Editor: **Cesare Pautasso**
University of Lugano
c.pautasso@ieee.org

Editor: **Olaf Zimmermann**
University of Applied Sciences
of Eastern Switzerland, Rapperswil
ozimmerm@hsr.ch

Microservices in Practice, Part 1

Reality Check and Service Design

Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis

MICROSERVICES are the latest fashion in software architecture and development practices. Since 2014, the discussion has been controversial, with opinions varying from "holy grail" and "must adopt" to "nothing new" and "won't work in the long run." Here, service-oriented architecture (SOA) and microservices insiders Mike Amundsen, James Lewis, and Nicolai Josuttis share their experiences and predictions with department editors Cesare Pautasso and Olaf Zimmermann.

Opening Positions

Olaf Zimmermann: Let's start with some fundamentals. Sam Newman wrote that we should "think of microservices as a specific approach for SOA in the same way that XP [Extreme Programming] or Scrum are specific approaches for agile software development."[1] Do you agree?

Cesare Pautasso: More specifically, how does REST [Representational State Transfer] fit in, and what was missing in previous SOA implementation approaches?

Mike Amundsen: Let's start with REST. What I find fascinating about Roy Fielding's description of his "network-based software architecture" style is that he chose to use a unique formula for defining it: architectural properties + domain requirements = interaction constraints. I've not seen anyone else take this approach to defining an architectural style. Most writers today focus on his list of constraints (client-server, stateless, cache, uniform interface, and so on) and don't talk much about his list of architectural properties (network performance, scalability, simplicity, and so on).[2] And I would say that it is this list of desired properties that ties back to what we've seen in SOA over the last decade and what we are now seeing in microservices. The properties you wish to "design for" will drive the constraints you need to apply to the implementation of a system. In the book *Microservice Architecture*, my fellow authors and I propose a capabilities model for this property–constraint relation.[3]

Nicolai Josuttis: Let me answer the question a bit differently. I tend to agree that microservices, according to my understanding, are a special approach for SOA. I had better skip the question about the difference between SOA and REST, which gets loaded and biased easily. Microservices recommend to constrain an SOA in very similar way that I recommend and teach to constrain any SOA to be successful.[4] This has to do with the context and complexity of

0740-7459/17/$33.00 © 2017 IEEE

a system landscape. For example, the larger the system landscape is, the more problematic it is to place business logic "in the middle" (that is, in the infrastructure or in a service bus). The problem with business logic "in the middle" is that a central enterprise service bus may become a bottleneck and is not governed by a service contract.

Microservices recommend the exact opposite: place all business intelligence and even message transformations at the endpoints. This way, service design becomes a distributed multilateral approach, which allows architectures to scale. And this is what I and other consultants and course instructors promote as the best approach for a scalable SOA. So, yes, microservices are in many ways a best-practice approach for realizing SOA.

> Microservices are in many ways a best-practice approach for realizing service-oriented architecture.

However, there is one additional aspect that comes into play: adopting the modern principles of operations—DevOps and automated deployment in particular.

James Lewis: I fully agree. I remember when writing the original article on microservices[5] that there was much debate between Martin Fowler and me as to whether we should either use the word "style" or describe what we were describing as a "definition." Some context may be useful here. We were trying to explain what we and others were seeing on projects inside ThoughtWorks and also approaches being taken by the wider industry. The discussion over whether microservices are a style or not was and still is pretty tricky—as Mike points out in his answer and as Olaf has just investigated.[6]

We avoided the formality of an approach such as Fielding's, not so much because we didn't think the topic could be approached in that way (it can, and in fact I have a number of attempts predating the publication of our article) but for two reasons. First, we were simply trying to tie together a number of better practices that had emerged over the previous few years (or decades in some cases). Second, Martin and I weren't attempting to write a paper with academic rigor but to put a name to a phenomenon that we were experiencing firsthand. We were trying to describe what we were seeing, and I think there is now a post hoc attempt to add more rigor to the original characteristics. I find it very flattering that others are doing that, including the other interview participants.

It's interesting that you mention XP in the question since I use XP as a reference when talking about the characteristics that we call out. Each of the XP practices is complementary to others. That is, each practice makes doing more of another practice more attractive. I think there is a pretty good but admittedly post hoc case to interpret the invention of continuous delivery as described by Jezz Humble and David Farley[7] as the end product of combining complementary XP practices.

For instance, test-driven development and continuous integration [CI] are complementary, writing more tests makes CI more attractive, and as we get advances in CI—build pipelines and so on—that makes writing more and different types of automated tests more attractive. Finally, you reach global maxima of fully automated build pipelines running through to production—something which, incidentally, I consider a fundamental prerequisite of a microservices-based architecture. I think it's the same with microservices; there are a number of complementary practices that we have built up over time. These include RESTful integration, consumer-driven contracts, and domain-driven design [DDD].

Business Context and Technology Adoption

Olaf: In which business domains and application genres have you applied microservices?

Cesare: And which particular concepts and technologies have you applied, and why?

James: Relating my answer here to my previous answer, I think most of the projects I've worked on have used some number of the characteristics over the last 10 years or more. I've been very lucky to have worked with some very deep thinkers in the space of continuous delivery, messaging, RESTful integration, and SOA at ThoughtWorks. These people have included Jim Webber,

Ian Robinson, Sam Newman, Fred George, and Erik Dörnenburg. My experience working with and being around these people has been that you need to apply different architectural approaches for each set of functional and cross-functional requirements as you understand them. Microservices, the full shebang if you like, are applicable across most domains, but only if you have the requirements that necessitate them.

For example, if you need to move fast with a really small team, you may not need to split your application up into a lot of small units immediately. You are going to pay a price—the "microservice premium," Martin Fowler calls it[8]—to spin up all the associated infrastructure. This could slow you down in the short term and is unnecessary if you are still proving a business case. However, if you know that you have a cross-functional requirement to support millions of users—say, you are replacing an existing system—then you may want to pay that premium earlier rather than later.

The first system I applied most of the characteristics to was a loyalty management platform replacement back in 2011. I talk about it in "Micro Services: Java, the Unix Way."[9] Because this project involved a platform replacement with the explicit goal of scaling users and therefore throughput an order of magnitude further than the existing system, we applied some commonsense patterns to the problem. First, it was a huge domain, so we broke it down into smaller capabilities. We wanted to get going quickly in order to prove out our thinking, so we applied evolutionary design. We structured small teams around each of these capabilities, using Conway's law. [Conway's law basically states that organizations that design systems produce designs reflecting the organizations' communication structures.[10]] We implemented a common application protocol to communicate between capabilities—we chose AtomPub and our own user types based on the work that Jim Webber, Savas Parastatidis, and Ian Robinson had been doing in that space.[11] We built a domain-specific language for build and deployment in order to take some of the pain away of using multiple small repositories for the services. We applied a shared-nothing approach to domain logic, meaning that we were always programming to external interfaces rather than reusing another team's domain library.

Since then, I've worked in the scientific-publishing domain and applied the same ideas successfully also in identity management, in retail and investment banking, and in insurance. What most of these programs had in common was an inability to get things done with an existing monolith and a decision to migrate to SOA and specifically microservices.

Mike: In the early 2000s, I worked in a small start-up company competing in the niche of content management for the entertainment industry. We managed the distribution of terabytes of content and eventually branched out into product licensing, too. This was a fast-changing, very custom-fit market with a couple of powerful, established players. To "win" in this space, we needed to be faster than our competitors, so we adopted a handful of practices to do that safely.

While we didn't know the name "microservice" at the time and were only vaguely aware of DevOps concepts, we applied many of the principles and characteristics we now ascribe to the microservice meme. For example, we engineered multiple production releases per day to get features and fixes out faster. We automated things, established test and sign-off practices, and eliminated as many bottlenecks as we could in order to get code into production. We also adopted a design approach that made screen design and workflow fully customizable without changing any code. We created what DDD calls a "ubiquitous language"[12] for our application domain and made sure all client and middleware components could operate within that fixed language. This is similar to establishing vocabularies or domain dictionaries that all developers use as a constraint on the implementation—we all had to operate within the same domain.

To keep things moving quickly without breaking things, we adopted the "no breaking changes" pledge. This was similar to something Jason Randolph talked about in his 2013 Yandex talk "API Design at GitHub."[13] And, to make sure future changes could be handled without breaking older implementations, we adopted a message-oriented, hypermedia-driven approach to the interface. We passed only messages, not RPC function handles, and we included links and forms in all the responses in order to support loosely coupled interoperability between our service components. This made all the required client-centric customization we had to deal with relatively cheap and easy to support.

Nicolai: I've contributed to a number of SOA projects since the early 2000s—for example, in the mobile-telecommunications business.[4] I have no experience with an explicitly named "microservice architecture,"

but similarly to Mike, I would claim that a lot of the microservice principles were used before the name was coined, such as "no intelligence in the enterprise service bus," "no distributed transactions," and, of course, "business-driven service design" (do we still have to say that?). By the way, anybody who claims to have built a microservices-oriented system should first define microservices and how they relate to SOA. For example, is a composed service still a microservice? Can stateful processes (such as handling an order) be represented as microservices?

Cesare: So just as the SOA style was sometimes teased as the "Same Old Architecture," microservices promote proven solutions that have been used before.

Olaf: And the definition landscape is fragmented indeed (see the sidebar, "Defining Microservices"). Anyway, as we all seem to agree on microservices' prospective value as an implementation approach to SOA, can you give any advice on how to "sell" an investment in microservices to business stakeholders?

Mike: The way I talk about selling microservices to stakeholders is pretty simple. Use microservices to enable business goals. That's it. If you can't tie your IT practices to business value, you're in a lot of trouble. I heard a great quote on this subject from Mark Bates at the 2016 Open Source Conference [OSCON]: "A [software] architecture that gets in the way of you getting to market is a bad architecture." That puts it very nicely, I think. Your approach to IT implementation must be grounded in business value and must not inhibit your ability to get features and fixes to the market. That's what drove my small team back in the early 2000s, and I think that applies to every team today, too.

James: In all honesty, I wouldn't try to sell microservices at all. I agree wholeheartedly with Mike; you should focus on business goals and how to enable them, but microservices in and of themselves are not a silver bullet. If your goal is to try to test a new idea in the market, then I would concentrate on the value of experimentation, getting to production early, and learning how to get good at the build-test-learn loop. If your goal is to get faster at what you are currently doing, then I would look at the hidden queues of work-in-progress in your development system, visualize them, and then apply ideas from Donald Reinertsen[14] to optimize that system of work.

Microservices may make up a part of that optimization process, but in my experience it's the way we are organized and how work flows from concept into production that is the first-order factor in getting software into the market. To paraphrase Dan North, your development team could work infinitely fast, but if it takes three months of sign-offs and testing before going into production, you are always going to take at least three months.

Nicolai: And never switch to distribution just for fun. You always pay for distribution. As with any form of loose coupling, a benefit such as more flexibility comes with a price such as more complexity when you're maintaining the system landscape. Note that I mean this in a conceptual sense. A monolithic system might internally be following the rules of distribution. I see a lot of people recommending microservices without telling all the real design problems you will face owing to the nature of distributed systems, such as versioning and error handling.

Service-Oriented Analysis and Design

Cesare: What's your strategy to break up (decompose) a monolith into multiple microservices?

Nicolai: As I just discussed, don't do it if you don't have a problem to solve with the existing monolith. In addition, I would claim that the key is not to break big systems into pieces but to clarify the domains and then clean up. This is usually accomplished by leveraging knowledge that people and teams involved with a system landscape have anyway. But when you start with the change, start business process by business process, not domain by domain.

James: This is worth a book in its own right! I'm afraid I'm going to go with the consultant's answer of "It depends." I've had some success with a number of different approaches, but not all of them are applicable every time. For example, one approach I have used successfully was to use static-analysis tools to understand the existing namespaces of a product and then to gradually refactor those namespaces into discrete Bounded Contexts,[15] duplicating code where necessary, until we were able to start extracting them as their own services. Unfortunately, it rather depends on the age and amount of debt in the code base. I've seen one 1.5-million-LOC product where the only cohesive structure was the whole 1.5 million lines. Refactoring that code base would have required a nontrivial effort (as one of my old physics lecturers used to say).

DEFINING MICROSERVICES

Both service-oriented architecture (SOA) and microservices are overloaded terms; you can find one definition for each term per blogger, book author, and consulting firm. Keeping within the tradition, here are excerpts from our interviewees' definitions.

Nicolai Josuttis positions SOA as an architectural paradigm that improves flexibility when you're dealing with business processes distributed over a large landscape of existing and new heterogeneous systems with different owners. Key SOA concepts are services (and interface contracts), interoperability, and 11 forms of loose coupling, as well as a service classification scheme.[1]

James Lewis (together with Martin Fowler) introduces microservices as

> *an approach to developing a single application as 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.*[2]

The services are independently scalable and can be replaced and upgraded independently, with the objective to speed up the release process.

Mike Amundsen and his coauthors see microservices helping achieve goals such as the speed and safety of change. From earlier definitions, they distill the following seven characteristics.[3] Microservices are

- small
- messaging enabled,
- bounded by contexts,
- autonomously developed,
- independently deployable,
- decentralized, and
- built and released with automated processes.

Despite the commonalities and overlap in these three (and other) SOA and microservices definitions—for example, emphasis on business-driven development, heterogeneity and polyglot programming or persistence, and decentralized design ownership—we observe three positions regarding the relationship between SOA and microservices:

- microservices as a different architectural style competing with SOA,
- microservices as a synonym for "SOA done right,"[4] and
- microservices as an implementation approach to SOA.[5]

A possible synthesis is to view microservices as a substyle refining SOA with additional constraints (for instance, less emphasis on central components such as the enterprise service bus or business-process-centric service choreography), amended with recent advances in service realization and deployment (for example, continuous delivery and DevOps). For a detailed analysis and comparison of microservices characteristics and principles by viewpoint and abstraction level, see "Microservices Tenets: Agile Approach to Service Development and Deployment."[6]

References

1. N. Josuttis, *SOA in Practice: The Art of Distributed System Design*, O'Reilly, 2007.
2. J. Lewis and M. Fowler, "Microservices: A Definition of This New Architectural Term," 25 Mar. 2014; martinfowler.com/articles/microservices.html.
3. I. Nadareishvili et al., *Microservice Architecture: Aligning Principles, Practices, and Culture*, O'Reilly, 2016.
4. M. Stiefel, "What Is So Special about Microservices? An Interview with Mark Little," *InfoQ*, 15 Feb. 2015; www.infoq.com/news/2015/02/special-microservices-mark-litle.
5. S. Newman, *Building Microservices: Designing Fine-Grained Systems*, O'Reilly, 2015.
6. O. Zimmermann, "Microservices Tenets: Agile Approach to Service Development and Deployment," *Computer Science—Research and Development*, 2016; doi:10.1007/s00450-016-0337-0.

Mike: There are two things I find that must be dealt with when refactoring a system: decoupling the shared interface from the internal models and source code, and resizing the system components to better

match the speed and safety requirements of your particular application domain. In the first case (decoupling the interface), you can do that no matter what the size or shape of components behind the interface is. This is almost always the first job I want to tackle. That means establishing standards and practices for defining and implementing interfaces—usually stateless, message-oriented, and hypermedia-driven. Once that's accomplished, you can safely refactor the service provider components themselves without breaking service consumers.

James: A similar approach we have used successfully on projects uses the user interface as the point of decoupling. An example of this would be to write a little bit of JavaScript that can switch between old and new versions of some small element of functionality, build the functionality as a microservice off to the side, and then deprecate and eventually remove the older version in the monolith. Of course this means you have an increase in overall complexity until you deprecate and remove the old version since you now have two things to worry about.

Mike: Indeed. The challenge is that most people want to tackle the code or boundary refactoring without dealing with the decoupled interface, and that leads to a lot of breakage and problems.

James: Absolutely. One of the major issues I've experienced is the difficulty many developers (including me) have in shifting paradigms from in-process calls to calls across a process boundary: "I just want it to be like invoking a method, but 'over there.'"

Nicolai: Looks like we are in agreement again! The cognitive load of designing a remote service interface certainly is higher than that of exposing a local programming interface—a lesson CORBA developers learned early a long time ago. Performance, for instance, can have a major impact on service granularity. Services tend to become either more coarse-grained if the overhead of multiple service calls is too high or more fine-grained if the overhead of processing unnecessary data is too high. In practice, this often leads to some redundancy having both coarse-grained and fine-grained services—an aspect that limits the reuse opportunities: "Why can't we have just one service delivering all customer data?"

Olaf: Talking about decoupling and granularity, can you comment on the role of Eric Evans' DDD[12] in identifying, specifying, and realizing microservices?

James: I'm going to unpack this into two parts. DDD is one of the foundational pieces of thinking in terms of microservices. However, the thinking behind the strategic design aspects of DDD is the most applicable in my mind. Breaking up something big into a set of smaller Bounded Contexts and then building teams around those smaller Bounded Contexts with concrete interfaces at their boundaries and a shared-nothing approach to domain logic originating in those Bounded Contexts allows us to decouple our teams as much as it allows us to decouple our software. This is the going-faster aspect of microservices.

I find the more tactical DDD patterns (Aggregate, Entity, and so on) to be of less importance to me more recently. Teams that successfully build and operate microservices tend to be stable and use the fact that services are small to constantly reassess their fitness and rewrite them as the developers learn more. This is one of the characteristics that Martin and I call out under "products not projects."[5]

I would say that it was a crucial observation from my perspective that as we learn more about the problems we are trying to solve, we should be free to reimplement our existing solutions and not become "stuck" with a suboptimal one due to the prohibitive cost of a large rewrite project. Building things out of smaller components, microservices in this case, helps in this regard since we are not having to chuck everything away every time our understanding increases. Take a look at, for example, Netflix, which famously uses this approach; it has rewritten its platform API a number of times.[16] Languages have changed, and they have moved to a reactive approach too. Of course this is possible if you have a monolith too, but the cost is usually prohibitive.

The second part of my answer is about the how. Teams I've worked with have used a number of techniques to understand different domains. A few examples would be event-storming workshops,[17] whiteboarding, and stacking user stories in piles. A good thing about user stories is that you can group them in any number of lightweight ways. I've also been involved in programs involving significant organizational change where top-down restructuring of business units into product lines has to a large extent created the Bounded Contexts automatically. The service architecture then follows the new business architecture; for instance, business-level value streams and com-

munication paths suggest service interaction and conversation patterns.

Most recently, I've been interested in a model of concentricity in organizational design—organizational units building up from small two-pizza teams to value streams, to capabilities, and so on. My current hypothesis is that building up organizations this way, using Conway's law for our benefit, may enable us to shortcut decisions about integration patterns to use, tools, and so on at the boundaries between the different organizational units. That's a work in progress, though.

Mike: In my experience, scalable software architecture is good at sharing domain understanding. That means either the architecture is designed around a single set of domains or the architecture is designed around the "shared domain understanding" concept itself—that's a much more abstract and more difficult challenge.

I like both strategic and tactical DDD in Evans' approach—especially when it comes to providing guidance on how to design and implement the running code of the system—things such as value objects, aggregates, factories, and so on. But, from my point of view, his material falls short for systems that focus on the interface between components. Erich Gamma and his colleagues have this classic quote in their book *Design Patterns*: "Program to an interface, not an implementation."[18] Application-level protocols such as HTTP, CoAP [Constrained Application Protocol], and others make this really easy to do since they emphasize message passing over fixed IP ports instead of remote procedure calls over TCP/IP.

So, when applying Evan's guidance for distributed-network apps (such as those that run over the Internet), I focus on the interface agreements. Instead of using code-centric objects as the shared understanding at the interface, I use standard message models such as HTML, Atom, Collection+JSON, HAL [Hypertext Application Language], and Siren. These all have varying levels of embedded hypermedia support "designed in." And that makes it possible to safely deal with changes in Bounded Contexts over time. As objects change shape and components modify their Bounded Contexts, services are still responding using the same standard message models but with new hypermedia controls (for example, new links and forms) to support the inevitable changes in Bounded Contexts that happen over the life of a system.

> Top-down restructuring of business units and event-storming workshops can help create Bounded Contexts.

Nicolai: No matter what your forward-engineering or top-down design approach may be, DDD or another business analysis and design method, I am a fan of bottom-up design. Take the Internet as an example. There usually is no central instance to specify which domain has which roles and responsibilities.

According to my experience, in large system landscapes, any centralized enterprise-architecture-management approach trying to solve the problem top-down is a waste of money. Instead, domains evolve or are given by the organizational structure, which eventually matches the system structure (according to Conway's law). So, as soon as people start to discuss use cases or business processes getting realized over multiple domains, you find out more and more about required changes in roles and responsibilities, just because of the way you have to deal with requirements smells.

So, establish the right organizational structures to deal with business domains effectively and efficiently. And then let these structures work on a day-to-day basis, and suited software interfaces will emerge by themselves.

James: This reminds me of what my colleague Evan Bottcher describes as the "inverse Conway maneuver."[19] This approach says, "Build the organization you want; the architecture will follow kicking and screaming."

Nicolai: Nicely said, but in my experience distributed-system landscapes always need a few generalists dealing with cross-cutting aspects such as cross-domain solution development, cross-domain testing, and cross-domain service fulfillment. Unfortunately, managers often see SOA as a project, so that with the end of the SOA transformation, the cross-domain generalists are gone. They usually learn the hard way that SOA

is a strategy. There is no end, and cross-domain generalists are getting part of the new culture, establishing corresponding organizational structures. If management starts to fight against this effect, their SOA or microservices approach will fail. You can't win if you fight against Conway.

Olaf: So Conway is king in the land of service design, but cross-cutting concerns complicate governance. And DDD is helpful but not the only tool required in the microservice designer silver toolbox. You choose your decomposition approach on the basis of the context, vision, requirements, and coupling criteria.

Cesare: Thanks to everyone for the insightful discussion so far. In the next installment, we'll look at additional architectural concerns and how to sustain microservices in the long run.

References

1. S. Newman, *Building Microservices: Designing Fine-Grained Systems*, O'Reilly, 2015.
2. R.T. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," PhD dissertation, Univ. of California, Irvine, 2000, section 2.3; www.ics.uci.edu/~fielding/pubs/dissertation/net_app_arch.htm#sec_2_3.
3. I. Nadareishvili et al., *Microservice Architecture: Aligning Principles, Practices, and Culture*, O'Reilly, 2016.
4. N. Josuttis, *SOA in Practice: The Art of Distributed System Design*, O'Reilly, 2007.
5. J. Lewis and M. Fowler, "Microservices: A Definition of This New Architectural Term," 25 Mar. 2014; martinfowler.com/articles/microservices.html.
6. O. Zimmermann, "Microservices Tenets: Agile Approach to Service Development and Deployment," *Computer Science—Research and Development*, 2016; doi:10.1007/s00450-016-0337-0.
7. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley, 2010.
8. M. Fowler, "Microservice Premium," 13 May 2015; martinfowler.com/bliki/MicroservicePremium.html.
9. J. Lewis, "Micro Services: Java, the Unix Way," *InfoQ*, 4 Jan. 2013; www.infoq.com/presentations/Micro-Services.
10. M. Conway, "Conway's Law"; www.melconway.com/Home/Conways_Law.html.
11. J. Webber, S. Parastatidis, and I. Robinson, *REST in Practice: Hypermedia and Systems Architecture*, O'Reilly, 2010.
12. E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.
13. J. Randolph, "API Design at GitHub," 2013; events.yandex.com/events/yac/2013/talks/42.
14. D. Reinertsen, *The Principles of Product Development Flow: Second Generation Lean Product Development*, Celeritas, 2009.
15. M. Fowler, "Bounded Context," 15 Jan. 2014; martinfowler.com/bliki/BoundedContext.html.
16. B. Christensen, "Optimizing the Netflix API," 15 Jan. 2013; http://techblog.netflix.com/2013/01/optimizing-netflix-api.html.
17. A. Brandolini, "Introducing Event Storming," blog, *Ziobrando's Lair*, 18 Nov. 2013; http://ziobrando.blogspot.de/2013/11/introducing-event-storming.html.
18. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.
19. "Inverse Conway Maneuver," *Technology Radar*, ThoughtWorks, 2016; www.thoughtworks.com/de/radar/techniques/inverse-conway-maneuver.

CESARE PAUTASSO is an associate professor at the University of Lugano's Faculty of Informatics. Contact him at c.pautasso@ieee.org.

OLAF ZIMMERMANN is a professor and institute partner at the University of Applied Sciences of Eastern Switzerland, Rapperswil. Contact him at ozimmerm@hsr.ch.

MIKE AMUNDSEN is the director of API architecture at the API Academy. Contact him at mca@mamund.com.

JAMES LEWIS is a principal consultant at ThoughtWorks. Contact him at jalewis@thoughtworks.com.

NICOLAI JOSUTTIS is an independent consultant. Contact him at nico@josuttis.com.