

MINI-BOOK
MB80 – JULY 2020



ORGANIZATION DYNAMICS WITH TEAM TOPOLOGIES

Team-sized software

Team-first tools and skills

Curated team interactions

Key industry insights in 6 articles

In association with TechBeacon



Organization Dynamics with Team Topologies

Team-first software delivery

Effective software teams are essential for any organization to deliver value continuously and sustainably.

Team Topologies provides a practical, step-by-step, adaptive model for organizational design and team interaction, treating teams as the fundamental means of delivery, where team structures and communication pathways are able to evolve with technological and organizational maturity

All the articles first appeared on TechBeacon.com and are reproduced here with permission.



In this mini-book:

	How to break apart a monolith without destroying your team	3
	Forget monoliths vs. microservices. Cognitive load is what matters	10
	Why teams fail with Kubernetes — and what to do about it	19
	How to find the right DevOps tools for your team	27
	Why you should hire DevOps enablers, not experts	39
	Are poor team interactions killing your DevOps transformation?	45

How to break apart a monolith without destroying your team

Matthew Skelton, co-author of *Team Topologies*

Many organizations try to increase business agility by splitting apart existing software systems into smaller chunks, believing that this enables safer, more rapid changes. But when moving from a monolithic software system to [more loosely coupled services](#), you must consider how the new architecture will affect the teams involved in building your software.

Without taking into account the team angle, you risk splitting the monolith in the wrong places, or even creating a complex, coupled mess of unmaintainable code — what's known as a "[distributed monolith](#)."

When I help organizations decouple their large systems into smaller segments, I take an approach to splitting up monoliths that starts with the teams, rather than the technology. Here are some of the common patterns and techniques I have adopted that I hope you'll find useful. But first let's step back a bit.

What is a monolith?

The word monolith literally means "single stone" in Greek—a big slab of stuff that's heavy and difficult to work with. In the software world, there are many different kinds of monoliths, and each requires a different approach to break it apart. Here are six of the more common types:

1. **Application monolith:** A single large application, with many dependencies and responsibilities, that possibly exposes many services or different user journeys.

Team-sized software

2. **Joined at the database:** Several applications or services, all coupled to the same database schema, making them difficult to change.
3. **Monolithic builds:** One gigantic continuous integration (CI) build that's done just to get a new version of any component.
4. **Monolithic releases:** Smaller components bundled together into a "release."
5. **Monolithic model:** Attempted language and model (representation) consistency across many different contexts. "Everyone can work on anything," leading to inconsistent or leaky domain models.
6. **Monolithic thinking:** One-size-fits-all thinking for teams that leads to unnecessary restrictions on technology and implementation approaches between teams.

This is not an exhaustive list; you may have other kinds of monoliths (or too-tight coupling). So before you start splitting your monolith, identify which kind you're dealing with, then invest time in good decoupling.

Some organizations have taken the time and effort to split up an [application monolith](#) into [microservices](#), only to produce a monolithic release farther down the deployment pipeline, wasting an opportunity to move faster and safer. To avoid creating downstream monoliths, always be on the lookout for the different kinds of monolithic software listed above.

Match your organizational architecture with your software architecture

Several studies have confirmed the core message of [Conway's Law](#) that "Any organization that designs a system ... will produce a design whose structure is a copy of the organization's communication structure." There are many subtleties to this in practice, but it boils down to this: If the intercommunication between teams does not reflect the actual or intended communication between software components, the software will be difficult to build and operate.

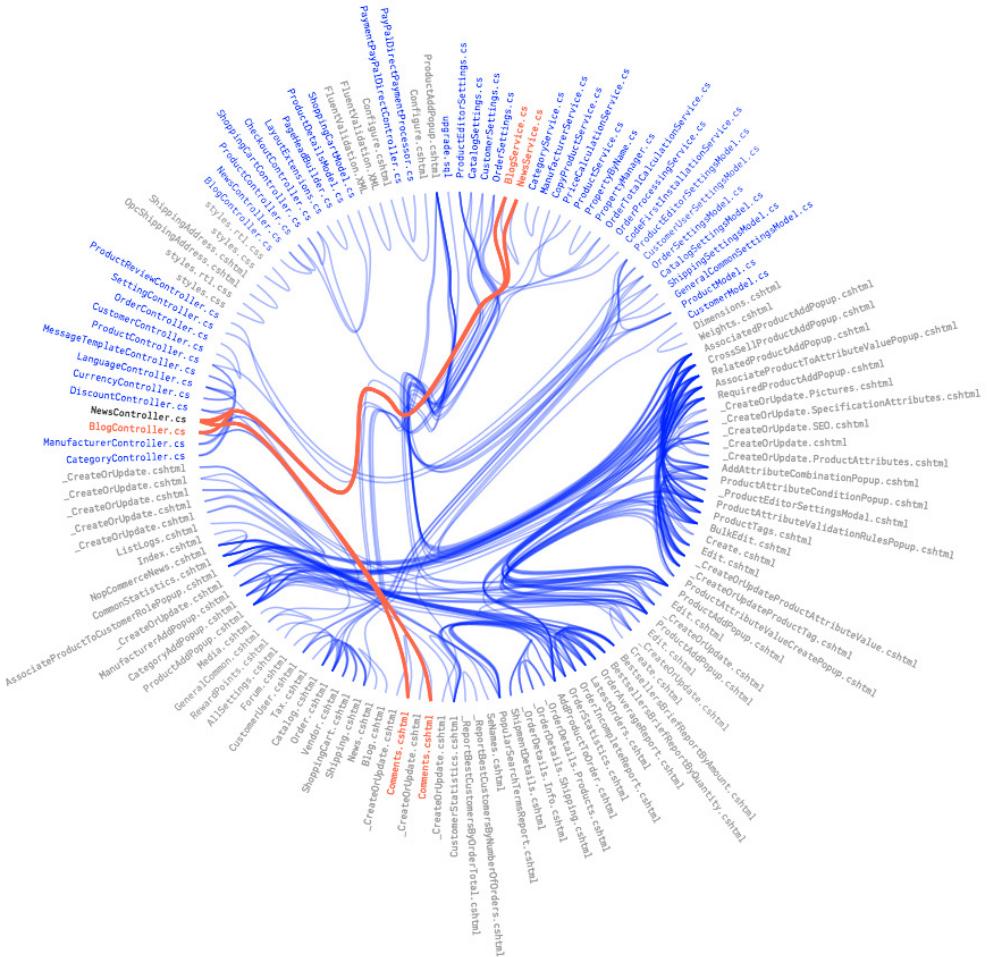
You can use "[Reverse Conway](#)" — changing the team structure to match the

Team-sized software

required system architecture — together with techniques such as [domain-driven design](#) (DDD) and code forensics, to reshape team responsibilities to align with [the software architecture](#) you need to produce in order to clarify boundaries and improve the development and operation of your systems. But how do you know where to split the monolith safely?

Use code forensics to help identify hidden coupling

By splitting up your monolithic code into smaller services, you may be able to make responsibility boundaries clearer, although that doesn't happen automatically. You can use tools such as [Codescene](#) and [Code Maat](#) to analyze your codebase and detect not only standard code metrics, such as cyclomatic complexity and static coupling, but also [temporal coupling](#) — places in your codebase where apparently unrelated files tend to change at the same time — as shown in the repository history.



Temporal coupling analysis by Codescene (with permission from Adam Tornhill)

Team-sized software

In his book *Your Code as a Crime Scene*, Codescene and Code Maat creator Adam Tornhill explained how to use police forensics techniques to analyze and understand the evolution of codebases. For example, he said that "information-poor abstract names are magnets for extra [unwanted] responsibilities." How you name things really does matter, since badly chosen names tend to accrete extra code, making your software harder to work with. Tornhill's new book, *Software Design X-Rays: Fix Technical Debt with Behavioral Code Analysis*, takes these ideas even further. I highly recommend both titles.

Cognitive load for teams determines the size of subsystems

For safe monolith splitting, it is crucial to consider the cognitive load on each team that works with your software. Cognitive load, as defined by psychologist John Sweller, is "the total amount of mental effort being used in the working memory." So cognitive load is important in activities that require mental agility—such as software development.

The maximum effective size for a software team is about nine people, and the maximum cognitive load for any given team is the combined and amalgamated capacity of all team members. One team's maximum cognitive load will differ from others. For example, a team of experienced engineers will have a higher cognitive load than will a team of less experienced people. But there is still a maximum effective size for every subsystem, and that is smaller than many software monoliths.

That means you should limit the size of each subsystem to be no greater than the cognitive load of the team that's building it. That's right: The size and shape of your software should be determined by the maximum cognitive load of your teams. By starting with the needs of each team, you can infer a software and systems architecture that best suits your team members.

A recipe for splitting monolithic software

Now that you have taken Conway's Law into account, used Code Maat or a similar tool to analyze your codebase for temporal coupling, and limited

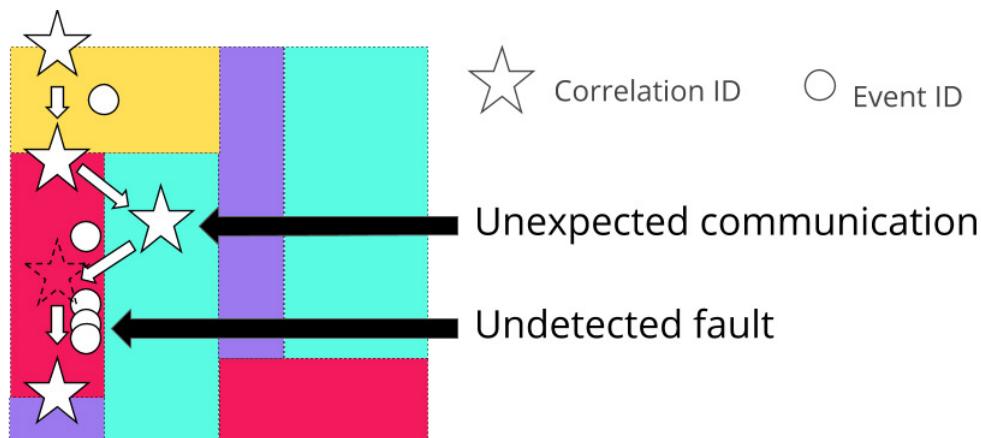
Team-sized software

the maximum size of each subsystem to match each team's cognitive load, you're ready to begin splitting your own monolith. But first let's validate a few assumptions.

Are you certain that your existing monolith works as expected? Are the internal responsibilities between packages/namespaces/modules neat and well-defined? What about subtle bugs that might become serious when you move from in-process calls to cross-machine HTTP calls?

To answer these questions, you need to instrument the code using modern logging, tracing, and metrics techniques to produce rich data about exactly how the software works at runtime. Specifically, you can use [event ID techniques in your logging](#) to detect unexpected actions and states reached in the code, along with call tracing. You can use tools such as [OpenTracing](#) or [Zipkin](#), and/or application performance monitoring (APM) tools to detect the exact code path used during a request or execution path.

These techniques will probably highlight areas where your subsystems are communicating unexpectedly, or they could find undetected fault conditions. Fix those problems before splitting your code. Otherwise, the problems of these extra calls and errors will be exacerbated when you move to a distributed microservices model.



Detect unexpected communications and faults in a monolith using logging and tracing.

Once you've fixed any unwanted calls and errors, you can begin to align a slice of the monolith to what is referred to in DDD as a "[business domain bounded context](#)" — a segment of the functionality within which the terminology is consistent and that has a single responsibility, such as taking payments or

Team-sized software

rendering a document. Where possible, split by DDD bounded context.

But sometimes you'll need an alternative split line (what I call a "fracture plane"), such as splitting by technology or risk. For example, to meet regulatory compliance (such as PCI-DSS), you may need to split along a data boundary. Similarly, to help achieve performance isolation—for a high-volume ticket-booking system, for example—you may need to split out technical aspects of the booking flow.

You can then split off the new team-aligned subsystems or services piece by piece, each time looking at the rich log data and metrics data you can use to validate your assumptions about how the software is working before and after you split out the code. You should:

1. Instrument the monolith using logging, tracing, and metrics.
2. Understand the data flows and fault responses and fix any problems.
3. Align teams to available segments based on suitable fracture planes.
4. Split off segments one by one, using logging and metrics to validate changes.

After you've split a segment from the monolith, ensure that the new segment has independence in every area, including a separate version control repository, a build and deployment pipeline, and probably either separate servers (if you're using virtual machines) or pods (when using Kubernetes). The new segment is independent from the monolith and other segments, enabling the team responsible for each segment to work independently.

Make your move: How to get started

Moving from monolithic software to smaller, decoupled services helps you release more rapidly and safely. But to [avoid creating a complex, distributed mess](#), first consider how teams will build and run the new services. Conway's Law warns that communications between teams will drive the new, decoupled architecture, and your new services should not be larger than the cognitive load of each team.

First, identify what kind of monolith you're dealing with. Then, before you split the code, use a code forensics tool such as Code Maat to identify temporal

Team-sized software

coupling. Use modern logging, tracing, and metrics tools that can identify unexpected calls and faults.

Only then can you identify suitable fracture planes within the code that can act as sensible split boundaries. Finally, split off segments one by one, validating system behavior with logging and metrics at each stage.

For more on monoliths and teams, watch my presentation from the Velocity Conference EU 2016. Thanks to Adam Tornhill for his input on code forensics, Daniel Bryant for early feedback on this material, and Chris O'Dell for additional insights.

[Original:

<https://techbeacon.com/app-dev-testing/how-break-apart-monolith-without-destroying-your-team>]

Forget monoliths vs. microservices. Cognitive load is what matters

Matthew Skelton and Manuel Pais, co-authors of Team Topologies

The “monoliths versus microservices” debate often focuses on technological hinking organizations are beginning with the team's cognitive load as the guiding principle for the effective delivery and operation of modern software systems.

Excessive cognitive load works against effective team ownership and supportability of software. Here’s why, and how to approach the problem.

Overview: Monoliths and microservices

Many organizations are moving from traditional, monolithic software architectures to designs based on microservices and serverless, allowing them to take advantage of newer runtimes that help teams to take ownership of software services.

However, it can be difficult for software architects, team leads, and other technical leaders to assess the “right size” for these services. Should a microservice be limited to 100 lines of code? Should you start with a monolith and extract microservices, as Tammer Saleh recommends, or start with microservices from the beginning, as advised by Stefan Tilkov? How do you avoid what Simon Brown calls a “distributed microservices big ball of mud”?

During the research for our book (*Team Topologies: Organizing Business and Technology Teams for Fast Flow*), and working with clients in different parts of the world, we realized that many organizations fail to consider an important dimension in the decisions around the size of software services: team cognitive load.

Team-sized software

Most of the confusion around the sizing of services goes away when you reframe the problem in terms of the cognitive load that a single service-owning team can handle, as you'll see below.

How to define cognitive load

But first, here's what we mean by cognitive load and how this applies to teams. Psychologist [John Sweller](#) defined cognitive load as "*the total amount of mental effort being used in the working memory*," and went on to describe three different kinds of cognitive load:

1. **Intrinsic cognitive load**, which relates to aspects of the task fundamental to the problem space. Example: How is a class defined in Java?
2. **Extraneous cognitive load**, which relates to the environment in which the task is being done. Example: How do I deploy this component, again?
3. **Germane cognitive load**, which relates to aspects of the task that need special attention for learning or high performance. Example: How should this service interact with the ABC service?

Broadly speaking, you should attempt to minimize the intrinsic cognitive load (through training, good choice of technologies, hiring, pair programming, etc.) and eliminate extraneous cognitive load (boring or superfluous tasks or commands that add little value to retain in working memory). This will leave more space for germane cognitive load (where "value-added" thinking lies).

For a great overview of how cognitive load applies to software development, see the article "[Managing Cognitive Load for Team Learning](#)", by Jo Pearce.

Cognitive load applied to teams

When you apply the concept of cognitive load to a whole team, you need to limit the size of the software system on which the team is expected to work. That is, don't allow a software subsystem to grow beyond the cognitive load of the team responsible for it. This has strong and quite radical implications for the shape and architecture of software systems: Software architecture becomes much

Team-sized software

more “team-shaped” as you explicitly consider cognitive load as an indicator of supportability and operability.

The drive to minimize extraneous cognitive load also leads to the need to focus on developer experience and operator experience. By using explicitly defined platforms and components, your teams will be able to reduce their extraneous cognitive load.

Some organizations have even begun to use cognitive load as an explicit input into software architecture and system boundary decisions.

Why you should use team cognitive load to right-size microservices

In a world of “[You build it, you run it](#),” where the whole team is responsible for the successful operation of software services, it is imperative to remove unnecessary barriers to team ownership of software. Obscure commands or arcane configuration options increase the (extraneous) cognitive load on team members, effectively reducing their capacity for acquiring or improving business-oriented aspects (germane cognitive load).

Another typical example is waiting for another team to provision tickets for infrastructure or to update configurations. This interrupts the flow of the dependent team, again resulting in a reduction in the effective use of cognitive capacity.

Reduced team cognitive capacity puts a strain on the team’s ability to fully own a software service. The team is spending so much time dealing with complicated configuration, error-prone procedures, and/or waiting for new environments or infrastructure changes that it cannot pay enough attention to important aspects of testability or runtime edge cases.

As software developer [Julia Evans](#) says, [reducing cognitive load for your team means setting interface boundaries](#). Every techie at your organization doesn’t need to be a Kubernetes expert.

Put another way, by ensuring that the cognitive load on a team is not too high,

Team-sized software

you have a better chance to enhance the supportability and operability of the software on which your the team is working. It can better own its services, because the team understands them better.

Three ways to reduce team cognitive load and improve flow

There is no magic formula for reducing cognitive load for teams, but having worked with many large organizations around the world (including in China, Europe, and the US), we recommend three helpful approaches: well-defined team interaction patterns, independent stream-aligned teams, and a thinnest viable platform.

1. Create well-defined team interaction patterns

Too often in organizations, the relationships between teams are not well defined or understood. As [Russell Ackoff](#) said, problems that arise in organizations “[are almost always the product of interactions of parts, never the action of a single part.](#)”

You've likely heard complaints such as “Why should we have to collaborate with that other team?” or “Why doesn't that team provide us what we need?” These are signs that the team interactions within the organization are ambiguous. In our Team Topologies book we identify three core team interaction modes to help clarify and define how teams should interact:

- 1. Collaboration:** Working together with another team for a defined period of time to discover new ways of working, new tools, or new solutions.
- 2. X-as-a-service:** Consuming or providing something “as a service,” with a clear API and clear expectations around service levels.
- 3. Facilitating:** Helping (or being helped by) a team to gain new skills or new domain awareness, or to adopt a new technology.

With these well-defined team interactions patterns in place, you can begin to

Team-sized software

listen for signals at the organization level for team interactions that are working well and those that are not, including problems with cognitive load.

For example, if a collaboration interaction goes on for too long, perhaps it's a signal that some aspect of the technology would be better provided as a service by a platform.

Similarly, if one team expects to consume a monitoring tool "as a service" but constantly needs to work with the providing team to diagnose problems, this could be a signal that there is too much cognitive load on the consuming team and you need to simplify the API.

2. Use independent, stream-aligned teams

It is increasingly common in large and small organizations to see small, cross-functional teams (with a mix of skills) owning an entire "slice" of the problem domain, from idea to live services. Such teams are often called product or feature teams.

But with the coming-of-age of IoT and ubiquitous connected services, we call them "stream-aligned" because "product" loses its meaning when you're talking about many-to-many interactions among physical devices, online services, and others. ("Product" is often a physical thing in these cases.)

Stream-aligned teams are aligned to the stream of change required by a segment of the organization, whether that's a line of business, a market segment, a specific geography, or a government service.

It is hugely important to ensure that stream-aligned teams can analyze, test, build, release, and monitor changes independently of other teams for the vast majority of their work. Dependencies introduce a substantial amount of cognitive load (e.g., waiting for other microservices or environments to be able to test, or not having microservices-focused monitoring).

Ensuring that stream-aligned teams are substantially independent in their day-to-day flow of work removes unhelpful extraneous cognitive load, allowing teams to focus on the intrinsic and germane (domain-relevant) aspects of the work. Part

Team-sized software

of this independence comes from being able to use an effective platform.

In larger organizations it's useful to align two or three teams in a close partnership when delivering large, complicated systems. That close relationship helps to avoid one team waiting on another.

Obviously, teams do depend on other services and associated teams for providing infrastructure, runtime APIs, tooling, and so on. But these dependencies don't block the flow of work of a stream-aligned team. Being able to self-service new test environments, deployment pipelines, or service monitoring are all examples of non-blocking dependencies. Stream-aligned teams can consume these independently as needed.

3. Build the thinnest viable platform

Stream-aligned teams should expect to consume services from a well-defined platform, but avoid the massive, unfriendly platforms of yesteryear. Instead, build the thinnest viable platform (TVP): the smallest set of APIs, documentation, and tools needed to accelerate the teams developing modern software services and systems.

Such a TVP could be as small as a single wiki page that defines which public cloud provider services other teams should use, and how. Larger organizations might decide to build additional services atop an underlying cloud or IoT platform, but those extra services should always be "just thick enough" to accelerate the flow of change in stream-aligned teams, and no thicker.

Avoid the frequent mistakes of the past, when internal platforms were bloated, slow, and buggy; had terrible user experience; and — to make matter worse — were mandatory to use.

A good platform acts as a force multiplier for stream-aligned teams, helping them to focus on core domain functionality through attention to the developer experience, ease of use, simplicity of tooling, and richness of documentation. In short, build and run the platform as a product or service itself, with stream-aligned teams as internal customers, using standard agile and DevOps practices

Team-sized software

within the platform itself.

The engineers at cloud communications company Twilio have taken this approach internally for their delivery squads. In a presentation at QCon in 2018, senior director of engineering [Justin Kitagawa](#) described how Twilio's internal platform has evolved to [reduce the engineers' cognitive load](#) by providing a unified self-service, declarative platform to build, deliver, and run thousands of global microservices.

Furthermore, the platform's developer experience is regularly assessed via feedback from internal customers using a [Net Promoter Score](#).

The internal platform at Twilio explicitly follows these key principles:

- **API-first:** Empower dev teams to innovate on platform features via automation.
- **Self-service over gatekeepers:** Help dev teams determine their own workflow.
- **Declarative over imperative:** Prefer "what" over "how."
- **Build with empathy:** Understand the needs and frustrations of people using the platform.

This approach has enabled Twilio to scale to a customer base of over 40,000 organizations worldwide.

By reducing cognitive load, a good platform helps dev teams focus on the differentiating aspects of a problem, increasing personal and team-level flow and allowing the whole team to be more effective.

Lighten the load

Team cognitive load is an important dimension when considering the size and shape of your software system boundaries. By ensuring that team cognitive load isn't too high, you can increase the chances that team members will be able to

Team-sized software

build and operate services effectively because they will properly understand the systems they are building.

We recommend the use of three core team interaction modes to clarify the interactions between teams and ultimately help to reduce cognitive load. When used with independent stream-aligned teams and a thinnest viable platform, these team interaction modes will help your organization detect when cognitive load is too high in different parts of your systems.

[Original:

<https://techbeacon.com/app-dev-testing/forget-monoliths-vs-microservices-cognitive-load-what-matters>]

Assessments

Assessment objectives



Uncover Blockers
to Fast Flow



Accelerate Flow
and Feedback



True Team
Autonomy

Core dimensions



Team Cognitive
Load



Flow &
Feedback



Conway's Law



Streams &
Boundaries



Continuous
Org Evolution

Benefits



Shared
Understanding



Clear
Roadmap



Energized
for Change

Why teams fail with Kubernetes — and what to do about it

Manuel Pais, co-author of Team Topologies

Kubernetes offers a powerful operating model for running cloud-native systems, but adopting it is anything but straightforward.

Yes, Kubernetes helps reduce the operational complexity of microservices, and it provides useful abstractions for deploying and running containers. But moving to Kubernetes is akin to adopting an elephant as a pet.

There are major implications to how teams must interact when you're using Kubernetes—especially as you scale. Fail to address those issues, and you'll put your entire endeavor at risk. Here's what you need to keep in mind.

It's all about team interactions

Kubernetes adoption is not just about the operations/infrastructure team migrating the infrastructure setup to Kubernetes clusters while product teams deploy and run services in Kubernetes pods. Those are the core inputs to the engine, but you'll face many other tasks and responsibilities when running Kubernetes — even if you're using a managed service.

Fail to address the questions "Who is responsible for x?" and "Who is affected by y?" and you'll put all your efforts at risk. For example, replace "x" above with "deciding on namespaces versus clusters for service and environment isolation" or "upgrading all clusters to a new Kubernetes version," and you start to see why you need to clarify the boundaries of responsibility and their impacts.

The way teams interact, and the behaviors promoted by your culture, are more accurate predictors of a successful Kubernetes adoption than are technical

Team-first tools and skills

expertise and infrastructure costs and metrics — that is, if you measure success as enabling faster and sustainable delivery of customer-focused value (via features, better user experience, more resilience).

Having clarity of purpose, and understanding the responsibilities and behaviors around the teams operating Kubernetes (operations/infrastructure/platform) as well as the teams using Kubernetes (product/feature/stream) are all key to success.

Abstractions, cognitive load, and DevEx

Using Kubernetes might be a sound decision from an engineering standpoint, but the developer experience (DevEx) is often subpar, and the abstractions are at a lower level than any individual developer would need because Kubernetes was designed as a generic platform to meet every possible use case.

Extraneous cognitive load is the amount of human working memory used to understand and perform a task that is not directly related to the business outcome you're trying to achieve.

Poor DevEx and complicated abstractions and interfaces mean that the cognitive load for the average developer who lacks deep Kubernetes expertise will increase steeply when you adopt Kubernetes. That is, unless you explicitly consider and manage that potential overload.

You need a digital platform on top of Kubernetes

Kelsey Hightower, staff developer advocate for the Google Cloud Platform, said Kubernetes should be an implementation detail of an organization's change management system.

In other words, you need to focus on clarifying the interfaces and enhancing the usage experience of the internal services that your product teams rely upon to quickly and safely build, deploy, and run the services they are responsible for. These systems can range from CI/CD pipelines to monitoring and metrics collection.

Team-first tools and skills

You need to abstract away the details that are extraneous to your organization's build and run processes. You need to increase the reliability, predictability, and security of that small set of critical internal services, and provide adequate support (including on-call support) and communication channels for fast feedback.

All of this is engraved in [Evan Bottcher](#)'s simple [definition of a digital platform](#): "A digital platform is a foundation of self-service APIs, tools, services, knowledge and support, which are arranged as a compelling internal product."

"Kubernetes is not a digital platform, although it might well be the foundation for one (regardless if under a managed service like Amazon Elastic Kubernetes Service or not). Failing to understand and address this difference is the prime reason for poor adoption in many organizations."

Not defining this internal platform leads to inconsistencies in the use of external services. It also leads to unreasonable demands on product teams that are already being pulled in many directions while, ironically, being pressured to deliver more features faster, since they now have Kubernetes.

But [Kubernetes is no silver bullet](#). Its complexity presents a steep learning curve for newcomers. If your engineers are being asked to rely on Kubernetes documentation to learn to solve their problems, no matter how good that documentation is, you do not have a digital platform.

You likely have a gap in operational capabilities and a maturity issue that needs to be addressed before you can reap the force-multiplier benefits that Kubernetes can bring about.

The size of a digital platform varies with mileage and scale. For a startup, a simple wiki page specifying which cloud services to use with some sensible defaults, tricks, and caveats might be enough. You might rely on your more experienced engineers to provide documentation and support on an as-needed basis. In our book, [Team Topologies: Organizing Business and Technology Teams for Fast Flow](#), [Matthew Skelton](#) and I call this a "thinnest viable platform."

Team-first tools and skills

As your startup grows, so will your platform, as the product teams begin to need more internal services. Eventually, a platform group might include multiple platform teams, each aligned to a small set of platform services. These teams need strong product management to create a compelling internal product that makes life easier for the other engineering teams (the platform clients).

How Airbnb enabled 1,000+ engineers with Kubernetes

Airbnb is a good example of a digital platform on top of Kubernetes that evolved based on the needs of its engineering teams. Melanie Cebula, infrastructure engineer at Airbnb, spoke at QCon London about the way her team wraps Kubernetes into easy-to-consume internal services for its development teams.

As she explained, instead of creating a set of dreaded YAML files (deployment, ConfigMap, service) per environment (dev, canary, production), development teams need only provide their project-specific, service-focused inputs and then run the internal service kube-gen (alias k gen).

This simple command takes care of generating all the required YAML files, ensuring their correctness (not just syntax-wise but also semantically in terms of expected values), and finally applying them in the corresponding Kubernetes cluster(s).

The infrastructure team at Airbnb is saving hundreds, if not thousands, of hours for 1,000+ engineers who can now use a much simpler abstraction that has been adapted to their needs, with a user experience that's familiar to them.

Other internal services provided by the infrastructure team include k deploy, to create new namespaces; k diagnose, to collect information from multiple sources on malfunctioning pods and services; and templates for new services and deployment pipelines.

Effectively, they are providing a digital platform for their engineers that embeds their evolving understanding of what engineering teams need to perform better, as well as good practices and tooling around security, logging, debugging, and so on. Crucially, they are doing this without asking for more of the engineering

Team-first tools and skills

teams' cognitive load. Instead, engineers are free to focus on business outcomes with clearly defined, simple service boundaries.

generating k8s configs

@MELANIECEBULA

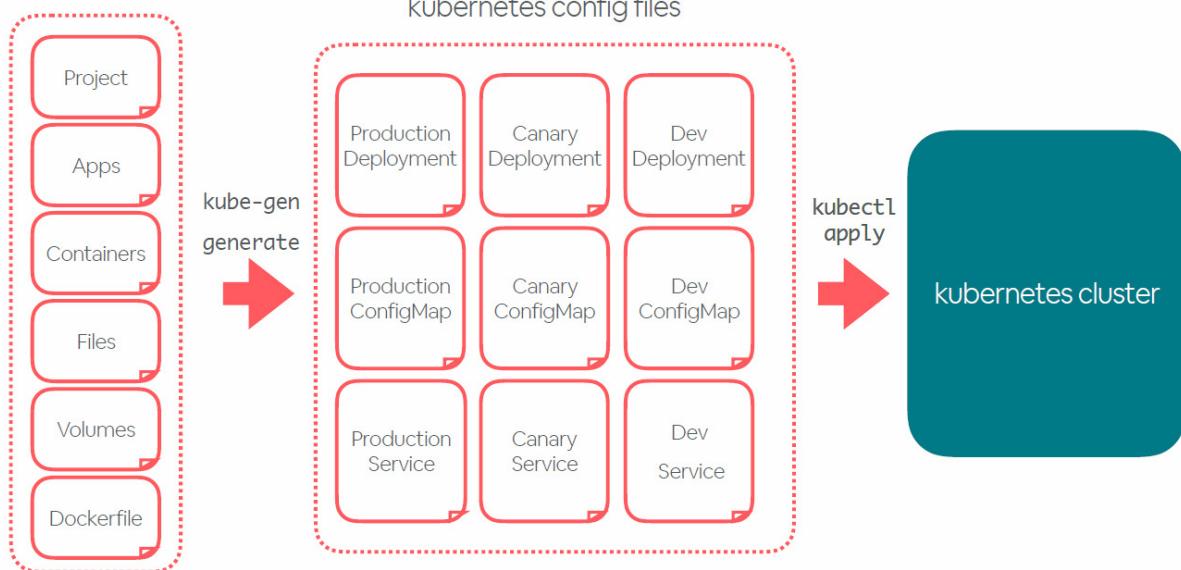


Figure 1. The kube-gen wrapper generates the needed configuration files per environment at Airbnb.
Source: *Melanie Cebula, Airbnb*.

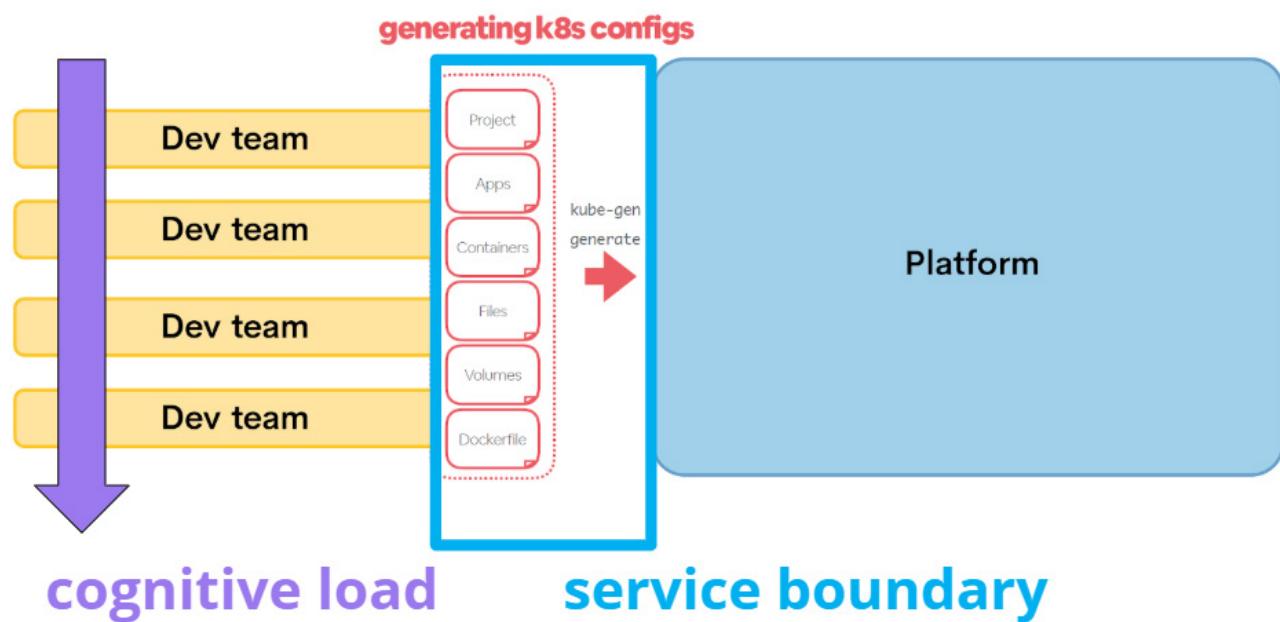


Figure 2. The infrastructure platform at Airbnb establishes clear boundaries and reduces the cognitive load on development teams. Source: *Team Topologies: Organizing Business and Technology Teams for Fast Flow*.

Team-first tools and skills

Clear team interactions are key to sustained success

The success of an internal platform is influenced by the behaviors and interaction modes of the responsible teams to a much larger extent than by its technical achievements. If the platform team does not see its mission as to reduce the extraneous cognitive load of engineering teams by means of a compelling internal product, then it might dwell in the technical complexity of a service and forget to check if it serves the needs of the team that requested it.

If the platform team does not collaborate closely with the product teams during initial stages of a new service or evolution to have fast feedback, then the developer experience will suffer, and usage will drop because the platform will stop being a compelling product.

If the platform team does not provide timely (on-call and office hours) support for its internal services with clear response times, service status pages, and communication channels, then the platform will not be seen as reliable and engineering teams might resort to other options.

On the other hand, product teams need to carefully reconsider whether they really need to go off the “paved road” provided by the platform for any specific service or tooling requirements. If they go off on their own without talking to the platform team and without a clear use case for adopting some new technology, then they will break the trust boundaries with the platform team and end up having too much unnecessary cognitive load.

Product teams need to be open and frank about their needs while understanding whatever limitations the platform teams might be working under. Blameless interactions are key.

A general pattern of interaction between product and platform team is to have close collaboration during the initial discovery stages for a new platform service (or evolution) required by a product team. Over a period of time, this intentional collaboration effort will diminish as the needs, boundaries, and interfaces for this service becomes clearer, until eventually it can be consumed by all product teams as a service.

Team-first tools and skills



Figure 3. The evolution pattern of team interactions for a new platform service (or evolution), from initial discovery with high collaboration to "X as a service" with no need to collaborate any more. Source: *Team Topologies: Organizing Business and Technology Teams for Fast Flow*.

In the end, it's all about teams having a clear purpose, responsibilities and ways of interacting in order to set the right expectations and behaviors.

How to get started

Take these three simple steps to nudge your organization's Kubernetes adoption with a human- and team-centric approach.

1. Assess cognitive load. Ask your teams if they truly understand how to build, deploy, and run the applications they are responsible for in Kubernetes.
2. Visualize the platform. Kubernetes is not your internal platform. Document how your organization is currently using it, along with your recommended practices, sensible defaults, and other useful information in a wiki page. Then start adding the missing pieces for a true digital platform.
3. Clarify team interactions. Set the right expectations between teams in terms of who is responsible for what, who is affected, and what types of behaviors to adopt in which circumstances.

Follow the initial steps above and you'll start to understand the gap between your current Kubernetes implementation and having an internal digital platform (and teams) that accelerates software delivery through reduced cognitive load, a first-class developer experience, and a compelling platform that is resilient and fit for purpose.

You'll also gain insights into how your teams interact today, and the anti-patterns and misaligned expectations that are creating friction between teams and

Team-first tools and skills

individuals. You'll be moving toward a healthier, more organic work environment that acknowledges the complex socio-technical nature of software systems today.

[Original:

<https://techbeacon.com/enterprise-it/why-teams-fail-kubernetes-what-do-about-it>]

How to find the right DevOps tools for your team

Matthew Skelton, Head of Consulting, Team Topologies

When you adopt a DevOps approach to building and operating software systems, you must rely on modern tools for almost every aspect of build, release, and operations activities. But before you get into the weeds of comparing one tool against another, you need to [think more broadly](#) about what you need.

And there are [many types of DevOps tools](#) to consider. With DevOps, many previously manual or semi-manual activities are fully automated, including version control (for application code, infrastructure code, and configuration), continuous integration (for application code and infrastructure code), artifact management (packages, container images, container applications), continuous delivery deployment pipelines, test automation (unit tests, component tests, integration tests, deployment tests, performance tests, security tests, etc.), environment automation and configuration, release management, log aggregation and search, metrics, monitoring, team communications (chat, video calling, screen sharing), and reporting.

You'll find plenty of excellent tools in all of these categories, but it's easy to get hung up on the pros and cons of using one tool versus another. And while sometimes that's the right debate to have, [confusion around tools](#) may be a symptom of deeper problems with respect to the way in which your team uses those tools, or how you introduce those tools to the team.

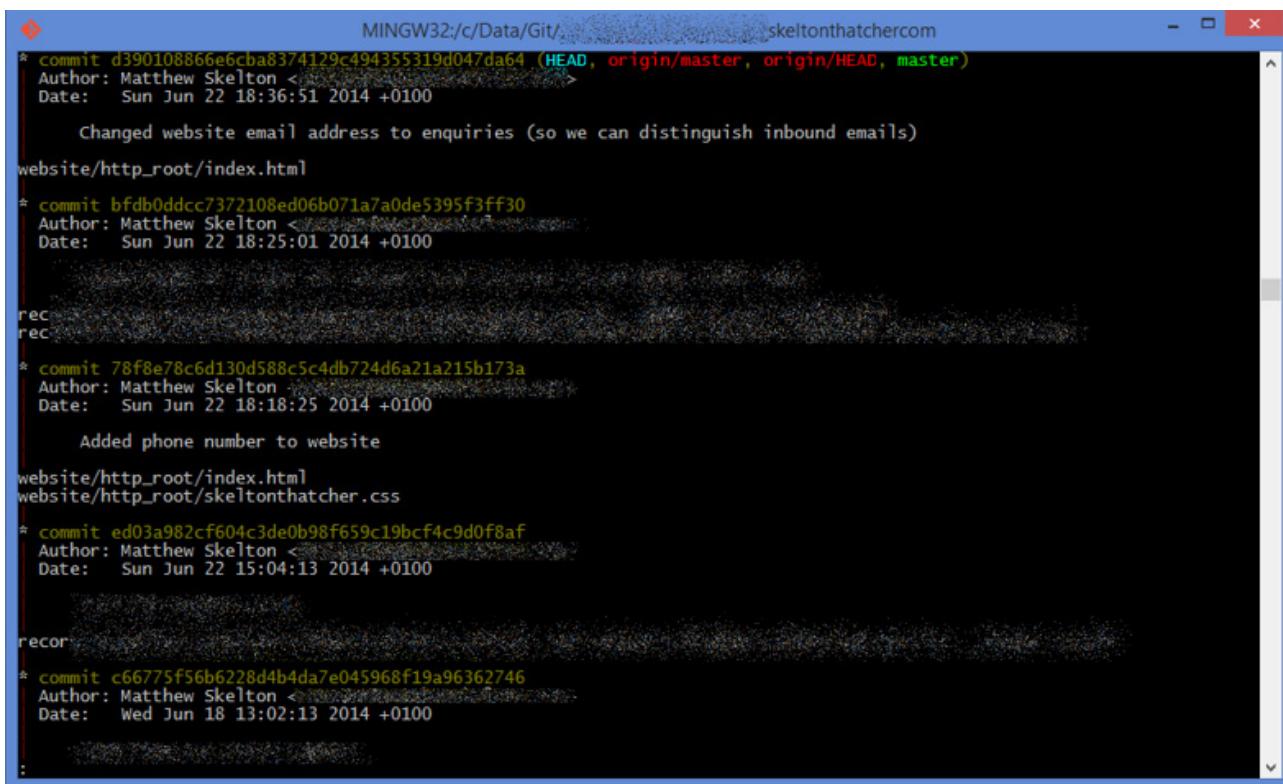
I have been using the guidelines below with clients since 2014, and we've managed to solve tooling-related problems that would otherwise have descended into an unhelpful product X-versus-Y shooting match. To become a high-performing organization, you must take into account the social dynamics of your organization and the trajectory of the rapidly evolving public cloud vendors.

Team-first tools and skills

Choose tools that facilitate collaboration

Having highly effective collaboration between teams is critical for DevOps. Some people think they need to buy a dedicated collaboration tool for this purpose, but there are many different tools you can use to enhance collaboration.

Consider one of the cornerstones of a DevOps approach: version control. Let's say you're trying to encourage more people in the organization to use version control, including for database scripts, configuration files, and so on. If you insist that everyone use only a command-line tool for version control, you'll miss out on collaboration opportunities:



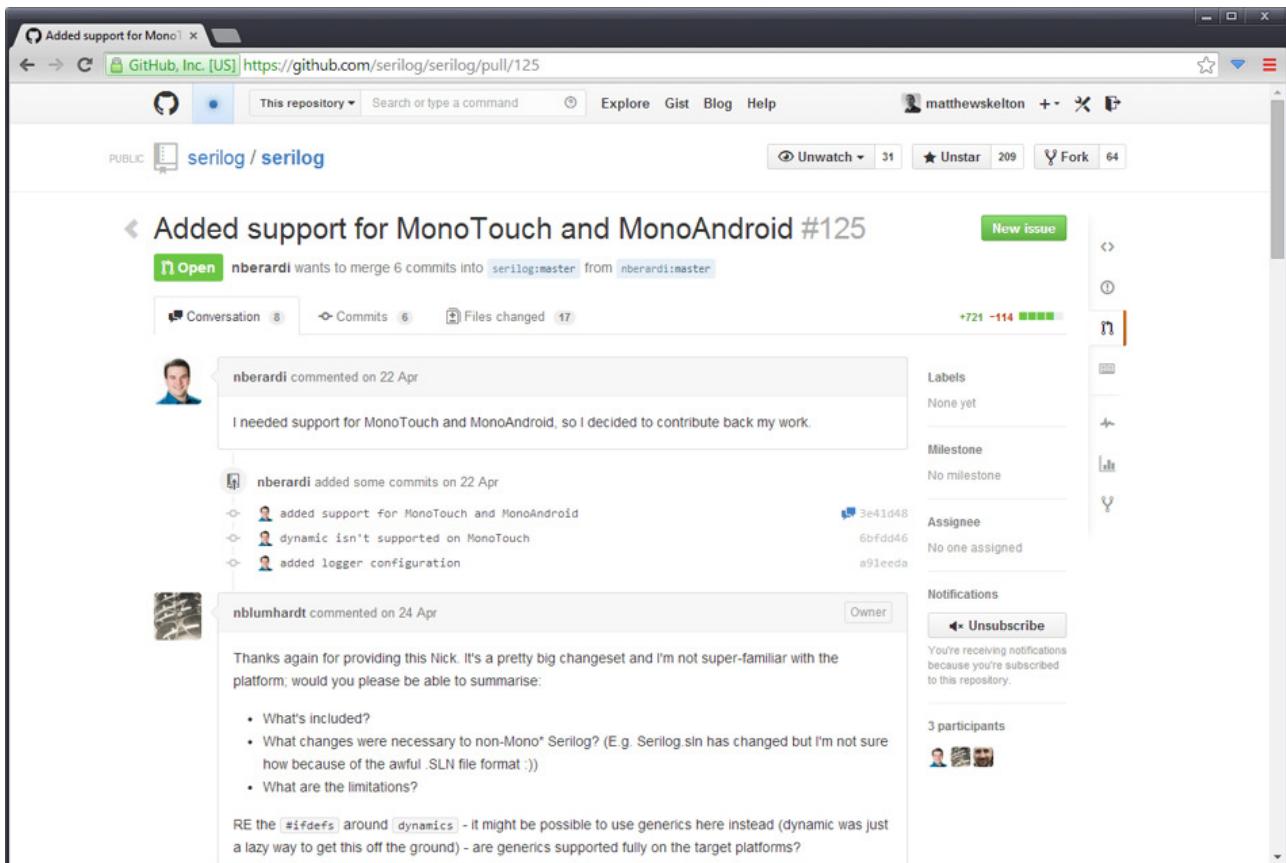
The screenshot shows a terminal window titled 'MINGW32:/c/Data/Git/'. It displays a list of Git commit history. The commits are as follows:

- * commit d390108866e6cba8374129c494355319d047da64 (HEAD, origin/master, origin/HEAD, master)
Author: Matthew Skelton <...>
Date: Sun Jun 22 18:36:51 2014 +0100
Changed website email address to enquiries (so we can distinguish inbound emails)
website/http_root/index.html
- * commit bfdb0ddcc7372108ed06b071a7a0de5395f3ff30
Author: Matthew Skelton <...>
Date: Sun Jun 22 18:25:01 2014 +0100
- rec
rec
- * commit 78f8e78c6d130d588c5c4db724d6a21a215b173a
Author: Matthew Skelton <...>
Date: Sun Jun 22 18:18:25 2014 +0100
Added phone number to website
website/http_root/index.html
website/http_root/skeltonthatcher.css
- * commit ed03a982cf604c3de0b98f659c19bcf4c9d0f8af
Author: Matthew Skelton <...>
Date: Sun Jun 22 15:04:13 2014 +0100
- rec
rec
- * commit c66775f56b6228d4b4da7e045968f19a96362746
Author: Matthew Skelton <...>
Date: Wed Jun 18 13:02:13 2014 +0100
- :

Command-line tools can be a barrier to collaboration for some people.

The command-line view of version control is certainly part of a DevOps tool set, but it is unfamiliar to many people — especially non-developers — and has no obvious collaboration potential. But if you use a richly featured version-control platform such as Github, Bitbucket, or Gitlab, you can take advantage of discussion threads around file changes to get people talking about why a file changed. This helps you collaborate with people who have different skills, and encourages more people to learn how to use version control:

Team-first tools and skills



Browser-based tools can help to encourage collaboration for people who are less technology-savvy.

Using a browser-based version-control platform opens up version control to a wider audience than just software developers, which in turn helps you to emphasize the importance of version control as a key DevOps practice. By choosing a version-control tool with discussion capabilities and making it available to a wide audience, you can enable rich communication between teams and groups within your organization.

The same approach works for many other tools, too. I once consulted with an organization that had a tool for log aggregation and search. The IT operations people found it valuable, but the developers did not have permission to search the logs from the production systems. Access was denied because, IT claimed, the data was of a sensitive nature. But the managers wanted to improve the way in which the IT Ops and Dev teams collaborated. So they opened up access to the log-search tool for developers and — surprise — developers and operations people collaborated more. The tool hadn't changed, but changing access permissions enhanced collaboration.

Team-first tools and skills

Key points:

- Value collaboration as a key selection aspect of tools.
- Look behind the tool's main purpose to find collaboration opportunities.
- Ask, "How does our use of this tool help or hinder people in collaboration?"

Favor tools with APIs

Modern software development needs delivery tools that are highly automatable, yet customizable. That means you need a fully featured API for each tool—preferably one that's HTTP-based. When you compose capabilities by gluing together API-rich tools, you enable easy wiring for alerts and other events. Avoid tools that try to do everything from within their own frames of reference; favor those that do one or more jobs well and integrate easily with other tools.

Given the speed of change in the software sector, it's particularly important to choose tools that meet these criteria. If you do, then when a new tool comes along you'll be able to replace your old tool with minimal disruption. Being stuck with a big, lumbering tool set that's only half-good at most things has been a source of significant pain for organizations trying to adopt DevOps. Keep your tooling nimble and composable to give your team the flexibility to adapt new approaches easily.

But beware of "spaghetti" tooling that's chained together with undocumented scripts. Treat your software delivery and operations tools like a proper production system. At the rapid pace enabled by DevOps, it's essential to be able to keep the tools you use for software delivery and operations running and working 24x7. Many companies make the mistake of adopting new tools without the operational support and care needed to make those tools work well. So when adopting new tooling, consider starting with SaaS-hosted offerings and running internal prototypes/demo versions before building an internal capability.

Key points:

- Choose tools that expose APIs.
- Aim for composition of new capabilities from multiple API-driven tools.
- Build and deployment are first-class concerns.

Favor tools that can store configuration in version control

One core tenet of DevOps is that you should store all configuration settings in version control. That includes the configuration not just for your custom software applications, but also for tools you use in software delivery and IT operations.

To be effective in a DevOps context, each tool must expose its configuration in such a way that you can store the configuration as text files that you can then expose to version control. Then you can track the history of configuration changes and test changes beforehand.

Why would you want to do that? If you cannot track and test configuration changes to your delivery and operations tooling, you risk breaking the machinery that makes DevOps work.

Key points:

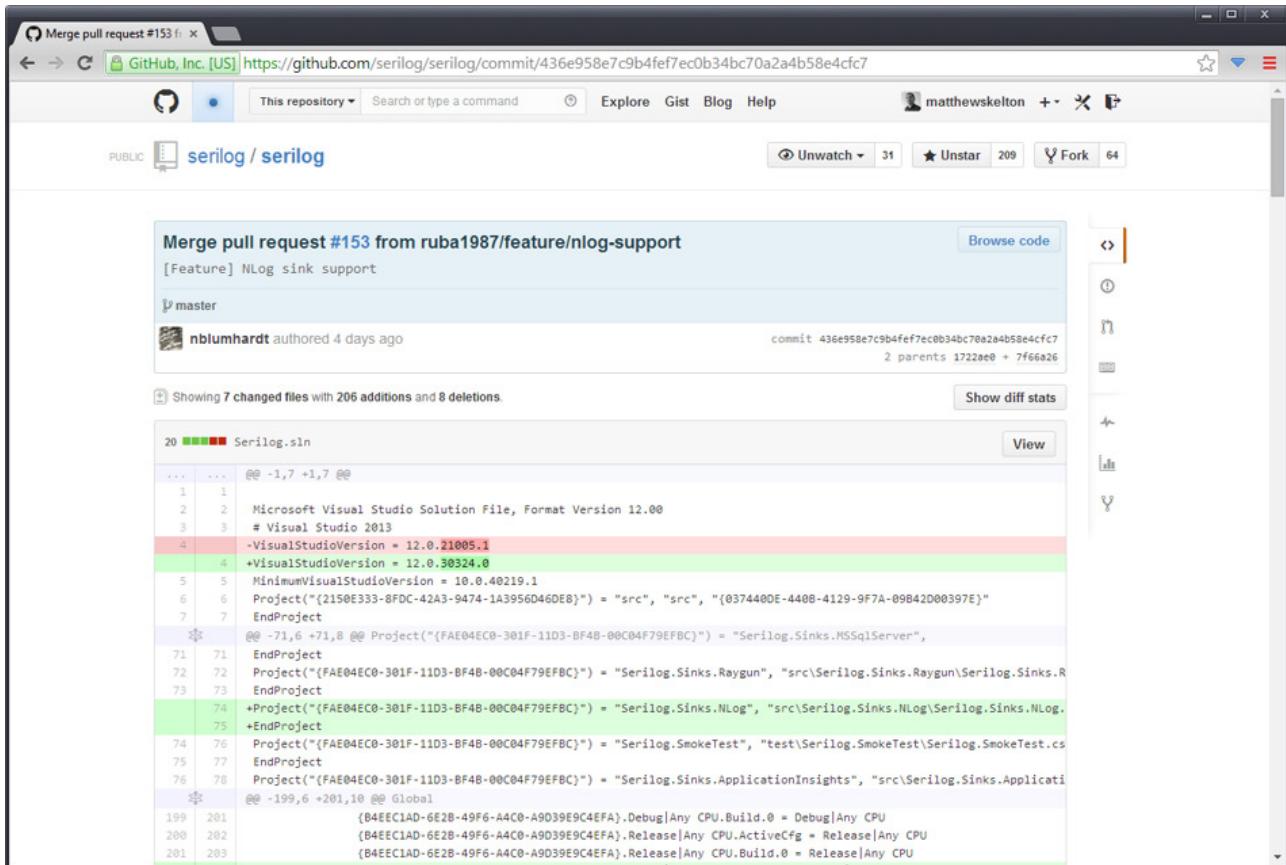
- Choose tools that expose configuration to version control.
- Point-and-click is no longer acceptable for configuration of tools.

Use your tools in a way that encourages learning

Some of the tools useful for DevOps are quite involved and complicated, especially for people new to them; don't expect everyone to understand or adopt difficult new tools immediately. In fact, if you introduce a tool that is too tricky, some people may become hostile, especially if you don't provide training or coaching. That sometimes happens when organizations select best-of-breed tools without considering how easy they are to use.

Assess the skills in your organization and devise a tools roadmap for moving teams to improved ways of working. Select tools that offer more than one way to use them (GUI, API, command-line) so people can learn at their own pace. And avoid leaving people behind on the climb to more advanced approaches by holding regular team show-and-tell sessions to demonstrate tools and techniques.

Team-first tools and skills



Command-line tools can be daunting for some people and may hinder collaboration unless you provide training. Tools with a more friendly UI can help to bring people on board to new ways of working, giving them the confidence to adopt command-line tools later.

For example, you might start with the browser-based interface, such as the one below, for people new to version control, giving them time to adjust to this approach before training them on the command-line tools for version control.

DevOps is a journey from mostly manual to fully automated, and not everyone starts from the same place. Give people time and space to become familiar with new tools and approaches. They might start with a simpler tool, then adopt a more powerful one later.

Key points:

- Bring people with you on your DevOps journey.
- Prefer achievable gains now over possible future state.
- Avoid a fear of too-scary tools by stepwise evolution.

Team-first tools and skills

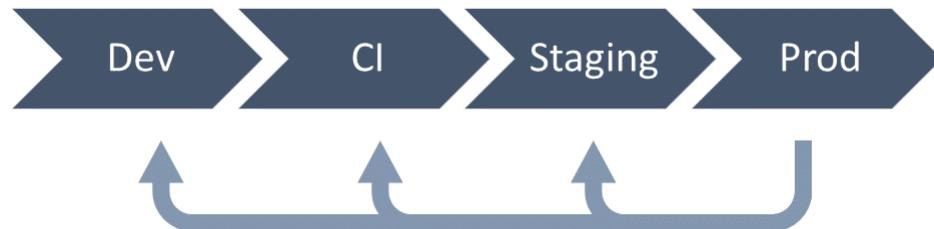
Avoid special production-only tools

The speed and frequency of change that DevOps gives you the means you need to emphasize the feedback loops within your delivery and operations processes. In particular, it is important that all technology people in your organization learn as much as possible about how the production environment works so they can build better-working, more resilient software. You also need to test changes to all parts of the software system before deploying new versions to production.



Production-only tools prevent teams from learning because production is treated as a special case.

For an effective DevOps approach, choose tools that work easily in nonproduction environments (development, continuous integration, staging, etc.). The tool should be cheap enough to buy or install so that you can install it in all environments, including developer laptops and the automated build-and-test system. A tool that is so expensive that you can only afford a license for production is not a good tool for DevOps. Such “singleton” tools tend to accrue an aura of magic, leading people to think that production is special. People become disengaged, and that’s a bad outcome. Good tools for DevOps are also easy to spin up in different environments using automated scripts. A tool that needs manual installation is not a good choice for DevOps.



Running the same tools in production as in all other environments enables rapid learning and increases engagement within teams.

Team-first tools and skills

In some sense, this “run it anywhere” approach to tools for DevOps makes production less special, and rightly so. Many of the problems with older, fragile IT systems are the result of production being treated in a special way, preventing developers and testers from learning how production works. With a DevOps approach, your aim is to choose tools that are easy to install and can spin up in multiple environments, even if the feature set is less impressive than that of a tool that is more advanced but difficult to configure. Aim to optimize globally across teams that need to collaborate, not just locally for production.

Key points: Production-only tools...

- Break the learning feedback loop.
- Make CI/CD more difficult.
- Underestimate the value of collaboration and learning.

Choose tools that enhance inter-team communications

One of the most common problems I see in organizations struggling to build and run modern software systems in a DevOps way is a mismatch between the responsibility boundaries for teams or departments and those for tools. The organization either has multiple tools when a single tool would suffice (in order to provide a common, shared view), or it has a single tool that’s causing problems because teams need separate ones.

In recent years, Conway’s Law has been observed and measured in many studies. The communication paths in our organization drive the resulting system architecture:

“Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations.”

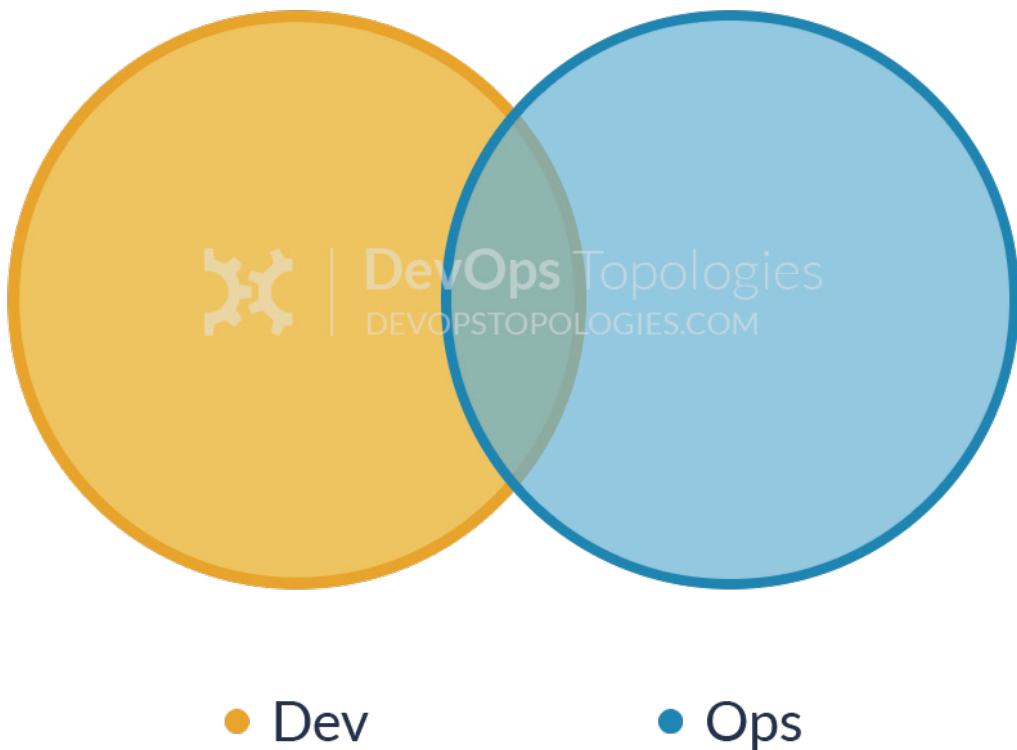
— Mel Conway

You therefore need to be mindful of the effect of shared tools on the way in

Team-first tools and skills

which teams interact. If you want your teams to collaborate, then shared tools make sense. But if you need a clear responsibility boundary between teams, separate tools may be best. Use my [DevOps team topologies patterns](#) to understand which DevOps model is right for your organization, and then choose the tools that fit that model.

If you need the development team to work closely with operations (the Type 1 model), then having separate ticketing or incident management tools for Dev and Ops will result in poor inter-team communication. To help these teams collaborate and communicate, choose a tool that can meet the needs of both groups. But be sure that you understand the user experience needs of each group, since a tool that infuriates your engineers is a sure way to stop a DevOps effort dead in its tracks.

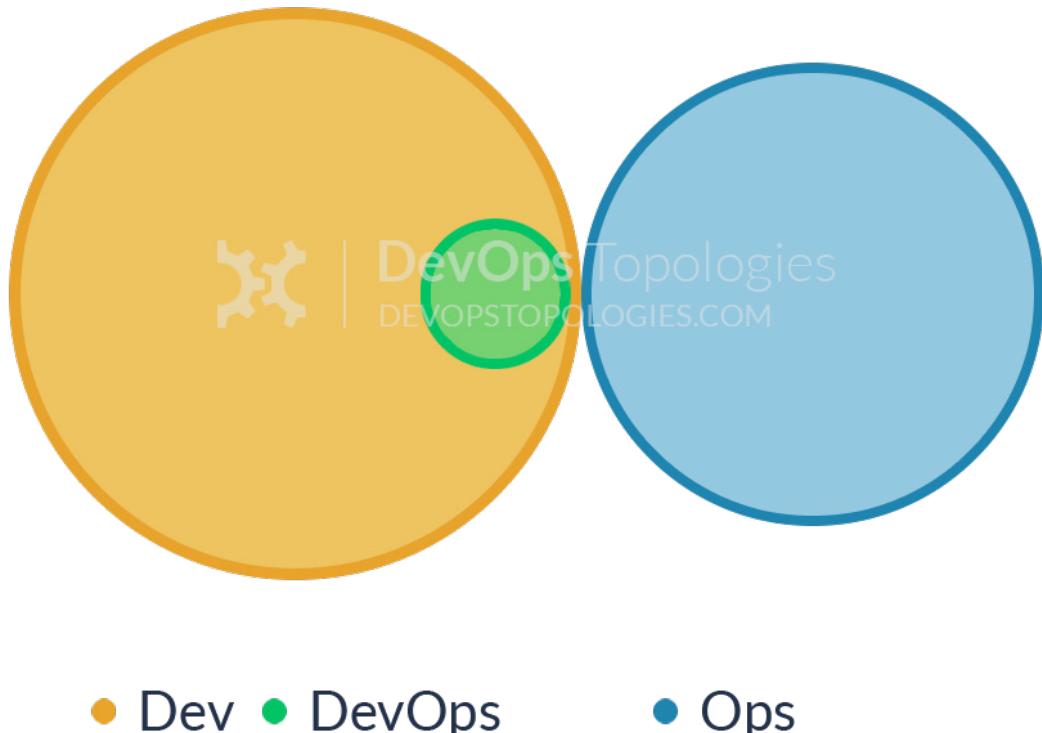


In a Type 1 platform model, smooth collaboration implies some shared tooling between Dev and Ops.
Image from [DevOpsTopologies.com](#) and licensed under [CC BY-SA license](#).

If, like many enterprises, yours is moving to a Type 3 (platform) model, then the platform team is not responsible for the live service of the applications; that's the responsibility of the product development teams. When responsibility boundaries don't overlap, you won't get much value from insisting on the same incident-tracking tool or even the same monitoring tool for the platform and development

Team-first tools and skills

teams. This becomes even clearer when IT operations has outsourced to a cloud service provider, since in that case there's no question about forcing the same tools on two different teams.



A Type 3, IT Ops as infrastructure-as-a-service (platform) implies little need for shared tooling between Dev and Ops. Image from DevOpsTopologies.com and licensed under CC BY-SA license.

In summary, don't select a single tool for the whole organization without considering team inter-relationships first.

Key points:

- See the whole organization as a system you're building.
- Have separate tools for separate teams.
- Deploy shared tools for collaborative teams.

Optimize for learning, collaboration, automation, and team dynamics

When choosing tools for DevOps, it's important to avoid product X-versus-Y tooling shootouts that simply compare lists of features side by side. Sometimes

Team-first tools and skills

that's needed, but only after you understand the broader implications of having one (or both) of those tools in place in your organization. Using tools in the wrong way — especially trying to make everyone use the same tool — can be counterproductive for DevOps.

Try to assess and understand where your team communication boundaries should be, using the DevOps team topologies patterns and Conway's Law, to avoid a one-size-fits-all approach to tools. Sometimes, using multiple, similar tools is the right approach, but that depends on your team boundaries.

Ensure that the tools you choose do not present a learning barrier to people who are new to DevOps approaches; expect to replace tools regularly as people develop their skills and establish new collaboration patterns.

Tools for DevOps need programmable APIs. Don't buy or use tools that need a human operator to click buttons on a browser application. With DevOps, you need to compose functionality from multiple, cooperating tools using APIs and "glue" scripts.

Finally, don't optimize for your production environment. A tool that exists only in the live production environment is a tool that you can't test upstream, and that's a dangerous approach in a fast-paced DevOps world.

[Original: <https://techbeacon.com/devops/how-find-right-devops-tools-your-team>]

Training

Training courses

Online interactive courses with up to 15 attendees

Each half-day session can be taken independently. Four sessions together (TT04 to TT07) form the *Essentials* course:

- TT04 - Stream-aligned Teams
- TT05 - Reducing Cognitive Load
- TT06 - Evolving Responsive Organizations
- TT07 - Architecture for Fast Flow
- TT08 - Modern Platforms

Fully remote training using videoconference and group work tools.

Training benefits



Shared
Understanding



Evolution, Not
Revolution



Working
Patterns

Why you should hire DevOps enablers, not experts

Manuel Pais, co-author of Team Topologies

We are regularly asked if we know any DevOps or site reliability engineering (SRE) experts available for hire. Our answer is, invariably, "Not really." It's a tough market out there.

DevOps and SRE (for large-scale software, at least) are critical approaches for success in modern software delivery and operations, as widely demonstrated every year in the [State of DevOps report](#) or the array of [presentations](#) at the [DevOps Enterprise Summit](#).

But if you think you can achieve DevOps by [hiring "DevOps experts,"](#) you are missing some contextual awareness. What exactly are you trying to improve in the first place? If your software delivery is slow because of work you're handing off among multiple teams with diverse schedules and priorities, will a new hire really help?

We're not suggesting that you not hire people with diverse skills and backgrounds — that can be quite valuable to bring in new perspectives and approaches. But [conventional hiring based on expertise alone is ineffective](#) and prevents organizations from developing the "learning muscles" that can help teams traverse the latest trends (DevOps, SRE, etc.) to their benefit at the right time, and in the right context.

Hiring experts for every need is like engaging in palliative care for organizational health. Preventive care would be to incorporate the necessary team structures and interactions — as well as a focus on people growth and sufficient slack — to effectively take in process, technology, and business changes.

Learning organizations smoothly morph as they adapt to new challenges, and they [unlearn existing ways of working](#) when they become limitations rather than enablers.

Curated team interactions

Hire with alignment of purpose in mind

In his book *Drive: The Surprising Truth About What Motivates Us*, Daniel Pink explains the three pillars of intrinsic motivation for knowledge workers:

- Autonomy
- Mastery
- Purpose

When you acknowledge that a team is the fundamental, indivisible unit of delivery and operations for a product or service, then it follows that a high-performing team needs to fulfill those three intrinsic motivators for all of its members.

People generally understand how to apply autonomy and mastery to a team, especially within the context of agile, but the purpose of a team is less clear.

In our book, *Team Topologies*, we identify and characterize the purpose of four fundamental topologies. Besides clarifying what each team is trying to achieve, you also want to ensure that new team members' individual purposes are aligned with those of the team.

Four fundamental team types and purposes

There are four different team types and purposes:

1. **Stream-aligned.** These are cross-functional teams whose purpose is to deliver a product or service to external customers via end-to-end ownership of the lifecycle, from ideation to operations.
2. **Platform.** This type of team's purpose is to provide internal services to reduce the (cognitive) effort that would be required from stream-aligned teams to develop these underlying services. In other words, such a team delivers services to internal customers.
3. **Enabling.** These are teams of specialists in a given technical (or product) domain whose purpose is to help other teams grow new capabilities in that

Curated team interactions

domain, reducing their learning curve when adopting new practices and technologies. They can be seen as internal consultants, and do not develop products or services.

4. **Complicated subsystem.** These are teams whose purpose is to build and maintain a highly complicated part of a system that depends heavily on specialist knowledge (think PhD-level specialists or niche technology), requiring full-time effort.

The last three team types work toward reducing the cognitive load of the stream-aligned teams, so the latter can focus on fully understanding and owning the products or services for which they are responsible without diversions such as, for example, setting up infrastructure and monitoring from scratch.

Failure to consider how a candidate fits in with your team's purpose can lead to a dysfunctional team, disengaged team members, and high turnover rates — especially for individuals hired based on their expertise in "hot" trends.

Alignment is key

A modern hiring process needs to consider alignment between individual goals and interests and the purpose of the team new hires are expected to join. For example:

- A candidate who strives to be multi-skilled and always learning should be a good fit in a stream-aligned team, but only if that person enjoys frequent feedback and contact with (sometimes upset) customers.
- A candidate who enjoys automating processes should fit in well in a platform team, but only if that person is genuinely interested in understanding the needs of other teams (as well as the organization), and in developing services based on feedback and *fitness-for-purpose*.
- A candidate who thrives on a particular technical or product domain or practice and wants to continuously stay ahead of the curve might fit in well in an enabling team, but only if naturally inclined to communicate, pair, and share knowledge in a non-judgmental way.

Curated team interactions

Hire with cognitive load in mind

There are three [types of cognitive load](#) that teams may face:

- **Intrinsic cognitive load**, which relates to aspects of the task that are fundamental to the problem. Example: How are classes defined in Java?
- **Extraneous cognitive load**, which relates to the environment in which the team is performing the task. Example: How do I deploy this app, again?
- **Germane cognitive load**, which relates to aspects of the task that need special attention for learning or high performance. Example: How do bank transfers work?

Broadly speaking, you want to minimize intrinsic cognitive load and eliminate extraneous cognitive load (boring or superfluous tasks or commands that add little value). This will free working memory for germane cognitive load (which is where value-added thinking lies).

Learning approaches to consider

Keyword-driven hiring focuses on finding experts with low intrinsic cognitive load; they have internalized tasks in their domain of expertise, like driving a car without thinking about all the actions involved.

But that only helps in the short term. An expert who cares more about the delivery mechanisms and technology, rather than how the actual products or services work and fit the needs of its users, will be increasing extraneous cognitive load and reducing space for business-focused germane cognitive load.

Also, there are plenty of approaches you can take to reduce intrinsic cognitive load by spreading knowledge within teams and organizations. These include pair and mob programming, [mentoring](#), [immersive dojos](#), [communities of practice](#), [brown bag lunches](#), more classical training, [conferences](#), and books.

Pick what's easiest to start with and evolve over time. What is often missing, however, is the vision to invest the necessary time and patience to start

Curated team interactions

harvesting the results of upskilled, empowered employees.

This means your organization should continuously look for ways to reduce extraneous cognitive load on its teams, rather than falling back to hiring experts as a palliative solution for an immediate need.

Hire with learning in mind

If hiring experts is the principal way you acquire expertise and skills (such as agile, DevOps, or site reliability engineering) in your organization, you will face a difficult challenge competing with many other organizations that are doing the same in a scarce labor market. And even if you successfully address that challenge, it will lead to atrophy of your organization's learning muscles. Learning organizations grow from the inside. They can detect when existing tools, practices, and processes are no longer effective for the challenges at hand and adapt continuously.

Bringing in people with new skills and points of view can help challenge assumptions and make progress, but it will not fundamentally transform a static, slow-changing organization into a fast-paced, adaptive one. Becoming a true learning organization requires not only setting up safe learning spaces and practices, but also adopting an integrated view of who you're hiring and why.

So rethink hiring as part of your larger strategy to become a learning organization. Your strategy should:

- Take into account existing team structures and interactions and how they are expected to evolve
- Empower knowledge-sharing activities, providing logistics and especially slack time
- Make it a priority to hire people who are a good fit for a team's purpose, expected behaviors, and interactions with others, above specific technology expertise

Also consider what options you have to reduce the current cognitive load on your

Curated team interactions

teams before hiring.

The [Team Topologies](#) approach provides a thousand-foot view of your organizational landscape, helping you see the forest (missing capabilities and blockers to learning) for the trees (specific skills and trends in need today).

[Original: <https://techbeacon.com/devops/why-you-should-hire-devops-enablers-not-experts>]

Curated team interactions

Are poor team interactions killing your DevOps transformation?

Matthew Skelton and Manuel Pais, co-authors of Team Topologies

The COVID-19 pandemic has ushered in a new remote-first world for IT, with many organizations struggling to catch up with new tooling and [ways of working](#). Some companies have embraced this new reality, ditched their expensive downtown offices, and told staff they can work from home permanently.

And some are discovering for the first time that the physical office was substituting for poorly defined teams and poorly defined areas of focus, threatening [their digital transformation efforts](#).

A successful remote-first approach requires that you explicitly design the communication among teams using physical and online spaces. Using simple tools for dependency tracking, and patterns such as a "team application programming interface" (a concept we developed and explain in our book, [Team Topologies](#)), organizations are finding that well-defined team interactions are key to effective IT delivery in the remote-first world.

Here's what you need to know to go that route.

What does an organization need to thrive in a remote-first world?

Many organizations have found to their dismay that rolling out a new chat or video tool for staff working remotely does not magically make the organization remote-first. Certainly, tools are needed and useful, but for a successful DevOps transformation — whether co-located or remote-first — the organization also needs [good psychological safety for teams](#) and an effective set of ground rules and practices for teams to use for working together.

Curated team interactions

The ground rules and practices define ways of working, set expectations, and provide easy-to-recognize patterns and modes of behavior that make it easy for people to work in well-defined ways. In particular, well-defined team interactions clarify the relationships among different groups in the organization and the purpose of different activities.

This in turn helps to [minimize the cognitive load on teams](#) and provides more head space for focusing on the most important aspects of work within the organization.

So, what techniques can your organization use to improve interactions among teams?

The team API approach can define and communicate responsibilities and team focus

So, what's a team API? An API, or application programming interface, is a technical term for the way one piece of software interacts with another piece of software programmatically. A team's API is a specification for how other teams in the organization can and should interact with that team.

A team API covers a wide range of things, including:

- Artifacts owned by the team (libraries, applications, services, etc.)
- Versioning and testing approach
- Wiki and documentation
- Practices and principles
- Road map and priorities
- Communication preferences (when/how)

By defining these things and making them discoverable by other teams, your team increases its clarity of purpose and helps other groups to understand how that team fits in the wider organization. (There's a free-to-use [template for the team API on GitHub](#).)

Curated team interactions

Track dependencies using simple tools and remove blocking dependencies

In a remote-first environment, it's impossible to simply walk up to the desk of someone on another team to ask about progress, and a constant stream of chat messages asking for status updates becomes a cognitive burden.

Instead of spending time waiting on other teams to finish their work, [focus on tracking and then removing these in-flow dependencies](#). Books such as *Making Work Visible* by Dominica Degrandis, explain useful techniques for visualizing team dependencies, many of which can easily be adapted to work in a fully remote context.

We recently published a [template for tracking team dependencies](#) on GitHub. Based on work from Spotify, the tracker template helps teams to frame conversations around improving flow, avoiding blocking waits, and ultimately moving to a more autonomous delivery model.

Consciously design inter-team communications using team interaction modes

There are many different chat tools available for remote-first working, and most organizations are using a chat tool (or several) these days. However, simply providing all-staff access to a chat tool is only the first step in making remote-first a success.

Too many organizations allow a kind of free rein within the chat tool, with little or no consistency about channel names, display names, the meaning of emojis, or even etiquette. This can rapidly lead to the chat tool becoming both essential to watch (in case you miss a vital message) and incredibly confusing and difficult to use.

For effective remote work, some chat tool conventions are needed. The virtual space inside the chat tool needs to be predictable and discoverable. Arbitrary channel names such as #homepage_discussion, #increase-conversions, and #ninjas make it difficult to know where to go to discuss a topic. If this is combined with multiple private channels, finding the right people to speak to is a

Curated team interactions

game of cat-and-mouse.

Instead, define a set of conventions that improve predictability and discoverability. For example, include the team name and type of team in the channel name for the team's main outward-facing chat channel. For example:

- `#streamteam-green` — the public channel for the stream-aligned team "Green"
- `#streamteam-blue` — the public channel for the stream-aligned team "Blue"
- `#platformteam-data` — the public channel for the platform team "Data"
- `#platformteam-infra` — the public channel for the platform team "Infra"
- `#enablingteam-k8s` — the public channel for the enabling team "k8s"

The team interaction modes from *Team Topologies* can help further increase the clarity of purpose for teams working together, including for these scenarios: collaboration (two teams working together for a defined discovery period to achieve a specific goal); x-as-a-service (one team provides something as a service, another team consumes); and facilitating (one team helps another to detect capability gaps or increase skills and awareness).

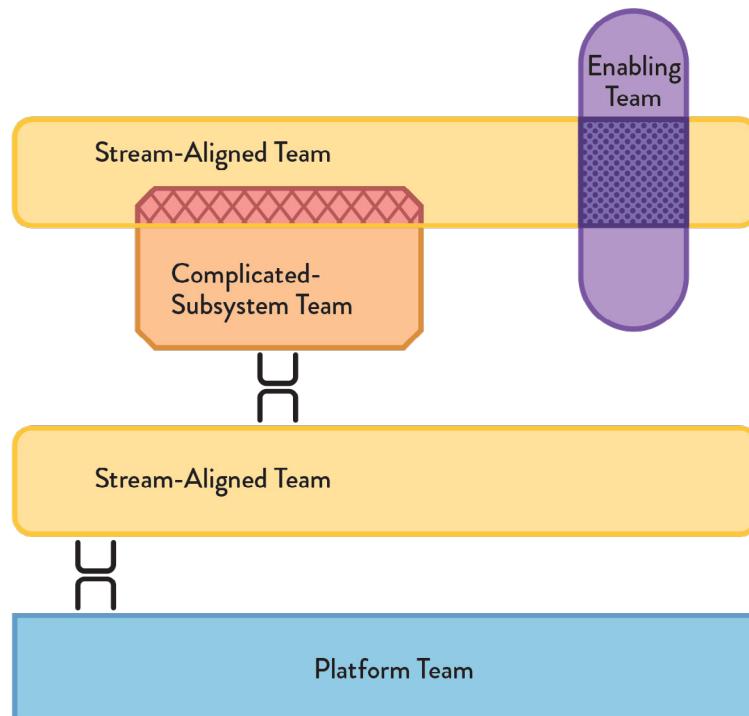


Figure 1. The four team types and their different interaction modes. Source: *Team Topologies* by Manuel Pais and Matthew Skelton.

Curated team interactions

For example, a stream-aligned team might currently be interacting with two other teams: a test-automation enabling team (using facilitating) and a face recognition “complicated subsystem” team (using collaboration). In this case, there could be two temporary chat tool channels to clarify these interactions:

- **#testautom-facilitating-green**—the test automation team is facilitating the green stream-aligned team
- **#facerecog-collaboration-green**—the face recognition team is collaborating with the green stream-aligned team

Furthermore, it can be hugely helpful to have channel names that make it clear where to get support or help for common or shared infrastructure or tools:

- **#support-environments**—the support channel for environments
- **#support-logging**—the support channel for logging

This makes it easy for people to “self-serve” and discover the best place to ask a question or ask for help. Similarly, set some conventions around the display name that shows in the chat for each person. “Jim” or “sara_b” provide much less context than something such as “Jim Ngo (infra platform team)” or “Sara Brown (green stream team).” With the more descriptive display names, we have immediate context for who they are and their team role.

Overcommunicate using just enough written documentation

In a remote work setting, it's vital to “overcommunicate.” Be very clear all the time about what you are working on, why, how, and when. Overcommunication feels almost like an externalization of your key decisions and reasoning so that people can easily reconstruct the sequence of thoughts that led you to your current work.

Overcommunication will take several forms: sharing small decisions in a chat tool, writing up larger decisions or designs in a wiki or document, and even creating a presentation or report to explain important concepts. Don't rely on people just

Curated team interactions

seeing scrolling messages in the chat tool.

"Being a good writer is an essential part of being a good remote worker," say Jason Fried and David Heinemeier Hansson in their classic 2013 book *REMOTE: Office Not Required*. The authors built the hugely successful company 37signals, starting fully remote in 2003. Among many other useful tips in their book, they explain that because most human-to-human interaction will be via chat and text media (such as wikis, documents, and so on), it is essential to emphasize good writing skills for remote work.

It's not just about typing lots of text, though. The text we type needs to have context when seen by itself. "Hi, what do you think?" requires a mental context-switch for the person reading the message (what does that question refer to?). But, "Hi. So do you think we should switch component A for component B due to the performance issues with A?" gives plenty of context for the reader.

Don't make it hard for people to discover meaning in written communications; make the messages self-contained.

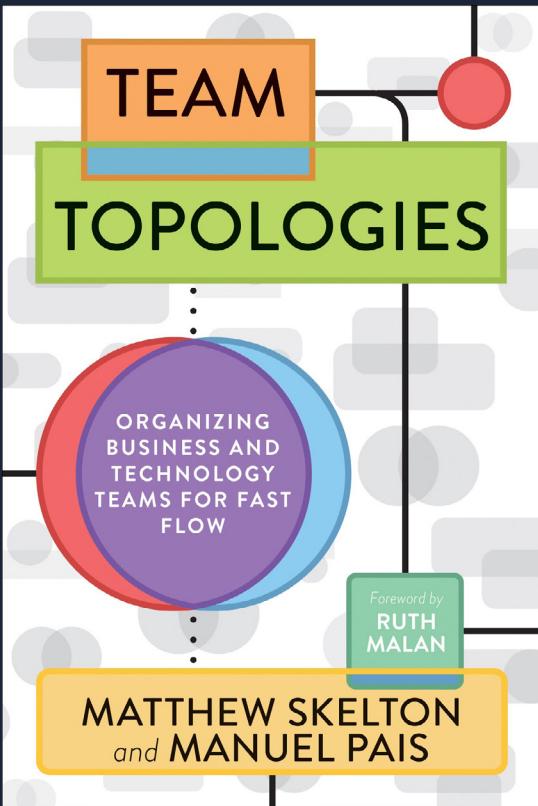
Design and define the ways that teams interact

Well-defined interactions are key to effective teams, and this is especially true for remote-work situations. Team-focused conventions within chat tools and wiki documentation increase discoverability and reduce cognitive load on communications.

By adopting clear ground rules and practices — such as team APIs and chat tool naming conventions — organizations can take advantage of remote-first ways of working to increase the chances of success with DevOps transformations, becoming more effective at software delivery.

[Original: <https://techbeacon.com/devops/are-poor-team-interactions-killing-your-devops-transformation>]

Read more



Team Topologies

Overview: teamtopologies.com/key-concepts

Book: Team Topologies (IT Revolution Press, 2019)
teamtopologies.com/book

More insights: teamtopologies.com/learn



TechBeacon.com is a digital hub by and for software engineering, IT and security professionals sharing practical and passionate guidance to real-world challenges.

Join the conversation:

techbeacon.com

About the authors

Matthew Skelton is co-author of *Team Topologies: organizing business and technology teams for fast flow*. Recognised by TechBeacon in 2018 and 2019 as one of the top 100 people to follow in DevOps, Matthew curates the well-known [DevOps team topologies patterns at devopstopologies.com](#). He is Head of Consulting at [Conflux](#) and specialises in **Continuous Delivery, operability, and organisation dynamics** for modern software systems.



Twitter: [@matthewpskelton](#) | LinkedIn: [linkedin.com/in/matthewskelton/](#)

Manuel Pais is co-author of *Team Topologies: organizing business and technology teams for fast flow*. Recognized by TechBeacon as a [DevOps thought leader](#), Manuel is an **independent IT organizational consultant and trainer**, focused on team interactions, delivery practices and accelerating flow. Manuel is also a LinkedIn instructor on [Accelerating Continuous Delivery in the Enterprise](#).



Twitter: [@manupaisable](#) | LinkedIn: [linkedin.com/in/manuelpais/](#)

About Team Topologies

Team Topologies is a clear, easy-to-follow approach to modern software delivery with an emphasis on optimizing team interactions for flow. Four fundamental types of team — team topologies — and three core team interaction modes combine with awareness of Conway's Law, team cognitive load, and responsive organization evolution to define a no-nonsense, team-friendly, humanistic approach to building and running software systems.

Devised by experienced IT consultants [Matthew Skelton and Manuel Pais](#), the Team Topologies approach is informed by the well-known [DevOps Team Topologies patterns](#) (also authored and curated by Matthew and Manuel). Matthew and Manuel have worked with many organizations around the world to help them shape their teams for modern software delivery, and Team Topologies is the result of that experience.



Team Topologies

organizing business and technology teams for fast flow:

book + training + consulting

teamtopologies.com

Copyright © 2017-2020 Conflux Digital, Ltd. All Rights Reserved.

Registered office: 67 Kirkstall Avenue, Leeds, LS5 3DW, UK

Registered in England and Wales, number 10890964. VAT registration number GB280146126