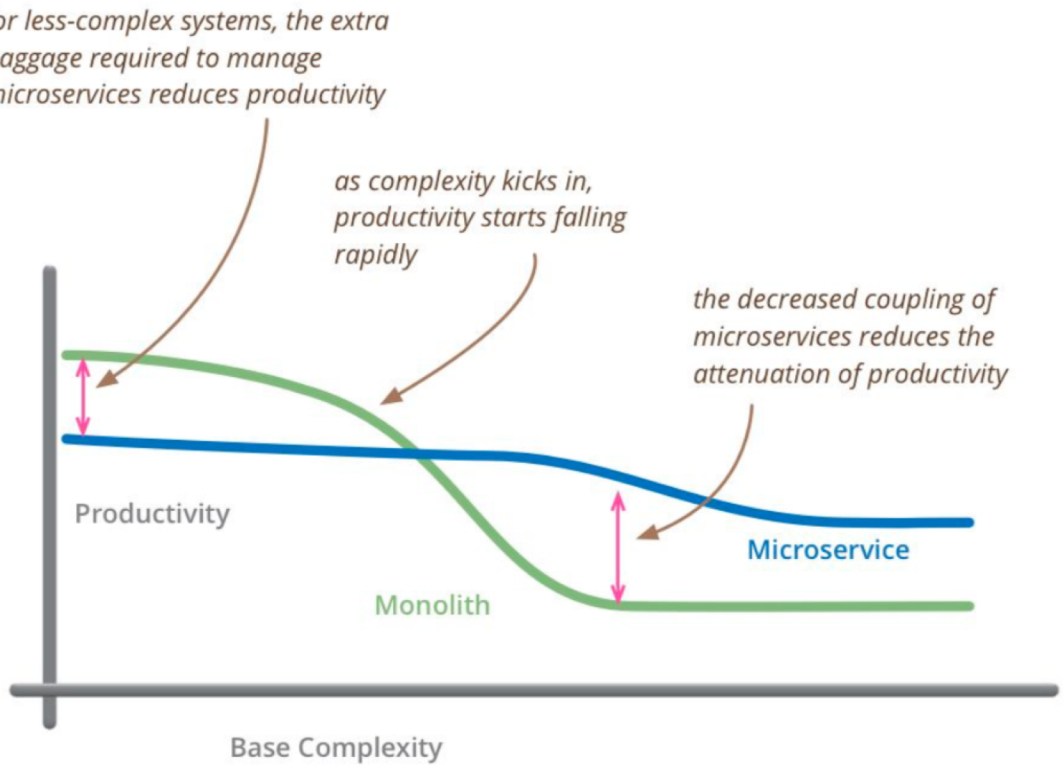
Benefits	Entire Dataset		Migration Consultant		Others	
	#	Median	#	Median	#	Median
Maintainability improvement	9	4	5	4	4	3.5
Scalability improvement	7	2	5	2	2	4
ROI	6	4	/	/	6	4
Architectural complexity reduction	6	3	/	/	6	3
Simplifies distributed work	2	3	2	3	/	/
Performance improvement	2	1	2	1	/	/
Testability	2	3	/	/	2	3
Separation of concerns	2	3	2	3		
Single clear responsibility	2	1	2	1	/	/
Suitability for Scrum	2	3	/	/	2	3
System understandability	1	4	1	4	/	/

The microservices style introduces its own set of (distributed systems related) complexities:

- automated deployment
- continuously monitoring
- dealing with failure
- eventual consistency
- security

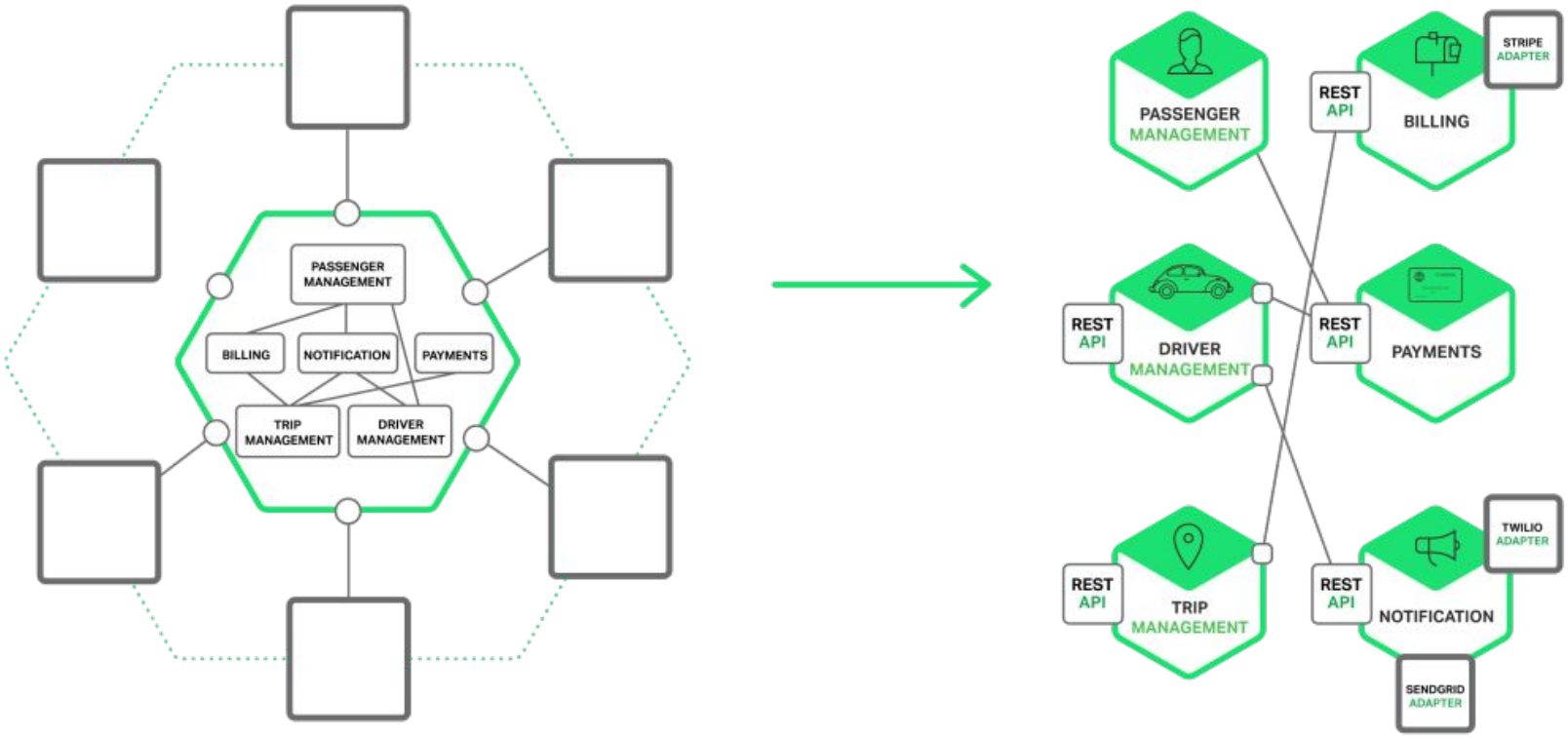
"The majority of software systems should be built as a single monolithic application. Do pay attention to good modularity within that monolith. Don't even consider microservices unless you have a system that's too complex to manage as a monolith."

-- Martin Fowler

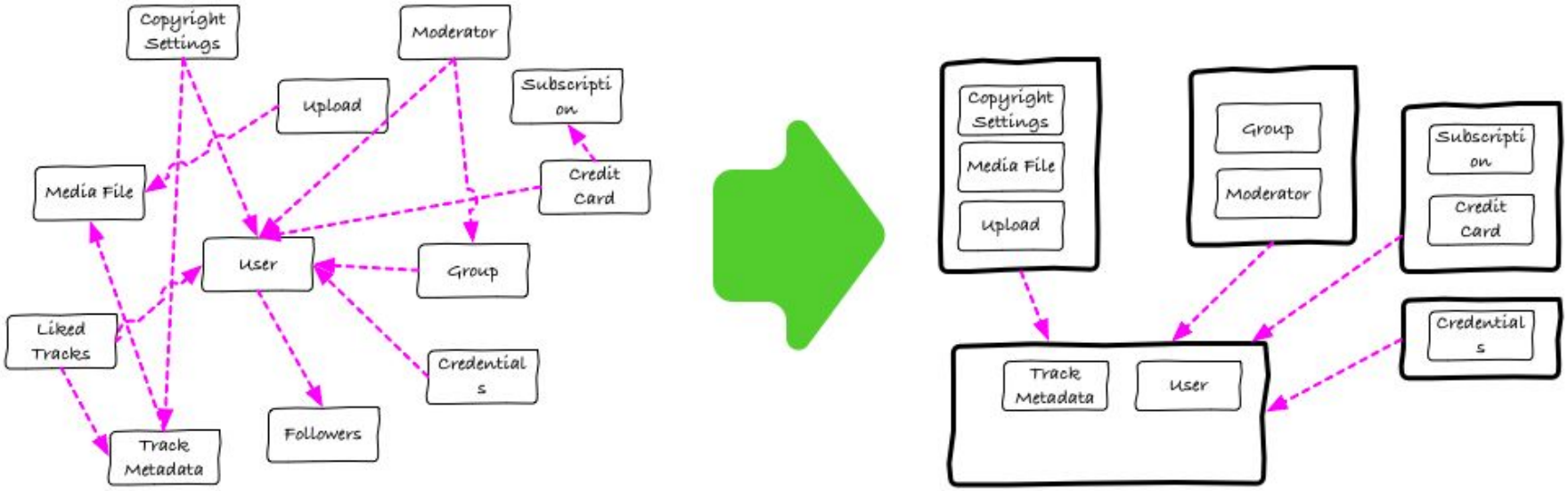


source: https://martinfowler.com/bliki/MicroservicePremium.html

Microservices Example: Uber



Microservices Example: SoundClou



Microservices Patterns

Architectural Patterns for Microservices: a Systematic Mapping Study

Davide Taibi[1] and Valentina Lenarduzzi [1] and Claus Pahl[2]
[1]*Tampere University of Technology, Tampere, Finland*
[2]*Free 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 style patterns and the principles that guide their definition. We conducted a systematic mapping study in order to identify reported usage of microservices and based on these use cases 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 systems to microservices. However companies 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 *architectural 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 *De*velopment and *Op*erations 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

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

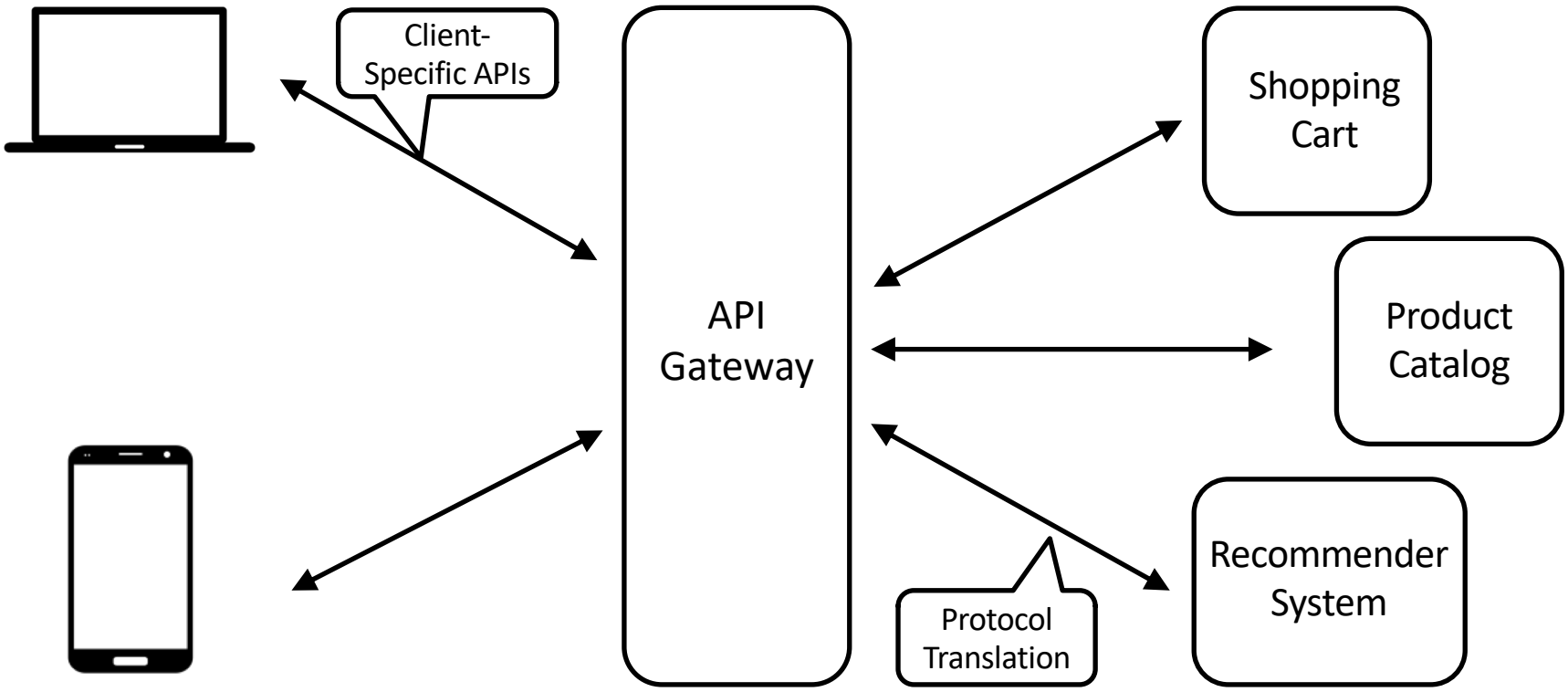
Pattern Categories

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.


Client-
Specific APIs
API
Gateway
Shopping
Cart
Product
Catalog
Recommender
System
Protocol
Translation

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

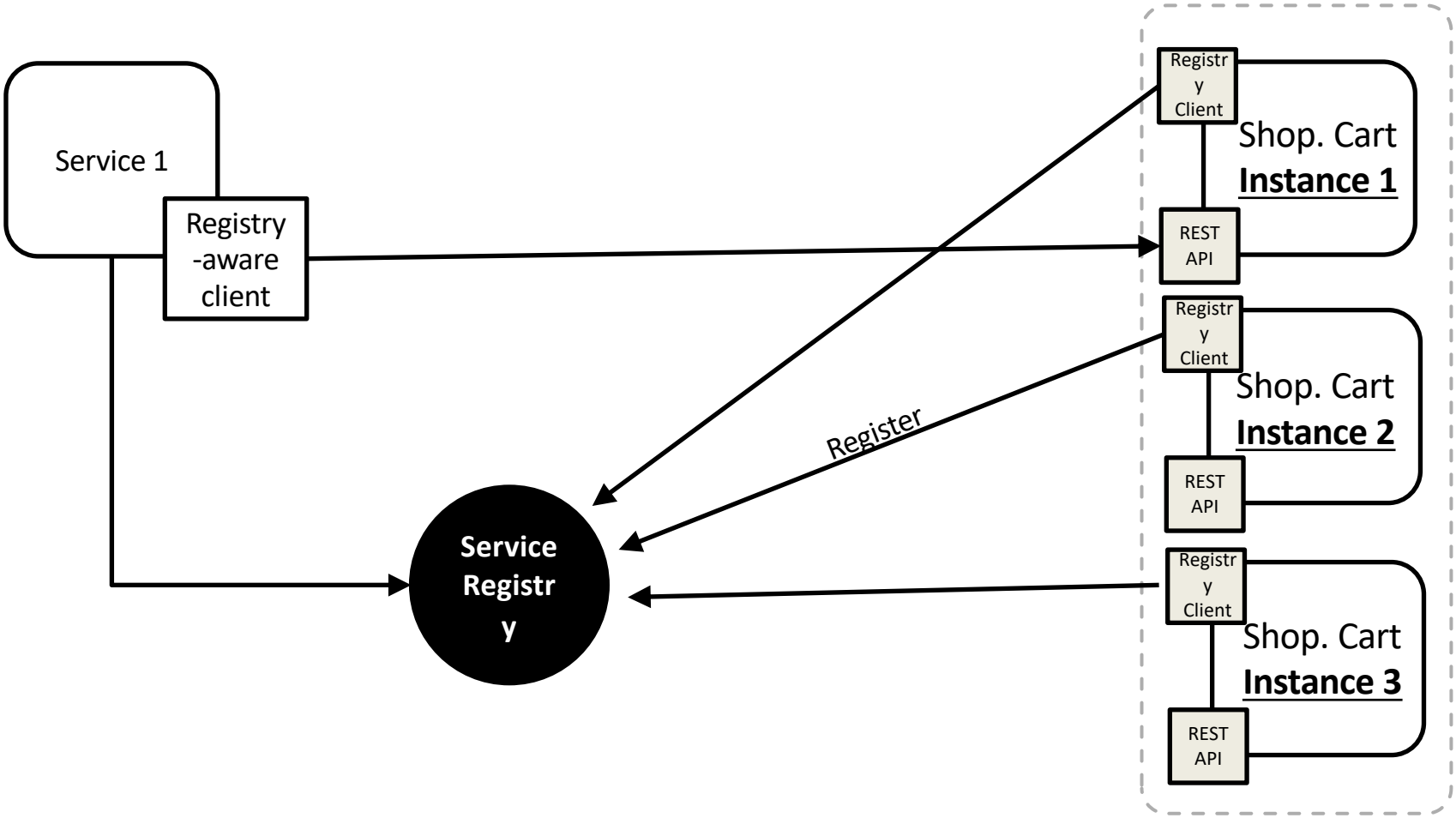
Properties:

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.

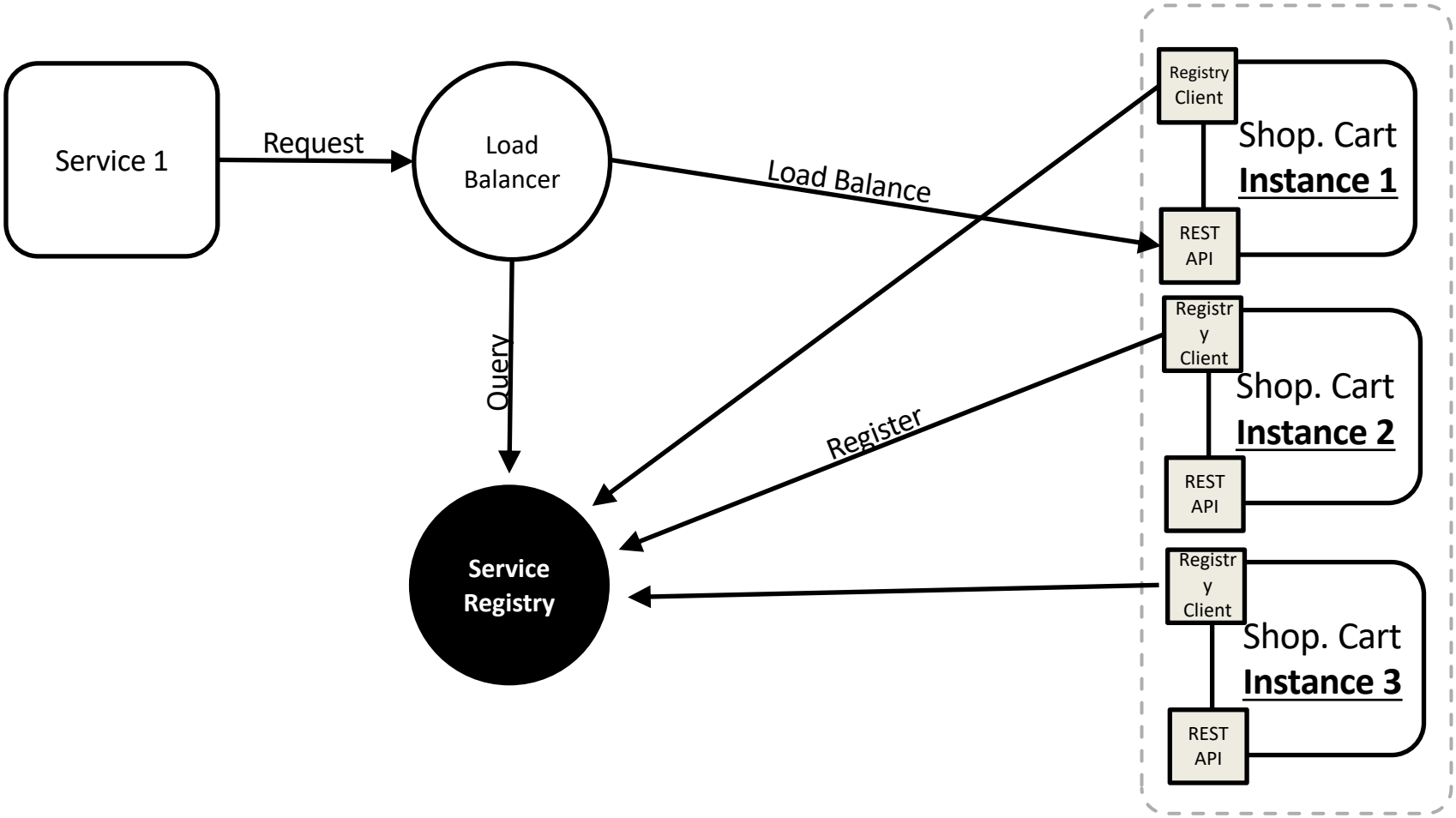
Moreover, the service discovery can also serve as load balancer of the requests. The client then uses a load-balancing algorithm to select one of the available service instances and makes a request.

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	-Increased maintainability -Can use different languages -Flexibility -Reuse -Physical isolation -Self-healing	-Development/testing complexity -Implementation effort -Network-related issue
	API Gateway	-Extension easiness -Market-centric Architecture -Backward compatibility	-Potential bottleneck -Development complexity -Scalability
	Service Registry	-Increased maintainability -Communic., developm., migration -Software understandability -Failure safety	-Interface design must be fixed -Service registry complexity -Reuse -Distributed system complexity
	Hybrid	-Migration easiness -Learning curve	-SOA/ESB integration issues
Deploy-ment	Multiple service per host	-Scalability -Performance	
	Single service per host	-Service isolation	-Scalability -Performance
Data Storage	DB per Service	-Scalability -Independent development -Security mechanism	-Data needs to be splitted -Data consistency
	DB Cluster	-Scalability -Implementation easiness	-Increase complexity -Failure risks
	Shared DB server	-Migration easiness -Data consistency	-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 possibily 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

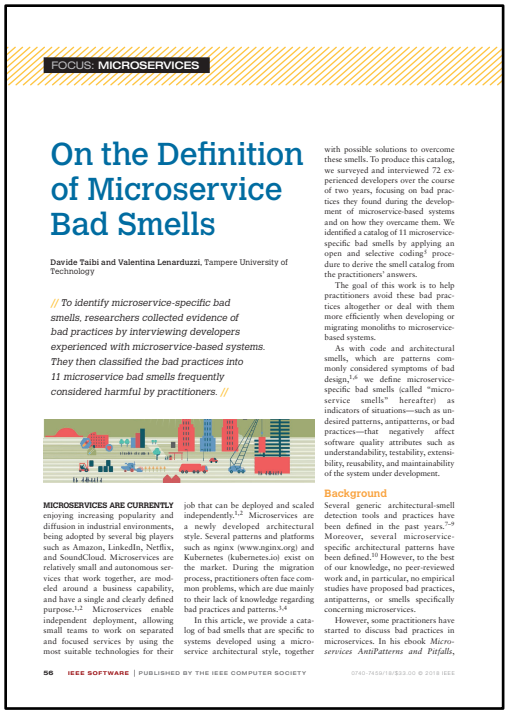
for **orchestration/coordination** and **storage**

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 diffusion in industrial environments, being adopted by several big players such as Amazon, LinkedIn, Netflix, and SoundCloud. Microservices are relatively small and autonomous services that work together, are modeled around a business capability, and have a single and clearly defined purpose.[1,2] Microservices enable independent deployment, 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.[1,2] Microservices are a newly developed architectural style. Several patterns and platforms such as nginx (www.nginx.org) and Kubernetes (kubernetes.io) exist on the market. During the migration process, practitioners often face common problems, which are due mainly to their lack of knowledge regarding bad practices and patterns.[3,4]

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. To produce this catalog, we surveyed and interviewed 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 coding[5] procedure to derive the smell catalog from the practitioners' answers.

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

As with code and architectural smells, which are patterns commonly considered symptoms of bad design,[1,6] we define microservice-specific bad smells (called "microservice smells" hereafter) as indicators of situations—such as undesired patterns, antipatterns, or bad practices—that negatively affect software quality attributes such as understandability, testability, extensibility, reusability, and maintainability of the system under development.

Background

Several generic architectural-smell detection tools and practices have been defined in the past years,[7–9] Moreover, several microservice-specific architectural patterns have been defined.[10] However, to the best of our knowledge, no peer-reviewed 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*,

0740-7459/18/$33.00 © 2018 IEEE

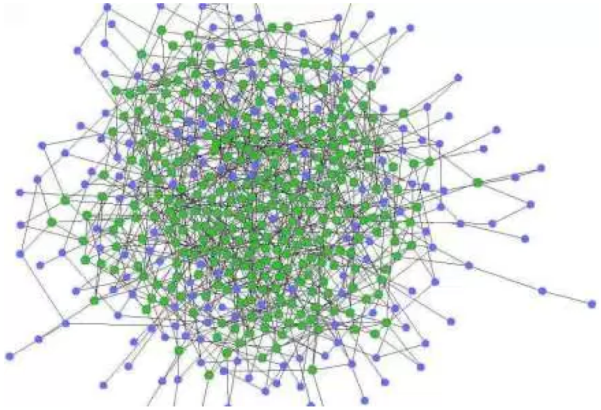
IEEE Software May/June 2018

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.

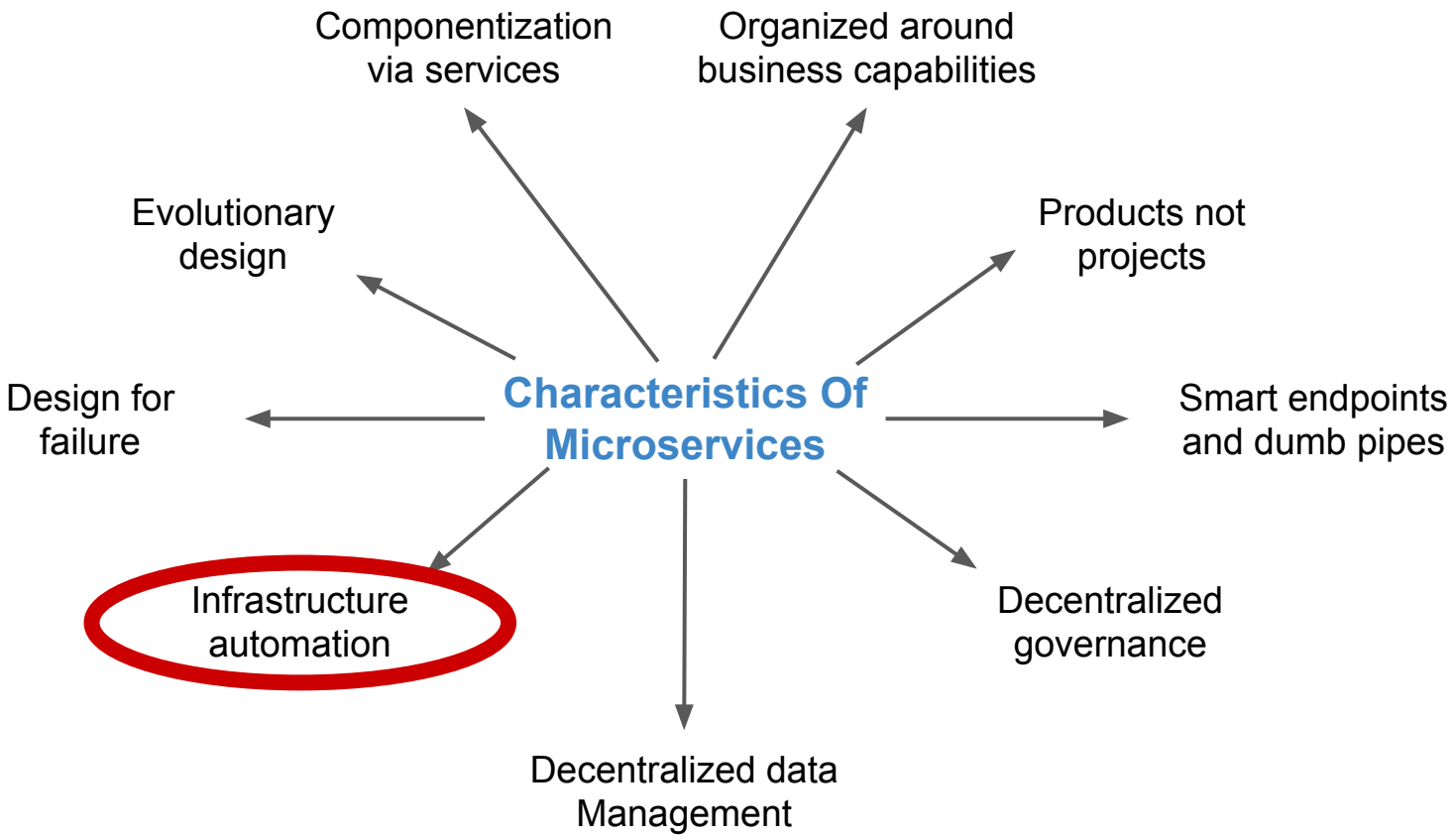
Microservice Bad Smells

Microservice Greedy	**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. **Det.:** Microservices with very limited functionalities (e.g., a microservice serving only one static HTML page).	**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. **S:** Carefully consider whether the new microservice is needed.
Not Having an API Gateway	**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. **Det.:** Direct communication between microservices.	**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. **S:** Apply the API Gateway pattern[2] to reduce the communication complexity between microservices.
Shared Libraries	**Desc./Det.:** Shared libraries between different microservices are used.	**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. **S:** Two possible solutions are to 1. accept the redundancy to increase dependency among teams, or 2. extract the library to a new shared service that can be deployed and developed independently by the connected microservices.
Shared Persistency	**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:** This smell highly couples the microservices connected to the same data, reducing team and service independence. **S:** Three possible solutions are to 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, or 3. use a private database schema for each service.

Microservice Bad Smells

Too Many Standards	**Desc./Det.:** Different development languages, protocols, frameworks, etc. are used. Also proposed as the Lust and Gluttony bad practices.[16]	**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. **S:** Carefully consider the adoption of different standards for different microservices, without following the latest hype.
Wrong Cuts	**Desc.:** Microservices are split on the basis of technical layers (presentation, business, and data layers) instead of business capabilities.	**P:** The wrong separation of concerns and increased data-splitting complexity can occur. **S:** Perform a clear analysis of business processes and the need for resources.

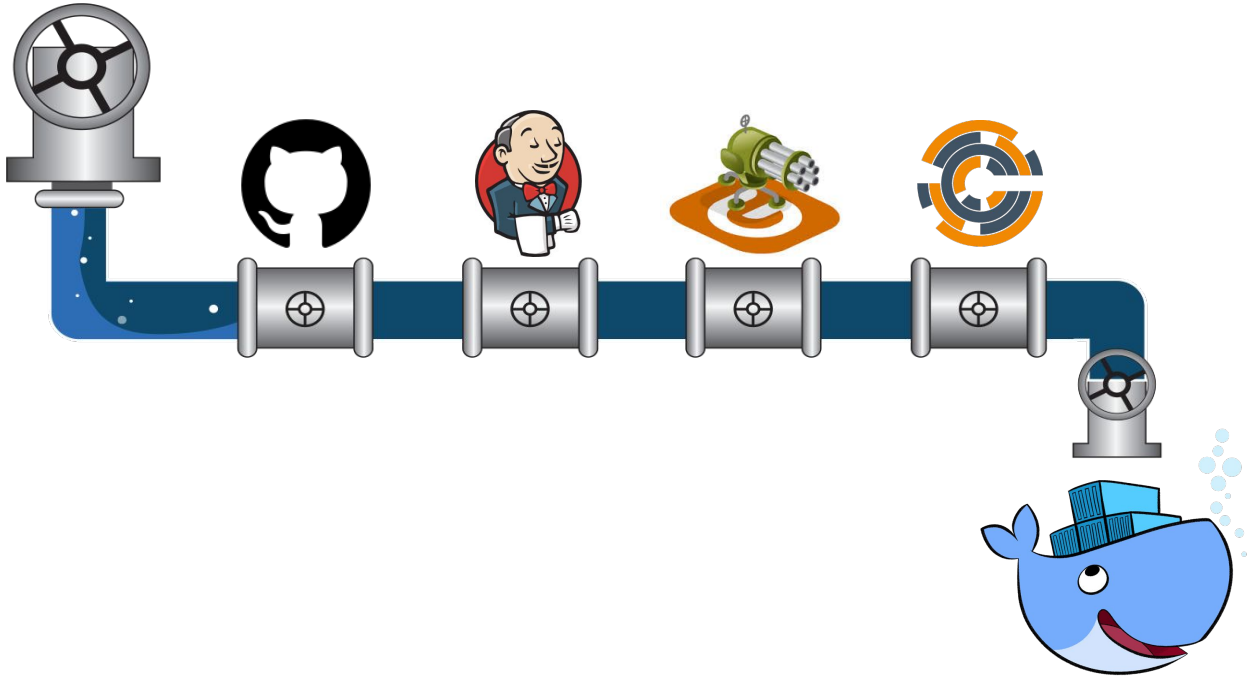
Microservices Characteristics



source: https://martinfowler.com/articles/microservices.html

Infrastructure Automation

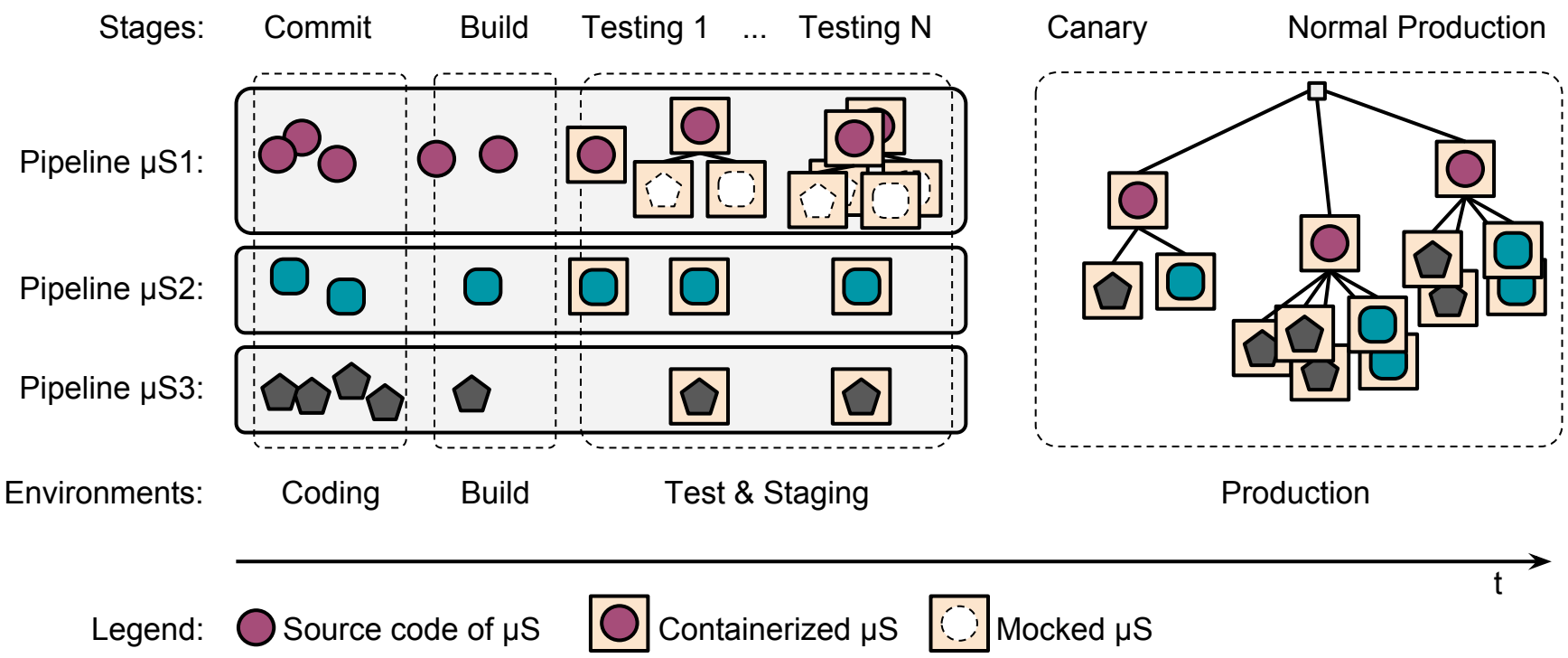
Enables the creation of continuous delivery (CD) microservices pipelines based on DevOps and other agile practices



source: https://www.neudesic.com/services/continuous-delivery-devops/ (adapted)

Infrastructure Automation (2)

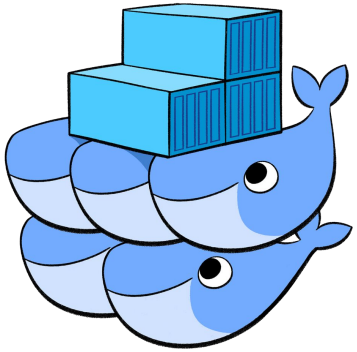
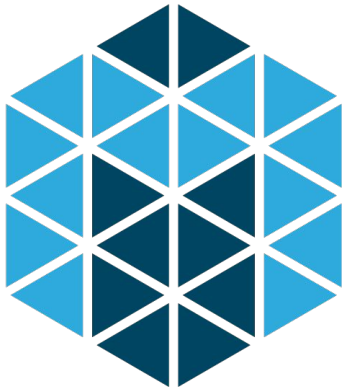
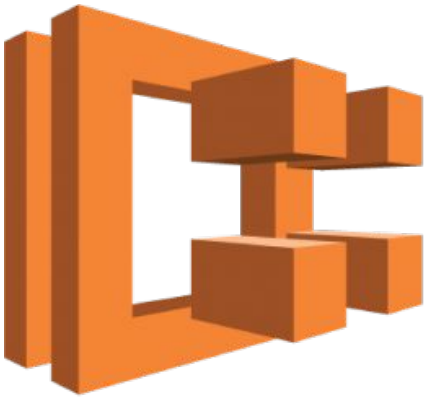
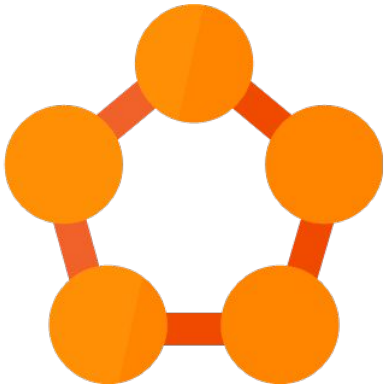
A typical CD strategy for microservices would look like this:



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

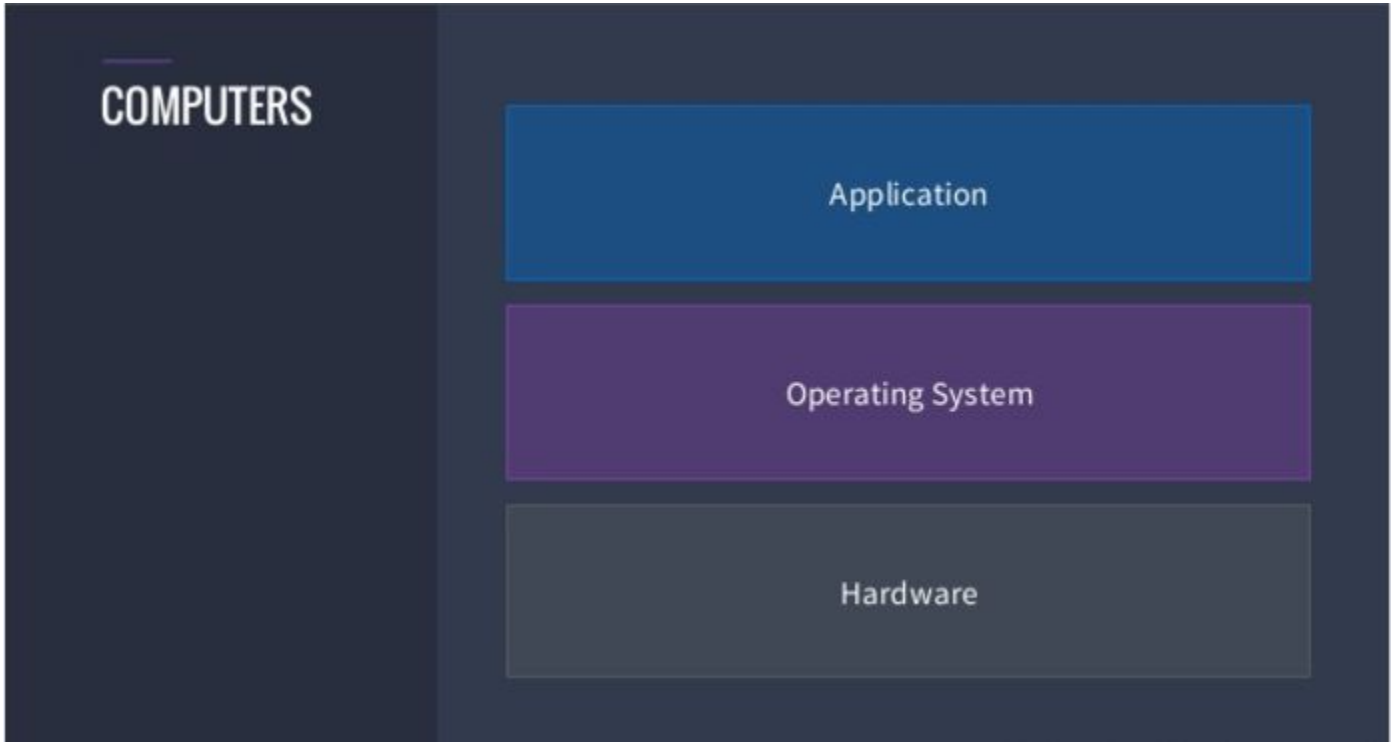
Infrastructure Automation (3)

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

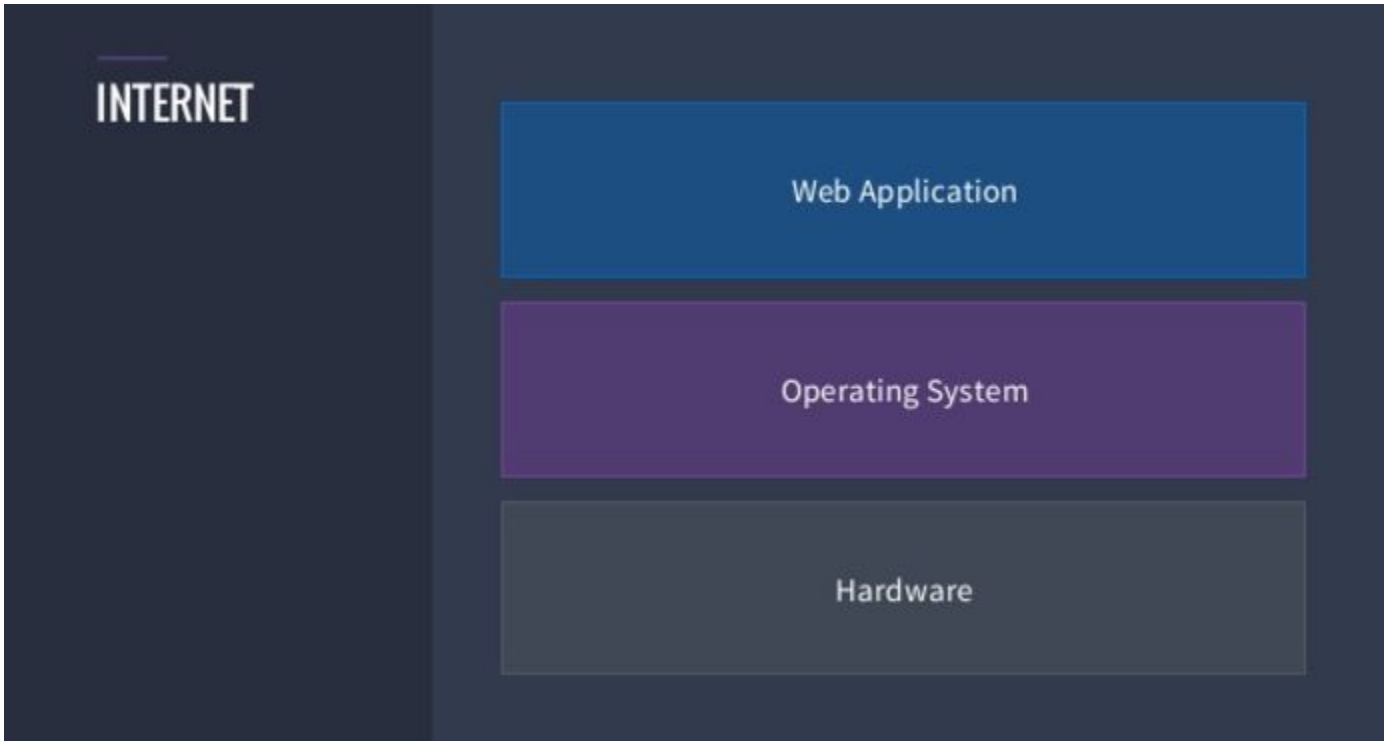
Historical perspective



source: https://www.slideshare.net/KarlIsenberg/container-orchestration-wars

Container Orchestration (2)

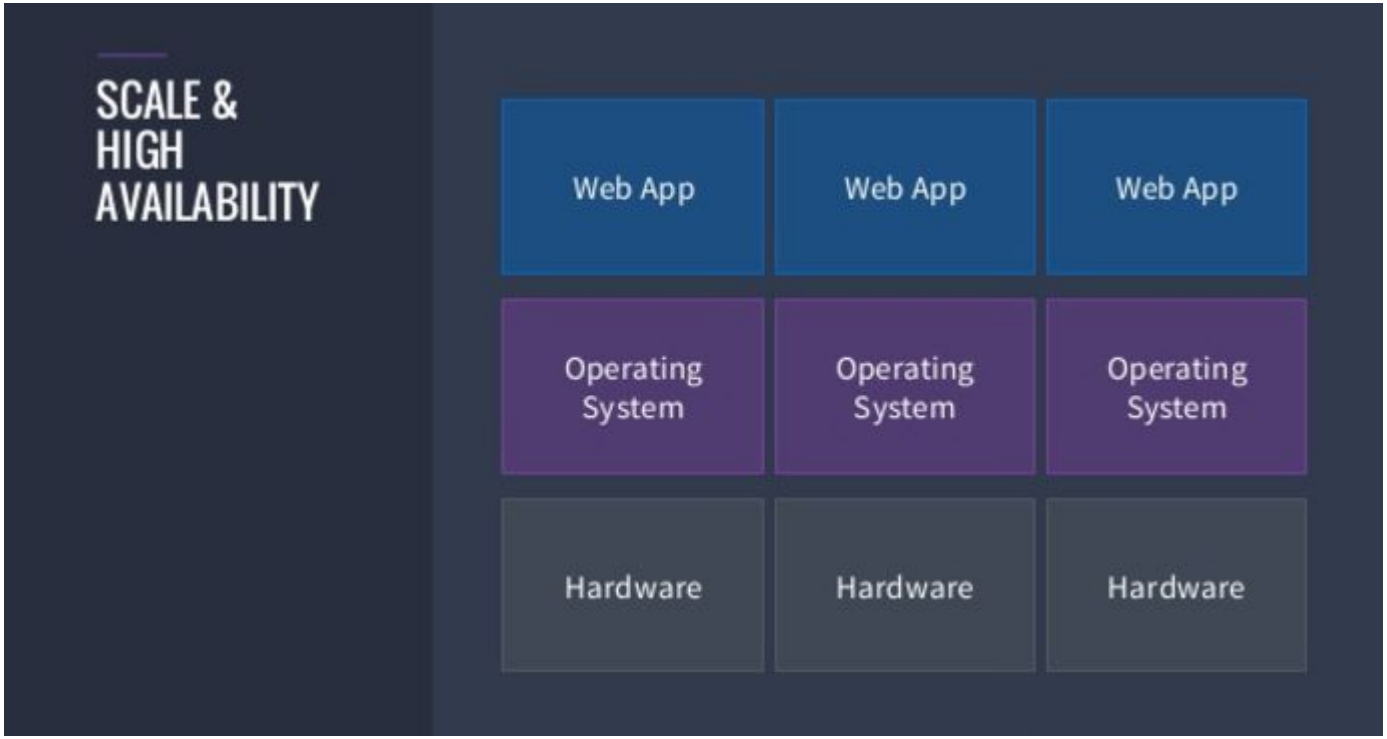
Historical perspective



source: https://www.slideshare.net/KarlIsenberg/container-orchestration-wars

Container Orchestration (3)

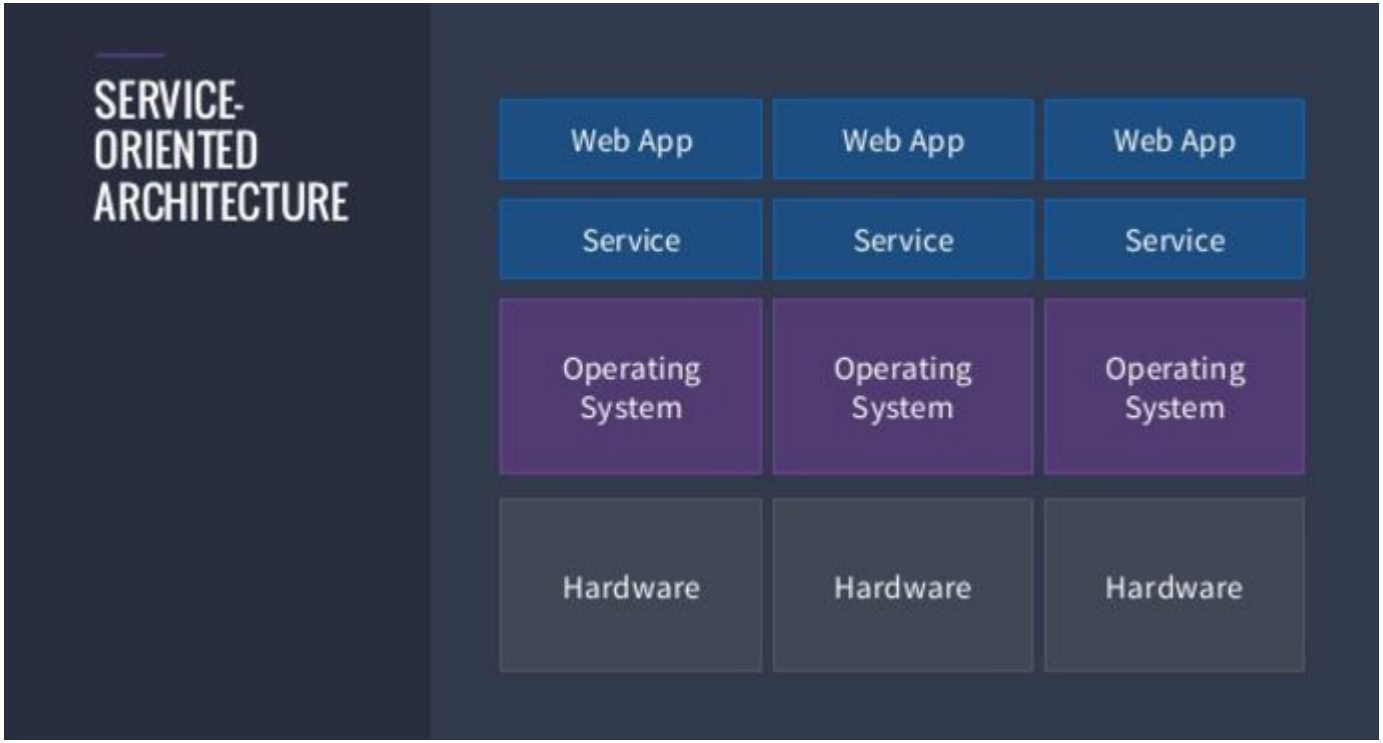
Historical perspective



source: https://www.slideshare.net/KarlIsenberg/container-orchestration-wars

Container Orchestration (4)

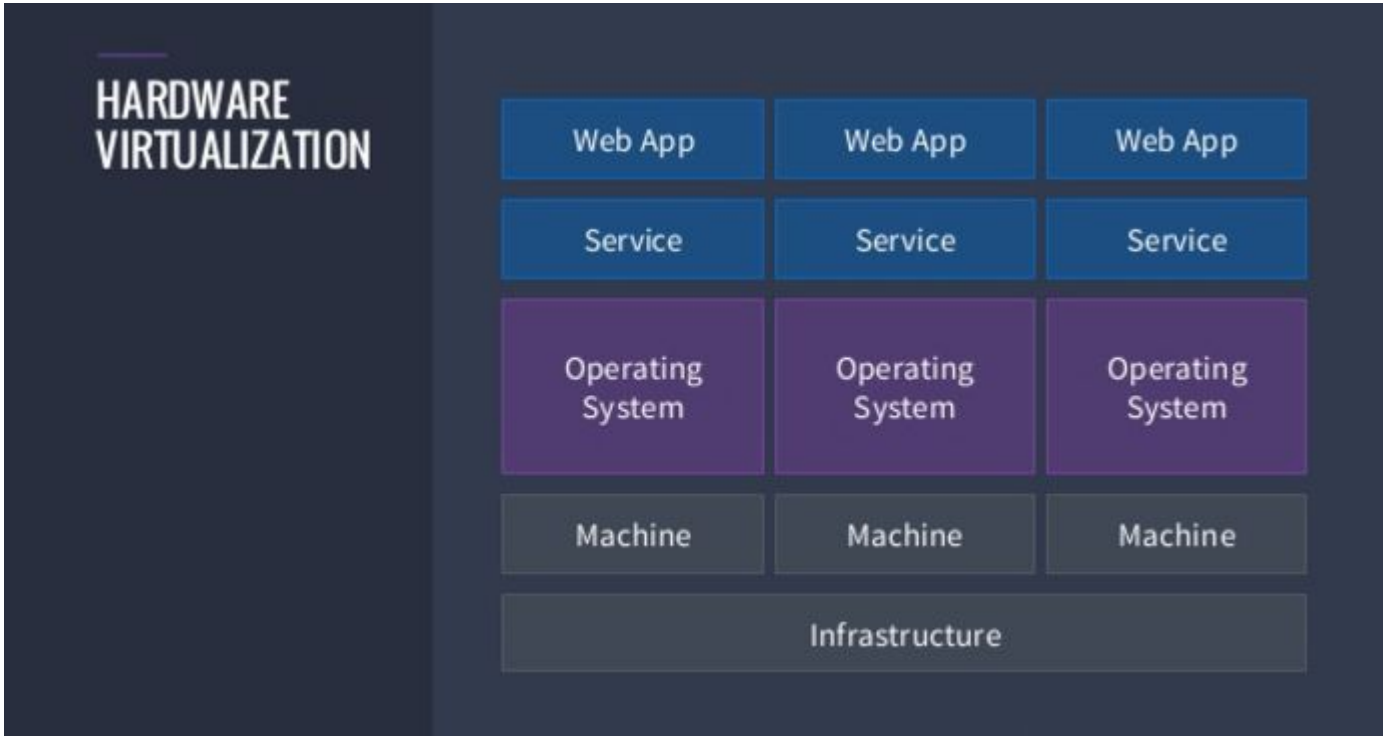
Historical perspective



source: https://www.slideshare.net/KarlIsenberg/container-orchestration-wars

Container Orchestration (5)

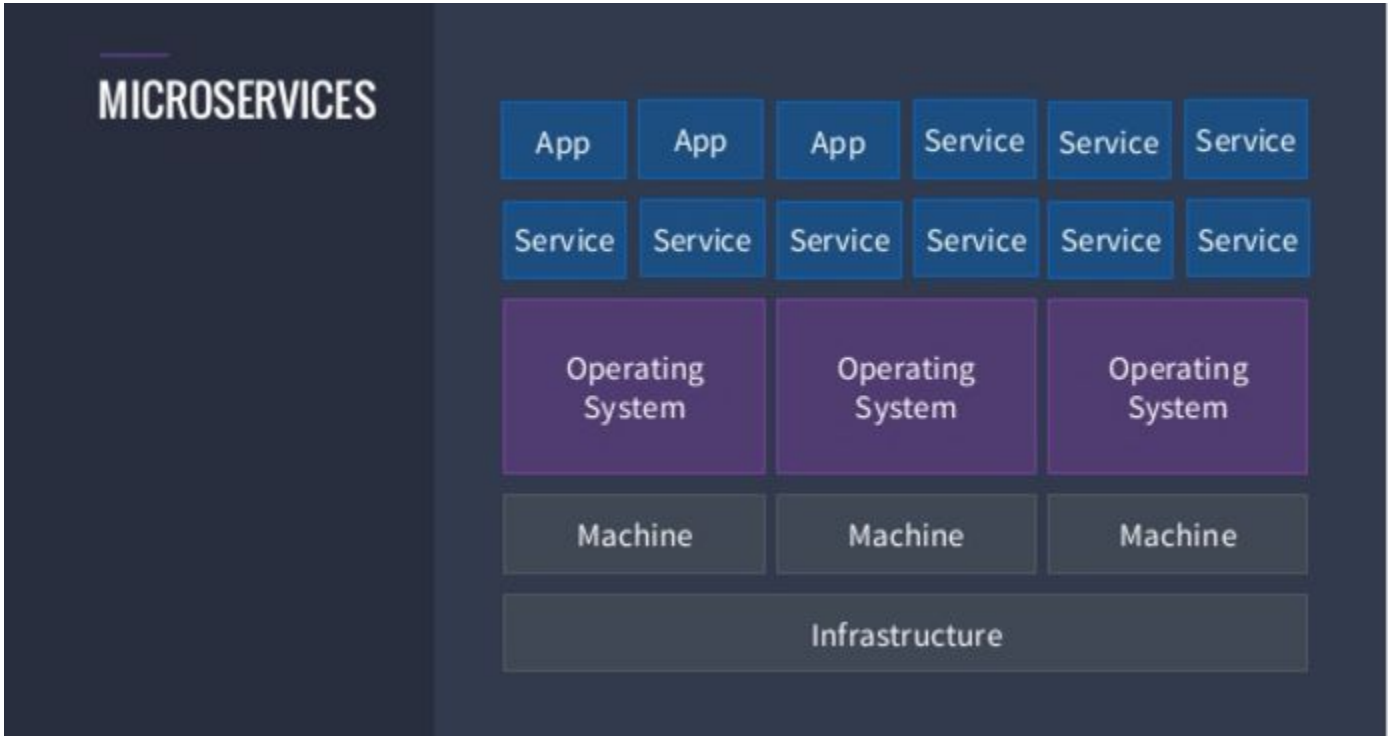
Historical perspective



source: https://www.slideshare.net/KarlIsenberg/container-orchestration-wars

Container Orchestration (6)

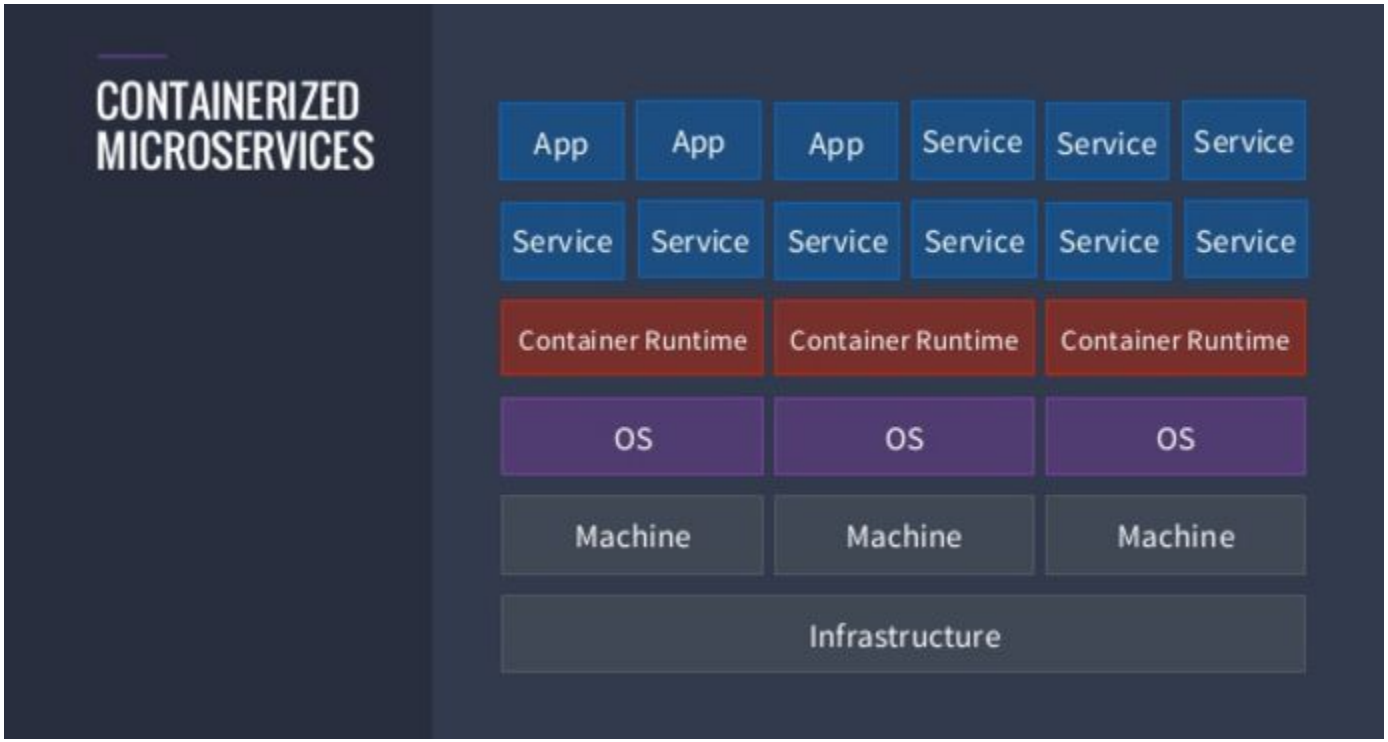
Historical perspective



source: https://www.slideshare.net/KarlIsenberg/container-orchestration-wars

Container Orchestration (7)

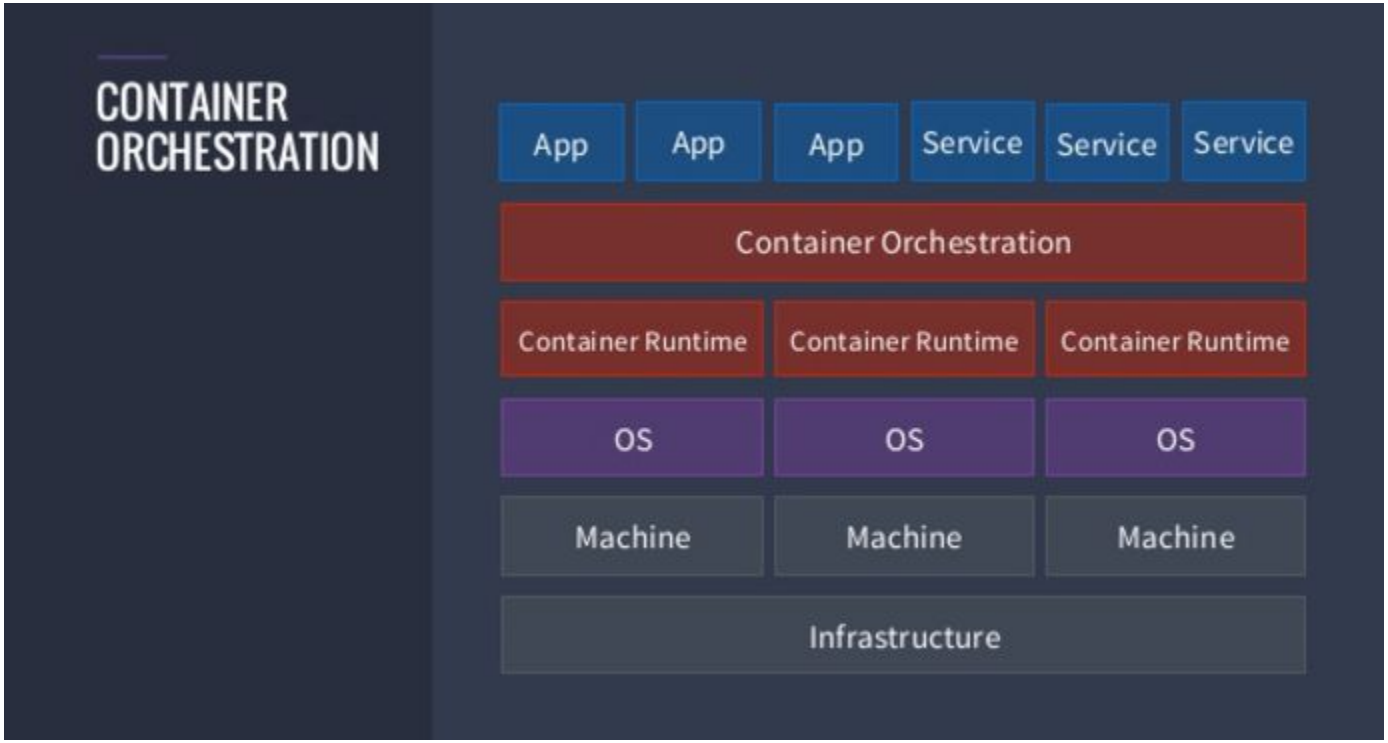
Historical perspective



source: https://www.slideshare.net/KarlIsenberg/container-orchestration-wars

Container Orchestration (8)

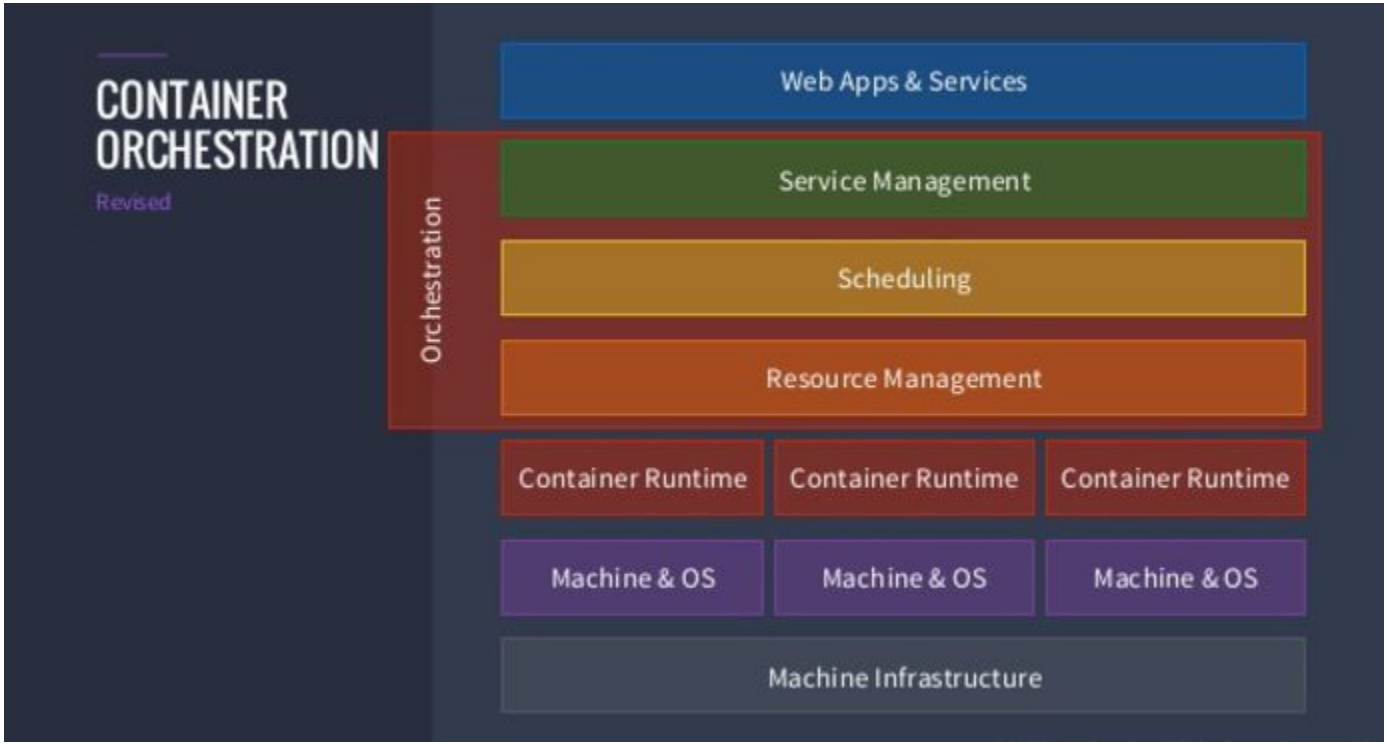
Historical perspective



source: https://www.slideshare.net/KarlIsenberg/container-orchestration-wars

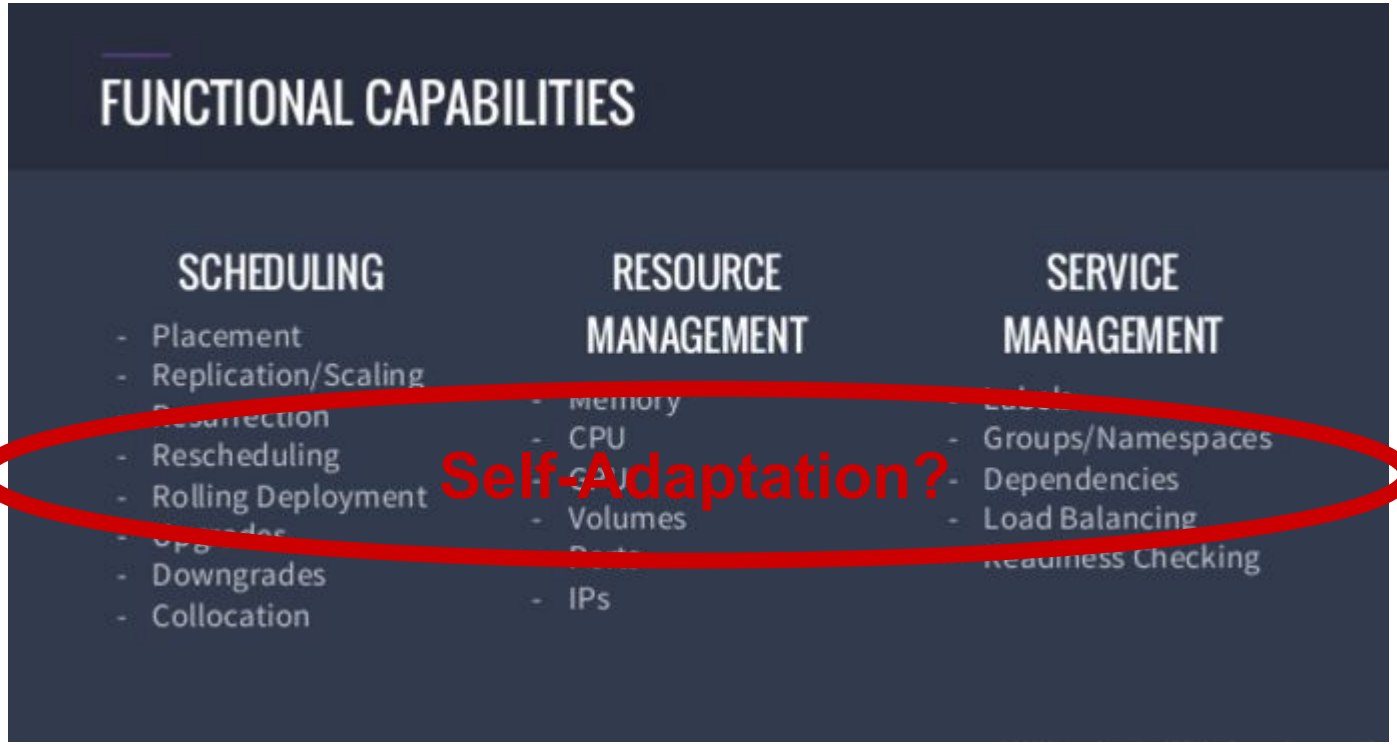
Container Orchestration (9)

Historical perspective



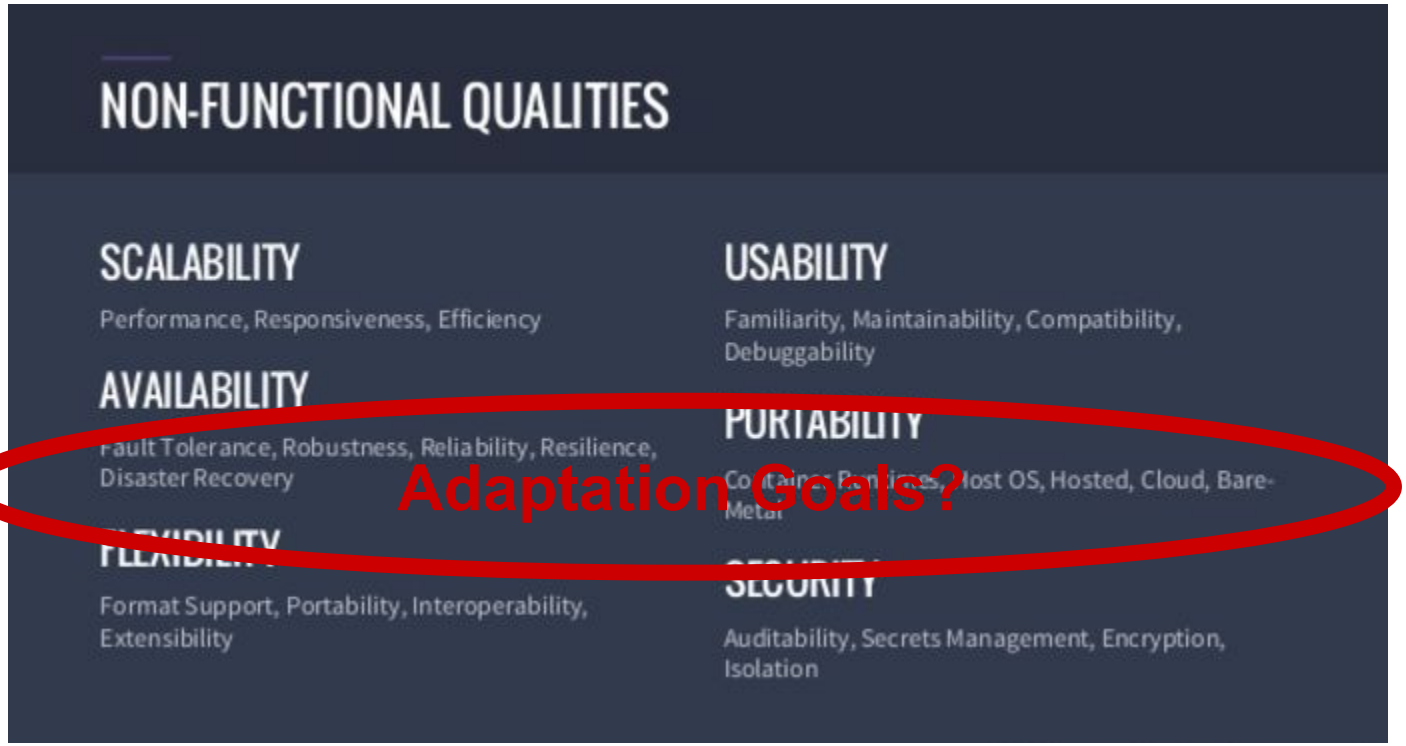
source: https://www.slideshare.net/KarlIsenberg/container-orchestration-wars

Container Orchestration (10)



source: https://www.slideshare.net/KarlIsenberg/container-orchestration-wars

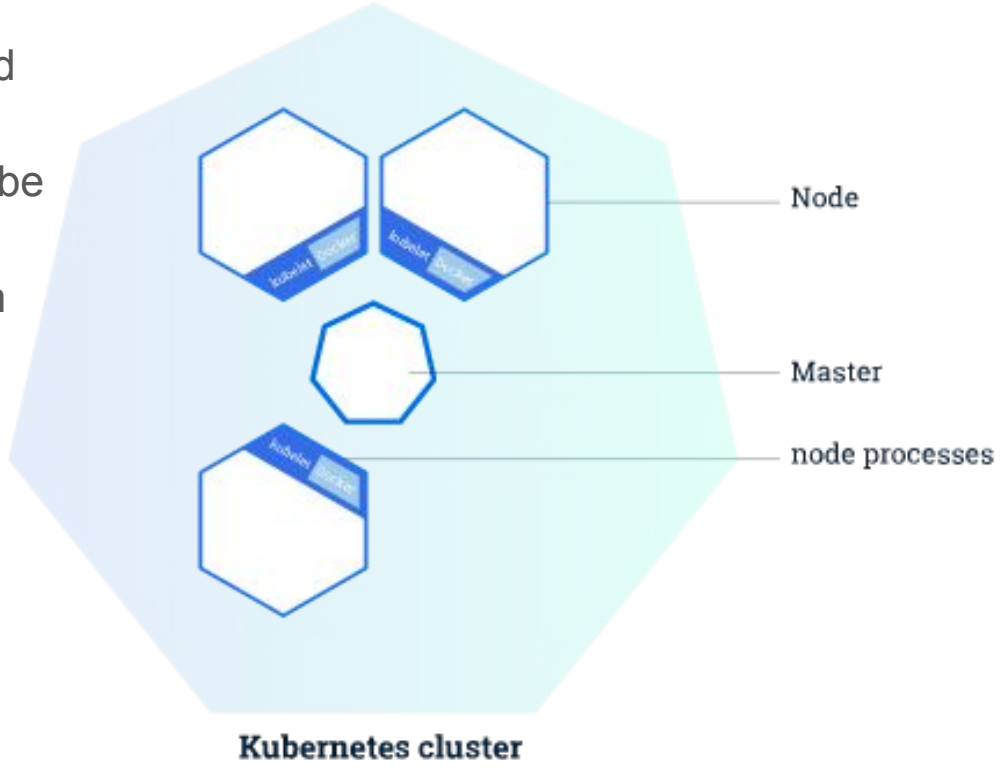
Container Orchestration (11)



source: https://www.slideshare.net/KarlIsenberg/container-orchestration-wars

Example: Kubernetes

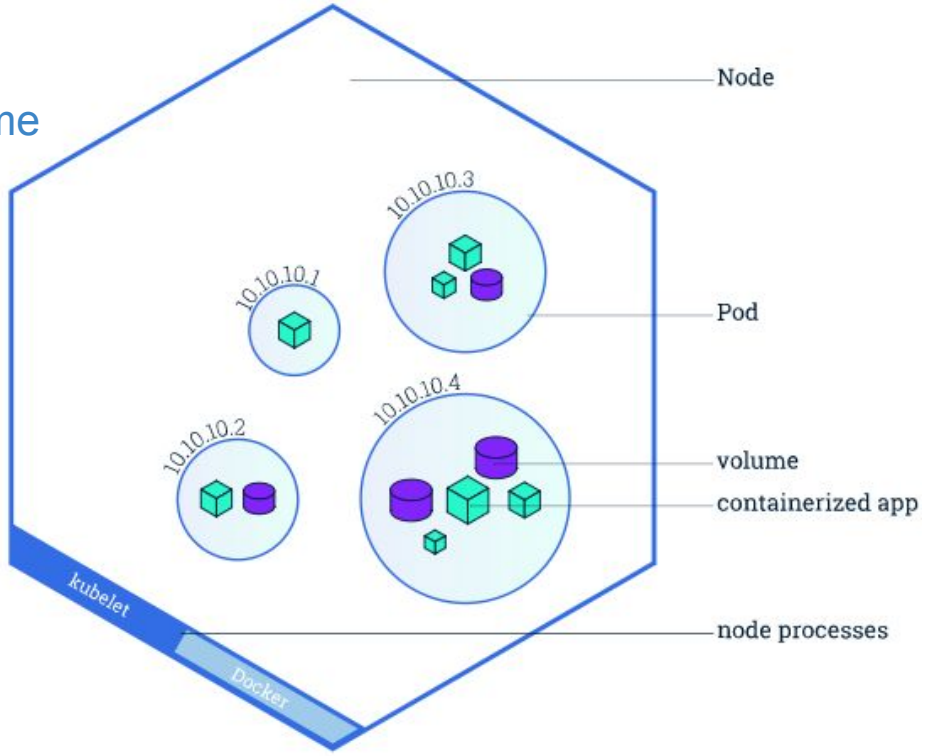
- A Kubernetes cluster is composed of a (replicated) master and multiple nodes
- The master coordinates all activities in the cluster, including scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates
- A node is a worker machine and may be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



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

Example: Kubernetes (2)

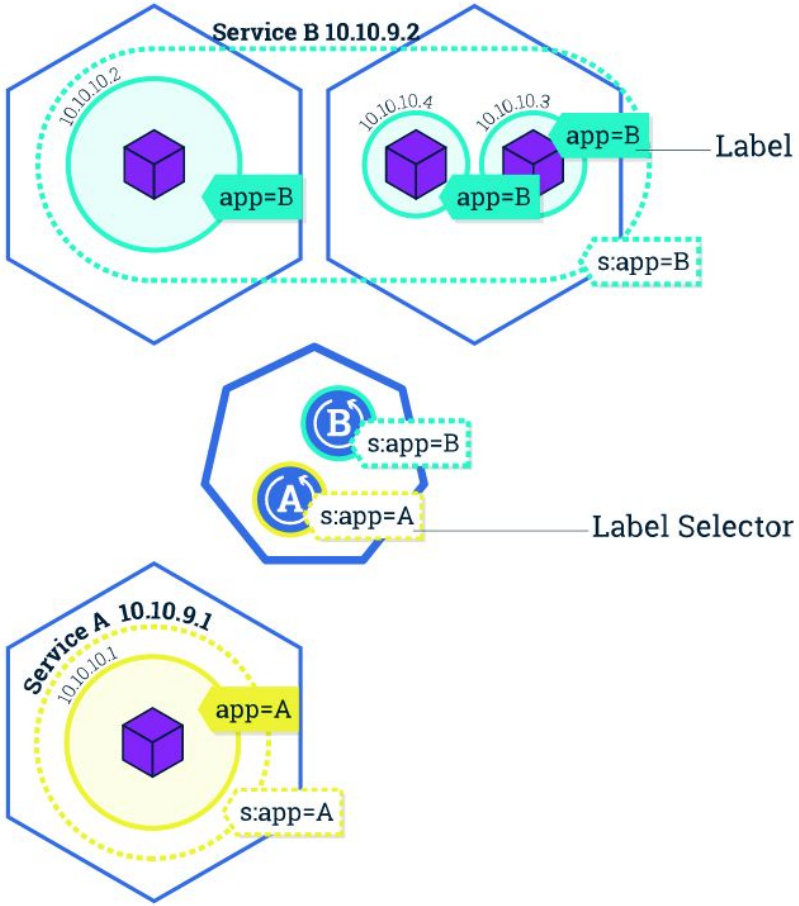
- A node can have multiple pods, which are automatically scheduled across the nodes in the cluster by the master
- A pod is a Kubernetes abstraction that represents a group of one or more application containers that are guaranteed to be co-located on the same node and can share resources (e.g, IP address,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



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

Example: Kubernetes (3)

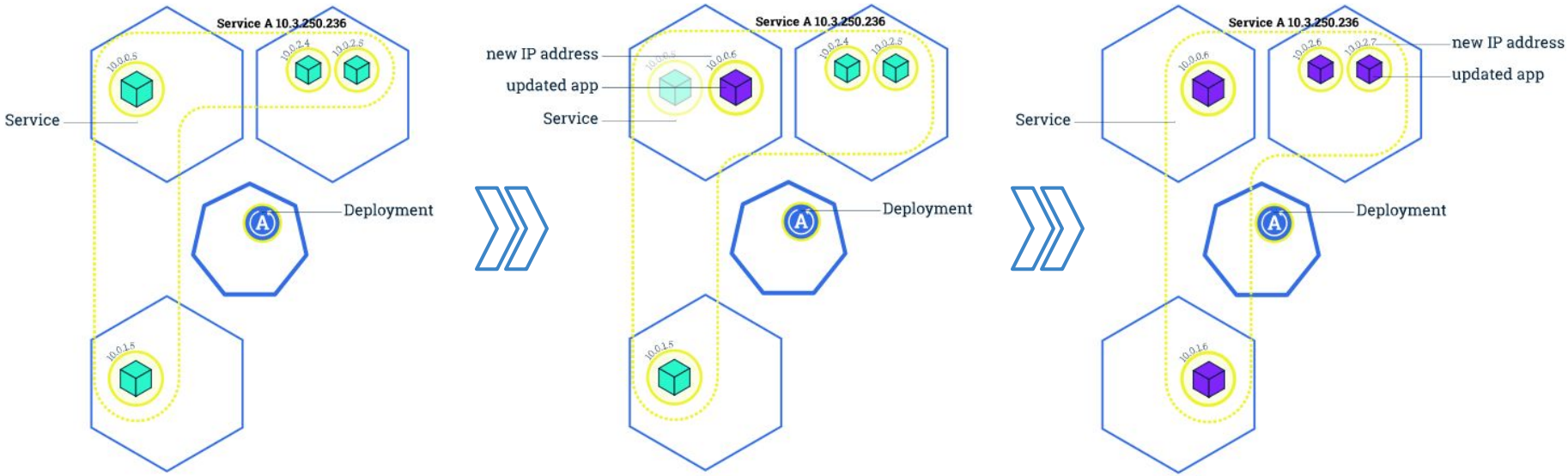
- A service is a set of pods that work together, such as one tier of a multi-tier application
- A deployment is special type of controller used to maintain a desired state for a service or a given set of pods
- Nodes and pods can be logically organized by assigning one or more labels to them. This allows 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)

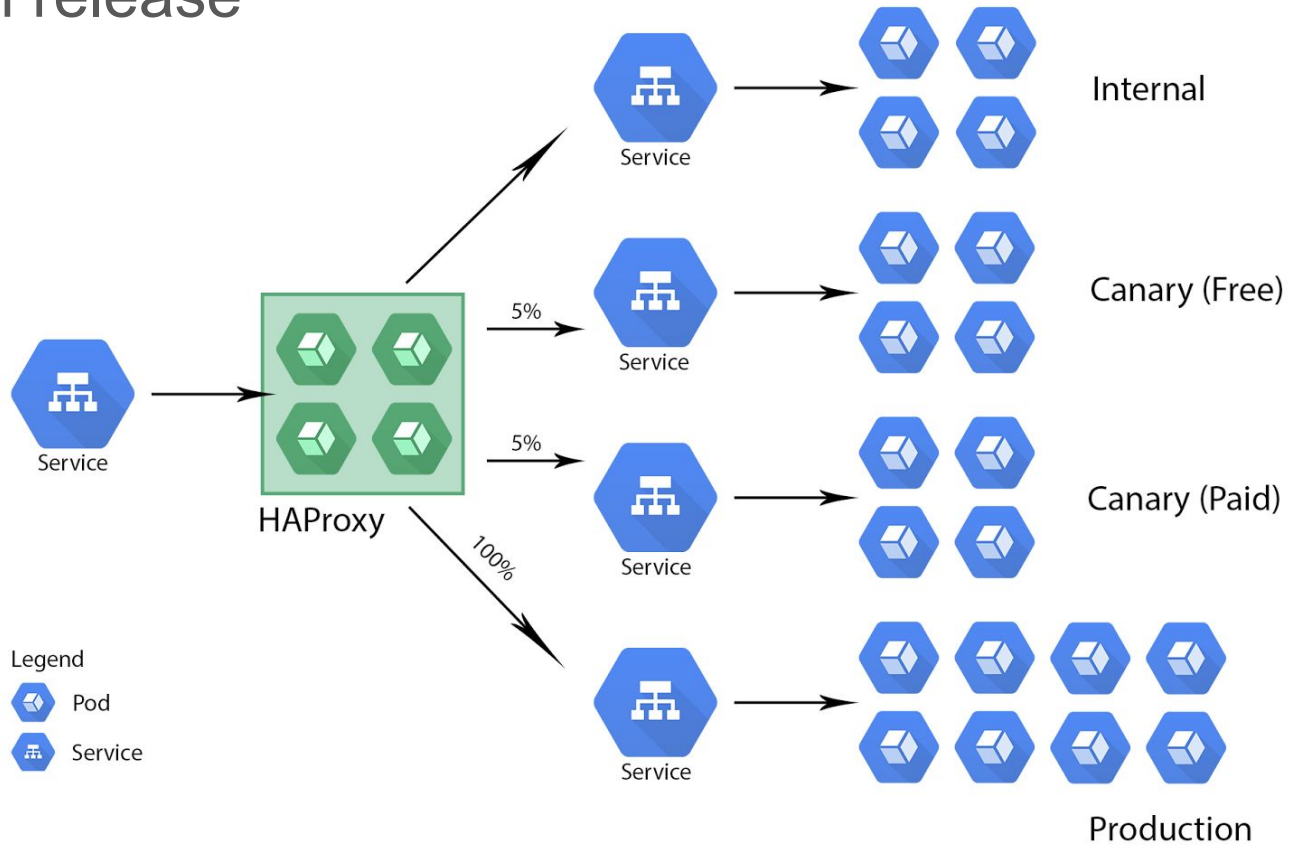
- A rolling update (or rollout) allow a deployment update to take place with zero downtime by incrementally replacing pod instances with new (updated) ones
- Kubernetes's default rollout algorithm looks like this:
    1. Create a new replication controller with the updated configuration
    2. Increase/decrease 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



source: http://blog.kubernetes.io/2017/04/multi-stage-canary-deployments-with-kubernetes-in-the-cloud-onprem.html

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

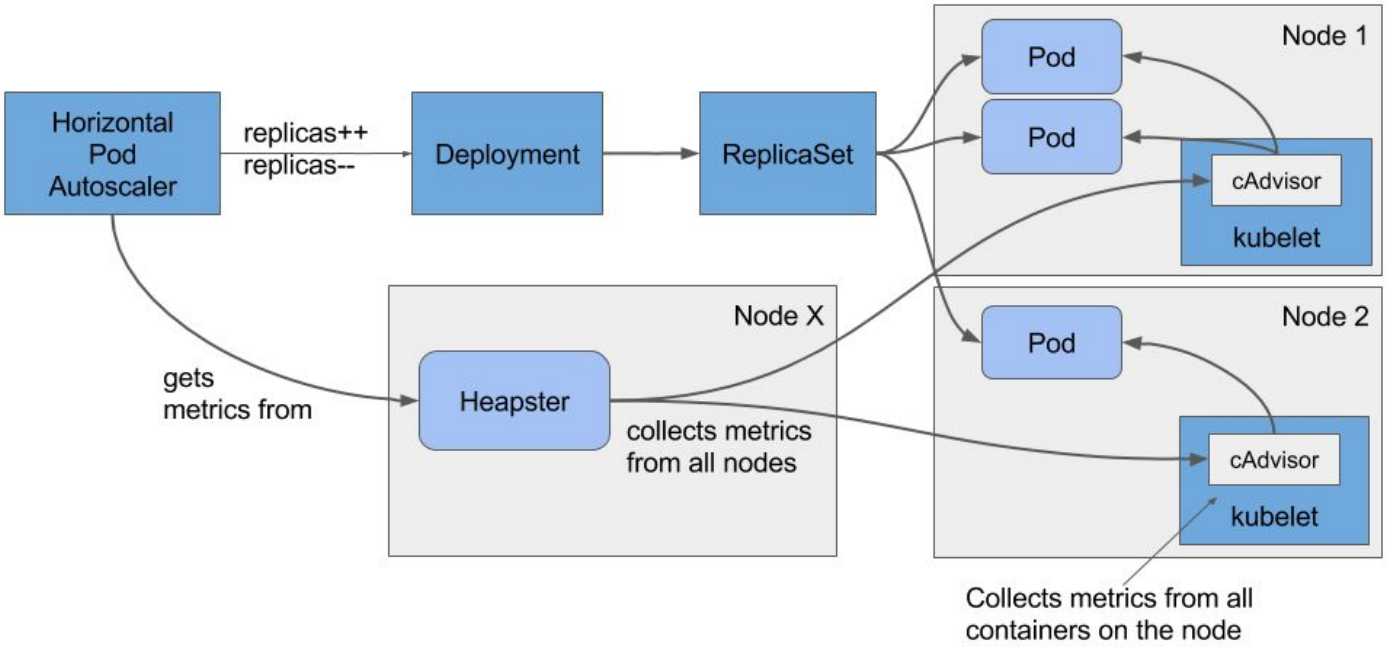
Some controller extensions available in Kubernetes:

- **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

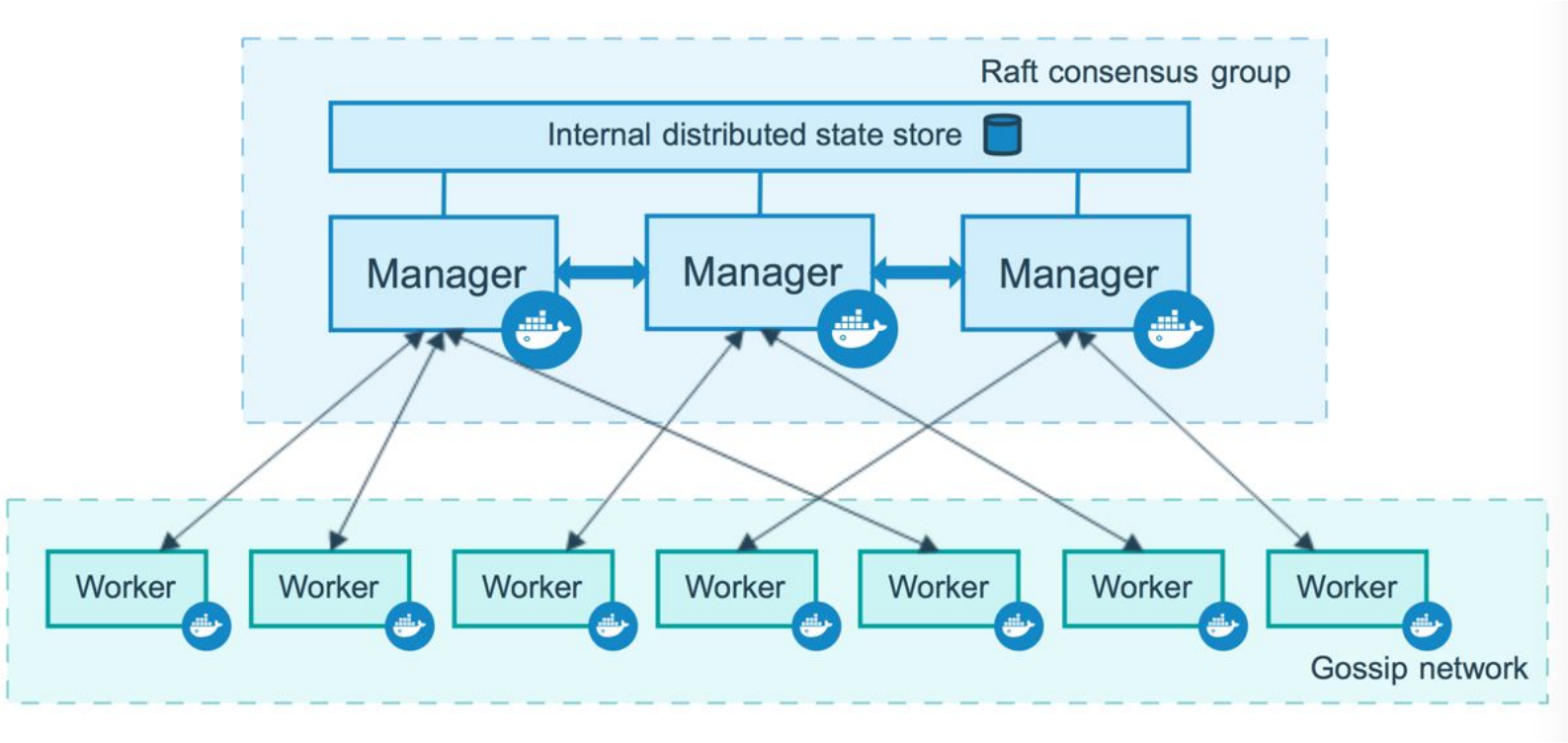


source: https://medium.com/@marko.luksa/

Example: Docker Swarm

Native container orchestration solution for Docker

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

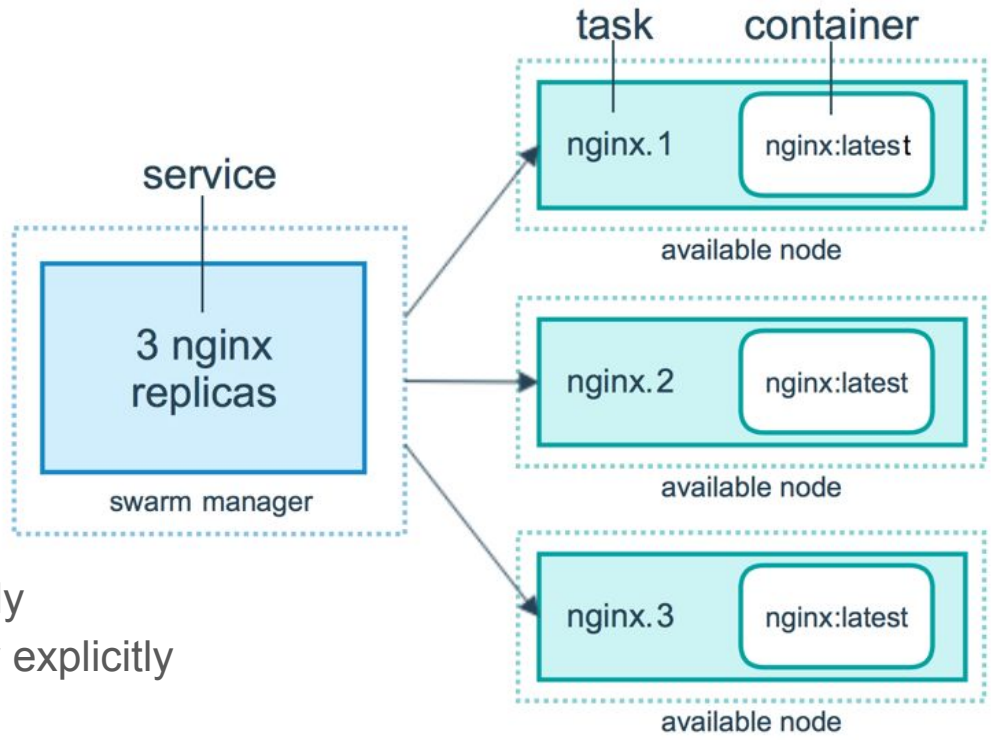


Source: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/

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

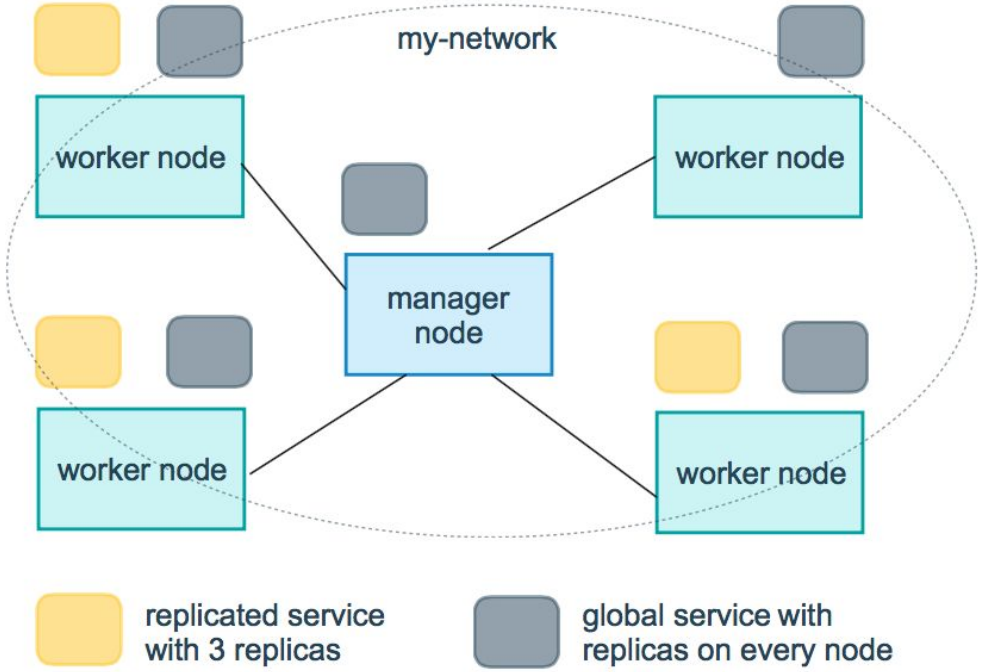


Source: https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/

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



Source: https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/

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. Microcervices. 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