

MINI-BOOK
MB81 – v1 MAY 2023



FINDING SOFTWARE BOUNDARIES FOR FAST FLOW

Team Topologies and Domain-Driven Design

Team-sized software

Team-first tools and skills

Curated team interactions

Key industry insights in 5 articles

Finding software boundaries for fast flow






Team Topologies and Domain-Driven Design

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.



In this mini-book:

	Introduction	3
	Building Adaptive Systems for a Fast Flow of Change	5
	About Team Topologies and Context Mapping	19
	Exploring Team and Service Relationships with Team Topologies and Context Maps	31
	Architecting Your Business with Domain-Driven Design and Team Topologies	42
	Finding Good Stream Boundaries with Independent Service Heuristics and User Needs Mapping	50

Introduction



Matthew Skelton, Co-author of Team Topologies



In the world of modern software development, speed is a major differentiator. The arrival of cloud computing has transformed the way in which software is developed and has substantially reduced delivery times. Any organization that cannot deliver (and sense the market) fast enough will struggle to compete, and therefore achieving a fast flow of change is essential. Business agility and faster software delivery requires organizations to not only consider the technical aspects of software development but also the social structures and team interactions. Effective flow of value requires an understanding of boundaries between domains, something that Domain Driven Design (DDD) has been helping to achieve for many years. However, it is also important to understand the dependencies and interactions between the teams that own those domains. This is something that can be achieved by looking at the organization through a Team Topologies (TT) lens.

In 2003, Eric Evans produced the first edition of [Domain-Driven Design: Tackling Complexity in the Heart of Software](#) with the aim of simplifying how software is developed by allowing developers, domain experts, business owners and clients to communicate effectively with each other in order to solve complex problems. For nearly 20 years this book has been the cornerstone of the DDD community and has long stood the test of time. During that time approaches to software development have evolved, the introduction of cloud computing etc has enabled a paradigm shift towards thinking about software boundaries differently and optimizing for the flow of value through an organization.

The book *Team Topologies* by Matthew Skelton and Manuel Pais — published in September 2019 — introduced a new way of thinking about organization dynamics and software architecture design, via the principles of well-defined team types and team interactions, by taking a team-first approach, and considering cognitive load as a constraining factor on team size and team

interactions. TT provides a fresh combination of principles and practices that help evolve organizations towards effective collaboration, autonomy, delivery focus and product alignment ultimately enabling a faster flow of change.

Several people in the wider DDD community, especially those engaged in the concept of socio-technical architectures, have identified some similar concepts and intentions between DDD and TT but with subtle differences in approach that encourage them to be complementary. This mini-book explores the similarities, differences and crossovers between DDD and TT in a series of insightful articles that will help you to apply the ideas from both areas effectively.

Building Adaptive Systems for a Fast Flow of Change

Susanne Kaiser, independent tech consultant

In a world of rapid changes and increasing uncertainty, organizations must adapt and evolve continuously to remain competitive and excel in the market. To adapt and evolve, an organization, its business strategy, and its architecture must be able to anticipate and absorb change.



Combining Wardley Mapping, Domain-Driven Design, and Team Topologies helps us to connect the dots between business strategy, software design and architecture, and team organization to build adaptive systems for a fast flow of change.

The business strategy perspective with Wardley Mapping

Wardley Mapping is a strategy framework invented by Simon Wardley. Wardley Mapping helps to design and evolve effective business strategies based on situational awareness and movement following a strategy cycle. According to Wardley, the strategy cycle 'is a representation of change and how we need to react to it'. The strategy cycle consists of five sections based on Sun Tzu's five factors that matter when competing (or going to war):

Adaptive Systems

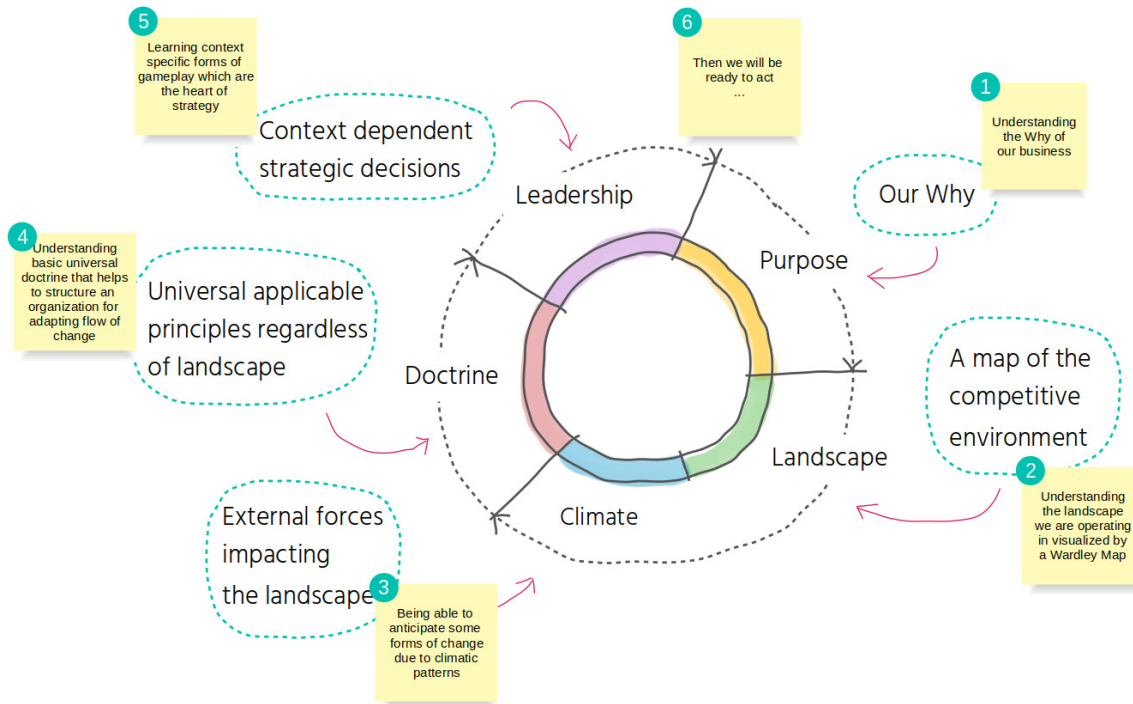


Figure 1: The strategy cycle of Wardley Mapping

The strategy cycle starts with the **purpose** — the 'why' of the business. The purpose represents the reason why customers choose your business over others and what inspires the customer to act. The **landscape** is a description of the competitive environment in which an organization is operating. The landscape is visualized by the Wardley Map. The **climate** consists of patterns that describe external forces and rules impacting the landscape, over which we have no control. Discovering and understanding the climatic patterns is important to see how the landscape is changing. Some of the climatic patterns can be anticipated, providing a competitive advantage. The **doctrine** describes universal, context-independent principles that all organizations can apply regardless of their landscape. These are principles for successful operation that enable organizations to absorb and adapt gracefully to a fast flow of change (organizational fitness). The **leadership** is the context-specific decision-making component of the organization: what strategy to choose, considering the landscape, climate, and doctrine.

In this article, we are focusing on the landscape and some doctrinal principles.

Adaptive Systems

Visualizing the landscape - the Wardley Map

The landscape is a description of the environment in which an organization is operating and competing. The landscape is represented on a map called the Wardley Map. The Wardley Map is composed of a y-axis depicting a value chain and an x-axis representing evolution stages. A value chain describes what is needed to add and deliver value to a user. Creating a Wardley Map starts with defining the value chain. From top to bottom, a value chain consists of users, user needs, and components fulfilling the user needs directly or facilitating other components in the value chain. Components at the top are most visible with the biggest value to the users, e.g. where users are interacting directly with the system. The components at the bottom are less visible to the users. Users and user needs form the anchor of the map; all subsequent components are positioned in relation to these elements.

Let's build an example value chain for a conference event planning solution that enables conference organizers to manage an event (see figure 2).

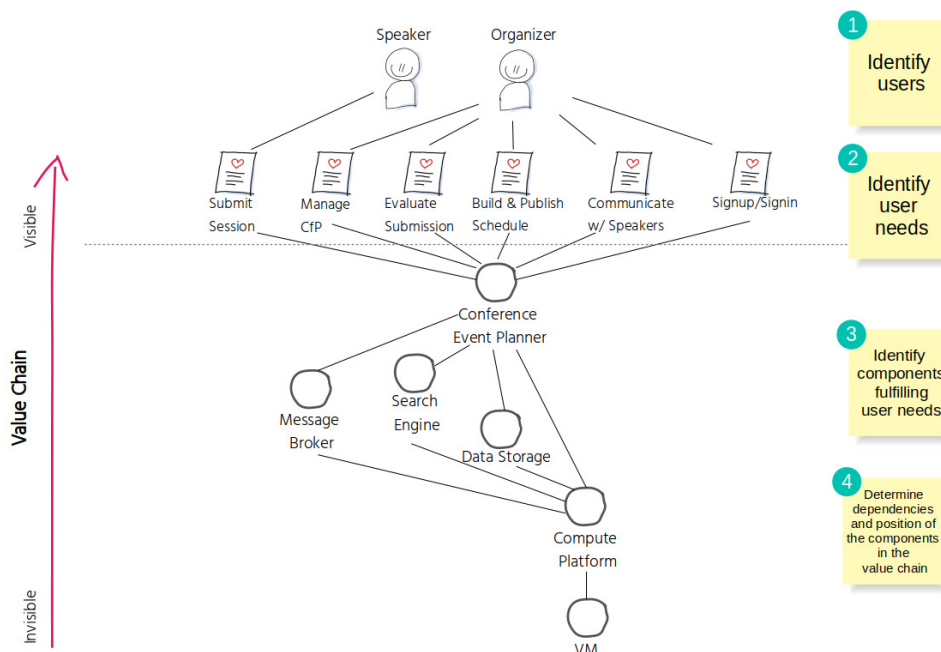


Figure 2: Example of a value chain for a conference event planning solution.

This solution mainly addresses the conference organizers and speakers as users. To identify the users' needs we can articulate a user journey. The conference organizers start their user journey with managing a call for papers (CfP) to which

Adaptive Systems

the speakers can submit their session proposals. The organizers need to evaluate the submitted session proposals, build and publish a schedule from the accepted sessions, and communicate with the speakers. Both conference organizers and speakers need to sign-up and sign-in as well.

In the next step, we go further down the value chain and identify the components that fulfil the users' needs. We start with a single conference event planner component that contains the business functions that fulfill the aforementioned user needs directly. Later, we are splitting each component into smaller parts when bringing in the Domain-Driven Design perspective. The conference event planner component needs to store some sort of state. For that purpose, we need a data storage component that facilitates the conference event planner component. It is less visible to the user, so is positioned further down on the y-axis. For search functionality and event-driven, decoupled communication, we need to integrate a search engine and a message broker. They both facilitate the conference event planner component. The conference event planner component, data storage, search engine, and message broker need an environment in which their software is executed: the compute platform. This compute platform is running on top of a virtual machine.

Once we have defined the value chain, the next step is to map each component of the value chain to evolution stages on the x-axis. The evolution stages start with Genesis on the left, a stage that involves addressing the characteristics of new, undefined, and constantly changing components. Then comes 'Custom-Built', then 'Product' (+rental), such as off-the-shelf products or open-source software, then 'Commodity' (+utility).

As the components of a value chain evolve, their characteristics change. Asking about the characteristics of each component helps us to identify which component belongs to which evolution stage. For example, how well-understood and well-defined is the component? Is the component available as a product or utility service or is it rather new? How widespread is this component? Are all competitors using this component? Is this component providing a competitive advantage? Our conference event planner component, providing a competitive advantage, shall go into 'custom-built'. We are planning to use open-source solutions for the search engine, data storage, and message broker components, which locates them in the 'product' (+rental) evolution stage. The components are going to be packaged in containers as a compute platform running on top

Adaptive Systems

of a cloud-hosted virtual machine residing in the 'commodity' (+utility) evolution stage.

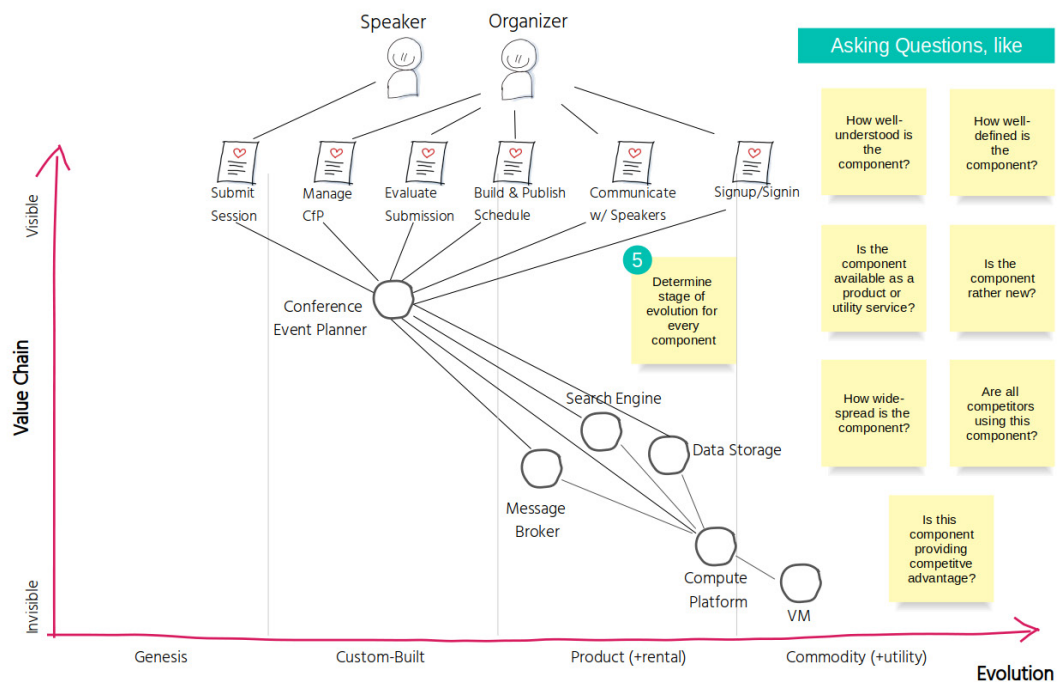


Figure 3: Example of components mapped to evolution stages.

Before switching to the climatic patterns as the next section of the strategy cycle, we would like to focus on the conference event planner component by bringing in the Domain-Driven Design perspective. This helps to analyze and understand the problem domain and partition the problem domain into modular parts.

The Domain-Driven Design perspective

Domain-Driven Design (DDD) is about designing software based on Domain Models, as proposed by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. The core concept of DDD is that to build better software, its design needs to align with the business domain, business needs, and business strategy. Before diving straight into developing a technical solution that solves the user needs, it is necessary to understand the problem domain, and that's where DDD is very helpful. An essential part of DDD is the collaboration between domain experts and development teams. They analyze the problem domain to obtain domain knowledge. The problem domain or business domain refers to the scope, area, or process of an organization. The domain

Adaptive Systems

knowledge should be free of technical jargon and described in terms of a shared, business-domain oriented language - the ubiquitous language.

DDD comes with patterns and practices consisting of strategic and tactical design. To illustrate these, I would like to reuse the y-axis of a Wardley Map and apply a top-down approach. We start with the problem space of strategic design at the top and go to the solution space of tactical design at the bottom. Analyzing the business domain and discovering subdomains as subparts of the problem domain help to build the problem space of strategic design. In connection with Wardley Mapping, the problem domain can be considered as the users and their user needs.

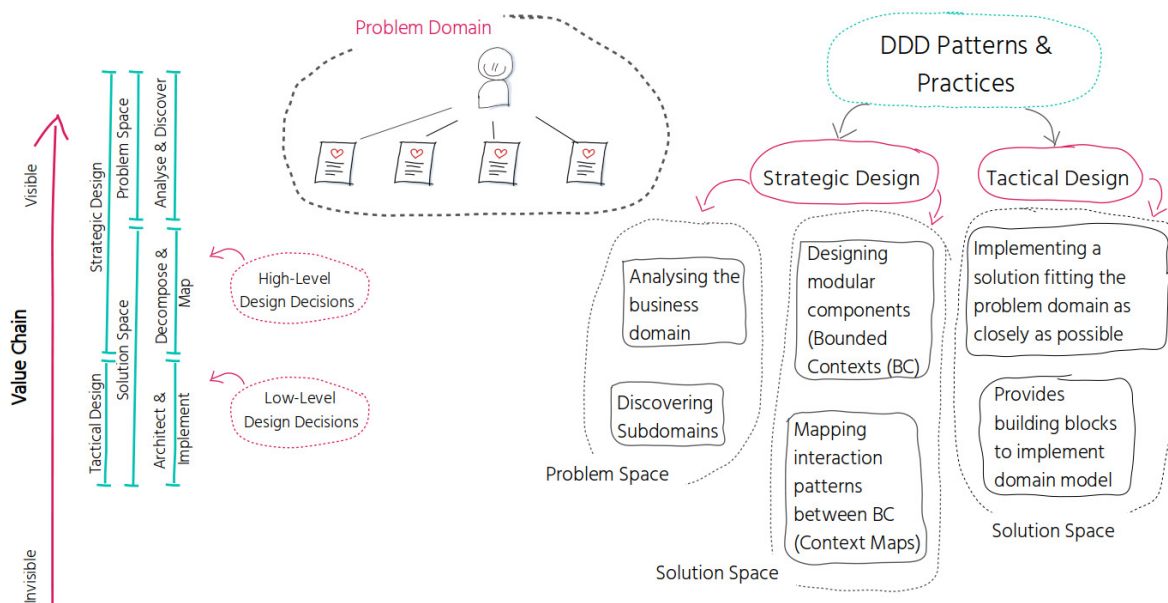


Figure 4: Strategic and tactical patterns & practices of DDD.

When we switch to the strategic design's solution space, we start by decomposing the problem domain (including subdomains) into modular components (the Bounded Contexts) and mapping interaction patterns between them (the context maps). Later, when we architect and implement a solution fitting the problem domain, we are moving to the tactical design's solution space.

In this article, we are focusing on strategic design with its subdomains and Bounded Contexts and leaving out context maps and tactical design (for simplicity).

Discovering subdomains and determining their evolution stages

Distilling the business domain reduces complexity by partitioning the broad, abstract business domain into smaller, concrete parts: the subdomains. However, not all subdomains are equal. Some are more valuable to the business than others and this is reflected by assigning them a category: core, supporting, or generic (see Figure 5). These different categories can help us find the subdomains and prioritize the development effort.

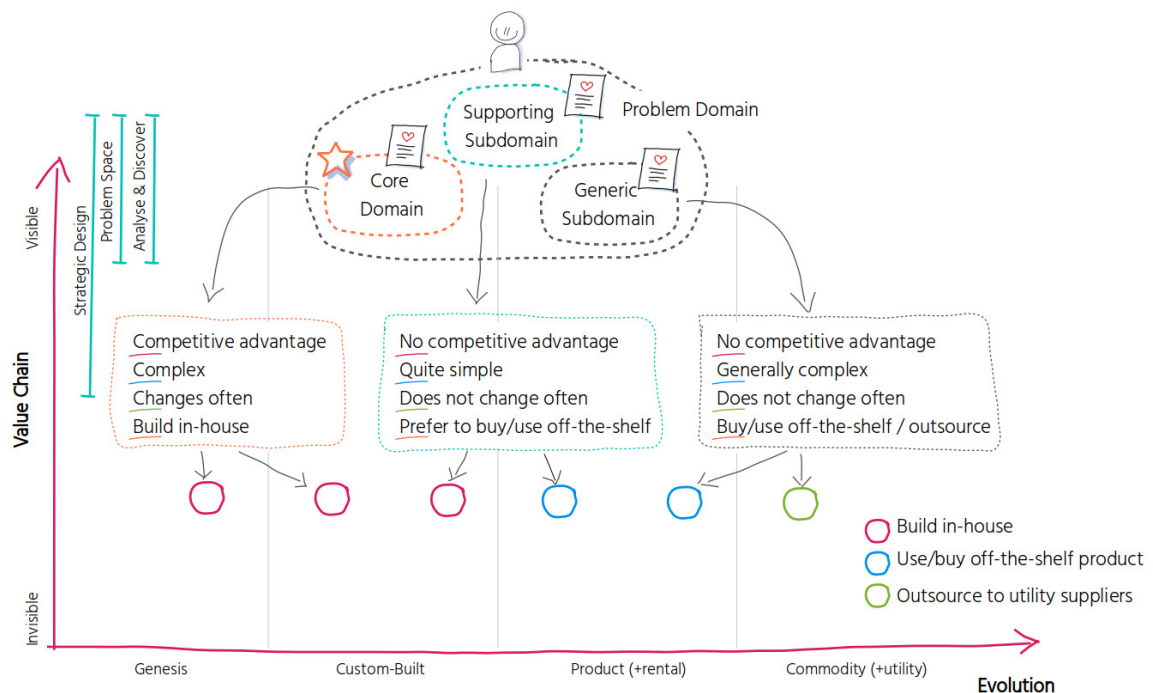


Figure 5: The subdomain categories and their evolution stages.

The subdomain categories can be mapped to evolution stages. One of Wardley's mapping principles is to use appropriate methods for each stage. There is no one-size-fits-all method, but appropriate methods per evolution stage, since each evolution stage comes with different characteristics that need to be handled differently. For Genesis and Custom-Built, it's appropriate to build the components in-house, preferably using agile methods. For product (+rental), it's more appropriate to use or buy off-the-shelf products or use open-source solutions, preferably using lean methods. For commodity (+utility), it's appropriate to outsource to utility suppliers, preferably applying Six Sigma methods.

Adaptive Systems

The **core domain** is the essential, business-critical part of the problem and it provides a competitive advantage. The core domain should be complex enough to make it difficult for competitors to copy or imitate. This subdomain tends to change often. The core domain is the subdomain to strategically invest in most and is supposed to be built in-house, residing predominantly in Genesis or Custom-Built stages. For example, the core domains for the conference event planner could revolve around managing the call for papers, handling submissions, evaluating sessions, and building the schedule.

The **supporting subdomain** helps to support the core domain but does not on its own provide a competitive advantage. It tends to be quite simple and does not change often, but might - in some cases - require some form of specialization. If possible, buying or using off-the-shelf products or using open-source software is a good approach. For this reason, the supporting subdomain is mapped to the Product (+rental) evolution stage. If some level of specialization is required, custom-building the supporting subdomain might be necessary. In this case, the supporting subdomain moves into the Custom-Built evolution stage. In the context of the aforementioned conference event planner, the messaging subdomain is a candidate for a supporting subdomain. It does not provide a competitive advantage but supports the core domain in such a way that it eases the interaction between the conference organizers and the speakers during the conference event planning process.

The **generic subdomain** is a subdomain that many business systems have. Examples include authentication and payment services. The generic subdomain is not core and does not provide a competitive advantage but is a category of products that businesses cannot work without. They can be complex but already solved by someone else. For the generic subdomain, it makes sense to buy or use off-the-shelf products or use open-source software or outsource to utility suppliers. This means that the generic subdomains are predominantly located in the Product (+rental) or Commodity (+utility) evolution stage. For the conference event planner, the identity and access management service that fulfills signing-in and signing-up of users is a good candidate for a generic subdomain.

Domain Models and Bounded Contexts

At this point, we have identified the subdomain categories in the problem space of strategic design. When switching to the solution space of strategic

Adaptive Systems

design, the next step is to decompose the solution, in this case the conference event planner, into modular components — the Bounded Contexts. A Bounded Context defines where a single Domain Model can be applied. A Domain Model represents the domain logic and business rules that are relevant to that area of the system. It forms a unit of mastery, purpose, and autonomy. But, a Domain Model cannot exist without a boundary, and that's where the Bounded Context comes in. A Bounded Context provides different types of boundaries. A Bounded Context forms a linguistic and semantic boundary around the Domain Model so that the language of the Domain Model is consistent and clear inside its Bounded Context. A Bounded Context serves as an ownership boundary: it should be implemented by one team only, but a single team can own multiple Bounded Contexts. A Bounded Context also serves as a physical boundary and can be implemented as a separate solution. In addition, the architectural and business logic implementation patterns can vary from context to context.

Conversations between domain experts and development teams about identifying the main outcomes of development may result in Bounded Contexts and Domain Models.

At some point, we might encounter ambiguity. For example, a Domain Model called 'session' may have more than one meaning. A session scheduled for the agenda has different attributes and business rules from a session proposal submitted by a speaker. This requires adjusting the ubiquitous language ('session') by introducing the phrases 'submitted session' and 'scheduled session' to make the meanings consistent and clear. Adjusting the language is a good indicator of a Bounded Context.

There are several techniques available to derive Bounded Contexts and Domain Models, such as EventStorming, Domain Storytelling, Example Mapping, User Story Mapping, etc.

For our example, we derived the Bounded Contexts of submission handling, CfP management, session evaluation, schedule management, messaging, and identity and access management. We placed them in their evolution stages according to their related subdomain categories (see Figure 6).

Adaptive Systems

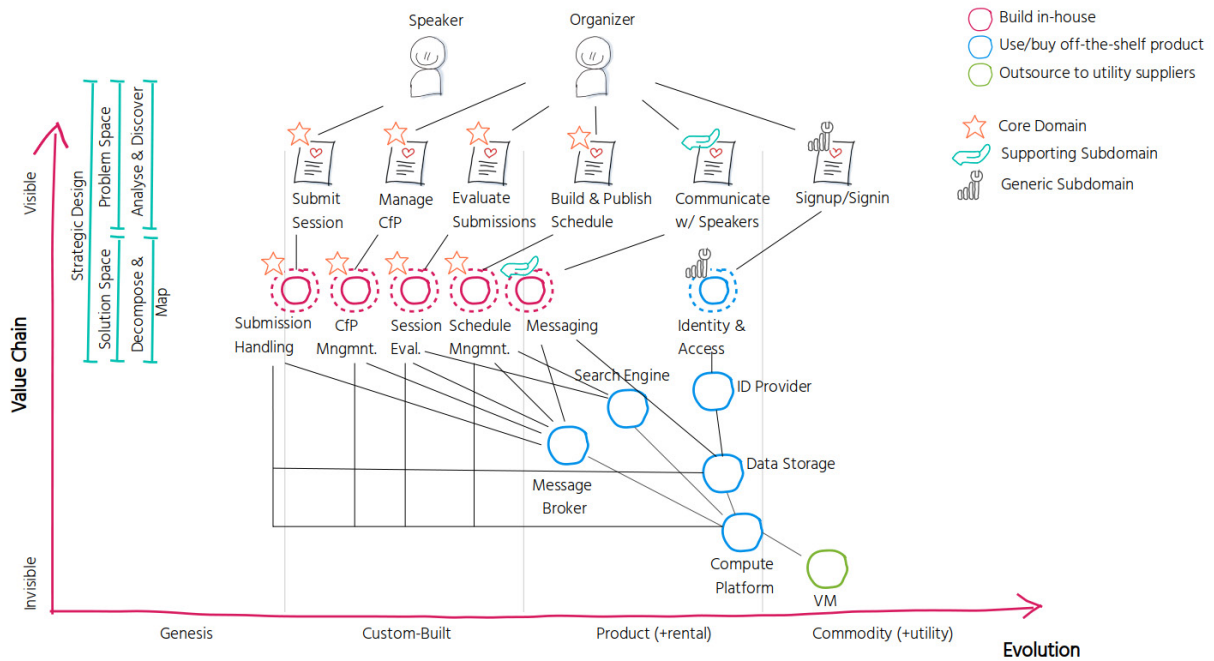


Figure 6: Bounded Contexts mapped to evolution stages.

Strategic design and Wardley's Doctrine

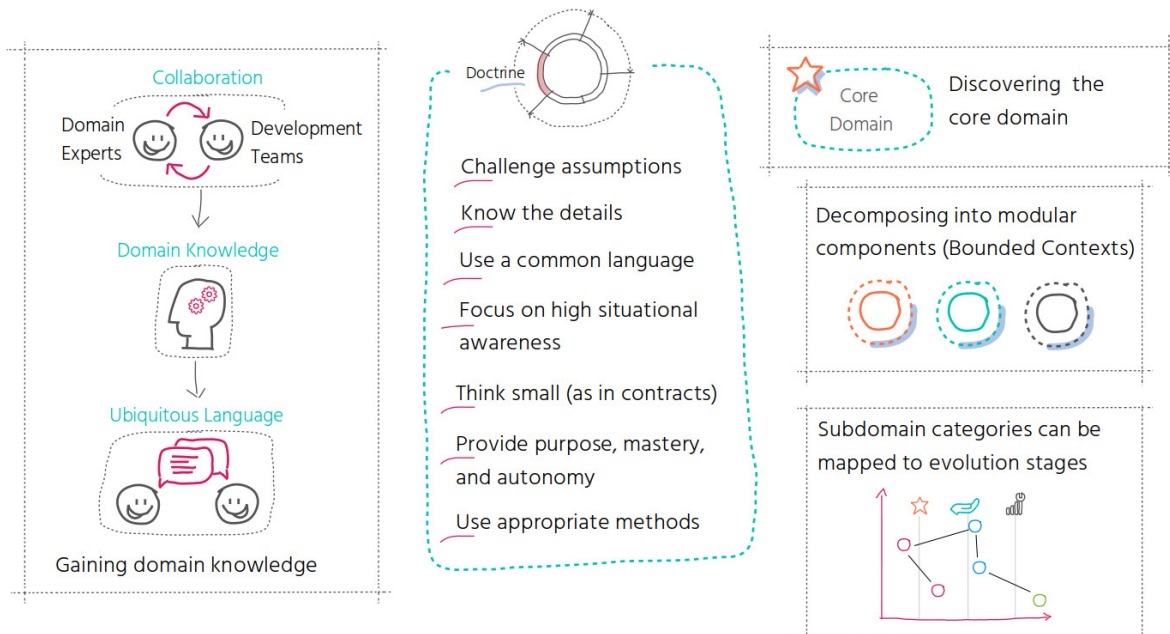


Figure 7: Strategic DDD applying Wardley's Doctrine.

The strategic design also helps to apply universal principles of Wardley's Doctrine (see Figure 7). At DDD's center is the close collaboration between domain experts and development teams, enabling us to analyze the business domain and challenge assumptions. Through this collaboration, we are gaining

Adaptive Systems

domain knowledge, which helps us to know the details of the business domain. The domain knowledge is described in a shared language. Discovering the core domain that provides a competitive advantage lets us focus on high situational awareness and understand the landscape in which we are operating and competing. Decomposing our system into modular Bounded Contexts enables us to partition our problem domain into smaller contracts. Bounded Contexts form a unit of mastery, purpose, and autonomy that conforms with the related doctrinal principle. The subdomain categories can be mapped to evolution stages, which leads to using appropriate methods per evolution stage.

After having visualized the landscape with a Wardley Map, discovered subdomain categories and their evolution stages, and decomposed the problem domain into Bounded Contexts, we next need to bring in the team organization perspective.

The team organization perspective with Team Topologies

Conway's Law states that 'any organization that designs a system [...] will produce a design whose structure is a copy of the organization's communication structure'. For example, an organization composed of functional silo teams, such as UI-, backend-, and database-administration teams will inevitably produce a multi-tier architecture consisting of presentation, business logic, and data storage tiers. There's nothing wrong with multi-tier architecture in general, but when implementing a change across several tiers requires handover between multiple teams, we increase communication and coordination efforts between teams, introducing bottlenecks, which impede delivery performance.

To optimize for a fast flow of change, we need to avoid functional silo teams and handovers. Instead, we need to aim for autonomous, cross-functional teams that are designing, developing, testing, deploying, and operating the systems for which they are responsible. Small, long-lived teams need to own the systems or subsystems on which they work. Minimizing the team's cognitive load is essential to avoid delivery bottlenecks. While communication within a team is desired, we have to restrict high-bandwidth communication between teams to enable fast flow.

And that's where Team Topologies comes in, as it 'advocates for organization

Adaptive Systems

design that optimizes for flow of change and feedback from running systems.' Team Topologies introduces four well-defined team types (see Figure 8) and three well-defined interaction modes (see Figure 9), promoting organizational effectiveness.

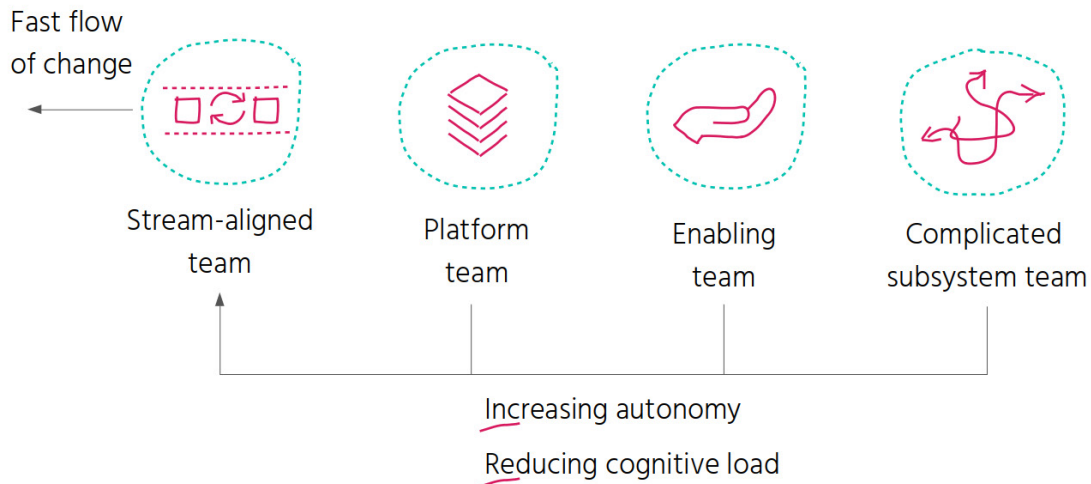


Figure 8: The four team types of Team Topologies.

Stream-aligned Teams are aligned to a continuous stream of work, e.g. a product, a service, a set of features, etc. They have end-to-end responsibility and aim to produce a steady flow of feature deliveries and incorporate feedback. To focus on a fast flow of change, the stream-aligned teams rely on the other team types that aim to increase the autonomy of the stream-aligned teams and reduce their team cognitive load. **Platform Teams** are responsible for platforms that typically abstract away infrastructure, networking, and cross-cutting capabilities. They provide internal, self-service resources and tools for easily consuming that platform. **Enabling Teams** help other teams to acquire missing capabilities, e.g. making suggestions on tooling, practices, and frameworks. **Complicated Subsystem Teams** support other teams on particularly complicated subsystems that require specialized knowledge.

Team Topologies also introduces three interaction modes (see Figure 9) for organizational effectiveness. With collaboration, teams work very closely together.

Adaptive Systems

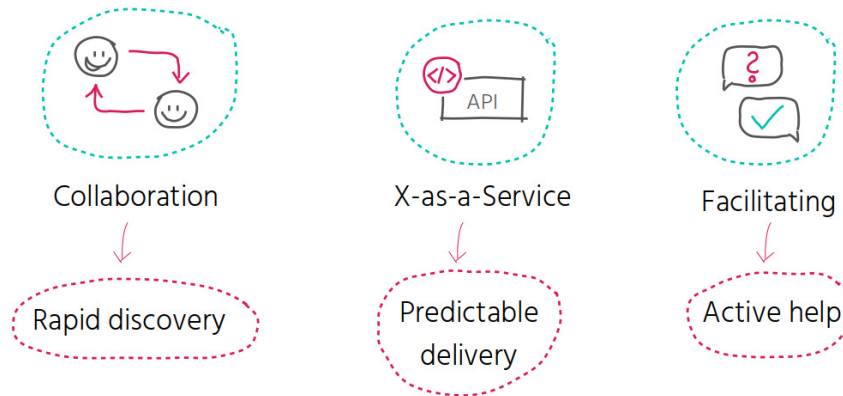


Figure 9: The three interaction modes of Team Topologies.

Collaboration is suitable for rapid discovery and innovation, e.g. when discovering new technologies. X-as-a-Service suits well when one team needs to use a code library, component, API, or platform that can be effectively provided by another team 'as a service'. Facilitating comes into play when one team would benefit from the active help of another team. This interaction mode enhances the productivity, effectiveness, and flow of the help-receiving team.

Architecture for flow

Let's put the three perspectives together and apply Team Topologies to our previously created Wardley Map including subdomain types and Bounded Contexts. Optimizing for a fast flow of change requires knowing where the most important changes in a system occur - the streams of changes. The first step in applying Team Topologies is identifying suitable streams of change. The previously identified user needs represent activity-oriented streams of change. Next is finding suitable team boundaries. The Bounded Contexts represent good team boundaries for stream-aligned teams as they serve as well-defined ownership boundaries forming a unit of purpose, mastery, and autonomy. Clear responsibility boundaries are necessary to optimize for team cognitive load. Assigning Bounded Contexts to a single team rather than sharing them across teams creates clear ownership. However, one team can own several Bounded Contexts. In addition, we need to match the boundary size to the team cognitive load and limit the number and complexity level of Bounded Contexts per team. The next step is to identify services needed to support a reliable flow of change. That brings the focus on the platform-related components of our Wardley Map, located in the Product (+rental) and Commodity (+utility) evolution stages. To focus on a fast flow of change, the stream-aligned teams rely on a platform team

Adaptive Systems

providing an easily consumable platform-as-a-service. The platform team might start with a thinnest viable platform that is 'just big enough' and can evolve later into a digital platform with self-service APIs and tools. That might result in a team constellation illustrated in figure 10.

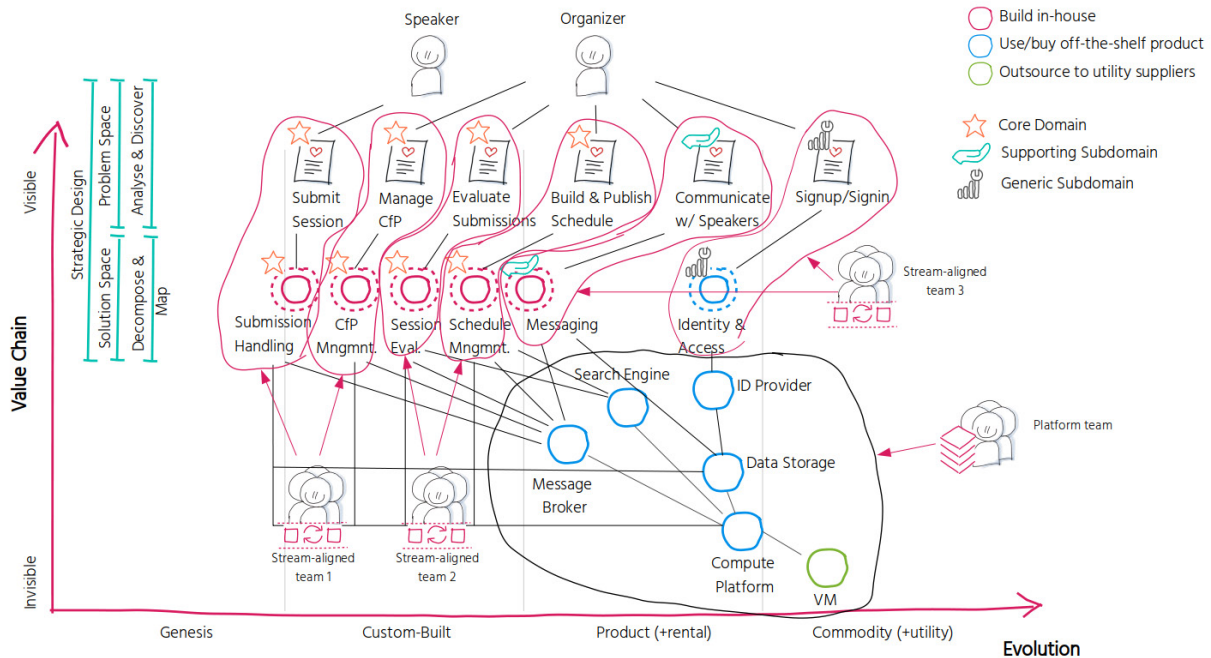


Figure 10: Possible team constellation.

Team Topologies and Wardley's Doctrine

The combination of the four team types and the three interaction modes of Team Topologies promotes organizational effectiveness also from the perspective of Wardley's doctrine (see Figure 11). Team Topologies helps to apply the doctrinal principles of 'thinking small as in teams', optimizing flow and reducing bottlenecks, providing purpose, mastery, and autonomy, and designing for constant evolution, where the system can handle a constant flow of changes without the need for constant restructuring.

About the author

Susanne is an independent tech consultant from Hamburg, Germany with a background in computer sciences and experience in software development for more than 17 years. Susanne is the author of Adaptive Systems with Domain-Driven Design, Wardley Mapping, and Team Topologies: Architecture for Flow and can be found at various international tech conferences as a speaker, co-organizer, or program committee member.

Twitter: @suksr

LinkedIn: susannekaiser1

About Team Topologies and Context Mapping

*Alberto Brandolini, EventStorming Creator
and founder of Avanscoperta*



The concept of Team Topologies, as depicted in the book by Matthew Skelton and Manuel Pais, is getting worldwide attention. Its focus on team structure and purpose is opening interesting discussions and many organizations are adopting the model as a reference for the organization of development teams.

At the same time, Domain-Driven Design (DDD) practitioners have a sense of *deja-vu* since the problem space seems to overlap with some of the advanced concepts of Context Mapping found in DDD. This is my attempt to see the best of both worlds and the pros and cons of the different approaches.

The problem space is mainly uncharted territory

Pais and Skelton hit a nerve with their book. Most organizations grow organically under the pressure of a never-empty backlog. As a result, teams growing in size will have to be split and competencies will progressively become more narrowed and specialized. However, the few criteria driving the splitting have been debated endlessly. Whichever the adopted decision - cut around *business capabilities* vs. cut around *technical skills* - the outcome is never perfect, leaving the decision-makers with a feeling of *maybe the other approach could have been better...*

To make matters worse, every organization places the responsibility of sizing and structuring teams in different hands: is this the responsibility of a team leader or head of development? Or CTO, maybe? Do we need a specific role for that? Do we always need to escalate to the ecosystem level or is the collaboration between teams just a local issue?

Have you been Spotified?

This lack of reference models also explains the popularity of the Spotify Model. An organization that dared to experiment with a different model for team structure and collaboration quickly became a source of inspiration, with a lot of unintended consequences. As also happened with lean and Toyota, other organizations followed the model, without paying too much attention to the ingredients that made it viable in that specific context.

In short, most nontrivial software development organizations experience some kind of friction in this space. Collaboration between software teams is one of the fundamental traits of successful companies, but frictionless collaboration seems to be a chimera, so it's not a surprise if IT professionals are seeking guidance.

Team Topologies categorizations

Team Topologies describes **four team types**:

- **Stream-aligned Team** focused on a specific business capability, ideally cross-functional;
- **Platform Team** providing common services to other teams;
- **Complicated Subsystem Team** with high specialization and specific knowledge about one portion of the system;
- **Enabling Team** experts who mentor and help other teams to evolve and improve.

While these concepts are not new — they stem from observation of many different ecosystems — it is interesting to see that they provide a vocabulary and a reference model. You may want to consider this list of archetypes as *attractors* for what a good team should be, but you should also note that *there are only four of them!*

Some weird things your organization is doing aren't listed here, *maybe for a good reason*. These are **reference implementations for teams that work well together**.

Three interaction modes

In terms of relationships between teams, only three patterns are described:

- **Collaboration** when two teams are working closely to a common goal for a limited period of time;
- **X-as-a-service** when there is a contact in terms of usage, potentially on an ongoing basis, but little specific collaboration;
- **Facilitation**, when a team is helping another to overcome impediments for a limited period of time.

Once again, the list is short (so patterns that are not listed are interesting for their absence), and the patterns are attractors. Failing to define the type of engagement will leave space for ambiguities that will turn into frictions and delays.

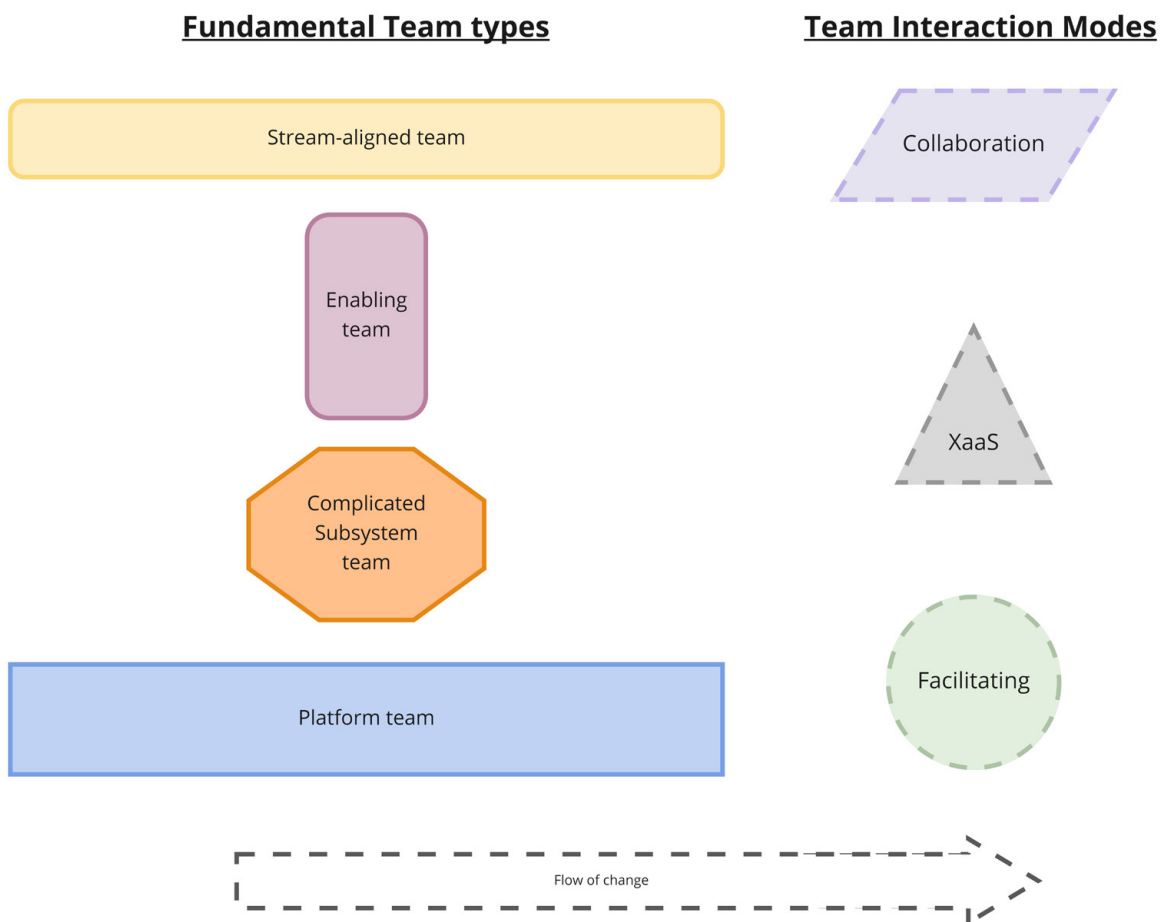


Figure 1: The Four Team Types and the Three Interaction Modes, directly from Team Topologies.

Context Mapping patterns

Strategic Domain-Driven Design offers an interesting point of view on the same problem space, but with different constructs.

There is little explicit reference to **teams** but the focus is mostly on **Bounded Contexts** (BCs) under the assumption that in a mid-sized software organization, there will be a close mapping between the two concepts.

Are Bounded Contexts teams?

No, Bounded Contexts are defined as *the limit of applicability of a given model*. But the sophisticated domain understanding that is a fundamental ingredient of Domain-Driven Design, is only possible if a single team is responsible for a given portion of the model. They need to connect on the entire socio-technical stack (talking to business representatives, writing and testing the code, interacting with users, and so on).

There's some well-accepted wisdom in the DDD community about it, so, let's make it explicit.

One team, multiple Bounded Contexts

Can a single team manage multiple BCs? Yes, this is the norm in small-sized organizations.

But, well... most small organizations don't seem to care much about boundary separation. In fact, the only way to achieve good separation is to explicitly enforce it: from big visible maps to segregation of the codebase.

The hardest thing to manage is in fact the cognitive load of the developers, especially if the backlog priorities force them to switch from one Bounded Context to another.

One can only be *deeply focused on one model at a time*. But without strong boundaries, it will be very easy to blur concepts that shouldn't be mixed. Just like working in a chocolate shop without ever eating it. The odds are not in your favor.

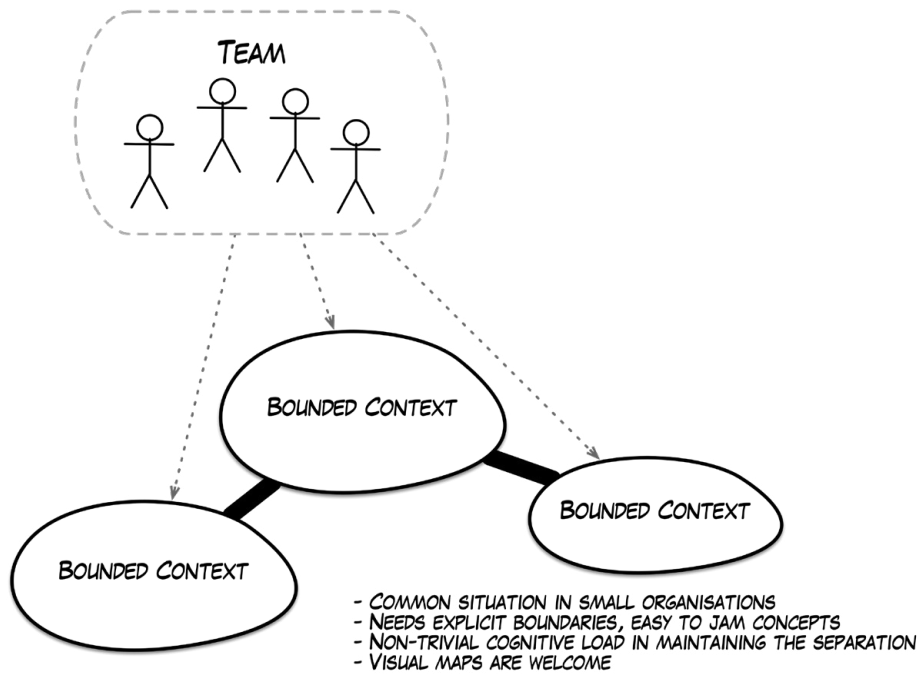


Figure 2: A single team working on multiple Bounded Contexts will pose some specific challenges.

Whilst one team/many models can happen, you've got to be very good at making the boundaries explicit.

More teams, one BC

Can we have two teams working on the same model? Apparently, yes. I mean: drawing it seems straightforward.

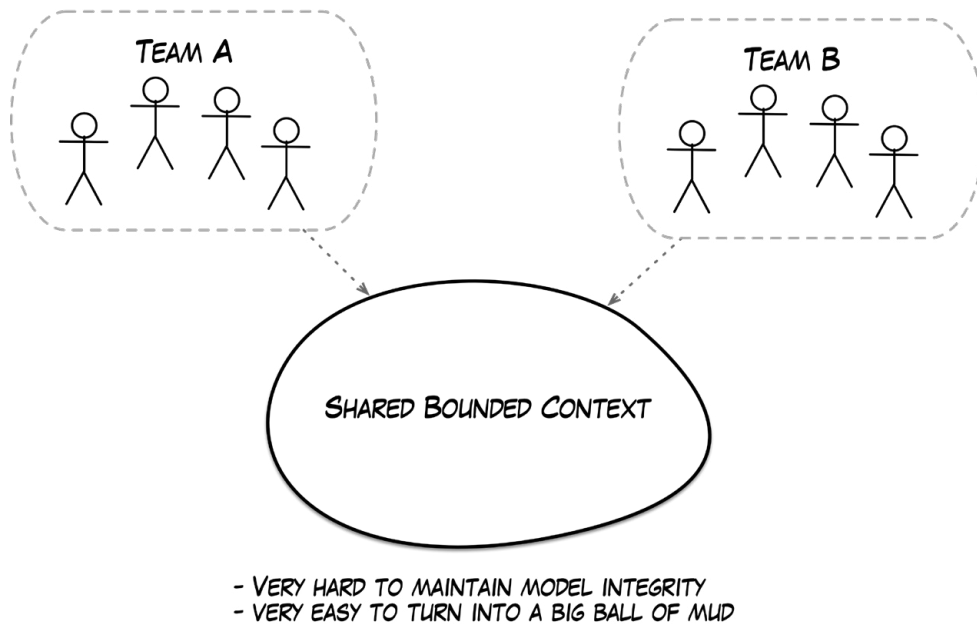


Figure 3: More teams on a shared Bounded Context seems like a highway to a Big Ball of Mud (BBoM).

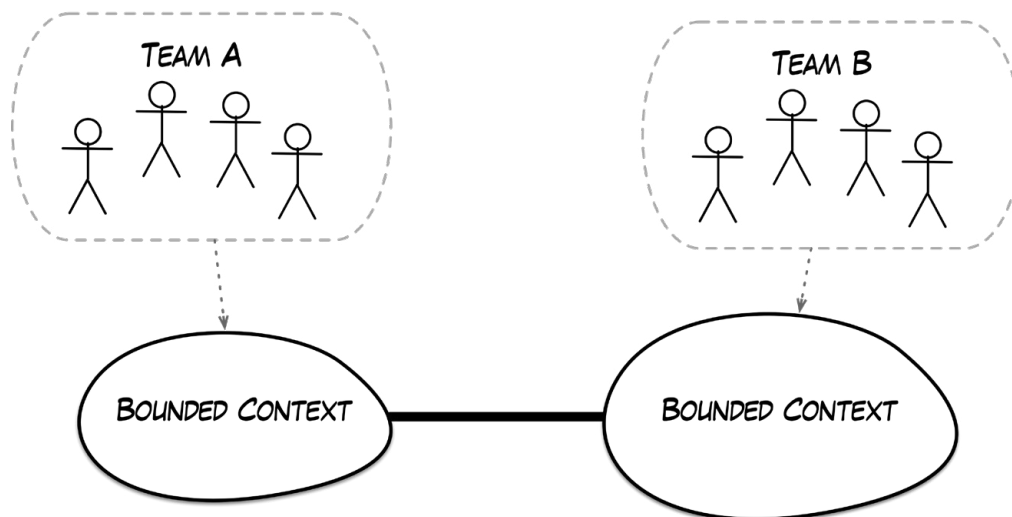
Context Maps

Unfortunately, two teams in the same Bounded Context is usually a prelude to a riot. Teams typically have different backlogs and priorities and ambiguity will quickly rot in the codebase, accelerating the transition towards a [Big Ball of Mud \(BBoM\)](#). In general, BCs won't be so big as to justify multiple teams working on them. So... No.

Interestingly, the Spotify model seems to move a little in that direction, relaxing the ownership of the codebase. This reduces coordination costs (the communication bandwidth, again), while mitigating the risks of decreasing quality through strong engineering practices. However, they don't have a BIG shared Bounded Context, but many smaller ones with relaxed ownership rules.

One team per BC

The perfect choice seems to be one team per Bounded Context.



- APPARENT NIRVANA STATE, MINIMAL COGNITIVE LOAD PER TEAM
- WELL DEFINED BOUNDARIES WILL MINIMISE THE NEED FOR CROSS-TEAM COORDINATION
- IMBALANCES IN EVOLUTION PRESSURE: NOT ALL BCs WILL REQUIRE THE SAME EFFORT

Figure 4: One team per Bounded Context could be the ideal, but there are challenges to manage.

One team, one Bounded Context. It seems the way to go, on paper. But... some details are left out of the picture. Like organization size, for example. I counted 20 different Bounded Contexts in our internal software the last time that I checked. Does it mean that we need $20 \cdot (7 \pm 2)$ developers to write our software? For a company of 6? Come on!

Size isn't the only problem here: the different BCs are not evolving at the same

pace. Some will be growing, others will stabilize and ideally be left untouched for a long time. But, the notion of an idle team isn't popular in the enterprise; one risk being filling up a backlog just because the team is available. The association between a team and a Bounded Context is temporary and will change over time.

Collaboration Patterns

The focus of Context Mapping isn't on the shape of the teams but on the relationships between models that have to support collaboration. One interesting concept is that of being upstream or downstream: mapping the political influence that parties hold within one model versus the other. Based on this type of reasoning, we have a few patterns describing possible collaboration types. Here is a quick summary:

- **Partnership:** two teams are mutually dependent and collaborate towards a common goal.
- **Customer-Supplier:** the goal is still common, but the dependency is less symmetrical, and priorities may differ. Negotiations are however possible.
- **Conformist:** no negotiations here. One model is adopted with minimal changes on the other side. The downstream party just gets what they are given.
- **Anti-Corruption Layer:** still on the downstream side, but we are not adopting the external model. In fact, we're writing a thick adapter to keep our model strictly separated from the outside one.
- **Open-Host:** on the upstream side, our model can be made available to external users, on our conditions. Of course, we'll need to make these conditions explicit with documentation and so on.
- **Published Language:** a common language on the communication channel could make a larger conversation possible, especially if the conversation language is maintained by a third party.
- **Shared Kernel:** a small portion of the software could be in common between different models, but this implies superior attention and quality in touching this code, not to mention the dependencies.
- **Separate Ways:** the best dependency is the one we don't have. Sometimes keeping things separated is the way.
- **Big Ball of Mud:** when boundaries are not in place and the codebase becomes a scary place to work in.

Context Maps

In DDD, this categorization becomes interesting when drawing a **Context Map** which is a useful artifact in a brownfield scenario. Drawing the map forces me to ask the relevant questions before starting the project. For example, I'd wave a red flag if my team was expected to be conformist on an unreliable platform, on a critical project for the business.

Satellite concepts

Since the early days, I enjoyed the ability of Context Mapping Patterns to capture the cost of the different approaches on different currencies. A Partnership requires less code but a lot of conversational bandwidth, while an Anti-Corruption Layer goes exactly in the opposite direction, writing more code since no conversation is possible. Conformist will let you go with little code and conversation, but asking a tribute in reduced quality instead. No such thing as a free lunch!

Conversational bandwidth is the key currency here. Collaboration won't happen if there's not enough bandwidth to support it.

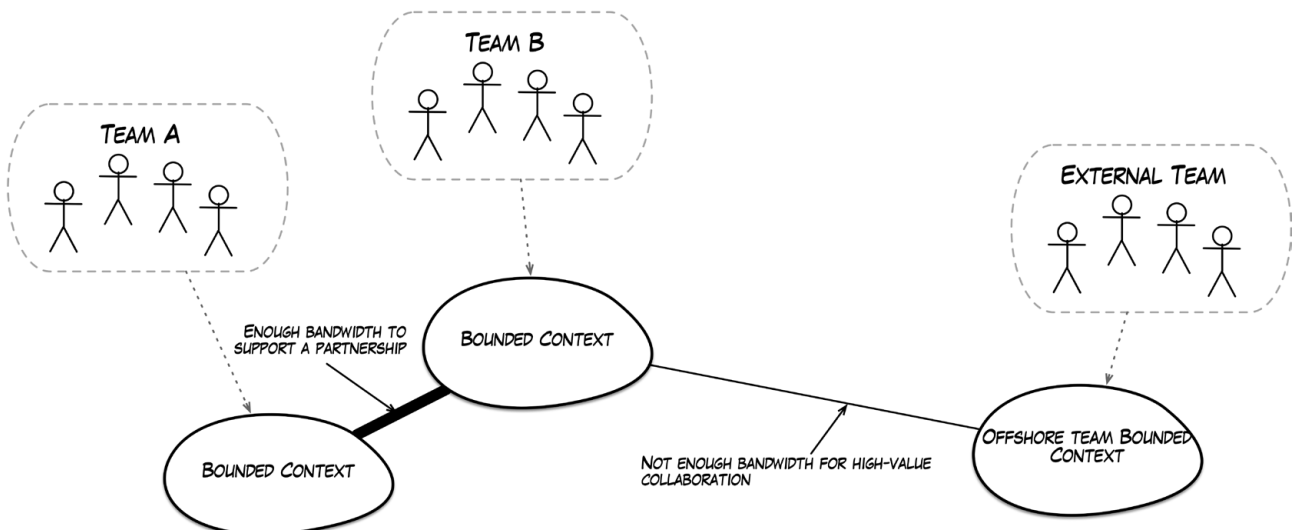


Figure 5: We can visualize bandwidth in a Context Map, with the thickness of the relationship.

Comparing the two approaches

The most obvious difference between Team Topologies and Context Mapping is that the latter doesn't explicitly talk about teams, but about *models* instead. On the other hand, Team Topologies suggests optimizing for *cognitive load*, which

maps pretty well to the notion of having one team per Bounded Context.

Present or future state

Talking about the different collaboration patterns, there's some overlap between the TT and DDD approaches, like DDD Open-Host and TT Platform Team, or DDD Partnership and Customer-Supplier with TT Collaboration. But the most interesting difference seems to be another one.

Team Topologies provides a reference towards a desirable to-be state, while DDD Context Mapping provides more fine-grained patterns for assessing the current state.

A Big Ball of Mud is clearly not a desirable state, but it's part of the dictionary needed to describe your horrible daily reality. I use Context Mapping to map the future state, but mainly as a software design tool, not so much as an organization design tool. Team Topologies seems to have an edge in being used as a reference model there.

Choices have consequences

One thing that the two approaches have in common is the ability to make the consequences of team-model allocation decisions explicit. **Collaborations are costly** and these costs need to be properly accounted for. I've been in too many places where the management was inviting teams to *communicate more* while filling backlogs and adding deadlines, making communication virtually impossible.

The Team Topologies lightweight dogmatism of *'there are only 3 types of interaction'* (paraphrased) is a good way to force management to make choices and own consequences. *Just telling people to collaborate and communicate is not a strategy.*

At the same time, quickly updating our current-state Context Map is still a great way to detect bullshit and inconsistencies in the current state of collaborations.

Context Maps

- "We are collaborating with team A. We have meetings every week!"
→ **Customer-Supplier?**
- "They just say 'No' to every single thing we ask." → **No... Conformist.**

Fracture planes

Team Topologies talks about fracture planes as a *natural* way to decompose the system into different streams. But I am too much into Domain-Driven Design to fall in love with this metaphor. Well, I know there are ideal places for cutting the system into loosely coupled Bounded Contexts and effectively splitting responsibilities between teams; I wrote an entire chapter in my book *Introducing EventStorming* about how to extract this information from a Big Picture EventStorming.

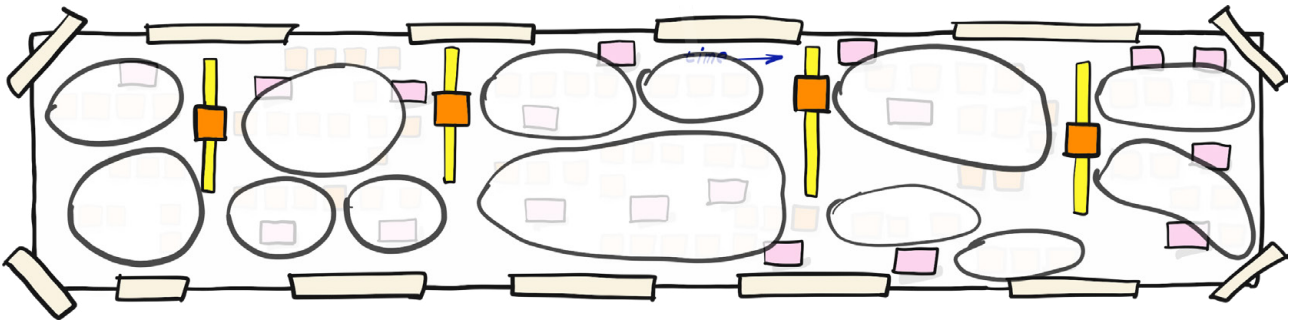


Figure 6: A Big Picture EventStorming will provide a massive amount of information about how to decompose your enterprise software system.

But... I also know that when it comes to *breaking the monolith*, it's never about cutting *slate*. It's more like chopping *trees where there are branches and nodes*. You'll try to follow the line, but you'll realize that something isn't getting separated that easily: there will be something that is *shared in the wrong way, but that's also big and dangerous to play with*.

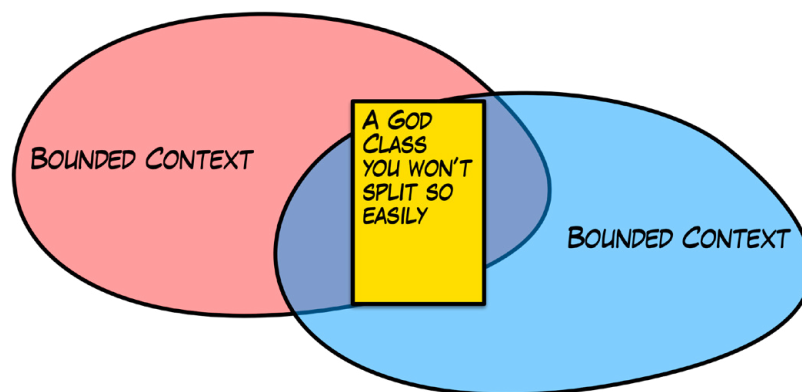


Figure 7: It can be a big class, a large table, or a combination of both. The big mess at the center of everything.

And this is where the metaphor sounds a little too easy for me. There are ways to deal with this problem (I talked about it in an old presentation: [What lies beneath](#)), but they require a given degree of mastery.

What's left out?

A few concepts are part of the game but seem to be missing from the conversation.

- **Organization size:** Team Topologies starts being interesting when your development shop has 20 or 50 or more people, while you'll have Bounded Contexts also when you're coding in solo mode.
- **Business pressure & portfolio management:** designing the perfect harmony between teams will fail miserably without some health check in terms of backlog pressure. If teams are under pressure, a healthy collaboration will not happen, but I still see too many organizations unable to plan for multiple dependent teams.

A journey, not a destination

I kept my favorite thought for last: I think the most important outcome of the discussion around Team Topologies nowadays is about what a good desirable state looks like. Most organizations have never seen a healthy organization (harsh but true), so providing a reference model is definitely a good thing.

Please don't fall into the trap of considering such a desirable state as stable or solving the problem of team structure once and forever. Every solution will be ephemeral by design. Collaborations are temporary and will need to be reviewed whenever the context changes. But you'll definitely have better tools to make the right decision.

About the author

EventStorming Creator, author of *Introducing EventStorming – An act of deliberate collective learning* and Founder of Avanscoperta, Alberto Brandolini is an all-around consultant in the Information Technology field. Alberto is a frequent speaker at software development related conferences in Italy and all over the world since rumors spread about his funny attitude. Besides consulting and running Avanscoperta, he has also been a trainer for UK-based company Skills Matters, where he taught Domain-Driven Design.

Twitter: [@ziobrand](#)

LinkedIn: [brand](#)

Learn

Academy

Increase awareness at scale with on-demand video learning on the Team Topologies Academy:

- **Team Topologies Distilled**
- **Platform as a Product**
- **Team Topologies for Managers**
- **Independent Value Streams with Domain-Driven Design**
- **and more**



Live online

Learn core Team Topologies principles and patterns with live online workshops delivered by official partners.



teamtopologies.com/learn

Exploring Team and Service Relationships with Team Topologies & Context Maps

Michael Plöd, Fellow at INNOQ

Over the past two years, there has been a great deal of enthusiasm in the Domain-Driven Design community around the book *Team Topologies* by Matthew Skelton and Manuel Pais. In particular, the book has been praised in the highest of terms by community members who are intensively involved with the topic of sociotechnical architectures. *Team Topologies* is primarily about setting expectations for team behavior and interactions (and therefore setting expectations for software behavior and interactions). In doing so, the authors have created an appealing verbal and visual language. However, in the area of strategic Domain-Driven Design there are also Context Maps, which focus at least partially on similar topics. This article will highlight where *Team Topologies* and Context Maps are similar, where there are differences, and, most importantly, how to combine both ideas well.



Introduction to Team Topologies and Context Maps

Team Topologies defines four different kinds of teams:

- **Stream-aligned Team:** aligned to a single valuable stream of work.
- **Complicated Subsystem Team:** builds and maintains a part of the system that depends heavily on specialist knowledge.
- **Platform Team:** provides a platform on which Stream-aligned Teams can deliver work autonomously.
- **Enabling Team:** Contains specialists who coach or mentor other (mostly stream-aligned) teams.

Context Maps

In addition, there are three different modes of interaction between those teams in Team Topologies:

- Collaboration
- X-as-a-Service
- Facilitating

Team Topologies uses a clear visual language:

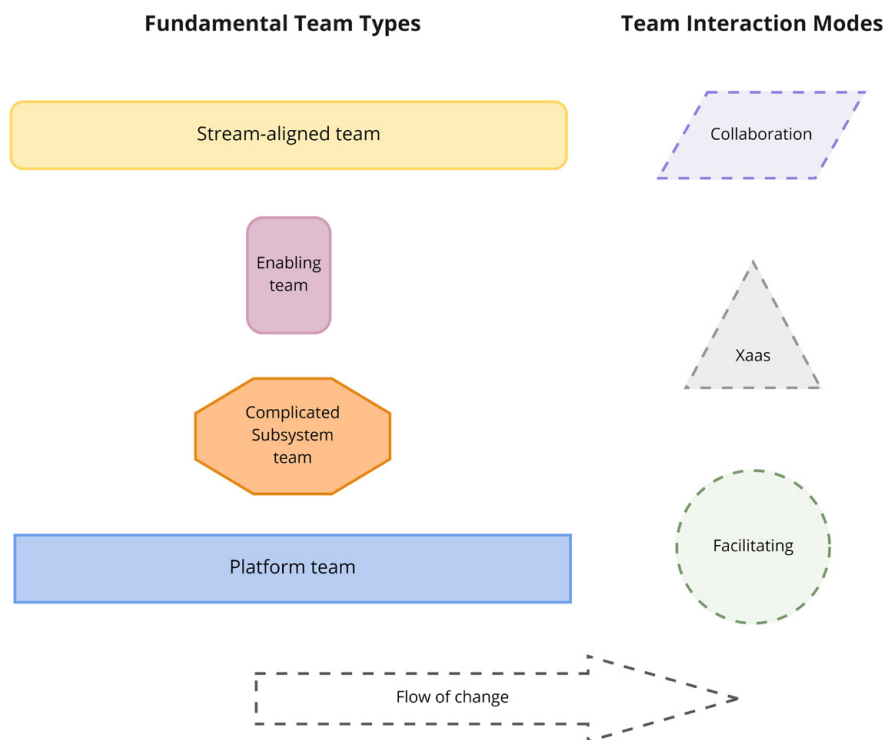


Figure 1: The visual language of Team Topologies

Stream-aligned Teams are shown as a yellow rounded rectangle (always horizontally to emphasize a left-to-right flow of change). Enabling Teams (in red) effectively 'cut across' the flow of change and are therefore shown vertically. Complicated Subsystem Teams deal with a discrete subsystem - shown as an orange octagon (or sometimes an orange diamond). Platform Teams are shown as a blue rectangle.

The Collaboration interaction mode is shown as a purple parallelogram. The X-as-a-Service interaction mode is shown as a gray triangle. The Facilitating interaction mode is shown as a green circle.

Context Maps

More details about the Team Topologies visual language can be found at shapes.teamtopologies.com

Context Maps were introduced 18 years ago in the original Domain-Driven Design book by Eric Evans. They define three types of team dependencies and 7 patterns. Vaughn Vernon added two additional patterns in his Implementing Domain-Driven Design book. Context Maps focus on the relationship between Bounded Contexts and the teams responsible for those Bounded Contexts. A Bounded Context can be defined as a boundary for a model expressed in a consistent ubiquitous language tailored around a specific purpose. Newer perspectives also mention the Bounded Context as a team-first boundary which minds the cognitive load of a team. The Team Topologies book itself dedicates a small chapter to this concept.

The team dependencies according to Context Maps are:

- **Mutually Dependent:** the actions of each team have an impact on the other team.
- **Upstream / Downstream:** The actions of one team have an impact on the other team, but no vice-versa.
- **Free:** there is no impact of actions on the other teams.

The patterns of Context Maps are determined by various questions like:

- Who provides services / interfaces (e.g. Open-host Service, Published Language)?
- How do systems deal with Domain Models (e.g. Conformist, Anticorruption Layer, Shared Kernel)?
- Who can raise requirements against others (e.g. Customer-Supplier)?
- How about the relationship between teams (e.g. Partnership, Separate Ways)?
- Which parts of a system are a mess (e.g. Big Ball Of Mud)?

An overview and description of all of the Context Map patterns can be found at github.com/ddc-crew/context-mapping.

How do Team Topologies and Context Maps correlate?

There are a couple of similarities between the two approaches. Both address the relationships between teams and to a certain degree the systems that they build. However, there are enough differences that make it interesting to combine both concepts.

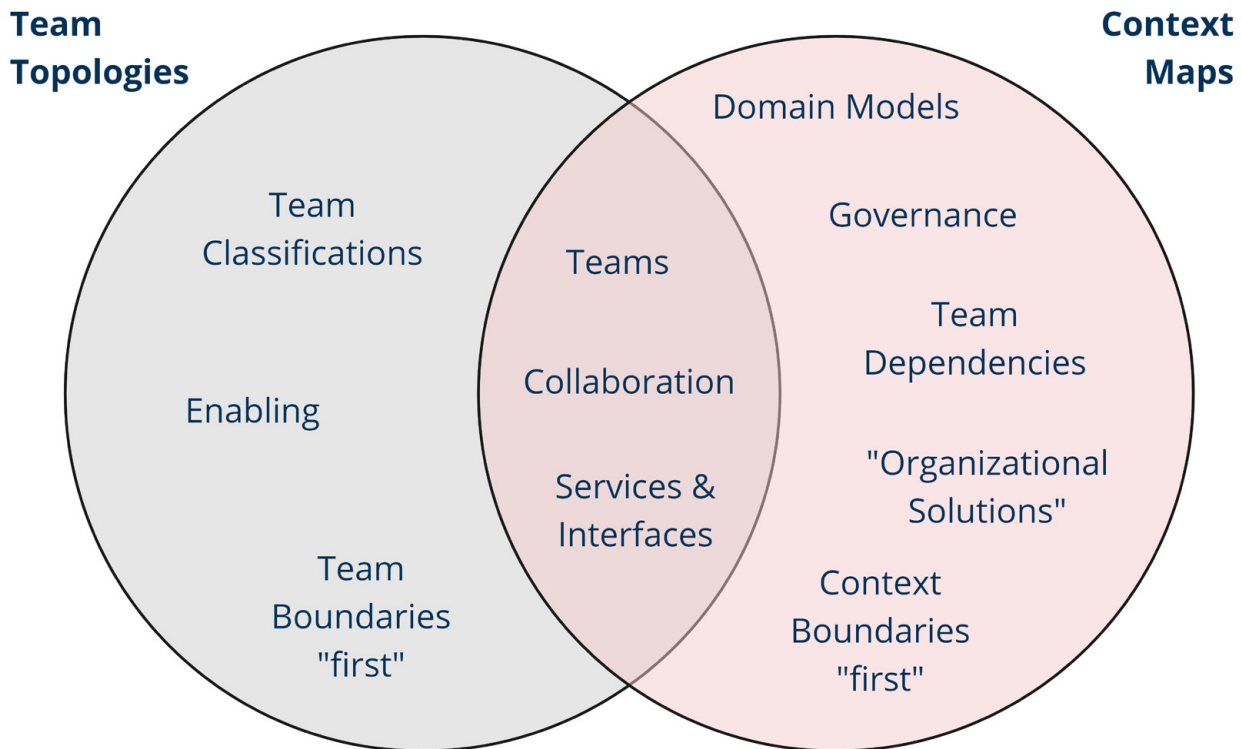


Figure 2: Similarities and differences between Team Topologies and Context Maps.

The similarities

Both ideas address the relationships between teams. Team Topologies focuses on this topic, whereas Context Maps contain various patterns that describe team-specific situations. In addition, there is a high degree of similarity between the Context Map's 'Open-host Service' pattern and the X-as-a-Service team relationship from Team Topologies. Both concepts also contain ways to describe a close collaboration between two teams and their systems. Team Topologies uses the Collaboration interaction mode for this while Context Maps use mutual dependency and the Partnership pattern for this scenario.

The differences and how to combine the approaches

The first and foremost difference is in the **perspective of each approach**. Context Maps first of all address the contact between Bounded Contexts whilst Team Topologies has a team-first perspective. In an ideal world, this would not be a big issue, since one would aim for a strong alignment between Bounded Contexts and teams. In this case, both Context Maps and Team Topologies will fit perfectly.

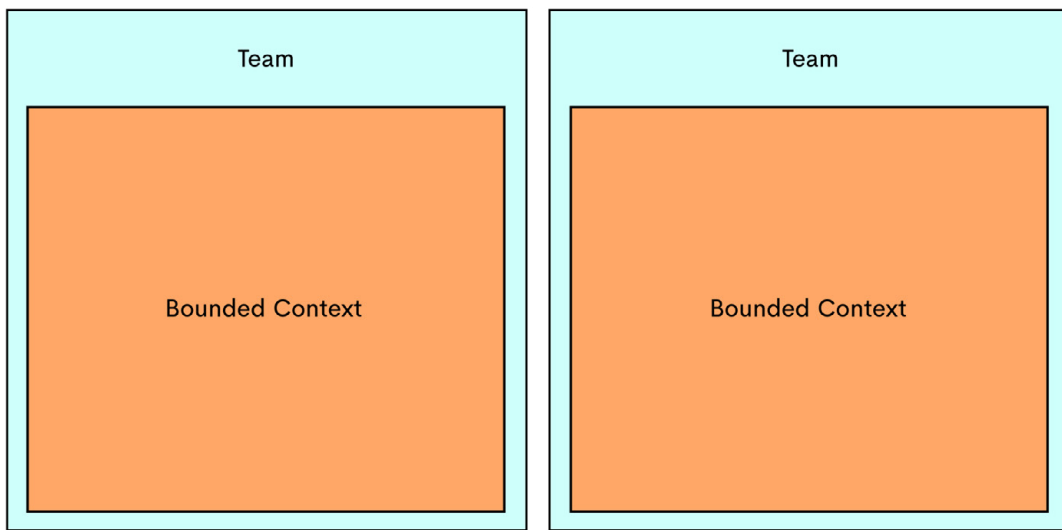


Figure 3: Visualizing a 1:1 relationship between teams and Bounded Contexts.

However, in the real world, we very often see one team being responsible for multiple Bounded Contexts:

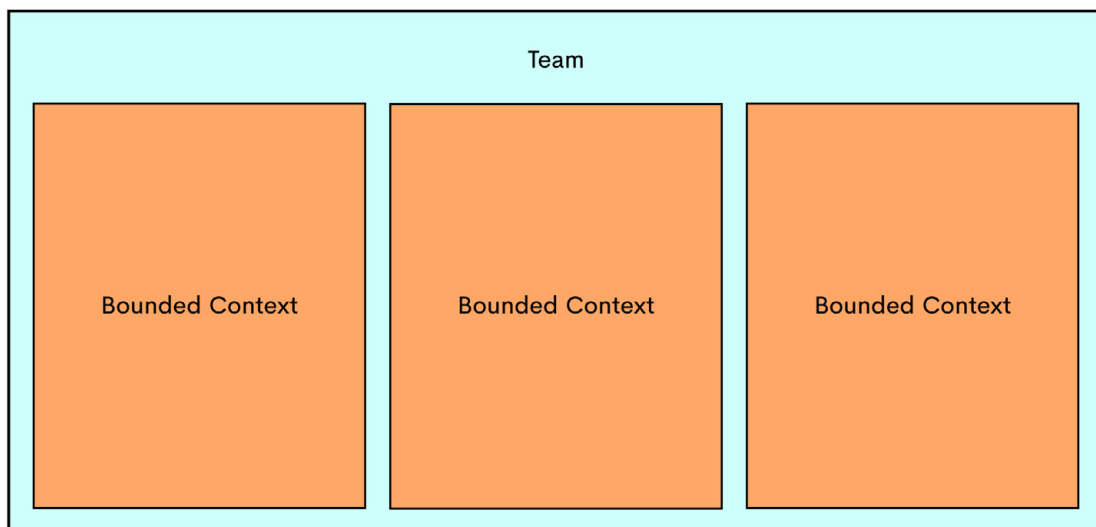


Figure 3: Visualizing a 1:1 relationship between teams and Bounded Contexts.

Context Maps

Since the Domain-Driven Design world does not talk about types of teams but Team Topologies does, we can combine both ideas. What kind of a team is it that is responsible for one or more Bounded Contexts? The suitable candidates are:

- Stream-aligned Teams
- Platform Teams
- Complicated Subsystem Teams

Note: The Enabling Team type from Team Topologies is not shown in this scenario because Enabling Teams do not have responsibility for business capabilities.

Another aspect with a great deal of potential for a combination of both approaches is the **X-as-a-Service** relationship of Team Topologies. In this scenario, 'one team consumes something that another team provides' (Quote from the *Team Topologies* book). The corresponding concepts in the Context Map are an Upstream/Downstream relationship between two teams with an Open-host Service on a Bounded Context providing/exposing functionality.

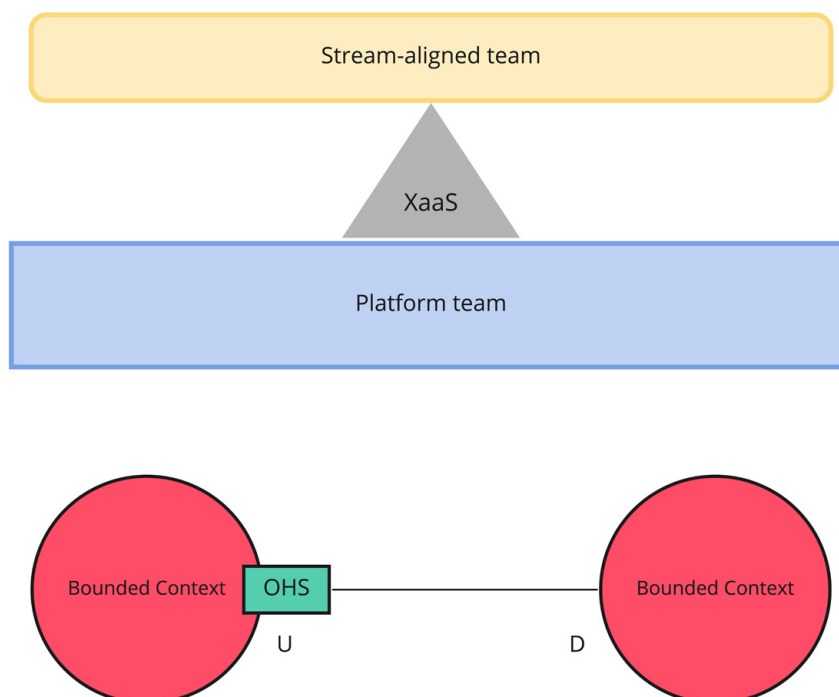


Figure 5: X-as-a-Service and Upstream/Downstream with an Open-host Service.

From this starting point lies a big potential for further deep dives into the relationship. In this case, you can get started with adding the team boundaries and team types to the Context Map. As a next step, we can dig deeper into this

Context Maps

X-as-a-Service relationship by using some of the Context Mapping patterns. We can visualize how the Domain Model provided by the Open-host Service propagates into the other Bounded Context by using the Anticorruption Layer and/or the Conformist pattern. This option allows you to take a closer look at how tightly or loosely two Bounded Contexts and even teams are coupled. Even if a team just consumes a service being provided, there is still a chance of a tight coupling on the downstream side if they conform to the model provided.

Another option that you can explore is whether or not the consuming side has or should have some influence on the providing side. I am aware that this is not the key intention of the X-as-a-Service relationship in Team Topologies, but there is something in between a very close collaboration between two teams and one team providing services in a 'take it or leave it' manner. This scenario can be described by the Customer-Supplier pattern: one team will typically have a certain, well-defined right to raise requirements against the other team.

Here are two examples. The image below depicts a Complicated Subsystem Team in the upstream which is responsible for multiple Bounded Contexts and provides a service to a Stream-aligned Team in the downstream which uses an Anti Corruption Layer:

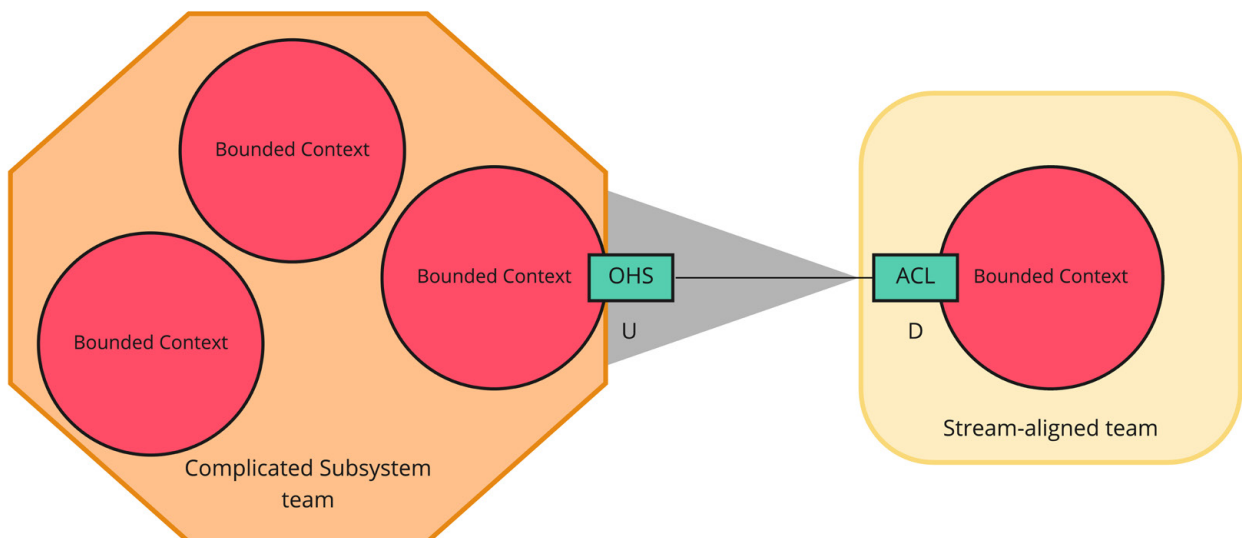


Figure 6: A relationship between a Complicated Subsystem Team and a Stream-aligned Team.

Context Maps

The next scenario depicts two teams in a X-as-a-Service relationship in which the downstream (consuming) side may be allowed to raise some requirements for the provider.

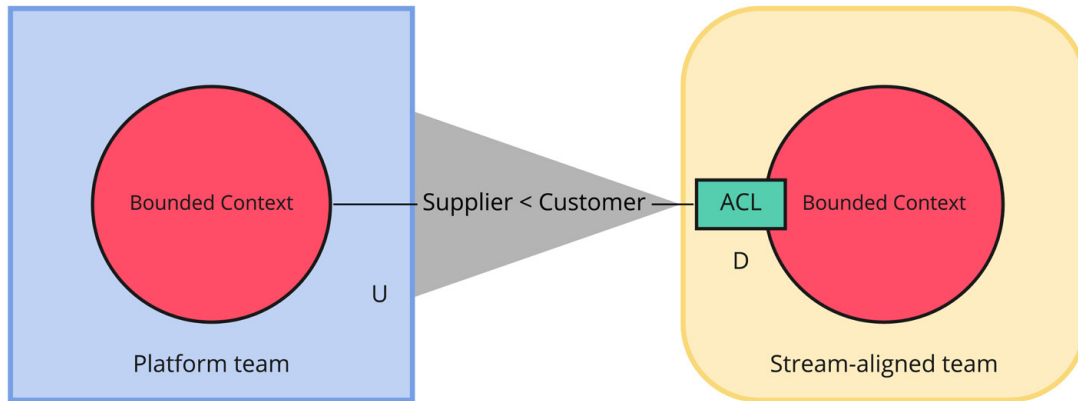


Figure 7: A Platform Team and a Stream-aligned Team in a X-as-a-Service relationship.

One more aspect that the team relationships of Team Topologies address rather indirectly is organizational solutions or manual steps. By organizational solutions or manual steps, I mean the lack of integration between two Bounded Contexts. Integration is often cumbersome and expensive to implement. Therefore, some teams avoid the effort and go for manual processes with minimal support from software. This is especially interesting when we aim to build Minimum Viable Products (MVPs). In this case, it is often viable that some back-office process steps do not get triggered through a perfectly implemented integration between the Bounded Contexts of two teams but rather through manual processes. Let's take a mortgage loan application as an example and think of two Bounded Contexts with corresponding teams: a loan application and a contract proposal. We don't expect many contracts in the first few months, so manually sending contract proposals may be a good idea for a minimum viable product.

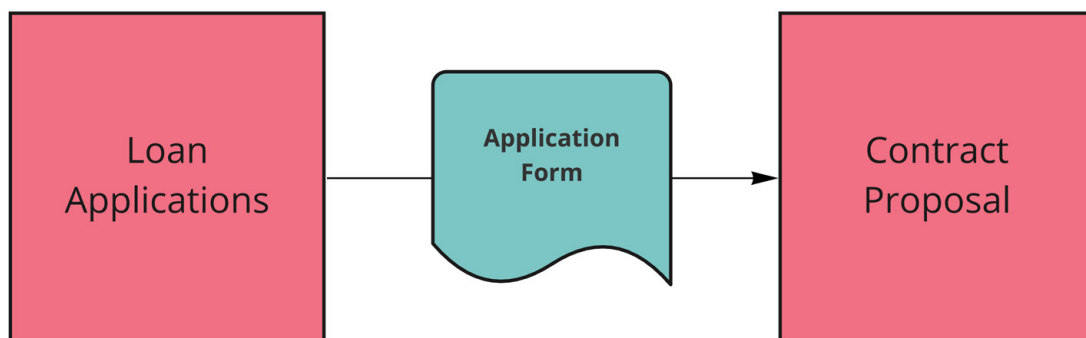


Figure 8: Two Bounded Contexts within a mortgage lending business that may give rise to a manual process (in the MVP state).

Context Maps

Which Team Topologies relationship would exist between those two teams? Still, X-as-a-Service, because there is still a (manual) service involved? I think it is better to make this relationship more explicit by adding the Separate Ways pattern to the observation. Yes, a service is being provided, but not with a perfectly implemented integration.

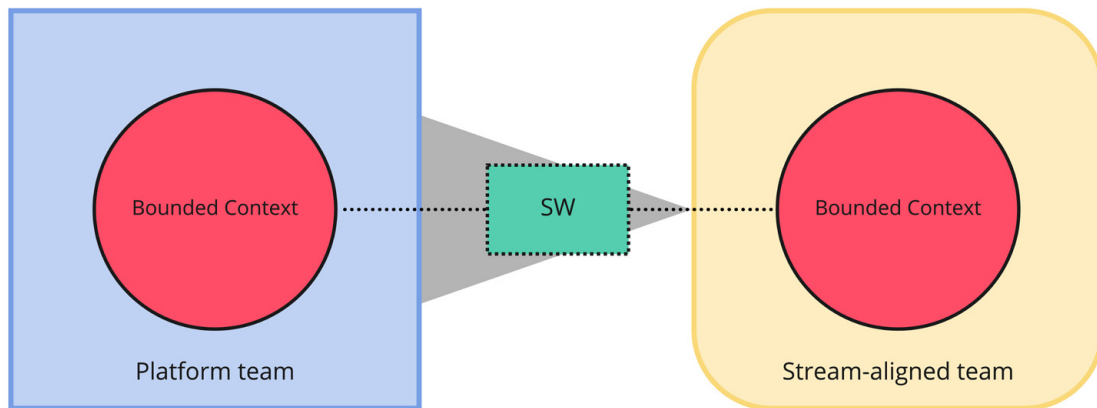


Figure 9: Two teams in a X-as-a-Service relationship but with a Separate Ways pattern.

Summary

Team Topologies and Context Maps have a certain degree of overlap in their intention. However, both can be combined in a variety of ways to unlock deeper insights into the relationships between teams and Bounded Contexts. As a starting point, I suggest beginning with Team Topologies and adding the Context Map patterns as needed. But please, don't overload your diagrams with everything you can squeeze into them, because they will become very hard to understand. Always keep the readers of your diagrams in mind: you are creating the diagrams for them, not for yourself. 😊

Further References:

Context Mapping at the DDD Crew on GitHub: github.com/ddd-crew/context-mapping

Visualizing Sociotechnical Architectures with Context Maps:

speakerdeck.com/mploed/visualizing-sociotechnical-architectures-with-context-maps

Alberto Brandolini: About Team Topologies and Context Maps:

blog.avanscoperta.it/2021/04/22/about-team-topologies-and-context-mapping/

About the author

Michael works as a Fellow for INNOQ in Germany. He has over 15 years of practical consulting experience in software development and architecture. His main areas of interest are currently Domain-Driven Design, Microservices, and Software Architectures in general. Michael is a regular speaker at national and international conferences.

Twitter: [@bitboss](https://twitter.com/bitboss)

LinkedIn: [michael-ploed](https://www.linkedin.com/in/michael-ploed)

Adopt

Accelerator Programme

Join our Accelerator Programme to speed adoption of Team Topologies with help from Team Topologies experts (TTVPs and authors)



Guided Workshops

Use our live online Guided Workshop sessions to accelerate adoption and increase practical awareness of Team Topologies:

- Blockers to Fast Flow
- Define and Evolve a Platform
- Team Topologies Applied
- and more...



teamtopologies.com/adopt

Architect Your Business with Domain-Driven Design and Team Topologies

Nick Tune, Principal Consultant at Empathy Software

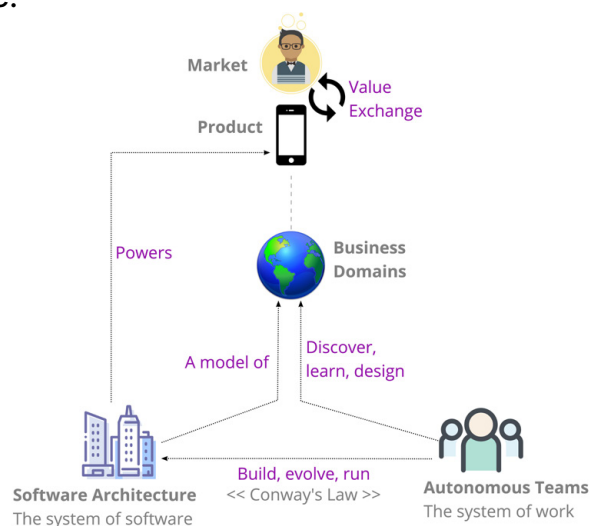
What does it take to become a high-performing technology-savvy organization? Is it sufficient to just have a revolutionary strategy, a few rockstar engineers, or cutting-edge tech? In my experience, it's extremely unlikely that being exceptional at any one thing and mediocre at everything else is going to result in high-performance. Excelling at many things, from strategy, to culture, to technology, is essential. I would go further and say that doing each of those things well is still not enough. It's essential to see the big picture and connect and co-design the dots of strategy, culture, and technology.



Why the combination of DDD and Team Topologies works so well

If connecting the dots is essential, then combining ideas from different communities and topics is also essential. That's why the combination of Domain-Driven Design (DDD) and Team Topologies is proving to be a successful formula. Both approaches address different aspects of becoming a high-performing organization. They are two pieces of the puzzle that fit together neatly. When you think about the architecture of a business, all of the pieces fall into place.

Figure 1: Key elements of a business architecture and their relationships.



Stream Boundaries

A company offers one or more products to one or more markets to create a **value exchange**. Value is created for customers and business value is received in return. Digital products are powered by software, which has an architecture. And the architecture is developed by engineering teams within the organization.

The relationship between the architecture of an organization and the architecture of the software is crucial, as **Conway's Law implies**. If a team owns a loosely-coupled part of the architecture, they can make changes quickly and improve the rate of product development. But if they own a part of the architecture that is highly-coupled to other parts, every change will require coordination with other teams and the rate of product development will be **orders of magnitude slower**.

So, how do we create loose coupling in software to enable more autonomous teams? The answer can be found by reflecting on what a software system is. A software system is a model of business concepts. In **Uber's software**, they have business concepts like trips and waypoints. The key to loose coupling is to find business concepts that change together and situate them in the same architectural component (like a microservice or monolith module). When changes normally occur within a single component, owned by a single team, there will be fewer technical and organizational dependencies.

Defining service boundaries, however, is a complex challenge that many organizations struggle with. Understanding business processes and concepts and modeling them as loosely-coupled architectural components is a little more work than just underlining nouns and verbs in a requirements document. This is where DDD aims to provide value, by being an approach to software development that treats domain discovery and modeling as a primary concern, with a particular emphasis on collaboration across disciplines.

EventStorming for fun and profit

One of the most DDD-esque activities is **EventStorming**, a collaborative domain discovery and modeling approach invented by Alberto Brandolini. EventStorming brings together domain experts, technology experts, and anybody else loosely involved in the software development process. Gathered around a large amount of wall space (minimum 8 meters), participants begin adding domain events, like Order Placed and Balance Transferred, for parts of the system with which they

Stream Boundaries

are familiar. Soon, everybody's domain knowledge is combined into one big

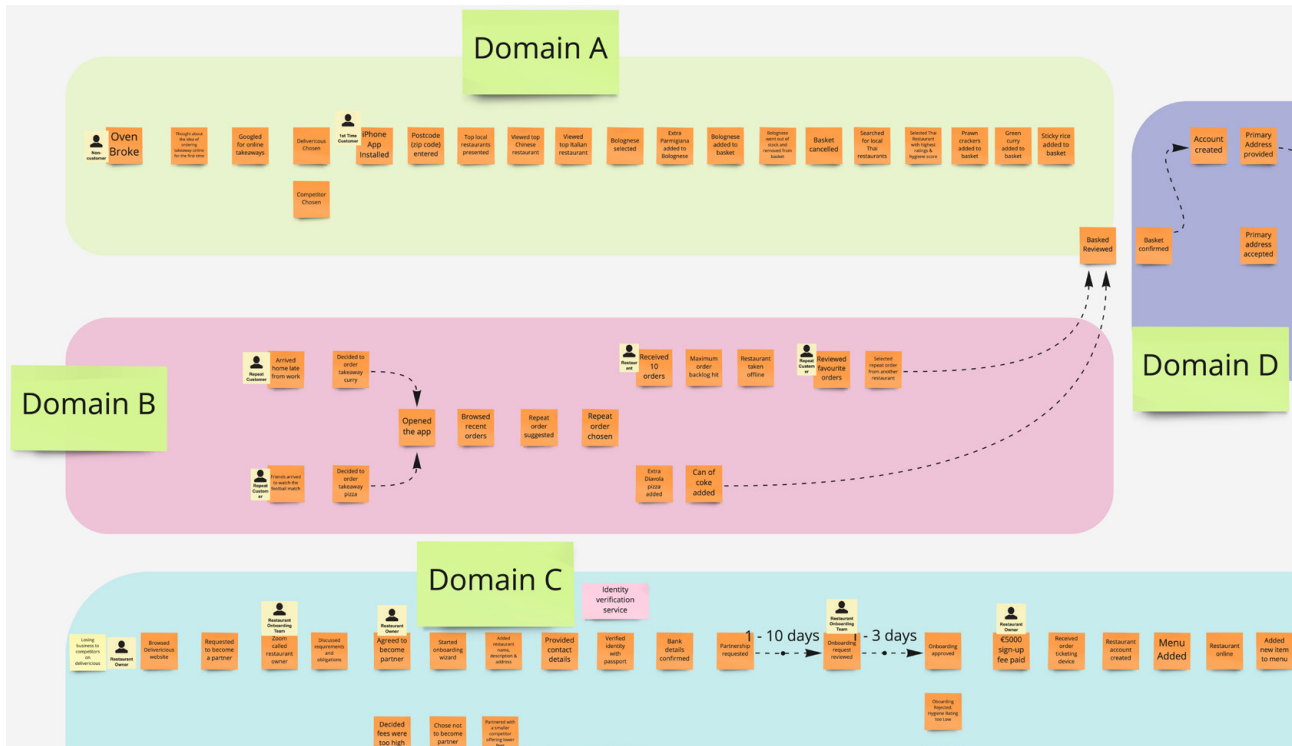


Figure 2. An EventStorming timeline representing end-to-end business processes.

Once formed, an event storm can be sliced up into chunks.¹ Each chunk contains a selection of business concepts that appear close together in the business process, indicating that they are likely to change together. If you remember, this was the key ingredient for creating a loosely-coupled system. So, the chunks on an EventStorm, referred to as domains or subdomains, become the candidate boundaries for the software architecture and teams. In any system, there will always be some coupling and co-change.

Some further benefits of EventStorming are that it also improves cross-team collaboration and visibility so that when multiple teams must work together on a new feature, they're better equipped to do this. Another benefit of EventStorming is that teams learn more about the domain and become more empowered to shape the products they are building. This is important because whole-team collaboration has been **identified** as a top source of product **innovation**.

¹ Keep in mind that not all domains will appear as neatly-shaped rectangles. Some domains will contain events appearing in different parts of an EventStorm, for example.

Stream Boundaries

Validating domain boundaries with Domain Message Flow Modeling

After identifying domain boundaries on an EventStorm, another DDD technique can help you validate and refine the boundaries and identify the responsibilities of each domain. **Domain Message Flow Modeling** is a technique for visualizing the communication between different domains in end-to-end flows such as placing an order. A Domain Message Flow uses commands, events, and queries to represent collaboration in the domain. This notation maps directly onto a software architecture, which helps to prove that the logical design will work in reality. This technique is also great for uncovering hidden coupling at an early stage.

Scenario: Placing an Online Order When All Items in Stock

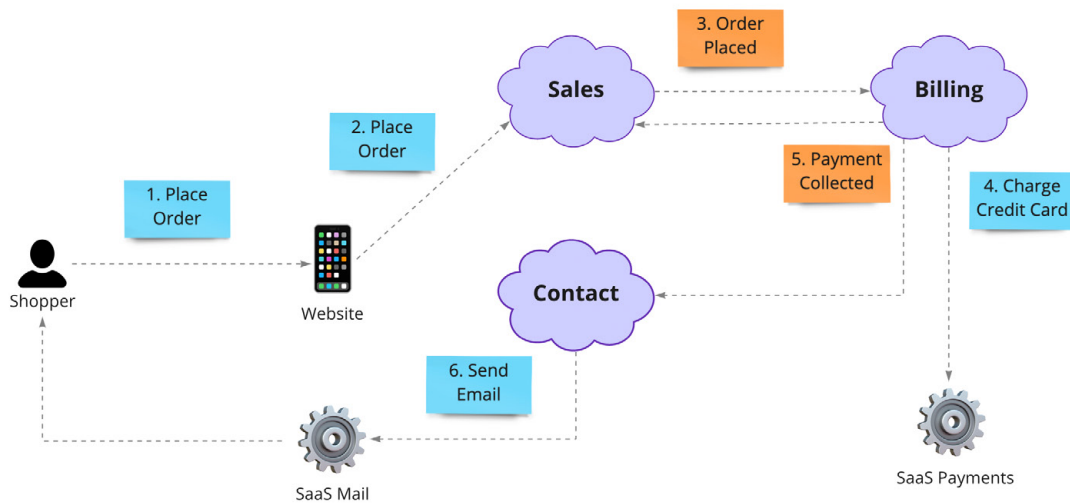


Figure 3: A simple Domain Message Flow Model example.

After identifying and refining domain boundaries, Team Topologies enters the picture and the focus shifts. The next step is to verify that the domain boundaries will map onto an organizational structure that enables a fast flow of changes. The first thing to consider is cognitive load: will each domain be manageable for a long-lived team of 5-9 people? If it seems too big or complex, it will need to be split up.

Stream Boundaries

Assessing cognitive load with the Bounded Context Canvas

The **Bounded Context Canvas** is a tool created by the DDD community which maps out the purpose, responsibilities, collaborators, and complexity of each service. This canvas can help you to see if a domain is too big or complex for a single team.

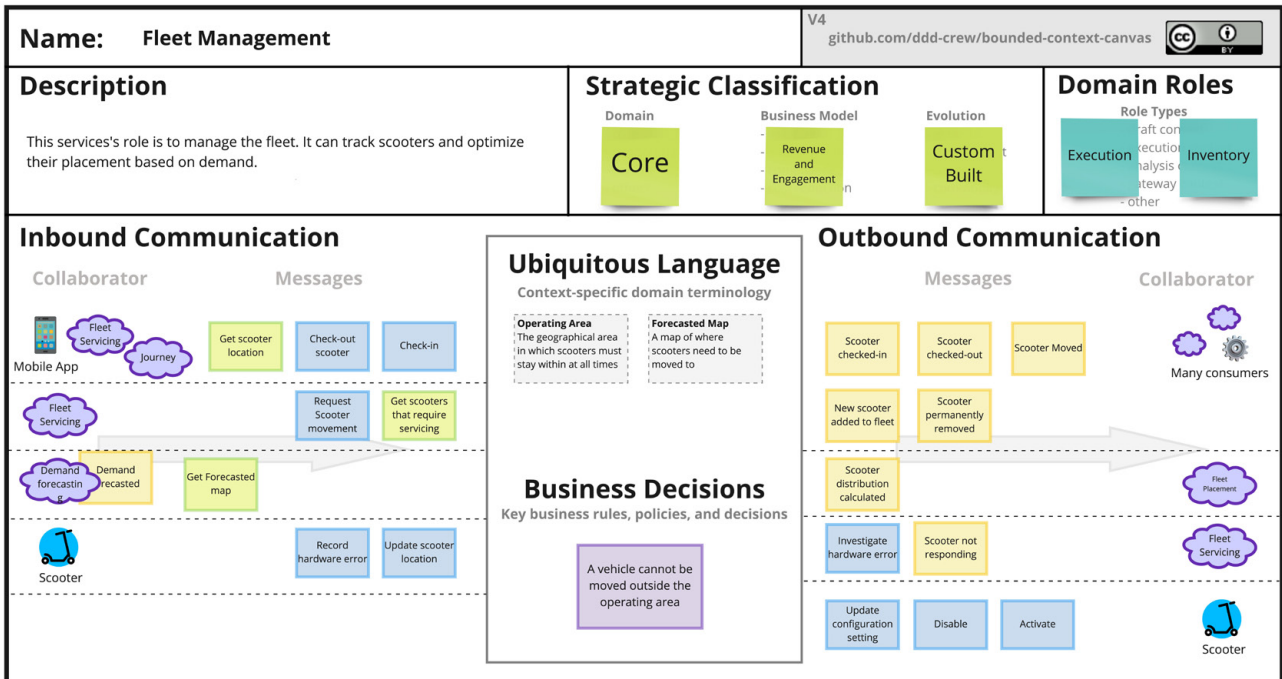


Figure 4: This Bounded Context Canvas shows a service with too many responsibilities, causing high cognitive load.

After validating and refining your domain boundaries, you can build a team topology around them. Each domain will be owned by a Stream-aligned, Complicated Subsystem, or Platform Team and the interactions between them will need to be carefully considered.

The design process may sound a bit waterfall, but it should be a continuous and evolutionary process. Both the team boundaries and domain boundaries may evolve and even diverge over time as knowledge is gained and the business context changes. All of the techniques mentioned in this article should be part of your toolbox and used as regularly as needed, not just at the beginning of an initiative.

Stream Boundaries

Core Domain Charts provide a strategic reference point

One other technique that often provides value is the [Core Domain Chart](#), another DDD community initiative. This tool helps you to visualize the strategic importance of each domain and ensure that your domain and organizational boundaries are optimized for maximum exploitation in core domains. Core domains are those that form the core of the business strategy.

By laying out each domain on a Core Domain Chart and overlaying the Team Topology, it becomes easier to identify mismatches which are likely to have strategic consequences. For example, when a Stream-aligned core domain team is collaborating with three Stream-aligned supporting domain teams, it's a warning sign that a core domain team has an excessive cognitive load and is being slowed down by too much collaboration. Or, when multiple core domains are all trying to evolve in different directions but they all depend on a Stream-aligned supporting domain team that is becoming a bottleneck. These problems are of high strategic importance because innovation in core domains is being stifled, and the core domains are where competitive advantage is gained and maintained.

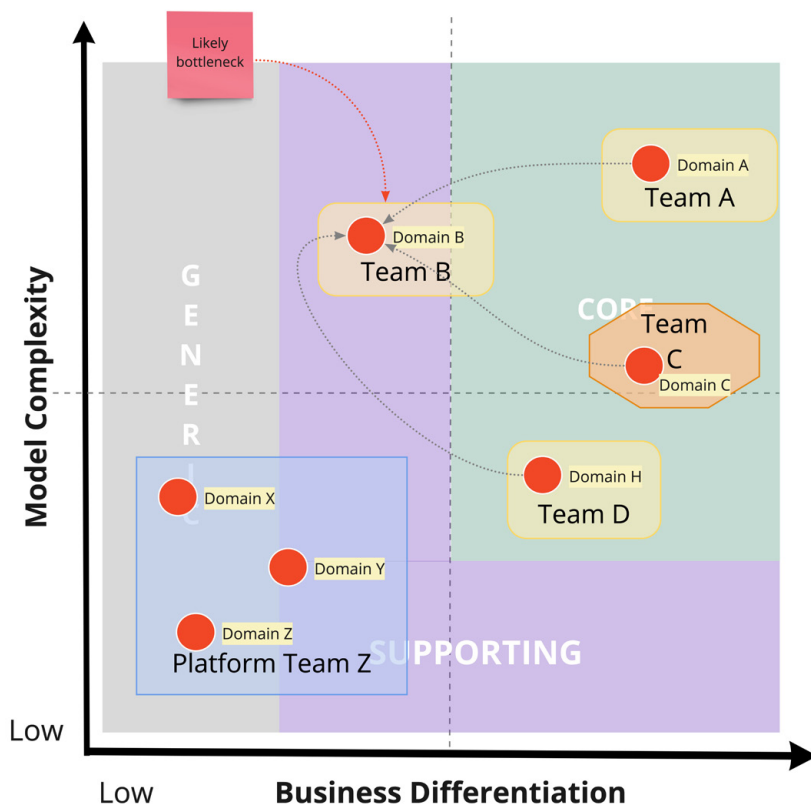


Figure 5: This core domain chart shows a bottleneck which may impact three Stream-aligned core domain teams.

Next Steps with DDD and Team Topologies

A high-performing organization excels in many areas. When used together, Domain-Driven Design and Team Topologies provide key insights and techniques for architecting your business if you understand their strengths and apply them appropriately.

If you're interested in learning more about Domain-Driven Design, check out [Welcome to DDD](#), and the [DDD Starter Modeling Process](#). These are completely free community initiatives created to help newcomers learn and apply DDD with a hands-on focus. You may also want to check out my free [Strategic DDD Kata](#) which provides an example Eventstorm that you can slice up into domains as well as practice [Domain Message Flow Modeling](#) and [Core Domain Charts](#).

About the author

Nick is the author/co-author of 3 books on architecture, organization design, and DDD: *Architecture Modernization: Product, Domain, and Team-Oriented*; *Patterns, Principles, and Practices of Domain-Driven Design*; and *Designing Autonomous Teams and Services*. Nick works with CTOs and technology leaders to define strategy, shape architecture, and build high-performing continuous delivery teams.

Mastodon: [@ntcoding](#)

LinkedIn: [nick-tune](#)

Transform

Enhance your transformation program with ongoing expertise

Our expert practitioners provide advice, insights, and guidance over 6-18 months as you define and scale your transformation program.



Regular workshops and insights

Increase confidence in the success of your transformation program via regular sessions from TT experts:

- **Workshops**
- **Q&A and talks**
- **Expert Insight sessions**
- **and more...**



teamtopologies.com/transform

Finding Good Stream Boundaries with Independent Service Heuristics and User Needs Mapping

Rich Allen and Matthew Skelton

When designing organizations for a fast flow of change, we need to find effective boundaries between different streams of change to ensure that we create **good** team boundaries. This can be achieved by identifying potential boundaries across services, domains, applications, or streams. This article considers different ways that you could approach boundary exploration.

Identifying boundaries with Domain-Driven Design

When we first think of the terms 'domain' or 'boundary' in a software context, it is likely that our first thoughts are of Domain-Driven Design (DDD). The book by Eric Evans, [Domain-Driven Design: Tackling Complexity in the Heart of Software](#), published in 2003, has stood the test of time and provides significant insights into how to structure software that can be aligned with existing business domains. The high-level definition of the practice is 'an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of core business concepts'. With the introduction of terms like 'Bounded Contexts' and 'ubiquitous language,' it provides a vast library of practices and techniques to help practitioners tame the complexities of modern software development. Since the original publication, there have been numerous others that have attempted to simplify and make



Stream Boundaries

the concepts more digestible, for example, [Domain-Driven Design Distilled](#) by Vaughn Vernon and [The Anatomy of Domain-Driven Design](#) by Scott Millet and Sam Knight.

There have also been many contributions from the wider DDD community, including new techniques such as EventStorming. The [DDD Crew](#) have a great set of resources available which includes a DDD Starter Modeling Process, Core Domain Chart examples, Context Map Cheat Sheets, an EventStorming Glossary Cheat Sheet, a Bounded Context Canvas, and many more. The value and benefits provided by a DDD approach are clear and taking the time to learn each of the techniques will be a sound long-term investment. However, many people struggle to get started adopting the practices as they can often be seen as overwhelming.

Taking a different approach: Independent Service Heuristics

Independent Service Heuristics (ISH) is a technique invented by the authors of [Team Topologies](#), Matthew Skelton and Manuel Pais. It has been subsequently refined by others, including [Team Topologies Valued Practitioners \(TTVPs\)](#) and members of the wider DDD community. You can find more information via the [Independent Service Heuristics GitHub repository](#) which is openly provided via the CC BY-SA license.

ISH is an intermediate approach that can help to introduce the principles of DDD without some of the abstract terminology that can often be a barrier to the adoption of DDD.

ISH provides simple rules-of-thumb or clues that can be used to identify candidate value streams and domain boundaries by seeing if they could be run as a separate SaaS/cloud product. It is intended to stimulate conversation and provide a frame of thinking about basic domain concepts. It does not attempt to be a perfect 'catch-all' tool. After using ISH to identify potential domain boundaries or value streams, it often makes sense to then delve deeper into the problem space using other DDD techniques.

Exploring boundaries using Independent Service Heuristics

Independent Service Heuristics (ISH) starts with a simple question, 'Could this thing be run as a cloud-hosted (SaaS) service or product?' On the surface, this almost seems too simple. How can that one question provide answers that help us to uncover potential domain boundaries and value streams? The terms Cloud and Software as a Service (SaaS) have been in the public domain for long enough that most people will understand what we mean when we ask the question. And the answer to the question is often either yes, no, or maybe. We can then follow up with a series of clarifying questions to determine whether the area under focus could truly be a potential domain boundary.

Choosing an area of focus

In any process or methodology, getting started and taking the first step is normally the hardest part. In the case of ISH, that first step is deciding where to focus your attention. Essentially, we just need to choose an area of the business that needs to be represented in software. This could be a user journey, a 'product', a possible business domain, a software service, an entire software application, a set of tasks for a single user persona, a possible value stream, etc.

The important thing here is that we actively choose an area and get started. The process is quick enough that we won't waste too much time if we happen to choose an area that does not naturally fit a domain boundary but at least we can discount it and move on to the next candidate.

An Independent Service Heuristics example

Imagine a fictional company called Footprints Tours which offers 'alternative' walking tours of cities exploring their social and cultural history. They provide both guided and self-guided tours and have implemented a monolith website and mobile application to serve all of their customer needs. The flow of development has slowed down significantly as the code base has grown over time. Using Independent Service Heuristics, they are looking to understand how they might re-organize the teams and therefore the applications/services to improve flow and alignment with the needs of their customer. The first step is to capture some possible fracture planes, such as those shown in the image below.

Stream Boundaries

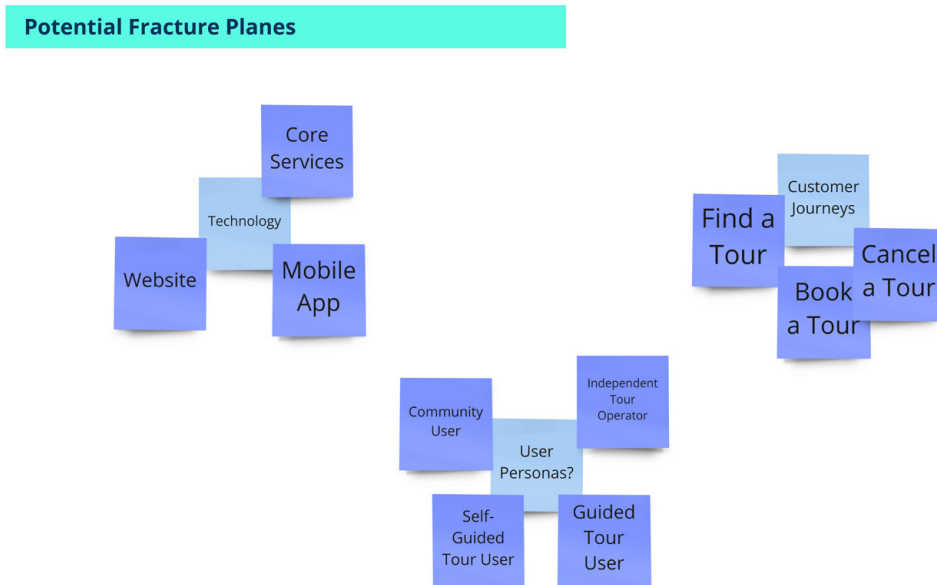


Figure 1: An example of potential fracture planes for Footprints taken from one of our Guided Discovery Workshops.

Uncovering potential domain or service boundaries

Once a candidate domain, service, application, or value stream has been identified, the next step is to go through a series of questions to identify whether or not we have found a good candidate for being a separate stream of change. The high-level checklist of questions is as follows (as of January 2023):

1. Sense-check: Could it make any logical sense to offer this thing 'as a service'?

- Is this thing sufficiently independent?
- Would consumers understand or value it?
- Would it simplify execution?

2. Brand: Could you imagine this thing branded as a public cloud service (like AvocadoOnline.com 🖱️)?

- Would it be a viable business (or 'micro-business') or service?
- Would it be a compelling offering?
- Could a marketing campaign be convincing?

Stream Boundaries

3. Revenue/Customers: Could this thing be managed as a viable cloud service in terms of revenue and customers?

- Would it be a viable business (or 'micro-business') or service?
- What would a subscription payment include?
- Is there a clearly defined customer base or segment?

4. Cost tracking: Could the organization currently track costs and investment in this thing separately from similar things?

- Are the full costs of running this thing transparent or possible to discover? Consider infrastructure, data storage, data transfer, license costs, etc.
- Is the thing too interconnected with other things in the organization? Or fairly separate?
- Does the organization track this separately?

5. Data: Is it possible to clearly define the input data (from other sources) that this thing needs?

- Is it dependent on lots of data from multiple sources? Or fairly independent?
- Are the sources internal (under our control) or external?
- Is the input data clean or messy?
- Is the input data provided in a self-service way? Can the team consume the input data 'as a service'?

6. User Personas: Could this thing have a small/well-defined set of user types or customers (user personas)?

- Is the thing meeting specific user needs?
- Do we know (or can we easily articulate) these user types and their needs?

7. Teams: Could a team or set of teams effectively build and operate a service based on this thing?

- Would the cognitive load (breadth of topics/context switching) be bounded to help the team focus and succeed?
- Would significant infrastructure or other platform abstractions be needed?

Stream Boundaries

8. Dependencies: Would this team be able to act independently of other teams for the majority of the time, to achieve their objectives?

- Is this thing logically independent from other things?
- Could the team 'self-serve' dependencies in a non-blocking manner from a platform?

9. Impact/Value: Would the scope of this thing provide a team with an impactful and engaging challenge?

- Is the scope big enough to provide an impact? Would the scope be engaging for talented people?
- Is there sufficient value to customers and the organization that the value would be clearly recognized?

10. Product Decisions: Would the team working on this thing be able to 'own' their product roadmap and the product direction?

- Does this thing provide discrete value in a well-defined sphere of execution?
- Can the team define their own roadmap based on what they discover is best for the product and its users or is the team always driven by the requirements and priorities of other teams?

Answer these questions for each of the candidate streams you have identified. The more 'yes' or 'maybe' answers a possible stream has, the greater the chance that you have found a good candidate for being a separate stream of change.¹

¹ N.B. The questions on dependencies, impact/value, and product decisions were added to ISH in December 2021 as a result of working closely with the team at [Zalora](#) after some of our workshops. See below for more details on how Zalora used the ISH approach.

Stream Boundaries

Independent Service Heuristics - Answers											Yes	Maybe	No	Fracture plane
Stream candidate	1	2	3	4	5	6	7	8	9	10	Language	Fracture Planes		
Self-Guided Tour User	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	User Personas		
Guided Tour User	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	User Personas		
Find a Tour	Yes	Maybe	Yes	No	Maybe	Yes	Yes	Yes	Yes	No	Yes	User Journey		
Book a Tour	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Maybe	Yes	Yes	User Journey		

Figure 2: Sample ISH answers taken from one of our Guided Discovery Workshops.

The highly visual style of the ISH exploration grid helps to facilitate discussions among stakeholders from different parts of the organization.

Delving Deeper

Answering these initial questions about the service should help to uncover potential candidates for separate streams of change, but it may be useful to consider other aspects too, such as whether the language used to describe services is the same or different and where services are currently tightly-coupled.

Case Study - using the Independent Service Heuristics at Zalora

ZALORA is the leading fashion & lifestyle destination for Southeast Asia, carrying an ever-expanding line-up of local and international brands. In 2021, as it scaled by more than 60%, it was on a drive to enhance customer experience and reduce time-to-market. As part of this, Zalora began exploring the use of Team Topologies ideas and practices.

During a multi-day workshop with a wide range of attendees (led by Team Topologies co-author Matthew Skelton), Zalora stakeholders were introduced to the Independent Service Heuristics. They learned how to use the ISH approach to find candidate flow-aligned boundaries and interpret the results, allowing the insights to guide and inform conversations with different parts of the business.

Stream Boundaries

After the workshop, Zalora ran further internal sessions, expanding the use of ISH to find further possible flow-aligned boundaries. The straightforward language of ISH helped to facilitate conversations between many different parts of Zalora, including: technology, product, warehouse, logistics, and business strategy. The Independent Service Heuristics acted as a frame or 'lens' through which to talk about priorities, flow, and ownership. The very visual style of the ISH exploration grid provided a way to frame conversations:



Figure 3: Screenshot of the results of an expanded Independent Service Heuristics discovery and discussion at Zalora.

Stream Boundaries

Zalora used the ISH questions so extensively that they contributed three new heuristics to the ISH collection: Dependencies, Impact/Value, and Product Decisions.

"We first used the Independent Service Heuristics as part of Team Topologies during our workshop with Matthew Skelton in August 2021. The framework and shared language of the ISH approach were transformational to the discussions we later had about our organization and team structure. Not only did this approach help us discover and align on new stream-aligned Tteams, but it also helped us redefine other teams as Platform, Complicated Subsystem, and Enabling Teams. Thanks to the Team Topologies and ISH framework, our team structure is more autonomous, meaningful, and productive.



We were able to take this simple but powerful framework in its visual, grid-based format and have further discussions which led to us expanding the scope to reflect additional angles that we knew would be essential for sustainable team boundaries. It was great to be able to contribute our updates to the ISH code repo so others can make use of our insights!"

— *Liam Hutchinson, Group Director of Product, Zalora*

Exploring boundaries from a user perspective with User Needs Mapping

The ISH approach looks at existing services, applications, or value streams to determine whether they might form good boundaries for teams. However, a slightly different perspective can be provided by User Needs Mapping.

The term User Needs Mapping was coined by [Rich Allen, a TTVP](#), during the preparation of some workshops focused on [Team Topologies](#). It is based on one of the early stages of the [Wardley Mapping](#) process. Wardley Mapping builds upon ancient principles taken from Sun Tzu's *The Art of War*. It provides a great

Stream Boundaries

way for business leaders to map out a strategy by taking into account several factors including purpose, landscape, climate, doctrine, and leadership in a continuous cycle of observing, orienting, deciding, and acting (the OODA loop from John Boyd).

One of the core principles (and potentially the most intimidating part) of Wardley Mapping is the use of an evolutionary axis that runs from Genesis on the left through Custom Built, Product, and finally Commodity on the right of the map. Items are plotted on the map with respect to how 'evolved' the item is. Something new to the world would be added to the Genesis column, whereas something widely available and undifferentiated would be plotted in the Commodity column. This allows the mapper and colleagues taking part in the mapping session to begin strategic conversations about the current state of items on the map and also discuss how they may evolve in the future (based on market trends etc). This means they can make more informed, strategic decisions about how to plan for the future.

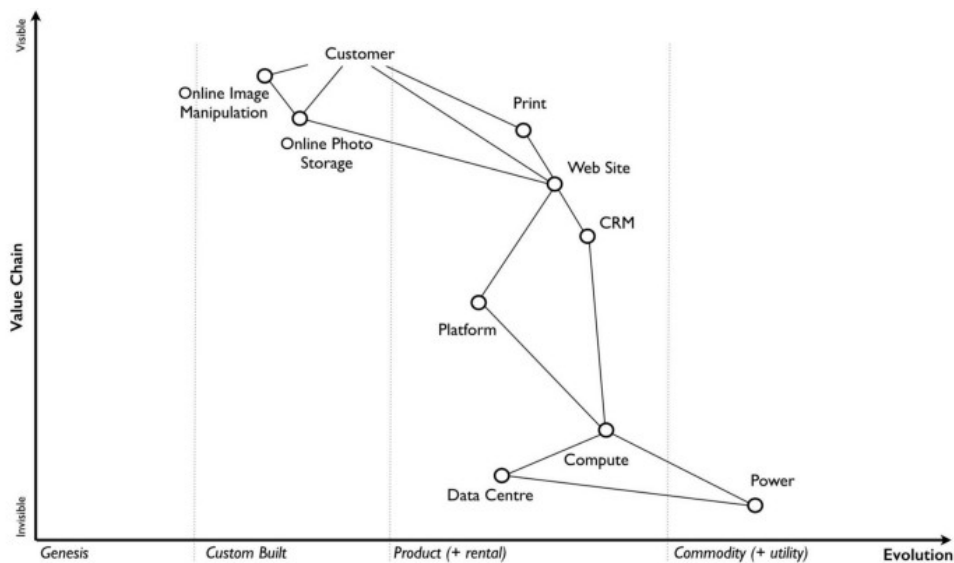


Figure 4: A completed Wardley Map showing the evolutionary axis.

The power of Wardley Mapping lies within this unique ability to capture 'movement' or evolution over time. Changing the position of an item changes its meaning on the map and puts the focus of the conversation onto the map. However, in much the same way as DDD, people can find the practice intimidating in the first instance, since there is so much to take in. In the same way that ISH provides a lightweight alternative to introduce DDD concepts, User

Stream Boundaries

Needs Mapping provides a lightweight entry into the world of Wardley Maps. The Wardley Mapping process consists of 5 steps:

1. **Define Your 'True North'** (i.e. our customer/user).
2. **User's Needs** – Needs to be met.
3. **Capabilities** – How you're going to meet your user's needs.
4. **Value Chain** – A list of users, needs, and capabilities becomes a value chain when you add dependency relationships.
5. **Wardley Map** – A value chain becomes a Wardley Map when you determine how evolved everything is and position it accordingly (left-to-right) on the evolutionary axis.

The term User Needs Mapping attempts to capture the first 4 steps of the Wardley Mapping process as we believe it can provide initial value for identifying potential team boundary issues without progressing into step 5 and the evolutionary world of Wardley Maps.

Mapping User Needs to explore boundaries

User Needs Mapping begins by simply asking the question 'Who are your users/customers?'. It is still surprising how many people are unable to concisely answer that seemingly simple question. Many people might know who their users are but haven't documented it or shared it with anyone. User Needs Mapping provides a simple canvas to begin the process and starts by capturing the user and their needs.

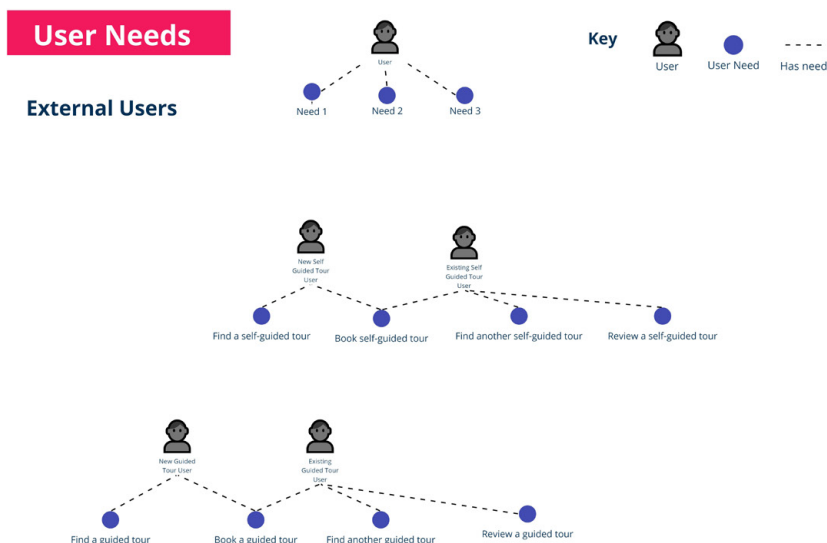


Figure 5: Capturing users and user needs in a simple visual way.

Stream Boundaries

After capturing some user needs, the next phase is mapping the dependency chain. This phase essentially uses the vertical 'value chain' axis of a Wardley Map without the evolutionary horizontal axis. The map is 'anchored' by the user at the top of the canvas and the user's needs are linked to each user. Focusing on one need at a time, we plot what services, dependencies, or business capabilities are used to meet that particular need. The vertical tour axis represents how visible the capability is to the user.

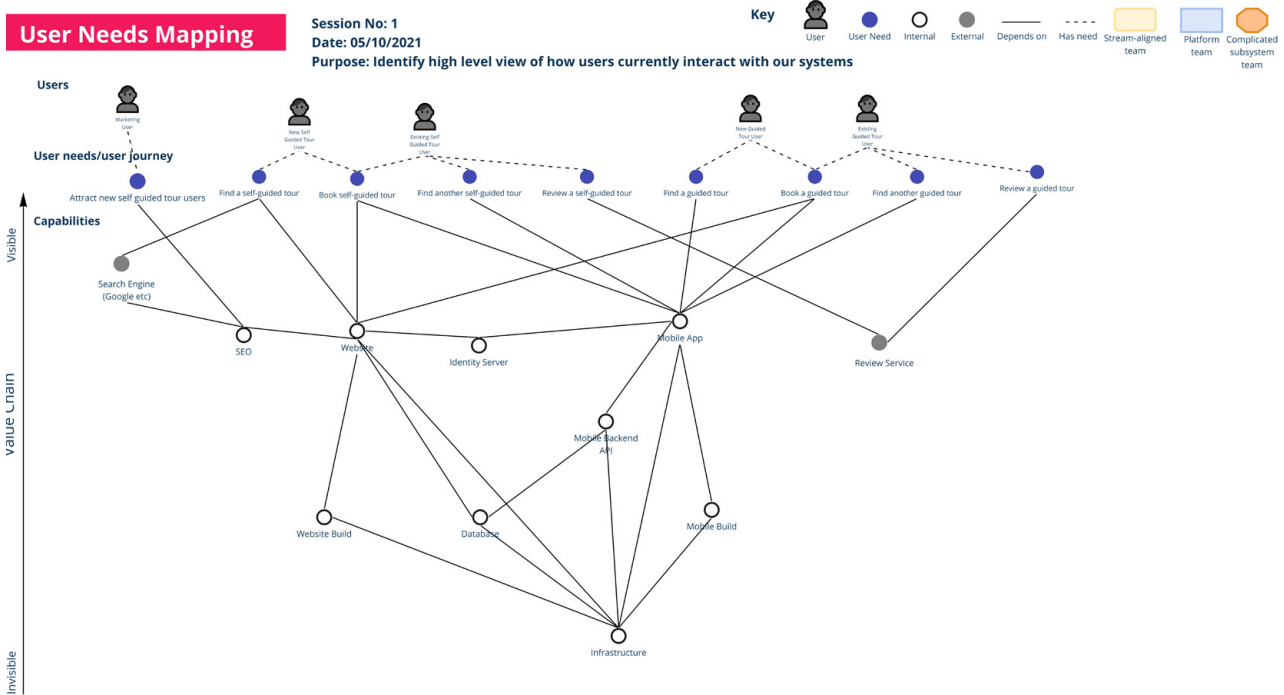


Figure 6: An example early stage User Needs Map highlighting potential team boundaries.

A User Needs Mapping example

The same fictional company Footprints Tours introduced in the previous section is now looking to use User Needs Mapping to understand how it might re-organize the teams to improve flow and alignment with the needs of their customer.

We could begin the exploration with a user journey such as 'Finding a self-guided tour'. In this scenario, we can imagine what would be required when a user needed to find a good self-guided tour. The first service they might use is a search engine such as Google or Bing (an external service). For our 'Find a tour' service to appear in the search engine results, we would need some Search Engine Optimization (SEO) and this would lead the user to that page on our website. Once the user has landed on the 'Find a tour' page, they might be

Stream Boundaries

encouraged to use the Tour Search service (an internal service dependency) to find a tour that looks good for them. The Tour Search service might require a database that contains data about the tours. The database is provided as a Platform as a Service offering from a cloud provider which then becomes a further external dependency that can be mapped on the canvas. As each dependency becomes less visible to the end user, it is plotted further down the vertical axis.

After performing an initial mapping process, we can explore overlaying the Team Topologies team types to highlight where we think some possible team boundaries might exist.

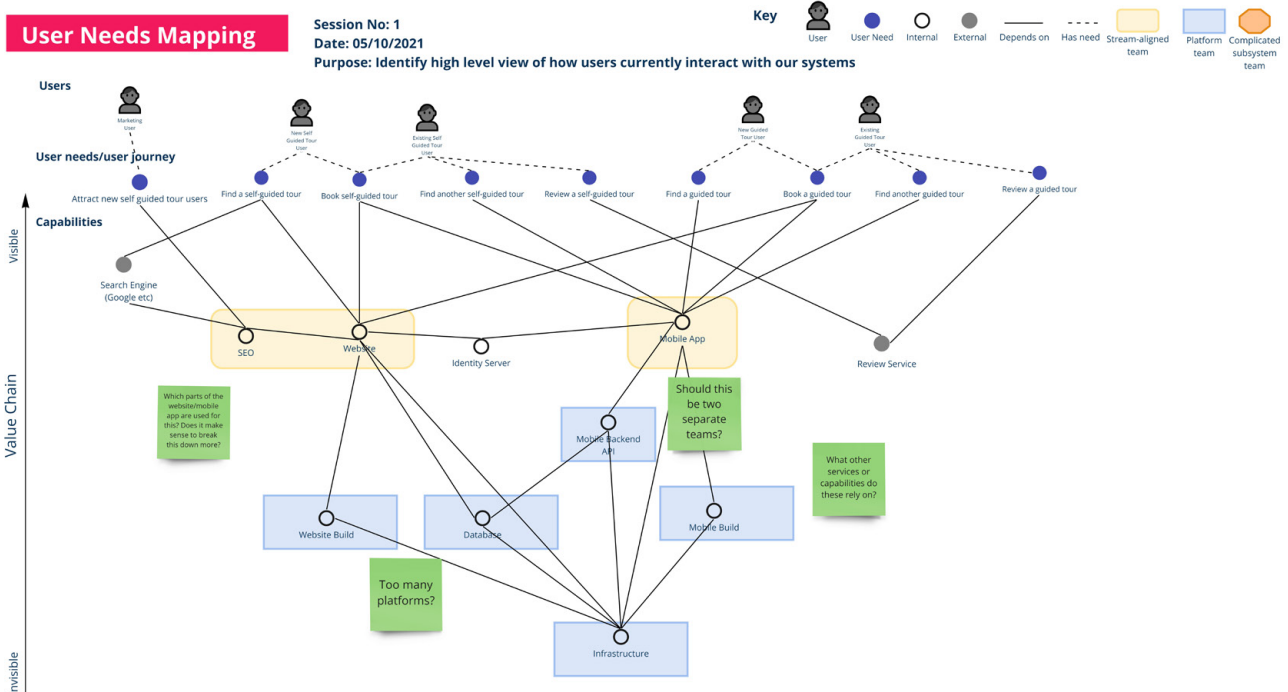


Figure 7: A User Needs Map after overlaying some initial Team Topologies team shapes.

After this initial session and seeking feedback, we might decide to 'drill in' to some areas such as the website to identify which parts of that system might be owned by specific teams and therefore be a good candidate for stream-alignment.

Stream Boundaries

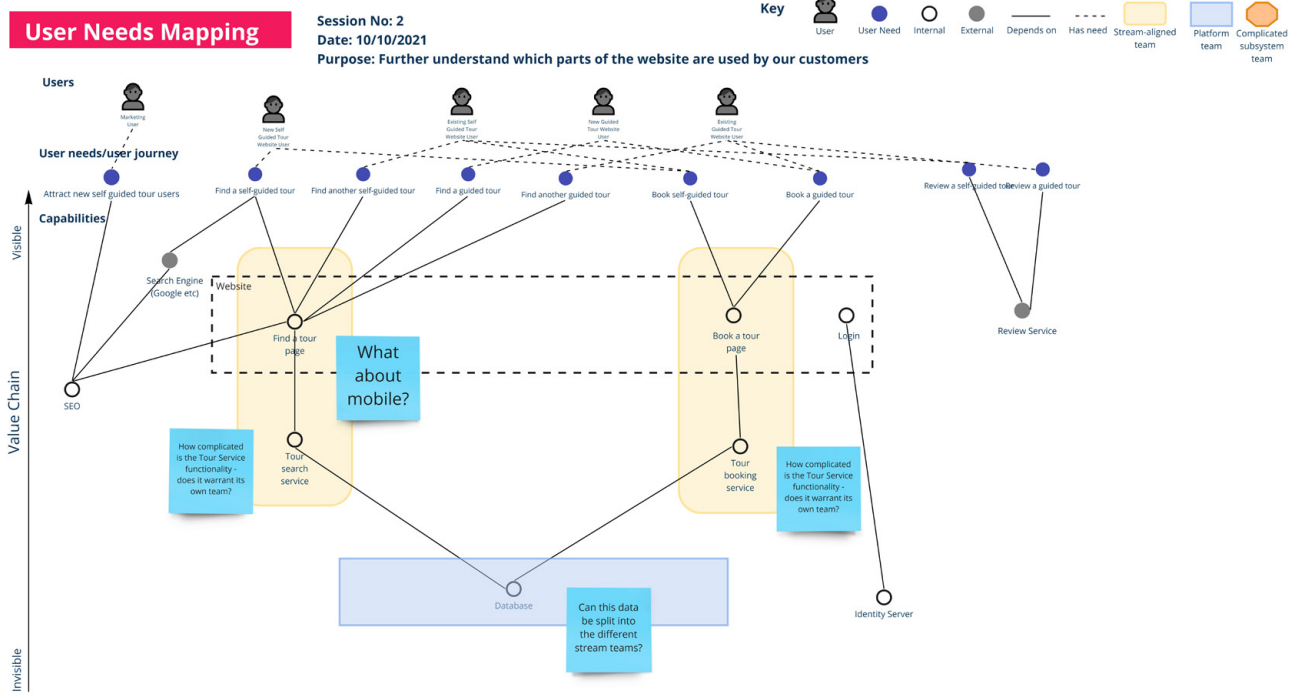


Figure 8: The User Needs Map has evolved after identifying potential opportunities for Stream-aligned and Platform Teams

As we can see, the database is potentially a shared dependency between the two stream-aligned teams. This raises questions such as whether data should be stored in a single database. Should the data be owned by a database platform team? Is there data that is only relevant to the individual streams? Could this database be split into two databases provided by a platform but owned by the streams?

After we have drilled down the dependency chain as far as we want to go, we then look at the next user need and repeat the process. As we do this, we begin to uncover and visualize the dependencies between the services and capabilities within our organization. The more we do this, the more we might spot patterns or opportunities to decouple services to provide faster flows of change or introduce other types of teams to help reduce the cognitive load of stream-aligned teams.

Stream Boundaries

The User Needs Mapping Process

In summary, the User Needs Mapping process is as follows:

1. Create a list of customer/user types.
2. Identify user needs (for each type of user).
3. Identify what capabilities/components/services are required to meet each user need.
4. Overlay potential team boundaries using the Team Topologies shapes.
5. Annotate the map with questions about suspect dependencies.
6. Discuss how the dependencies might be broken and capture your thoughts on other ways to organize the dependencies.
7. Repeat steps 1 to 5 as necessary until you identify potential team boundaries that 'feel' right.

Delving Deeper

After you have completed the User Needs Mapping process, the next logical step may be to introduce the horizontal evolutionary axis of Wardley Mapping. It can often be an interesting thought experiment to consider whether products or services you are currently 'custom building' with specific teams should actually be purchased as a 'product' or even used as a 'commodity'. Or maybe the products and services you are building now will evolve within the next couple of years? This might prompt the question of whether to start preparing for the inevitable evolution now.

Summary

In this article, we looked at how we could use two approaches, Independent Service Heuristics and User Needs Mapping, as a lightweight introduction/alternative to DDD concepts and to explore application and service boundaries that could lead to good stream and team boundaries. With the goal of achieving a fast flow of change, taking a team-first approach and understanding how those teams interact is of utmost importance. Why not give Independent Service Heuristics and User Needs Mapping a try the next time you need to identify boundaries within your organization?

About the authors

Rich Allen is Head of Consulting at Conjur Solutions and is a Team Topologies Valued Practitioner (TTVP) helping to introduce organizations to Team Topologies patterns and principles through talks and guided workshops. Rich has been developing software and helping organizations to implement lean and agile ways of working for over two decades.

Twitter: [@rich_allen](#)

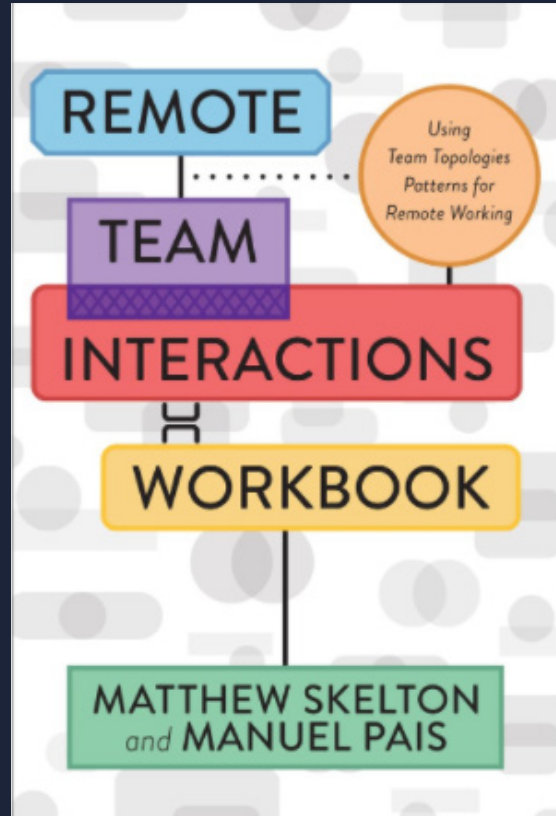
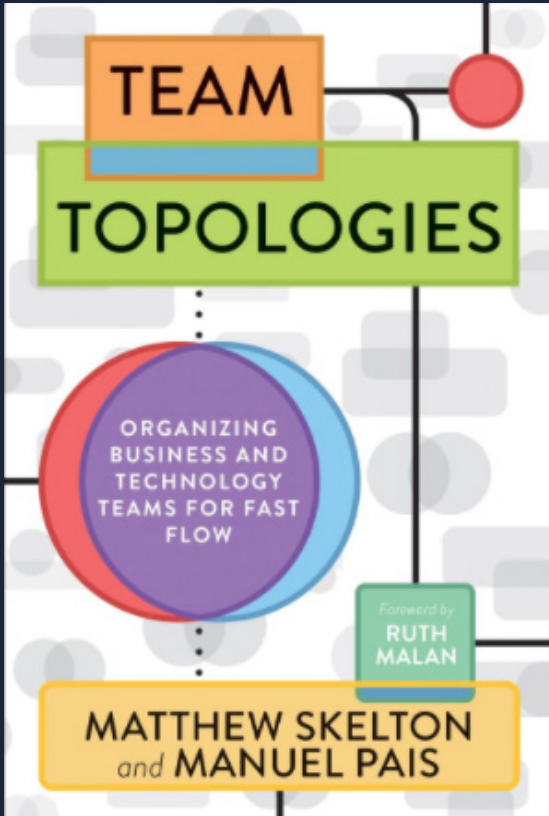
LinkedIn: [richardallen](#)

Matthew Skelton is co-author of Team Topologies: organizing business and technology teams for fast flow. Head of Consulting at Conflux, he specializes in Continuous Delivery, operability, and organization dynamics for software in manufacturing, ecommerce, and online services, including cloud, IoT, and embedded software.

Mastodon: [@matthewskelton@mastodon.social](#)

LinkedIn: [matthewskelton](#)

Read the books



teamtopologies.com/books



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.



Mastodon: @matthewskelton@mastodon.social
LinkedIn: [linkedin.com/in/matthewskelton/](https://www.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](https://twitter.com/manupaisable) | LinkedIn: [linkedin.com/in/manuelpais/](https://www.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 © 2017-2023 Team Topologies Ltd. All Rights Reserved.

Registered office: West One, 114 Wellington Street, Leeds, LS1 1BA, UK

Registered in England and Wales, number 13684580. VAT registration number GB393377361