Microservices

The Journey So Far and Challenges Ahead

Pooyan Jamshidi, Carnegie Mellon University

Claus Pahl, Free University of Bozen-Bolzano

Nabor C. Mendonça, University of Fortaleza

James Lewis, ThoughtWorks

Stefan Tilkov, INNOQ

MICROSERVICES ARE THE latest trend in software service design, development, and delivery.[1] They constitute an approach to software and systems architecture that builds on the well-established concept of modularization but emphasizes technical boundaries. Each module—each microservice—is implemented and operated as a small yet independent system, offering access to its internal logic and data through a well-defined network interface.[2] This increases software agility because each microservice becomes an independent unit of development, deployment, operations, versioning, and scaling.

Microservices and Service-Oriented Architecture

In relying on independent services with clear boundaries, microservices are similar to the more traditional service-oriented architecture (SOA).[3] Arguably, you could claim that microservices are a particular subtype of SOA. But although SOA tends to rely strongly on products such as enterprise service buses or other, similarly heavyweight middleware, microservices rely only on lightweight technologies.

In addition, SOA is often associated with web services protocols, tools, and formats such as SOAP, WSDL (Web Services Description Language), and the WS-* family of standards. In contrast, microservices typically rely on REST (Representational State Transfer) and HTTP[4] or other formats perceived as being lightweight and native for web development. Finally, SOA is viewed mostly as an integration solution, whereas microservices are typically applied to build individual software applications.

Microservices' Benefits

A number of benefits are often associated with microservices. Three of the most important ones are faster delivery, improved scalability, and greater autonomy.[2] It's no coincidence that these benefits tend to relate to market forces experienced by many organizations. This explains why it sometimes seems as if microservices have become the standard way of doing things in many development projects.[5]

Although microservice architectures come in different flavors, they all aim to speed up delivery—turning an idea on some product manager's or other project member's whiteboard into a feature running in production, as quickly as possible. Many organizations now see IT as the main facilitator of greater agility in terms of adapting to market changes, as opposed to its past role as a cost center best kept to a minimum. To achieve that goal, microservices typically are packaged and deployed in the cloud using lightweight container technologies,[6] following industry-proven DevOps practices[7] and supported by fully automated software integration and delivery machinery.[8] This enables rapid deployment of microservices in multiple execution environments (for example, testing, staging, and canary release) on arbitrary schedules, with a bare minimum of centralized management.[9]

The term "scalability" is somewhat ambiguous. It could refer to the system's runtime scalability—for example, its adaptability (at a reasonable cost) to changes in the number of users accessing it. Or, it could refer to the development process's ability to accommodate many developers working on it in parallel.

With microservices, the unit of scaling is each microservice. So, at runtime, services can be scaled differently according to their specific requirements. But the microservice is also the unit of development and deployment. So, each service can be developed, deployed, and operated by a different team, allowing for a more parallel introduction of new features.

Finally, related to the concept of a scalable organization, each microservice is expected to offer an autonomous, bounded unit of both development and runtime decisions. This lets a team make localized decisions for each service—for example, in terms of the programming language, libraries, or frameworks used; the database technology (if any) employed; or any other aspect of its implementation strategy. This allows for a best-of-breed approach, with each team selecting the optimal choice for its area of responsibility.

Microservice Evolution

According to James Lewis and Martin Fowler, the term "microservices" was first discussed at a May 2011 software architecture workshop, to denote a common architectural approach the workshop participants had been exploring.[2] Previously, prominent industry experts had already been exploring some of the same ideas, albeit under different guises. For instance, Werner Vogels at Amazon had described its architectural approach as "encapsulating the data with the business logic that operates on the data, with the only access through a published service interface,"[10] whereas Adrian Cockcroft, then at Netflix, referred to "loosely coupled service-oriented architecture with bounded contexts."[11] Other terms used in industry at that time to convey similar concepts were "fine-grained SOA" and "SOA done right."

Those early SOA-related terms were all testimony to the fact that
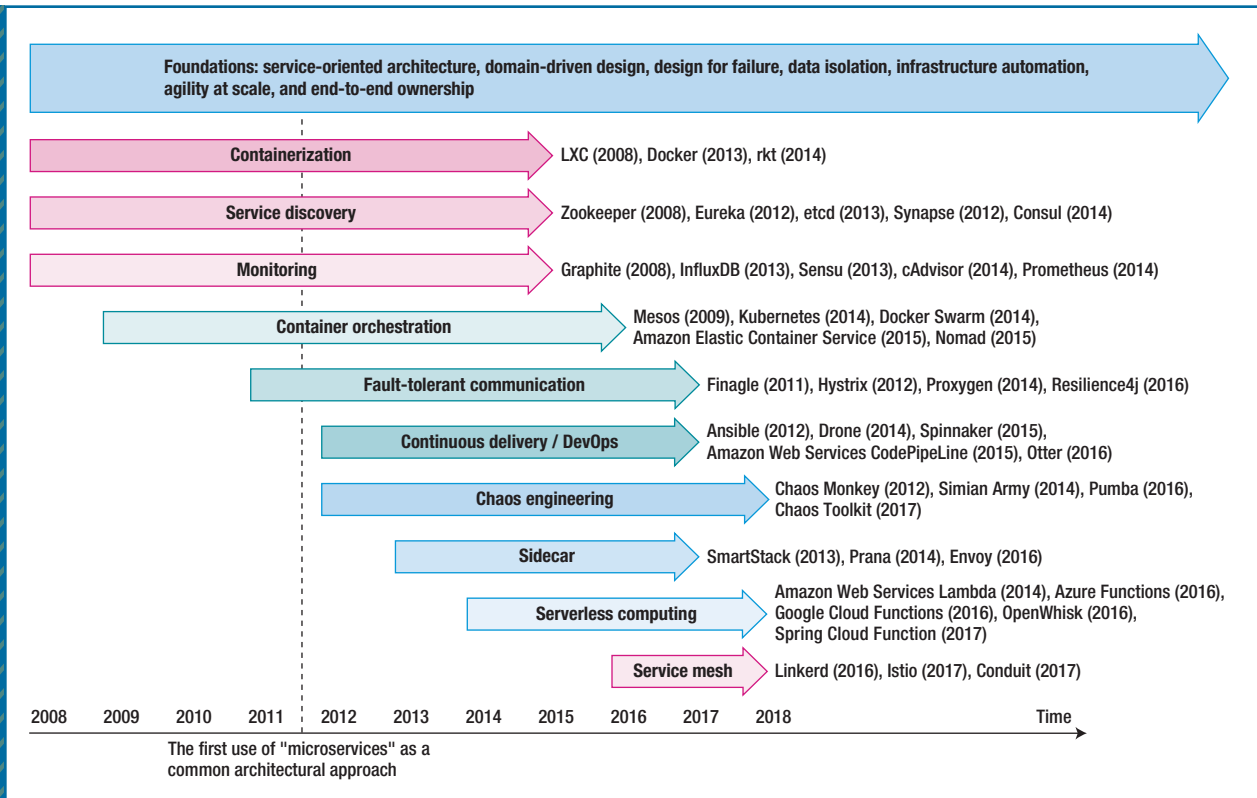
FIGURE 1. A microservice technologies timeline.

microservices are rooted firmly in SOA. However, they also reveal that the industry itself wasn't fully satisfied with SOA, as evidenced by the en masse switch from SOAP to REST, a more lightweight and simpler service invocation protocol.

Other important software development concepts played a key role in the emergence of microservices. This was especially true for domain-driven design (DDD), a model-based-development approach guided by principles such as bounded organization contexts and continuous software integration.[12] Also highly influential were approaches such as design for failure, data isolation, infrastructure automation, agility at scale, cross-functional teams, and end-to-end product ownership.[2]

These approaches solved many challenges of distributed web-scale applications (Facebook, Spotify, and so on) as well as organizational issues that large-scale companies faced.

Here, we look at microservices' evolution from both the technological and architectural perspectives.

The Technological Perspective

From the technological perspective, early microservice applications were strongly influenced by a new generation of software development, deployment, and management tools. As microservice architectures became more popular, those tools continued evolving to support a wider, more diverse user base, leading to the creation of even more-advanced technologies.

Figure 1 shows a timeline with 10 "waves" of software technologies, including some of their most representative tools, that have influenced microservice application development, deployment, and operation over the last decade.

The first five technological waves already existed before the term "microservices" was generally adopted. The first wave comprises lightweight container technologies (for example, LXC and Docker), which allow individual services to be more effectively packaged, deployed, and managed at runtime. The second wave comprises service discovery technologies (for example, Eureka and Consul), which let services communicate with each other without explicitly referring to their network locations.

The third wave comprises monitoring technologies (for example, Graphite and Sensu), which enable runtime monitoring and analysis of the behavior of microservice resources at different levels of detail. The fourth wave comprises container orchestration technologies (for example, Mesos and Kubernetes), which automate container allocation and management tasks, essentially abstracting away the underlying physical or virtual infrastructure from service developers. The fifth wave comprises latency and fault-tolerant communication libraries (for example, Finagle and Hystrix), which let services communicate more efficiently and reliably.

The other five waves emerged in response to microservices' increasing popularity. The sixth wave comprises continuous-delivery technologies (for example, Ansible and Drone), which provide general integration solutions to automate many of the DevOps practices typically used in a web-scale microservice production environment.[13] The seventh wave comprises chaos-engineering technologies (for example, Simian Army and Chaos Toolkit), which automate the execution of critical systemwide reliability and security testing techniques, such as failure and attack injection.[14]

The eighth wave comprises sidecar technologies (for example, Prana and Envoy), which encapsulate communication-related features such as service discovery and the use of protocol-specific and fault-tolerant communication libraries, so as to abstract them from service developers.[15] The ninth wave comprises "serverless" computing technologies (for example, AWS Lambda and OpenWhisk), which implement the function-as-a-service (FaaS) cloud model.[16] (AWS stands for Amazon Web Services.) This model lets cloud users develop, deploy, and deliver into production more fine-grained service functionalities—or functions—without the complexity of creating and managing (for example, to cope with inconsistent traffic patterns) the infrastructure resources necessary for their execution. Finally, the tenth wave comprises service mesh technologies (for example, Linkerd and Istio), which build on sidecar technologies to provide a fully integrated service-to-service communication monitoring and management environment.[17]

The vast majority of the tools in Figure 1 originated from industry. One exception is Mesos, which originated from a research prototype developed at UC Berkeley. Despite their industry origin, most of these tools are publicly available as open source projects. Table 1 gives the URL for each tool.

The Architectural Perspective

Those waves' impact has been reflected in how microservice applications have evolved from an architectural perspective. Figure 2 illustrates four generations of microservice architectures.

In the first generation (see Figure 2a), individual services were packed using lightweight container technologies, such as LXC. They were then deployed and managed at runtime using a container orchestration tool, such as Mesos. Each service was responsible for keeping track of the location of the other services, which were invoked following specific communication protocols. Any failure-handling mechanism, such as retry and fall back, was implemented directly in the services' source code. As the number of services per application increased and the need to deploy and redeploy services in different execution environments became more frequent, locating the appropriate service instances to invoke became a huge issue. Also, as new services were implemented using different programming languages, reusing existing discovery and failure-handling code became increasingly difficult.

To address some of those issues, the second generation (see Figure 2b) introduced discovery services and reusable fault-tolerant communication libraries. Services used a common discovery service, such as Consul, to register their provided functionalities. Client services could then dynamically discover and invoke these functionalities without any explicit reference to the invoked services' location. During service invocation, all protocol-specific and failure-handling features were delegated to an appropriate communication library, such as Finagle. This strategy not only simplified service implementation and testing but also allowed reuse of boilerplate communication code across services.

However, as those libraries became increasingly sophisticated, and because reimplementing them in a new programming language isn't a trivial task, developers were often forced to implement new services using only the languages for which those libraries were already available. Consequently, developers no longer explored microservices' full benefits, especially regarding developers' supposed autonomy to choose any programming language or development technology they deemed the most appropriate to satisfy specific service needs.

In response, the third generation (see Figure 2c) introduced standard

Table 1. URLs for the microservice tools in Figure 1.

Tool category	Tool name	URL
Container engine	LXC	linuxcontainers.org
	Docker	www.docker.com
	rkt	coreos.com/rkt
Service discovery	ZooKeeper	zookeeper.apache.org
	Eureka	github.com/Netflix/eureka
	etcd	coreos.com/etcd
	Synapse	github.com/airbnb/synapse
	Consul	www.consul.io
Monitoring	Graphite	graphiteapp.org
	InfluxDB	github.com/influxdata/influxdb
	Sensu	sensuapp.org
	cAdvisor	github.com/google/cadvisor
	Prometheus	prometheus.io
Container orchestration	Mesos	mesos.apache.org
	Kubernetes	kubernetes.io
	Docker Swarm	docs.docker.com/engine/swarm
	Amazon Elastic Container Service	aws.amazon.com/ecs
	Nomad	www.nomadproject.io
Fault tolerance	Finagle	twitter.github.io/finagle
	Hystrix	github.com/Netflix/Hystrix
	Proxygen	github.com/facebook/proxygen
	Resilience4j	github.com/resilience4j/resilience4j
Continuous delivery	Ansible	www.ansible.com
	Drone	drone.io
	Spinnaker	www.spinnaker.io
	Amazon Web Services CodePipeLine	aws.amazon.com/codepipeline
	Otter	inedo.com/otter
Chaos engineering	Chaos Monkey	github.com/Netflix/chaosmonkey
	Simian Army	github.com/Netflix/SimianArmy
	Pumba	github.com/alexei-led/pumba
	Chaos Toolkit	chaostoolkit.org

Table 1. URLs for the microservice tools in Figure 1 (cont.).

Tool category	Tool name	URL
Sidecar	SmartStack	nerds.airbnb.com/smartstack-service-discovery-cloud
	Prana	github.com/Netflix/Prana
	Envoy	www.envoyproxy.io
Serverless computing	Amazon Web Services Lambda	aws.amazon.com/lambda
	Azure Functions	azure.microsoft.com/services/functions
	Google Cloud Functions	cloud.google.com/functions
	OpenWhisk	openwhisk.apache.org
	Spring Cloud Function	cloud.spring.io/spring-cloud-function
Service mesh	Linkerd	linkerd.io
	Istio	istio.io
	Conduit	conduit.io

service proxies, or sidecars, such as Envoy, as transparent service intermediates. The idea was to further improve software reusability by having sidecars encapsulate all service discovery and communication features. Because each sidecar is a self-contained service, this strategy immediately brought the full benefits of existing fault-tolerant communication libraries to any new programming language, thus also increasing development autonomy.

When used as network intermediaries, sidecars become the natural locus for monitoring the behavior of all service interactions in a microservice application. This is exactly the idea behind service mesh technologies such as Linkerd. These tools extend the notion of self-contained sidecars to provide a more integrated service communication solution. Application operators can dynamically monitor and manage the behavior of multiple distributed sidecars,

by means of a centralized control plane.[18] In this way, operators can exert more fine-grained control over a variety of service-to-service communication features, including service discovery, load balancing, fault tolerance, message routing, and security.

The fourth generation (see Figure 2d) aims to bring microservice applications to a new realm. The idea is to exploit recent FaaS and severless-computing technologies, such as AWS Lambda, to further simplify microservice development and delivery. With this serverless architecture, microservice applications would essentially turn into collections of "ephemeral" functions, each of which could be created, updated, replaced, and deleted as quickly and arbitrarily as necessary.[19]

One interesting aspect of the serverless architecture is whether communication-centric technologies, such as sidecars and service meshes,

would still be necessary. Existing FaaS platforms don't yet provide all the communication and traffic management features that those two technologies provide. So, you could arguably conceive of a scenario in which sidecar-like functions are created to intermediate all function-to-function interactions in a serverless application (see Figure 2d). A higher-level control plane function could then monitor and manage those sidecar functions' behavior, forming a new kind of service (or function) mesh.

Future Challenges

As an obvious downside of microservices' increased popularity, they're more likely to be used in situations in which the costs far outweigh the benefits. One reason could be that a project would best be developed in monolithic fashion. This doesn't mean it shouldn't be designed to be modular, just that its
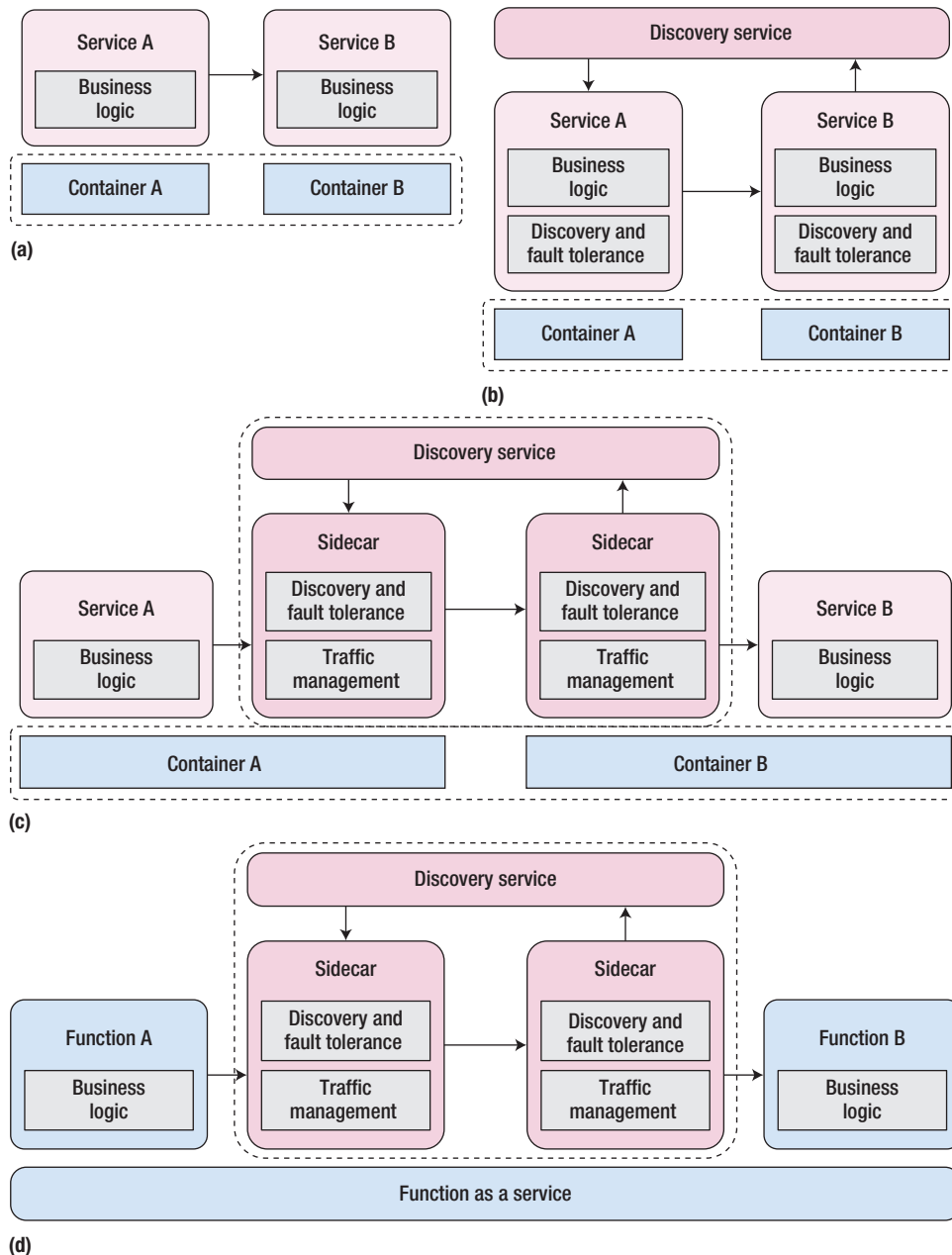
FIGURE 2. Four generations of microservice architecture. (a) Container orchestration. (b) Service discovery and fault tolerance. (c) Sidecar and service mesh. (d) Serverless architecture.

modules don't need to be as isolated as microservices.

Microservices aren't, and never will be, the right solution in all cases. More interesting, though, are situations in which microservices would be a good fit but teams don't implement them successfully. There are many possible reasons for this. Future developments could perhaps address some of these challenges, as we discuss next.

Service Modularization and Refactoring

With any approach to modularization, finding the right modules, with the right size, the right assignment of responsibilities, and well-designed interfaces, is a challenge. This is especially true for microservices and other approaches in which badly designed boundaries can lead to increased network communication. Such an increase might yield a system unsuitable for its intended tasks owing to abysmal performance and instability.

That general challenge will remain, but two aspects could be addressed. First, refactoring a system composed of microservices could be made easier through tooling, even though this might have unintended consequences regarding an infrastructure or development dependency. Second, a move toward asynchronous communication, more pervasive use of libraries implementing stability patterns, and more sophisticated runtime environments could help address stability issues. At the far end of this spectrum, a purely serverless approach places a lot of trust in the underlying platform and the services it offers. Again, this is at the cost of drastically increased dependency on a particular environment. Such a result runs counter to one of microservices' original goals and might remind some of you of SOA infrastructure.

Service Granularity

Another issue is the lack of agreement on the right size of microservices. Although the name itself seems to suggest that microservices should be as small as possible, project teams tend to interpret this tenet in drastically different ways. Some teams have microservices with only a few to a few dozen LOC. Others encapsulate a few KLOC, along with a few dozen classes and possibly database entities, into a microservice. Communication among these mid-sized services can be synchronous or asynchronous. The self-contained-systems approach, arguably just a variant of microservices, advocates including UIs, specifically web UIs, as part of a service and suggests relying on front-end integration as much as possible.[20]

Each approach we just described has merit. However, the fact that they're all labeled "microservices" shows that the potential exists for establishing a set of patterns to help with design decisions when you're splitting a domain into microservices and sizing each service.

Front-End Integration

UIs generally are a critical component in microservice architectures. This is mostly because they haven't been the focus of many of the approach's advocates, who typically are architects dealing with back-end aspects. This can lead to systems in which a monolithic front end uses a number of back-end microservices. Such architectures occasionally can be perfectly fine, but they often inhibit the goals of microservices because all the downsides of a monolithic architecture still exist.

Modularization of front ends of different kinds, whether web, native, or hybrid, along with their association as part of either microservices or a collaborating entity, is often badly needed to help organizations implement a full-stack microservice environment.

Resource Monitoring and Management

As microservice applications' size and complexity grow, the number and diversity of infrastructure resources (for example, virtual machines, containers, services, messages, thread pools, and logs) that must be continuously monitored and managed at runtime also increase. In addition, services might be deployed across multiple regions and availability zones, which exacerbates the challenge of collecting up-to-date information about their status and behavior. Ultimately, with the increasing level of automation that current monitoring technologies provide, application developers might find themselves amid a flood of monitoring events, unable to make timely management decisions.

A crucial issue here is how to define appropriate alert thresholds and filters, so as to notify developers whenever something goes wrong without overloading them with redundant or irrelevant information. Even more challenging is the issue of how to learn from past events and actions, to better inform (and potentially automate) resource management decisions. As in many other big data scenarios, control theory and machine learning should play important roles toward the development of more scalable microservice monitoring and resource management.

Failure, Recovery, and Self-Repair

Like any type of distributed system, microservices typically are fragile. For many reasons, such as network, hardware, or application-level issues, they might become unavailable, become inaccessible, or simply fail. Owing to service dependencies, any service can become temporarily inaccessible to its consumers. In any distributed setting, communication will fail from time to time. Regarding a microservice system as a whole, communication failures will likely occur often simply because

of the number of messages passing between services.

To minimize the impact of partial outages, developers must build fault-tolerant services that can gracefully respond to certain types of failure. Researchers and practitioners have proposed ways to isolate failures so that they won't propagate throughout a distributed system. However, more work is needed on automated failure management and on self-repair and self-healing solutions to fix a system after a failure.

Organizational Culture and Coordination

Having many autonomous teams developing independently deployed services might be a double-edged sword. One one hand, each team can make local decisions without always having to negotiate with other teams. On the other hand, it increases the risk of teams failing to see the big picture—that is, to understand whether their local decisions are justifiable and coherent in the context of the system's overall architecture and business goals.

This risk might manifest itself in a variety of situations, from the adoption of infrastructure solutions that are difficult to communicate and re-use across services, to the creation of a conflict-avoidance culture in which teams try to fix locally problems that are other teams' responsibility. As a mitigation approach, microservice developers will need better coordination structures and models that not only promote team autonomy but also take into account system- and organization-wide requirements and goals.

Some organizations (for example, Netflix) have adopted cross-team regular discussions as a part of their organizational culture. Some (for

example, Spotify) have developed utilities that log their services and developed technologies in a central repository. Others (such as Uber) decide about technologies and programming languages outside the teams. No matter what works for an organization, these issues must be dealt with as part of the organizational culture to scale well to several hundreds or even thousands of microservices.[21]

Addressing the Challenges

Each of the challenges we just discussed, as well as a number of others we didn't, could potentially be addressed by software- and systems-engineering researchers in academia. Indeed, microservices are rapidly becoming a hot research topic, with a growing number of published research papers.[22] However, those papers have had little if any impact on microservice practice. One possible reason is that researchers in academia have limited access to industry-scale microservice applications. This makes it difficult for them to replicate and experiment with all sorts of technical and non-technical issues that might occur in real-world microservice production environments.[21]

To remedy this situation, we envision two distinct yet complimentary strategies. First, there should be more incentives for industry–academia collaboration, on both sides. Second, practitioners and researchers should strive to develop and share a common microservice infrastructure that can emulate, as accurately as possible, the production environments of typical microservice applications. Such an infrastructure would enable the microservice research community to not only tackle problems that are more representative of the

issues practitioners face but also conduct more repeatable and industry-focused empirical studies.[23]

In This Issue

For this theme issue, we received 26 submissions, from which four were accepted after two rounds of reviews. Ultimately, five articles were selected, including one from *IEEE Software*'s publication queue.

The selected articles look at microservice development from both engineering and reengineering perspectives. Generally, the processes are organized in a stepwise fashion, guided by quality attributes—for example, maintainability, scalability, and reliability. And, the processes are supported by different engineering approaches—for example, domain-driven design, model-driven engineering, and pattern- and antipattern-based refactoring. Furthermore, the articles use existing domain models, experience reports, and empirical studies to support their claims.

In "Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective," Florian Rademacher and his colleagues present a methodology for designing microservice architectures based on domain-driven design (DDD) and model-driven development (MDD). They discuss the challenges of applying DDD to microservice design and propose MDD strategies to cope with them.

In "Using Microservices for Legacy Software Modernization," Holger Knoche and Wilhelm Hasselbring describe a migration process that decomposes an application stepwise into microservices. This process defines an abstract service facade, adapts it, and uses it as a target architecture. Knoche and Hasselbring

discuss their experience applying this process in an ongoing software modernization project that aims to improve the evolvability of an existing legacy application.

In "From Monolithic to Microservices: An Experience Report from the Banking Domain," Antonio Bucchiarone and his colleagues report on their experience migrating a mission-critical monolithic banking application to a microservice architecture. They show how reimplementing a monolithic architecture into microservices can improve scalability. They also discuss other benefits of migration to microservices, including reduced complexity, lower coupling, higher cohesion, and simplified integration.

In "On the Definition of Microservice Bad Smells," Davide Taibi and Valentina Lenarduzzi investigate how architectural antipatterns (bad smells) support the design of cloud-native microservice applications. Their empirical study identified common bad practices in microservice development, which they classified into a catalog of 11 microservice-specific bad smells. Microservice developers can use these results as guidelines for avoiding similar difficulties in their projects.

Finally, in "Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency," Andrei Furda and his colleagues propose pattern-based microservice refactoring that focuses on three challenges: multitenancy, statefulness, and data consistency. They explain how multitenancy enables different organizations with distinctive requirements to use microservices, why statefulness affects a microservice system's availability and reliability, and why data consistency challenges occur

during migration of legacy code that operates with a centralized data repository.

The microservice technologies, architectures, and challenges we've discussed, along with the results and experiences described in the articles in this theme issue, are only a small sample of the technical, organizational, and business decisions you need to take into account when engineering production-quality microservice applications at scale. Nevertheless, we hope they provide a timely incentive for software practitioners, researchers, and tool developers to further advance this promising architectural approach. ⓢ

Acknowledgments

References

1. O. Zimmermann, "Microservices Tenets: Agile Approach to Service Development and Deployment," *Computer Science—Research and Development*, vol. 32, nos. 3–4, 2017, pp. 301–310.
2. J. Lewis and M. Fowler, "Microservices," 25 Mar. 2014; martinfowler.com/articles/microservices.html.
3. N.M. Josuttis, *SOA in Practice: The Art of Distributed System Design*, O'Reilly, 2007.
4. R.T. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," doctoral diss., Univ. of California, Irvine, 2000.

ABOUT THE AUTHORS

POOYAN JAMSHIDI is a postdoctoral research associate at Carnegie Mellon University's School of Computer Science. His research interests are at the intersection of software engineering, systems, and machine learning. Jamshidi received a PhD in computing from Dublin City University. Contact him at pjamshid@cs.cmu.edu.

JAMES LEWIS is a principal consultant at ThoughtWorks, where he specializes in service-oriented architecture (SOA) and distributed systems. His favorite topics include domain-driven design, SOA and the future of the web, and agile adoption patterns and lean thinking. Lewis received an MSc in computer science from the University of London. He's a member of IEEE and ACM. Contact him at jalewis@thoughtworks.com.

CLAUS PAHL is an associate professor of computer science at the Free University of Bozen-Bolzano, where he heads the Software and Systems Engineering Group. His research interests include software engineering in service and cloud computing, specifically migration, architecture specification, dynamic quality, performance engineering, and scalability. Pahl received a PhD in computing from the University of Dortmund. Contact him at cpahl@unibz.it.

STEFAN TILKOV is a cofounder of and a principal consultant at INNOQ, a technology-consulting company. He has been involved in designing large-scale distributed systems for more than two decades. Tilkov received a BSc in computer science from Berufsakademie Stuttgart. Contact him at stefan.tilkov@innoq.com.

NABOR C. MENDONÇA is a full professor of computer science at the University of Fortaleza and a visiting scholar at Carnegie Mellon University's School of Computer Science. His research interests include software engineering, distributed systems, and cloud computing. Mendonça received a PhD in computing from Imperial College London. He has received Microsoft Research's Software Engineering Innovation Foundation Award. He's a member of IEEE, ACM, and the Brazilian Computer Society. Contact him at nabor@unifor.br.

5. C. Pautasso et al., "Microservices in Practice, Part 1: Reality Check and Service Design," *IEEE Software*, vol. 34, no. 1, 2017, pp. 91–98.

6. C. Pahl, "Containerization and the PaaS Cloud," *IEEE Cloud Computing*, vol. 2, no. 3, 2015, pp. 24–31.

7. L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*, Addison-Wesley, 2015.

8. D. Farley and J. Humble, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010.

9. S. Newman, *Building Microservices*, O'Reilly, 2015.

10. J. Gray, "A Conversation with Werner Vogels," *ACM Queue*, vol. 4, no. 4, 2006, pp. 14–22.

11. A. Cockcroft, "The Evolution of Microservices," presentation at 2016

ACM Learning Webinar, 2016; learning.acm.org/webinar_pdfs /EvolutionOfMicroservices _WebinarSlides.pdf.

12. E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.

13. A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," *IEEE Software*, vol. 33, no. 3, 2016, pp. 42–52.

14. C. Rosenthal et al., *Chaos Engineering: Building Confidence in System Behavior through Experiments*, O'Reilly, 2017.

15. B. Burns and D. Oppenheimer, "Design Patterns for Container-Based Distributed Systems," *Proc. 8th USENIX Workshop Hot Topics in Cloud Computing* (HotCloud 16), 2016; www.usenix.org/system/files /conference/hotcloud16/hotcloud16 _burns.pdf.

16. M. Roberts, "Serverless Architectures," 4 Aug. 2016; martinfowler .com/articles/serverless.html.

17. W. Morgan, "What's a Service Mesh? And Why Do I Need One?," 25 Apr. 2017; buoyant.io/2017/04/25 /whats-a-service-mesh-and-why-do -i-need-one.

18. P. Calçado, "Pattern: Service Mesh," 3 Aug. 2017; philcalcado.com/2017 /08/03/pattern_service_mesh.html.

19. A. Cockcroft, "Cloud Trends: Principles, Evolution, and Chaos ...," presentation at GOTO Copenhagen 2017, 2017; gotocph .com/5/sessions/185/slides.

20. *Self-Contained Systems: Assembling Software from Independent Systems*; scs-architecture.org.

21. M. Ranney, "What I Wish I Had Known before Scaling Uber to 1,000 Services," presentation at GOTO Chicago 2016, 2016; gotocon.com/dl /goto-chicago-2016/slides/MattRanney _WhatIWishIHadKnownBeforeScaling UberTo1000Services.pdf.

22. C. Pahl and P. Jamshidi, "Microservices: A Systematic Mapping Study," *Proc. 6th Int'l Conf. Cloud Computing and Services Science* (CLOSER 16), 2016, pp. 137–146.

23. C.M. Aderaldo et al., "Benchmark Requirements for Microservices Architecture Research," *Proc. 1st Int'l Workshop Establishing the Community-Wide Infrastructure for Architecture-Based Software Eng.* (ECASE 17), 2017, pp. 8–13.