NEW TECH FORUM

By Zubin Irani, InfoWorld

SEP 8, 2016

5 common pitfalls of CICD -- and how to avoid them

What's the secret to devops success? Start with continuous integration and continuous deployment

Devops may be one of the haziest terms in software development, but most of us agree that five activities make devops what it is: continuous integration, continuous delivery, cloud infrastructure, test automation, and configuration management. If you do these five things, you do devops. Clearly, all five are important to get right, but all too easy to get wrong. In particular, continuous integration and continuous delivery (CICD) may be the most difficult devops moves to master.

Continuous integration (CI) is a process in which developers and testers collaboratively validate new code. Traditionally, developers wrote code and integrated it once a month for testing. That was inefficient -- a mistake in code from four weeks ago could force the developers to revise code written one week ago. To overcome that problem, CI depends on automation to integrate and test code continuously. Scrum teams using CI commit code daily at the very least, while a majority of them commit code for every change introduced.

Continuous delivery (CD) is the process of continuously creating releasable artifacts. Some companies release to users once or even multiple times a day, while others release the software at a slower pace for market reasons. Either way, the ability to release is tested continuously. Continuous *deployment* is possible thanks to cloud environments. Servers are set up such that you can deploy to production without shutting down and manually updating servers.

Thus, CICD is a process for continuous development, testing, and delivery of new code. Some companies like Facebook and Netflix use CICD to complete 10 or more releases per week. Other companies struggle to hit that pace because they succumb to one or more of five pitfalls I'll discuss next.

Pitfall No. 1: Automating the wrong processes first

This trap tends to strike organizations making the shift from waterfall development to devops. New organizations have the advantage of implementing CICD from scratch. Existing companies have to journey gradually from manual to highly automated development. The full transition can take several months, which means you need to be iterative in how you adopt CICD.

When you ask, "Does this need to be automated now?" run through the following checklist:

1. How frequently is the process or scenario repeated?
2. How long is the process?
3. What people and resource dependencies are involved in the process? Are they causing delays in CICD?
4. Is the process error-prone if it is not automated?
5. What is the urgency in getting the process automated?

Using this checklist, you can prioritize the steps in a CICD implementation. First and foremost, automate the process for compiling code. Ideally, you will integrate code multiple times per day (1). Manually, the process takes a few minutes to a couple of

hours (2). That stalls output until the compiler finishes the task (3). It is also susceptible to human error (4), and because CICD is a pipe dream without automated integration, this is urgent (5).

We can run the same checklist on testing. As you transition to CICD, you might wonder: Should we automate functional testing or UI testing first? Both will be repeated at least once per day (1). Both can take two to three hours for a medium-sized application (2). But they involve multiple dependencies (3). If you automate functional testing, you may not have to update the automation script that frequently. The UI, on the other hand, often changes and thus requires frequent script changes. Although both are error-prone (4), you should prioritize functional testing before UI testing to make the best use of your resources (5).

Let's do this one more time with the process of setting up environments. This scenario is only repeated frequently if you're on a hiring spree or experiencing heavy churn (1). It's a rather time-consuming process that can take several hours if not days (2). New team members can't do anything helpful without environments, so clearly there is a dependency and delay (3). I wouldn't say that the process is error-prone (4), so is it still urgent (5)? I lean toward yes, but I'd still prioritize integration and functional testing first.

There is no such thing as overautomating. If you had unlimited resources, you would automate everything possible. That said, you *cannot* achieve total test automation. Sometimes you can break down tasks into smaller segments and automate in patches. Sometimes you should simply document the process in detail and execute it manually.

Pitfall No. 2: Confusing continuous deployment for continuous delivery

Continuous deployment is the concept that every change made in the code base will be deployed almost immediately to production if the results of the pipeline are successful. This is terrifying to most organizations because rapid product changes can scare away users.

Companies believe that if they do not practice continuous deployment, they are not doing CD. They fail to distinguish between continuous deployment and continuous delivery.

Continuous delivery is the concept that every change to the code base goes through the pipeline up to the point of deploying to nonproduction environments. The team finds and addresses issues immediately, not later when they plan to release the code base.

The code base is always at a quality level that is safe for release. *When* to release the code base to production is a business decision.

Whereas continuous deployment unsettles most organizations, continuous delivery resonates with them. Continuous delivery gives them control over product rollout, functionality, and risk factors. There is time for alpha testing, for beta customers, for early adopters, and so on.

Pitfall No. 3: Lack of meaningful dashboards and metrics

In CICD implementations, the scrum team may create a dashboard before members know what they need to track. As a result, the team falls prey to a logical fallacy: "These are the metrics we have, so they must be important." Instead, perform a progressive assessment *before* designing a dashboard.

Different members of an IT organization, and even various members of a scrum team, have different priorities. For instance, the folks in a network operation center (NOC) love red, yellow, and green indicators. Such traffic light dashboards enable NOC staff to distinguish problems without reading dense text or taxing their analytical abilities. Traffic lights help make hundreds of servers manageable.

You might be tempted to use a traffic light dashboard for CICD too. Green, we're on track. Yellow, we're off track, but we have a plan to address that. Red, we're off track and likely need to change our objectives.

That dashboard is probably useful to a scrum master, but what about the VP of development or the CTO? If a scrum team has 350 hours of work ahead for a two-week sprint, and its 10 members are accountable for 35 hours each, they would receive a

corresponding number of story points. Upper management might be less interested in the status of story points and more curious about the "burndown" rate: the speed of task completion. Do team members carry their loads? How quickly? Are they improving over time?

Unfortunately, burndown rates could be misleading if the various stakeholders don't understand the scrum team's agreed-upon habits. Some teams burn down points early as they go. Others wait until near the end of the sprint to burn down open points. The dashboard should take that into account.

If you can assess what data everyone wants and establish a standard narrative for what that data means, then you can design a useful dashboard. But don't obsess over substance at the expense of appearance. Ask how stakeholders want it to look. Would graphs, text, or numbers be best?

These are the considerations to investigate in a progressive assessment. They illustrate how tricky it is to make a useful CICD dashboard -- and to make everyone happy. Too often, the most vocal team member hijacks the process, and others feel frustrated that the dashboard meets only one person's preferences. Listen to everyone.

Pitfall No. 4: Lack of coordination between CI and CD

This pitfall takes us back to our consensus definition of devops, which holds that CI and CD are two different items. CI *feeds* CD. Implementing a decent CI pipeline and a full CD system takes months and requires collaboration. Quality assurance, the devops team, ops engineers, scrum masters -- all must contribute. Perhaps the toughest aspect of CICD is this human factor rather than any technical challenge we've discussed. Just as you can't program a healthy relationship between two people, you cannot automate collaboration and communication.

To gauge this level of coordination, benchmark your CICD process against the best in the business. Companies like Netflix can complete integration, testing, and delivery in a matter of two to three hours. They established a system that passes code from hand to hand without indecision and discussion. No, it's not 100 percent automated because that is impossible with current technology.

Pitfall No. 5: Balancing the frequency of running CI jobs and resource utilization

CI jobs are supposed to be triggered for every change that is introduced in the code. Successful jobs allow the changes to go through while failures reject the changes. This encourages developers to check in smaller chunks of code, triggering more builds in a day. However, unnecessary CI jobs consume resources, which wastes time and money.

Because this process involves a lot of resource utilization (CPU, power, time), the software should be broken into smaller components to create faster-running pipelines. Or the CI jobs should be designed to batch check-ins that are first tested locally. The goal is to find a balance between the frequency of executing CI jobs and the utilization of resources.

Keep the goal in sight

As we dig into the pitfalls of CICD -- complete with all of its esoteric terminology -- it's easy to lose sight of *why* this matters. Ultimately, CICD is essential because it meets business goals.

Technology executives know that continuous evolution, quick fixes, and quality results create and retain customers. They know that a failed release invites a bludgeon to App Store reviews, and regaining high reviews is harder than keeping them. Devops might create a better work experience for your team, but that is not why companies implement devops.

Simply put, the pitfalls of CICD are worth reviewing because billions of dollars are at stake. While I don't suggest you add a stock ticker or App Store review tracker to your CICD dashboard, I do urge you to stay cognizant of this. A lot depends on the minutiae of CICD.

Zubin Irani is co-founder and CEO of cPrime, a full-service consultancy that implements agile transformations and delivers agile solutions for more than 50 Fortune 100 firms and many of Silicon Valley's biggest employers.

New Tech Forum provides a venue to explore and discuss emerging enterprise technology in unprecedented depth and breadth. The selection is subjective, based on our pick of the technologies we believe to be important and of greatest interest to InfoWorld readers. InfoWorld does not accept marketing collateral for publication and reserves the right to edit all contributed content. Send all inquiries to newtechforum@infoworld.com.