circleci

The New Metrics CEOs Track to Increase Speed

To move fast, you have to know more than just churn

No CEO would think of running a startup without knowing MRR, churn, or burn rate. But in the battle to move fast, you've got to be able to push a product update out immediately when something in the market changes (as it always does). If you can be fast enough that you can take advantage of those changes--often measured in hours--you'll gain key market share against any competitor that can't adjust as readily.

To maximize speed, CEOs should think about non-traditional metrics like commit-to-deploy time, queue time, and test coverage. These metrics can give CEOs a sharper image of their software trajectory, and of what may be holding them back.

Commit-to-Deploy Time (CDT):

This is the time it takes for code to go from commit to deploy. In between, it could go through testing, QA, and staging, depending on your organization. The goal of measuring CDT is understand how long it's taking code to get from one end of the pipeline to the other and what roadblocks you're encountering, if any.

Ideally, if you're doing CI best practices, tests are good quality, have been automated, and you can get from commit to deploy-ready status in mere minutes, even seconds for a microservice. If you have a largely manual QA process, that will likely mean your commit-to-deploy time is longer and can reveal where you have room to improve.

Most fast-moving organizations (e.g. Facebook, Amazon) deploy hundreds of times a day. For smaller organizations, daily deployments would be a good goal. The smaller your commits are, the faster they can get into production, and the faster you'll be able to fix things when they go wrong... and at some point, they definitely will go wrong. More frequent deployments will also get your team accustomed to doing so, which will hopefully mean they'll get better and faster at doing it.

How do you measure CDT? Timestamps. For every commit that makes it through code review and merged, record when that happened. Using git means you get this sort of granular information for free. On the deploy side, you'll do the same thing — deployed means your users can see/use it. Dig into the documentation for Heroku or AWS to figure out the exact time the code went live. The "average" CDT really depends on the size and complexity of your application, but less is nearly always more, so you'll want to do what you can to push that time down.

These types of improvements could be more of the technical side (e.g. our tests are flaky) or more process-oriented (we use complex integration tests where only unit tests are needed) or some combination of the two. At any rate, your goal should be to incrementally improve your commit-to-deploy time.

> "For smaller organizations, daily deployments would be a good goal. The smaller your commits are, the faster they can get into production."

Build Time

One of the worst — but somewhat unavoidable — wastes of time is when engineers and developers sit around waiting for tests to finish running. The bigger and more comprehensive these tests, the more time they tend to take. No matter how you stack them, you've got well-paid staff checking Twitter or getting distracted by GIFs on Slack instead of coding.

Let's look at the costs: if you have 2 developers waiting on a test, both paid $50/hour (~$100,000 per year), then a 10-minute build time will cost you about $17 in lost productivity. That may seem like chump change, but if each dev runs 5 similar tests a day, it would add up to $833 a week ($43,000 a year). To put that in larger context, some tests can take an hour to run. In addition to these obvious immediate costs, there's the hard-to-quantify "lost opportunity" cost of not getting to market sooner, and the mental "switching cost" of devs waiting around, then hopping back into development.

> "Placing related tests together can prevent unnecessary setup and teardown phases."

Build time actually encompasses both the running of tests, as well as any preparation needed to make those tests run. These preparations include creating a test database, seeding that database with test data, and setup/teardown of the database between test suites so you're not tainting your data.

Like CDT, a "good" build time depends on context, but lower is nearly always better. Placing related tests together can prevent unnecessary setup and teardown phases. CI best practices also recommend placing faster tests first in the workflow: if there are errors, you want to catch them as fast as possible, cancel the build, and fix them.

Queue Time

More subtle than build time is the amount of time engineers have to wait before their build even executes. This metric is highly dependent on the size of your organization, as well as the number of simultaneous features in development. It can be further exacerbated by running builds on each commit which is, ironically, exactly what you should be doing.

But long queue times are expensive. While engineers could work on another project, they run the risk of losing valuable context on the feature they just wrote. Instead of focusing on their next project, they'll be tied to the change they're waiting to test. What makes this worse than build time is that engineers can interrupt a build if a test fails, cutting short the "maximum build time"; queue times grant no such shortcuts.

For every change that's ready to be tested, track how long it takes to get to the front of the queue. This can be as simple as asking engineers to record how long they spend waiting before their build begins executing.

Keep queue times low and allow engineers to move on with their lives.

How Often Master Is Red

Developers use the 'master branch' as a starting point for all new work. This is the main source of truth, so if it goes "red", it means everyone's stuck. No one can ship. The entire delivery pipeline grinds to a halt. If you have someone who's trying to ship a key security fix, they'll have to wait until 'master' is fixed.

There are many organizations where if 'master' breaks, it can take hours to fix. That's not where you want to be. If organizations don't focus on fixing a red 'master' ASAP, it indicates a few things:

Master can easily go red if you have unstable test suites or flaky tests. The percentage of your code-

base that's covered by tests will also determine the likelihood of master going red: bad test coverage leads to broken builds.

Long-running feature branches can end in sadness and merge conflicts. The bigger the changeset, the more likely it is you've diverged from what's happening elsewhere in the code. One CI/CI principle to help avoid this is committing small sets of changes rather than doing "big bang" merges.

Leaving master in a bad state can also simply be an indication of poor dev practices/culture. Your organization might not have enough urgency around fixing the source of truth in the first place. While master is red, it creates a bottleneck for commits, increasing recovery time and delaying development.

"While master is red, it creates a bottleneck for commits, increasing recovery time and delaying development."

One principle of continuous delivery is an emphasis on always keeping software "green": in a deployable state. And if it's not green, you fix it the minute it breaks instead of letting it linger. Whoever broke it should completely concentrate on resolution, bringing in help from other team members if needed. This is somewhat related to the "red button" idea from Kanban: anyone can and should press the red button any time they notice a defect so things can be fixed as soon as possible. That said, once the red button is pushed, everyone needs to get the "assembly line" moving again ASAP.

How do you measure this? Every time master breaks, start a cumulative timer. Divide that number by the rest of the time in the year so far. That's the percentage of time master spends "red". For more granularity, you could break this down by month or day. Or: calculate the average time it takes to get master from red back to green. This should be no more than an hour.

Keeping master green is all about preventing bottlenecks and keeping your company nimble.

Engineering Overhead

This may seem like COGS, but it's a little different. Engineering overhead includes things like headcount and how much is spent on things like licenses and AWS, but it also includes tool maintenance. While many CEOs track the cost of their tools per seat, they don't look at how much time it takes to configure, maintain, or monitor those tools. Do you have people on your engineering team whose main function is to maintain a team the tool uses? How many hours are engineers spending on this type of process work vs. output?

If a tool is consistently taking a lot of time and attention to function, you might want to re-assess its value. The amount of time engineers spend on tooling reduces the amount of time they spend working on the product. Engineers want to work with the best tools: don't give them an excuse to leave by compromising on quality of infrastructure or tooling.

With an emphasis on feature development, time spent maintaining existing tools can fly under the radar. A great solution here is to require engineers to estimate and track maintenance work in an issue tracker like JIRA. Comparing the time spent on these tickets to feature work can give you a rough idea of how much engineering time is being chewed up by inefficient process or tooling.

What Else?

These are five of the larger non-traditional metrics CEOs should know, but here are a few more questions they should get answered to ensure they're staying competitive:

- How many times a day are developers merging to 'master'?
- How often is my code in a releasable state?
- How much of my codebase is covered by tests?
- Have I optimized my tooling and infrastructure?
- What are the potential speed gains and savings of alternative tooling/infrastructure solutions?