circleci

The Building Blocks of a High-Performing DevOps Culture

METRICS AND METHODS FOR IMPROVING ENGINEERING VELOCITY AND EFFICIENCY

WHY READ THIS GUIDE?

Today's engineering leaders are under immense pressure to ship code. Customers have high demands and businesses need new ways to evolve, grow, and adapt. Employing DevOps principles such as continuous integration and delivery (CI/CD) enables teams to ship high integrity code, faster. But relying exclusively on tools will not get you there. Across thousands of teams, we've seen that the top teams have something in common: they've fostered a culture in which DevOps practices can thrive. This guide will show you what you need to build a DevOps-friendly culture in your organization.

This guide will cover:

- **TRACKING THE RIGHT SUCCESS METRICS**
- **STRUCTURING OPERATIONS**
- **PRACTICES OF TOP-PERFORMING TEAMS**

> Continuous integration (CI) and continuous delivery (CD) embody a culture, set of operating principles, and collection of practices that enable application development teams to deliver code changes more frequently and reliably. The implementation is also known as the CI/CD pipeline."

from InfoWorld

Almost every engineering team experiences tight deadlines and pressure: task backlogs are endless, and there aren't enough hours in the day to manage a full workload. There's too much to do and too little time.

It's this reason why engineering teams are moving to a model of continuous integration and delivery: instead of launching updates in infrequent batches, you can deploy updates on a continuous basis. The idea is that you can shorten the path from ideation to execution and deployment. When teams find momentum, more gets done. Consider the example of Etsy, as an example. According to the company's engineering blog, the people responsible for developing code are responsible for shipping it. A group of engineers, known internally as a push train, work together to release updates up to 50 times per day.

"This strategy has been successful for a lot of reasons, but especially because each deploy is handled by the people most familiar with the changes that are shipping," writes Sasha Friedenberg, an engineer who has been with Etsy for more than five years. "Those that wrote the code are in the best position to recognize it breaking, and then fix it. Because of that, developers should be empowered to deploy code as needed, and remain close to its rollout."

To empower your engineering team, you need the right workflows and processes. But these routines aren't always easy to develop. You likely don't have time for trial-and-error, either. It's important to remember that DevOps—a synthesis between development and operations— is a relatively new business function, which means that many companies may not necessarily have their technical workflows in alignment.

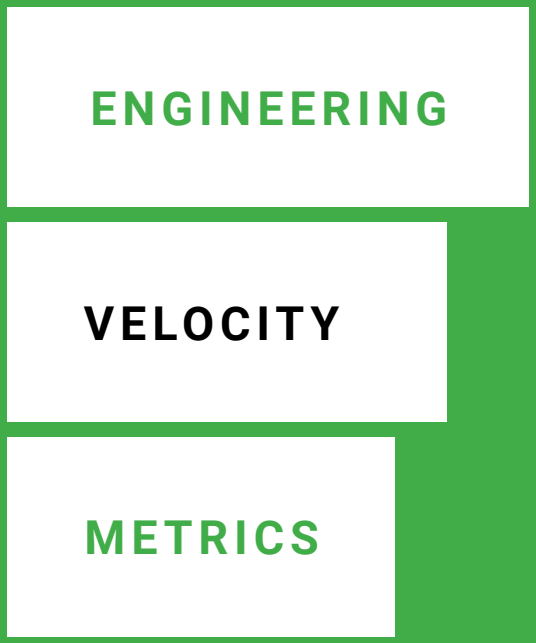
We wrote this guide to help engineering team leaders unite people and processes, without arduous trial-and-error. Part I introduces metrics that your team can monitor. In Part II, you'll get tips and best practices from CircleCI customers.

PART I

HOW TO MEASURE TEAM ALIGNMENT

The CircleCI team is constantly evaluating and analyzing the practices used by our customers to understand what the most successful engineering cultures share in common. In one study, we queried a sample of projects on GitHub and Bitbucket, all built on CircleCI's cloud platform, and matched these entities to Global Alexa Internet Rankings. We enriched this data with ClearBit. We compared the top 10% of Alexa Internet ranked organizations to the list as a whole, to understand what top performers do differently. We found there were 3 metrics that formed the foundation of our top teams' ability to move quickly.

Let's look into these key velocity metrics:


ENGINEERING
VELOCITY
METRICS

MAINLINE BRANCH STABILITY

DEFINITION

This is the master mold for every feature branch that your developers create. CircleCI measures it as the percentage of wall-clock time as a project's default brand spent in a failed state.

VALUE

Mainline branch stability is a measure of deployment readiness. If your mainline branch isn't stable, you're not going to be able to push code live.

BENCHMARKS

Median stability is 98.5%, and the top percentile is 99.9%.

80% of all organizations keep their master branch stable 90% of the time

DEPLOY TIME

DEFINITION

After code has been written, reviewed, and tested, it still needs to be delivered to users. The time it takes for code to move from the mainline branch to production can range from a few minutes to many hours.

CircleCI measures deploy time as the number of wall-clock minutes in between queuing and completing a build.

VALUE

Deploy time is a measurement of deploy cost. The lower the deploy time, the less expensive it is to change your product.

Engineers waste less time waiting for deploys, allowing them to start new work more quickly. Product owners can conduct more experiments and build more prototypes.

Customers see changes faster, and bugs can be patched within minutes of being spotted.

BENCHMARKS

80.2% of organizations deploy in under 15 minutes. The fastest organizations (95th percentile) deploy in 2.7 minutes, while the median is at 7.6 minutes.

From there, a long tail extends to 30 minutes for the bottom 5th percentile. Among top performers (10th percentile of AIR orgs), 80% deploy in less than 17 minutes, with the top 5th percentile at 2.6 minutes.

The median for these organizations is 7.9 minutes, and the bottom 5th percentile is at 36.1 minutes.

DEPLOY FREQUENCY

DEFINITION

This metric captures deployment speed. It's a signal that releases are making their way to the market—and problems are getting solved—faster.

VALUE

CircleCI measures deploy frequency as the median number of "default-branch" builds run on its platform, with a valid deploy step per week.

BENCHMARKS

75% of all organizations deploy their most active project less than 13 times a week. Top performers (95th percentile) deploy their mainline branch 32 times per week; that's over 5 times the median and nearly 24 times the bottom 5th percentile.

Velocity is only half the story; you're going fast, but you need reliable metrics to capture the value you create. How effective is your team in keeping up with the needs of your market? Are your efforts yielding an impact? While engineering velocity metrics tell you whether your team's operational processes are in good working order, efficiency metrics help ensure that you're headed in the right direction.


ENGINEERING
EFFICIENCY
METRICS

COMMIT-TO-DEPLOY TIME (CDT)

ENGINEERING

EFFICIENCY

METRICS

DEFINITION

Engineering overhead includes things like headcount and how much is spent on things like licenses and AWS, but it also includes tool maintenance.

VALUE

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.

BENCHMARKS

If a tool is consistently taking a lot of time and attention to function, you might want to reassess its value. The amount of time engineers spend on tooling reduces the amount of time they spend working on the product.

BUILD TIME

ENGINEERING

EFFICIENCY

METRICS

DEFINITION

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.

VALUE

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

BENCHMARKS

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

QUEUE TIME

DEFINITION

More subtle than build time is the amount of time engineers have to wait before their build even executes. Long queue times are expensive.

VALUE

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.

BENCHMARKS

This metric is highly dependent on the size of your organization, as well as the number of simultaneous features in development.

MASTER DOWNTIME LENGTH

DEFINITION

One of the biggest 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.

VALUE

If you have two 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.

If each developer runs five similar tests a day, it would add up to $833 a week ($43,000 a year)

BENCHMARKS

Build times vary between tests, teams and organizations. Aim to keep this number as low as possible to conserve valuable time.

ENGINEERING OVERHEAD

DEFINITION

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.

VALUE

Ideally, if you've implemented continuous integration (CI) best practices and you have sufficient automated test coverage, 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 you have room to improve.

BENCHMARKS

Fast-moving organizations make hundreds of deployments a day.

Slower-paced teams make deployments daily or even weekly.

The range varies by business model and team structure.

Velocity and efficiency metrics, together, can help you assess how effectively your engineering team is operating. The goal is to reduce overhead, eliminate downtime, and make sure that work gets done as smoothly as possible. The underlying story behind these numbers is how well-equipped the people on your team are to do their jobs.

An effective culture means giving engineers the resources, support, and tools that they need to do their best work.

PART II

18 IDEAS TO RESOLVE DEVOPS BOTTLENECKS, INCREASE VELOCITY, AND IMPROVE EFFICIENCY

When we asked our top engineering teams what contributed to their success, we heard the same practices come up again and again. Each of these ideas has a direct impact on one or more of the velocity and efficiency metrics referenced above.

Remember that every DevOps culture is unique: before implementing any new processes, you'll want to test them in a controlled setting. Be creative in how you tackle your own bottlenecks and challenges.

The following recommendations can help you get started.

1 **Document everything.** Mistakes are inevitable. If something breaks, figure out why. Understand what you can improve, so tomorrow is a better day. Keep learning. Keep evolving. Document everything, so you can share your observations with your team during meetings or through updates via email.

2 **Make lots of changes rather than one big one.** Follow the rules of probability. If you spread your updates into multiple releases, you're less likely to break something in a big way. Don't put all your deployment eggs in one basket.

3 **Implement peer review processes.** Avoid error by pairing developers together. Review one another's work before your code has a chance to reach its staging environment.

4 **Focus on team morale.** Even if feature releases don't warrant a discussion, it's still important to share status updates. When teams get things done, they feel efficient. Encourage everyone to move on and accomplish more.

5 **Release during downtime.** Consider making deployments during low-traffic times, at nights and weekends, to avoid the potential for error. This means you may need to operate in a non-feature-flag world. Your press releases won't coincide with your code shipments, for instance. But keep in mind that feature flagging isn't going to save you from performance regression or a problem. Awareness of a feature can never be taken back. Choosing to release during downtime—or a time of high-traffic—will depend on the parameters you've set up for safety. Remember that no team is immune to error, no matter how talented and experienced your developers are.

6 **Test in production.** The world moves quickly, which means that it doesn't always make sense to run tests before you launch—especially if you're considering moving production data into staging environments. Instead, you can launch and test the feature on a percentage of your customer base. You're likely to get more accurate results and can pull the feature if something goes wrong.

7 **Keep code review discussions high-level.** If a test passes, the risk of something breaking is low. Code review processes will be more valuable if you spend your time understanding changes that are happening on a high level. Give engineers the opportunity to talk about architecture or where the codebase is going long-term, so everyone can focus on efficiency, velocity, and strategy.

8 **Run lightweight demos.** Especially when bringing teammates together to review new features, keep discussions short and scope-focused. Limit discussion to five minutes. With multiple eyes on the same initiative, you'll surface potential errors quickly.

9 **Pair-develop.** At some of the organizations we analyzed for this guide, engineers develop code in pairs. This may eliminate the need for code reviews altogether. You may be able to prevent errors by getting two people on an initiative, off-the-bat, reviewing each others' work as they move along. If you don't feel confident eliminating your code review process, you can run intermittent validity tests, for quality control.

10 **Create self-contained groups.** Teams of 6-7 people are in a strong position to try ideas on their own, without running into hiccups. Following a clearly-defined company protocol, these teams can have the freedom to experiment with their own processes, as a testbed for trying new initiatives companywide.

11 **Run regular retrospectives.** Once a month (or on a regular basis), sit down to analyze what went wrong—and what went right. These meetings make it easier for teammates to learn from one another, adapt to fast-moving changes, and improve upon processes. These meetings will help ensure momentum while alleviating bottlenecks. Celebrate the value that your team created.

12 **Avoid blame games.** DevOps environments have the potential to be high pressure and fast-paced. Mistakes are bound to happen. When something goes wrong, encourage discussions. Focus on learning from the problem instead of assigning blame.

13 **Protect your users.** Errors have the potential to impact lives. If you're deploying code to manage bank accounts or medical documentation, you need to have a fail-safe or manual recovery plan for problems that may arise. This process may include backups and restorations.

14 **Create bowling alley bumpers.** At one extreme, companies command and control from the top down. At another extreme, nobody coordinates with anyone else and mistakes can arise. One happy medium is to democratize best practices around common workflows (i.e. have a master branch that's always shippable, a bug database for tracking known problems, and SSL certificates on public-facing websites). With parameters in place, developers have the freedom to build efficiently, without the potential for a bottleneck.

15 **Keep meetings small.** At meetings, make sure that there are enough engineers to be productive but not so many that people are losing context into what others are doing. This approach will prevent repeated or conflicting work, which increase the probability of your mainline branch going red.

16 **Maintain communication around code conflicts.** It might seem logical to organize teams around parts of the tech stack (backend, frontend, operations, etc.), but these structures have weaknesses: horizontal slices of engineers have more blind spots and less insight into how their code affects others. One alternative is to organize engineers into functional groups of small sizes. This approach makes it easier for teams to meet every day, for instance in a daily standup.

17 **Connect engineering goals to overall priorities.** CI/CD should not exist in a vacuum. Prioritize feature releases and deployments by connecting engineering efforts to overall company goals. Engage in cross-department planning sessions on a quarterly or monthly basis. Figure out what needs to get done, prioritize it, and divvy it up week by week.

18 **Create alignment around user-centered goals.** Every iota of code is a step towards a larger organizational objective. For instance, your engineering team may be responsible for processing food stamp applications. In this case, your goal could be as simple as "get people food." This perspective will help teams prioritize their time, day by day.