TEAM TOPOLOGIES

ORGANIZING BUSINESS AND TECHNOLOGY TEAMS FOR FAST FLOW

Foreword by RUTH MALAN

MATTHEW SKELTON *and* MANUEL PAIS

Praise for

TEAM TOPOLOGIES

"*Team Topologies* provides fresh insights on how to anticipate and adapt to market and technology changes. To survive, enterprises need to unlearn existing command and control structures and instead move authority to leaders with the best information to take action and respond. This book will help executives and business leaders focus on the key strategies of high-performance teams to effectively address the needs of today and the evolving landscape of tomorrow."

—**Barry O'Reilly**, Founder of ExecCamp, Business Advisor, and Author of *Unlearn* and *Lean Enterprise*

"There is nothing more fundamental to management than how you structure your organization and what behaviors you encourage. Despite this, few have attempted to catalog and analyze the organizational design patterns of IT organizations going through digital, DevOps, and SRE transformations. Skelton and Pais have not only accepted this bold challenge, but they've also hit the mark by creating an indispensable and unique resource."

—**Damon Edwards**, Co-Founder of Rundeck

"*Team Topologies* provides a much-needed framework for evaluating and optimizing team organization for increased flow. Teams that have the right size, the right boundaries, and the right level of communication are poised to deliver value to the company and satisfaction to the team members. *Team Topologies* combines a methodical approach with real-world case studies to unlock the full potential of your tech teams."

—**Greg Burrell**, Senior Reliability Engineer at Netflix

"*Team Topologies* by Matthew Skelton and Manuel Pais is unique. It is going to have a big influence across tech companies. We need a structured and methodical approach to shaping teams for continuous delivery instead of copying a few Spotify rituals. This is the book."

—**Nick Tune,** API Platform Lead, Navico

"At Condé Nast International, [the DevOps Topologies] was crucial in understanding our current DevOps state and in defining the vision for our aspirational DevOps operating model. We were able to navigate round the pitfalls and organizational anti-patterns as excellently described in the models.... I am extremely pleased that Matthew and Manuel are growing on the success of the DevOps Topologies and turning their further learnings into the far-reaching book *Team Topologies* for organization design."

—**Crystal Hirschorn**, VP of Engineering, Global Strategy and Operations at Condé Nast

"The high-performing team is the core generator of value in the modern digital economy. But cultivating and scaling an adaptive ecosystem of such teams is a too-often elusive goal. In *Team Topologies*, Skelton and Pais provide innovative tools and concepts for structuring the next generation digital operating model. Recommended for CIOs, enterprise architects, and digital product strategists worldwide."

—**Charles Betz**, Principal Analyst, Forrester Research

"Matthew Skelton and Manuel Pais say '*Team Topologies* is meant to be a functional book'—and it is. It's well constructed and sign-posted, based in sound thinking, and challenges readers to assume, like them, that an organization is a sociotechnical system or ecosystem. From this assumption comes practical suggestions, no prescriptions, and skill in explaining an approach that provides for effective tech/human organization design. For anyone in the tech/organization design field, [*Team Topologies* is] well worth reading."

—**Dr. Naomi Stanford**, Organization Design Practitioner, Teacher, and Author

"I have found Matthew and Manuel's work on patterns and language to be incredibly valuable in both shaping strategies to transform team contexts over time across our organization, as well as in helping business and technology leadership connect with the topics of flow and continuous delivery."

—**Richard James**, Head of Digital Technology & Engineering at Nationwide

"Teams are the fundamental building block of organizations, how those teams work and the system they operate in are the difference between average and high performance. This book is a deep well of information for how you can optimize your organization's system for your current context."

—**Jeremy Brown**, Director, Red Hat Open Innovation Labs EMEA

"DevOps is great, but how do real-world organizations actually structure themselves to do it? You can't just put everyone on a single, silo-less team, all sitting together in one giant open-plan office and going out to lunch or playing foosball together. *Team Topologies* provides a practical set of templates for addressing the key DevOps question that other guides leave as an exercise for the student."

—**Jeff Sussna**, Founder & CEO, Sussna Associates, and Author of *Designing Delivery*

"If you're looking for an analysis of the challenges with the traditional ways of working, and also some practical guidance on mitigation strategies (e.g., new interaction modes, reducing cognitive load, and creating appropriate 'Team APIs'), then this is the book for you!"

—**Daniel Bryant**, Technical Consultant/Advisor and News Manager at *InfoQ*

"*Team Topologies* makes for a fascinating read as it explores the symbiotic relationship between teams and the IT architecture they support. It goes beyond the common approach of static org charts or self-organizing chaos and shows how to evolve the people system and IT system together."

—**Mirco Hering**, Global DevOps Lead Accenture and Author of *DevOps for the Modern Enterprise*

TEAM TOPOLOGIES

ORGANIZING BUSINESS AND TECHNOLOGY TEAMS FOR FAST FLOW

MATTHEW SKELTON
and MANUEL PAIS

Foreword by Ruth Malan

IT Revolution
Portland, Oregon

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

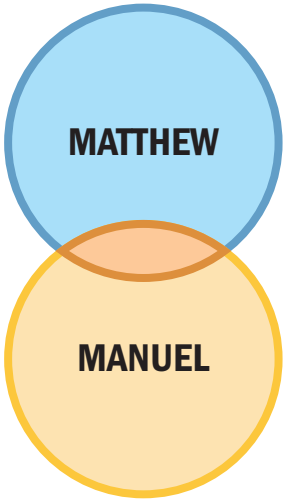
Cover and book design by Devon Smith

Library of Congress Catalog-in-Publication Data
Available upon request

ISBN: 978-1942788-812
eBook ISBN: 978-1942788-829
Kindle ISBN: 978-1942788-836
Web PDF ISBN: 978-1942788-843

For information about special discounts for bulk purchases, or for information
on booking authors for an event, please visit our website atITRevolution.com.

TEAM TOPOLOGIES



To my wife, Suzy Beck—for all your support and inspiration.

To Katie, my life partner and family stronghold—thanks for your tireless love and support.

To Dan and Ben, daily sources of warmth—hopefully this book can help you understand what Daddy does for a living.

CONTENTS

FIGURES & TABLES

FIGURES

TABLES

CASE STUDIES & INDUSTRY EXAMPLES

Chapter 8

FOREWORD

Keeping our systems small and simple is a worthy goal, yet it is also one that most successful systems defy. Lehman's laws of software evolution, and, in particular, continuing growth, captures the evolutionary pressure to add capabilities as systems are used and new demands or opportunities are perceived. Being able to cope with, and even harness, this increasing complexity raises the importance of dual design arenas: the design of systems and the design of the organization that creates and evolves systems. We have a considerable body of work focused on the former; that is, on systems and software design and architecture, including an ever growing number of books on domain driven design and software architecture. *Team Topologies* addresses the design of the software development organization, with Conway's law in view.

> The basic thesis [....] is that organizations which design systems [....] are constrained to produce designs which are copies of the communication structures of these organizations. We have seen that this fact has important implications for the management of system design. Primarily, we have found a criterion for the structuring of design organizations: a design effort should be organized according to the need for communication.[1]

The above quote from the conclusion of Mel Conway's classic paper on organizational design for software development is a most fitting beginning to this book. *Team Topologies* describes organizational patterns for team structure and modes of interaction, taking the force that the organization exerts on the system as a driving design concern.

As the complexity of the system increases, so, generally, do the cognitive demands on the organization building and evolving it. Managing cognitive load through teams with clear responsibilities and boundaries is a distinguishing focus of team design in the Team Topologies approach. To achieve duly scoped,

bounded responsibilities, natural—and relatively independent—system (sub) structure is sought to align teams to. This takes Conway's law into account and leverages it to help maintain cohesive structures with clear boundaries and loose coupling (known as the reverse Conway maneuver, and described herein).

If this was the extent of it, *Team Topologies* would be a useful elaboration of Conway's paper, setting it in the current context. Of course, *Team Topologies* is even more than that. Notably, it identifies four team patterns, describing their outcomes, form, and the forces they address and are shaped by. Stream-aligned teams are the primary team form. These are teams that are optimized for flow, with all they need to effect continuous delivery of value and be fully responsive to the associated feedback cycles. This means that system design seeks not just loose coupling but a decomposition that supports flow and lowers dependencies and coordination needs between stream-aligned teams. Complicated-subsystem and platform teams reduce load for stream-aligned teams, where the latter are internal customers of the former's subsystem or platform capabilities (supporting all phases of development, delivery, and operations for multiple stream teams). Enabling teams likewise serve other teams, but they are service providers, helping stream-aligned teams learn new techniques, investigate new technologies, and so forth, allowing stream-aligned teams to retain focus while growing effectiveness.

Matthew Skelton and Manuel Pais have brought their considerable experience to bear, describing what these various team forms need to be successful, but also highlighting variations in context, identifying the design implications thereof, and indicating anti-patterns to avoid. They also, with great generosity, weave in insights from and offer pointers to related work. This, along with a set of case studies, further textures the book.

Team Topologies informs and enriches our understanding of organizational architecture, via the nuanced presentation of these key structural patterns, interaction modes or dynamics, and considerations for evolution. And, due to its clarity and focus, it serves as a pragmatic guide whether forming teams and enabling them to meet their challenges or helping existing teams become more effective at responsive value delivery.

—**Ruth Malan**, Architecture Consultant at
Bredemeyer Consulting

PREFACE

> [Modern] organisational design . . . is about designing for collaborative technologies, for the voice of the customer.
>
> —**Naomi Stanford**, *Guide to Organization Design*

Teams are always works in progress, but they are also your best shot at delivering value continuously and sustainably by aligning them with the business. Ideally, teams should be long lived and autonomous, with engaged team members. However, teams don't live in isolation. They need to understand how and when to interact with each other. And these team interactions need to evolve over time to support the distinct phases of discovery and execution that products and technology go through during their lifetimes. In short, organizations not only need to strive for autonomous teams, they also need to continuously think about and evolve themselves in order to deliver value quickly to customers.

This book offers a practical, step-by-step, adaptive model for organizational design that we have used and seen work across businesses at varying levels of maturity: Team Topologies.

However, Team Topologies is not a universal formula for building and running software systems successfully. There are teams and organizations who succeed with organizational dynamics very different from those described and recommended here (particularly in organizations with excellent culture and best practices already in place).

Team Topologies *is* meant to provide clear patterns that are straightforward for many different teams and organizations to follow and interpret, not to dictate to outstanding players how to perform. We like to think of Team Topologies as a set of music parts for an orchestra or big band, not the melody for a top jazz trumpeter. Printed music for a large musical ensemble helps the group to succeed but does not dictate every aspect of performance; lots of detail is left for the ensemble to interpret to suit the occasion, venue, or mix of players. Likewise, there is huge value in agreeing to a coherent vocabulary and way of working together across teams to achieve good software delivery.

The Team Topologies approach helps organizations that are struggling to find a way to optimize their team structure, or for those that are not yet aware of the impact team design can have on good business outcomes and software systems in particular. Team Topologies helps organizations succeed more quickly and more continuously than before.

This book is for anyone who cares about the effectiveness of the delivery and operations of software systems: C-level leaders (including CTOs/CIOs, CEOs, CFOs, and so on) managers, heads of department, software architects and systems architects, and anyone else involved in building or running software systems who wants or needs to make the delivery and running of those systems more effective.

How We Came to Write This Book

In 2013, while introducing DevOps and Continuous Delivery at a company in the UK, Matthew devised the original DevOps Topologies patterns (and antipatterns) in a blog post titled "What Team Structure Is Right for DevOps to Flourish?"[1] At the time, the company he was consulting with was struggling to adopt modern approaches to software delivery, and the early topology patterns Matthew created provided the company a way to explore different options.

Manuel interviewed Matthew at the QCon London software development conference back in 2015, where Matthew was speaking on Conway's law and the early DevOps Topology patterns. The resulting article, "How Different Team Topologies Influence DevOps Culture," was published by *InfoQ* and translated into several languages.[2] Later that year, Manuel helped to expand the DevOps Topology patterns and there were contributions from the community.

Since then, the use of DevOps Topology patterns has exploded. They have been referenced over and over again in talks, articles, and conversations. They have helped organizations of all sizes and from varying industries around the

world to think about the relationships between teams and how their interactions influence both organizational culture and software architecture.

Over time, we realized that the original DevOps Topologies presented a static view of team interrelationships that, while useful for initial discussions, was quite limited in scope. Through our combined experience with training and consulting organizations from across the world, we discovered that some teams work better relatively isolated or autonomous, while other teams work better with strong collaboration. We asked ourselves why, and we kept evolving our ideas based on feedback from our clients.

Eventually, this led to the Team Topologies as you see them presented in this book: a dynamic and evolving approach to organizational design based on real scenarios from across different geographies and industries.

How to Use This Book

Team Topologies is meant to be a functional book. It is our intention to provide content that is interactive and delivers as much learning as we are able to fit within these pages. To help with that, we have made some design choices that will help you navigate this book.

First, the book is organized in three parts:

Part I of the book explores Conway's law, the way organizational interrelationships constrain the design of systems we build, and how we can use this tendency to our advantage. We then define what we mean by teams and look at some practical constraints that affect effective teamwork.

In Part II, we investigate a set of static team patterns that have been proven in the industry and the implications of choosing one pattern over another with Conway's law and organizational context in mind. This section should help you think about team topologies that are broadly suitable for your organizational context. This part also provides some guidance in deciding how to align teams to areas of the system, taking into account Conway's law and fundamental team topologies.

Finally, in Part III, we deal with ways to evolve the organization design to provide powerful capabilities for innovation and rapid delivery in response to a quickly changing operating context. We explain how to use the Team Topologies approach to create a sensing organization that responds to the market and user demands, and accounts for the implications this has for hiring and skills.

Each part opens with a breakdown of key takeaways from each of the chapters. Throughout the chapters, we have included figures and callouts to

highlight information we think is helpful to know and/or reference. We also provide easy-to-recognize scenarios, case studies, and explicit recommendations for different situations along the way.

Finally, the shapes, colors, and patterns found within many of the figures also have consistent meaning throughout much of the book. Here is the key:

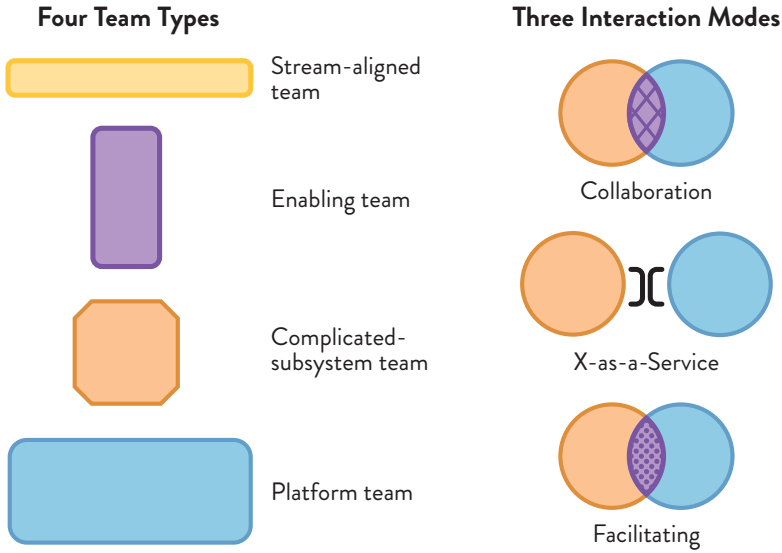


Figure 0.1: The Four Team Types and Three Interaction Modes

For the fullest understanding, you should read the book from cover to cover, as the subject matter builds up chapter by chapter. However, we have written the material so that each section is fairly independent.

In that spirit, here are some scenarios with corresponding ways to read the book that might match with your current situation:

- I need more clarity about different team types and which team types are effective.
  - Review Chapter 1 (overview), then Chapter 4 (static topologies), then Chapter 5 (fundamental topologies).
- I need to split up a large, monolithic software system.
  - Review Chapter 6 (boundaries) and then Chapter 3 (the team).
- I need to improve the architecture of the software system.

    - Review Chapter 2 (Conway's law), then Chapter 4 (static topologies), then Chapter 6 (boundaries).
- I need to improve the effectiveness of software development teams.
    - Review Chapter 3 (the team), then Chapter 6 (boundaries), then Chapter 5 (fundamental topologies).
- I need to improve morale and effectiveness within teams.
    - Review Chapter 3 (the team) and then Chapter 5 (fundamental topologies).
- I need to understand where to invest effort to help with projected growth.
    - Review Chapter 1 (overview), then Chapter 5 (fundamental topologies), then Chapter 8 (topology evolution).
- I need to understand how to evolve team topologies to meet changing business needs.
    - Review Chapter 7 (dynamic aspects) and then Chapter 8 (topology evolution and organizational sensing).

Key Influences that Informed this Book

In addition to our own experience, this book is strongly influenced by several related approaches and sets of thinking. First, we assume that an organization is a sociotechnical system or ecosystem that is shaped by the interaction of individuals and the teams within it; in other words, that an organization is the interaction between people and technology. In this aspect, the book fits with ideas from the fields of: cybernetics (especially the use of the organization as a "sensing mechanism," which goes back as far as 1948, when Norbert Wiener's book *Cybernetics: Or Control and Communication in the Animal and the Machine* was first published), systems thinking (particularly the work of W. Edwards Deming), and approaches such as the Cynefin framework for assessing domain complexity (described by Dave Snowden and Mary Boone in their 2007 *Harvard Business Review* paper titled "A Leader's Framework for Decision Making"), and adaptive structuration theory (a term coined by Gerardine DeSanctis and Marshall Scott Poole in their *Organization Science* article, "Capturing the Complexity in Advanced Technology Use: Adaptive Structuration Theory," where they emphasized that the impact of technology is not a given, as it depends on how groups and organizations perceive it).

Second, we assume that "the team" is something that behaves differently from a mere collection of individuals, and that the team should be nurtured and

supported in its evolution and operation. In this respect, we draw on ideas from Bruce Tuckman (who proposed the four-stages model—forming, storming, norming, performing—for team development in his 1965 paper "Developmental Sequence in Small Groups"), Russ Forrester and Allan Drexler (who explored team-based organization performance in their 1999 paper "A Model for Team-Based Organization Performance"), Pamela Knight (who found evidence that storming takes place throughout the entire lifetime of a team in her 2007 paper "Acquisition Community Team Dynamics: The Tuckman Model vs. the DAU Model"), Patrick Lencioni (who explores common interaction issues in his seminal book *The Five Dysfunctions of a Team: A Leadership Fable)*, and similar team-focused theories and research.

Third, we assume that Conway's law (or a variant of it) is a strong driver of software product shape and that organizations would benefit from explicitly addressing the implications of this law. In this regard, we draw on writing and ideas from Mel Conway; from software architecture consultant and team organization design award-winner Ruth Malan; from ThoughtWorks technical director and one of the "reverse Conway maneuver" proponents James Lewis; and from similar authors and practitioners.

Finally, we draw on numerous sources that describe practical successes developing and running software systems at scale, including organizations such as Adidas, Auto Trader, Ericsson, Netflix, Spotify, TransUnion, and others. The size and speed of these organizations has made it possible for them to see tangible gains from changes in organization structure and team interaction over the space of several months to a few years.

As you travel through this book, we hope you get inspired to challenge how you think about teams, their structures, and how they function.

PART I

Teams As the Means of Delivery

KEY TAKEAWAYS

CHAPTER 1

- Conway's law suggests major gains from designing software architectures and team interactions together, since they are similar forces.
- Team Topologies clarifies team purpose and responsibilities, increasing the effectiveness of their interrelationships.
- Team Topologies takes a humanistic approach to building software systems while setting up organizations for strategic adaptability.

CHAPTER 2

- Organizations are constrained to produce designs that reflect communication paths.
- The design of the organization constrains the "solution search space," limiting possible software designs.
- Requiring everyone to communicate with everyone else is a recipe for a mess.
- Choose software architectures that encourage team-scoped flow.
- Limiting communication paths to well-defined team interactions produces modular, decoupled systems.

CHAPTER 3

- The team is the most effective means of software delivery, not individuals.
- Limit the size of multi-team groupings within the organization based on Dunbar's number.
- Restrict team responsibilities to match the maximum team cognitive load.
- Establish clear boundaries of responsibility for teams.
- Change the team working environment to help teams succeed.

1 The Problem with Org Charts

> Organizations should be viewed as complex and adaptive organisms rather than mechanistic and linear systems.
>
> —**Naomi Stanford**, Guide to Organisation Design

Technology workers are in a constant state of action: creating and updating systems at an unbelievable rate, and combining different types of technology to create a compelling user experience. Mobile applications; cloud-based services; web applications; and embedded, wearable, or industrial IoT devices all need to interoperate effectively to achieve the desired business outcomes.

Today, these systems affect nearly every aspect of people's day-to-day lives in ways that are increasingly profound. If software is poorly designed—or more importantly, if there is a mismatch in the interaction of the software, the provider, and the customer—people will be adversely affected. They can be stranded long distances from home if a taxi-hailing application fails. They may be unable to pay rent if the software or processes for internet banking fail. They may even see their life in danger if a medical device fails. Never before has explicit sociotechnical design been so important.

Building and running these highly complex, interconnected software systems is a team activity, requiring the combined efforts of people with different skills across different platforms. In addition, modern IT organizations must deliver and operate software systems rapidly *and* safely, while simultaneously growing and adapting to changes and pressures in the business or regulatory

environment. Businesses can no longer choose between optimizing for stability and optimizing for speed.

But despite these risks and demands, many organizations are still organizing their people and teams in ways that are counterproductive to modern software development and operations. Organizations that rely too heavily on org charts and matrixes to split and control work often fail to create the necessary conditions to embrace innovation while still delivering at a fast pace. In order to succeed at that, organizations need stable teams and effective team patterns and interactions. They need to invest in empowered, skilled teams as the foundation for agility and adaptability. To stay alive in ever more competitive markets, organizations need teams and people who are able to sense when context changes and evolve accordingly.

The good news is that it *is* possible to be fast and safe with the right mindset and with tools that emphasize adaptability as well as repeatability, while putting teams and people at the center. As Mark Schwartz and co-authors put it in their 2016 paper *Thinking Environments*, "the organizational structure must coordinate accountabilities to support the goals of delivering high-quality, impactful software."[1]

As members of the technology teams managing these interfaces, we must shift our thinking from treating teams as collections of interchangeable individuals that will succeed as long as they follow the "right" process and use the "right" tools, to treating people and technology as a single human/computer carbon/silicon sociotechnical ecosystem. At the same time, we need to ensure that teams are intrinsically motivated and are given a real chance of doing their best work within such a system.

This chapter will introduce Team Topologies as an adaptive model for technology organization design allowing businesses to achieve speed *and* stability. But first, let's look at how real communication structures in most organizations are often quite distinct from what the org chart tells us, and what the implications of that are.

Communication Structures of an Organization

Most organizations want or are required to have a single view of their teams and people called the "org chart." This chart depicts the teams, departments, units, and other organizational entities, as well as how they relate to each other. It usually shows hierarchical lines of reporting, which imply lines of communication running "up and down" the organization.

The org chart does have its uses in the context of building software systems, specifically around regulatory and legal compliance. However, in a highly collaborative context filled with uncertainty over outcomes, relying on the org chart as a principal mechanism of splitting the work to be done leads to unrealistic expectations. We need to rely instead on decoupled, long-lived teams that can collaborate effectively to meet the challenge of balancing speed and safety.

The problem with taking the org chart at face value is that we end up trying to architect people as if they were software, neatly keeping their communication within the accepted lines. But people don't restrict their communications only to those connected lines on the chart. We reach out to whomever we depend on to get work done. We bend the rules when required to achieve our goals. That's why *actual* communication lines look quite different from the org chart, as shown in Figure 1.1 (see page 6).

Org Chart Thinking Is the Problem

Traditional org charts don't help us understand what the actual patterns of communication in our organization are, as illustrated in Figure 1.1. Instead, organizations need to develop more realistic pictures of the expected and actual communication happening between individuals and teams. The gaps will help inform what types of systems are a better fit for the organization.

Furthermore, decisions based on org-chart structure tend to optimize for only part of the organization, ignoring upstream and downstream effects. Local optimizations help the teams directly involved, but they don't necessarily help improve the overall delivery of value to customers. Their impact might be negligent if there are larger bottlenecks in the stream of work. For example, having teams adopting cloud and infrastructure-as-code can reduce the time to provision new infrastructure from weeks or months to minutes or hours. But if every change requires deployment (to production) approval from a board that meets once a week, then delivery speed will remain weekly at best.

Systems thinking focuses on optimizing for the whole, looking at the overall flow of work, identifying what the largest bottleneck is today, and eliminating it. Then repeat. Team Topologies focuses on how to set up dynamic team structures and interaction modes that can help teams adapt quickly to new conditions, and achieve fast and safe software delivery. This might not be your largest bottleneck today, but eventually, you will face the issue of rigid team structures with poor communication and/or inadequate processes, slowing down delivery.

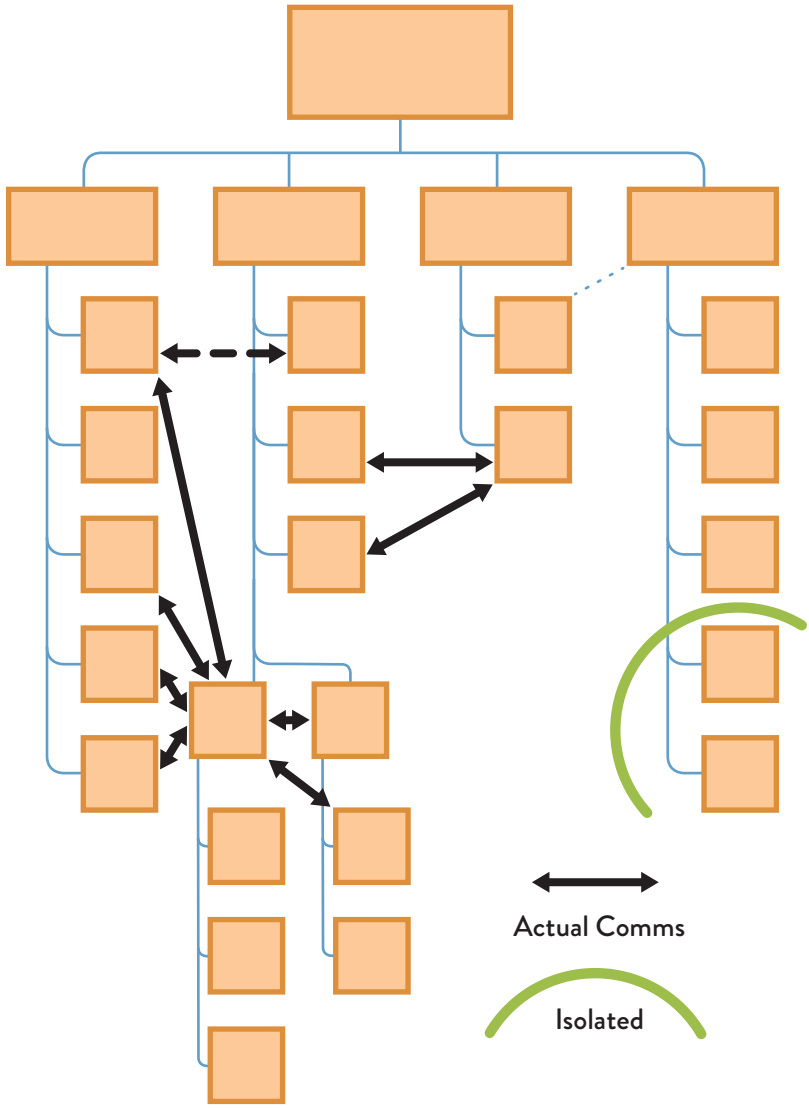


Figure 1.1: Org Chart with Actual Lines of Communication
In practice, people communicate laterally or "horizontally" with people from other reporting lines in order to get work done. This creativity and problem solving needs to be nurtured for the benefit of the organization, not restricted to optimize for top-down/bottom-up communication and reporting.

Thinking of the org chart as a faithful representation of how work gets done and how teams interact with each other leads to ineffective decisions around allocation of work and responsibilities. Much like a software architecture document gets outdated as soon as the actual software development starts, an org chart is always out of sync with reality.

Naturally, we are by no means the first to acknowledge the imbalance between formal organization structures and the way work actually gets done. Geary Rummler and Alan Brache's book *Improving Performance: How to Manage the White Space on the Organization Chart* set the stage for continuous business process improvement and management. The recent focus (at least within IT) on product and team centricity, as illustrated by Mik Kersten's book on moving from *Project to Product*, is another major milestone. We like to think that Team Topologies is another piece of this puzzle—in particular, having clear and fluid team structures, responsibilities, and interaction modes.

Beyond the Org Chart

So if org charts are not an accurate representation of organizational structures, what is? Niels Pflaeging, author of *Organize for Complexity*, identifies not one but three different organizational structures in every organization:[2]

1. Formal structure (the org chart)—facilitates compliance
2. Informal structure—the "realm of influence" between individuals
3. Value creation structure—how work actually gets done based on inter-personal and inter-team reputation

Pflaeging suggests that the key to successful knowledge work organizations is in the interactions between the informal structure and the value creation structure (that is, the interactions between people and teams).[3] Other authors have proposed similar characterizations, such as Frédéric Laloux in *Reinventing Organizations* or Brian Robertson's *Holacracy* approach.[4]

The Team Topologies approach acknowledges the importance of informal and value creation structures as defined by Pflaeging. By empowering teams, and treating them as fundamental building blocks, individuals inside those teams move closer together to act as a team rather than just a group of people. On the other hand, by explicitly agreeing on interaction modes with other teams, expectations on behaviors become clearer and inter-team trust grows.

Over the last several decades, there have been many new approaches to organizing businesses, but usually the new design remains a static view of

the organization that does not take into consideration the real behaviors and structures that emerge after reorganization. For instance, the "matrix management" approach that started in the 1990s—and became quite popular over the next couple of decades—tried to address the inherent complexity of highly uncertain, highly skilled work by having individuals report to both business and functional managers. Despite a clearer focus on business value compared to a purely functional organization of teams, this is still a static view of the world that becomes outdated as the business and technology domains quickly evolve.

For workers, re-orgs, like introducing matrix management, can bring a lot of fear and worry. Often, it's seen as a time and effort drain that is more likely to set the business back rather than move it forward. And once the next technological or methodological revolution hits, the business undertakes yet another re-org, breaking down established forms of communication and splitting up teams that were just starting to get their mojo.

It is increasingly clear that relying on a single, static organizational structure, like the org chart or matrix management, is untenable for effective outcomes with modern software systems. Instead of a single structure, what is needed is a model that is adaptable to the current situation—one that takes into consideration how teams grow and interact with each other. Team Topologies provides the (r)evolutionary approach required to keep teams, processes, and technology aligned for all kinds of organizations.

The Team Topologies approach adds the dynamic and sensing aspects required for technology organizations that are missing from traditional organization design.

In her excellent 2015 book, *Guide to Organisation Design: Creating High-Performing and Adaptable Enterprises*, Naomi Stanford lists five rules of thumb for designing organizations:[5]

1. Design when there is a compelling reason.
2. Develop options for deciding on a design.
3. Choose the right time to design.
4. Look for clues that things are out of alignment.
5. Stay alert to the future.

As we continue to move through this book, we will explore how to address these five heuristics for organization design.

Team Topologies: A New Way of Thinking about Teams

The Team Topologies approach brings new thinking around effective team structures for enterprise software delivery. It provides a consistent, actionable guide for evolving team design to continuously cope with technology, people, and business changes, covering size, shape, placement, responsibilities, boundaries, and interaction of teams building and running modern software systems.

Team Topologies provides four fundamental team types—*stream-aligned*, *platform, enabling*, and *complicated-subsystem*—and three core team interaction modes—*collaboration*, *X-as-a-Service*, and *facilitating*. Together with awareness of Conway's law, team cognitive load, and how to become a sensing organization, Team Topologies results in an effective and humanistic approach to building and running software systems.

In particular, it looks at ways in which different team topologies can evolve with technological and organizational maturity. Periods of technical and product discovery typically require a highly collaborative environment (with overlapping team boundaries) to succeed. But keeping the same structures when discovery is over (established technologies and product) can lead to wasted effort and misunderstandings.

By emphasizing an adaptive model for organization design and actively prioritizing the interrelationship of teams, the Team Topologies approach provides a key technology-agnostic mechanism for modern software-intensive enterprises to sense when a change in strategy is required (either from a business or technology viewpoint). The end goal is to help teams produce software that aligns with customer needs and is easier to build, run, and own.

Team Topologies also emphasizes a humanistic approach to designing and building software systems. It sees the team as an indivisible element of software delivery and acknowledges that teams have a finite cognitive capacity that needs to be respected. Together with the dynamic team design solidly grounded on Conway's law, Team Topologies becomes a strategic tool for solution discovery.

The Revival of Conway's Law

We've mentioned the importance of Conway's law as a driver for team design and evolution. But what is this law exactly?

In 1968, the computer systems researcher Mel Conway published a paper in *Datamation* called "How Do Committees Invent?" in which he explored the relationship between organizational structure and the resulting design of

systems. The article is full of sparkling insights, some of which we cover later in this chapter, but this is the phrase that became known as *Conway's law*: "Organizations which design systems . . . are constrained to produce designs which are copies of the communication structures of these organizations."[6]

Conway based his observation on organizations building early electronic computer systems. In his words, this "law" indicates the strong correlation between an organization's real communication paths (the value creation structures mentioned by Pflaeging) and the resulting software architecture,[7] or what author Allan Kelly calls the "homomorphic force."[8] This homomorphic force tends to make things the same shape between the software architecture and team structures. In other words, building software requires an understanding of communication across teams in order to realistically consider what kind of software architectures are feasible. If the desired theoretical system architecture does not fit the organizational model, then one of the two will need to change.

Eric Raymond stated this in a humorous way in his book *The New Hacker's Dictionary:* "If you have four groups working on a compiler, you'll get a 4-pass compiler."[9]

Since 1968, it has become increasingly clear that Conway's law continues to apply to all software built. Those of us who have built software systems that had to comply with an "architecture blueprint" can surely remember having times when it felt like we were fighting against the architecture rather than it helping steer our work in the right direction. Well, that's Conway's law in action.

Team structures must match the required software architecture or risk producing unintended designs.

A sort of "revival" of Conway's law took place around 2015, when microservices architectures were on the rise. In particular, James Lewis, Technical Director at Thoughtworks, and others came up with the idea of applying an "inverse Conway maneuver" (or reverse Conway maneuver), whereby an organization focuses on organizing team structures to match the architecture they want the system to exhibit rather than expecting teams to follow a mandated architecture design.[10]

The key takeaway here is that thinking of software architecture as a standalone concept that can be designed in isolation and then implemented by any group of teams is fundamentally wrong. This gap between architecture and team structures is visible across all types of architectures, from client-server to SOA and even microservices. Specifically, that is why monoliths need to be

broken down (in particular, any indivisible software part that exceeds the cognitive capacity of any one team) while keeping a team focus, a topic we will discuss in depth in Chapter 6.

Cognitive Load and Bottlenecks

When we talk about cognitive load, it's easy to understand that any one person has a limit on how much information they can hold in their brains at any given moment. The same happens for any one team by simply adding up all the team members' cognitive capacities.

However, we hardly ever discuss cognitive load when assigning responsibilities or software parts to a given team. Perhaps because it's hard to quantify both the available capacity and what the cognitive load will be. Or perhaps because the team is expected to adapt to what it's being asked to do, no questions asked.

When cognitive load isn't considered, teams are spread thin trying to cover an excessive amount of responsibilities and domains. Such a team lacks bandwidth to pursue mastery of their trade and struggles with the costs of switching contexts.

Miguel Antunes, R&D Principle Software Engineer at OutSystems, a low-code platform vendor, relayed an example of this very challenge. Their Engineering Productivity team at OutSystems was five years old. The team's mission was to help product teams run their builds efficiently, maintain infrastructure, and improve test execution. The team kept growing and took on extra responsibilities around continuous integration (CI), continuous delivery (CD), and infrastructure automation.

Victims of their own success, sprint planning for the now eight-person-strong team was a mix and match of requests across their stack of responsibilities. Prioritization was hard, and the frequent context switching even throughout a single sprint led to a dip in people's motivation. This is not surprising if we consider Dan Pink's three elements of intrinsic motivation: autonomy (quashed by constant juggling of requests and priorities from multiple teams), mastery ("jack of all trades, master of none"), and purpose (too many domains of responsibility).[11]

While the team in this industry example was providing internal services to development teams, the effect is the same for teams working on software for external customers. The number of services and components for which a product team is responsible (in other words, the demand on the team)

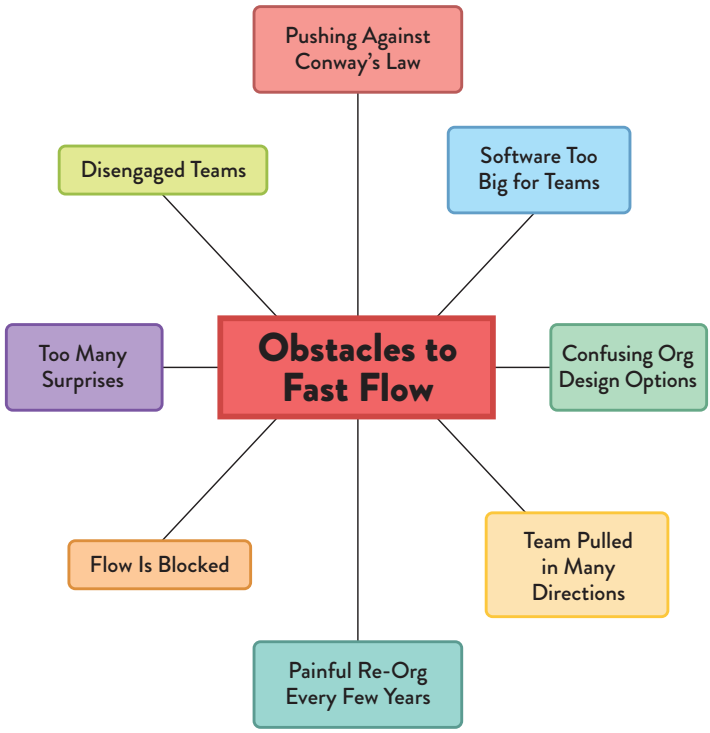


Figure 1.2: Obstacles to Fast Flow

typically keeps growing over time. However, the development of new services is often planned as if the team had full-time availability and zero cognitive load to start with. This neglect is problematic because the team is still required to fix and enhance existing services. Ultimately, the team becomes a delivery bottleneck, as their cognitive capacity has been largely exceeded, leading to delays, quality issues, and often, a decrease in team members' motivation.

We need to put the team first, advocating for restricting their cognitive loads. Explicitly thinking about cognitive load can be a powerful tool for deciding on team size, assigning responsibilities, and establishing boundaries with other teams. (We will cover this in detail in Chapter 3.)

Overall, the Team Topologies approach advocates for organization design that optimizes for *flow of change* and feedback from running systems. This requires restricting cognitive load on teams and explicitly designing the intercom-

munications between them to help produce the software-systems architecture that we need (based on Conway's law).

Summary: Rethink Team Structures, Purpose, and Interactions

Developing and operating software effectively for modern, interconnected systems and services requires organizations to consider many different dimensions. Historically, most organizations have seen software development as a kind of manufacturing to be completed by separate individuals arranged into functional specialties, with large projects planned up front and with little consideration for sociotechnical dynamics. This led to the prevailing problems depicted in Figure 1.2 on page 12.

The Agile, Lean IT, and DevOps movements helped demonstrate the enormous value of smaller, more autonomous teams that were aligned to the flow of business, developing and releasing in small, iterative cycles, and course correcting based on feedback from users. Lean IT and DevOps also encouraged big strides in telemetry and metrics tooling for both systems and teams, helping people building and running software to make proactive, early decisions based on past trends, rather than simply responding to incidents and problems as they arose.

However, traditional organizations have often been limited in their ability to fully reap the benefits of Agile, Lean IT, and DevOps due to their organizational models. It's no surprise that there is a strong focus on the more immediate automation and tooling adoption, while cultural and organizational changes are haphazardly addressed. The latter changes are much harder to visualize, let alone to measure their effectiveness. Yet having the right team structure, approach, and interaction in place, and understanding their need to evolve over time is a key differentiator for success in the long run.

In particular, traditional org charts are out of sync with this new reality of frequent (re)shaping of teams for collaborative knowledge work in environments filled with uncertainty and novelty. Instead, we need to take advantage of Conway's law (organizational design prevails over software architecture design), cognitive load restrictions, and a team-first approach in order to design teams with clear purposes and promote team interactions that prioritize flow of software delivery and strategic adaptability.

The goal of *Team Topologies* is to give you the approach and mental tools to enable your organization to adapt and dynamically find the places and timing

when collaboration is needed, as well as when it is best to focus on execution and reduce communication overhead.

> ***NOTE***
>
> We found a fascinating example of strategic and collaborative interaction in a totally different field when researching for this book. It turns out that grouper fish and moray eels, seemingly unrelated species (silos, anyone?), explicitly collaborate (via signals) to hunt down smaller fishes that hide in crevices. The eel sneaks into the crevices and scares off smaller fish, which are then forced to come out and become easy prey for the grouper. Read on to find out how to enable the groupers and eels in your organization to join forces for better flow and business outcomes!

2 Conway's Law and Why It Matters

> [Conway's law] creates an imperative to keep asking: "Is there a better design that is not available to us because of our organization?"
>
> —**Mel Conway**, *Toward Simplifying Application Development, in a Dozen Lessons*

In Chapter 1, we discussed why organizations need to consider team organization as an integral factor to success. We also discussed the underpinning ideas and principles that help us understand how teams work within an organization. We introduced some key concepts that we will begin to build on throughout the book. In the remaining chapters of Part I, we will discuss in more detail what Conway's law reveals about teams, organization structure, and software architecture; then we will dig into what a team-first approach means. The goal in Part I is to give you the foundational principles for organization and team design that you will need to understand as you consider team topologies, starting with Conway's law.

Understanding and Using Conway's Law

Conway's law is critical to understanding the forces at play when organizing teams amidst the long-lasting, unattended impact they can have on our software systems as the latter have become larger and more interconnected than ever before. But you might wonder if a law from 1968 about software architecture has stood the test of time.

We've come a long way after all: microservices, the cloud, containers, serverless. In our experience, such novelties can help teams improve locally, but the larger the organization, the harder it becomes to reap the full benefits. The way teams are set up and interact is often based on past projects and/or legacy technologies (reflecting the latest org-chart design, which might be years old, if not decades).

If you've ever worked for a large organization, you have likely encountered examples of monolithic shared databases powering an entire business. There were, of course, valid historical reasons for the predominance of monolithic databases (such as the rise in specialism of people and teams on technical stack layers) up until DevOps and microservices gained traction. Factors such as project orientation, cost cutting via outsourcing, or junior teams without sufficient experience have contributed to the perpetuation of this (now recognizable) anti-pattern. Monolithic databases couple the applications that depend on them and become magnets for small-business logic changes at the database level (more on this in Chapter 6). Yet, to avoid them, organizations need not only good architectural practices but also actual team structures and composition that align with this new way of thinking.

Sportswear company Adidas went through an interesting transformation where they explicitly looked at Conway's law as a driver for organization design. As Fernando Cornago, Senior Director of Platform Engineering, and Markus Rautert, Vice President of Platform Engineering and Architecture, explained their IT department went from being seen as a cost center, with a single vendor providing most of the software (requiring frequent hand-offs) and only a few in-house engineers (doing more managing than engineering), to a product-oriented team organization. Adidas invested 80% of its engineering resources to creating in-house software delivery capabilities via cross-functional teams aligned with business needs. The other 20% were dedicated to a central-platform team taking care of engineering platforms and technical evolution, as well as consulting and onboarding new professionals. Adidas was able to increase release frequency of their digital products sixtyfold, while positively impacting software quality as well.[1]

Besides empirical experience, there's also an increasing body of research that generally confirms the tendencies outlined by Conway. Alan MacCormack and colleagues at Harvard Business School undertook studies of various open-source and closed-source software products and found "strong evidence to support the hypothesis that a product's architecture tends to mirror the structure of the organization in which it is developed."[2]

Studies in other industries, such as vehicle manufacturing and aircraft engine design, also corroborate this idea.[3] In fact, there has been enough industry research undertaken to show that the homomorphic force identified by Conway's law applies broadly.

This quote from Ruth Malan provides what could be seen as the modern version of Conway's law: "If the architecture of the system and the architecture of the organization are at odds, the architecture of the organization wins."[4] Malan reminds us that the organization is constrained to produce designs that match or mimic the real, on-the-ground communication structure of the organization. This has significant strategic implications for any organization designing and building software systems, whether in-house or via suppliers.

In particular, an organization that is arranged in functional silos (where teams specialize in a particular function, such as QA, DBA, or security) is unlikely to ever produce software systems that are well-architected for end-to-end flow. Similarly, an organization that is arranged primarily around sales channels for different geographic regions unlikely to produce effective software architecture that provides multiple different software services to all global regions.

Why are organizations unlikely to discover or sustain certain architectures? Conway provides some clues in his 1968 article: "Given any [particular] team organization, there is a class of design alternatives which cannot be effectively pursued by such an organization because the necessary communication paths do not exist."[5]

Communication paths (along formal reporting lines or not) within an organization effectively restrict the kinds of solutions that the organization can devise. But we can use this to our strategic advantage. If we want to discourage certain kinds of designs—perhaps those that are too focused on technical internals—we can reshape the organization to avoid this. Similarly, if we want our organization to discover and adopt certain designs—perhaps those more amenable to flow—then we can reshape the organization to help make that happen. There is, of course, no guarantee that the organization will find and use the designs we want, but at least by shaping the communication paths, we are making it more likely.

Organization design using Conway's law becomes a key strategic activity that can greatly accelerate the discovery of effective software designs and help avoid those less effective. (In Chapter 8, we go into more detail on how to evolve an organization strategically with Conway's law in mind.)

The Reverse Conway Maneuver

To increase an organization's chances of building effective software systems optimized for flow, a reverse Conway maneuver (or inverse Conway maneuver) can be undertaken to reconfigure the team intercommunications before the software is finished. Although you might get initial pushback, with sufficient willpower from management and awareness from teams this approach can and does work.

The reverse Conway maneuver gained traction in the technology world around 2015 and has been applied in many organizations since. *Accelerate: The Science of Dev Ops* by Nicole Forsgren, PhD, Jez Humble, and Gene Kim supports the importance of this strategy for high-performing organizations:

> Our research lends support to what is sometimes called the "inverse Conway maneuver," which states that organizations should evolve their team and organizational structure to achieve the desired architecture. The goal is for your architecture to support the ability of teams to get their work done—from design through to deployment—without requiring high-bandwidth communication between teams.[6]

Remember the monolithic database anti-pattern we mentioned earlier? We've seen extreme cases where, because there were no stable teams and all changes were made via temporary projects (mostly outsourced), applications became deeply coupled at the database level (shared data and procedures). This later impeded adoption of commodity systems for certain parts of the business since the latter could not be decoupled from the rest of the business logic. Instead of freeing up in-house engineers to work on differentiating features that meet evolving customer needs, accruing technical debt like this curtails an organization's ability to move faster and make a difference against competitors.

So, how can the reverse Conway maneuver help steer team organization to obtain the desired software architecture?

Let's look at a deliberate simplification of Conway's law in an organization building software to illustrate the ideas and forces at work. Let's say that four independent teams, each comprised of front-end and back-end developers, work on different parts of a system and then hand over to a database administrator (DBA) for database changes. The flow of changes may look conceptually like the diagram in Figure 2.1.

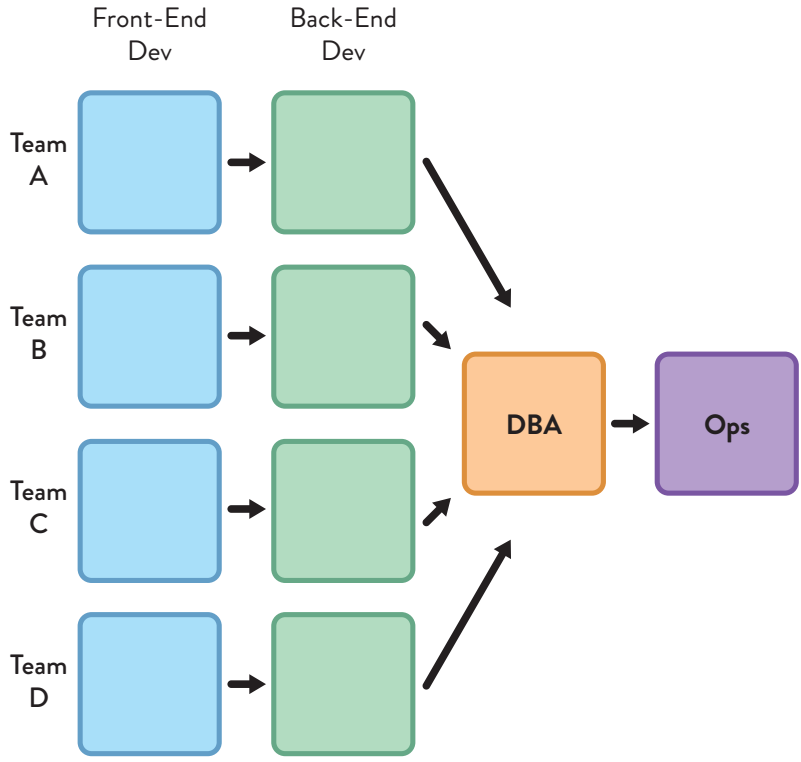


Figure 2.1: Four Teams Working on a Software System
Four separate teams consisting of front-end and back-end developers work on a software system. Front-end devs communicate only with back-end devs, who communicate with a single DBA for the database changes.

According to Conway's law, the software architecture that naturally emerges from such a team design would have separate front-end and back-end components for each team, and a single, shared core database (Figure 2.2, see page 20).

In other words, the use of a shared DBA team is likely to drive the emergence of a single shared database; and the use of separate front-end and back-end developers is likely to drive a separation between UI and app tiers, due to the nature of the communication taking place. If this single shared database and four, two-tier apps is the software architecture we want, then all is well.

However, if we do *not* want a single shared database, we have a problem. The homomorphic force identified by Conway's law is exerting a strong pull on

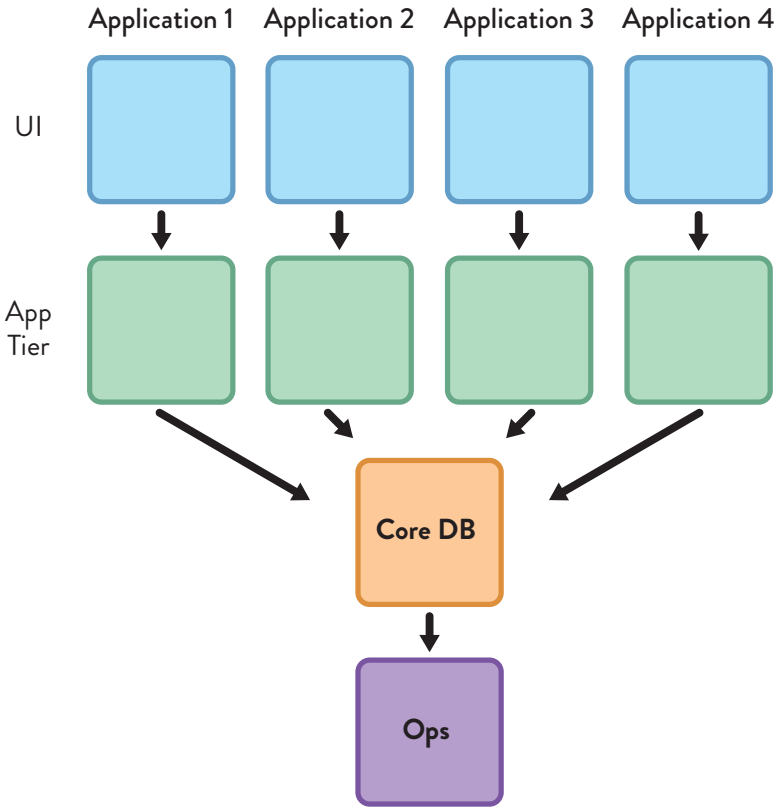


Figure 2.2: Software Architecture from Four-Team Organization
Four separate applications, each with a separate user interface (UI) and a back-end application tier that communicate with a single shared database. This reflects and matches the team communication architecture from Figure 2.1; the diagram has simply been rotated ninety degrees.

the "natural" software architecture to emerge from the current organization design and communication paths.

For example, let's say that we want to use a microservices architecture for some new cloud-based software systems, where each separate service is independent and has its own data store (Figure 2.3, see page 21).

By applying the reverse Conway maneuver, we can design our teams to "match" the required software architecture by having separate developers for the client applications and the API, and a database developer within the team rather than separate from it (Figure 2.4, see page 22).

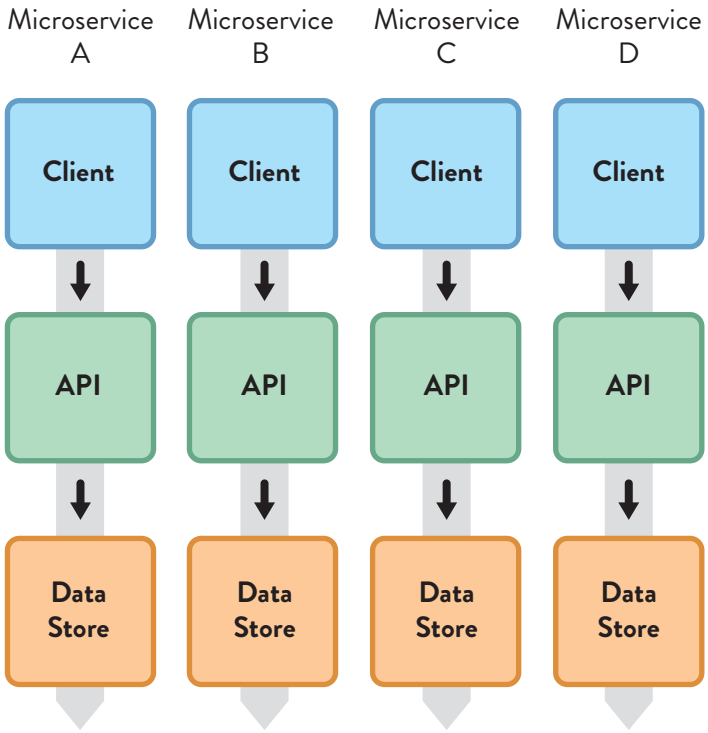


Figure 2.3: Microservices Architecture with Independent Services and Data Stores
A microservices-based architecture with four separate services, each with its own data store, API layer, and front-end client.

According to Conway's law, this team design will most "naturally" produce the desired software architecture. If we want our data store to be aligned with the business domain, then we need to avoid having a single "fan-in" database person or team (perhaps by adding a data capability within the application-development team).

Software Architectures that Encourage Team-Scoped Flow

Conway's law tells us that we need to understand what software architecture is needed *before* we organize our teams, otherwise the communication paths and incentives in the organization will end up dictating the software

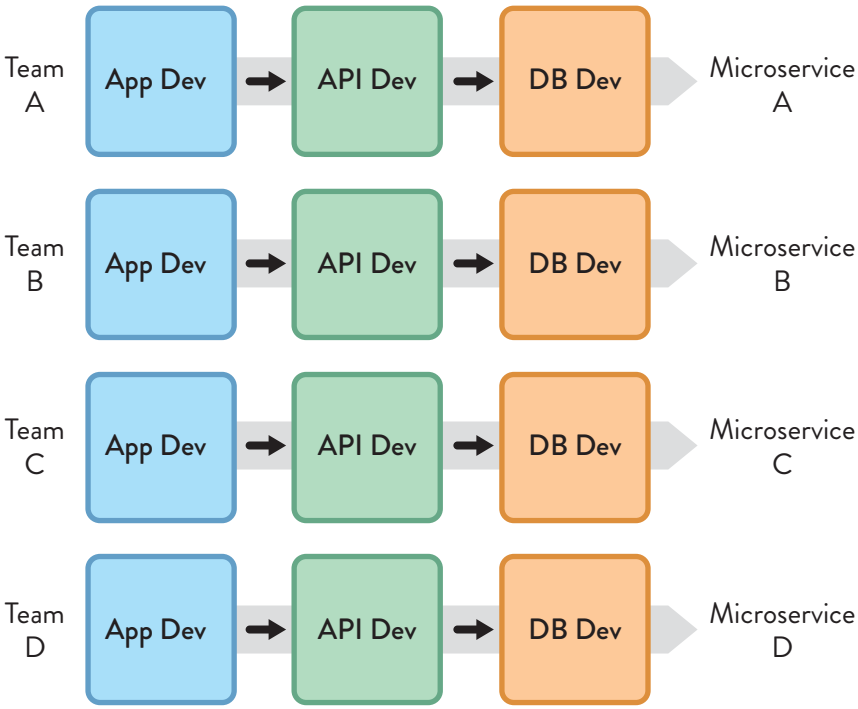


Figure 2.4: Team Design for Microservices Architecture with Independent Services and Data Stores

An organization design that anticipates the homomorphic force behind Conway's law to help produce a software architecture with four independent microservices. (Again, this is basically the diagram in Figure 2.3 rotated ninety degrees.)

architecture. As Michael Nygard says: "Team assignments are the first draft of the architecture."[7]

For a safe, rapid flow of changes, we need to consider team-scoped flow and design the software architecture to fit it. The fundamental means of delivery is the team (see more in Chapter 3), so the system architecture needs to enable and encourage fast flow within each team. Thankfully, in practice, this means that we can follow proven software-architecture good practices:

- Loose coupling—components do not hold strong dependencies on other components
- High cohesion—components have clearly bounded responsibilities, and their internal elements are strongly related
- Clear and appropriate version compatibility

- Clear and appropriate cross-team testing

At a conceptual level, software architectures should resemble the flows of change they enable; instead of a series of interconnected components, we should be designing flows on top of an underlying platform (we will cover platforms in Chapter 5).

By keeping things team sized, we help to achieve what MacCormack and colleagues call "an 'architecture for participation' that promotes ease of understanding by limiting module size, and ease of contribution by minimizing the propagation of design changes."[8] In other words, we need a team-first software architecture that maximizes people's ability to work with it.

Keeping things decoupled and team-scoped should be a key, ongoing organization test because, as John Roberts says in *The Modern Firm*, "real gains in performance can often be achieved by adopting designs that adhere to [a] disaggregated model."[9] These performance gains are partly due to the increased rate of flow of change and partly due to the organization's ability to change the architecture to suit new contexts.

Don Reinertsen, author of *The Principles of Product Development Flow*, says "we can also exploit architecture as an enabler of rapid changes. We do this by partitioning our architecture to gracefully absorb change."[10] Architecture thus becomes an enabler, not a hindrance, but only if we take a team-first approach informed by Conway's law.

Organization Design Requires Technical Expertise

If we accept that the self-similar force (between architecture and team organization) described by Conway is real, then we also need to accept that anyone who makes decisions about the shape and placement of engineering teams is strongly influencing the software systems architecture. There is a logical implication of Conway's law here, in the words of Ruth Malan: "if we have managers deciding . . . which services will be built, by which teams, we implicitly have managers deciding on the system architecture."[11]

How much awareness does the HR department have about software systems? Does the group of department leaders deciding how to allocate budget across teams know of the likely effects of their choices on the viability of the software architecture?

Given that there is increasing evidence for the homomorphism behind Conway's law, it is very ineffective (perhaps irresponsible) for organizations

that build software systems to decide on the shape, responsibilities, and boundaries of teams without input from technical leaders.

Organization design and software design are, in practice, two sides of the same coin, and both need to be undertaken by the same informed group of people. Allan Kelly's view of a software architect's role expands further on this idea:

> More than ever I believe that someone who claims to be an Architect needs both technical and social skills, they need to understand people and work within the social framework. They also need a remit that is broader than pure technology—they need to have a say in organizational structures and personnel issues, i.e. they need to be a manager too.[12]

Fundamentally, we need to involve technical people in organization design because they understand key software design concepts, such as APIs and interfaces, abstraction, encapsulation, and so on. Naomi Stanford puts it like this: "departments and divisions, systems, and business processes . . . can be designed independently as long as interfaces and boundaries with the wider organization form part of the design."[13]

Restrict Unnecessary Communication

One key implication of Conway's law is that not all communication and collaboration is good. Thus it is important to define "team interfaces" to set expectations around what kind of work requires strong collaboration and what doesn't. Many organizations assume that more communication is always better, but this is not really the case.

What we need is *focused* communication between specific teams. We need to look for unexpected communication and address the cause; as Manuel Sosa and colleagues found in their 2004 research into aircraft manufacturing, "managers should focus their efforts on understanding the causes of unaddressed design interfaces . . . and unpredicted team interactions . . . across modular systems."[14]

Mike Cohn, one of the originators of the Scrum product-development approach, asks these questions to assess the health of inter-team communication within an organization: "Does the structure minimize the number of communication paths between teams? . . . Does the structure encourage teams to communicate who wouldn't otherwise do so?[15]

Here, Cohn is addressing the need to ensure that if, logically, two teams shouldn't need to communicate based on the software architecture design, then something must be wrong if the teams *are* communicating. Is the API not good enough? Is the platform not suitable? Is a component missing? If we can achieve low-bandwidth communication—or even zero-bandwidth communication—between teams and still build and release software in a safe, effective, rapid way, then we should. This is visualized in Figure 2.5, which is based on Henrik Kniberg's "Real Life Agile Scaling."[16]

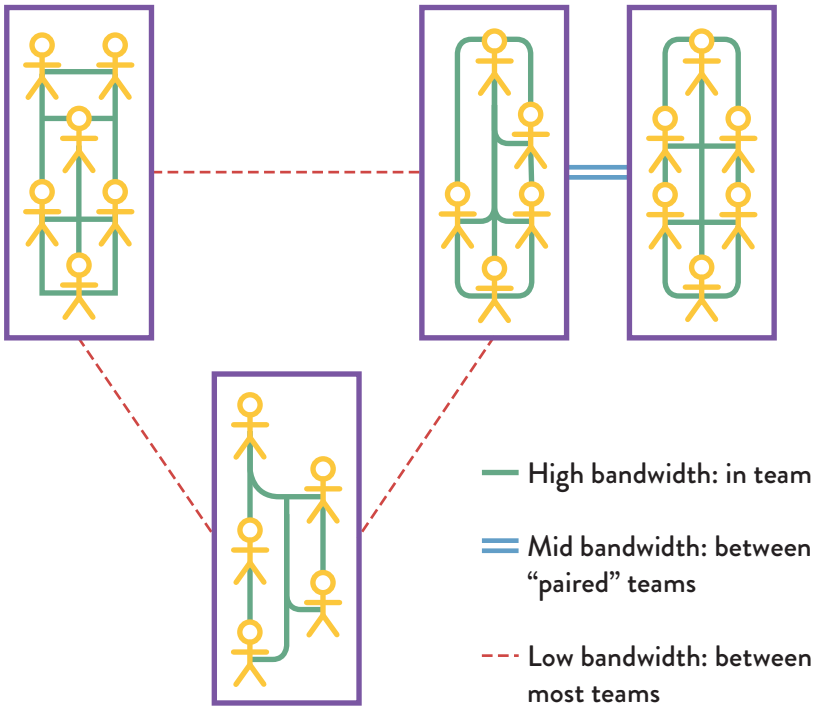


Figure 2.5: Inter-Team Communication
Communication within teams is high bandwidth. Communication between two "paired" teams can be mid bandwidth. Communication between most teams should be low bandwidth.

A simple way to restrict communication is to move two teams to different parts of the office, different floors, or even different buildings. If the teams are virtual or mostly communicate over a chat messenger tool, the volume

and patterns of the team-to-team communications can help identify communications that do not match the interactions expected for the software architecture.

Similarly, if a large team regularly deals with two separate areas of the system, it can be useful to split this team into two smaller teams dedicated to each part, although only if it's the same team members who work on different systems. If the whole team works on more than one part of the system by design (for example, a newer service and an older component), keep the team together. (See Chapter 9 for more on patterns for long-term "continuity of care" for older software systems.)

Sometimes, two or more teams may feel the need to communicate on software purely because the code for their parts of the system is in the same version-control repository or is even part of the same application or service, whereas logically, it should be separate. In these cases, we need to use "fracture plane" patterns (which will be discussed in Chapter 6) to split up the software into smaller chunks that can live in separate repositories.

Everyone Does Not Need to Communicate with Everyone

With open-plan offices and, particularly, with ubiquitous, instant communication via chat tools, anyone can communicate with anyone else. In this situation, one can accidentally fall into a pattern of communication and interaction where everyone *needs to* communicate with everyone else (putting the onus on the consumer to distill what is relevant) in order to get work done. From the viewpoint of Conway's law, this will drive unintended consequences for the software systems, especially a lack of modularity between subsystems.

If the organization has an expectation that "everyone should see every message in the chat" or "everyone needs to attend the massive standup meetings" or "everyone needs to be present in meetings" to approve decisions, then we have an organization design problem. Conway's law suggests that this kind of many-to-many communication will tend to produce monolithic, tangled, highly coupled, interdependent systems that do not support fast flow. More communication is not necessarily a good thing.

Beware: Naive Uses of Conway's Law

There is a danger of misinterpreting Conway's law and creating a set of teams that appear to map well to the required architecture but, in fact, work strongly against fast flow. Furthermore, the relationship between cross-team tools and

communication is often missed or ignored, but such tooling can be a powerful driver of self-similar design. In this section, we identify some potential pitfalls resulting from the naive application of Conway's law.

Tool Choices Drive Communication Patterns

The way in which teams use software communication tools can have a strong influence on communication patterns between teams. A common problem in organizations struggling to build and run modern software systems is a mismatch between the responsibility boundaries for teams or departments and those for tools. Sometimes an organization has multiple tools when a single one would suffice (providing a common, shared view). Other times, a single tool is used and problems arise because teams need separate ones.

As we've seen, Conway's law tells us that an organization is constrained to produce designs that are copies of its communication structures. We therefore need to be mindful of the effect of shared tools on the way teams interact. If we want teams to collaborate, then shared tools make sense. If we need a clear responsibility boundary between teams, then separate tools (or separate instances of the same tool) may be best.

Let's say we need a software development team to work closely with the IT operations team; having separate ticketing or incident-management tools for the two teams will likely result in poor inter-team communication. To help these teams collaborate and communicate, we should choose a tool that can meet the needs of both groups. Similarly, having a special "production only" tool that is limited to teams with security access to production should be avoided. If that tool interacts with or measures the software being built, then the restricted access to the tool is likely to drive a communication gap between teams with access and teams without. The tool can help or hinder communication flow and, therefore, the effective interaction of teams.

TIP

Make information visible while keeping security in place.
Log-aggregation tools provide a simple solution for application teams that need to consult production logs (for debugging purposes, for instance) but do not have access to production environments. Such tools ship all the logs to an external location, where they get processed and indexed together (and anonymized if need be), making it

faster to search and correlate events than individual logs. Teams get access to the information they need while production security controls remain intact (other than ensuring logs are being transferred in a secure fashion).

However, when responsibility boundaries between two teams do *not* overlap (when the teams have very distinct roles without much need to collaborate), we will not get much value from insisting on the same incident-tracking tool or even the same monitoring tool for the two teams, particularly if one of the teams is outside the organization providing a service.

In summary, don't select a single tool for the whole organization without considering team inter-relationships first. Have separate tools for independent teams, and use shared tools for collaborative teams.

Many Different Component Teams

Some organizations have naively used Conway's law to create many different component teams focused on building small parts of systems. Component teams—better called complicated-subsystem teams (see Chapter 5)—are occasionally needed but only for exceptional cases, where very detailed expertise is required. Generally speaking, we need to optimize for fast flow, so stream-aligned teams are preferred. We will cover these aspects more in Chapter 5.

Repeated Reorganizations that Create Fiefdoms or Reduce Headcount

The underlying aim of many "reorganizations" in the past was to reduce staff or create fiefdoms of power for managers and leaders. When we change the organization structure to accommodate Conway's law, we are aiming to improve the space (context, constraints, etc.) in which organizations search for solutions with software systems. These two approaches are mutually exclusive. With software and "product" companies, structure should anticipate product architecture. Combined with a team-first approach, regular reorganizations for management reasons should become a thing of the past.

To put this in the strongest way, regular reorganizations for the sake of management convenience or reducing headcount actively destroy the ability of organizations to build and operate software systems effectively. Reorganiza-

tions that ignore Conway's law, team cognitive load, and related dynamics risk acting like open heart surgery performed by a child: highly destructive.

Summary: Conway's Law Is Critical for Efficient Team Design in Tech

Conway's law tells us that an organization's structure and the actual communication paths between teams persevere in the resulting architecture of the systems built. They void the attempts of designing software as a separate activity from the design of the teams themselves.

The effects of this simple law are far reaching. On one hand, the organization's design limits the number of possible solutions for a given system's architecture. On the other hand, the speed of software delivery is strongly affected by how many team dependencies the organization design instills.

Fast flow requires restricting communication between teams. Team collaboration is important for gray areas of development, where discovery and expertise is needed to make progress. But in areas where execution prevails—not discovery—communication becomes an unnecessary overhead.

One key approach to achieving the software architecture (and associated benefits like speed of delivery or time to recover from failure) is to apply the reverse Conway maneuver: designing teams to match the desired architecture. We provided a simple example where an organization could avoid a monolithic database by embedding database skills in the application team, so that they had sufficient autonomy to maintain a separate data store (perhaps relying on a centralized DBA team for recommendations on database design or synchronization with other databases).

In short, by considering the impact of Conway's law when designing software architectures and/or reorganizing team structures, you will be able to takeadvantage of the isomorphic force at play, which converges the software architecture and the team design.

3 Team-First Thinking

> Disbanding high-performing teams is worse than vandalism: it is corporate psychopathy.
>
> —**Allan Kelly**, *Project Myopia*

Experts in organizational behavior have known for decades that modern complex systems require effective team performance: in particular, Driskell and Salas found that teams working as a cohesive unit perform far better than collections of individuals for knowledge-rich, problem-solving tasks that require high amounts of information.[1] Even previously hierarchical organizations such as the US Army have adopted the team as the fundamental unit of operation. In the bestselling book *Team of Teams*, retired US Army General Stanley McChrystal notes that the best-performing teams "accomplish remarkable feats not simply because of the individual qualifications of their members but because *those members coalesce into a single organism*."[2] (italics added)

In software development specifically, the speed, frequency, complexity, and diversity of changes needed for modern software-rich systems means that teams are essential. Relying on individuals to comprehend and effectively deal with the volume and nature of information required to build and evolve modern software is not sustainable. In fact, research by Google on their own teams found that who is on the team matters less than the team dynamics; and that when it comes to measuring performance, teams matter more than individuals.[3] We must, therefore, start with the team for effective software delivery.

There are multiple aspects to consider and nurture: team size, team lifespan, team relationships, and team cognition.

Use Small, Long-Lived Teams as the Standard

In this book, "team" has a very specific meaning. By team, we mean a stable grouping of five to nine people who work toward a shared goal as a unit. We consider the team to be the smallest entity of delivery within the organization. Therefore, an organization should never assign work to individuals; only to teams. In all aspects of software design, delivery, and operation, we start with the team.

In most organizations, an effective team has a maximum size of around seven to nine people. Amazon, for instance, is known for limiting the size of its software teams to those that can be fed by two pizzas.[4] This limit, recommended by popular frameworks such as Scrum, derives from evolutionary limits on group recognition and trust known as Dunbar's number (after anthropologist Robin Dunbar). Dunbar found fifteen to be the limit of the number of people one person can trust deeply.[5] From those, only around five people can be known and trusted closely.[6]

Allowing teams to grow beyond the magic seven-to-nine size imperils the viability of the software being built by that team, because trust will begin to break down and unsuitable decisions might ensue. Organizations need to maximize trust between people on a team, and that means limiting the number of team members.

When delivering changes rapidly, it is important to ensure that high trust is explicitly valued and designed for. High trust is what enables a team to innovate and experiment. If trust is missing or reduced due to a larger group of people, speed and safety of delivery will suffer.

> ***NOTE***
>
> **High-trust organizations may sustain larger teams.**
> There are exceptions to the seven-to-nine rule, but these are rare. If an organization has engendered a very strong culture of trust, mutual respect, and acceptance of failure, teams might work at up to around fifteen people. However, in our experience, very few organizations fit this criteria.

Smaller Size Fosters Trust

The limit on team size and Dunbar's number extends to groupings of teams, departments, streams of work, lines of business, and so on. In addition to Dunbar's number, anthropological research shows that the type and depth of relationship we can have with people has clear limits:[7]

- Around five people—limit of people with whom we can hold close personal relationships and working memory
- Around fifteen people—limit of people with whom we can experience deep trust
- Around fifty people—limit of people with whom we can have mutual trust
- Around 150 people—limit of people whose capabilities we can remember

Some researchers have identified possible limits to effective social relationships at around 500 and 1,500 (there is roughly a three times multiplier at work here). The key point is that—whether we like it or not—there are natural restrictions on the size of effective groupings within any organization. As the size of a group increases, the dynamics and behaviors between group members will be subtly or radically different, and patterns and rules that worked at a smaller scale will probably fail to work at a larger scale.

Teams need trust to operate effectively, but if the size of a group grows too large for the necessary level of trust, that group can no longer be as effective as it was when it was a smaller unit. Within an organization building and running software systems, it is therefore important to consciously limit the size of team groupings to Dunbar's number to help achieve predictable behavior and interactions from those teams:

- A single team: around five to eight people (based on industry experience)
  - In high-trust organizations: no more than fifteen people
- Families ("tribes"): groupings of teams of no more than fifty people
  - In high-trust organizations: groupings of no more than 150 people
- Divisions/streams/profit & loss (P&L) lines: groupings of no more than 150 or 500 people

Organizations can be composed from Dunbar-compatible groupings of these sizes; when one of the limits is reached, the need to split off another unit as a semi-independent grouping arises. We can visualize this "scaling by Dunbar" as concentric circles of increasingly larger or smaller groups (see Figure 3.1, based on the "onion" concept from James Lewis[8]):

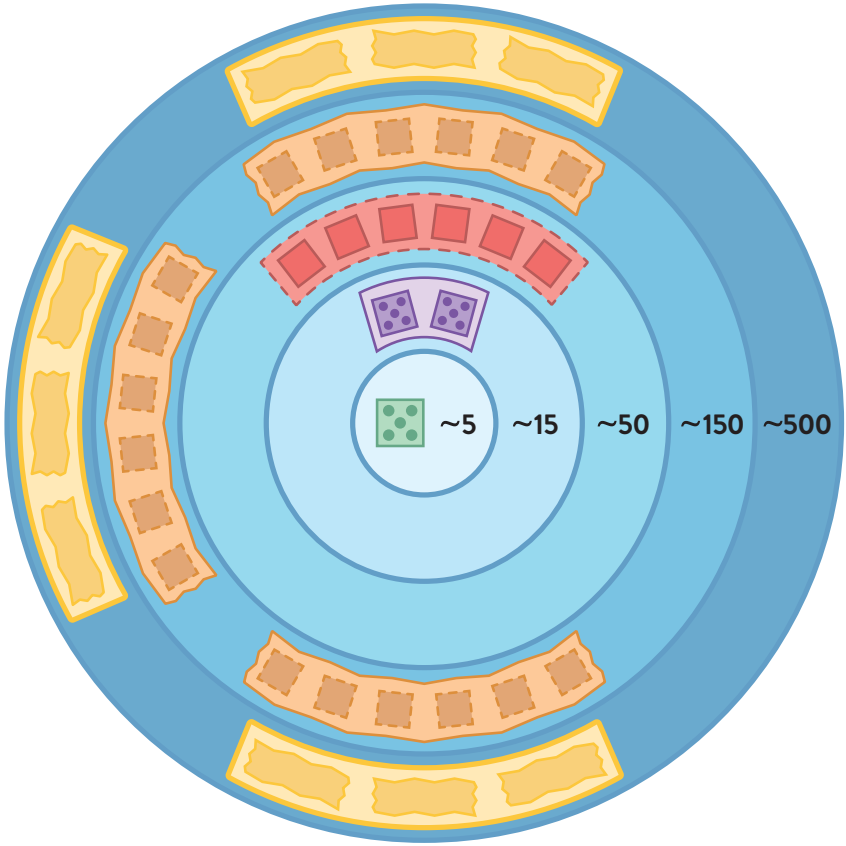


Figure 3.1: Scaling Teams Using Dunbar's Number
Organizational groupings should follow Dunbar's number, beginning with around five people (or eight for software teams), then increasing to around fifteen people, then fifty, then 150, then 500, and so on.

In the context of products and services enabled by software systems, the limits exposed by Dunbar's number mean that the number of people in differ-

ent business lines or streams of work should also explicitly be limited when the number of people in a department exceeds fifty (or 150, or 500), the internal and external dynamics with other groupings will change. This, in turn, means that the software architecture needs to be realigned with the new team groupings so that teams can continue to own the architecture effectively. This is an example of what we like to call "team-first architecture," which requires a substantially new way of thinking for many organizations; but companies like Amazon (with its "two-pizza" rule) have proven it can be a highly successful and scalable approach.[9]

TIP

Team-first software architecture is driven by Dunbar's number. Expect to change the architecture of software systems to fit with the limits on human interactions set by Dunbar's number. Approaches like microservices can help if applied with a team-first perspective.

Work Flows to Long-Lived Teams

Teams take time to form and be effective. Typically, a team can take from two weeks to three months or more to become a cohesive unit. When (or if) a team reaches that special state, it can be many times more effective than individuals alone. If it takes three months for a team to become highly effective, we need to provide stability around and within the team to allow them to reach that level.

There is little value in reassigning people to different teams after a six-month project where the team has just begun to perform well. As Fred Brooks points out in his classic book *The Mythical Man-Month*, adding new people to a team doesn't immediately increase its capacity (this became known as *Brooks's law*).[10] In fact, it quite possibly reduces capacity during an initial stage. There's a ramp-up period necessary to bring people up to speed, but the communication lines inside the team also increase significantly with every new member. Not only that, but there is an emotional adaptation required both from new and old team members in order to understand and accommodate each other's points of view and work habits (the "storming" stage of Tuckman's team-development model).[11]

The best approach to team lifespans is to keep the team stable and "flow the work to the team," as Allan Kelly says in his 2018 book *Project Myopia*.[12]

Teams should be stable but not static, changing only occasionally and when necessary.

In high-trust organizations, people may change teams once a year without major detrimental effects on team performance. For example, at cloud software specialist Pivotal, "an engineer would switch teams about every 9 to 12 months."[13] In typical organizations with lower levels of trust, people should remain in the same team for longer (perhaps eighteen months or two years), and the team should be given coaching to improve and sustain team cohesion.

NOTE

Beyond the Tuckman Teal Performance Model

The Tuckman model describes how teams perform in four stages:

1. Forming: assembling for the first time
2. Storming: working through initial differences in personality and ways of working
3. Norming: evolving standard ways of working together
4. Performing: reaching a state of high effectiveness

However, in recent years, research by people like Pamela Knight has found that this model is not quite accurate, and that storming actually takes places continually throughout the life of the team.[14] Organizations should continually nurture team dynamics to maintain high performance.

The Team Owns the Software

With small, long-lived teams in place, we can begin to improve the ownership of software. Team ownership helps to provide the vital "continuity of care" that modern systems need in order to retain their operability and stay fit for purpose. Team ownership also enables a team to think in multiple "horizons"—from exploration stages to exploitation and execution—to better care for software and its viability. As Jez Humble, Joanne Molesky, and Barry O'Reilly put it in their book *Lean Enterprise*,[15] Horizon 1 covers the immediate future with products and services that will deliver results the same year; Horizon 2 covers the next few periods, with an expanding reach of the products and services; and Horizon 3 covers many months ahead, where experimentation is needed to assess market fit and suitability of new services, products, and features.

The danger of allowing multiple teams to change the same system or subsystem is that no one owns either the changes made or the resulting mess. However, when a single team owns the system or subsystem, and the team has the autonomy to plan their own work, then that team can make sensible decisions about short-term fixes with the knowledge that they will be removing any dirty fixes in the next few weeks. Awareness of and ownership over these different time horizons helps a team care for the code more effectively.

Every part of the software system needs to be owned by exactly one team. This means there should be no shared ownership of components, libraries, or code. Teams may use shared services at runtime, but every running service, application, or subsystem is owned by only one team. Outside teams may submit pull requests or suggestions for change to the owning team, but they cannot make changes themselves. The owning team may even trust another team so much that they grant them access to the code for a period of time, but only the original team retains ownership.

Note that team ownership of code should not be a territorial thing. The team takes responsibility for the code and cares for it, but individual team members should not feel like the code is theirs to the exclusion of others. Instead, teams should view themselves as stewards or caretakers as opposed to private owners. Think of code as gardening, not policing.

Team Members Need a Team-First Mindset

The team should be the fundamental means of delivery rather than the individual. If we follow this team-first approach, we need to ensure that the people *within* our teams also have (or develop) a team-first mindset. This may be unfamiliar to some people, but with the right coaching and time to learn, many people adapt.

For teams to work, team members should put the needs of the team above their own. They should:

- Arrive for stand-ups and meetings on time.
- Keep discussions and investigations on track.
- Encourage a focus on team goals.
- Help unblock other team members before starting on new work.
- Mentor new or less experienced team members.
- Avoid "winning" arguments and, instead, agree to explore options.

However, even with coaching, some people are unsuitable to work on teams or are unwilling to put team needs above their own. Such people can

destroy teamwork and, in extreme cases, destroy teams. These people are "team toxic" and need to be removed before damage is done. There is a good amount of research in this area. For example, one study found that "collectively oriented team members were more likely to attend to the task inputs of other team members and to improve their performance during team interaction than egocentric team members."[16]

Embrace Diversity in Teams

In the context of rapidly changing requirements and technologies, teams must continuously find novel and creative ways to address the challenges placed upon them and to communicate effectively with other teams. Recent research in both civilian and military contexts strongly suggests that teams with members of diverse backgrounds tend to produce more creative solutions more rapidly and tend to be better at empathizing with other teams' needs.[17]

This diverse mix of people also appears to foster better results, as team members make fewer assumptions about the context and needs of their software users. Tom DeMarco and Timothy Lister, authors of the influential book *Peopleware*, observe that "a little bit of heterogeneity can be an enormous aid to create a jelled team."[18] In the context of discovering new possibilities, having a variety of viewpoints and experiences helps teams traverse the landscape of solutions much more rapidly. As Naomi Stanford, author of *Guide to Organisation Design*, puts it: "people and organizations benefit from a diverse workforce where differences spark positive energy."[19]

Reward the Whole Team, Not Individuals

W. Edwards Deming, author of *Out of the Crisis* and a pivotal figure in the Lean manufacturing movement, identified one of his key fourteen points for management as "abolishment of the annual or merit rating and of management by objective."[20] Looking to reward individual performance in modern organizations tends to drive poor results and damages staff behavior. One particularly insidious usage of individual bonuses is when companies use it to leverage their end-of-year profitability. Outstanding individual efforts might receive limited or no bonuses because of a crisis year. This increases the misalignment between the individual's merits and the bonus they actually receive, leading to frustration and demotivation.

With a team-first approach, the whole team is rewarded for their combined effort. One of the defining features of work at technology company Nokia during its hugely successful years in the 1990s and 2000s was: "Pay dif-

ferences across the organization were muted. Bonuses were small and typically paid on a team basis and on overall company performance, not individually."[21]

The same can be applied to training budgets. With a team-first approach, the whole team rather than each individual gets a single training budget. If the team wants to send the same person to six or seven conferences during the year because they are so good at reporting back to the team, that should be the team's decision.

Good Boundaries Minimize Cognitive Load

Having established the team as the fundamental means of delivery, organizations also need to ensure that the cognitive load on a team is not too high. A team working with software systems that require too high of a cognitive load cannot effectively own or safely evolve the software. In this section, we will identify ways in which the cognitive load on teams can be detected and limited in order to safely promote fast flow of change.

Restrict Team Responsibilities to Match Team Cognitive Load

One of the least acknowledged factors that increases friction in modern software delivery is the ever-increasing size and complexity of codebases that teams have to work with. This creates an unbounded cognitive load on teams.

Cognitive load also applies to teams that do less coding and more execution of tasks, like a traditional operations or infrastructure team. They can also suffer from excessive cognitive load in terms of domains of responsibility, number of applications they need to operate, and tools they need to manage.

With a team-first approach, the team's responsibilities are matched to the cognitive load that the team can handle. The positive ripple effect of this can change how teams are designed and how they interact with each other across an organization.

For software-delivery teams, a team-first approach to cognitive load means limiting the size of the software system that a team is expected to work with; that is, organizations should not allow a software subsystem to grow beyond the cognitive load of the team responsible for the software. This has strong and quite radical implications for the shape and architecture of software systems, as we shall see later in the book.

Cognitive load was characterized in 1988 by psychologist John Sweller as "the total amount of mental effort being used in the working memory."[22] Sweller defines three different kinds of cognitive load:

- Intrinsic cognitive load—relates to aspects of the task fundamental to the problem space (e.g., "What is the structure of a Java class?" "How do I create a new method?")
- Extraneous cognitive load—relates to the environment in which the task is being done (e.g., "How do I deploy this component again?" "How do I configure this service?")
- Germane cognitive load—relates to aspects of the task that need special attention for learning or high performance (e.g., "How should this service interact with the ABC service?")

For example, the *intrinsic cognitive load* for a web application developer could be the knowledge of the computer language being used (on top of the fundamentals of programming), the *extraneous cognitive load* might be details of the commands needed to instantiate a dynamic testing environment (which needs multiple hard-to-remember console commands), and the *germane cognitive load* could be the specific aspects of the business domain that the application developer is programming (such as an invoicing system or a video-processing algorithm). Jo Pearce's work on cognitive load in the context of software development provides numerous additional examples.[23]

Broadly speaking, for effective delivery and operations of modern software systems, organizations should attempt to minimize intrinsic cognitive load (through training, good choice of technologies, hiring, pair programming, etc.) and eliminate extraneous cognitive load altogether (boring or superfluous tasks or commands that add little value to retain in the working memory and can often be automated away), leaving more space for germane cognitive load (which is where the "value add" thinking lies).

As we have seen earlier in this chapter, there is an effective maximum size of seven to nine members for a team building and running software systems (see Figure 3.1 on page 34), so it follows that there is a maximum amount of cognitive load that a certain team can deal with. Many organizations do not consider the cognitive load on teams when assigning responsibility for parts of a software system, instead assuming that by adding more teams to the problem, the cognitive load will be shared across the teams. Instead, the teams will suffer from similar communication and interaction strains as mentioned in Brooks's law.

If we stress the team by giving it responsibility for part of the system that is beyond its cognitive load capacity, it ceases to act like a high-performing unit and starts to behave like a loosely associated group of individuals, each trying

to accomplish their individual tasks without the space to consider if those are in the team's best interest.

Limiting the cognitive load for a team means limiting the size of the subsystem or area on which the team works, a tactic suggested by Driskell and colleagues in their research paper: "For those settings in which effective teamwork is critical, it may be necessary to structure the task to make it less demanding (i.e., by delegating subtasks), so that attention can be maintained on essential task and teamwork cues.[24]

At the same time, the team needs the space to continuously try to reduce the amount of intrinsic and extraneous load they currently have to deal with (via training, practice, automation, and any other useful techniques).

Measure the Cognitive Load Using Relative Domain Complexity

A simple and quick way to assess cognitive load is to ask the team, in a non-judgmental way: "Do you feel like you're effective and able to respond in a timely fashion to the work you are asked to do?"

While not an accurate measure, the answer will help gauge whether teams are feeling overloaded. If the answer is clearly negative, organizations can apply some heuristics to understand if and why cognitive load is too high. If it is, the organization needs to take the necessary steps to reduce cognitive load, thus ensuring that the team is able to be effective and proactive again. Incidentally, this will increase motivational levels within the team as members see more value and purpose in their work.

Trying to determine the cognitive load of software using simple measures such as lines of code, number of modules, classes, or methods is misguided. Computer researcher Graylin Jay and colleagues found in 2009 that some programming languages are more verbose than others (and after the emergence of microservices, polyglot systems became increasingly more common), and teams using more abstractions and reusing code will have smaller but not necessarily simpler codebases.[25]

When measuring cognitive load, what we really care about is the domain complexity—how complex is the problem that we're trying to solve with software? A domain is a more largely applicable concept than software size. For example, running and evolving a toolchain to support continuous delivery typically requires a fair amount of tool integration and testing. Some automation code will be needed, but orders of magnitude less than the code needed for building a customer-facing application. Domains help us think across the board and use common heuristics.

While there is no formula for cognitive load, we can assess the number and relative complexity (internal to the organization) of domains for which a given team is responsible. The Engineering Productivity team at OutSystems that we mentioned in Chapter 1 realized that the different domains they were responsible for (build and continuous integration, continuous delivery, test automation, and infrastructure automation) had caused them to become overloaded. The team was constantly faced with too much work and context switching prevailed, with tasks coming in from different product areas simultaneously. There was a general sense in the team that they lacked sufficient domain knowledge, but they had no time to invest in acquiring it. In fact, most of their cognitive load was extraneous, leaving very little capacity for value-add intrinsic or germane cognitive load.

The team made a bold decision to split into microteams, each responsible for a single domain/product area: IDE productivity, platform-server productivity, and infrastructure automation. The two productivity microteams were aligned (and colocated) with the respective product areas (IDE and platform server). Changes that overlapped domains were infrequent; therefore, the previous single-team model was optimizing for the exceptions rather than the rule. With the new structure, the teams collaborated closely (even creating temporary microteams when necessary) on cross-domain issues that required a period of solution discovery but not as a permanent structure.

After only a few months, the results were above their best expectations. Motivation went up as each microteam could now focus on mastering a single domain (plus they didn't have a lead anymore, empowering team decisions). The mission for each team was clear, with less context switching and frequent intra-team communication (thanks to a single shared purpose rather than a collection of purposes). Overall, the flow and quality of the work (in terms of fitness of the solutions for product teams) increased significantly.

Limit the Number and Type of Domains per Team

To be clear, there is no final answer for "Is this the right number and type of domain for this team?" Domains are not static and neither is the team's cognitive capacity. But the reasoning around relative domain complexity can help shape teams' responsibilities and boundaries. When in doubt about the complexity of a domain, always prioritize how the responsible team feels about it. Downplaying the complexity (e.g., "There are plenty of tools for continuous delivery—it's not difficult.") in order to "fit in" more domains with a single team will only lead to failure.

To get started, identify distinct domains that each team has to deal with, and classify these domains into simple (most of the work has a clear path of action), complicated (changes need to be analyzed and might require a few iterations on the solution to get it right), or complex (solutions require a lot of experimentation and discovery). You should finetune the resulting classification by comparing pairs of domains across teams: How does domain A stack against domain B? Do they have similar complexity or is one clearly more complex than the other? Does the current domain classification reflect that?

The first heuristic is to assign each domain to a single team. If a domain is too large for a team, instead of splitting responsibilities of a single domain to multiple teams, first split the domain into subdomains and then assign each new subdomain to a single team. (See Chapter 6 for more help on how to break down large domains.)

The second heuristic is that a single team (considering the golden seven-to-nine team size) should be able to accommodate two to three "simple" domains. Because such domains are quite procedural, the cost of context switching between domains is more bearable, as responses are more mechanical. In this context, a simple domain for a team might be an older software system that has only minor, occasional, straightforward changes. However, there is a risk here of diminishing team members' motivation due to the more routine nature of their work.

The third heuristic is that a team responsible for a complex domain should not have any more domains assigned to them—not even a simple one. This is due to the cost of disrupting the flow of work (solving complex problems takes time and focus) and prioritization (there will be a tendency to resolve the simple, predictable problems as soon as they come in, causing further delays in the resolution of complex problems, which are often the most important for the business).

The last heuristic is to avoid a single team responsible for two complicated domains. This might seem feasible with a larger team of eight or nine people, but in practice, the team will behave as two subteams (one for each domain), yet everyone will be expected to know about both domains, which increases cognitive load and cost of coordination. Instead, it's best to split the team into two separate teams of five people (by recruiting one or two more team members), so they can each be more focused and autonomous. (See Figure 3.2 on page 44.)

As always, these are only recommendations, not a definitive path to success. Use these guidelines as a starting point from which to adapt as your

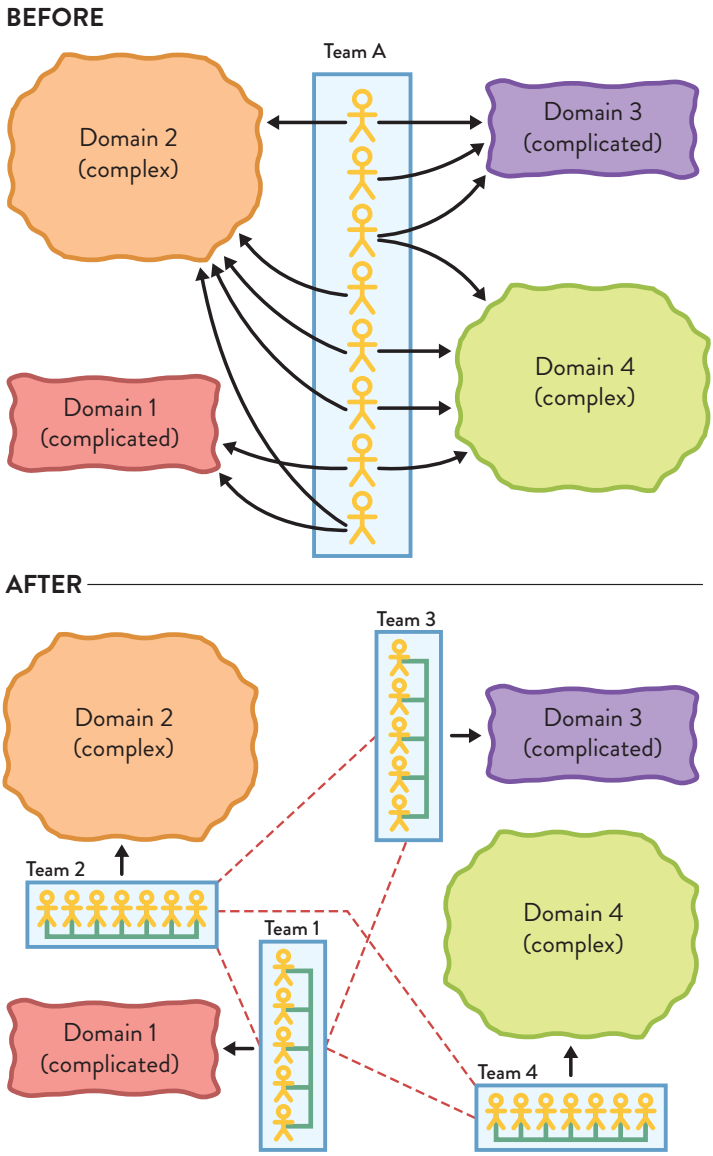


Figure 3.2: No More than One Complicated or Complex Domain per Team
Before: a larger team is spread thin across four domains (two complicated and two complex) and struggles to perform well. Intra-team morale is negatively affected, with frequent context switches and individual disengagement. After: with multiple smaller teams each focusing on a single domain, motivation rises and the team delivers faster and more predictably. Low bandwidth inter-team collaboration allows solving occasional issues affecting two or more domains.

organization evolves and learns. Always remember that, in the end, even if the allocation of domains seems to make sense, if the teams doing the work are still feeling overwhelmed, stress builds up and morale weakens, leading to poor results.

Match Software Boundary Size to Team Cognitive Load

To keep software delivery teams effective and able to own and evolve parts of the software systems, we need to take a team-first approach to the size of software subsystems and the placement of boundaries. Instead of designing a

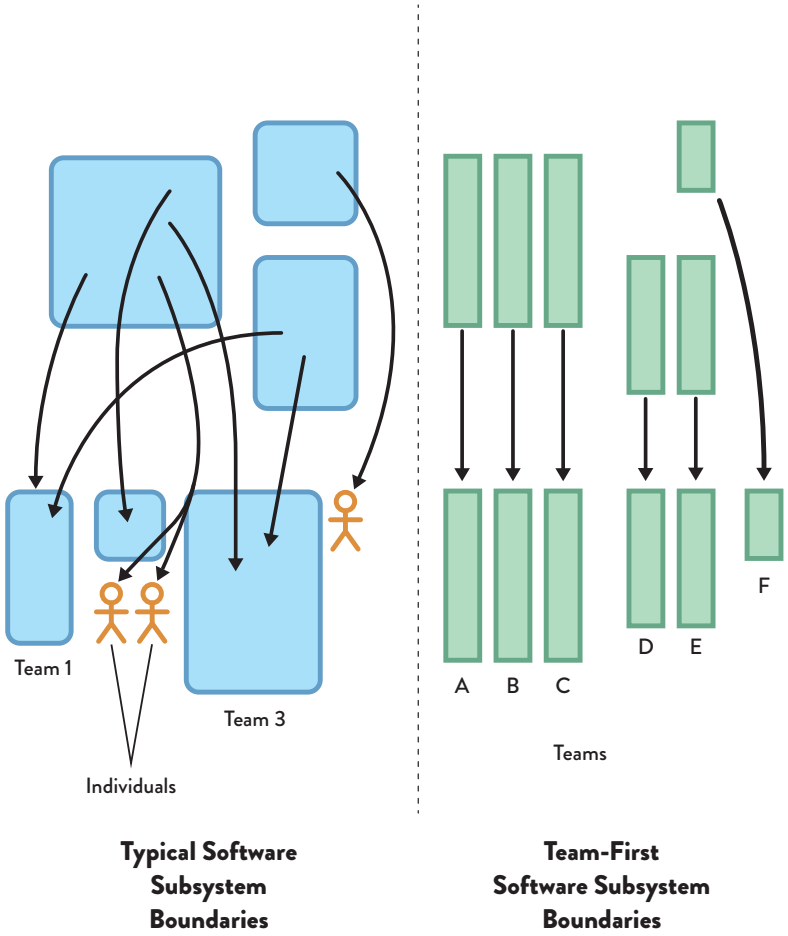


Figure 3.3: Typical vs. Team-First Software Subsystem Boundaries

system in the abstract, we need to design the system and its software boundaries to fit the available cognitive load within delivery teams.

Instead of choosing between a monolithic architecture or a microservices architecture, design the software to fit the maximum team cognitive load. Only then can we hope to achieve sustainable, safe, rapid software delivery. This team-first approach to software boundaries leads to favoring certain styles of software architecture, such as small, decoupled services. We can visualize this team-first approach to software subsystem boundaries in Figure 3.3 (see page 45).

On the left, we see typical software subsystem boundaries, with different parts of systems or products assigned to a mix of multiple teams, single teams, and individuals. On the right, we see the Team Topologies' team-first approach to software subsystem boundaries, with every part of the system being team sized and owned by one team.

To increase the size of a software subsystem or domain for which a team is responsible, tune the ecosystem in which the team works in order to maximize the cognitive capacity of the team (by reducing the intrinsic and extraneous types of load):

- Provide a team-first working environment (physical or virtual). (You'll see more later in this chapter).
- Minimize team distractions during the workweek by limiting meetings, reducing emails, assigning a dedicated team or person to support queries, and so forth.
- Change the management style by communicating goals and outcomes rather than obsessing over the "how," what McChrystal calls "Eyes On, Hands Off" in *Team of Teams*.[26]
- Increase the quality of developer experience (DevEx) for other teams using your team's code and APIs through good documentation, consistency, good UX, and other DevEx practices.
- Use a platform that is explicitly designed to reduce cognitive load for teams building software on top of it.

By actively reducing extraneous mental overheads for teams and team members through these and similar approaches, organizations can give teams more cognitive space to take on more challenging parts of the software systems. Conversely, if an organization does not have team-first office space, good management practices, and especially a team-first platform, then the size of software subsystems that teams can take on will be smaller. A larger number

of smaller parts requires more teams to work on them, costing more. Taking a team-first approach to software subsystem boundaries by designing for cognitive load means happier teams and (eventually) lower costs.

Albert Bertilsson, Solution Team Lead, and Gustaf Nilsson Kotte, Web Developer, felt the weight of a continuously increasing cognitive load on the mobile team they were leading at IKEA back in 2017. As they relayed to us, in the previous year, the team kept growing as a result of successful delivery of multiple projects in a short period of time and across multiple markets.

This high-performing team kept adding more and more responsibilities on their shoulders, as the number of software products they maintained kept increasing. Eventually, they started to run into problems due to some work streams preventing the releases of others. Despite understandable pushback from the team, Bertilsson and Kotte managed to convince team members that they really had two products in the same codebase and needed to split the team in two, following Conway's law. An interesting bit to retain here is that this was a high-performing team with all the intrinsic motivators (autonomy, mastery, and purpose), yet they were still feeling the pains of cognitive overload.

A further benefit of taking a team-first approach to software boundaries is that the team tends to easily develop a shared mental model of the software being worked on. Research has shown that the similarity of team mental models is a good predictor of team performance, meaning fewer mistakes, more coherent code, and more rapid delivery of outcomes.[27] As we begin to optimize more and more for the team, the benefits begin to compound in a positive way.

TIP

"Minimize cognitive load for others" is one of the most useful heuristics for good software development.

Design "Team APIs" and Facilitate Team Interactions

Now that we see the team as the fundamental means of delivery, we can begin to design other things around the team. In this section, we explore concepts such as the team API and well-defined team interactions as ways to produce a coherent, dynamic network of cleanly communicating teams.

Define "Team APIs" that Include Code, Documentation, and User Experience

With stable, long-lived teams that own specific bits of the software systems, we can begin to build a stable *team API*: an API surrounding each team. An API (application programming interface) is a description and specification for how to interact programmatically with software, so we extend this idea to entire interactions with the team. The team API includes:

- Code: runtime endpoints, libraries, clients, UI, etc. produced by the team
- Versioning: how the team communicates changes to its code and services (e.g., using semantic versioning [SemVer] as a "team promise" not to break things)
- Wiki and documentation: especially how-to guides for the software owned by the team
- Practices and principles: the team's preferred ways of working
- Communication: the team's approach to remote communication tools, such as chat tools and video conferencing
- Work information: what the team is working on now, what's coming next, and overall priorities in the short to medium term
- Other: anything else that other teams need to use to interact with the team

The team API should explicitly consider usability by other teams: Will other teams find it easy and straightforward to interact with us, or will it be difficult and confusing? How easy will it be for a new team to get on board with our code and working practices? How do we respond to pull requests and other suggestions from other teams? Is our team backlog and product roadmap easily visible and understandable by other teams?

For effective team-first ownership of software, teams need to continuously define, advertise, test, and evolve their team API to ensure that it is fit for purpose for the consumers of that API: other teams. In *Dynamic Reteaming* (by Heidi Helfand), Evan Wiley, Director of Program Management at Pivotal Cloud Foundry (PCF), a major enterprise Platform-as-a-Service (PaaS) provider, describes how more than fifty teams are seen at PCF:

> We really try to maintain as much contract based, *API-based separation of concerns between teams* [emphasis added] as we can. We try not to share

> code bases between teams. All the git repos for a particular team's feature are wholly owned by that team and if another team is going to make an addition or change to that code base, they'll either do it with a pull request or through cross-team pairing, where we would kind of send one half of a pair over to the dependency holding team and one half of that team's pair back to the upstream team to work on that feature.[28]

An even more stringent team API approach is taken at cloud vendor AWS, where CEO Jeff Bezos insisted on almost paranoid levels of separation between teams. For example, each team at AWS must assume that "every [other team] becomes a potential DOS [denial of service] attacker requiring service levels, quotas, and throttling."[29]

Many of the behaviors and patterns that make a good team API also make for a good platform and good team interactions in general. (See Chapter 5 for more details about what makes a good platform, and Chapter 7 for details about promise theory, a team-based approach to cooperation in sociotechnical systems.)

Facilitate Team Interactions for Trust, Awareness, and Learning

It is important to provide time, space, and money to enable and encourage people from different teams with similar skills and expertise to come together to learn from each other and to develop their professional competencies.

By explicitly setting aside time and space for teams and people to intercommunicate and learn, organizations can make learning and trust building part of the rhythm that facilitates effective team interactions. Two critical ways this can help teams build trust and awareness and learn new things are: (1) a consciously designed physical and virtual environment; and (2) time away from desks at guilds, communities of practice (a group of people who regularly get together on a voluntary basis to collectively learn and share knowledge about a domain of interest, internal tech conferences, etc.

Because this team interaction is outside the everyday building and running of the main software systems, Conway's law plays a much less obvious role, and a freer cross-association between teams can take place. Crucially, teams that have a chance to rehearse their team interactions in these contexts tend to find it easier to interact with other teams when building and running software systems, as found in the groundbreaking research by Robert Axelrod and author Mark Burgess.[30]

Explicitly Design the Physical and Virtual Environments to Help Team Interactions

Consciously designed physical and virtual environments are necessary for teams to learn and build trust. However, different people need different environments at different times to be productive. Some tasks (e.g., implementing and testing a complicated algorithm) might require full concentration and low levels of noise. Other tasks require a very collaborative approach (e.g., defining user stories and acceptance criteria). People who work all day with headphones on are seen as anti-social, and their behavior does not promote interaction and collaboration; but it could well be that the office environment is generally noisy and these people require a quiet environment to be effective.

Neither individual cubicles nor fully open-plan seating is generally suitable for teams: we need something better. Teams need the ability to collaborate frequently, internally and only occasionally externally (with other teams). This balance is hard to achieve both in an open-plan layout (no dedicated work area for the team) and in an individual-workspaces layout (time together needs to be planned ahead of time and meeting rooms are often scarce). Spotify recognized this early on in their growth and arranged their office space to support both needs.[31] Back in 2012, Henrik Kniberg and Anders Ivarsson—then working at Spotify—talked about how "squads in a tribe are all physically in the same office, normally right next to each other, and the lounge areas nearby promote collaboration between the squads."[32]

Office design for effective software delivery should accommodate all of the following modes of work: focused individual work, collaborative intra-team work, and collaborative inter-team work.

Having workspaces that clearly indicate the type of work going on also helps reduce disturbance and unnecessary interruptions.

CASE STUDY: TEAM-FOCUSED OFFICE SPACE AT CDL

Michael Lambert, Head of Development, CDL
Andy Rubio, Development Team Leader, CDL

CDL is a UK-based company that is a market leader in the highly competitive retail-insurance sector.

Here at CDL, our Agile journey has seen us evolve in many ways. One aspect many people are interested in is how we organize the working environment for our teams. From the start, we have always had the

luxury of being able to colocate our Agile teams. After moving to new offices and then quickly outgrowing them, we moved many of our development project teams back to our old headquarters, which gave us multiple small project rooms where a development team could set up home. We liked the space and ownership this brought, but cross-team communication and visibility of other teams was less optimal. When our new home, "The Codeworks," was built, we thought long and hard about what the layout of the development areas should be.

We visualized everything, so lots of magnetic whiteboards [were] essential. We liked the team space our old building gave us, but we needed less isolation of teams, and we had the usual physical numbers and space constraints. If teams did not have enough space or only had small cubical clusters or tight horseshoe arrangements, then availability of meeting rooms for team ceremonies would become a big problem. Ideally, we wanted both: team space for the team to get their stuff done and openness for the teams to collaborate and share.

What we came up with was a "benched bay" approach, with one long bench for each team, and each bench was flanked by whiteboard partitions. Where a team butted up to an end wall, we painted it with smart-surface paint so we could draw on it (see Figure 3.4 on page 52).

The size and growth of teams is also an important factor in design. Some teams may be smaller while others may need to grow fast. The bench arrangement allowed for easy growth, especially if you haven't got supporting legs and pedestals in the way. Small teams could spread out while growing teams could squeeze up a bit. Of course, there is a limit on this. When the team is too big, we split it into two smaller teams, each taking functionally half of the backlog to make their own. The beauty of this is each team takes the culture of the old team with them, and they will diverge and grow themselves over time; but you can (with luck!) skip the "storming" and "norming" phases of starting a team from scratch. We deliberately have differing sizes of bays, where an extra table or two can be accommodated.

Initially, team benches were set centrally and symmetrically between the dividing whiteboard partitions, but we soon realized that an asymmetrical arrangement worked much better, where the bench was closer to one partition. This provided more space on one side to gather the team yet still allowed the opposite whiteboards to be used effectively.

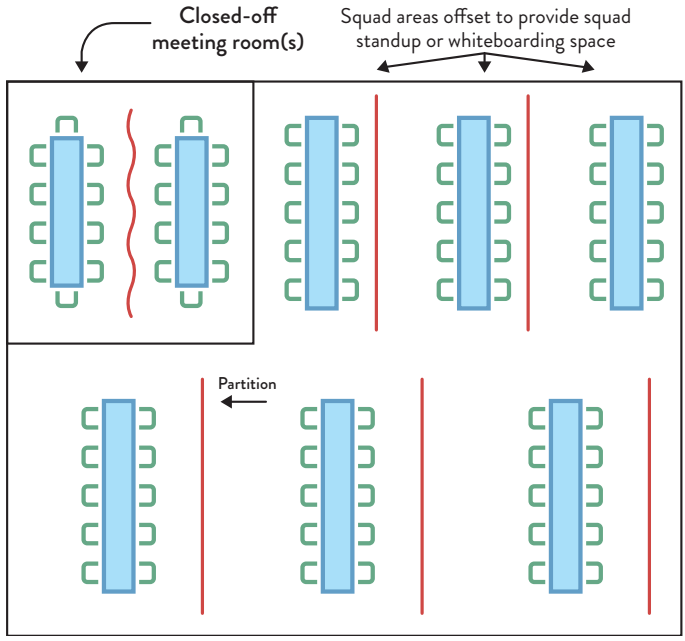


Figure 3.4: Office Layout at CDL

We used what we had learned from this arrangement when it came to fitting out the top floor for our new digital teams. Our original partitions were expensive, heavyweight structures that could only be moved at some expense. For the new digital space, we opted for lots of large, portable, but still substantial, whiteboards. Teams could now reposition and make breaks as they organized themselves.

This design is by no means perfect. All spaces are compromised in one way or another. We get things wrong, but we continue to learn and adapt. One such experiment was to remove the small glass partitions running down the center of the team benches. Another was to have height-adjustable sections on the ends of each set of benches for standing or for people who needed extra legroom.

As the case study from CDL shows, the physical work environment has a significant effect on the ability of teams to interact in useful ways. Successful organizations make sure to spend time and money achieving a good physical environment for their staff.

For example, the bank ING Netherlands explicitly redesigned its office space as part of a major organizational change around 2015 to align teams to value streams.[33] At ING, several stream-aligned "squads" working on similar products and services within a stream form a "tribe." Each tribe has a separate area within the office, including multiple team-sized spaces, one for each squad. The thought-out design of the office layout means that people from other squads or tribes can easily recognize aspects of other teams' work (such as kanban boards, WIP limits, status radiators, and so on) and rapidly learn new approaches. Some organizations have taken this even further, aligning entire floors of their office space to separate business streams, promoting high flow and easier collaboration within a stream.

Jeremy Brown from Red Hat Open Innovation Labs told us how they had everything on wheels (even plants!) in order to frequently reconfigure their physical environment for different types of work, and for teams to emerge and evolve their own space.[34] In their 2012 book *Make Space*, Scott Doorley and Scott Witthoft present many other creative ideas for arranging physical space in ways that ignite creativity and useful team interactions.[35]

CASE STUDY: STREAM-ALIGNED OFFICE LAYOUT FOR FLOW-BASED COLLABORATION AT AUTO TRADER

Dave Whyte, Operations Engineering Lead, Auto Trader
Andy Humphrey, Head of Customer Operations, Auto Trader

Back in 2013, as we started to move from a print-based business with many different offices around the country to a 100% digital business, we began to look at ways we could improve collaboration and optimize for the flow of work. We reorganized from fifteen offices into three, with our main office in Manchester, UK, on only two floors. The working environment was created to be as open plan as possible, with all senior managers sitting with their teams and no private offices. This made it much easier for people to communicate with each other, and we finally started bridging the gap between "the business" and IT.

Our new offices were built for collaboration, from the way the desks could be laid out to the limits on the number of monitor screens that one person could have at their desk (to avoid people "hiding" behind screens). Over the past few years, we have experimented with different

office layouts and seating plans to help the right teams communicate and to promote flow:

- Organizing technical and non-technical teams on the same floors and in the same areas: This helped break down barriers between departments that shared the same goals and customers. The equipment given to sales, product, service, and technology became more aligned so that we could share tools more widely and work in the same way (e.g., all our sales and service colleagues have laptops; you don't have to be a rockstar developer to get a MacBook anymore).
- Clear-desk policy: We provided lockers for personal belongings and encouraged people to move around the office and sit where they needed to be that day in order to add value and not be limited to sitting at the same desk in the same team.
- Technology restrictions: The desks were designed with single monitors so that people could see those sitting opposite them and interact more freely. It was common for some technical staff to have two or three monitors, so this was not popular; but it's an interesting example of becoming a digital organization by actually restricting the use of some technology in order to meet the goal of being more collaborative. The desks even had recessed legs, creating a bench effect, so that people could move between them without snagging [their] legs—helping pairing and sitting with other people.
- Writable walls: To encourage more informal, creative conversations, the walls were made writable so that people could draw as they discussed, whether they were in a corridor or next to a car. Most meeting rooms were made of glass so that people could see who was in there and work out if they needed to be in there too. We also created more informal meeting spaces—sofas, soft chairs, etc.—so that people could sit down for a chat with a colleague without needing to plan a meeting room in advance.
- Event spaces: We also have event spaces designed into all our buildings, so we can get together as a company and even invite our local community by hosting events and meetups that help us get to know and work with people outside our organization.

> We now have all the people in a certain business division sitting together. For example, private advertising is one of our business areas, handling vehicle sales by private individuals, and everyone involved in this stream of business sits on the same floor: marketing people, sales people, developers, testers, product managers, and so on. This means that everyone in the same business stream can "feel the pain" together and all decisions are more jointly owned. We have found that you start seeing things from other people's viewpoints when you sit with them.
>
> Our office layout is quite deliberately designed to help flow and specific collaboration. We based our teams loosely on the model from Spotify, so we have squads of around eight people that build specific parts of a system, and collections of squads known as tribes. Each squad has its own team area located close to other squad areas from the same tribe. This enables squads from the same tribe to talk easily to each other—collaborating on similar parts of the system—while being physically separated from other tribes by walls and floors.
>
> This layout helps teams focus on their business stream area, minimizing the need to talk with teams from other business areas to get their day-to-day work done. We bring teams together for cross-tribe learning by holding regular guild learning sessions and evening meetups.

The virtual environment is increasingly important as many organizations adopt a remote-first policy. The virtual environment comprises digital spaces such as a wiki, internal and external blogs and organization websites, chat tools, work tracking systems, and so forth. Effective remote work goes beyond having the necessary tools; teams need to agree on ground rules around working hours, response times, video conferencing, tone of communication, and other practical aspects that, if underestimated, can make or break a distributed team, even when all the right tools are available. In their 2013 book *Remote: Office Not Required*, Jason Fried and David Heinemeir Hansson go through how to address these and many other important aspects for remote teams.[36]

From an efficient-communication perspective, the virtual environment should be easy to navigate, guiding people to the right answer quickly. In particular, chat tools should have channel names or space names that are easy to predict and search for, with prefixes to group chats:

#deploy-pre-production
...
#practices-engineering
#practices-testing
...
#support-environments
#support-logging
#support-onboarding
...
#team-vesuvius
#team-kilimanjaro
#team-krakatoa

In a virtual environment, it can be useful to use naming conventions in usernames to make it easy for people to identify who's in a particular team, especially if that team is a central X-as-a-Service team, providing a platform or component (more on this in Chapter 5). Instead of simply "Jai Kale" as the display name within the chat tool and wiki, use something like "[Platform] Jai Kale" to identify that Jai Kale is in the platform team.

Warning: Engineering Practices Are Foundational

At the end of the day, technology teams need to invest in proven team practices like continuous delivery, test-first development, and a focus on software operability and releasability. Without them, all the effort invested in a team-first approach to work and flow will be greatly undermined or at least underachieved.

Continuous delivery practices support hypothesis-driven development and automation, operability practices provide early and ongoing operational checks and discovery, testability practices and test-first development enhance the design and fitness for purpose of solutions, and releasability practices ensure delivery pipelines are treated as a first-grade product. All of them are critical for fast flow and require an ongoing effort by all engineering teams.

Summary: Limit Teams' Cognitive Load and Facilitate Team Interactions to Go Faster

In a fast-changing and challenging context, teams are more effective than groups of individuals. Successful organizations—from the US military to

corporations large and small—treat the team as the fundamental means of getting work done. Teams are generally small, stable, and long lived, allowing team members the time and space to develop their working patterns and team dynamics.

Importantly, due to limits on team size (Dunbar's number), there is an effective upper limit on the cognitive load that a single team can bear. This strongly suggests a limit on the size of the software systems and complexity of domains that any team should work with. The team needs to own the system or subsystems they are responsible for. Teams working on multiple codebases lack ownership and, especially, the mental space to understand and keep the corresponding systems healthy.

The team-first approach provides opportunities for many kinds of people to thrive in an organization. Instead of needing a thick skin or resilience in order to survive in an organization that atomizes individuals, people in a team-first organization have the space and support to develop their skills and practices within the context of a team.

Crucially, because communication between individuals is de-emphasized in favor of communication between teams for day-to-day work, the organization supports a wide range of communication preferences, from those people who communicate best one to one to those who like large group conversations. Furthermore, the effect of previously destructive individuals is curtailed. This humanistic approach is a huge benefit of choosing teams first.

Want more?

Access all downloads & extra materials at itrevoluton.com/team-topologies

CHAPTER 4

- Ad hoc or constantly changing team design slows down software delivery.
- There is no single definitive team topology but several inadequate topologies for any one organization.
- Technical and cultural maturity, org scale, and engineering discipline are critical aspects when considering which topology to adopt.
- In particular, the feature-team/product-team pattern is powerful but only works with a supportive surrounding environment.
- Splitting a team's responsibilities can break down silos and empower other teams.

CHAPTER 5

- The four fundamental team topologies simplify modern software team interactions.
- Mapping common industry team types to the fundamental topologies sets up organizations for success, removing gray areas of ownership and overloaded/underloaded teams.
- The main topology is (business) stream-aligned; all other topologies support this type.
- The other topologies are enabling, complicated-subsystems, and platform.
- The topologies are often "fractal" (self-similar) at large scale: teams of teams.

CHAPTER 6

- Choose software boundaries using a team-first approach.
- Beware of hidden monoliths and coupling in the software-delivery chain.
- Use software boundaries defined by business-domain bounded contexts.
- Consider alternative software boundaries when necessary and suitable.