the Tao *of* Microservices

Richard Rodger

MANNING

SAMPLE CHAPTER

The Tao of Microservices

by Richard Rodger

Chapter 8

brief contents

8 People

In this chapter

- Navigating the politics of large organizations
- Accommodating stakeholder needs
- Understanding the new politics created by microservices
- Organizing developers and teams for microservices
- Challenging the rituals of orthodox software development

Software development is a human activity, and that fact has a massive impact on the outcomes of software development projects. This book communicates a strong message: that the engineering problems of software development have for too long been neglected in favor of meandering arguments about process. Microservices are effective because they address the engineering problems. However, this book doesn't claim that microservices by themselves deliver great projects. They must be implemented by people, and you can't ignore the human factor. Nor can you ignore the organizations where those people work. The behaviors of organizations are emergent properties of human nature and will definitely be your concern as an architect.

8.1 Dealing with institutional politics

Software development and institutional politics are both detail-oriented activities. As a software architect, you're probably better at politics than you think.

This book doesn't give you grand strategies for the victorious delivery of successful software projects; rather, you must prepare for an unending war of attrition, using many tactics on a daily basis to slowly change the organization and create an environment in which you can succeed. Your secret weapon is the technical efficiency of the microservice approach, compared to the current state of the art for software architectures. But that by itself isn't enough. You must be prepared to work the system, build trust, develop alliances and support, win over skeptics, neutralize foes, and apply many disparate minor weapons. Accepting this fact is the first step on the road to success. Every organization is unhappy in its own way, so you must also contextualize the tactics discussed in the following sections.

8.1.1 Accepting hard constraints

Every organization contains entrenched power centers that impose constraints on your working practices. Sometimes, you'll have enough political capital to change these constraints, but often, you won't. The ambitious vice president who allocated the budget for your project may not be prepared to go all in and may hedge on contentious interactions with other parts of the organization. Don't be surprised by this. Even if you have the political capital to break a hard constraint, consider spending it elsewhere, because you'll have many important battles to fight.

To handle hard constraints, you first need to enumerate them. Take time at the start of the project to understand what you can and can't do. It's a serious error to make assumptions or to take assurances at face value. Even if you're told that you'll be able to do something, push deeper, and verify. Ask for policy documents, and talk to lower-level staff in the relevant department. Make sure of the facts. Then, put your understanding in writing, and copy all relevant parties.

When you've identified the hard constraints, bring them into the light. Document them, make them part of the project, and indicate how you'll work with and around them, and the impact they'll have on the project. Use them to redefine the interpretation of success.

If you're lucky, something wonderful will happen: constraints that are hard at the beginning of the project can soften when you build trust and confidence. Identifying constraints early gives you the ability to keep highlighting them and discussing their impact with stakeholders, which creates opportunities to defeat them. This is ugly work, but it's necessary.

8.1.2 Finding sponsors

Who is the primary sponsor of your project? Is it that ambitious vice president or a mid-level manager? Do they understand and buy into the microservice approach, or is their sponsorship based on personal affinity for you? Are *you* the sponsor?

Just because you have project approval doesn't mean you'll get the resources you need to be successful. Nor does it mean other parts of the organization will get out of your way. Finding and working with sponsors is another ugly piece of work, but you're introducing a new approach to your organization, so you need to do it—because you're vulnerable. Minor issues can be magnified by your foes. And sometimes, projects fail because the benign indifference of the universe works that way. To help avoid these issues, you'll need sponsorship.

Nurturing sponsors is an ongoing task for you as the architect of the system. You need to constantly strengthen existing sponsors and find new ones. How? Existing sponsors need care and feeding—that means giving them information. Ask them what format they prefer: a weekly meeting, a status-update email, a dashboard, and so on. Make sure you keep your sponsors up to date, and never let them be caught by surprise because you didn't provide them with the information they needed.

This is easy advice to give, but it's difficult to do in practice. When you're busy on a project, it's probably in your nature as a software developer to focus on the code and put in long hours solving technical problems. You prioritize effort, and things like weekly update emails get pushed to the side. Don't do this. Prioritizing your sponsors is one of the best ways to ensure project success.

How do you find new sponsors? Take every opportunity to network internally. The sponsors you need are those with real power in the organization—and these people aren't necessarily the ones with titles. Identify people who are early employees, who've been in the organization for years, who have external credibility, or who are charming and influential. Go where they are, and meet them. Attend business events, such as internal talks given by guest speakers. You'll find the right kinds of sponsors at these events, because influential people invest time in building their networks. You should, too.

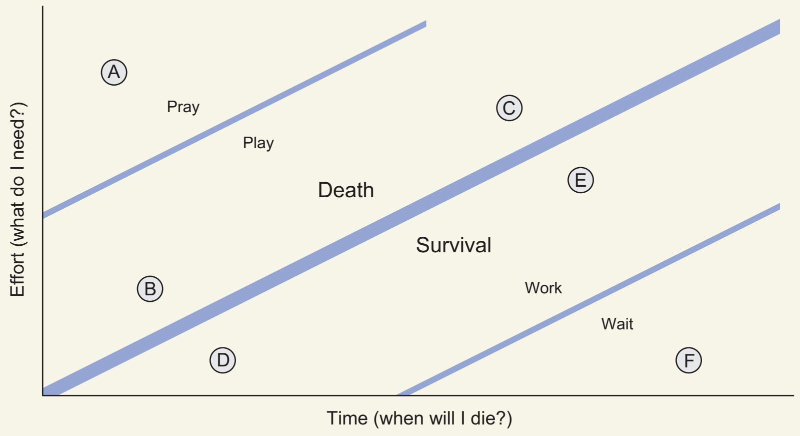
Again, you may find it difficult to invest time in this activity when you get busy. This is a mistake. Networking always pays off; many a project has been saved by a chance encounter with the right person.

How to decide what to do next

There are many detailed, complex ways to decide what to work on next, and you can spend a lot of time building prioritization matrixes for this purpose. But you probably already have a reasonably rough idea of what's important. Here's a quick way to communicate your thinking to your team and thereby validate your analysis.

Start by listing your known deliverables and their deadlines. Then, estimate the full effort to deliver each one, using a comfortable metric for estimation, such as developer days. These are rough estimates, and it's more important that they be accurate relative to each other than accurate in an absolute sense. Now, chart this data, as shown here.

(continued)



Survival chart

The bold line upward from the origin (Death/Survival) is the maximum amount of effort you can exert in a given time: the line of the possible. Things above the line are impossible; you'll definitely fail to meet those deadlines, because you don't have enough resources in the given time frame. This is the Death zone. Below the line, you have enough resources and time. This is the Survival zone.[1]

The Death zone has two subdivisions: Pray and Play. The Pray zone contains problems so bad you might as well ignore them (A) or else start praying, depending on your world view. The descent phase of a thermonuclear warhead aimed at your location would be a good example. The Play zone contains problems that can only be solved by playing politics (B). Reducing the feature set sufficiently will move the problem vertically down below the line of the possible, so that you have enough resources to deliver it. Extend the deadline, and the problem moves to the right, hopefully crossing the line at some point.

Deliverables in the left half of the Play zone are your immediate political priority. In terms of the chart, spend political capital on B, not C (deliverables that can wait awhile).

The Survival zone also has two subdivisions: Work and Wait. The Work zone is where you achieve feasible results. Deliverables in the left half of the Work zone are your immediate development priority; work on D before E. The Wait zone contains deliverables that require few resources and are far in the future (F). Don't spend time on these yet.

[1] The deadlines are charted independently of each other. You deliberately don't take into account cumulative resource expenditure. To do that, remove the highest-priority item, and rechart. But really, this is about identifying the single most important thing to do next.

(continued)

It's a useful exercise to rebuild the chart at the start of each iteration. Engineers are prone to work on items in the Pray and Wait zones, because they're the most, and least, challenging. This is a mistake, because it's the least optimal use of resources.

There's an obvious mapping of microservices to deliverables, so this analysis can help your team decide which microservices need to be built next.[2]

8.1.3 Building alliances

In every job I've ever had, there are two key people I like to get friendly with as soon as I can: the system and office administrators. They're often busy and underappreciated, yet hard-working and competent. They also have deep knowledge of the organization. Align yourself with these key individuals, and your life will be much easier.

It's enough to show them that you understand the challenges of their world. Recognition is a powerful and ethical political act, and people appreciate it. It's obvious that you'll need to spend time building alliances with senior people, and everybody will be aware that you're doing that. But alliances with people on the ground are valuable too and can help you solve different sorts of problems. Every time you solve a problem using a senior ally's authority, you burn political capital in a public way and create enemies. It's much better to receive voluntary help from others. Project success can also depend on gatekeepers looking the other way when you need them to.

You'll need to build traditional alliances as well, but you knew that already. When you're building a microservice architecture for the first time, you need to find a way to work with the operations and IT groups in your company. Of all the groups that can stymie your efforts, these are the most dangerous. You're the representative of many bad things: DevOps culture, the move to the cloud, heterogeneous environments,[3] pager calls at 4:00 a.m., compliance breaches, security headaches, and so on. Operations and IT have every reason to be wary. You won't solve this one overnight, but you must open a dialogue and try to create personal connections that can help overcome the inevitable conflicts.

It may be tempting to use senior allies to override the sysadmins. This must be a last resort. You'll pay dearly if you go to war with them.

8.1.4 Value-focused delivery

I'm repeating this tactic because it's that important. You must get away from the drudgery of feature delivery and arguing over bugs versus enhancements—you'll lose that battle every time. Business value isn't an abstract concept; it's shorthand for identifying the measurements that most closely track the business goals your leadership

[2] This style of thinking is very much inspired by Andy Weir's *The Martian* (Crown, 2014) and Neal Stephenson's *Seveneves* (William Morrow, 2015). Consider both essential reading for the aspiring software architect.

[3] No developer should ever need administrator access to their own machine. The very thought!

cares about. Get all those involved to agree to measure, get agreement about the measurements, and then track them aggressively. Doing so will keep everyone honest.

In every situation, ask, “Does this improve the numbers?” This will give you objective criteria for making and evaluating decisions. It will also protect you from higher-ranking colleagues who are defending their territories. This tactic is the one to fight for and on which you should expend political capital. Delightfully, it aligns perfectly with the philosophy of microservices.

8.1.5 Acceptable error rates

This is a difficult point to win. It will be easier if you first win the point about measurement, because measuring the error rate is predicated on the acceptability of measurement as an activity.

The best approach is to not reveal your hand at first. Ask lots of questions about current problems: the level of customer dissatisfaction, the frequency of customer complaints, and so on. Ask for data on performance, uptime, and failed transactions. If it doesn’t exist, ask to track it.

Once you’ve established a credible current error rate, you can use it as a baseline to judge your team. If the business is surviving with a given error rate, then errors per se can be seen as nonfatal. Reduce the power of errors to terrify, and you’re halfway there.

Now, you can argue that anything that improves the error rate is good, and that’s what your team intends to do. Measure the error rate from the first day of the project, and use the microservice deployment pipeline to manage your risk and stay below that error rate, on average.

The ability to deploy code into production on a continuous basis relies on the company accepting that this will cause transient errors. Getting the business to understand and accept that this trade-off is not only acceptable but also one of the key productivity improvements delivered by the microservice architecture should be a primary goal of your transition project.

8.1.6 Dropping features

Don’t be afraid to drop features. Software systems accumulate features over time, because there’s little business incentive to remove them. This happens despite the significant cost these legacy features impose through technical debt. The value of individual features varies over time; if you chart the value, you’ll end up with a power-law distribution.[4] Now, chart those features against the complexity they introduce. The measure of complexity can be crude (lines of code, anyone?)—it doesn’t matter for the purposes of this analysis. Figure 8.1 shows the stereotypical 2 × 2 matrix beloved of consultants everywhere.

[4] Admittedly, this is *anecdata* from observation, experience, and reading, rather than repeated experiments. I’m not aware of any studies on this topic.

The lower-right box is the problem: these features deliver low value but have high complexity. It's valid to question their continued existence. Removing a feature without cause is asking for trouble—you have to expend effort, and you might annoy a vocal minority. But if reimplementing, refactoring, or providing supporting logic for a feature has a negative impact on your development velocity, you can legitimately question the need for the feature. Propose that it be removed. You won't always win, but if you don't ask, you won't get.

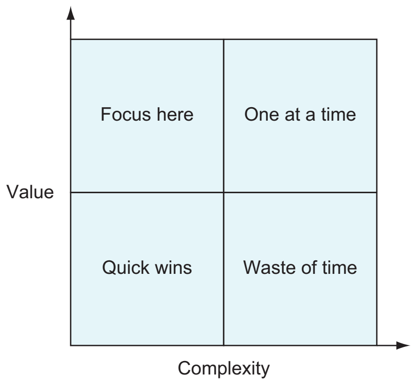


Figure 8.1 Value versus complexity matrix

8.1.7 Stop abstracting

Developers are trained to abstract the world. Abstractions are the things they *develop* and then express in code. The trouble with abstractions is that they have unbounded growth and eventually collapse under their own weight. Your challenge is to get your team—many of whom may not be familiar with the core ideas of the microservice architecture or may not completely buy into it—to stop abstracting.

This is a difficult people-management challenge. Developers love abstracting and get into a positive mental feedback loop when doing so. This is the dreaded sin of overengineering. It's much easier to control if you keep your microservices small and default to implementing new features using new microservices. But your team will push back, because they've been taught to solve problems by extending data models. They need to learn to use pattern matching.

You need to get your team to stop using the tactic of going from the specific to the general, which is the common experience of programming, and instead go from the general to the specific. This is a message you'll need to broadcast and defend on a daily basis. New team members will have to be brought on board. If you don't keep an eye on this, you'll end up with a distributed monolith.

8.1.8 Deprogramming

Software projects are difficult to investigate using the scientific method. It's hard to distinguish between mere correlations and actual causation. Strong experiments with proper protocols are expensive, and there are few examples of such experiments in the literature.

When humans try to comprehend things we don't understand, we use our powerful pattern-matching brains to invent superstitions. Perhaps we can't prove that superstitious rituals work,[5] but we perform them anyway, just in case. One ritual might not

[5] For example, it's bad luck to wish an actor "Good luck!" before they go on-stage. "Break a leg!" is the preferred send-off.

be too expensive, but accumulate a few together, and you can get a big drain on productivity and resources.

Software development methodologies, processes, and best practices are all suspect. Everybody in development suffers from unsubstantiated beliefs, and developers are the worst. There's a reason for the cliché that developers are "religious."[6]

I guarantee that you and your team suffer from preconceptions. Some are beneficial to microservices (decent unit testing), some are neutral (agile methodology du jour), and some are harmful (Don't Repeat Yourself). You'll need to be honest and explicit with your team about the fact that you're going to break some established conventions. You'll have to ask for their trust. If you aren't up front about this, your efforts will be negatively affected by both conscious and subconscious performance of the cargo-cult[7] rituals of modern enterprise software development.

8.1.9 External validation

Many organizations actively promote themselves in the developer community, as a recruitment tactic. If your organization already does this, you're well placed to talk about what you're doing with your new, innovative microservice approach. If your organization doesn't do this, start talking to those who care about recruitment, and show them how effective developer engagement can be.

The benefit to you is that external validation builds internal political capital. You're making your organization and your boss look good, you're solving a problem—recruitment—for both yourself and others, and you're generating enthusiasm for your efforts. These effects combine to make organizational roadblocks easier to overcome.

What if you aren't naturally inclined to present your work in public? This is a skill you need to develop, because it's become essential for any significant architect role. Start small: go to meetups, and get a feel for the territory and the types of talks they look for. Trust me, all meetup organizers are desperate to fill their speaking calendars; it's a monthly headache. Many meetups encourage and support new speakers. They want new people to present. Cut your teeth at meetups, and before you know it, you'll be speaking at conferences. You can give the same talk again and again—that's what everyone does. It's how you end up with a professional, polished presentation.

8.1.10 Team solidarity

The microservice architecture is an engineering tactic that's a partial solution to the larger problem of effective software delivery. But it doesn't free you from the problems of managing a team—you still have to get all the humans in the room to work together.

---

[6] Watch the "tabs versus spaces" scene from season 3, episode 6 of *Silicon Valley*. It's worth it.

[7] *Cargo culting* is the performing of rituals that have no effect or mechanism and can't affect the thing you care about. The term arises from the observed behaviors of indigenous islanders in the South Pacific, who copied the appearance of airfields in the early twentieth century in the hope that planes filled with cargo would appear and land. The term is pejorative.

There are time-honored practices for making a team work—far too many to enumerate or discuss here. It's your duty and responsibility to study, learn, and apply excellent professional practices for managing people, and to keep studying, learning, and applying them.

Your biggest weakness as a coder comes from your greatest strength: you can solve problems with code, so you try to solve *all* problems with code. Trying to solve people problems with code is a mistake, but we've all done it: instead of dealing with mismatched delivery expectations, we work all weekend to get it done the way "they" want it, so we don't have to deal with conflict. Try to swing the pendulum far in the other direction. How many code problems can you solve by engaging with people? That's why you ask if you can drop features. Many other problem-resolution opportunities are purely people based. Go look for them.

You should also measure your team as a whole. Do this collaboratively; don't impose the process on them. The purpose is not to measure performance, but to expose systematic issues. A weekly survey, which may or may not be anonymous, can help to raise issues that are hidden or that everybody is collectively ignoring. It's hard to ignore bad numbers, and they're a good way to start a discussion. You can score these measures on a scale of 1 of 10, or use 5 stars or whatever works for you, as long as you can quantify them. Here are some examples of things you can measure:

- *Overall happiness*—An emerging low score can tell you that organizational issues you may not be aware of are affecting your team.
- *Level of recommendation*—Would team members recommend to a friend that they join this team? This is a good way to get honest feedback about your performance as a manager, because the question is indirect.
- *Technical assessments*—Is performance OK? Do team members expect catastrophic failures? Are they happy with code quality? These questions can help you catch deviations from the team's desired technical outcomes. Code rots when you don't pay attention to it.

8.1.11 Respect the organization

Microservices are sometimes portrayed as a way to escape from Conway's Law. This is the observation, made in 1967 by Melvin Conway, that "Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure."[8] In my first job, for a web consultancy in the 1990s, our version of this was, "You need a menu tab for each department in the client company."

There's no reason the organizational design of a group of political humans should determine a good design for a system's software architecture. Software developers rail against Conway's Law as an example of the polluting influence of corporate politics: if only they were free to design the system properly without keeping the department heads happy, then they could deliver on time.

[8] For details of the original paper, see www.melconway.com/Home/Conways_Law.html.

Conway's Law deserves a more subtle reading. It should be taken as an observation of a natural force that creates movement in a certain direction, not as an immutable law or something inherently bad. It's a social observation. You can choose to fight it, or you can choose to use it. Either response may be appropriate, depending on your circumstances.

How you handle Conway's Law shouldn't be determined by your choice of the microservice architecture. Microservices don't compel you to reverse Conway's Law; they just make it easier, if that's what you want to do.

8.2 The politics of microservices

The microservice architecture creates its own political dynamics. The way these dynamics will play out in the long term, and the best approaches to dealing with them, will take years to understand fully. Therefore, if you're leading a microservices project, you'll need to pay close attention to the forces that drive team behavior. Be prepared to adjust course.

The most important principle to preserve is the decision-making pathway that decides what microservices to build. Begin with business requirements, express them as message flows, and then assign the messages to services. Developers who are writing services can undermine this principle by making individual services more important than they should be. This is a form of technical debt that will slow you down. Services must remain the least important thing in a microservices architecture.

8.2.1 Who owns what?

How do you distribute microservices among teams? Consider these scenarios:

- *1–1*—One team per service.[9] These are probably macroservices.
- *1–n*—One team owns many services. These are probably microservices.
- *n–1*—Many teams own one service. This is a monolith.
- *n–n*—Everybody works on all services. These are microservices, without a hierarchical ownership protocol.

As you migrate away from a monolith (*n*–1), you'll end up with a configuration of 1–1 and 1–*n* teams. Ideally, you'll end up with all teams in a 1–*n* configuration when the last macroservice is retired, but this rarely happens in practice.

The *n*–*n* configuration at the interteam level isn't practical, because it essentially negates the concept of a team; at the time of writing, no organization has demonstrated defensible evidence that this degree of flatness can work.[10] Even open source development has loose hierarchies.

9 *t–s*, where *t* is number of teams and *s* is the number of services. Cardinalities are denoted by 1 for one and *n* for many.

10 *Holacracy* is an organizational design that can be seen as a recent attempt to make this work. The results have been mixed and inconclusive. Holacracy seems to require a great many rules in order to work, which tends to defeat the purpose.

The *n*–*n* configuration *within* a team is a different story. This works and is the best configuration. Every team member can work on every service owned by the team, and no service is owned by any individual team member. This isn't a natural state for a team: team members will gravitate toward their specialties, assert implicit ownership over code they wrote and architectural structures they proposed, and avoid code they don't understand. In other words, individuals tend toward the 1–*n* configuration. You have to encourage *n*–*n* instead.

Why is *n*–*n* best for teams? Because it removes privileged code. No code or microservice is special. Everything is disposable and reconfigurable.

Microservices give you the technical ability to reconfigure your system cost effectively, but they don't automatically give you that on the human side. Be explicit with your teams: *teams*, not individuals, own microservices.

The tyranny of shared libraries

The term *shared library* refers to code components that are used by more than one microservice. They introduce complexity because you need to know which versions of which library should run on which microservices, what the current distribution of versions is, and what the incompatibilities are. To update a version of a shared library, you need to engage in complex planning that affects many microservices simultaneously. This is pretty much the opposite of what you're looking for from the microservice architecture.

There are two types of shared library: utility and business logic. Business logic in a shared library is the real killer and is almost impossible to justify. It's dangerous because it necessarily describes behaviors of the system that affect users. Breakage is highly noticeable—and breakage is what you'll get when you try to keep that business logic consistent over many teams and microservices. Instead, shared business logic should go in a separate microservice and be accessed via message flows. This is a fundamental tenet of the architecture.

Utility code is different, but not *that* different. Low-level utility code is safe enough—especially open source libraries. Updates tend to be the local decision of a single developer working on one microservice, so you should be able to catch breakage via the deployment pipeline. You're not trying to update the entire system at once.

Writing your own utility libraries can also be justified, but be conservative. These libraries have to be maintained, and the maintainers will be a small group, so you're exposed if they aren't available for some reason. Because the utility code is specific to your project, it can introduce system-wide compatibility issues and require system-wide updates. Nonetheless, you'll probably end up with shared libraries for logging, data access, and the like.

There's one shared library you can't avoid: the message abstraction layer. Fortunately, if you build it to be lenient when accepting messages, you won't suffer much from compatibility issues, even if you run multiple version of the layer in production.

8.2.2 Who's on call?

The *DevOps* movement is independent of the microservice movement but has heavily influenced the thinking about microservices. You can do DevOps without doing microservices, but it's difficult to do microservices without having a DevOps mentality. Microservices move complexity into the network, and that complexity is most fully understood by the developers writing the microservices.

Does the term *DevOps* means developers get to touch live production machines? Or does it mean developers and sysadmins should get better at collaborating? This is a wide-ranging discussion, so let's narrow our scope to the question of DevOps for microservices.

In the microservices world, nobody touches production machines. You have to automate. This makes it much safer for amateurs, such as developers, to modify the production system. But you still require expertise to run and maintain that system. This expertise is necessarily specialist, especially for large systems.[11]

The structure you end up using for the microservice architecture means the term *DevOps* in a microservice context refers to the operation of the upper layer of the automation system, rather than the more traditional system administration duties that full DevOps includes. This is where the developers affect the live system, through a well-defined set of operations that enable microservice deployment patterns. The lower layer is operated by systems engineers as a service for the developers.

There are many variants of this architecture; the distinction between the upper and lower layers isn't a hard boundary, merely a limit that you naturally approach when building very large systems. In the early days, when the greenfield microservice system is small, members of the development team often work on the full stack, crossing the boundary without restriction. The existing operations team runs the monolith and slowly moves into roles supporting the lower layer as the system grows.

Once the greenfield infrastructure is up and running, you have to address the question of how a development team maintains a deployment pipeline and a production system on an ongoing basis. Yes, the operations group is there to help and maintain the upper layer, but who operates it?

The solution that works best is to rotate each developer on the team through an on-call iteration (one week is best). For that week, the developer

- *Doesn't code.* This is the most important rule. The on-call team member needs to be in an interrupt-driven frame of mind. They can't do good, focused work in this state.[12]
- *Does pager duty.* Nobody likes this, but it's the trade-off for flexible and faster deployment to production. You need to be clear and up front with your team about the need to commit to pager duty.

[11] The Google role of Site Reliability Engineer is perhaps the best example of the need for systems specialists at scale.

[12] This guideline derives from the observation that human minds can enter a state of high productivity and focus, known as *flow*. Interruptions break flow, so you can't write great code when you're on call. For more, see the writings of Mihály Csíkszentmihályi.

- *Becomes the team concierge.* During each person's on-call week, they clean the toilets (figuratively—although, in a startup, it could be literally). Their job is to remove technical roadblocks for the team. That means they're on point for bugs, issues, support, meetings, and moving furniture. They keep the deployment pipeline healthy, liaise with operations, and talk to other teams. But they don't do politics; that's the job of the project leader and architect.
- *Performs deployments.* Only one person does deployments. Yes, technically you have a deployment pipeline that can support free deployments by anyone, because you've measured and contained the risk. But you don't have an organization that can handle it. Somebody needs to be directly responsible and act as the contact point with people outside the team.

Everyone on the team rotates through being on call—even you, the project leader.

8.2.3 Who decides what to code?

Distributed intelligence is a powerful way to solve problems. Let those who know the most make the detailed decisions. Provide a set of common goals, and then sit back and watch the magic happen.

That's the theory. In practice, it can go horribly wrong if you don't put supporting structures in place.

If you allow teams and team members to pluck work items from a backlog, even if you prioritize the backlog using some definition of business value, you'll quickly lose many of the benefits of the microservice architecture. It's worth stressing again: microservices make it *possible* to go faster, but they don't make it inevitable. Teams and individual developers tend to Balkanize into antagonistic, isolated domains. This is a natural social phenomenon: the formation of tribes. The effect is self-supporting. Small initial variations will build until you're back to the situation of ossified frontend and backend developers, and frontend and backend teams. The benefits you were hoping for, such as being able to quickly redeploy people and distribute knowledge to create redundancy, will erode and disappear. Let's look at some tactics you can use to prevent this from happening.

AUTONOMY AND VALUES

The question of who decides what to code is really a question of how choices are resolved. All decisions are reified in code,[13] so you need to find a balance between full autonomy and strict hierarchy. The former leads to Balkanization, the latter to low-grade problem solving.

This is one of the first discussions you need to have with your team. Openly discuss the level of autonomy the team can expect. Acknowledge that the issue is difficult and probably unsolvable. Collectively come up with a set of operating principles that

[13] The lawyer and digital-rights advocate Lawrence Lessig has made the argument that, because code controls much of the modern world, code is what determines what's allowed, not people. Laws may say one thing, but it's their (perhaps inaccurate) translation into code that's enforced.

produce the balance you and the team are looking for. Recognize that not only can you change the principles over time to better reach the balance you seek, but you can also change the balance point. You'll need to change the balance point as the project progresses and less freedom is required, given that many problems have been "solved."

This approach is an explicit rejection of the idea that there's an optimal software development methodology, that you should be a cult follower of one or another school or author, and that the methodology, once in place, can never be changed. The remaining tactics discussed in this section should all be considered in this light; they've been found useful by many teams, but they aren't religions.

Everybody codes

Everybody should write at least some code. To refine this idea, everybody should contribute construction effort, not just coordination. Some code full time; others, especially at the higher levels, may be distracted by management duties, depending on your organizational context.

To make it possible for everybody to write a little code, consider some practical approaches. Coding requires extended periods of concentration. Senior people will have trouble achieving this unless you create formal time periods that are safe from interruptions. This can be done, but it's difficult to enforce. A better approach is for senior people to restrict themselves to longer-term work, such as utility code or better algorithm implementations, that can be worked on over a period of weeks without affecting delivery. Senior people will also, of course, suffer the on-call rotation periodically, and will be able to flex their coding muscles on bug fixes.

For those who aren't able to write code, every project requires grunt work: manually verifying the user interface, replicating user-reported issues, doing usability testing, performing detailed business analysis, and so on. There's always something to do.

If everybody codes, then everybody gets some exposure to the problems on the ground. At a management level, it's easy to solve problems with the blunt force of labor. You command, and it's done, usually by people working late. This means your team ends up working hard when you need them to be working smart. You can't understand, evaluate, or direct solutions that reduce Toil[14] when you don't understand its dynamics.

If everybody codes, then knowledge of the system is deeper and more widely distributed. Group problem solving is more efficient because you can spend less time on communication. Politics is fed by lack of shared understanding and lack of agreement about shared facts: increase both of these, and you reduce the amount of energy you have to expend on politics to get decisions made.

The Tower of Babel

Before you build your first production microservice system, you may view a polyglot services architecture—one where any developer can build any service in whatever language they like—as either a massive advantage or a terrible disadvantage. Here's what

[14] See section 5.7.2.

happens in practice, regardless of your initial decision. Let's say you decide to stick to a single language. You'll encounter situations where you need to use other languages—sometimes for performance, sometimes for specialist features, sometimes because it's the quickest way to turn existing code into a service. A small percentage of services will be implemented in other languages.

Conversely, let's say you allow complete choice. One language will start off with a small advantage. Perhaps it's the new language everyone wants to learn, maybe it's the old language everyone already knows, or perhaps your most productive developer is just cranking out code in their language of choice. Soon, everyone will have to work with this language on a regular basis, and it will become the default choice for new services because it has low friction. Over time, although other languages will be present, one language will dominate.

Whichever starting point you use, the result is the same: our old favorite, a power-law distribution. Most services are written in the dominant language, and then there's a long tail of oddities. How do you deal with this situation from a maintenance perspective? Who looks after the oddities? If you lose a key person who understands the language R, say, what happens when you need to make changes?

This situation can't be avoided. You *will* end up with nonstandard services. Use it as an opportunity for team members to expand their knowledge of the programming universe. Use the rotation already in place to make sure everybody is exposed to everything over time. Make a team decision that if you introduce a language, you need to mentor those who want to work with it.

This won't provide full backup, and losing key people will still hurt, but the remaining people will include understudies who can keep things going. Also, remember that you can always replace services by rewriting them; this may be the best option, even if you have to compromise on performance or capacity cost.

Starter kits

The strategy of refinement means you'll build a lot of new services on an ongoing basis. You should make it easy to build new services in your primary language. Doing so will also reduce the enthusiasm for polyglot services, because building them will be more difficult, with more manual labor: the capabilities of the non-primary language will have to provide a significant advantage to overcome the ease of creating a service using the primary language.

Create a set of templates so developers can quickly get started coding a new service. At first, build a core template. Later, it may make sense to have different templates for such things as data exposure, user interface elements, business rules, and so forth. Who builds and maintains the templates? The same group that looks after the messaging library.

BIG-PICTURE THINKING

Despite the fact that the monolith is one large codebase in one large repository, developers can mentally isolate themselves from most of the system. To understand other parts of the system, you have to read the code for those parts to see how everything fits together—and that's hard work. Monoliths are opaque, because it's difficult to comprehend large-scale structures.

The microservice architecture is different because it offers another level above the code: message flows. Although this level will grow in time to be on the order of many hundreds or thousands of message flows, it's possible for a single human mind to understand the bigger picture in a usefully detailed way. With the monolith, all developers may understand the top-level architecture, but they'll find it difficult to get information about anything below that (apart from the local code they're working on).

Everybody can move from business requirements to messages. As the system grows, you can create conceptual groupings to organize messages. This aids global understanding: all team members can participate in and understand big-picture architectural discussions, because a large element of those discussions is the message flows they engender or the effect of the message flows on infrastructure decisions.

DISPOSABLE CODE

The idea that code can be disposable isn't something that requires microservices, but is something they enable. This book has argued that by following the basic principles of the architecture, you'll end up with microservices that can mostly be rewritten from scratch in one iteration. You can dispose of any given microservice by writing a replacement. As I've said multiple times, "replaceable within one iteration" is an excellent standard for determining the right size for a microservice.

What does this mean from a political perspective? First, you should openly discuss and acknowledge this principle with your team. Anyone's code can be replaced. Write something better, deploy it side by side with the existing code in production, measure the results, and may the best microservice win. Of course, you probably wouldn't want your team to be aggressively competitive in this manner—it's more that everybody is aware that their code is ephemeral and transient. This has the following effects:

- Mistakes are easier to unwind because you can focus on the technical decision, rather than the emotional impact. Nobody has too much skin in the game; even if a change stings a little, it's only one microservice among many.
- Technical debt is kept in check because there's no point in investing in complexity within one microservice, and little time to do so.
- Utility code and shared libraries are better quality. They aren't ephemeral and will be used by many generations of microservices. The effort to write quality code, which is a long-term effort involving many revisions, goes into core code, rather than business logic, which can be swept away in an instant if business strategy changes.

From a business perspective, disposability makes it easier to perform business experiments. The technical team won't actively push back. When working on a traditional monolith, you push back forcefully, because you can't afford the time for experiments—and you'll be blamed for the breakage they cause elsewhere.

In the e-commerce example in this chapter, you can perform complex A/B testing of new user-interaction flows and expect to dispose of most of them. It's worth it to find the user interactions that work well. Or consider a feature such as special offers. You can use the scatter/gather pattern to implement an ongoing evolution of many different kinds of special offers; then, you can measure which ones are more effective and allow those to dominate over time.[15]

VALUABLE MISTAKES

There will be mistakes. There will be downtime.[16] How you deal with this will directly affect your development speed and your ability to succeed in the long term. If your team is afraid of making mistakes, then they'll slow down to reduce risk, and you'll lose most of the benefits you're looking for. Monolithic development is slow in part because people are so afraid of breaking things.

Allow mistakes. Let people learn from them. When a mistake happens, let the person who made the mistake run the post mortem and explain it to others. Hold this meeting on a blame-free basis. You'll need to work hard to make this possible: remind everybody that complex systems fail because they *are* complex, and sometimes there's no root cause.

You also need to make sure information about mistakes isn't presented or leaked in the wrong way higher up the chain of command. Aim to establish a feedback loop that prevents you from making the same mistake twice.

Just because mistakes are allowed doesn't mean you should invite them. You need to maintain trust with the rest of the organization. You've agreed to an acceptable error rate—stick to it! Stay safe by using your deployment process to manage risk, as described in chapter 5.

8.3 Summary

- Microservices create a new environment for software development, one that's conducive to getting things done. But this is an engineering advance, not a political one. You must turn it into a political tool. Be aware that you're a disruptive influence in an established political scene, and prepare to face the consequences.
- Focus on visibly and quantifiably delivering business value. Doing so builds trust, and thus political capital, allowing you to remove even more roadblocks and accelerate even faster.

[15] Consider using *multi-armed bandit* algorithms.
[16] Section 3.6 reminds you of all the ways microservices can fail.

- When you move fast, you break things. To create enough safe space to do this, get the organization to accept that there's an existing error rate and that you can operate below it. Make errors acceptable, to reduce their impact. This is one of the more difficult political battles you'll face, but it's essential to win.
- There are many other tactics and considerations you'll need to pay attention to. Large human organizations are complex things; you must embrace the work of politics and accept that it's as important as getting the technical details right.
- The microservices architecture isn't a promised land that removes politics from the creation of software. It brings its own pitfalls and forces. Design your rules of ownership carefully, and recognize the difference between the way teams interact and the way developers on a team interact.

SOFTWARE DEVELOPMENT

the Tao *of* Microservices

Richard Rodger

Free eBook
SEE INSERT

An application, even a complex one, can be designed as a system of independent components, each of which handles a single responsibility. Individual microservices are easy for small teams without extensive knowledge of the entire system design to build and maintain. Microservice applications rely on modern patterns like asynchronous, message-based communication, and they can be optimized to work well in cloud and container-centric environments.

The Tao of Microservices guides you on the path to understanding and building microservices. Based on the invaluable experience of microservices guru Richard Rodger, this book exposes the thinking behind microservice designs. You'll master individual concepts like asynchronous messaging, service APIs, and encapsulation as you learn to apply microservices architecture to real-world projects. Along the way, you'll dig deep into detailed case studies with source code and documentation and explore best practices for team development, planning for change, and tool choice.

What's inside

- Principles of microservice architecture
- Breaking down real-world case studies
- Implementing large-scale systems
- When not to use microservices

This book is for developers and architects. Examples use JavaScript and Node.js.

Richard Rodger, CEO of voxgig, a social network for the events industry, has many years of experience building microservice-based systems for major global companies.

"The author has practical experience as well as great understanding of the concepts—a rare combination!"
—Sujith S. Pillai, Cloud Maxima

"Chock-full of useful advice for real-life microservice implementers."
—Victor Tatai, Fitbit

"A novel, in-depth, and philosophical approach to the subject. Very engaging and thought provoking!"
—Łukasz Sowa, Iterators

"Exactly what the title describes. The book shows you 'the path'—actually 'the true path'—to microservices."
—Peter Perlepes, Growth

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/the-tao-of-microservices

ISBN-13: 978-1-61729-314-6
ISBN-10: 1-61729-314-8
54999
9 781617 293146

MANNING $49.99 / Can $65.99 [Including eBOOK]