optimizing application ecosystems

How to Screw Up DevOps:

A Field Guide

WHITE PAPER

Table of Contents

How to Screw Up DevOps – A Field Guide

Executive Summary

There are many terrific sources outlining keys to successful DevOps adoption. Knowing what to do and what steps to take in your drive to DevOps is key. But knowing what to avoid in your DevOps journey is essential. Making these mistakes can ruin your results and convince managers and colleagues that "DevOps just doesn't work here."

This whitepaper addresses specific DevOps mistakes, how to avoid them, and how to recover from them.

Technology Mistakes

Mistake #1: Baking Cupcakes

A relatively new class of technology mistakes that many teams make is to over rely on Desired State Configuration, also known as DSC. Perhaps today's IT teams grew up watching too much "Star Trek: The Next Generation", taking to heart Captain Picard's frequent command to "make it so". Working towards the end state, your desired state, is an enticing and logical sounding proposition. Focusing on the desired state can be a major time saver, especially under conditions where
1) the person or system that is granting your wish for a certain end result already knows the full process of achieving your goal (of baking and decorating a cupcake in our example) and
2) the desired state is truly "set" with few anticipated modifications.

But desired states aren't enough in and of themselves in more complex IT systems. They almost always need to be orchestrated to reach the final goal. An orchestrated states effort runs more along the lines of: "I need a white cupcake with pink frosting at the same time as a bowl of orange punch, before the time of the birthday party, but after the house cleaning. Also, yellow balloons." Real-life IT infrastructure requires at least this level of complexity and timing.

Unlike a cupcake, which has an end (baked and decorated) state, IT systems are reviewed, revised, approved, updated, upgraded, changed and sometimes even rolled back. They are never truly "baked". Imagine 'unbaking' that cupcake and then adjusting ratios of ingredients and baking time. While DSC is helpful, it is not enough in and of itself. The best DevOps solutions consider both transitory states and final states as well as the interactions between them.

Mistake #2: Managing Server #17

Servers don't matter. Applications matter. Customers and CIOs care that applications such as ecommerce, flight reservations, insurance quotes or even games are working properly. Customers interact with and do business with organizations because of their applications, not their servers.

As those applications improve and address more customer needs, they also become more and more complex over time. An organization may have 10 to 12 critical applications that depend on 400 additional applications to accomplish their tasks. So it is puzzling that even though IT and consumers are moving to a very application-centric world, many DevOps solutions still take a server-centric approach. When managing complex applications (especially with the cloud involved), the underlying servers should be immaterial. Those servers are a means to an end to provide *processing* which powers applications.

Consider: Does the CIO actually care about what's going on with server number 17? Or do they care that the ecommerce application is performing well on Black Friday?

In addition to asking these Server-Centric Questions:	**Also ask these Application-Centric Questions:**
• What's going on with this server?	✓ What's going on with this application?
• What steps do I take to make a change to this application on this server?	✓ How does changing my application affect the larger ecosystem?
• What are the configurations of this server?	✓ What are the configurations of this application across the servers on which it's running?

In a one-to-one, app to server world, the server became a stand-in for the application. If the server was having problems, the app was having problems. So the focus was naturally on server management. But in a virtualized world, continuing to focus our DevOps conversations and tool selection only on server management and server performance is a mistake. DevOps teams need to stop solving problems as if they are always working with a set number of physical servers that have only one application running on them. Instead they need to reorient their thinking and their systems management tools to reflect the reality of a one-to-many world: one application driven by many physical, virtualized and cloud servers.

Mistake #3: Using Scissors

A critical DevOps mistake is to repurpose a tool that was built and optimized for a different use case, to solve a completely different problem. For example, IT Ops may have purchased an application build automation solution many years ago and now they need to automate the deployment of those builds from Development to Production. Their smart, well-intentioned engineer says it's no problem to add deployment to your current solution – "all it requires is adding some deployment scripts and giving access to other teams." In a similar vein, companies have a great *server* configuration management tool, and now they need to manage *application* configurations. And again, that well-intentioned engineer says it's no problem to add application configuration management to your current server management solution – "all it requires is some coding to have it manage application configurations and middleware."

Scissors are great…

But sometimes you need a hammer

This can be a big mistake because the tool has now been repurposed and overextended to attempt to address a specific task beyond its scope. This 1) allows more users into the tool (all with different objectives and skill levels) thereby potentially compromising security and 2) makes the tool much harder to maintain, because more and more custom scripts will be needed to extend it.

Also, the further the engineer goes down the modification path the more they realize the tool doesn't suit everyone's needs, and a three month project soon turns into two years of work going back and forth between balancing needs and changes for all the different teams, reports, and audit-trails now required by a single solution. This eventually turns the tool into a slow "spaghettified" silo that no one wants to use.

If you've adopted a commercially available solution to help solve a specific DevOps technology problem and you're looking to extend it, look up the tool's advertised features, capabilities, and problems it solves. If the new problem you're trying to solve isn't listed, the tool isn't focused on it. Use your current tool for what it's intended, and look for another solution to solve your current problem.

You will be more efficient if you have many tools that do exactly what they're meant to do and have them all coordinating between each other, as opposed to trying to kludge everything into one tool and rendering it useless.

Mistake #4: Fly Fishing

A husband buys "his and hers" fishing poles, tackle sets and a fly fishing vacation as an "anniversary gift". Nice try. His wife doesn't fish and doesn't want to learn. This same scenario plays out in the IT and DevOps world repeatedly. One team wants a tool and then buys that tool. Then they introduce it to their counterparts and call it a "DevOps" tool – because that's the trendy word of the day. Many tools will naturally be a bit more useful to either Dev or Ops. That is understandable. But if that tool becomes a new *burden* to one of the teams, requiring extra work or even an additional "source of truth", then it is destined to fail as a cross-team tool.

At the very least, a DevOps solution should do no harm to any team. And a true DevOps solution will appeal to Dev and Ops - and ideally to Security as well. If their daily routine will be impacted by the new "DevOps" tool, getting early buy-in from affected teams is key. Otherwise it will be extremely hard to get the other team to adopt the solution and it will never realize its full potential.

Mistake #5: Showcase Showdown

Enabling the technology side of DevOps by implementing multiple open automation frameworks is like winning a showcase on the Price Is Right. You win! New toys! When you first win you're so excited about your new Jet Ski and hot tub. Then the items get delivered... You realize you need to get the Jet Ski registered and pay insurance on it, you also have to somehow move the hot tub from your curb to the backyard. The IT world calls this "implementation". The hot tub doesn't quite fit where you wanted to put it, so now you are stuck with an unintended, unbudgeted home renovation project. Not to mention the upkeep on the hot tub over time.

The same thing applies to implementing automation frameworks. Each automation framework is exactly that – a framework. Who writes and maintains all the scripts that make the framework useful? At this point maintaining your framework will be similar to maintaining a homegrown or open-source tool with no support. What happens when the person who wrote the scripts goes on vacation and something goes wrong? What happens when they move on?

Look for solutions that are more than just automation frameworks. Consider:

1) Does the "solution" rely on community or custom scripts?
2) Does the "solution" place the burden of security and compliance on the diligence of the team?
3) Does the "solution" require extensive tribal knowledge to work properly?

The common DevOps trap is settling for an automation framework when you actually need a *solution*. While frameworks may appear to be more "flexible", robust DevOps solutions eliminate the wasted time in writing and maintaining scripts, responding to fire drills, recovering from outages or picking up the pieces after key personnel leave the company.

Process Mistakes

Mistake #6: Perfection

DevOps teams are constantly improving software release and tackling new challenges. Once they have prioritized which problem they are trying to solve, they begin to evaluate the best tools to solve the problem. One of the first decisions is whether to build the tool in house or look for a third party solution. Either way, the first step is to build a list of requirements. Requirements lists can be ten pages or more – big excel spreadsheets with multiple tabs and hundreds of lines. Sometimes this may be necessary, but more often than not at least a few of these line items are "nice-to-haves-in-the-future-maybe". Since each requirement likely adds time and costs, every requirement should be scrutinized to verify just how necessary it is.

For instance, in DevOps projects a common requirement is that “the tool must integrate with our Configuration Management Data Base (CMDB)”. In the verification process, the most common answer is “well, we’re not there yet, but we’ve been working on getting our CMDB set up for multiple years and we want to make sure you have a built-in integration to it.” Future proofing is admirable but it has its limits. It is a mistake to build your DevOps plans around an expensive and complex requirement that is not going to happen any time soon. Instead of focusing on a unified CMDB, focus on what a CMDB would give you. So here are some good questions to ask at this point: What kind of information would you like to put in the CMDB? What information are you already storing in your CMDB that you would like this tool to access? If the answer is murky, consider dropping this requirement.

The point here is to avoid the all-too-common DevOps mistake of constraining your selection based on nice-to-haves or futures that may *never* happen. Perfect becomes the enemy of the good. Vendors will waste their time and yours on perceived constraints. Costs will grow and timelines will unnecessarily expand. Whether the ‘requirement’ in question is a CMDB or another constraint, it is essential to avoid confusing the means-to-the-end and the end goal.

Mistake #7: Misjudging the Finish Line

DevOps and IT Automation veterans are often looking for ways to move complex, multi-tier applications from Development to Production. When surveying teams about their deployment pipeline, it is often described that “the application moves from Development, through Test/QA, into the Pre-Production environment, and then ultimately Production.” More than 80% of the time no one ever mentions a Post-Production environment. This is puzzling because most IT organizations still treat Pre-Production as a sandbox. It is often directly accessible to developers, QA, and anyone else who needs to test a change. When this is the case, undoubtedly a configuration or application gets “tested” or tweaked in Pre-Production. This makes it impossible to reliably compare the applications and configurations running in Production against the Pre-Production environment.

If you are using your Pre-Production environment correctly, you may not need a dedicated Post-Production environment. But, ask yourself – is Pre-Production completely hands off? Who has access to Pre-Production? Does Pre-Production always look identical to Production? If not, a dedicated Post-Production environment is needed as a required step in your DevOps pipeline. Also, adopt solutions that allow you to easily maintain your Post-Production environment (server configurations, middleware configurations, and deployed applications), compare between Post-Production and Production environments to more easily remediate Production failures, and secure the Post-Production environment so only necessary individuals have access.

The Post-Production environment is too important to leave to chance or to take for granted. Misjudging the finish line is a common but preventable way to screw up DevOps.

People Mistakes

Mistake #8: Forgetting the Hate

Analysis paralysis can undermine your DevOps goals. Decision processes that meander and attempt to accommodate too many inputs are destined to sap team energy. If there are multiple paths to achieve your goal, you can get stuck in a doom-loop of no action.

As a case in point, the OrcaConfig team was having a design meeting to discuss how to implement a new design feature. Each talented engineer had their own idea of how it should be done. After hours of tediously debating the merits of each approach the conversation still sounded like this: "Well I like this part of option two and that part of option three, but I also kind of like option one."

Finally to break the logjam, a team member stood up and asked everyone "which option do you hate the most?" This was a much easier question to answer, and thus the Hate Matrix was born.

"the Hate Matrix was born"

Asking a colleague what they "hate" is the faster way to eliminate options and get to a decision, especially if more than three options are involved, and more than two people are involved in the decision making. For some unknown reason (wink), people of the IT world more easily identify something they dislike and instantly become more decisive. Teams can either fight this mentality or relish it, work the way their brains are thinking at that moment, and make decisions faster. An initial concern was that some coworkers would find the Hate Matrix insulting or counterintuitive, but just days after its inception a colleague brought in a dedicated "hate board". Now anytime the OrcaConfig team has a meeting that results in a stalemate, one or more peers suggest doing a quick Hate Matrix.

How the Hate Matrix works – Each option up for debate is written on the left hand side of the white board. Each decision maker's name is at the top of the white board. Start with one decision maker and ask them – "which option do you hate the most?" They will probably answer much more quickly than if you asked them which option they *like* the most. The option they hate the most is scored a "1". The option they hate the second most is scored a "2", and so on. Move on to the next decision maker. Once all decision makers have voted, add up the total for each option (row). The row with the highest total is the winner – this best option is the least "hated". And rather than taking it personally, some team members will "hate" their own idea once they see it in the light of a hate matrix and compare it to other alternatives.

Slow decision-making processes torpedo fast-moving DevOps environments. A well-executed hate matrix can be used to regain that momentum.

Mistake #9: Three Pizzas

Teams that have a lot to accomplish are often tempted to enlist more "hands on deck". **T**ogether **E**veryone **A**chieves **M**ore. Or so we're told. Scheduling big meetings is appropriate only if you are making an announcement or conducting one-to-many type communications. But for working meetings where net new accomplishments are expected of the attendees, large meetings nearly ensure that nothing important will get done.

If you're inviting a distribution list to your meetings, that's your first mistake. Remember the two pizza rule.

What exactly is a "big meeting?" Amazon and Jeff Bezos provide the answer. If you require more than two pizzas to feed the attendees, your meeting is too big. Organizational Psychologist Richard Hackman's research tells us that inefficiencies in communicating expand as group size expands. Hackman points out that optimal working size for meetings is 5 people, 10 at most. So if you're inviting a distribution list to your meetings, that's your first mistake.

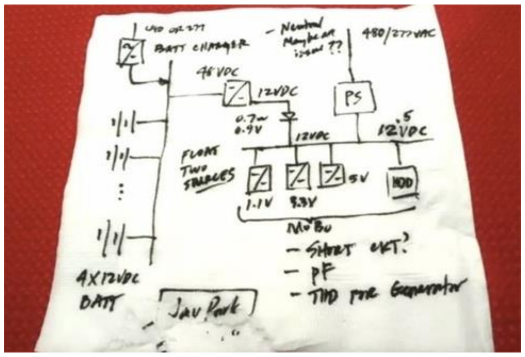
If you feel obligated by corporate politics to invite more and more stakeholders, consider using R-A-P-I-D to help you decide who to invite (and who to spare having to attend another meeting). RAPID was developed by Bain & Co. to systematize decision-making and to eliminate miscommunication and misunderstandings. Here are the R-A-P-I-D roles:

Recommend:	In a software development example, the "Recommender" is likely the one who calls the meeting. This person may be an IT Ops Manager who has ownership over the success of a major application release. This person may package alternative approaches for a Decider.
Approver:	Approvers are often a sign-off for regulatory or other compliance questions. This role may not always be required.
Perform:	People actually performing the work may include people from both Dev and Ops. These attendees are generally tasked with doing the job once the Decision has been made.

Influence:	The Influencer should provide useful background information to the Recommender for their consideration.
Decide:	In smaller companies, this may be the same person as the “Recommender”. In larger organizations, this may be someone who can serve as a tie-breaker.

What’s missing from that list? All of the “stakeholders” and peripheral groups who might be *interested* but not *invested* in the success of the project. Inviting too many stakeholders is an invitation to slow down progress and to waste your time and theirs. Your Decider should make their decision in consideration of the other stakeholders in the organization. So those stakeholders do not need to be in the room with you…eating your pizzas. With only two pizzas, you’ve fed the handful of people who actually need to attend and accomplish something at your working meeting – and you’ve avoided the ‘death by meeting’ trap that screws up DevOps.

Mistake #10: Working



Open Compute Concept original datacenter electrical design - Jay Park of Facebook

Fast moving teams can be tempted to “keep their noses to the grindstone” to stay focused and get things done in the office. This is a mistake. Working in the office is reasonable and it’s productive, until it’s not. For mainstream work, the central location of the office provides its own efficiencies. However, the office itself invites a type of corporate-speak and corporate-think.

Offsite venues offer a change of scenery that in turn opens new conversations and new thinking. Once colleagues are sitting shoulder to shoulder at a restaurant or bar, they will have a totally different type of conversation that they would never have via email or in a stuffy conference room. Truths get told and new ideas get hatched. How many breakthrough ideas have been hatched on a cocktail napkin versus yet another scheduled office meeting?

Do you want to know what is really going on in the office? Get out of the office and find out. You learn more about what’s working and what’s not working. Something you may have thought was working just fine is actually a problem. Or maybe you thought you were the only one struggling with a process, only to find out everyone else is struggling too. This is especially true in IT, where Development and Operations have a main goal of software delivery with entirely different approaches of how to achieve it.

If your DevOps team is having problems with another team when implementing a new process or tool, consider taking someone from the other team to happy hour to talk things through. You may more easily understand their hesitations or concerns. You might call this method your "Return on Scotch" – the return on investment someone gets from dropping everything and having a scotch (or maybe a beer) with a coworker to have a candid conversation and solve a work related problem outside the workplace.

What's your "Return on Scotch"?

Staying in your office, approaching the same problems in the same way with the same corporate-speak is a tried and true way to screw up DevOps.

Summary

There are no guarantees for launching a successful DevOps program or creating a thriving DevOps culture. The list of pitfalls outlined here is long but it is certainly not complete. But at least these mistakes are quite common and therefore quite predictable, and preventable. Just like a sports team, victory may simply come down to eliminating unforced errors. Knowing what not to do in your DevOps journey is as important as knowing what to do.

About the author:

Kristy McDougal is a co-founder of OrcaConfig where she leads Product Engineering for Orca, an application-centric and middleware configuration management software solution for DevOps teams. Prior to launching Orca, Kristy was a Senior Pre-Sales Consultant within BMC Software's world-wide DevOps specialist team. Before BMC, Kristy held technical positions at VaraLogix, Virtual Bridges, Global Foundries, and Advanced Micro Devices. Her experience includes Unix systems administration, systems engineering, pre-sales consulting, and post-sales services.

Kristy is ITIL Foundation certified and is an Electrical Engineering graduate of the University of Texas at Austin.