Containers *will not fix* your broken culture [and other hard truths]

BRIDGET KROMHOUT

**COMPLEX
SOCIO-TECHNICAL
SYSTEMS
ARE HARD;
FILM AT 11**

We focus so often on technical anti-patterns, neglecting similar problems inside our social structures. Spoiler alert: the solutions to many difficulties that seem technical can be found by examining our interactions with others. Let's talk about five things you'll want to know when working with those pesky creatures known as humans.

1. TECH IS NOT A PANACEA

According to noted thought leader [Jane Austen](#), it is a truth universally acknowledged that a techie in possession of any production code whatsoever must be in want of a container platform.

Or is it? Let's deconstruct the unspoken assumptions. Don't get me wrong—containers are delightful! But let's be real: we're unlikely to solve the vast majority of problems in a given organization via the judicious application of kernel features. If you have contention between your ops team and your dev team(s)—and maybe they're all facing off with some ill-considered DevOps silo inexplicably stuck between them—then [cgroups and namespaces](#) won't have a prayer of solving that.

Development teams love the idea of shipping their dependencies bundled with their apps, imagining limitless portability. Someone in security is weeping for the unpatched [CVEs](#), but feature velocity is so desirable that security's pleas go unheard. Platform operators are happy (well, less surly) knowing they can upgrade the underlying infrastructure without affecting the dependencies for any applications, until they realize the heavyweight app containers shipping a full operating system aren't being maintained at all.

Ah, but, you say, at our org we do this right (for sufficiently non-terrible values of "right")! We inject credentials at runtime, and run exactly the same containers in every environment. Perhaps we even ship lightweight containers with only statically linked binaries. Okay, but traffic patterns and data tested across various environments are likely not close to the same. As the old joke goes:

Proposal: rename 'staging' to 'theory'. "It works in theory, not on production." —[Najaf Ali](#)

Microservices won't prevent your two-pizza teams from needing to have conversations with one another over that pizza.

There is no substitute for experimentation in your real production environment; containers are orthogonal to that, while cross-org communication is crucial to clarity of both purpose and intent. [Observability](#) being key is a fundamental tenet of the gospel according to [Charity Majors](#). The conflicts inherent in misaligned incentives continue to manifest no matter where the lines of responsibilities are drawn. [Andrew Clay Shafer](#) calls the state of any running system “continuous partial failure;” good tooling is necessary (but not sufficient) to operate a robust fault-tolerant system.

Relying on health checks in your continuous delivery is all well and good, until the health check is full of deceit and lies because it says everything is 200 OK, and all those instances are staying in the load balancer, and yet nothing is *working*. (My on-call PTSD may be showing.)

In a world of ever-increasing complexity, how do we evaluate our progress toward a [Container Store utopia](#)? How do we know when to course-correct? How do we react when it seems like there's always something new we should have done last month? Must I really orchestrate my containers? Could they maybe just do some improv jazz?

2. GOOD TEAM INTERACTIONS:

BUILD, BECAUSE YOU CAN'T BUY

We hold in our heads the intricate composition of our complex distributed systems, and there's increasingly even more state we can't fit into those necessarily incomplete mental models. Microservices aren't defined by lines of code so much as by the scope and breadth an individual service covers. And, no, microservices won't prevent your two-pizza

teams from needing to have conversations with one another over that pizza. (Also, how hungry are these people, and how large are the pizzas? So many unanswered questions!)

[Adrian Cockcroft](#) points out that a monolith has as much complexity as microservices; it's just hidden. Okay, so we're going to deconstruct that dreaded monolith and keep on rockin' in the microservices world! That will solve everything! Clean abstractions and well-defined handoffs sound great, until you realize that you're moving the consequences of decisions (and the conflict inherent in any set of tradeoffs) into another part of your stack, which [Tim Gross](#) calls "conservation of complexity".

Breaking into individuated teams doesn't change the fact that the teams have to agree where the boundaries lie at any given moment. Writing in the 1960s, [Mel Conway](#) could have been talking about today—except for the title, because "[How Do Committees Invent?](#)" very much buries the lede; today it would be a clickbait listicle.

Conway wrote that any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure. This came to be known as Conway's Law.

Contrary to popular belief, [Conway's Law](#) does not say your org chart has to look exactly like your [death-star software architecture diagram](#), and a cursory inspection of either would lead us to believe that plan would never scale, anyway. No matter what design decisions you make around thermal exhaust ports, no amount of industrial-strength job scheduling makes your organization immune to Conway's Law.

The most important word in Conway's Law is

communication. In your excitingly deconstructed world, how is communication about breaking changes handled? What about schema migrations, because state is real? (The pesky thing about storing state is that your value often exists there. The money types get awfully touchy about anything that could adversely affect systems of record.)

Creative problem solvers have a way of routing around process that we find inconvenient. If your heavyweight change control process applies *except* in case of emergencies, then [spoiler alert] you're going to see a surprisingly high rate of sorry-not-sorry "emergencies."

[Dunbar's number](#), a cognitive limit on the number of people with whom an individual can maintain stable social relationships, is demonstrably valid. If working in a larger organization, you'll need to communicate in smaller groups, but those groups should be cross-functional to eliminate bottlenecks and misunderstandings.

Communication doesn't just mean talking with our human voices or replying to interminable email threads, either; much like [Consul's gossip protocol](#), we need cross-talk in our orgs to keep communication flowing.

We've all heard "we only communicate through APIs" but technology alone does not solve all communication problems. If you launch a new version of the API, does that mean you'll ever be able to deprecate the old one? Is well-labeled versioning sufficient for current needs of all your API's consumers? How about future, conflicting, overlapping needs? At some point, you'll have to talk to each other. (Bring some pizza!)

3. TECH, LIKE SOYLENT GREEN, IS MADE OF PEOPLE

[Andrew Clay Shafer](#) likes to opine that 90 percent of tech is tribalism and fashion. Tools are important, but people are an integral part of any human-designed complex system. We've all seen the ridiculously expensive migrations gone wrong, the years-long lift-and-shift projects that accomplish only a fraction of their goals because of the necessity of maintaining business continuity, the "DevOps initiatives" that last only long enough for somebody's vice-presidency level-up to complete. Examining the motivations driving these decisions (even if reconstructed by observing consequences) can frequently reveal the probable genesis of suboptimal decisions.

Nobody is doing résumé-driven development with shell scripts; I'm willing to bet that all the janky bash ever written was meant to solve a real problem. When we start getting fancier, there are often motivations less pure than "Let's do this well," and even if there aren't, intention alone doesn't create maintainable software. [The trough of disillusionment](#) is where we all land when dreams meet reality. Whatever slow-burning tire fire results from a given IT project, it's a sure bet it will burn for a good long while. Software is "done" when it's decommissioned; until that point, day one is short, while day two lasts until the heat death of the universe.

A good mental model is Simon Wardley's "[Pioneers, Settlers, Town Planners](#)". While a proof of concept can be "done" enough to ship it, operationalizing it takes longer, and keeping it running in production is an ongoing project. Entropy increases, as the second law of thermodynamics explains.

Obviously, striving for iterative IT improvement matters, but it's not an end state. We've all been in those meetings where people are not so much listening as just waiting for their turn to talk. [Software is made of feelings](#), as [Astrid Atkinson](#) puts it. We need to consider our impact on each other. Certifying people in DevOps is like celebrating their graduation from kindergarten. "Congratulations! You learned not to eat the crayons and to play nicely with the other children!"

Does this mean that DevOps has failed in its promise of increased efficiency brought to you by collaboration? Not in the slightest. Talk to the fine folks at DORA ([DevOps Research and Assessment](#)): a measurable impact shows up in the research when we center IT improvement. We can't buy DevOps, despite what some in the ecosystem might promise that a given tool offers. We have to live it; change for the better is a choice we make every day through our actions of listening empathetically and acting compassionately. Tools can and do help, but they can't make us care.

4. GOOD FENCES MAKE GOOD NEIGHBORS

Boundary objects and abstractions give needed structure, and containers make good boundary objects, but they do not eliminate the liminal space between the metaphorical [or all-too-real] dev and ops. When you implement microservices, how micro is micro? Even if you have a well-defined service that does one thing (somewhat) well, a good rubric is whether the service's health endpoint can answer unambiguously. If the answer to "Is this working?" is "Welllllll...", that service isn't micro enough.

Deciding what's *yours* and what's *theirs* is the basis of every sibling-rivalry détente. In [Eric Brewer's CAP theorem](#) you can pick two of consistency, availability, and partition tolerance as long as one of them is partition tolerance, because, as distributed systems expert [Caitie McCaffrey](#) puts it, “physics and math.” In a distributed system that contains humans in multiple time zones, you're inevitably going to have partitions, and waiting 10 hours for headquarters to wake up and make a decision is nobody's idea of a good time. But decentralized decision making means distributing power to your human edge nodes (sometimes a hard sell).

Empowering developer choice is facilitated by containers; there's always a tension between what someone else dictates and what you're convinced you need. Making thoughtful decisions about tools and architecture can help; well-considered constraints can free us from the decisions that aren't bringing us distinguishable benefit. Containers can help define scope and reach of a given tool or project, and deconstructing systems to human scale allows us to comprehend their complexity.

Being able to reproduce a build allows for separation of concerns. We want this to be effective and yet not introduce unnecessary barriers. The proverbial [wall of confusion](#) is all too real, built on the tension between having incentive to ship changes and being rewarded for stability. Building just the right abstractions that empower independent teams is worth taking the time to iterate on (and, no, nobody gets it right immediately, because “right” will evolve over time).

We want to empower people with as much agency as possible within the constraints that work for our organizations. To determine the right constraints for you, you need to talk to your teams. Think in terms of TCP instead of UDP; you'll need to SYN/ACK to really understand what other humans want. [Nonviolent communication](#), where you restate what you heard, is an effective way to checksum your human communications. (Bonus: techies will appreciate this logic!)

5. AVOIDING SADNESS AS A SERVICE

Hindsight being what it is, we can look back and recognize inflection points. It's harder to recognize change in the moment, but the days of operating your own data centers, where your unit of currency is the virtual machine, are coming to a definite middle. The hipsters among us will say that's over and sell you on serverless (which is just servers you can't ssh into), but we're talking about the realities of enterprise adoption here, and they're about at the point of taking containers seriously. Application container clustering is better for utilization and flexibility of workload placement, and using containerized abstractions makes for better portability, including for those orgs looking toward public cloud.

W. Edwards Deming, a leader in the field of quality control, said, "It's not necessary to change. Survival is not mandatory." Change is hard. Not changing is even worse. Tools are essential, but how we implement the tools and grow the culture and practices in our organizations needs even more attention. As it turns out, it's not mandatory to write a Markov bot to parse the front page of Hacker

News, then yolo absolutely everything out to production instantly!

Whether you're just starting to implement technical and organizational change, or facing the prospect that you already have legacy microservices, it's worth considering the *why* and *how* of our behaviors, not just the *what*. If legacy weren't important, you could just turn it off. But this is where your customers and money live. Glorifying exciting greenfield projects is all well and good, but the reality is that bimodal IT is a lie. It's ludicrous to tell people that some of them have to stay in "sad mode" indefinitely, while others catapult ahead in "awesome mode." Change is on a continuum; absolutely every change ever doesn't happen at the same instant.

We succeed when we share responsibility and have agency, when we move past learned helplessness to active listening. Don't be a named pipe; you're not keyboard-as-a-service. Assuming we can all read code, putting detail in your commit messages can be a lot more useful than soon-to-be-outdated comments. Tell future-you why you did that thing; they can read but don't know what you intended. Oral tradition is like never writing state to disk; flush those buffers. There is no flowchart, no checklist, no shopping list of ticky boxes that will make everything better. ["Anyone who says differently is selling something"](#), as *The Princess Bride* teaches us. Orgs have "the way we do things" because process is the scar tissue of past failures.

You can't take delivery of a shipping container with 800 units of DevOps, and have 600 of them go to the people in awesome mode, while the people in sad mode can look at the other 200 but not touch them. DevOps is something

Related articles

➡ The Verification of a Distributed System
Caitie McCaffrey

A practitioner's guide to increasing
confidence in system correctness
<https://queue.acm.org/detail.cfm?id=2889274>

➡ Adopting DevOps Practices in
Quality Assurance
James Roche

Merging the art and science of
software development
<https://queue.acm.org/detail.cfm?id=2540984>

➡ Bad Software Architecture is
a People Problem
Kate Matsudaira
When people don't work well together they
make bad decisions.
<http://queue.acm.org/detail.cfm?id=2974011>

you do, not something a
vendor implements for you
with today's shiniest tools.
Change for the better is a
decision we make together.

Tools are necessary but not
sufficient. To build a future we
all can live with, we have to
build it together.

Bridget Kromhout is a principal
cloud developer advocate
at Microsoft. Her computer
science degree emphasis was
in theory, but she now deals
with the concrete (if the cloud
can be considered tangible).
After 15 years as an operations
engineer, she traded being on
call for being on a plane. A

frequent speaker and program committee member for tech
conferences, she leads the devopsdays organization globally
and the DevOps community at home in Minneapolis. She
podcasts with Arrested DevOps, blogs at bridgetkromhout.com, and is active in a Twitterverse near you.

Copyright © 2017 held by owner/author. Publication rights licensed to ACM.