DevOps Done Wrong

A study of seemingly obvious common corporate failure modes

Johannes J. Meixner

Aarau, November 2022

In the following article I expose some seemingly obvious and yet very common failure modes of applying DevOps principles to your organization. I look at using Big Data frameworks, the dangers of Environment Multiplication, Missing Code Reviews, the everlasting practice of Continuing Silos, playing Hybrid Games, the joys of Agile Waterfalls and ultimately conclude with the horrors of Fragile Processes.

Contents

Big Data Frameworks

One of the most curious results I've noticed in the last two years:

Frameworks that exist to supposedly make big amounts of data handling "easier" on average fail horribly, all in the same ways.

When used outside of narrowly-defined means of the user interface (point & click), they obscure so many relevant details you will spend 3/4th of your time trying to navigate their ill-documented APIs - when all they allow you to do is changing a file on a file system, which can be done by a junior sysadmin in an instant.

This is, of course, fine and dandy if you are working in a body-leasing sweatshop and get billed out to clients by the hour (or day) receiving a fixed salary, because obviously you will want to learn myriad new ways of changing files in the cloud, all on a customer's dime, so you can list experience with the latest and greatest frameworks among customer's testimonials and in your resume.

Whereas, if you were self-employed, billing by terabytes processed per hour, you'd want to go the "extra mile", and eliminate extant fragility, one source at a time, until there's almost nothing left.

You would probably set up a highly repeatable cluster installation using very dumb tools (Perl 5.x, Bourne Shell, maybe Python) on bare metal, and focus on the underlying software; never the framework to manage it.

Incentives matter.

Environment Multiplication

Systems development has been cursed recently with the blessings of "DevOps", making it somewhat cheaper to spin up very large amounts of very similarly configured servers.

When software developers noticed this, one of the first things they did was to convince management they need more environments to satisfy different audiences (other developers - QA & testing - integration - paying clients).

Project management happily agreed, thinking this would introduce some level of risk management: As the myth goes, what has successfully deployed to both Test and Integration environments will not cause production outages.

This is probably a sensible measure if you have a piece of software generating large amounts of dollars: legacy applications running on mainframes.

Yet with the advent of virtual machines, every niche group within a firm now wants their own environment: There are data integration engineers, electronic data interchange specialists, pre-sales engineers and managers that need to present to their bosses bosses,

and they couldn't possibly use the same environment - because one would be fragile to changes made by another.

So to decouple this, some previous projects I had worked on have spun up a dozen (!!!) different PERMANENT environments, not even counting temporary containers spun up for test suites.

Which, as you can probably guess, created a lot of make-work in environment maintenance. To satisfy some niche group's vanity, at the expense of the system's overall stability.

Incentives matter.

Missing Code Reviews

In most industries where lives are on the line, not using process checklists and multiple reviewers is considered unthinkable. In those, reviewers are often actively encouraged to criticize people of much senior status when they fail to perform basic duties.

Meanwhile, in corporate IT: Most firms use file hosting systems with built in code review functionality. Thankfully, changes often require at least another pair of eyes to approve them. And yet, these are more often than not a formality. Approval is given without actively engaging with the difference in functionality before and after, often within seconds of opening the code review's URL.

Some of this might be understandable. A junior developer new to a codebase might not think they are in any way qualified to actually approve anything. A senior developer may have a reputation for good work. Other human factors might come into play.

And all of it at one point or another will generate excess cost. Sooner or later, some not well thought out code will go into production, and cause outages[1].

Something that might prevent these is to encourage everyone, no matter how experienced, to participate actively in code reviews. This means, at a minimum, asking questions:

- Often enough, code that produces negative side-effects has a certain "funky look" to it. If something seems unclear, unnecessary, out of line, or in some way dubious, this is an easy way in to leaving a comment along the lines of "What's this do?"
- "Clever hacks" - that is, anything that takes longer than expected to understand - should have comments documenting intended behavior. If they are missing, here is another way in.
- Last but not least, when you see someone using a pattern you are not yet familiar with, you could ask where (which language, perhaps) this comes from.

This document will give you some ideas. But first and foremost, it gives you the license to question. At the price of a duty to perform code review.

[1] The author of these lines has produced these effects often enough, "for research purposes."

Continuing Silos

Under traditional methodologies, waterfall and the likes, companies have compartmentalized software development and platform operations into two dedicated organizational units, often with entirely different management lines imposed from the top.

This model can be continued by adding a third component into the mix - hiring dedicated "DevOps staff" to sit in-between between developers and operators.

Companies need to satisfy stakeholder demands for state of the art in-house knowledge on deployment software of the day, be it "configuration management" or "infrastructure as code" software.

Popular with this model is to keep actual infrastructure changes within operations. Development teams "provide the software". DevOps teams "provide the infra glue". Operations "provide deployments". Neatly separated.

What obtains is that people writing the software are twice-removed from its effects, and that those who write the "infra glue" do not have their skin in the game during deployments either.

Combine that with on-call duties separated into yet another team (!) and you can virtually guarantee hilarity ensuing left right and center.

Software developers work very differently when their sleep is put at risk. DevOps staff will be much more diligent in ensuring everything is well tested and stable before it touches integration environments.

What seems to work much less badly: forming mixed teams, involving all three groups mentioned above, rotating on-call duties - and checklists.

Hybrid Games

With the advent of Cloud computing, established companies face changing business models. They are scrambling to transform their IT departments running on "legacy systems" from cost centers to more customer-focused operations.

Likewise, cloud-first startups grow into sizes where purely relying on AWS (OpEx), versus building up their own data center (through CapEx) is not as cost-effective at scale as previously thought.

At both ends of the spectrum, successful transformation is required to maintain margins. This can be done in a myriad ways, many of which yield only partial implementation and as such will lean towards half-bakedness.

To follow the first case: A common approach is to dive headfirst into new Cloud platforms, leveraging quick-to-scale services into a very complex architecture while bringing "on premise" mindsets (long-running, mutable infrastructure that requires patchdays).

A more cautious approach uses "On-Premise Cloud" services, provided by big players like VMware, Oracle, SAP, and the various OpenStack backers. It involves virtual machines operated through "self service portals". The caveat nobody talks about is that these are typically operated by a very limited class of people, so are never actually self-service in practice.

Mindsets consist of intellectual habits - ways of thinking about things - which take some time to obtain, and are achieved only by deliberate practice. What we do most often shapes how we think about things ("when all you have is a hammer"). Hence new paradigms require new mindsets; whether you move from in-house architecture to the Cloud, or migrate in the reverse direction.

It is absurd to assume that an application written for AIX will be easily ported to Amazon Linux and will work flawless in auto-scaling groups without a complete rewrite. Likewise, it is absurd to assume that change management processes can be ported to the Cloud without redesign.

Similarly, it is absurd to assume that immutable infrastructure works just as well via KVM over IP. Reimaging a server the cost of a middle-class car still requires more than ten seconds, and consequently, slighthly more careful operations.

Growing "on-prem" or "cloud-only" operations into "hybrid operations" can be done if supplied with helpful management that fully supports the bimodality in operations models.

Everything else will end in tears.

Scrummy Waterfalls

One important challenge in modern enterprise is reconciling the volatility and uncertainty around agile processes with stakeholders expecting waterfall-style outcomes, their inherent predictability and sense of covering your ass.

SCRUM brings its daily standups, weekly meetings, plannings, reviews, refinement and retrospective meetings. It is a complete suite for achieving product development through verifiable and (sort of) plannable increments.

Now we learn that this can be reconciled, merged and layered with standard waterfall project management mechanisms, especially those that have been used for decades in large companies and state-sponsored enterprises.

This, of course, yields the worst of both worlds: you are neither agile, nor will you ever be able to "throw planned results over the wall to operations". All the paper trails you leave to satisfy dinosaur processes, such as writing project specifications to obtain internal funding, cost valuable resources you cannot spend building the thing you are supposed to be shipping.

To escape that trap, challenge the notion of Hybrid approaches. It will work better if you pick one or the other.

Fragile Processes

As a business (as opposed to a Company of One, like yours truly) your strategic advantage comes from codifying repeated workflows into proper processes. This leads to desirable side effects, one of them being the saved time, money and effort in doing things the Proper Way you've always done it.

What I've noticed is that many companies go to great lengths to codify complex interactions, but . . . end up either overthinking things, making the process more complicated than necessary; or the very opposite: forgetting to codify things, and then forgetting to execute.

Two examples that I've noticed in recent years:

- It's probably a good idea to immediately revoke access and offboard a contractor whose invoice you have no desire to pay. American firms are *very* good at this. Privately held European corporations, especially the larger firms, also. Some startups as well, when the founders have at least one security-person on the board. Otherwise, it's like watching a slow-moving train wreck: Imagine being subjected to an external audit, the auditor seeing an unpaid invoice, and write access (or worse) to the core parts of the infrastructure. ;-)
- It's probably a good idea not to overthink new processes for corporate open-source platforms you've never used. This is something that you'll work out in time, typically within the first two years after setting up new infrastructure. People change, platforms do in subtle ways, and you'd want to make sure your documentation remains easily adaptable. Preferably on the leaner side.

With some creativity, these two very opposite cases are easily solvable. There might be a middle ground between the two extremes - pure laissez-faire, and pure top-down design - and you might be able to find it in due course.