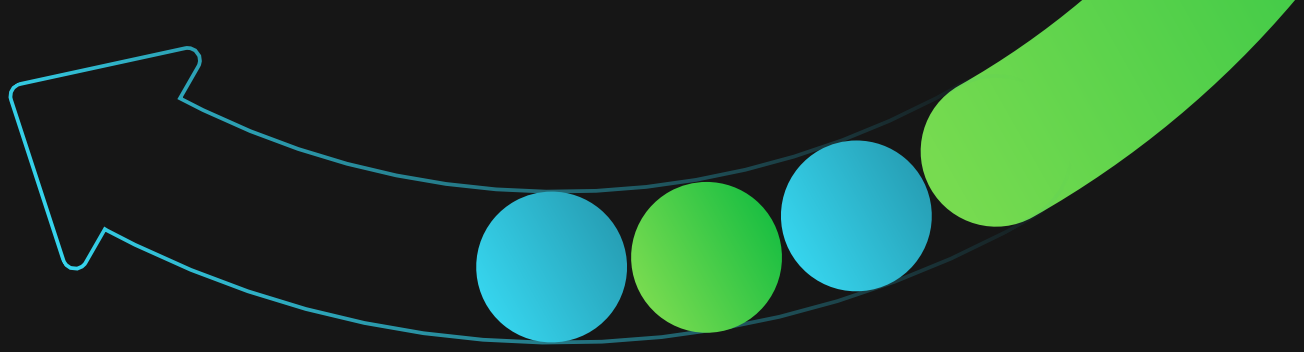
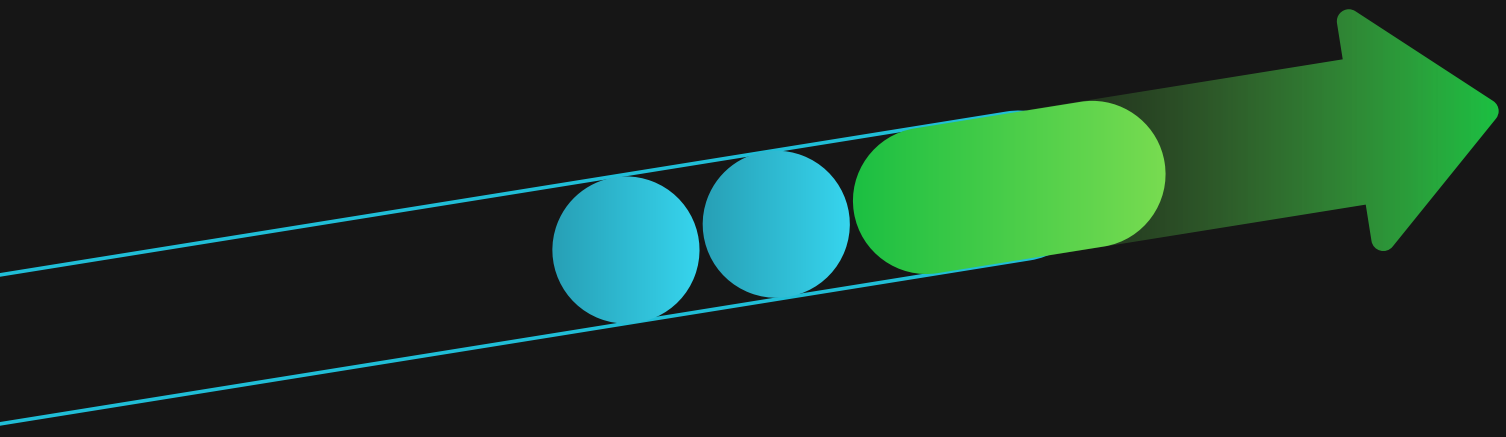


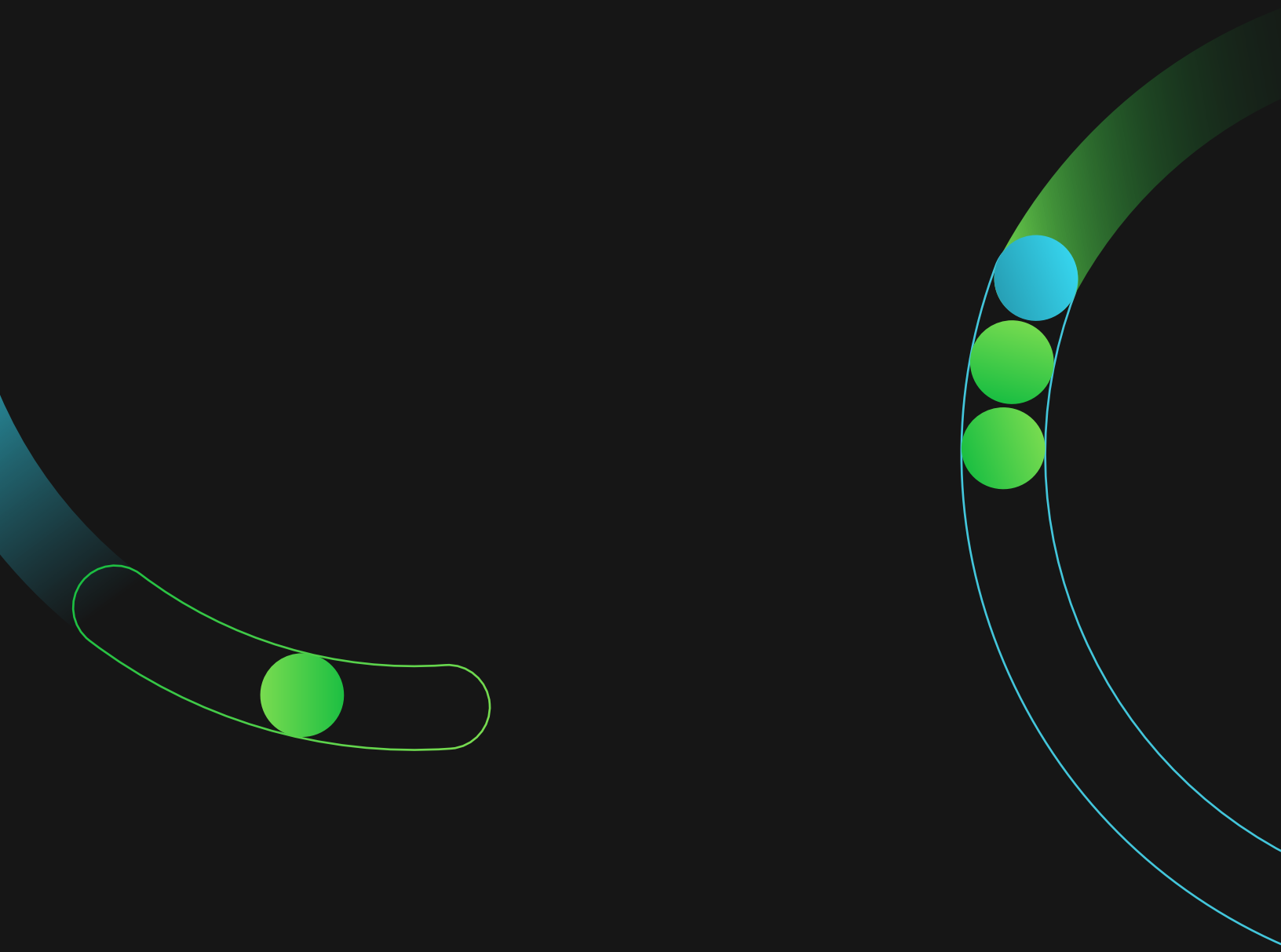
Three Critical Development Metrics for Engineering Velocity

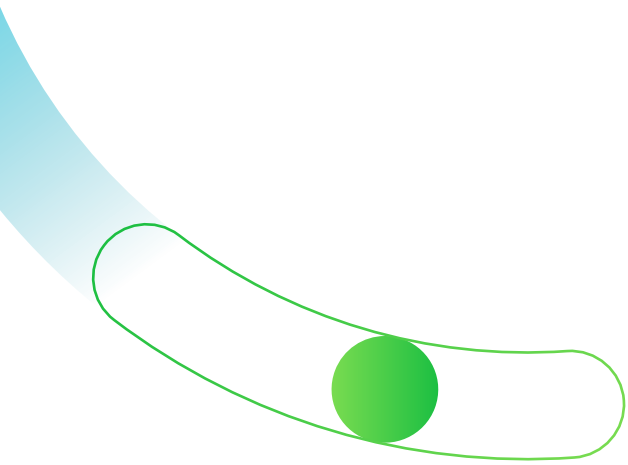


circleci

Table of Contents

Introduction





If you're an engineering leader or your business depends on software (hint: it does), you're probably spending a lot of time thinking about DevOps.

This is a fusion of two traditional divisions in the software world: development and operations. By tearing down the wall between these teams, companies can quickly create meaningful products without sacrificing stability or performance. But DevOps isn't a team; it's a cultural movement, one that is rapidly becoming the gold-standard for delivering code.

Companies who embrace DevOps practices are more successful, regardless of industry. Most engineering leaders are sold on the idea of DevOps but, when they try to institute these practices, they stumble.

There are many reasons why adopting DevOps is so challenging: organizational friction, change management, and cultural inertia are some of the obvious offenders. Ultimately, though, you can't manage what you can't measure; without knowing what to track or how to define progress, organizations can't be confident that their digital transformation is moving in the right direction.

Purpose

As a CI/CD platform, CircleCI occupies a critical role in the world of software development: we know how often a mainline branch is in a deployable state, how long it takes to deploy a change, and how often companies are deploying. And we think these metrics—stability, commit-to-deploy time (CDT), and deploy frequency–are the best predictors of an organization's velocity and growth.

There were two goals of this report: **1) investigate these metrics' correlations with business growth, and 2) uncover concrete practices organizations can implement to improve these metrics.** CircleCI's position in the software development lifecycle grants us access to an impressive dataset: in just one hour, our users build around 5,400 branches—that's about 3,900 hours of cumulative build time. At time of writing, we process over 6.2 million builds per month.[1]

We used this dataset to identify top performers for each metric, then asked teams how they write and deploy software. Our aim in highlighting these best practices is to provide a tangible path for introducing DevOps principles to your own organization.

---

[1] For those interested in the infrastructure behind these numbers, check out this deep dive on StackShare.

Methodology

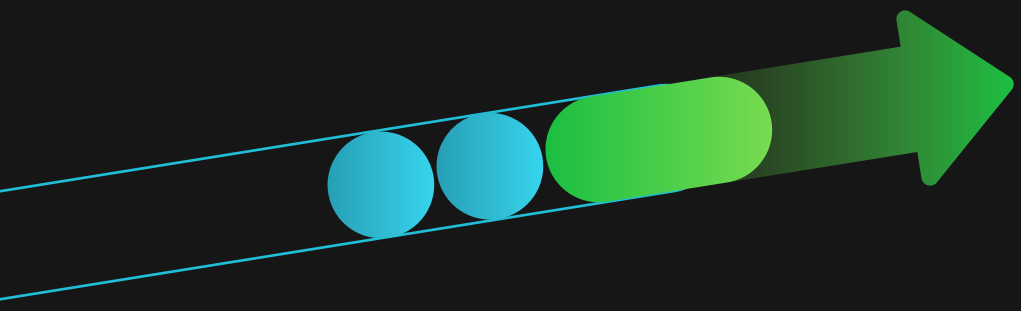
For this report, we examined a sample of GitHub and Bitbucket organizations[2] built on CircleCI's cloud platform from June 1 to July 31 of 2017. To mitigate the effects of inactive projects, organizations must have had a) tested and deployed a project and b) merged to master at least once on at least one project during the time range. Future iterations of this study will be less restrictive, likely using clustering or classification.

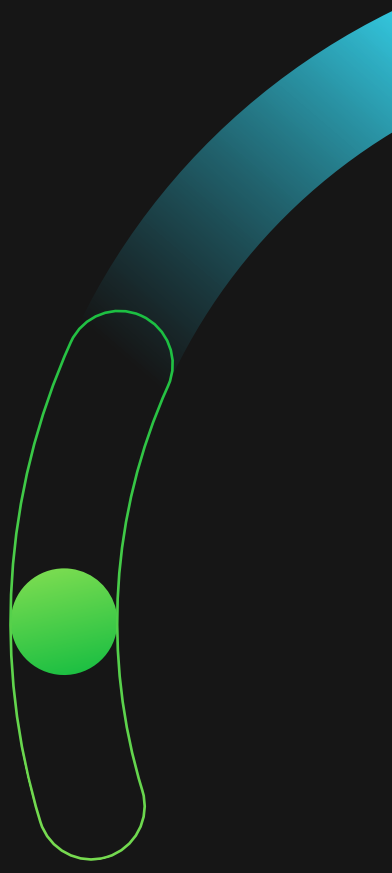
For the growth metric, we selected Global Alexa Internet Ranking and used enrichment services from Clearbit. Our entity match rate between organizations and Clearbit was 82.02%. Future iterations will use other sources to find matchable domains.

---

[2] CircleCI takes security and privacy seriously. All named customers have given their permission to publish this report. Please see our privacy policy for more information.

Three Metrics





STABILITY OF MAINLINE BRANCH

> "If you had to deploy right now, could you?"

The mainline branch is your application's source of truth. It's the master mold for every feature branch created by your developers. If the mainline branch is broken, your team is paralyzed: they can't start building new features, and their ability to address major incidents is hampered.

In other words, mainline branch stability is a measurement of **deploy readiness**. It answers the question, "If you had to deploy *right now*, could you?" If the answer is no, it won't matter how long a deploy takes because a deploy will be impossible. Consequently, mainline branch stability also directly affects deploy frequency.

In this study, stability was measured as the percentage of wall-clock time a project's default branch spent in a failed state. Wall-clock time represent real-world passage of time, as opposed to total time consumed by all of a project's containers—virtual machines in which software is tested. The default branch is the branch a project has chosen as the project's "master" branch.

DEPLOY TIME

> The lower the deploy time, the less expensive it is to change your product.

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. And this cost is incurred every time an organization's codebase changes, whether that's for a new feature or bugfix.

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.

In this study, deploy time was measured as the number of wall-clock minutes between queueing a build and completing the build.

DEPLOY FREQUENCY

> A vital sign—the heartbeat of an organization. With each pulse, an org is delivering value, discovering customer needs, and fixing problems.

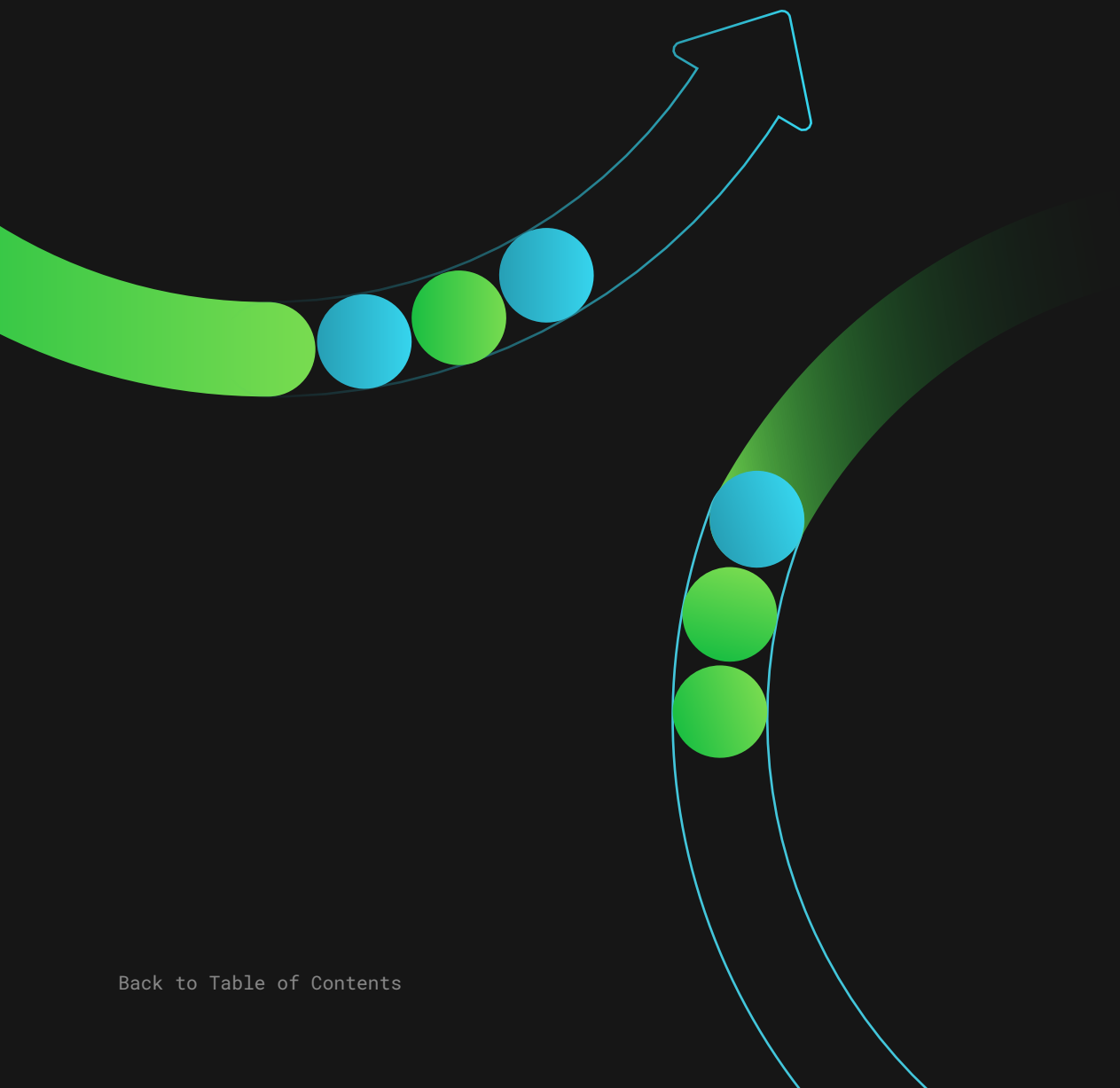
While the above metrics are benchmarks for how often an org could be moving, it is deploy frequency that indicates a company's actual speed. Deploy frequency is also a function of the previous two metrics: high stability and low deploy time both encourage higher deploy frequency.

Deploy frequency is a vital sign—the heartbeat of an organization. With each pulse, an organization is delivering value, discovering customer needs, and fixing problems. Frequency is also directly affected by the previous metric: when the cost of a deploy goes down, engineers are encouraged to deploy more often.

"We see that deployment frequency is a key indicator of high performing organizations," says Dr. Nicole Forsgren, author of Accelerate and CEO/Chief Scientist of DevOps Research and Assessment (DORA). "High performers can deploy on demand, while their low-performing peers can only deploy their code once per month or longer. This difference in deployment speed makes all the difference in delivering value, delighting your customers, and keeping up with compliance and regulatory changes."

In this study, deploy frequency was measured as the median number of "default-branch" builds run on our cloud platform with a valid deploy step, per week per organization.

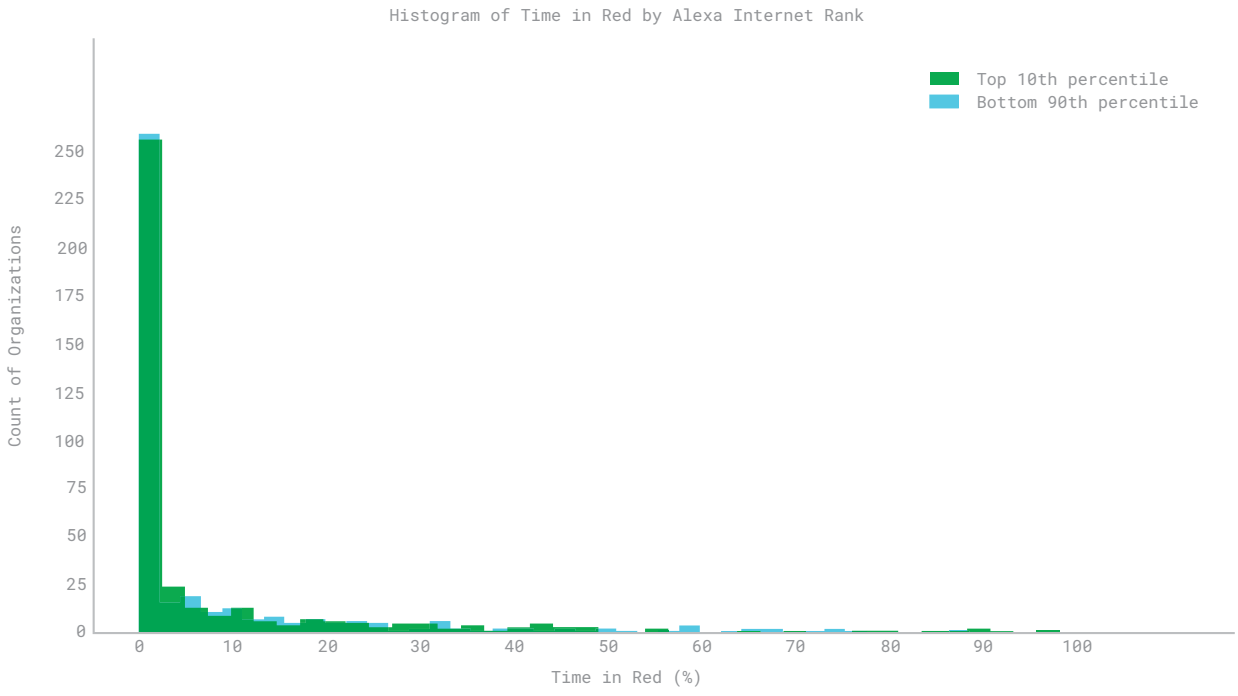
Findings, Analysis & Best Practices



Mainline Branch Stability

Generally, stability is a key metric for companies, with 80% of organizations keeping their mainline branch deployable over 90% of the time. The data are exponentially distributed, with organizations in the 95th percentile maintaining stability 99.9% of the time. Median stability is at 98.7% stability, while the mainline branches of the lower 5th percentile spend 47% of their time in a failed state.

For the top 10th percentile of Alexa Internet Ranked orgs, a similar distribution exists, with the bottom 5th percentile spending 46% of their time in a failed state. Median stability is 98.5%, and the top percentile is 99.9%. Again, 80% of all organizations keep their master branch stable 90% of the time.



Most organizations, whether they're in the Fortune 100 or a three-person start-up, seem to understand the value of high mainline branch stability. But reaching and maintaining that stability requires rethinking team structure and changing traditional process.

Best Practices

Robust Suite of Tests

If the mainline branch fails, production grinds to a halt. The best companies know this and invest in robust suites of automated tests. By testing each change, engineers can ensure their code functions as expected. These test suites act as a kind of preventative healthcare for software; it's far easier to catch bugs by performing routine checkups—and the more routine, the better.

Ben Sheldon, Senior Software Engineer at Code for America, believes good tests reduce cognitive load, saying, "In most of our code reviews, we have high confidence that, if the tests pass, the risk of actually breaking something is low. So, our code review process is less around preventing problems and more around understanding changes that are happening. Ultimately, we want engineers to have the opportunity to talk at a high level about architecture, or where the codebase is going in the long term."

Having tests not only keeps bugs out of the mainline branch, they also allow engineers to spend precious mental energy on more important decisions. Instead of wasting time debating small decisions, they can discuss high-level system choices. This leads to higher overall code quality and, in turn, more stability.

Feature Flags

Even the most sophisticated test suites can't capture the complexity of the real application. Instead of depending on tests to catch every potential issue, CircleCI ships features behind feature flags with LaunchDarkly for safer releases.

Tim Wong of LaunchDarkly, says that "feature flags decouple the release of functionality from the deployment. In a traditional shop, both of these things are the same: The code ships when the deploy is complete. With feature flags, you get to decide when to deploy the code, and when to release your code. You test in the real world, then you can decide whether you're prepared to give that to your users. That's a very stark difference."

A bonus effect of feature flags is that they can be used defensively during incidents. If there's a problem in production, for example, developers can simply turn off the offending feature, effectively quarantining the problematic code until a fix is produced. This is usually faster than rolling back an entire change, thus increasing overall stability.

Efficient Recovery from Failure

Even with the best test suites, bugs will inevitably find their way into production. When that happens, it's important to have a fast, reliable recovery process in place.

Stripe's publication *Increment* conducted a study of incident responses across leading companies (including Amazon, Facebook, and Google) and found that best practices were generally aligned. Most had well-defined processes for dealing with production outages, from paging on-call engineers to runbooks for consistently triaging issues. Defining and practicing these routines is essential for maintaining the mainline branch's health.

One finding relevant to our study is that leading companies mitigate before they resolve. This often means rolling back a change instead of fixing the root cause. Debugging an error is time-consuming and results in more time in the red, whereas reverting a change can quickly staunch the flow.

Communicating the Cost of Failure

Bugs aren't just errors in the codebase; they have real consequences, for both you and your users. Identifying and understanding the impact of these consequences helps prioritize work and increases motivation to resolve technical issues.

At GetCalFresh, a division of Code For America, engineers block out an hour each day to observe how people fill out food stamp applications. John O'Duinn, former "Infrastructure Guy" at Code For America, says this is intentional because "5 o'clock is a cutoff for government paperwork. It's not just an arbitrary time; it's heads down and very focused. Everyone's thinking, 'Get people food,' and this puts a constant pressure in that hour."

If anything breaks in that hour, engineers witness it firsthand. This makes abstract "errors" more concrete and increases motivation for fixing those problems. Not only do engineers see the impact of their work, they're also able to see how problems affect lives. A little bit of paranoia can be healthy, although teams must be vigilant of this slipping into a repressive culture of fear.

Communicating Potential Conflicts

While tests are the most obvious way to prevent failure, there are other, more subtle variables involved. The most insidious of these is a lack of awareness and communication around potential 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.

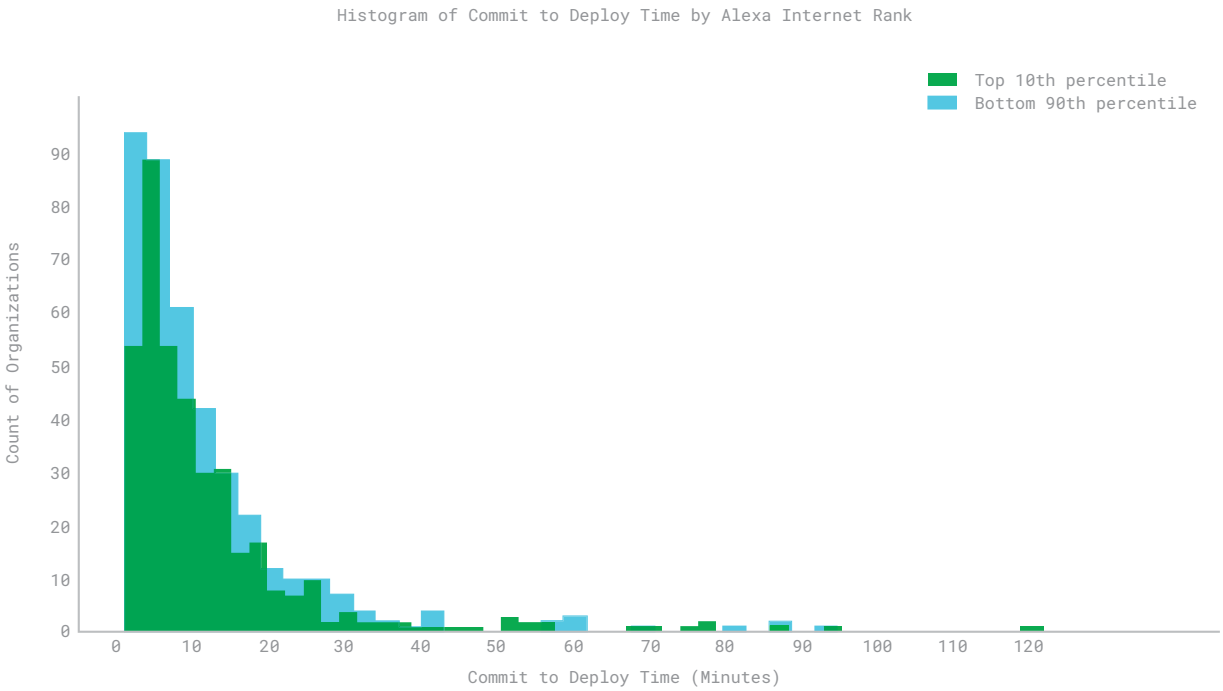
At Code.org, engineers are organized into functional, vertical "cabals" of three to seven engineers. Jeremy Stone, Head of Engineering at Code.org, says they "broke Code.org into missions and grouped people who worked on similar projects. At small sizes, it was easy for teams to meet every day without straining people's attention spans."

Each cabal participates in a daily standup, where they share unplanned work and highlight potential conflicts. These daily standups only work because of a Goldilocks effect: there are just enough engineers to be productive, but not so many that people are losing context into what others are doing. And this context is crucial in preventing repeated or conflicting work, both of which increase the odds of the mainline branch entering a bad state.

Deploy Time

Deploy time is largely kept under control, with 80.2% of organizations deploying 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.



The fact that we pulled these data from organizations using CircleCI gives us some insight into these low CDT times. While we're clearly biased, we're pleased to see that developers who use a dedicated CI/CD platform spend less time waiting for their code to deploy.

Freedom from Manual QA

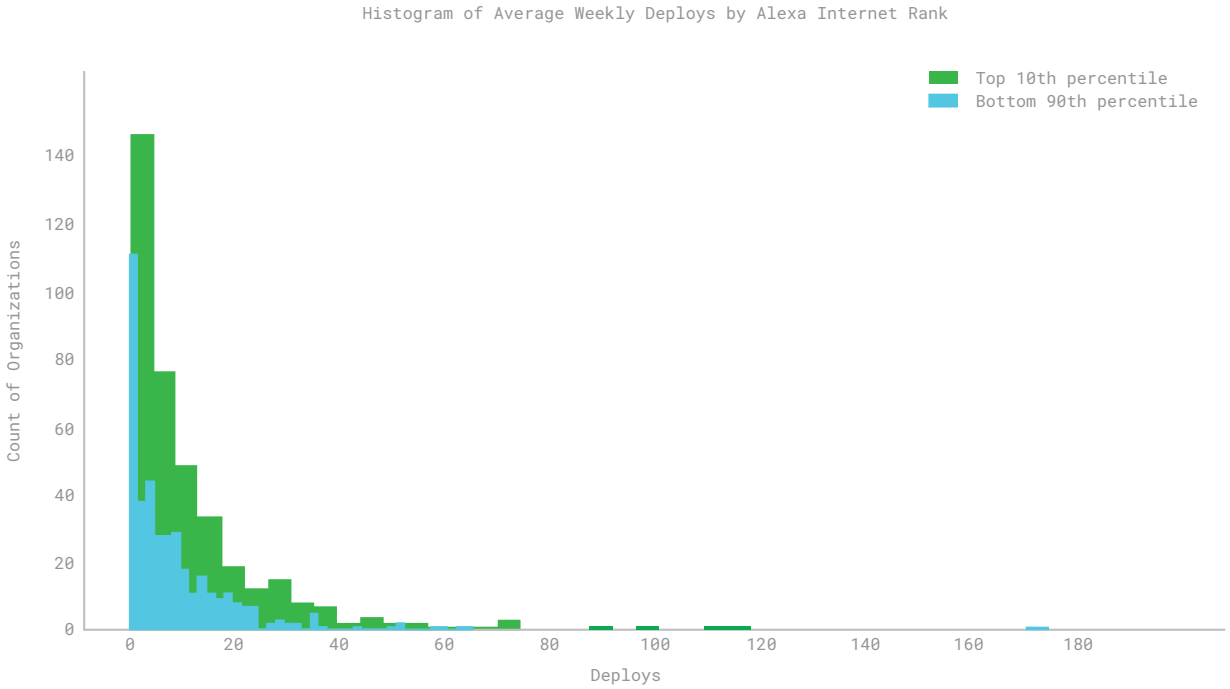
One byproduct of having a robust test suite is a reduction of time spent performing manual QA. Organizations can only do this when processes are highly optimized: few bugs, efficient recovery from failure, and constant monitoring.

Having these pieces in place allows orgs to treat *customers* as the QA team—at least where it's safe to do so. This doesn't mean that QA *isn't* important, only that there's a definite advantage to automating expensive manual work where possible.

Deploy Frequency

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.

For organizations in the top 10th percentile of Alexa Internet Ranked orgs, we saw an increase at the leading edge and even greater spread, with the 95th percentile deploying 42 times per week — 40 times the bottom 5th percentile, but still 5 times the median (8 deploys/week). A slight difference in range is also reflected, with 75% of organizations deploying less than 16 times a week.



Of these three metrics, we see top Alexa Internet Ranked orgs moving the fastest. There is a slightly higher distribution between high-scoring organizations and the rest, with the top percentile of Alexa Internet Ranked orgs pushing code more at the high end. This is a reflection of the importance of velocity: to be the best in any category, organizations must maximize their deploy rates.

Best Practices

Small, Short-Lived Pull Requests

Deploy frequency is a composite metric, dependent on both mainline branch stability and CDT. One well-known but powerful tactic is to limit the size and lifespan of pull requests.

“Generally,” says Jeremy of Code.org, “we have a culture against long-lived branches, to the extent that around 80% of them get merged within 24 hours.”

Brad Buchanan, Lead Software Engineer at Code.org adds further details: “We’ll see pull requests as small as one or two lines of code, and a bulk of them fall under 200 lines.”

While this might seem impossible, it’s a testament to Code.org’s commitment to splitting work into small, shippable units. By doing this, they’ve created a culture of incremental development, allowing them to get code out of review and into users’ hands more quickly.

Feature Flags

Another way to shorten PRs and increase deploy frequency is to use feature flags. Tim of Launch Darkly says that "large PRs are large because of dependencies. By shipping dependent pieces independently of each other, you reduce the amount of code in PR rot."

Breaking work into digestible chunks is at the core of continuous software development. Feature flags empower developers to ship work they're doing as they complete it. Developers are happy because they're finishing tasks, and users are happy because their product still works.

Metric Interactions

While these metrics have been presented discretely, they are not isolated variables; optimizing for one will affect the others. For example, organizations that spend less than 5% of their time in the red have a median of 5.3 deploys per week and 6.7 minutes of CDT. By contrast, everyone else has a median of 8.7 deploys per week and 11.7 minutes of CDT.

Moving quickly without a proper test suite in place will result in lower stability and higher deploy frequency. It's possible that organizations focusing only on velocity have to spend more time fixing their mistakes. This hypothesis might explain why, despite higher CDT, these companies still deploy more—not because they planned to, but because they have to.

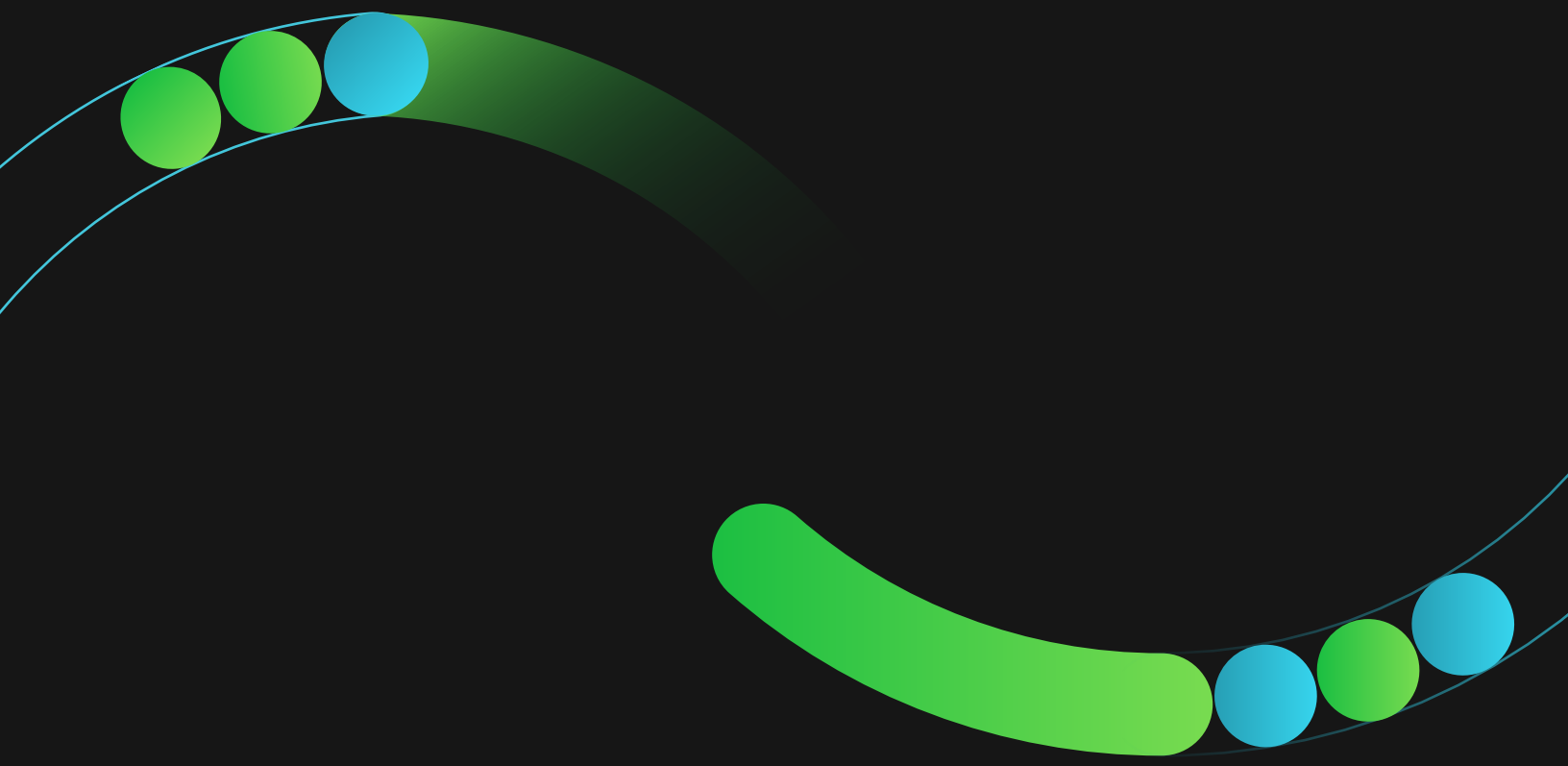
Organizations that only prioritize deploy frequency increase the likelihood of instability as cruft and technical debt increase. Shipping buggy code also

increases the need for more deploys to fix problems. With this in mind, deploy frequency can't be safely treated as the definitive measure of an organizations' velocity.

Organizations building in 12 minutes or less have a median of 5.3 deploys per week and an instability of 0.2%. Organizations over 12 minutes of build have a median of 8.3 deploys per week and instability of 2.7%. This reinforces the theory so far: **orgs pushing themselves to quickly meet customer needs may find themselves in bad states or waiting longer for projects to build.**

Finding a balance is key.

Demographics

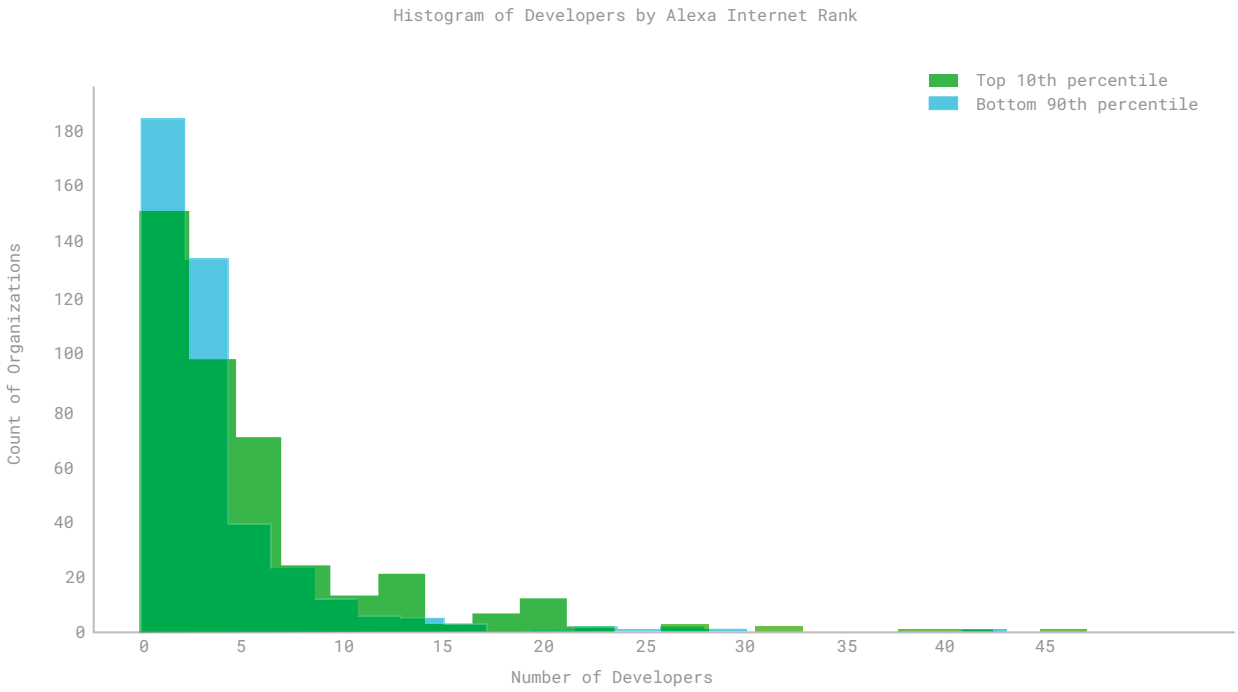


In addition to our three primary metrics, we also examined several demographic variables, including geographic distribution, number of people running CircleCI builds, and programming languages.

Geographic Distribution

Of the sampled organizations, 79% had all their developers in one country. For 56% of these mono-national dev teams, that country was Japan, the United Kingdom, or the United States. Of those in the top 20th percentile of Alexa Internet Ranking organizations, 74.2% were in one country.

This implies that leveraging CI for growth is independent of your team's geographic distribution. There is no significant difference between teams consolidated in one location and well-organized distributed teams. Whether your team is in one room or spread across time zones, you should feel confident that team distribution has little to do with its success.



Number of Builders

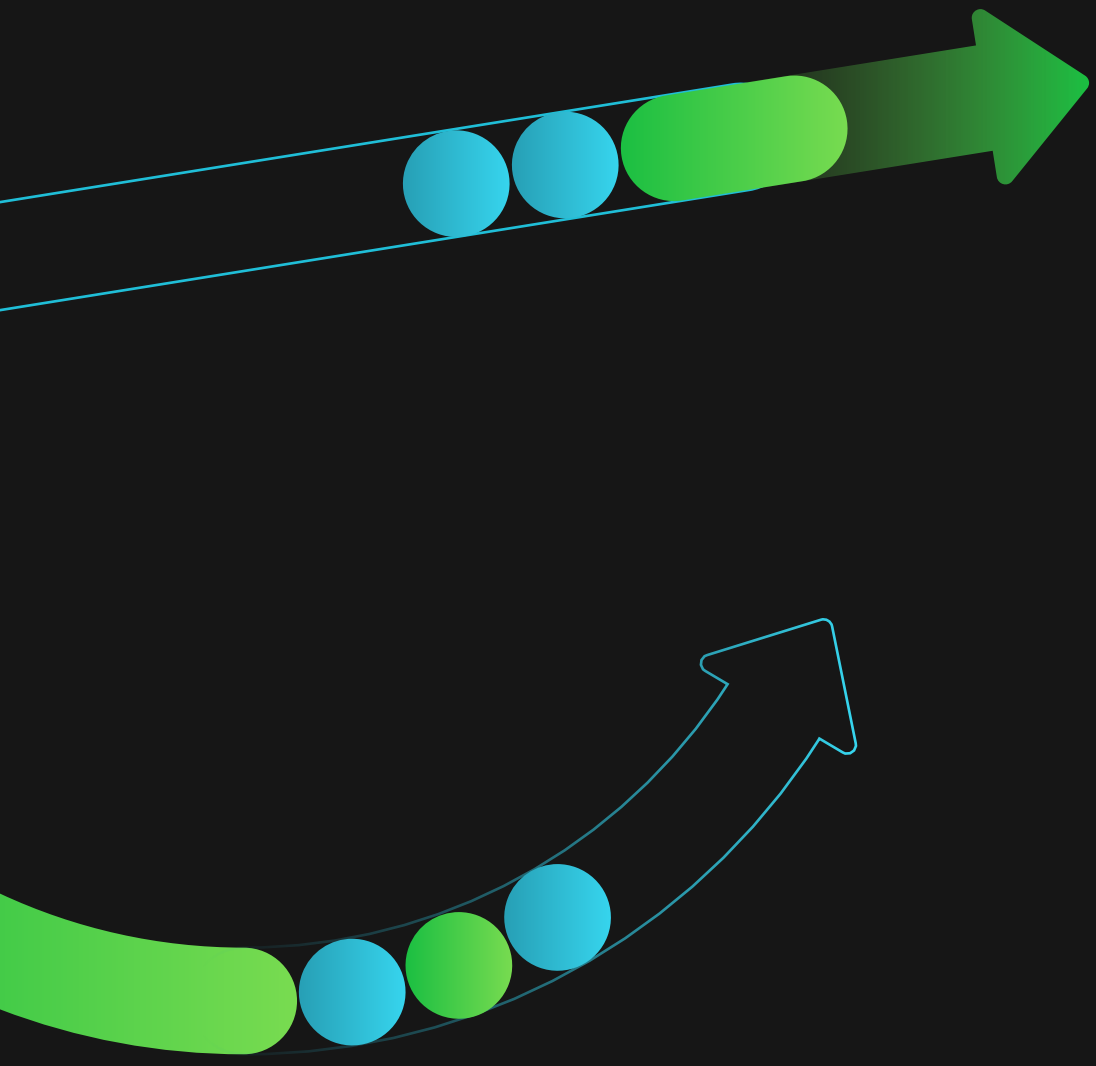
95.4% of sampled organizations had an average of ten or fewer developers building every week. That same average was seen by 85.3% of organizations in the top 10th percentile of AIR.

We think top performers are more likely to split projects into small teams. As we saw above with Code.org, small teams tend to be more effective at breaking work into units and keeping communication loops tight.

Dominant Programming Languages

For projects with a “dominant language”, 48.2% were written primarily in JavaScript or Ruby. However, these two languages only accounted for 49.3% of projects in the top 10th percentile of AIR organizations. These data imply that engineers’ language choice have a negligible effect on their organization’s success.

Conclusion



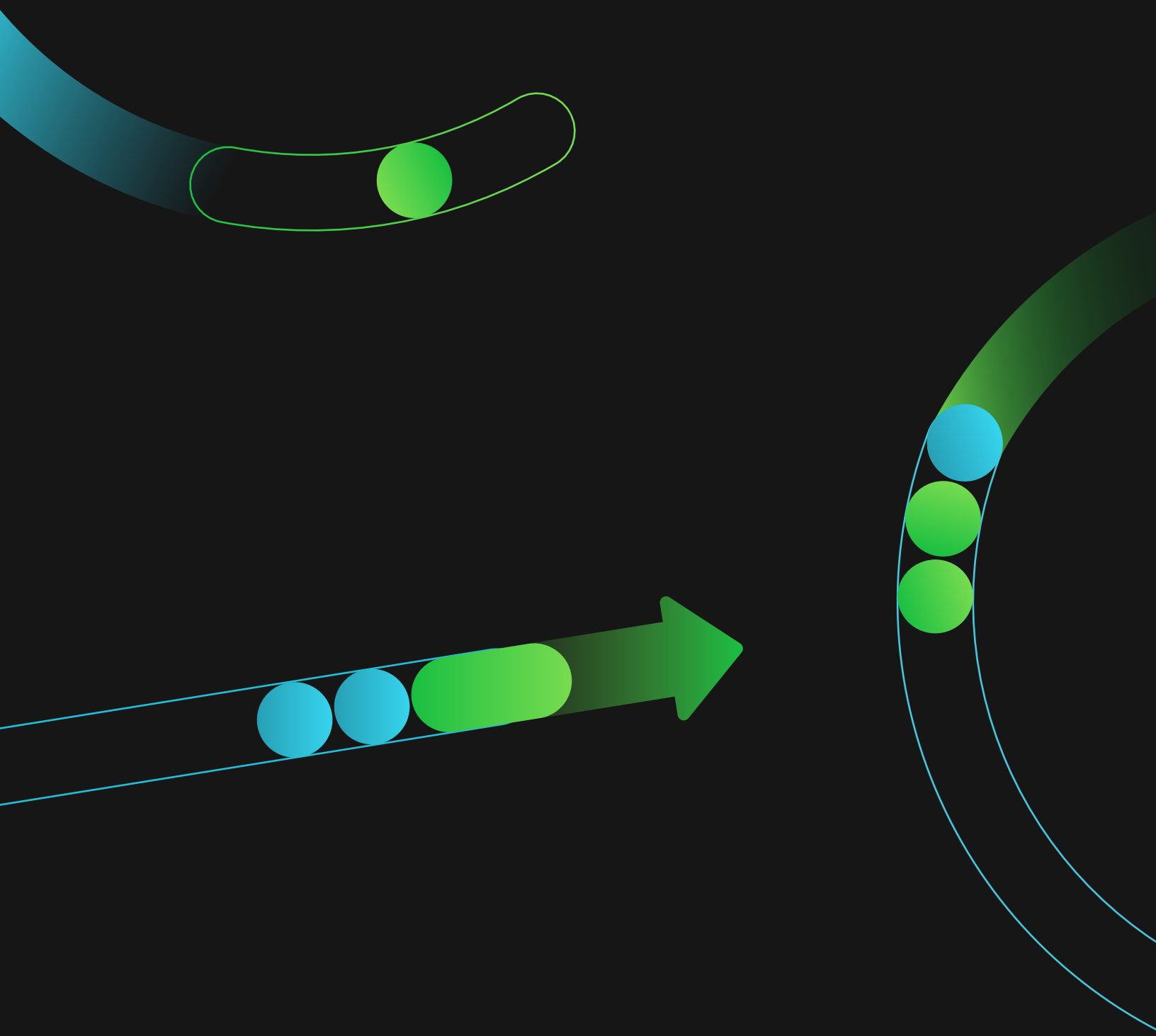
To survive in an accelerating world, organizations must be constantly adapting. Embracing DevOps culture requires altering your organization to *enable* change; a company's success depends on its ability to change itself for a world in constant flux.

But it's a mistake to believe that a "digital transformation" can happen instantaneously. Instead, the best companies treat their software development pipelines as strategic investments or better, company values. They're continuously integrating the best practices of continuous integration.

The rewards they reap are many: increased throughput, happier engineers, and better insight into customer needs. And these are just a few of the advantages an organization gains by adopting a new, more nimble development philosophy. Organizations optimizing for the three metrics we've identified have set themselves up for success. By following their best practices, other organizations can pull the idea of DevOps from philosophy to practice.

This is the first time we've conducted a study at this scale. We think there's something interesting here—and we're not entirely sure we've gotten it right—but consider this the initial foray into hitherto unknown territory. As with all our work, we'll iterate: by shipping our initial thoughts, we'll start and maintain a conversation with our users and colleagues. When we learn, we'll continuously integrate those lessons into future studies, inching closer to an ever more accurate impression of the world of software development.

We hope you'll join us.



circleci

Start Building Now

www.circleci.com