Matthew Skelton *and* Manuel Pais

Monoliths vs Microservices is Missing the Point—Start with Team Cognitive Load

DevOps Enterprise Summit London 2019

IT REVOLUTION

25 NW 23rd Pl, Suite 6314
Portland, OR 97210

The contents of this eBook are a transcript of the complete presentation given by Matthew Skelton and Manuel Pais, "Monoliths vs Microservices is Missing the Point—Start with Team Cognitive Load" at the DevOps Enterprise Summit London 2019. To view the original presentation, please visit https://www.youtube.com/watch?v=haejb5rzKsM

eBook published 2019 by IT Revolution Press.
For further information on this or any other books and materials produced by IT Revolution Press, please visit our website at ITRevolution.com

Matthew S.: Hi. Good afternoon everyone. It's good to see you here. My name's Matthew Skelton.

Manuel P.: And I'm Manuel Pais.

Matthew S.: And together, we are the co-authors of a new book called *Team Topologies*. We're here today to share with you some insights, advice, and experiences on how to size software services, and the focus of that is team cognitive load.

So, today's talk will look something like this. We'll have a section where we're looking at monoliths and microservices, different kinds of sizes of software. We'll then look at what we mean by team cognitive load. We've actually had this term mentioned a couple of times already today in some of the earlier talks. Manuel will then take us through some case studies. Organizations that have used team cognitive load as a way of helping them to evolve their software systems. And then, right at the end, we'll look at a few tips for getting started with this approach.

So, in the past few years, many organizations have started to adopt microservices as a way of being able to deploy software systems more rapidly with greater focus on specific areas of the system. But there's often lots of debate around what size microservice should be. Should it be ten lines of code? Should it be a hundred lines of code? And it starts to look a little bit like this. It's like kind of a Mortal Kombat thing. So, in the blue, we've got Tammer Saleh who says, "Start with monolith and extract microservices." And then, over the other side of the arena, we've got Stefan Tilkov who says, "Don't start with a monolith when your goal is a microservice." And then the wise words of Simon Brown, who says, "If you can't build a monolith, what makes you think microservices are the answer?"

So, what's going on here? Where should we actually focus? And I think that Daniel Terhorst-North has got it right when he, in his phrase, talks about software that fits in your head. And there's an awful lot of experience and awareness behind that recommendation or that phrase. In the context of teams, if we're thinking about building software in the context of teams, teams owning and running software, we might rephrase this to be software that fits in our heads as a team. But the intent is the same.

Who has yet to buy or read a copy of *Accelerate* (by Nicole Forsgren, PhD, Jez Humble, and Gene Kim). Put your hand up and be shamed. Right. Fine. So you need to get yourself a copy from the stand. Very straightforward. These are the four key metrics from *Accelerate* based on five years' worth of *State of DevOps Reports* and assessment from many thousands of companies around the world. These are the four key metrics that are strongly indicative of high organizational performance: lead time, deployment frequency, mean time to restore (MTTR), and change fail percentage.

The problem is if the software which we're working with does not fit in our heads, these things are going to be very difficult to improve upon. If the lead time is the time from, say, version control to production, if the software is too big, we're likely to distrust the kind of tests, we're likely to want to take more time to find out what's going on, the lead time's going to extend. Same with the prog frequency. If we don't understand the software well enough, are we going to have the confidence to deploy more and more frequently? Probably not. We're probably going to want to restrict how many times we deploy and so on. If the software we're working with is too complex and too complicated, and fails in really awkward ways in production, it's going to be difficult for us to restore that service quickly. So, again our MTTR will extend.

So, if we want to start to move toward these kind of . . . improving these kind of faulty metrics, as recommended by *Accelerate*, then we need to start thinking about the size software that we're expecting teams to work with. Software that is too big for our heads works against organizational agility.

And this is a kind of different starting point compared to how many organizations, many people, have started thinking about software and architecture and so on. Because often in the past, we've thought we've started with bits of technology. We've started with a database. We've started with a message bus. We've started with something else. If we start with the team and the cognitive load for that team, we get some different results.

So, let's have a little look at what we mean by team cognitive load. It was defined in 1998 by psychologist John Sweller as the total amount of mental effort being used in the working memory. And there are three kinds of cognitive load that John Sweller identified: intrinsic, extraneous, and germane. And in the context of software development, we can think of them in these three ways. So, we can think of intrinsic as something like how classes are defined in Java It's something that we kind of . . . it's just a fundamental of how we're kind of working with, in this case, software systems. We don't have that front and foremost all the time. Once we've spent six months or a year doing Java development, that sort of becomes naturally . . . becomes an intrinsic part of how we work.

Extraneous is something that works against what we're doing. Something that is kind of like a distraction. So, how do I deploy this app again? I can't remember. It's really awkward. I've got to set this concrete property. Blah, blah, blah. This is extraneous cognitive load, and it's effectively valueless. We don't want to have this kind of cognitive load on our teams.

Germane cognitive load is load that we have to deal with because it is the part of the business problem we're trying to solve. So, if we're building an app for online banking, then part of the germane cognitive load of the software developer or tester or whoever is building the application to that point might be how do bank transfers work? Because you need to have that kind of load in your head as you're building the software.

So, you can sort of see these in a software context. As intrinsic is kind of like the skills that we bring to the table. Extraneous is stuff to do with the mechanisms of how we do things in a software world. And germane is sort of like the domain focus. It's a bit more involved than that, but that's how you could think of it. And what we're really trying to do is maximize, give the most space to, the germane details, the germane cognitive load. The intrinsic we have to deal with, we can't get rid of it. It's just: we're working with software, we're working with computers, the stuff we just have to know. We're trying to minimize and squeeze the extraneous cognitive load. Get rid of that as much as possible. If possible, just get rid of it entirely so that we've got the most space available for the germane cognitive load, the business focus of the problem we're trying to deal with.

If you want to know more, by the way, about this in some detail, there's a great presentation by Jo Pearce if you search for "Hacking Your Head," then you will find lots of slides, lots of videos, and so on. There's some really good material there.

So, this is the implication of what we've just been talking about. We should be thinking about limiting the size of software, services, and products to the cognitive load that the team can handle. So, we're starting to take a socio-technical approach to building our software systems here. We don't just pretend that we can throw any kind of software architecture or design or technology at a team and they'll just have to deal with it. We're actually using the kind of, if you like, constraints or properties of the human systems that we have in our organizations and working with them to produce more effective delivery and more effective software systems.

So this again is software that fits in our heads. This is quite a different approach to thinking about, kind of, software boundaries. This will feel very unfamiliar to many people. Not to everyone, there are organizations already doing this, as we'll see very shortly. But it does feel a bit unusual.

When we're talking about teams, we're talking about a group of people, probably less than about nine people in size. There are evolutionary reasons for this. Some organizations have found patterns where you're able to bring two of these kind of teams together in close harmony. You can think about a rugby team. You have effectively got two closely operating teams together. You've got the forwards and the people at the back. I don't play rugby, but I spoke to people who do and they do say it feels a little bit like there's two separate teams working really closely together.

So, some organizations have found ways in which they can do that, but generally speaking, we're talking about a cohesive, long lived group of people who work together on the same set of business problems for an extended period. And that group of people is less than about nine.

We've heard from many of the talks this morning about ownership of software services and how important that is. We need to move to the point where every service must be fully owned by a team with sufficient cognitive capacity to build and operate it. In the words of Andy Burgin from Sky Betting and Gaming earlier on, it was you build it, you run it, you fix it, you support it, you diagnose it, and so on. That's what we're talking about here. There's no services, there's no products which do not have an owner.

We know that there are techniques to help us do that, this kind of stuff. We've got techniques like mobbing, which apply to the whole team, which help the team to own that service. We've got techniques like domain driven design (DDD) to help us choose domain boundaries in an effective way that really works for the business context.

We've heard many people talk about the importance of developer experience. Particularly when building a platform, making sure that platform is very compelling and very easy and natural for product teams and development teams to use. So, we're making sure we're explicitly addressing developer experience when we're… particularly when we're building a platform, but to be honest, when we're doing anything inside our organization where other people need to use our software.

But we also need to think about the operator experience. What about the people who actually need to run this stuff? People who are on call? How easy is it to diagnose these systems and so on. If it's . . . If we've built a system that's fine for our team but we've handed it over to another team and it's terrible, it's really difficult to operate that stuff, the cognitive load is way too high. We're in a bad place. We need to focus on operability to make our stuff work.

And another technique what we've put in the book is called thinnest viable platform, which is an approach where we explicitly define what the platform looks like. So, again, from Andy Burgin's talk this morning, there's a really nice slide where he showed the very beginning of their kind of platform evolution. They had a page, a wiki page, which defined exactly what that platform was aiming to do and listed the services it provided.

So, being super explicit about what our platform is, is important. It's also important to make sure that it's not bigger than necessary, hence "thinnest viable." If you're a start-up and you're quite small and there's only maybe ten, fifteen people in your organization, then the underlying platform is going to be something like AWS or Azure or Google Cloud or whatever. You might decide to build an extra layer, platform layer, on top of that, but your platform might simply be a wiki page listing the five services that you are going to use from AWS. And if you don't need to build anything more, don't build anything more. That's enough. That is your thinnest viable platform: just a wiki page with the list of five services.

We're not trying to build a huge great thing. We need to make sure that the . . . whatever we build is compelling to use, has strong developer experience. We're treating product teams or what we call stream-aligned teams, as . . . we're treating them as customers. We're treating them as people whom we need to speak to to understand what they need, and we need to be set up to meet their needs.

So, I've talked about a few different team types. In the book, we've identified four different kinds of team, which as far as we can see are really the only types of teams that are really needed in this kind of context in building modern software systems. And the first team type is the most fundamental and this is the stream-aligned team. The team that is aligned to part of the value stream for the business ,and they have end-to-end responsibility for building, deploying, running, supporting, and eventually retiring that slice of the business domain or that slice of service. And really, the other kind of teams listed below are effectively there to reduce the cognitive load of the stream-aligned team. That's how we see it.

If we've chosen our domain boundaries well, the stream-aligned team should have everything they need to deploy changes for that business, that part of the business system.

But they can't do everything. They need some supporting services, from a platform for example. We heard a great talk from Tom this morning about the platform at ICV. We need some support from the platform so we don't have to think about how do we spin a Kubernetes cluster? Because that would be too much increased cognitive load compared to deploying something more business focused.

Likewise for a complicated-subsystem team. If there's a part of the system where, let's say, in the case of media streaming we need to write a specialized video transcoding component, we'll probably hire some people with PhDs in math or something like this and get them to work on a complicated subsystem. We're taking the cognitive load off the stream-aligned team to focus on more kind of custom end-to-end experience.

Enabling teams kind of helps to up-skill the stream-aligned teams. Perhaps on a temporary basis. Typically on a temporary basis. And also to detect if there's any gaps in the platform or gaps in what the stream-aligned teams are expected to do.

So, this [see Figure 1] is maybe an organization here where maybe we've got three stream-aligned teams. We've got a platform underneath. We've got a complicated subsystem on the left in red. And toward the right-hand side we've got one of those enabling teams kind of facilitating two of the stream-aligned teams. Perhaps they're moving from one container platform to another. Something like that. So, they're just trying to get up to speed.

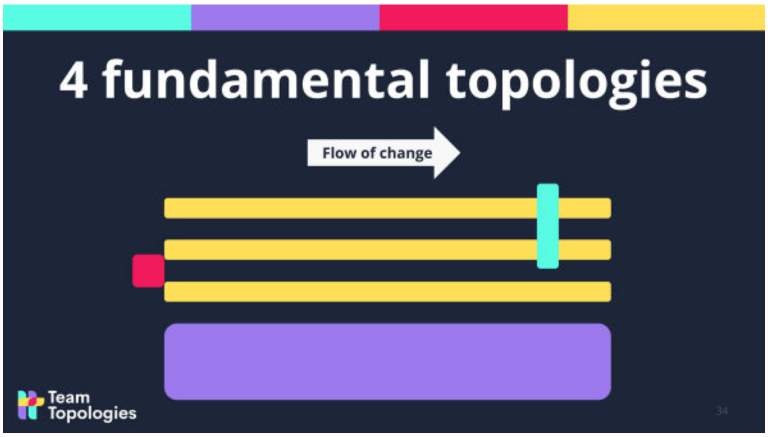

Figure 1: The Four Fundamental Topologies

Another key idea in the book that we've identified is the need to be much more explicit about the ways in which teams interact. Because what we can see from our experience, and what we hear from other people talking about their experience, that in many organizations teams don't understand why or how they should interact with other teams. So, what we've defined is three interaction modes, and part of the purpose of these three interaction modes is to help reduce confusion and effectively reduce the, kind of irrelevant, cognitive load so that it's easier for teams to understand how they should be operating effectively.

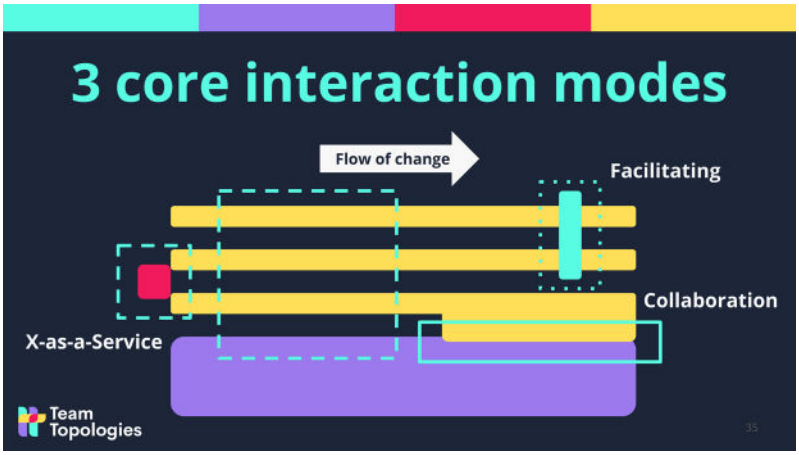

Figure 2: Three Core Interaction Modes

So, if the complicated-subsystem team is transcoding components, let's say . . . If that team is busy building that, if we set up the expectation that they're simply providing that component, if you like, as a service to these two teams at the bottom then those . . . all those three teams involved in that interaction have a clear understanding about how they're supposed to interact, how they're supposed to provide something or consume something.

So, we've minimized the kind of cognitive load around how we should kind of operate as a team. And similarly, the stream-aligned team at the bottom here is currently collaborating with the platform to discover something about, let's say, logging or a better way of doing committees, something like this. They know that for a period of time, their cognitive load is going to be higher because they're working together closely with another team.

But perhaps after, say, three months, we finish that discovery and we go back to consuming the container platform as a service. So, the mechanisms here that . . . if we define much more clearly ways of working with other teams, we're actually able to address cognitive load too and minimize that in different parts of the organization.

So, we're now going to look at some case studies from organizations.

Manuel P.: All right. So, I'm going to talk about the case studies from the book. The first one is a large, worldwide retailer. And they're still growing into new markets. And back in 2016, they decided they wanted new mobile site for one of these new markets. They put a team together from scratch. A cross-functional team with business people directly involved in the team. They had all the technical skills to kind of have this end-to-end ownership that Matthew was talking about. They had good DevOps practices. Everything was in the cloud. Kind of the typical success story that you would include in a presentation like this. And so given that success, they were able to quickly release working versions of the mobile website and then iterate frequently.

So, after a while they were asked to do the same for a new market. New mobile site. Though they wanted this to be rather independent, that it could evolve the different sites for different markets more or less independently, but in the backend they started to have a need for a little bit more complexity. They needed a content management system so they could upload content to different sites. But overall, this was working quite well still.

And of course, over time, they were asked to do even more markets, more sites. And the backend started to get a little bit more complicated. They needed a subsistent to handle product management, product catalog. So, different markets are going to have different sets of products and versions available, and pricing, etc. So, they needed to manage that. They also started this framework which is a collection of common services to all the sites. Things like searching for a product or uploading static files to a CDN, things like this that all the sites would need, but you wouldn't want to repeat it for every code base.

So, I think you can probably tell what's happening here. The system is growing, and the team is growing along with it. So, by now, they had far more people than in the beginning. And it's becoming a little bit of a monolith, right? And some of the people on the team start to actually realize, now we also have different work streams going through the teams. So, you have maybe feature requests for one of the market's sites. Other feature requests for other markets. You also might have changes that need to be done in the CMS for the content editors and so on. And the fact that the system was a little bit monolithic by now meant that these work streams were kind of impeding each other. There were dependencies, and they were actually slowing down pace of delivery. The thing that had made them so successful in the beginning was now harder to achieve.

So . . . Particularly two people in this team who had a kind of senior architect role started to realize this, and even though the team worked quite well together, they were a high performing team, if you like, they noticed these dependencies. And also, people had to start specializing in certain parts of the system. While before it was pretty fluid, you'd get a change request or a feature and it would go... you would know exactly which parts of the system to change and get it out, now people were starting to specialize in specific parts.

So, these two people, these two senior architects, proposed to split the team in two, and they got a lot of pushback because the team members felt that they were working well together. But eventually they did this. So, they got into this pattern Matthew mentioned, kind of a paired team. So, obviously there was a lot of communication going on, on a regular basis. But after doing some refactoring of the system and re-architecting a bit, they were able to kind of split into two teams essentially. One team more focused on the customer facing applications and markets. And the other team focusing more on the CMS and this framework.

So, this worked quite well for them. And now these two teams were able to deliver more independently. There was still obviously some correlation between the roadmaps for these two teams and they had this communication going on, on a regular basis, but they were much more independent than at this point.

So, they realized at this point that there was too much cognitive load. The system was too large to handle as efficiently as before. And from what we've heard, they went on to actually further break down this team. So, I believe now they have smaller teams aligned to markets on the customer facing side and they have split the CMS and the frame, which is a kind of platform team, as Matthew was mentioning, with common services. This worked quite well for them.

So, the key point here is that as they grew and were successful, the system became larger and the team became larger. And things were starting not to work as well. Their flow of work was getting blocked, or at least significantly delayed. The critical thing here is that some people in the team were listening to the signals that something was not as efficient as it was before. So, the software was getting too large in this kind of monolithic architecture. Some people were overspecialized. So, if you read *The Phoenix Project*, it's kind of the brand syndrome where only this person or this couple of people know how to change that part of the system. So, you're introducing this dependency. Even inside one team, this dependency that only when those people are available we'll be able to get this out the door. And overall just increasing the need to coordinate releases and make sure when is that part done so I can do this other part. And introducing delays in delivery.

But it's not always just about the size of the software that teams are responsible for, there are other types of responsibilities. In the case of OutSystems, who is one of the leading low-code platform vendors in the world, a few years ago, they started an engineering productivity team. This team, in the beginning, were responsible to… they worked as an enabling team around build and continuous integration and also test automation. That's what they started with. Their goal was actually to reduce cognitive load for the other engineering teams who were in fact their customers, if you like. So, they were helping them adopt good practices around these areas. Set up tooling in a good way. And just overall helped the engineering teams increase their maturity in this areas.

So, again, they were quite successful. And what happened was they took on more domains. Particularly, infrastructure automation and CD (continuous delivery) enablement. And the team grew to cope with that. And the interesting fact here is that as this was happening the other engineering teams were getting really more mature, more advanced, in the way they used test automation, CICD, etc. And so they were coming back to them with requests for help that were much more specific, much more domain specific for those teams.

What this productivity team was facing now was a large number of requests across different domains and coming from different teams with specific needs. They were barely able to keep afloat, let's say, and respond on a timely enough basis to these requests. And inside the team, what happened was that it became very difficult for any one team member to work to understand all these different domains. So, people were in practice working on only one or perhaps two domains, and motivation went down significantly. Some of the people felt like they didn't have enough effort available to actually master the domains that they were supposed to support and understand in detail. And at same time, they were spending a lot of time in planning meetings. In standup meetings, where most of the things being discussed were not directly related to the work that they were doing.

So, at this point, and this is quite recent, in late 2018 they took . . . they made a bold decision to split into smaller teams. So, almost micro-teams where any one team was only responsible for one of these domains. And the early results were quite positive. So, motivation went up, people felt like they had more autonomy to actually decide what the priorities for their domain of responsibility were. Also, they interacted much more closely with the other engineering teams, their clients if you like. And they were able to really understand, what are the problems we have? What are the solutions, the best solutions I can find for you? And they had a little bit of breathing space to actually master this domain, understand good practices, perhaps come to conferences like this and get to know what other people were doing. And so the motivation really went up. And there was a feeling of shared purpose inside each of these teams.

And obviously, there were still issues and maybe requests that were crosscutting across some of these domains as they are closely related, but it turns out those are kind of the exception. So, when that happens, the people from different teams will come together. If needed, they will create a temporary team to work on that specific problem or need. And then, go back to their original teams.

In fact, before they were optimizing for this situation, which is the exception that their actual needs and requests are across multiple domains. So, this has worked quite well for them for now. And you also can see there's still communication going on between different teams, but the required bandwidth there is much lower. So, the key is that it's not always about software size, but actually aligning the number and complexity of the domains that the teams are responsible for to their cognitive capacity.

And if you aim for this kind of pattern, with smaller teams with high cohesion internally, high communication internally, and shared purpose, then you need some synchronization with other teams, but that can be much lower bandwidth. So, you don't need to be communicating across all teams all the time. That can work quite well.

And then, finally, they again were listing to the signals that what worked for us in the past, in the beginning, is now becoming a problem. So, awkward interactions. Some people were not really invested. Some people may be almost burned out because they were trying to really keep up with all these different domains. So, we'd have to put in a lot of extra time to actually understand all of this. And definitely frequent context switching inside the team.

The last example, it's not from the book, it's actually from a recent talk again from Sky Betting and Gaming. And besides getting a slide of a cat in the presentation, it's also just to show you, is this always the good pattern to split into smaller teams? Well, not necessarily. In this case, they decided to keep a kind of large team of twelve people because they had different applications. So, some older applications that were making money today and new applications, more experimentation, trying new markets.

And what happened was that within the same business domain, the demand for working on one part, older applications or newer, would change over time. So, in one quarter, maybe we need to increase the resilience of the older systems most of the time. So, spend most of the time on that. Next quarter, maybe we want to push out new applications and try new things so, it made sense to keep the same team, but within the team there were clear work streams and people knew, now we're focusing on this part, this older systems or the older systems.

Matthew S.: So, how do we get started with this kind of approach? A few ideas here. Simply speaking, just ask team members, just do a survey of members in a given team how well they understand the software they're working on. Give it a score of one to five or something like this. And just get a very rough idea of which teams are currently struggling with the cognitive load of the systems they're being asked to own and develop. Could there be some things that are candidates for pushing into a platform? Don't rush ahead and do it, but come up with a candidate list to start with and have some conversations. We're looking for missing skills or capabilities. That could be, it could be that within the team they are actually missing skills. It could be that the organization as a whole is missing skills.

If we adopted these three team interaction patterns that we saw earlier on... So, a kind of closed collaboration, so we know our cognitive load's going to be higher. Or x-as-a-service, where we know we're just supposed to consume something. Or if we're facilitating...So, we're kind of helping or being helped. What would happen if we adopted these patterns? How would teams actually react and behave in this context? Because you need to sense your organizational situation. How you're maturity or the dynamics within your organization… as to where to start to apply some of these sort of practices. Don't just rush in and do it.

Is your platform well defined? If not, go ahead and define it and really quite carefully. You'll probably be surprised that there's far more services that are actually being run by a small group of nearly burned out platform engineers. And so, it's time to do something about that. What is the thinnest platform that could work in your context? It doesn't have to be thin, but the thinnest and no more.

So, as mentioned, here's the book. We've got the signing at half-past seven this evening. It goes on sale in September. You can pre-order now if you go to teamtopologies.com/book. And so, bookstores all over the world are currently stocking it which is great. We've got some training if you're interested, give us a shout. And you can sign up for some news and tips if you go to teamtopologies.com. So, thank you very much everyone for attending today. Hoping it's useful.

About the Authors

MATTHEW SKELTON has been building, deploying, and operating commercial software systems since 1998. Head of Consulting at Conflux, he specializes in Continuous Delivery, operability and organization design for software in manufacturing, ecommerce, and online services, including cloud, IoT, and embedded software. He currently lives in the UK.

MANUEL PAIS is a DevOps and Delivery Coach and Consultant, focused on teams and flow first. He helps organizations adopt test automation and continuous delivery, as well as understand DevOps from both technical and human perspectives. Manuel has been in the industry since 2000, having worked in Belgium, Portugal, Spain, and the UK. He currently lives in Madrid, Spain.