The 2020 State of Software Delivery

2020

Data-Backed Benchmarks for Engineering Teams

By Ron Powell and Michael Stahnke

circleci

What does a high-performing software delivery team look like?

How do you know if your team is doing well?

How can engineering leaders set meaningful goals?

What metrics are important?

What are the industry benchmarks?

Executive Summary

At CircleCI, we have massive amounts of unique data on how technology delivery teams behave in the wild. This report examines over 55 million data points from more than 44,000 organizations and 160,000 projects.

We don't believe in one-size-fits-all success metrics for delivery; every team is different. However, the software delivery patterns we've observed on our platform, especially the data points from top delivery teams, show key similarities that suggest valuable benchmarks for teams to use as goals.

Our comprehensive data on engineering team performance has identified these four benchmarks:

- **Throughput:** the number of workflow runs matters less than being at a deploy-ready state most or all of the time
- **Duration:** teams want to aim for workflow durations in the range of five to ten minutes
- **Mean time to recovery:** teams should aim to recover from any failed runs by fixing or reverting in under an hour
- **Success rate:** success rates above 90% should be your standard for the default branch of an application

While some teams may have business-specific reasons for choosing different metrics as goals, any effort to improve engineering productivity or process will hinge on your ability to measure your baseline metrics and make incremental improvements.

Finally, it hardly bears repeating that 2020 has been an outlier year. We look closely at team performance during the peak of global economic uncertainty and collective anxiety and present our recommendations on how technical leaders should set their teams up for success. Namely, we suggest that leaders focus on building resilient teams and preventing individual burnout. One method for doing this is building larger teams. Bigger teams are more flexible and can tackle new feature development, support healthy maintenance, and handle urgent issues without being swamped. **Ambitious teams need both tools and people processes that can scale.**

Introduction

The rollercoaster of 2020 has highlighted the competitive differentiator that being a well-oiled software delivery team provides. The minute Covid-19 hit and every organization had to become not just remote-first but remote-only, many teams were forced to reckon with the number of manual processes they had in place. Suddenly, they could no longer rely on the fact that there was a build machine under someone's desk, and that if that machine had a problem, they could just reboot it. Suddenly, they needed to automate everything.

This idea of automation, this idea of being able to move quickly and reliably, has become not just ‘nice to have’; it's become core to what you have to do as a software delivery team.

In 2019, we showed that teams using continuous integration practices, even a little, are much higher performing than teams that don’t. That said, we want to acknowledge that CI and DevOps practices have exploded in popularity: many of your competitors are now also using these techniques.

If your team’s ability to deliver is a competitive advantage (and it is), how can you stay ahead of the curve? **This year, we set the first-ever benchmarks for teams practicing CI/CD.** What do high-performing engineering teams look like, quantitatively?

A brief history of continuous integration

The Agile method of software development was created to help software teams deliver quality products more effectively by being open and responsive to change and optimizing for shorter work sprints and delivery cycles. This is the development philosophy that made it okay, cool even, to fail and pivot. Companies like Google, Etsy, and Thoughtworks were able to turn this ideology into practice, and in doing so, became some of the most influential companies in the world. Agile became the default method of software development.

Agile was not the only development philosophy that increased an organization's ability to deliver. Agile led to very fast and iterative development teams willing to make frequent changes. But there was still a bottleneck preventing equally speedy delivery to market, namely the operations teams tasked with QA and deployment. Enter DevOps methodologies. The same iterative principles that were being applied to development teams started to be used on operations teams. Pipelines were created to take code updates and deliver them to the products in consumer’s hands. Companies that were able to adopt DevOps practices were able to differentiate themselves from the competition by their ability to move faster. Today, DevOps practices are used by most modern companies.

That brings us to continuous integration: the embodiment of Agile development in practice. Adding integration into the build pipeline meant that development and operations teams could build testing suites that run on every change to their codebase. Automated testing allowed teams to innovate iteratively and update at rapid velocity. But a key element in this ability to deliver faster is exhaustive test coverage, which means tests will return a high degree of confidence. When enabled through CI, proper test coverage gives teams the confidence to deploy at will and include automatic deployment scenarios in their pipelines. This is where we find ourselves today.

Throughout this report, we reference the idea of returning a meaningful signal from your CI pipeline. Extracting a meaningful signal requires exhaustive test coverage of your codebase. Without complete coverage, passing tests don’t necessarily indicate an absence of bugs — they can just as easily indicate an absence of test coverage (or a faulty or nondeterministic test).

Only with full test coverage can you interpret a passing run as a guarantee of code confidence. We believe exhaustive test coverage is paramount to getting value out of CircleCI or any CI tool. The 2020 State of DevOps Report found that the #1 challenge to automating change management across all groups surveyed was incomplete test coverage. For more on this topic, see Software Testing for DevOps-Driven Teams.

How do you compete?

Continuous integration and continuous delivery (CI/CD) pipelines that allow teams to move quickly and reliably have transcended the ‘nice to have.’ If you are a company that delivers software, and you probably are, then quality delivery, with speed and at scale, is the key to staying relevant and competitive. But what is the standard for teams practicing continuous integration? How do you measure and improve upon your CI practice to keep honing your ability to deliver?

As the world’s largest standalone CI provider, we have a unique opportunity to investigate what software delivery looks like quantitatively: across tens of thousands of teams, commit by commit. We looked to data from 11 million workflows on our platform to see how teams were building and deploying software in practice.

We wanted to answer this question once and for all: **What does a high-performing team really look like?**

Before we begin, let’s cover a few key terms that will help us measure performance.

- **Continuous integration:** the foundation of software delivery. CI defines the automated steps behind building, testing, and deploying software.
- **Duration:** the length of time it takes for a workflow to run
- **Throughput:** the average number of workflow runs per day
- **Mean Time to Recovery:** the average time between failures and their next success
- **Success Rate:** the number of successful runs divided by the total number of runs over a period of time

And, since this data is pulled from CircleCI, a few terms that will help you understand how we process work through our platform

- **Jobs:** a set of commands run in order.
- **Workflows:** are made up of jobs. A workflow defines the order in which a set of job are run.
- **Pipelines:** contain one or more workflows. Your pipeline orchestrates all of the activities that take your code from commit to deploy.

Optimizing **Duration, Throughput, Mean Time to Recovery,** and **Success Rate** gives a team tremendous advantage over organizations that are not as far along their path to DevOps maturity. The data that supports this is clear.

Let’s look at each of these metrics.

Duration

Duration is defined as the length of time it takes for a workflow to run. It is the most important metric in the list because creating a fast feedback cycle (including Throughput and Mean Time to Recovery) hinges on Duration. In other words, you can't push a fix, even a much-needed one, faster than the time it takes your workflow to run. Duration also represents the speed with which your developers can get a meaningful signal ("did my workflow run pass or fail?"). A short duration requires an optimized workflow.

Not all workflows produce the same end-state. For instance, some workflows only run specific tests depending on the part of the application codebase that changed. Duration, therefore, is not an explicit measure of how long it takes to deploy to production. It is just a measure of how long it takes to get to a workflow's conclusion.

The ultimate goal of CI is fast feedback. A failed build signal needs to get to developers as soon as possible; you can't fix what you're not aware of. But awareness is not the only consideration. Developers also need information from their failed builds. Getting the right information comes from writing rigorous tests for your software.

It is important to emphasize here that speed alone is not the goal. A workflow without tests can run quickly and return green, a signal that is not helpful to anyone. Teams need to be able to act on a failure as quickly as possible and with as much information as they can get from the failure. Without a quality testing suite, workflows with short durations aren't contributing valuable information to the feedback cycle. The goal, then, is rich information combined with short Duration.

Testing is often described in the binary terms of pass or fail, or green or red builds. However, it's more helpful to think in terms of three possible outcomes, where errors are the third state:

- Build passed, experiment succeeded.
- Build failed, experiment failed.
- Errors: experiment failed to conclude.

It's worth noting that most of your time should be spent in the first two scenarios: passing and failing builds. These are the two conclusive outcomes in which developers are getting valuable signal. Time developers spend on errors equates to time spent waiting for a signal that never comes. Errors can result from faults in the infrastructures or capacity starvation, which can get complex quickly. If your existing CI pipeline throws frequent errors, it may be time to revisit your tooling.

Mean Time to Recovery

Mean Time to Recovery is defined as the average time between failures and their next success. This is the second most important metric in the list: after you get a failed signal to your team, their ability to address the issue quickly is invaluable. Because Mean Time to Recovery improves with more comprehensive test coverage, this metric can be a proxy for how well-tested your application is.

Failed build, valuable signal, rapid fix, passing build: continuous integration makes these rapid feedback loops possible. The fast signals enable teams to try new things and respond to any impact immediately. Likewise, solid test coverage reduces the fear of introducing broken code into your production codebase, allowing you to challenge your engineering teams to be creative and nimble with the solutions they develop.

Throughput

Throughput is defined as the average number of workflow runs per day. A workflow is triggered when a developer makes an update to the codebase in a shared repository. A push to your version control system (VCS) triggers a CI pipeline that contains your workflow.

The number of workflow runs indicates how many discrete units of work move through your application development pipeline. One component of throughput reflects the size of your commits: are you pushing many small changes or fewer large changes? The right size will depend on your team, but the goal is to have units of work small enough that you can debug quickly and easily but large enough that you're deploying meaningful change.

We recommend monitoring throughput rates vs. setting goals. It is important to see how often things are happening, and Throughput is a direct measurement of commit frequency. Fluctuations in Throughput can occur in situations like onboarding, where two devs may work through the same tasks together and push fewer commits as a result. Establishing baseline metrics for your organization can prepare you for this type of impact, allowing you to forecast engineering productivity through these predictable events. When you encounter unforeseen circumstances, your baseline can help you determine the volume of work that went undone.

Throughput and deployment frequency are different, and both have been taken as a proxy for productivity, though this can be misleading. For example, one thing we don't explicitly measure in Throughput is whether a deploy occurred. Even the scenario of many deploy workflows per day doesn't indicate meaningful work getting done — you can push small, inconsequential changes to increase this number.

As is often the case, we're interested in quality over quantity. However, with this nuanced understanding, both throughput and deploy rate can be valuable metrics in order to track, troubleshoot, and fine-tune your ability to deliver.

When a well-tested application is in a state where it can be deployed at any time, it's because every new change has been continuously validated. Without a fully automated software delivery pipeline, a team is subject to deploy emergencies and fire drills, often at inopportune times (Friday nights, for example). With a fully automated software delivery pipeline, it is up to you how frequently (and when) updates are delivered to your end users: hot-fixes immediately; feature upgrades as they are developed; large-scale changes on a calendar set by your business demands. **A particular number of deploys/day is not the goal, but continuous validation of your codebase via your pipeline is.**

Success Rate

Success Rate is defined as the number of passing runs divided by the total number of runs over a period of time. Git-flow models that rely on topic branch development (vs. default branch development) enable teams to keep their default branches green. One important thing to note is that we expect to see high variability of success depending on whether the workflow is run on the default or topic branch. In many git-flow models, topic branches are where the majority of work is done, and therefore the majority of signal-generating passing and failing experiments. By scoping feature development to topic branches, we can differentiate between intentional experiments (where failing builds are valuable and expected) and stability issues (where failing builds are undesirable). **Success rate on the default branch is a more meaningful metric than success rate on a topic branch.**

Developing and testing new features or bug fixes off the default branch allows teams to only merge code to their default branch that has been well-tested on topic branches. This means that topic branches are where you want to get your fastest signal. It is safe to fail here without negatively affecting your end users. Beyond that, it won't even touch most of the team; only the other developers working on the same branch. A project may have many topic branches being developed simultaneously.

Throughout this report, we will refer to an application's production-ready branch as the "default" branch. Other common names for this branch include `trunk`, `master`, and `main`. We share a more in-depth discussion of branch naming and some historical context further on in this report.

What does the data tell us?

The following section explores data from CircleCI from over 11 million workflows observed between August 1 and August 30, 2020.

It represents:

- 2 million jobs run per day
- More than 44,000 organizations
- Over 160,000 projects

In 2019, we examined 30 million workflows over 90 days. This year we chose to look at a single month to reduce the amount of variability over the window of time examined.

While this report looks at these workflows in aggregate across our customer data set, individual teams looking to access these metrics on a per-workflow basis can do so in their CircleCI Insights dashboard.

Duration

Before we discuss duration data, a caveat: **workflows with the shortest possible duration are not the goal.** For example, the shortest 10th percentile of workflows as measured by duration finish in under 30 seconds. While this may provide adequate coverage in some scenarios, for most workflows, 30 seconds is not enough time for robust tests to run. Tests are what provide you with a rich understanding of any failed workflows, so test coverage (and the meaningful signal it can provide) should be the primary goal. Once that's established, you can optimize for shorter Duration.

> Additionally, while we believe Duration is the most important metric to consider, we are not interested in speed alone. The goal is confidence in quality. We assure quality with robust testing.

Half of the 11 million workflows measured finished in under four minutes. Three quarters finished in under 11 minutes. Of all the workflows measured, 95% finished in under 35 minutes.

> What Duration should your team aim for? In our experience, Durations between five and 10 minutes are going to be short enough to get information quickly and long enough to give your team the valuable insight needed when a build fails.

Shorter than five to 10 minutes means your feedback might not be complete, and you'd counterintuitively increase your time to recovery. Longer than this and you lose the attention of your developers; while waiting for long tests, they'll move onto other work. In that scenario, time spent in a failure state will extend as they must regain the context necessary to fix any issues that arise once the tests finish.

Mean Time to Recovery

The fastest Mean Times to Recovery were on the order of a couple of minutes. Once again, the absolute shortest possible time isn't necessarily the gold standard. One way to get a fix this quickly would involve a developer knowing that they just submitted bad code and then immediately pushing a fix while the original workflow is still running. Alternatively, it could indicate that two developers had pushed commits minutes apart onto the same main branch — one that failed and one that succeeded. Both of these are somewhat unlikely scenarios. As Mean Time to Recovery is the second most important metric to consider, our goals should reflect realistic targets.

In our dataset, 50% of the workflows recovered in ~55 minutes on average. That is over 5 million workflows with an average recovery time of under an hour — a substantial feat. At the 75th percentile, the set includes workflows with an average time to recovery of up to nine and a half hours — a marked increase.

Establishing the right target for your team will involve many considerations. For instance, an organization with developers all located in one town working typical business hours may not be able to resolve failed builds before leaving for home. An organization with a global development team likely has processes in place to hand off work between teams, eliminating this daily delay. The most important thing is the ability to measure where you are today. This first bit of insight allows you to start making iterative progress towards lowering your time.

> Ultimately, resolving failed builds in under an hour is highly desirable for fast resolution.

You may want to set even more aggressive targets for your team.

Throughput

Year after year, we hear in the industry that high-performing teams are deploying dozens of times per day. While those stats make for great conference talks, our data doesn't corroborate this metric -- we do not see workflows being run at this rate by the vast majority of teams.

It's important to call out that this data set includes every workflow that was run on CircleCI in the month of August. Most teams will have workflows they run rarely, some that are the workhorses of their projects, and some that get run once, discarded, but not deleted. We do not attempt to normalize workflows and whether they are in a state of active use in this report.

On CircleCI, 50% of workflows in our data set were run less than once per day (0.7 times per day on average). However, the 95th percentile includes workflows that were run over 35 times per day or about once every 45 minutes throughout a 24-hour period.

Where should your team land? Your software delivery cadence should be dependent on the software your team is building and on your business needs. Measuring your baseline Throughput and then monitoring for fluctuations will tell you more about the health of your development pipeline than aiming for an arbitrary Throughput number or comparing your stat to others.

> When you are in a position to deploy at will, the number of deployments per day is no longer an indication of whether you are a high-performing team. With good test coverage and technology that gets out of the way, your deployment frequency can increase and decrease in response to the priorities of your business. The goal is the ability to deploy anytime, not actually doing it.

Success Rate

The importance of Success Rate can be variable, depending on the structure and size of your team and what is being worked on. If a branch represents one person working on a topic or feature that no one else is relying on, a red build is less concerning. On the other hand, if the entire engineering department is blocked from doing work until the main branch is green, then preventing red builds is of paramount importance. Our data set includes workflows from both of these scenarios and everything in between.

Some in our sample saw no failed builds in the month observed, and some saw many. The median was 61% success. Splitting the workflows by the ones run on default branches vs. the ones run on non-default provides more insight. When split, the median rates are 80% for the default branch and 58% for the non-default branch. This indicates that many teams are using a git-flow model of application development where topic, or non-default branches, are where developers are experimenting and where failed builds are expected. Limiting this work to topic branches prevents errors from showing up on the default branch.

Your workflow's Success Rate will depend on the git-flow model chosen for development. If development on topic branches is the choice for your team, you should set a high target for Success Rate on a default branch with no expectation set for the non-default branch.

> A 90% Success Rate or above for the default branch is a realistic target.

Many teams are accomplishing this with workflows that run on their default branch. In any case, the ability to measure the Success Rate of your current workflows will be essential in establishing targets for your team. **Remember, failed builds are not necessarily a bad thing, especially if you are getting a fast signal, and your team can resolve issues quickly.**

How should you set targets for your team?

While there is no universal standard that every team should aspire to, our data and the software delivery patterns we've observed on our platform show that there are reasonable benchmarks for teams to set as goals. Ultimately, your ability to measure your baseline and make incremental improvements on these metrics is more valuable than chasing "ideal" numbers. To see your own team's metrics and how you measure up, visit the Insights dashboard in CircleCI.

	Median CircleCI Developer	Suggested Benchmarks
Throughput The average number of workflow runs per day	0.7 times/day	Merge on any pull request
Duration The average length of time for a workflow to run	< 4 minutes	5 - 10 minutes *
Mean time to recovery The average time between failures and their next success	< 56 minutes	Under 1 hour
Success rate The number of successful runs divided by the total number of runs over a period of time	80% for default branch	90% or better on the default branch

* Whether or not this length makes sense for your project depends entirely on what your workflow is accomplishing. Knowing your baseline Duration and striving to improve it is more important than hitting an arbitrary number.

When it comes to production, engineering maturity around mindset, tooling, and processes have come a long way. Organizations have a deep understanding of their running services, allowing them to optimize production and root cause outages faster than ever.We're excited to see that this same rigor is now being directed inward into how we build our software. As the number of contributors and the complexity of our codebases increase, it's never been more critical that CI doesn't become a bottleneck.

We've seen a similar trend as CircleCI, with organizations treating failures in the default branch as a development outage and addressing this with utmost urgency. Development outages incur serious business repercussions, associated costs, and productivity loss. What engineering leaders have known intuitively is finally being exposed quantitatively by having deeper visibility into CI.

Every high-performing engineering organization should be able to answer two questions:

- How long does it take us to fix the default branch when it breaks?
- How often do we break the default branch?

Just like in production, having a grasp of your mean time to resolution (MTTR) and mean time between failure (MTBF) allows you to improve it. And giving your engineers the proper tooling empowers them to keep the default branch green.

BRYAN LEE

Product Manager, DataDog

As a WebOps platform that helps teams treat their websites like digital products/software, we're often helping accelerate the cadence from a quarterly (or longer) release cycle, to something that's weekly, and that this is hugely transformational. Being able to evolve your design and functionality at this pace is a transformational shift away from betting it all on "big bang" relaunch. Developing these releases online with cloud based environments and CI is a huge assist in this because contrary to the Silicon Valley cliche of "move fast and break things" most teams are not allowed to break their website.

The other thing we're seeing is a lot more automation of behavioral/functional "outside-in" testing where people are automating the kind of manual "smoketest" QA using a script-controlled headless browser and visual regression testing, which has the benefit of catching edge cases that humans miss because they're looking for changes from the intended area. I saw Brett Slatkin from Google talk about using this in his development of their Consumer Survey's product way back in 2013, but we're now seeing these techniques become more and more mainstream, which is exciting.

JOSH KOENIG

Head of Product, Pantheon

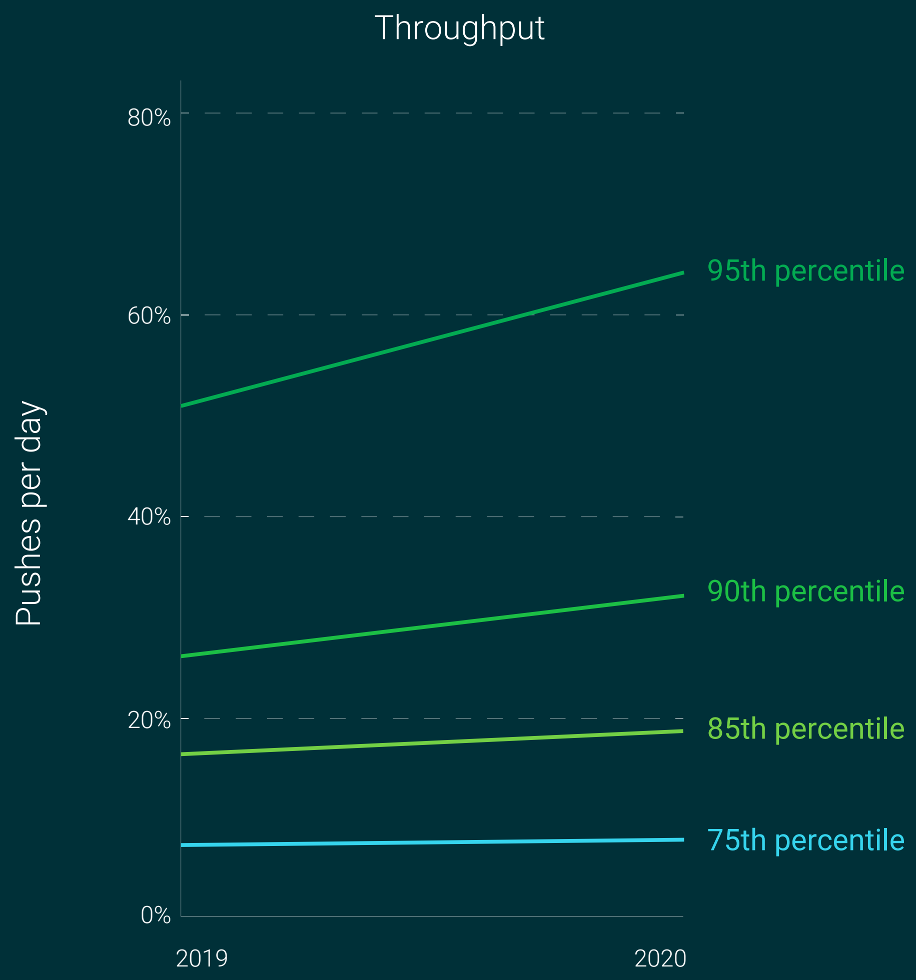
How does 2020 stack up to 2019 in our data?

This is the second year for CircleCI's annual metrics report. Now that we have two years and two data sets (which cover workflows run between August 1 and August 30 in 2019 and 2020, respectively), we were curious to examine the evolution of software delivery over the past 12 months.

What we found:

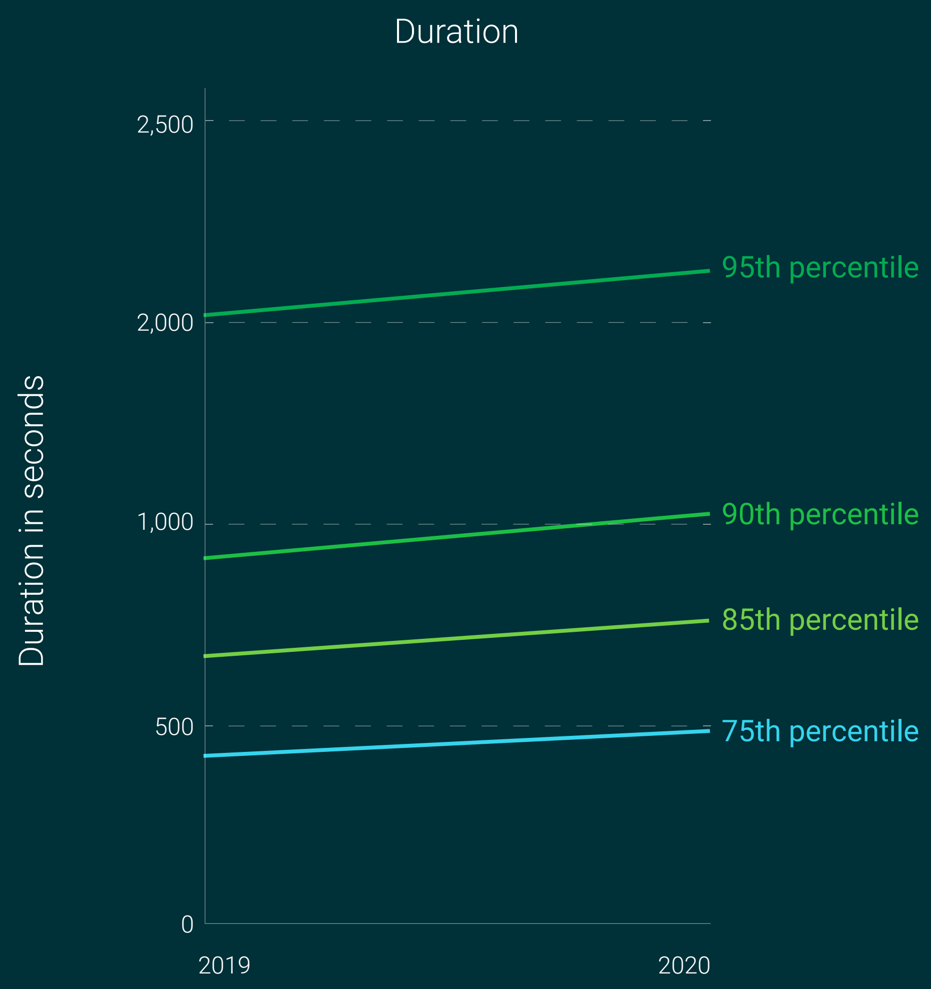
The Throughput threshold for the highest performers was raised in 2020.

Throughput values for workflows included in the 75th percentile and above increased. While there are still many workflows that are run less than once per day, the most-run workflows are run more often.



Workflow Duration is up, increased testing is a strong possibility as to why.

All workflows are running for a longer Duration. The smallest change was recorded for the fastest running workflows. We don't believe that workflows running on the order of seconds can return enough valuable information to make the CI feedback cycle efficient. The longest-running workflows are also not running much longer than they were one year ago. The workflows in the interquartile range have a duration that is about 15% longer than the workflows in this range in 2019. We suspect that increased testing practices are increasing these Durations. Keep in mind that the 75th percentile includes workflows with a duration of up to 11 minutes. We believe that this is near the ideal length of time for a workflow run. Better testing may increase your workflow's Duration, but if it reduces your time to recovery, it is a worthwhile investment. It will be interesting to see if the trend to longer Duration continues next year.



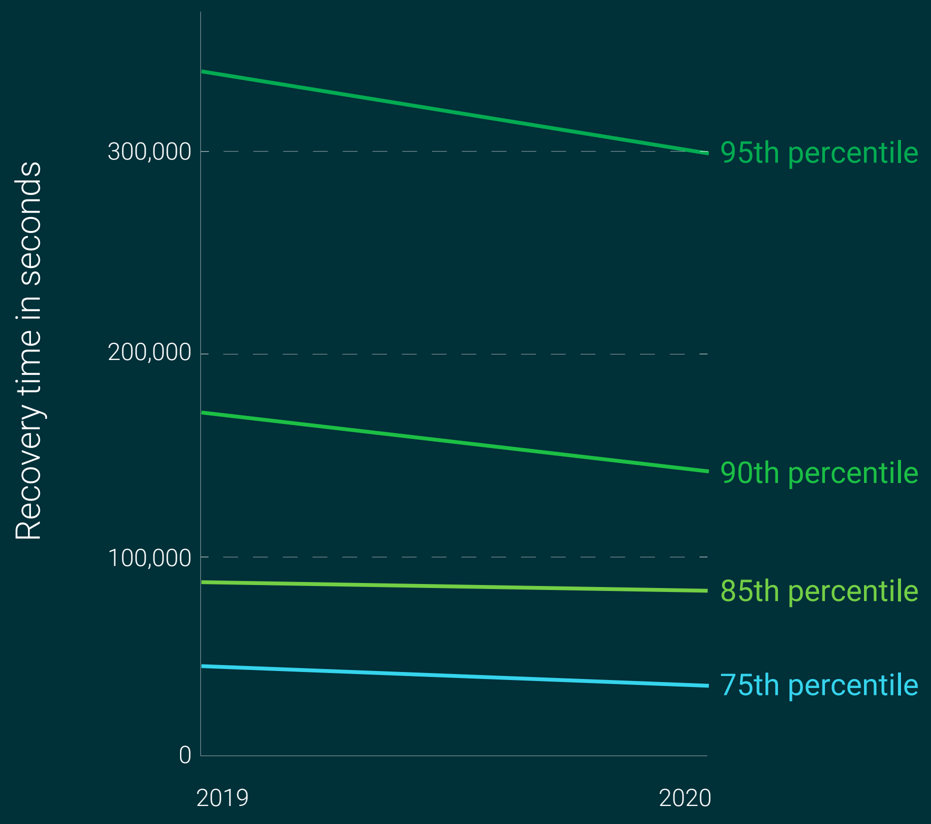
Mean Time to Recovery improved in 2020.

The longest recovery times recorded in 2020, those included in the 75th percentile and above, are shorter than those included in the 75th percentile and above in 2019. This is the signal that, when combined with the observed increase in workflow Duration, tells us that longer-running workflows are better-tested workflows and result in shorter recovery times.

Success Rate held steady from 2019 to 2020.

The median Success Rate in August of 2019 was 60%. This increased to 61% in August of 2020. As we explained previously, we expect teams using git-flow models to achieve high Success Rates — approaching 100% — on the default branch. Failure should be an expected part of the development process on topic branches.

Mean Time to Recovery



50th percentile Success Rate:

60%
2019

61%
2020

2020: With chaos everywhere, what did engineering teams do?

While we did not see (or expect to see) significant leaps in our four key metrics from 2019 to 2020, we felt it was important to examine the ramifications of global economic uncertainty, collective anxiety, and unprecedented change in the software development industry.

How did COVID-19 affect team productivity?

Did teams change their naming conventions in response to the Black Lives Matter movement?

What lessons should leaders take from this year on how to build resilient teams?

Reflections on the impact of COVID-19

While the previous section of this report looks at data from August 2020 in comparison to August 2019, this year, we took the additional step of analyzing data during the times where we expected to see the largest impact from the COVID-19 global pandemic. We observed data for the month of March, April, and May of 2020 to see how these months compared to the months of August observed for the annual report.

Here is a summary of what we found:

In times of uncertainty, developers looked to automation to create stability in systems they can control.

The peak of Throughput and Success Rates for the year was in April. Our hypothesis is that when shelter-in-place orders were instituted across the globe, teams scrambled to automate as many of their manual tasks as possible. We see Throughput and Success Rates lift to reflect that in April.

We previously discussed the difference between Success Rates on your default and topic branches. When Success Rates lift overall across branches, it suggests that teams might be taking a step back from innovation on topic branches and focusing on shoring up their business-critical systems.

There are a few other possibilities that we find interesting to explore. In our experience with developers, this group of humans cares deeply about resolving issues and finding solutions to problems. We think it’s possible that engineers followed the global trend of increased online presence during this time and tackled tickets, testing, and codebase maintenance, both as a way to solve and mitigate problems and as a means to mediate feelings of uncertainty.

In addition, April saw the end of physical conferences, as well as any kind of travel, for work or personal reasons. Companies had the rare experience of full or nearly-full workforces. A workforce at full capacity would naturally lead to an increase in productivity. Throughput was highest in April and then decreased from May to August.

The increase in Throughput was also observed even over the weekends in April. More time at home would have made it easier for developers to work at off-hours. It makes sense that Throughput would fall from this level by August. Developers burned out.

Did branch naming change in response to the Black Lives Matter movement?

The deeply problematic use of the terms “master” and “slave” in software engineering is not a new discussion, and has been an effort that many have led and worked on for years. While this discussion did not begin in 2020, it certainly gained additional prominence in reflection of larger societal movement. The May, 2020 death of George Floyd in Minneapolis, MN brought renewed attention and energy to this question of addressing systemic racial inequality. The discussion we witnessed in the software industry often revolved around the renaming of the default branch of projects from master to main or another option.

If large numbers of teams were to change the name of their default branch, this is something that we would be able to see in our data. Our expectation was that we’d see a large number of organizations rename branches. However, our data did not bear this out.

This begs the question as to why. It is certainly possible that social media falsely amplified the signal that teams were undergoing this change. It is important to note that changing the name of a default branch in your organization's repositories has massive impact and requires a lot of engineering effort to avoid breaking changes. It is likely that the engineering challenge led teams away from implementing the change. However, **GitHub is rolling out a best-practices guide** for organizations who are interested in changing the name of their default branch. Already, every new project on GitHub now has a default branch with the name ‘main.’ We will be tracking the data around this naming trend to evaluate in our report next year.

Building resilient teams in times of uncertainty

Three out of four of our key metrics show correlation between larger team size and better engineering performance. Team size (specifically, number of contributors to a particular project) matters, and in times of uncertainty, leaders should consider building larger teams to better absorb shocks and deal with change.

Duration is longest for teams of one and decreased as team size increased. It is likely that bigger organizations split work across smaller teams that are able to make smaller, more incremental changes to the code they are responsible for, keeping workflows short by testing only the relevant code. Mean Time to Recovery decreased as team size increased, too. It makes sense that more hands on deck to triage failures would bring resolution times down. This also explains why larger teams have higher Throughput.

The data suggests that adding more people to your team increases the amount of work you can push through your system (Throughput) and decreases Mean Time to Recovery. But is there an ideal team size? While the ideal team size is going to depend on experience, the total scope of responsibilities, on-call burden, and more, our data shows that somewhere between 5 and 20 code contributors is the right place to aim.

The idea that bigger teams are better is also borne out by the anecdotal experience we see in team performance. One thing 2020 has shown us is how life gets in the way of our best-laid plans. If your teams are too small and somebody goes on extended leave, has an emergency, or simply needs a break, you’re creating more kindling for the fire of burnout. An ideal team size is one that has enough hands on deck to absorb the shocks of life.

Resiliency in your platform or service, and in your technology, certainly relies on good CI/CD practices, monitoring, testing, and other DevOps processes. But the human component is essential.

> Lags in performance are often fatigue. Resilient systems and teams depend on prioritizing the people who run them.

How do you build a resilient team? In a fully distributed organization, having a team big enough to handle the day-to-day while also being able to innovate is crucial. This is especially true in a world where user expectations and demands have dramatically increased. Teams need enough individual contributors to handle the continually increasing overhead of maintenance and escalation. If a team is too small, the pace of innovation goes down.

Keeping pressure in the hose — the feeling of forward momentum — is critical for both your team and your customers. Teams need to maintain a steady heartbeat of innovation to provide customers with confidence that progress is being made.

A resilient software delivery team will be able to balance three priorities:

- Building user-focused features — you're working on something that will improve the lives of your users.
- Technical investment — what do you need to do to maintain your system? Where is it going to tip over? Refactoring and working through technical debt are critical tasks of successful teams.
- Availability for escalations and defects — is the system not behaving the way it's supposed to? Did the system break, preventing users from accomplishing their goals?

How a team splits their time between those three priorities will depend on the maturity of the product, number of users they serve, ease of creating fixes, and other factors unique to their business.

As a system gets bigger, the amount of time dedicated to maintaining the system naturally goes up. You have to increase the size of your team to maintain the ratio of customer-focused work. Eventually, the team gets too large, and that's when you split them into smaller groups. Think of Amazon's Two Pizza Rule.

Of course, there's an upper size limit where the cost of communication and coordination gets high, but teams should strive to have enough developers to both maintain the service and continue to innovate it.

To build resilience, invite a friend to your project. Expand your pool of contributors. Our data proves that creating software is a collaborative team sport.

When you think of the year we've had in 2020 it's particularly important to consider issues such as burnout and empathy in developer teams. It's all too easy for the industry to have an obsession with speed and velocity at the expense of everything else. If 2020 has shown us anything it's that people need to be able to work in ways where they feel comfortable, with reasonable expectations and support in order to avoid burnout. One really useful aspect of the report's findings, then, is a focus in setting reasonable key metrics and performance benchmarks.

CircleCI explicitly argues that number of deployments per day is not the right metric for high-performing, resilient teams. Obviously there are organizations doing hundreds of production deploys per day - we tend to practically worship them almost as an industry, the likes of Netflix - but for most organizations this will not be the most relevant metric.

Having well-thought-of expectations for durations of these workflows is valuable. What are your peers doing? CircleCI here bases it on telemetry. Getting under this from a data-driven perspective in terms of real utilization of a system is extremely helpful.

I found it pretty admirable that the report set out to examine the data in terms of empathy for the developers and operations people doing the work. 2020 has been a tough year, but the lessons we've learned about looking after our people better will hopefully be maintained going forward.

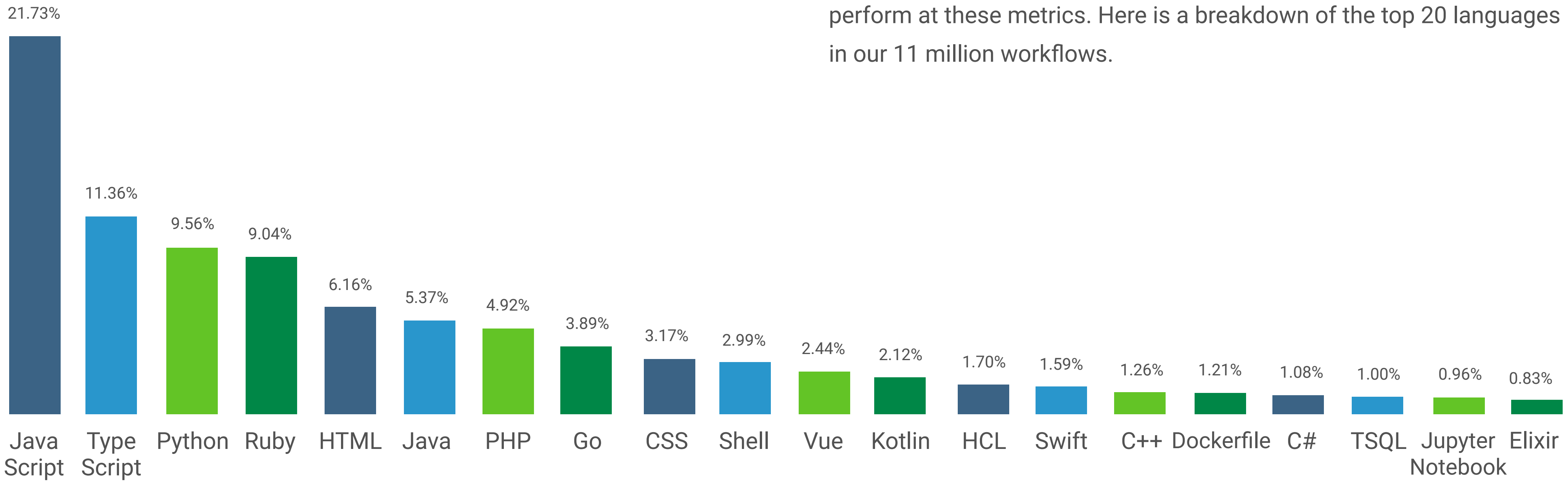
JAMES GOVERNOR

Analyst and co-founder, RedMonk

A fun aside on language choice

20 most-used languages on CircleCI

While not as important for determining goals for your development teams, it is always fascinating to see how workflows split by language perform at these metrics. Here is a breakdown of the top 20 languages in our 11 million workflows.



Language by Throughput

BY MEDIAN PERFORMANCE

1. Ruby
2. TypeScript
3. Go
4. Python
5. Kotlin
6. Elixir
7. Swift
8. HCL
9. JavaScript
10. TSQL
11. PHP
12. Java
13. C#
14. Jupyter Notebook
15. Shell
16. Vue
17. C++
18. HTML
19. CSS
20. Dockerfile

Here we see Ruby and TypeScript leading the way in terms of throughput, which suggests developers use smaller batches of work and use CI more often.

Language by Success Rate

BY MEDIAN PERFORMANCE

1. Vue
2. CSS
3. Shell
4. Dockerfile
5. TSQL
6. HTML
7. HCL
8. Go
9. TypeScript
10. JavaScript
11. Elixir
12. PHP
13. Jupyter Notebook
14. Python
15. Ruby
16. Java
17. Kotlin
18. C#
19. C++
20. Swift

High success rates for several languages near the top of this list may be an indicator of not many tests happening, as most of those top languages are not known for their testing robustness, but are likely steps producing artifacts and output in part of a larger project. From there, you see Go lead the way for success, followed by most dynamic languages. At the bottom end you see compiled languages, likely because there are build and tests steps for each of those languages -- with Go being somewhat of an anomaly in that breakdown.

Language by fastest TTR

BY MEDIAN PERFORMANCE

1. Go
2. JavaScript
3. Elicir
4. HCL
5. Shell
6. Python
7. TypeScript
8. CSS
9. C#
10. HTML
11. Vue
12. Jupyter Notebook
13. Kotlin
14. Java
15. Scala
16. Ruby
17. PHP
18. TSQL
19. Swift
20. C++

Go developers monitor their pipelines better than others. Again we see groupings of dynamic languages and then compiled languages. Recovery time could be slower here because duration is slower on some of these languages, or it could be that some languages are in more complicated workflows.

Language by shortest duration

BY MEDIAN PERFORMANCE

1. Shell
2. HCL
3. CSS
4. HTML
5. Gherkin
6. JavaScript
7. Vue
8. Go
9. Jupyter Notebook
10. Python
11. PHP
12. TypeScript
13. Java
14. Elixir
15. TSQL
16. Kotlin
17. Scala
18. Ruby
19. C++
20. Swift

Here we see languages with few build steps generally finishing first. Many shell scripts just run and exit and do not have robust testing. They may just upload an artifact. Again JavaScript and Go show up very favorably in terms of duration.

To improve delivery of outcomes, start with CI

Every company is now a tech company, and many of them are already doing continuous integration. Are they doing it well? Our data supports that you don't need to be an expert at CI to see a material increase in the metrics most important to your development teams. Average users of CircleCI would rank among the highest performing teams in the industry.

So what can you do to give your team a competitive edge? While we see average use of CircleCI enabling elite stats for performance, we also measured workflows that far exceeded these median values, and we have some ideas about how they get there.

- If you don't know how well your team is doing, it is impossible to set realistic targets for them. The ability to measure your engineering productivity to establish a baseline is absolutely necessary for staying competitive. If you are a current CircleCI user, you have access to the Insights dashboard, and measurements of each of the metrics in this report, for all of your workflows.
- CircleCI offers a wide range of machine types and class sizes. Selecting larger machines to run your workflow can reduce the time it takes for that workflow to run.

- Speed: CircleCI offers a fleet of convenience images for our users. These images have been optimized for CI so that they are more deterministic and faster to load.
- Intelligent test-splitting and parallelization options allow for robust test suites to run in significantly shorter times. Due to this, increasing your testing does not cause a proportional increase in workflow duration.
- Advanced caching options available on the platform significantly reduce the duration of a workflow when optimized. Many packages used to build your application can be cached and reused, saving you the time involved with downloading these packages on every run. Docker layer caching, a premium feature on CircleCI, can allow for an even greater reduction in workflow duration.
- Debugging failed builds is best when you have access to the machine where the workflow failed. CircleCI offers the ability to rerun a failed workflow and to use SSH to gain access to the machine that fails. Getting a signal fast is only one side of the CI feedback loop. The other side is the ability to quickly recover.
- CircleCI offers orbs, which are reusable packages of configuration. Abstracting layers of code from CI configuration files into open source, community-created and validated components allows for adding and replacing services without risk of failure. Using well-tested configuration components reduces the sources of errors, and using orbs to integrate testing suites into your CI pipeline means getting more information from your runs. We observed that recovery time for workflows decreases with increased orb usage (from 0 to 1 orb and from 1 orb to many).
- Premium support includes the option for configuration review by DevOps experts at CircleCI. These reviews find optimization opportunities that can greatly reduce workflow Duration and other configuration bottlenecks.

> Continuous integration isn't new, but it's not as widely adopted outside technology-centric organizations as you might expect. Without a modern CI environment, teams can struggle to adopt other modern software processes and tools that often fit neatly into the pipeline. Security is one such example, where increasingly shifting security testing into CI pipelines helps developers ship secure software faster. That's why a metrics-driven approach can be so useful, it helps build the business case for adopting, or expanding, CI inside your organization.

GARETH RUSHGROVE

Director of Product Management, Snyk

Methodology

The data in this report was collected for 30 day periods starting on the first day of August 2019, March 2020, April 2020, May 2020, and August 2020. It has been filtered to reflect only those workflows from projects that use GitHub as their VCS. It represents over 55 million data points, more than 44,000 organizations, and over 160,000 projects.

REPORT AUTHORS

- Ron Powell (@whyD0My3y3sHurt)
- Michael Stahnke (@stahnma)

REPORT EDITORS

- Emma Webb
- Gillian Jakob Kieser

REPORT CONTRIBUTORS

- James Governor, RedMonk
- Josh Koenig, Pantheon
- Gareth Rushgrove, Snyk
- Bryan Lee, DataDog
- Jen Riggins, The New Stack

REPORT DESIGNER

- Alex Moran

ACKNOWLEDGEMENTS

- Dawit Gebregziabher
- Melissa Santos