What's CI/CD?

Lots of small mistakes instead of one big one

Justin
Aug 26, 2020

The TL;DR

Continuous Integration (CI) and Continuous Delivery (CD) are philosophies and toolsets for shipping code: instead of big, infrequent releases, developers push and test code *constantly* in smaller batches.

- Historically, it was expensive and risky to ship software, so **updates were infrequent** and bundled together
- Cheaper servers and better tooling have changed the equation: now developers can **ship frequently and test often**
- **Continuous Integration** means integrating and testing your code changes automatically, while **Continuous Delivery** means you deploy those changes often
- **Tools like CircleCI** let developers write automated tests, monitor builds, and locate problems fast so they can iterate and fix quickly

CI/CD is quickly becoming the de-facto standard for how startups ship code, and is even (slowly) making its way into larger companies.

Big scary code changes

Not to sound like an old man screaming into the clouds, but software used to be a *lot* harder to write, for *a lot* of reasons – one of those reasons was the deployment process. There's a big difference between getting software you've written to run on your laptop, and putting it on a server ready to support thousands (hopefully more) of users.

> 🧠 **Jog your memory** 🧠
>
> Applications in the cloud are just a bunch of code running on big servers. Developers build software locally (on their laptops) and then push that code

> to servers when it's ready to go.
>
> 🧠 **Jog your memory** 🧠

A lot of the tools that developers take for granted today – think Github for managing version control, public cloud providers like AWS, high speed internet, and monitoring tools like Datadog – are actually pretty new. So just 10 or 15 years ago, if you were writing software, you'd need to:

- Manage your remote repositories manually (no Github)
- Run your own data centers, or rent someone else's (no AWS)
- Move code between locations slowly (no 1gbps internet)
- Manually run tests and ping servers for monitoring (no Datadog)

In other words, deploying code used to be a real pain in the ass (it still is, but it used to be too). So it's not surprising that developers would do it extremely infrequently: new features and bug fixes would get **batched together** and shipped as one monolithic unit (think: monthly timeframes or longer). This kind of *sporadic integration* and *sporadic delivery* helped avoid the annoying overhead of deploys, but came with its own baggage:

- Infrequent releases mean products improve very, very slowly
- Batching multiple features and fixes together made it difficult to isolate problems
- Rolling back changes to fix a single problem meant eliminating an *entire* release

Over the past decade though, new tools and cheaper infrastructure have helped alleviate some of these core issues, and that in turn has enabled developers to integrate and ship *a lot more often*. That's where CI/CD comes in; we'll tackle each in a separate section, but the philosophy behind them is the same.

CI: integrate often

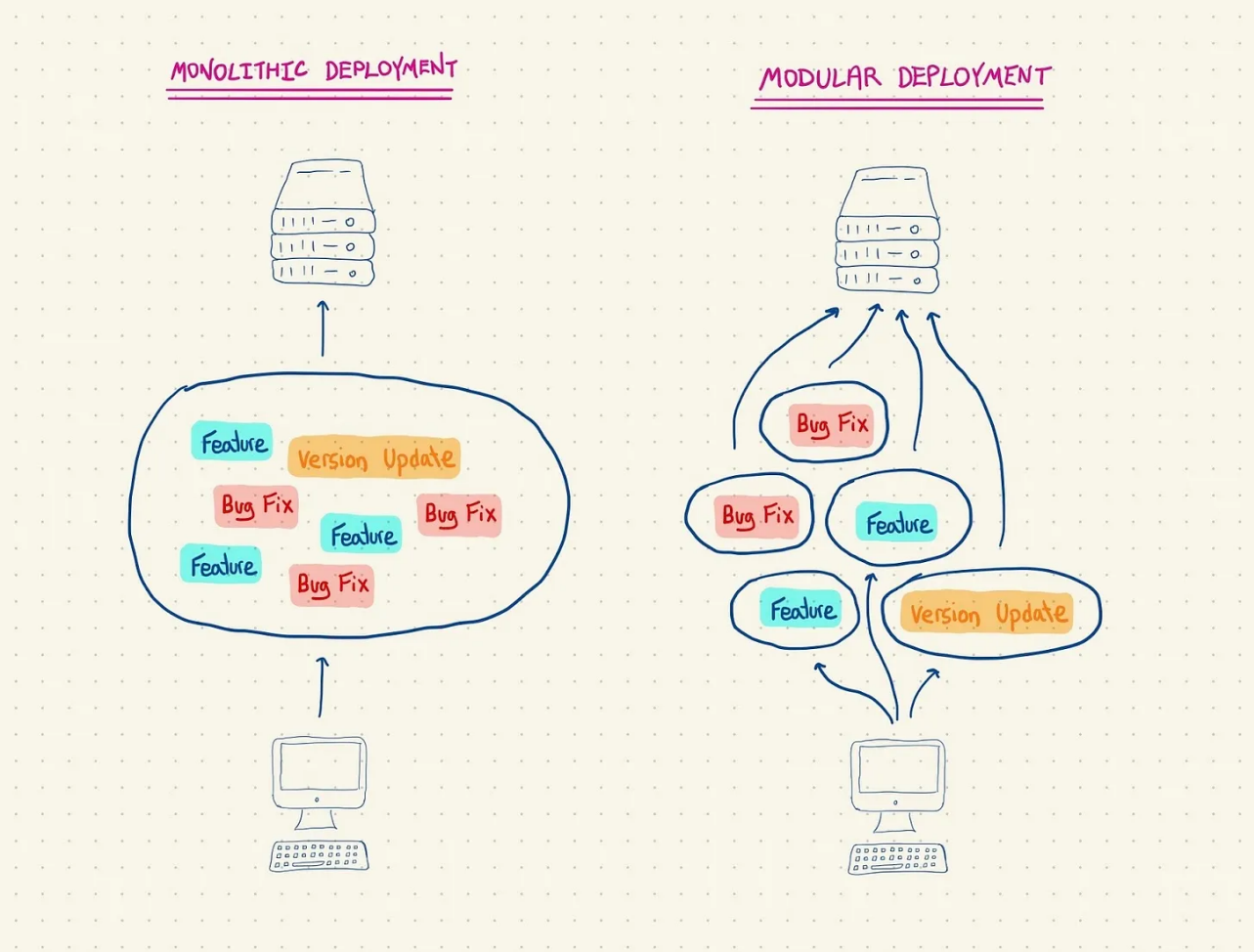
People often talk about CI as a tool or a framework, but it's not: it's a **philosophy** for how code should be written and shipped. Instead of developers working on self contained big features, they push code in modular increments that are

unlikely to break things and help isolate where (if!) things go wrong. This is what the *integration* in CI means: integrating your new code with the existing codebase.

To help illustrate what this means exactly, let's imagine we're Twitter, and our engineering team is shipping a new feature that allows users to write ephemeral tweets (called Fleets) that disappear after 24 hours, AKA Twitter Stories. If we're the product manager for Fleets, we'd probably break the feature down into a couple of requirements:

- The ability for tweets to self delete
- A new UI for tweeting a Fleet
- Blocking Fleets from getting likes, retweets, etc.

Traditional engineering says: build all of this together, integrate it with the existing codebase once, and then push it to production once. CI, on the other hand, prescribes *the opposite*: build each bullet point as a modular feature, integrate it with the codebase and test it independently, and then deploy it independently. That way if something goes wrong, it's much easier to isolate and fix.

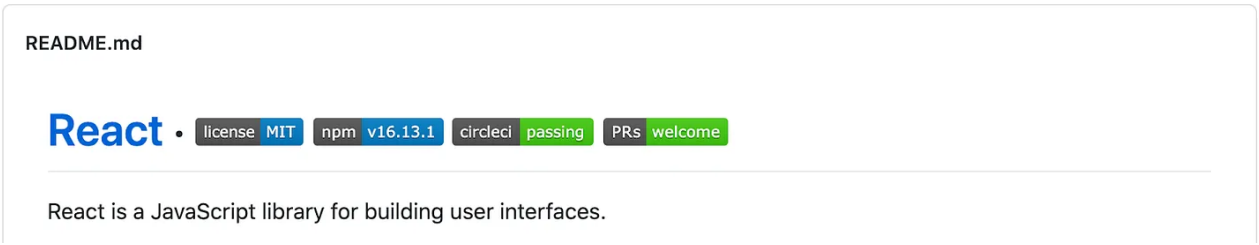


Another critical part of the CI philosophy is **automated testing**. Running a tight software operation means constant testing: does my application build properly? Does my code compile? Do all of my classes and functions do what I expect them to do? Mature software tends to be covered in **unit tests** that test individual software units and make sure they're working properly. Have all of those passed? CI prescribes that these tests should be automatic: whenever you integrate your new code into the existing codebase, these tests should run, and you should get notified of any failures.

> **🔍 Deeper Look 🔍**
>
> **Unit tests** are a really important part of stable software, because they help (a) make sure your stuff works, and (b) are localized enough to help you figure out why things aren't working. An example of a unit test in our Twitter Fleets example: a Twitter dev might write a test that checks a Fleet after 24 hours to make sure it got deleted.

> **🔍 Deeper Look 🔍**

CI is pretty much everywhere you look in the wild: tons of popular open source software like React and Tensorflow show you the latest status of their builds. For example, here's React's codebase showing a "passing" status from their CI tool (CircleCI).



Because CI is more of a philosophy than a tool, you don't *need* to use specialized software to do it (see this half-parody post about a rubber chicken), but there are a bunch of ready-made solutions that set up a CI server and make it easy to run tests and integrate with other software. A few popular ones: CircleCI, TravisCI, and Jenkins.

CD: deploy often

CI tells you how you should *integrate* your new code with the existing codebase, and CD tells you how you should *deploy that code to your users*. The philosophy is ultimately the same: small, incremental, frequent deployments are better than large, batched, infrequent ones. Once you've integrated your code and it passes tests, you should deploy (read: push to the server) as soon as possible. For most startups today this tends to mean *daily* deploys at the minimum, and often *multiple per day* (for bug fixes in particular).

To actually get this working, teams set up **staging servers** – they're environments that mimic the actual app as closely as possible, so you can test reliably before sending the new build out to users. Companies will typically have domains for their staging servers that you can access: if we're shipping a new design on the marketing site, we'd deploy it to staging first so we can see how it looks. If all seems solid, it goes to production.

> 🚨 **Confusion Alert** 🚨
>
> A lot of the complexity in deploying software lies in the fact that you develop software locally (on your computer), and then push it to a server; that server is *very different* than your laptop, and a million things can go wrong.

> Deploying to staging first helps solve most of those problems without risking the actual app breaking – as long as your staging environment is very, very similar to production.
>
> 🚨 **Confusion Alert** 🚨

There aren't really any specialized tools for CD, as it's more of an operational workflow. But that gets us into the important point here: CI and CD can't really be separated (which is why everyone says CI/CD together). They're two sides of the same coin – modularizing software deployment – and just refer to two pieces of a process. If you look at the websites of any of the solutions for CI (e.g. CircleCI) you'll see that they talk about CD as part of the product too; the usual terminology here is **CI/CD pipeline**.

Terms and concepts covered

```
Continuous integration
```

```
Continuous delivery
```

```
Automated testing
```

```
Unit tests
```

```
Staging server
```

Further reading

- The Wikipedia article on Software Deployment is one of the craziest things I've seen this week

- There are broader debates about how software should be developed: CI/CD tends to align with the Agile methodology
- You might not know this, but there's a big Github competitor called Gitlab, and one of its differentiating features is built-in CI

3 Comments

Write a comment...

Investing_panda Sep 12, 2020 Liked by Justin

What are your thoughts on Jfrog's products? Do their products cover the entire CI/CD area? Thanks!

♡ 1 Reply Gift a subscription Collapse •••

1 reply by Justin

Apoorv Agrawal Oct 9, 2020

Investing_panda: JFrog's main product is an artifact manager not a CICD tool. Though they have since then launched a CICD product called pipelines

♡ Reply Collapse •••

1 more comment...