Microservices

A **microservice** architecture – a variant of the service-oriented architecture (SOA) structural style – arranges an application as a collection of loosely-coupled services. In a microservices architecture, services are fine-grained and the protocols are lightweight. The goal is that teams can bring their services to life independent of others. Loose coupling reduces all types of dependencies and the complexities around it, as service developers do not need to care about the users of the service, they do not force their changes onto users of the service. Therefore it allows organizations developing software to grow fast, and big, as well as use off the shelf services easier. Communication requirements are less. But it comes at a cost to maintain the decoupling. Interfaces need to be designed carefully and treated as a public API. Techniques like having multiple interfaces on the same service, or multiple versions of the same service, to not break existing users code.

Contents

Introduction

There is no single definition for microservices. A consensus view has evolved over time in the industry. Some of the defining characteristics that are frequently cited include:

- Services in a microservice architecture are often processes that communicate over a network to fulfill a goal using technology-agnostic protocols such as HTTP.[1][2][3]
- Services are organized around business capabilities.[4]
- Services can be implemented using different programming languages, databases, hardware and software environments, depending on what fits best.[5]
- Services are small in size, messaging-enabled, bounded by contexts, autonomously developed, independently deployable,[6][5] decentralized and built and released with automated processes.[6]

A microservice is not a layer within a monolithic application (example, the web controller, or the backend-for-frontend).[7] Rather, it is a self-contained piece of business functionality with clear interfaces, and may, through its own internal components, implement a layered architecture. From a strategy perspective, microservice architecture essentially follows the Unix philosophy of "Do one thing and do it well".[8] Martin Fowler describes a microservices-based architecture as having the following properties:[1]

- Lends itself to a continuous delivery software development process. A change to a small part of the application only requires rebuilding and redeploying only one or a small number of services.[9]
- Adheres to principles such as fine-grained interfaces (to independently deployable services), business-driven development (e.g. domain-driven design).[10]

It is common for microservices architectures to be adopted for cloud-native applications, serverless computing, and applications using lightweight container deployment. According to Fowler, because of the large number (when compared to monolithic application implementations) of services, decentralized continuous delivery and DevOps with holistic service monitoring are necessary to effectively develop, maintain, and operate such applications.[11] A consequence of (and rationale for) following this approach is that the individual microservices can be individually scaled. In the monolithic approach, an application supporting three functions would have to be scaled in its entirety even if only one of these functions had a resource constraint.[12] With microservices, only the microservice supporting the function with resource constraints needs to be scaled out, thus providing resource and cost optimization benefits.[13]

History

There are numerous claims as to the origin of the term microservices. Whilst vice president of ThoughtWorks in 2004, Fred George began working on prototype architectures based on what he called the "Baysean Principals" named after Jeff Bay.[14]

As early as 2005, Peter Rodgers introduced the term "Micro-Web-Services" during a presentation at the Web Services Edge conference. Against conventional thinking and at the height of the SOAP SOA architecture hype curve he argued for "REST-services" and on slide #4 of the conference presentation, he discusses "Software components are Micro-Web-Services".[15] He goes on to say "Micro-Services are composed using Unix-like pipelines (the Web meets Unix = true loose-coupling). Services can call services (+multiple language run-times). Complex service-assemblies are abstracted behind simple URI interfaces. Any service, at any granularity, can be exposed." He described how a well-designed microservices platform "applies the underlying architectural principles of the Web and REST services together with Unix-like scheduling and pipelines to provide radical flexibility and improved simplicity in service-oriented architectures.[16]

Rodgers' work originated in 1999 with the Dexter research project at Hewlett Packard Labs, whose aim was to make code less brittle and to make large-scale, complex software systems robust to change.[17] Ultimately this path of research led to the development of resource-oriented computing (ROC), a generalized computation abstraction in which REST is a special subset.

In 2007, Juval Löwy in his writing[18] and speaking[19][20] called for building systems in which every class was a service. Löwy realized this required the use of a technology that can support such granular use of services, and he extended Windows Communication Foundation (WCF) to do just that,[21][22] taking every class and treating it as a service while maintaining the conventional programming model of classes.

A workshop of software architects held near Venice in May 2011 used the term "microservice" to describe what the participants saw as a common architectural style that many of them had been recently exploring.[23] In May 2012, the same group decided on "microservices" as the most appropriate name. James Lewis presented some of those ideas as a case study in March 2012 at 33rd Degree in Kraków in Micro services - Java, the Unix Way,[24] as did Fred George[25] about the same time. Adrian Cockcroft, former director for the Cloud Systems at Netflix,[26] described this approach as "fine grained SOA", pioneered the style at web scale, as did many of the others mentioned in this article - Joe Walnes, Dan North, Evan Bottcher, and Graham Tackley.[27]

Microservices is a specialization of an implementation approach for service-oriented architectures (SOA) used to build flexible, independently deployable software systems.[4] The microservices approach is a first realisation of SOA that followed the introduction of DevOps and is becoming more popular for building continuously deployed systems.[28]

In February 2020, the Cloud Microservices Market Research Report predicted that the global microservice architecture market size will increase at a CAGR of 21.37% from 2019 to 2026 and reach $3.1 billion by 2026.[29]

Service granularity

A key step in defining a microservice architecture is figuring out how big an individual microservice has to be. There is no consensus or litmus test for this, as the right answer depends on the business and organizational context.[30] For instance, Amazon uses a service-oriented architecture where a service often maps 1:1 with a team of 3 to 10 engineers.[31] Generally, the terminology goes as such: services that are dedicated to a single task, such as calling a particular backend system or making a particular type of calculation, are called as *atomic services*. Similarly, services that call such atomic services in order to consolidate an output, are called as *composite services*.

It is considered bad practice to make the service too small, as then the runtime overhead and the operational complexity can overwhelm the benefits of the approach. When things get too fine-grained, alternative approaches must be considered - such as packaging the function as a library, moving the function into other microservices.[4]

If domain-driven design is being employed in modeling the domain for which the system is being built, then a microservice could be as small as an aggregate or as large as a bounded Context.[32]

Benefits

The benefit of decomposing an application into different smaller services are numerous:

- Modularity: This makes the application easier to understand, develop, test, and become more resilient to architecture erosion.[5] This benefit is often argued in comparison to the complexity of monolithic architectures.[33]
- Scalability: Since microservices are implemented and deployed independently of each other, i.e. they run within independent processes, they can be monitored and scaled independently.[34]
- Integration of heterogeneous and legacy systems: microservices is considered as a viable means for modernizing existing monolithic software application.[35][36] There are experience reports of several companies who have successfully replaced (parts of) their existing

software by microservices, or are in the process of doing so.[37] The process for Software modernization of legacy applications is done using an incremental approach.[38]
- Distributed development: it parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently.[39] It also allows the architecture of an individual service to emerge through continuous refactoring.[40] Microservice-based architectures facilitate continuous integration, continuous delivery and deployment.[41]

Criticism and concerns

The microservices approach is subject to criticism for a number of issues:

- Services form information barriers.[42]
- Inter-service calls over a network have a higher cost in terms of network latency and message processing time than in-process calls within a monolithic service process.[1]
- Testing and deployment are more complicated.[43][44]
- Moving responsibilities between services is more difficult.[5] It may involve communication between different teams, rewriting the functionality in another language or fitting it into a different infrastructure.[1] However, microservices can be deployed independently from the rest of the application, while teams working on monoliths need to synchronize to deploy together.[38]
- Viewing the size of services as the primary structuring mechanism can lead to too many services when the alternative of internal modularization may lead to a simpler design.[45] This requires understanding the overall architecture of the applications and interdependencies between components.[46]
- Two-phased commits are regarded as an anti-pattern in microservices-based architectures as this results in a tighter coupling of all the participants within the transaction. However, lack of this technology causes awkward dances which have to be implemented by all the transaction participants in order to maintain data consistency.[47]
- Development and support of many services is more challenging if they are built with different tools and technologies - this is especially a problem if engineers move between projects frequently.[48]
- The protocol typically used with microservices (HTTP) was designed for public-facing services, and as such is unsuitable for working internal microservices that often must be impeccably reliable.[49]
- While not specific to microservices, the decomposition methodology often uses functional decomposition, which does not handle changes in the requirements while still adds the complexity of services.[49]
- The very concept of microservice is misleading, since there are only services. There is no sound definition of when a service starts or stops being a microservice.[49]

Cognitive load

The architecture introduces additional complexity and new problems to deal with, such as network latency, message format design,[50] Backup/Availability/Consistency (BAC),[51] load balancing and fault tolerance.[44] All of these problems have to be addressed at scale. The complexity of a monolithic application does not disappear if it is re-implemented as a set of microservices. Some of the complexity gets translated into operational complexity.[52] Other places where the complexity manifests itself is in increased

network traffic and resulting slower performance. Also, an application made up of any number of microservices has a larger number of interface points to access its respective ecosystem, which increases the architectural complexity.[53] Various organizing principles (such as HATEOAS, interface and data model documentation captured via Swagger, etc.) have been applied to reduce the impact of such additional complexity.

Technologies

Computer microservices can be implemented in different programming languages and might use different infrastructures. Therefore, the most important technology choices are the way microservices communicate with each other (synchronous, asynchronous, UI integration) and the protocols used for the communication (RESTful HTTP, messaging, GraphQL ...). In a traditional system, most technology choices like the programming language impact the whole system. Therefore, the approach for choosing technologies is quite different.[54]

The Eclipse Foundation has published a specification for developing microservices, Eclipse MicroProfile.[55][56]

Service mesh

In a service mesh, each service instance is paired with an instance of a reverse proxy server, called a service proxy, sidecar proxy, or sidecar. The service instance and sidecar proxy share a container, and the containers are managed by a container orchestration tool such as Kubernetes, Nomad, Docker Swarm, or DC/OS. The service proxies are responsible for communication with other service instances and can support capabilities such as service (instance) discovery, load balancing, authentication and authorization, secure communications, and others.

In a service mesh, the service instances and their sidecar proxies are said to make up the data plane, which includes not only data management but also request processing and response. The service mesh also includes a control plane for managing the interaction between services, mediated by their sidecar proxies. There are several options for service mesh architecture: Open Service Mesh, Istio (a joint project among Google, IBM, and Lyft), Linkerd (a CNCF project led by Buoyant[57]), Consul (a HashiCorp product) and many others in the service mesh landscape (https://layer5.io/landscape). The service mesh management plane, Meshery (https://meshery.io), provides lifecycle, configuration, and performance management across service mesh deployments.

A comparison of platforms

Implementing a microservice architecture is very difficult. There are many concerns (see table below) that any microservice architecture needs to address. Netflix developed a microservice framework to support their internal applications, and then open-sourced[58] many portions of that framework. Many of these tools have been popularized via the Spring Framework – they have been re-implemented as Spring-based tools under the umbrella of the Spring Cloud[59] project. The table below shows a comparison of an implementing feature from the Kubernetes ecosystem with an equivalent from the Spring Cloud world.[60] One noteworthy aspect of the Spring Cloud ecosystem is that they are all Java-based technologies, whereas Kubernetes is a polyglot runtime platform.

Microservices concern	Spring Cloud & Netflix OSS	Kubernetes
Configuration management: configuration for a microservice application needs to be externalized from the code and be retrievable via a simple service call.	Spring Config Server, Netflix Archaius both support a Git-repository—based location for configuration. Archaius supports data typing of configuration.	Kubernetes ConfigMaps exposes the configuration stored in etcd via services. Kubernetes Secrets supports the service-based secure deployment and usage of sensitive configuration information (such as passwords, certificates, etc.).
Service discovery: maintain a list of service instances that are available for work within a microservice domain.	Spring Cloud Eureka allows clients to register to it, maintains a heartbeat with registered clients, and maps service names to hostnames for clients that lookup services by service name.	Kubernetes Services provide deployment-time registration of instances of services that are internally available within the cluster. Ingress is a mechanism whereby a service can be exposed to clients outside the cluster.
Load balancing: The key to scaling a distributed system is being able to run more than one instance of a component. Load has to be then distributed across those instances via a load balancer.	Spring Cloud Ribbon provides the ability for service clients to load balance across instances of the service.	Kubernetes Service provides the ability for the service to be load-balanced across service instances. This is not the equivalent of what Ribbon provides.
API gateway: The granularity of APIs provided by microservices is often different than what a service client needs. API Gateways implement facades and provide additional services like proxying, and protocol translation, and other management functions.	Spring Cloud Zuul provides configuration-based API facades	Kubernetes Service and Ingress resources, Istio, Ambassador are solutions that provide both north–south (traffic into and out of data center) as well as east–west (traffic across data centers or clouds or regions) API gateway functions. Zuul can also be implemented along with Kubernetes, providing configuration at individual service level.
Security concerns: Many security concerns are pushed to the API gateway implementation. With distributed microservice applications, it makes sense to not reinvent the security wheel and allow for policy definition and implementation in components that are shared by all services.	Spring Cloud Security addresses many security concerns through Spring Cloud Zuul	The Kubernetes ecosystem provides service meshes like Istio, which are capable of providing security through their API gateway mechanisms.
Centralized logging: It is important to have a centralized log gathering and analysis infrastructure to manage a plethora of services – many of which are operating in a distributed fashion.	ELK Stack (Elasticsearch, LogStash, Kibana)	EFK Stack (Elasticsearch, Fluentd, Kibana)
Centralized metrics: A centralized area where the health and performance of the individual services and overall system can be monitored is essential to proper operations.	Spring Spectator & Atlas	Heapster, Prometheus, & Grafana

Distributed tracing: Per-process logging and metric monitoring have their place, but neither can reconstruct the complex paths that transactions take as they propagate across a distributed system. Distributed tracing is an essential tool for a microservices platform.	Spring Cloud Sleuth	Hawkular, Jaeger
Resilience and fault tolerance: Distributed systems must be capable of auto-routing around failures, and be capable of routing requests to the service instance that will provide an optimum response.	Spring Hystrix, Turbine, & Ribbon	Health check, service meshes (example: Istio)[61]
Autoscaling and self-healing: Distributed systems respond to higher load by scaling horizontally: the platform must detect and auto-respond to such conditions. Furthermore, the system needs to detect failures and attempt auto-restarts without operator input.	-	Health check, self-healing, and auto-scaling
Packaging, deployment, and scheduling: Large-scale systems require robust package management, and deployment systems to manage rolling or blue-green deployments, and rollbacks if necessary. A scheduler helps determine which particular execution node a new set of services can be deployed to based on current conditions.	Spring Boot, Apache Maven. The Spring Cloud system does not have a true scheduler.	Docker, Rkt, Kubernetes Scheduler & Deployment, Helm[62]
Job management: scheduled computations disconnected from any individual user requests.	Spring Batch	Kubernetes Jobs and Scheduled Jobs
Singleton application: limit a specific service to run as the only instance of that service within the entire system.	Spring Cloud Cluster	Kubernetes Pods

See also

- Conway's law
- Cross-cutting concern
- Data mesh, a domain-oriented data architecture
- DevOps
- Fallacies of distributed computing
- GraphQL
- gRPC
- Representational state transfer (REST)
- Service-oriented architecture (SOA)
- Software modernization
- Unix philosophy
- Self-contained system (software)
- Serverless computing
- Web-oriented architecture (WOA)

References

1. Martin Fowler. "Microservices" (http://martinfowler.com/articles/microservices.html). Archived (https://web.archive.org/web/20180214171522/https://martinfowler.com/articles/microservices.html) from the original on 14 February 2018.
2. Newman, Sam (2015-02-20). *Building Microservices*. O'Reilly Media. ISBN 978-1491950357.
3. Wolff, Eberhard (2016-10-12). *Microservices: Flexible Software Architectures*. ISBN 978-0134602417.
4. Pautasso, Cesare (2017). "Microservices in Practice, Part 1: Reality Check and Service Design". *IEEE Software*. **34** (1): 91–98. doi:10.1109/MS.2017.24 (https://doi.org/10.1109%2FMS.2017.24). S2CID 5635705 (https://api.semanticscholar.org/CorpusID:5635705).
5. Chen, Lianping (2018). *Microservices: Architecting for Continuous Delivery and DevOps* (https://www.researchgate.net/publication/323944215). The IEEE International Conference on Software Architecture (ICSA 2018) (http://icsa-conferences.org/2018/). IEEE.
6. Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M., Microservice Architecture: Aligning Principles, Practices, and Culture, O'Reilly 2016
7. "Backends For Frontends Pattern" (https://docs.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends). *Microsoft Azure Cloud Design Patterns*. Microsoft.
8. Lucas Krause. *Microservices: Patterns and Applications*. ASIN B00VJ3NP4A (https://www.amazon.com/dp/B00VJ3NP4A).
9. Designing microservices: Continuous integration (https://docs.microsoft.com/en-us/azure/architecture/microservices/ci-cd) *Microsoft* Retrieved 9 January 2018
10. Josuttis, N. (2007). SOA in Practice. Sebastopol, CA, USA: O'Reilly. ISBN 978-0-596-52955-0.
11. Martin Fowler. "Microservice Prerequisites" (https://martinfowler.com/bliki/MicroservicePrerequisites.html).
12. Richardson, Chris (November 2018). "1.4.1 Scale cube and microservices". *Microservice Patterns*. Manning Publications. ISBN 9781617294549.
13. Mendonca, Nabor C.; Jamshidi, Pooyan; Garlan, David; Pahl, Claus (2019-10-16). "Developing Self-Adaptive Microservice Systems: Challenges and Directions". *IEEE Software*. **38** (2): 70–79. arXiv:1910.07660 (https://arxiv.org/abs/1910.07660). doi:10.1109/MS.2019.2955937 (https://doi.org/10.1109%2FMS.2019.2955937). S2CID 204744007 (https://api.semanticscholar.org/CorpusID:204744007).
14. "That Tech Show: The Grandfather of Microservices, Fred George" (https://play.acast.com/s/that-tech-show/fred-george-the-grandfather-of-microservices).
15. Rodgers, Peter. "Service-Oriented Development on NetKernel- Patterns, Processes & Products to Reduce System Complexity Web Services Edge 2005 East: CS-3" (https://web.archive.org/web/20180520124343/http://www.cloudcomputingexpo.com/node/80883). *CloudComputingExpo 2005*. SYS-CON TV. Archived from the original (http://www.cloudcomputingexpo.com/node/80883) on 20 May 2018. Retrieved 3 July 2017.
16. Rodgers, Peter. "Service-Oriented Development on NetKernel- Patterns, Processes & Products to Reduce System Complexity" (https://web.archive.org/web/20180520124343/http://www.cloudcomputingexpo.com/node/80883). *CloudComputingExpo*. SYS-CON Media. Archived from the original (http://www.cloudcomputingexpo.com/node/80883) on 20 May 2018. Retrieved 19 August 2015.
17. Russell, Perry; Rodgers, Peter; Sellman, Royston (2004). "Architecture and Design of an XML Application Platform" (http://www.hpl.hp.com/techreports/2004/HPL-2004-23.html). *HP Technical Reports*. p. 62. Retrieved 20 August 2015.

18. Löwy, Juval (2007). *Programming WCF Services, 1st ed*. O'Reilly Media. pp. 543–553. ISBN 978-0-596-52699-3.
19. Juval Löwy *"Every Class a WCF Service (https://web.archive.org/web/20191102212339/https://channel9.msdn.com/Shows/ARCast.TV/ARCastTV-Every-Class-a-WCF-Service-with-Juval-Lowy)". (Channel9, ARCast.TV, October 2007).*
20. Juval Löwy *"Every Class As a Service (https://web.archive.org/web/20191102212339/https://blogs.msdn.microsoft.com/drnick/2009/04/29/wcf-at-teched-2009/)" (Microsoft TechEd Conference, May 2009), SOA206. Archived from the original (https://web.archive.org/web/20191102212339/https://www.youtube.com/watch?v=w-Hxc6uWCPg) on 2010.*
21. Löwy, Juval (2007). *Programming WCF Services, 1st ed*. O'Reilly Media. pp. 48–51. ISBN 978-0-596-52699-3.
22. Löwy, Juval (2010). *Programming WCF Services, 3rd ed*. O'Reilly Media. pp. 74–75. ISBN 978-0-596-80548-7.
23. Dragoni, Nicola; Giallorenzo, Saverio; Lafuente, Alberto Lluch; Mazzara, Manuel; Montesi, Fabrizio; Mustafin, Ruslan; Safina, Larisa (2017). "Microservices: yesterday, today, and tomorrow". *Present and Ulterior Software Engineering*: 195–216. arXiv:1606.04036 (https://arxiv.org/abs/1606.04036). doi:10.1007/978-3-319-67425-4_12 (https://doi.org/10.1007%2F978-3-319-67425-4_12). ISBN 978-3-319-67424-7. S2CID 14612986 (https://api.semanticscholar.org/CorpusID:14612986).
24. James Lewis. "Micro services - Java, the Unix Way" (http://2012.33degree.org/talk/show/67).
25. Fred George (2013-03-20). "MicroService Architecture: A Personal Journey of Discovery" (https://www.slideshare.net/fredgeorge/micro-service-architecure).
26. Farrow, Rik (2012). "Netflix heads into the clouds" (https://www.usenix.org/system/files/login/articles/cockcroft_0.pdf) (PDF).
27. James Lewis and Martin Fowler. "Microservices" (http://martinfowler.com/articles/microservices.html).
28. "Continuous Deployment: Strategies" (https://www.javacodegeeks.com/2014/12/continuous-deployment-strategies.html). *javacodegeeks.com*. 10 December 2014. Retrieved 28 December 2016.
29. Research, Verified Market. "Cloud Microservices Market 2020 Trends, Market Share, Industry Size, Opportunities, Analysis and Forecast by 2026 – Instant Tech Market News" (https://www.instanttechnews.com/technology-news/2020/02/16/cloud-microservices-market-2020-trends-market-share-industry-size-opportunities-analysis-and-forecast-by-2026/). Retrieved 2020-02-18.
30. O. Zimmermann, Domain-Specific Service Decomposition with Microservice API Patterns, Microservices 2019, https://www.conf-micro.services/2019/slides//keynotes/Zimmerman.pdf
31. "Amazon SOA mandate" (https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/). 13 October 2011.
32. Vaughn, Vernon (2016). *Domain-Driven Design Distilled*. Addison-Wesley Professional. ISBN 978-0-13-443442-1.
33. Yousif, Mazin (2016). "Microservices". *IEEE Cloud Computing*. **3** (5): 4–5. doi:10.1109/MCC.2016.101 (https://doi.org/10.1109%2FMCC.2016.101).

34. Dragoni, Nicola; Lanese, Ivan; Larsen, Stephan Thordal; Mazzara, Manuel; Mustafin, Ruslan; Safina, Larisa (2017). "Microservices: How to make your application scale" (https://hal.inria.fr/hal-01636132/file/microservices-make-application.pdf) (PDF). *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Lecture Notes in Computer Science. **10742**: 95–104. arXiv:1702.07149 (https://arxiv.org/abs/1702.07149). Bibcode:2017arXiv170207149D (https://ui.adsabs.harvard.edu/abs/2017arXiv170207149D). doi:10.1007/978-3-319-74313-4_8 (https://doi.org/10.1007%2F978-3-319-74313-4_8). ISBN 978-3-319-74312-7. S2CID 1643730 (https://api.semanticscholar.org/CorpusID:1643730).
35. Newman, Sam (2015). *Building Microservices*. O'Reilly. ISBN 978-1491950357.
36. Wolff, Eberhard (2016). *Microservices: Flexible Software Architecture*. Addison Wesley. ISBN 978-0134602417.
37. Knoche, Holger; Hasselbring, Wilhelm (2019). "Drivers and Barriers for Microservice Adoption – A Survey among Professionals in Germany" (https://www.researchgate.net/publication/330442182). *Enterprise Modelling and Information Systems Architectures*. **14**: 1:1–35–1:1–35. doi:10.18417/emisa.14.1 (https://doi.org/10.18417%2Femisa.14.1).
38. Taibi, Davide; Lenarduzzi, Valentina; Pahl, Claus; Janes, Andrea (2017). "Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages" (https://www.researchgate.net/publication/319131505). *Proceedings of the XP2017 Scientific Workshops*. doi:10.1145/3120459.3120483 (https://doi.org/10.1145%2F3120459.3120483). S2CID 28134110 (https://api.semanticscholar.org/CorpusID:28134110).
39. Richardson, Chris. "Microservice architecture pattern" (http://microservices.io/patterns/microservices.html). *microservices.io*. Retrieved 2017-03-19.
40. Chen, Lianping; Ali Babar, Muhammad (2014). "Towards an Evidence-Based Understanding of Emergence of Architecture through Continuous Refactoring in Agile Software Development". *Proceedings Working IEEE/IFIP Conference on Software Architecture 2014 WICSA 2014*. The 11th Working IEEE/IFIP Conference on Software Architecture(WICSA 2014) (https://web.archive.org/web/20140730053454/http://wicsa2014.org/). IEEE. doi:10.1109/WICSA.2014.45 (https://doi.org/10.1109%2FWICSA.2014.45).
41. Balalaie, Armin; Heydarnoori, Abbas; Jamshidi, Pooyan (May 2016). "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture" (http://spiral.imperial.ac.uk/bitstream/10044/1/40557/8/SO_SWSI-2015-10-0149.R1_Balalaie.pdf) (PDF). *IEEE Software*. **33** (3): 42–52. doi:10.1109/ms.2016.64 (https://doi.org/10.1109%2Fms.2016.64). hdl:10044/1/40557 (https://hdl.handle.net/10044%2F1%2F40557). ISSN 0740-7459 (https://www.worldcat.org/issn/0740-7459). S2CID 18802650 (https://api.semanticscholar.org/CorpusID:18802650).
42. Stenberg, Jan (11 August 2014). "Experiences from Failing with Microservices" (http://www.infoq.com/news/2014/08/failing-microservices).
43. Calandra, Mariano (7 April 2021). "Why unit testing is not enough when it comes to microservices" (https://medium.com/swlh/why-unit-testing-is-not-enough-when-it-comes-to-microservices-c3b0dde14174).
44. "Developing Microservices for PaaS with Spring and Cloud Foundry" (http://www.infoq.com/presentations/microservices-pass-spring-cloud-foundry).
45. Tilkov, Stefan (17 November 2014). "How small should your microservice be?" (https://www.innoq.com/blog/st/2014/11/how-small-should-your-microservice-be/). *Innoq*. Retrieved 4 January 2017.
46. Lanza, Michele; Ducasse, Stéphane (2002). "Understanding Software Evolution using a Combination of Software Visualization and Software Metrics" (https://rmod.inria.fr/archives/papers/Lanz02aEvolutionMatrix.pdf) (PDF). *In Proceedings of LMO 2002 (Langages et Modèles à Objets)*: 135–149.

47. Richardson, Chris (November 2018). *Microservice Patterns*. Chapter 4. Managing transactions with sagas: Manning Publications. ISBN 978-1-61729454-9.
48. https://www.youtube.com/watch?v=X0tjziAQfNQ
49. Löwy, Juval (2019). *Righting Software 1st ed*. Addison-Wesley Professional. pp. 73–75. ISBN 978-0136524038.
50. Pautasso, Cesare (2017). "Microservices in Practice, Part 2: Service Integration and Sustainability". *IEEE Software*. **34** (2): 97–104. doi:10.1109/MS.2017.56 (https://doi.org/10.1109%2FMS.2017.56). S2CID 30256045 (https://api.semanticscholar.org/CorpusID:30256045).
51. Pautasso, Cesare (2018). "Consistent Disaster Recovery for Microservices: the BAC Theorem". *IEEE Cloud Computing*. **5** (1): 49–59. doi:10.1109/MCC.2018.011791714 (https://doi.org/10.1109%2FMCC.2018.011791714). S2CID 4560021 (https://api.semanticscholar.org/CorpusID:4560021).
52. Fowler, Martin. "Microservice Trade-Offs" (https://www.martinfowler.com/articles/microservice-trade-offs.html#ops).
53. "BRASS Building Resource Adaptive Software Systems". U.S. Government. DARPA. April 7, 2015. "Access to system components and the interfaces between clients and their applications, however, are mediated via a number of often unrelated mechanisms, including informally documented application programming interfaces (APIs), idiosyncratic foreign function interfaces, complex ill-understood model definitions, or *ad hoc* data formats. These mechanisms usually provide only partial and incomplete understanding of the semantics of the components themselves. In the presence of such complexity, it is not surprising that applications typically bake-in many assumptions about the expected behavior of the ecosystem they interact with".
54. Wolff, Eberhard (2018-04-15). *Microservices - A Practical Guide* (http://practical-microservices.com). ISBN 978-1717075901.
55. Swart, Stephanie (14 December 2016). "Eclipse MicroProfile" (https://projects.eclipse.org/projects/technology.microprofile). *projects.eclipse.org*.
56. "MicroProfile" (https://microprofile.io/). *MicroProfile*. Retrieved 2021-04-11.
57. "What's a service mesh?" (https://blog.buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/). *Buoyant*. Buoyant. 2017-04-25. Retrieved 5 December 2018.
58. *Netflix OSS* (https://netflix.github.io/), Git Hub
59. *Cloud* (http://spring.io/projects/spring-cloud), Spring
60. "Spring Cloud for Microservices Compared to Kubernetes" (https://developers.redhat.com/blog/2016/12/09/spring-cloud-for-microservices-compared-to-kubernetes/), *Developers*, Red hat, 2016-12-09
61. *Managing microservices with the Istio service mesh* (https://kubernetes.io/blog/2017/05/managing-microservices-with-istio-service-mesh/), Kubernetes, May 2017
62. *The Kubernetes Package Manager* (https://helm.sh/), Helm

Further reading

- Special theme issue on microservices, IEEE Software 35(3), May/June 2018, https://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=8354413
- I. Nadareishvili et al., Microservices Architecture – Aligning Principles, Practices and Culture (https://www.ca.com/content/dam/ca/us/files/ebook/microservice-architecture-aligning-principles-practices-and-culture.pdf), O'Reilly, 2016, ISBN 978-1-491-95979-4
- S. Newman, Building Microservices – Designing Fine-Grained Systems, O'Reilly, 2015 ISBN 978-1491950357

- Wijesuriya, Viraj Brian (2016-08-29) *Microservice Architecture, Lecture Notes (https://www.slideshare.net/tyrantbrian/microservice-architecture-65505794)* - University of Colombo School of Computing, Sri Lanka
- Christudas Binildas (June 27, 2019). Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud. Apress. ISBN 978-1484245002.
