DevOps at the organization level

If you search images for "DevOps", you'll likely see an image like the one below.

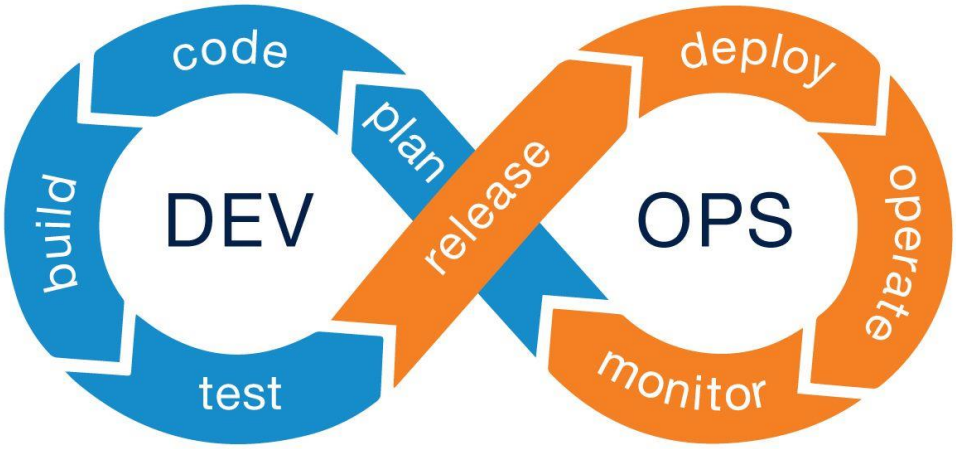


Photo credits: University of California at San Fransisco

DevOps was originally envisioned as a way to structure engineering organizations to efficiently integrate user feedback:

1. A plan is made for what users might need, often in scrum meetings between product managers and developers.
2. Developers write code based on what was planned.
3. The code is built into artifacts (containers, apps, executables, etc.)
4. Those artifacts are tested automatically and/or by QA teams.
5. Once the artifacts are confirmed to be error free by testing and QA, they are combined into a release (version 2.0.0, for example)
6. That release is distributed (the file is hosted, the containers are deployed, the apps are published, etc)
7. After the release is distributed, the application need to be monitored for code errors and scaling problems.
8. Monitoring keeps track of key metrics: Does the product feel slow, is it down?

Finally, usage metrics and feedback are incorporated into the planning phase (often in the form of Agile and "user stories").

The three pillars of DevOps Engineering

Another common definition for DevOps is as an umbrella term for "infrastructure for building and deploying code." Many developers have titles like "DevOps Engineer" or "Platform Engineer." Their jobs have little do to with planning or coding, and more to do with three key pillars:

- Pull request automation
- Deployment automation
- Application Performance Management (APM)

The goal of DevOps engineering is to release high quality software quickly, and make sure it continues being delightful and bug-free for end users.

Pillar #1: Pull request automation

Developers commonly share work with each other by proposing new sets of changes. The likeliest tools used are GitHub, BitBucket, and GitLab. Developers use git and push sets of changes together and open a "pull request" (or "merge request") to get those sets of changes integrated into the primary codebase.

Commonly, each set of changes is reviewed by another developer on the same team, but the review process can include many stakeholders in larger teams. The likeliest people to conduct reviews are:

- The developers that know the areas of code being changed (often called "code owners")
- An engineering manager or product manager in charge of the functionality being proposed.
- For visual changes: The designers that created the original specification.
- Translators, accessibility stakeholders, security reviewers, and other non-technical parties.

What can be automated

The business goal for a DevOps Engineer working on pull request automation is to speed up the review process. Some examples of functionality that can be built out include:

- Automated test running with a CI provider (what is CI?), which gives developers confidence that the change does not break existing functionality
- Per-change ephemeral environments, which helps interested parties actually interact with the proposed change to ensure it solves all required business goals.
- Automated security scanning, which helps ensure that proposed changes do not introduce new vulnerabilities into the product.
- Notifications to reviewers automatically, so that the correct reviewers can quickly request changes to a pull request.

Ideally, changes should be reviewed and merged within 24 hours of when they are made. The most common reason that software developers are slowed down is a long review cycle.

Pillar #2: Deployment automation

In a famous post from the year 2000, Joel Spolsky (founder of Stack Overflow) places "Can you make a build in one step?" as the second most important question for the health of a code base.

The efficiency of a build process isn't the only goal of Deployment automation, other goals include:

- Deploying a feature to a certain set of users as a final test before rolling it out more publicly (feature flagging and canary deployments)
- Starting new versions of services without causing downtime (blue/green deployments and rolling deployments)
- Rolling back to the prior version in case something does go wrong.

It's easy to overcomplicate deployments. Many companies have complex internal platforms for building and distributing releases.

Broadly, success in deployment automation is finding the appropriate deployment tools to fulfill business goals, and configuring them. In an ideal world, there should be little-to-no custom code for deploying.

Pillar #3: Application performance management

Even the best code can be hamstrung by operational errors. User generated content brought down Stack Overflow when a user submitted a popular post with a lot of leading spaces, for example.

The core goal of this pillar is to ensure that your service continues to perform well in production. It has four core targets for automation:

9. **Metrics**: Numerical measurements of key numbers in production. Usually in the form of finite resources (disk space, memory usage) and times (average response time, job processing time, etc)
10. **Logging:** Text descriptions of what is happening during processing. Logs often come with metadata about their source, time, and related metrics.
11. **Monitoring:** Take the metrics and logs and convert them into health metrics: Does the product feel slow, is it down, are all of the features working without errors?
12. **Alerting:** If monitoring detects a problem, the correct problem-solver should be notified automatically: In the case of an outage, an on-call engineer should be notified. Performance issues and errors with features often automatically log tickets.

Example DevOps Engineering timelines

A new product should not dive into complete automation for all of the pillars of DevOps engineering. Developers would add automation as the situation required, especially in response to user churn.

13. **A new startup with no users building a webapp**: Pillars #2 and #3 are essentially useless - outages will not be noticed by anyone. Such a product should only ensure developers can collaborate quickly. Stack: Netlify/Vercel/webapp.io
14. **A team building an app for 10 enterprise users:** Enterprise users are more sensitive to downtime. Test coverage

and business-hours alerting should be priorities. Stack: Sentry, PagerDuty, CodeCov, Bitrise/CircleCI

15. **A social media app like Reddit:** Reddit's users are in many timezones, and very sensitive to downtime. All three pillars are of vital importance. Stack: Sentry, Elasticsearch/Logstash/Kibana, Pingdom, LaunchDarkly, Terraform

Conclusion

DevOps engineering is vital for developer teams. Without being cognizant of its three pillars, customers will have a confusing and disappointing experience.

New products do not need to automate very much, but as a product matures and gets more users, it's important to dedicate resources to DevOps engineering.

What is Test Driven Development?

Test driven development is a coding methodology where tests are written before the code is written.

Colin Chartier

Colin is co-founder & CEO at Layer. He previously worked as CTO at ParseHub and as a software design lecturer at the University of Toronto. You can reach him at colin@layerci.com

More posts by Colin Chartier.

COLIN CHARTIER

27 APR 2021 • 2 MIN READ

Test Driven Development has been around for a long time: Kent Beck popularized it in the early 2000s with Test Driven Development: By Example. The idea is simple, but requires knowledge of how software testing usually works.

Software testing

Common words in software development such as "quality assurance" and "unit test" have roots in factories building physical products.

If you were running a factory building coffee makers, you would test that it worked at varying levels of completion:

Unit tests: Ensure individual components work on their own. Does the heater work? Does the tank hold water?

Integration tests: Ensure a few components work together. Does the heater heat the water in the tank?

System (end-to-end) tests: Ensure everything works together. Does the coffee maker brew a cup of coffee?

Acceptance tests: After being launched (sent to consumers), are they satisfied with the result? Are they confused with the button layout or breaking their coffee maker within the warranty period?

All of these tests have software analogues - it's useful to know which component broke in order to diagnose a problem, but it's also useful to know that the whole system is working correctly.

Test driven development: Before and after

Most developers that aren't using test driven development have a similar workflow:

Choose something to work on (the specification is usually written somewhere by someone in the product team)

Build it (ensuring that the thing built satisfies the specifications)

Test it (write small scripts that ensure that the thing which was built does not break in the future)

Steps 1 and 3 are very connected. The tests written at the end essentially codify the specification. What is success for building a coffee maker? It should heat up in 5 seconds, so write a test for that. It should brew coffee of sufficient strength, so test for that, etc.

Test driven development uses the similarity of steps 1 & 3 to flip this process on its head:

Choose something to work on.

Write tests that would pass if the software actually worked.

Keep building until all of the tests pass.

Here, the end result is the same (the software is built and tested), but all of the specifications can be clarified as the tests are being written. Test Driven Development ensures that the software created always fulfills its specification.

Conclusion

Test Driven Development is an alternative way of writing well tested software. It uses tests as a way of building a specification for how things should work before a single line of code is written.

What is CI?

Continuous Integration (CI) refers to developers continuously pushing small changes to a central Git repository numerous times per day. These changes are verified by automated software that runs comprehensive tests and ensures that no major issues are ever seen by customers.

Continuous Integration (CI) refers to developers continuously pushing small changes to a central Git repository numerous times per day. These changes are verified by automated software that runs comprehensive tests and ensures that no major issues are ever seen by customers.

First, what are tests?

CI is usually defined in terms of tests. If a developer added a feature which let users enter their credit card information, they would push:

- The code for the feature itself
- Little scripts called "tests" that made sure that the code was working properly.

There are many types of tests, you can read more about them in our article on Test Driven Development.

Why should I use CI?

CI is the first step to understanding DevOps. Imagine the simplest possible scenario: A single developer is working on a program that will

be used by a small number of users. That developer makes the original program, releases it, and the project slowly builds traction.

Now, imagine that the developer has a critical bug found a year after they've last worked on the project. The developer goes back to work on the old code and says, "Gee, this is bad code – what was I thinking? I don't understand what's going on here."

This is how development works - developers get better year after year, and need to read and understand "bad code" that might only be a year old. The only way to be confident making changes to such code is to have a test suite that ensures that new changes do not break existing functionality, and the easiest way to run these tests is via CI.

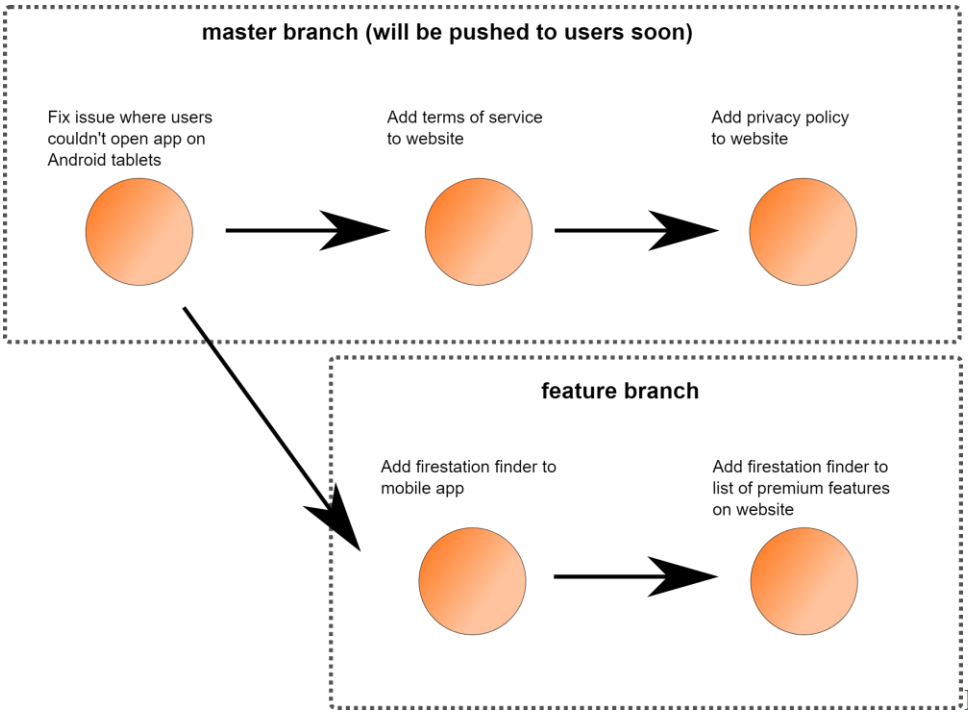
CI improves developer speed because new changes can be made confidently without having to worry about breaking existing code if the tests pass.

CI also reduces customer churn and improves user satisfaction because problems with the software are unlikely to be merged into the codebase - If the CI test runner deems there to be a bug, that change is not allowed to be pushed to users.

How do I integrate CI into my development process?
The most common approach to development at software development companies is a four-step approach that includes CI:

16. Developers work on a feature branch and push at least a commit a day to that branch on a central git repository.
17. After every commit to the feature branch, a CI server picks up the commits from the repository and runs tests, reporting any errors back to the developers of the code.
18. Once the feature is complete, the developers open a "pull request" (github) or "merge request" (gitlab) to have their changes be merged into the "master" branch. Another developer on the same team makes sure that the feature makes sense, and that there are no stylistic issues in a "code review."

19. At some point, the "master" branch gets "prod pushed" (pushed to production) so that end users see the code. This might happen automatically in CD (continuous deployment) scenarios, but is often triggered manually for smaller projects.

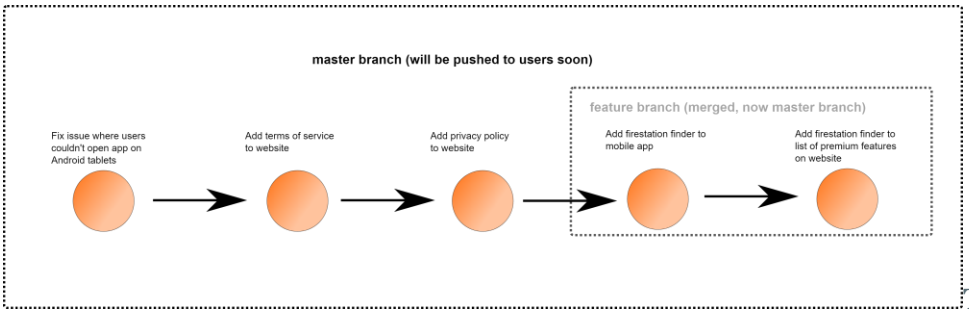


Depiction of a master branch and feature branch. Orange circles are git commits, with their respective messages written above them.

In the scenario above, there are two "branches" being worked on by separate developers. The "master branch" has been changing the website, and includes changes that add terms of service and privacy policy. The "feature branch" has been adding a new feature to the mobile app.

If there was a "prod push" at this point, the feature would not be shown to the users. Only the last commit on the master branch is ever sent to users after a prod push.

It's necessary for the developers of the feature branch to "merge" their changes into the master branch before they will be visible to users. That's where a solution like webapp.io can ensure that the newly added feature does not break existing functionality.



The feature branch is "merged", it will soon be displayed to users.

Does this process cost me anything?

Central git repositories like GitHub, Gitlab and BitBucket all have generous free tiers that don't need a developer's attention to set up.

CI providers like webapp.io also have generous free tiers - webapp.io integrates effortlessly with GitHub. All you have to do is authenticate it with your repositories, then ensure that tests are set up by anyone making new features.

Conclusion

CI is a vital tool for teams with any developers on them. It ensures that features written by another developer, or a long time ago continue to work as new changes are made. Following best practices and using Git and CI ensures that features aren't broken accidentally, which greatly reduces customer churn and improves user satisfaction.

If you need any help setting up developer infrastructure like a repository or CI, I'd be happy to help you. Send me an email at colin@webapp.io and I'll personally get you started!

What is autoscaling?

Autoscaling automates horizontal scaling to ensure that the number of workers is proportional to the load on the system.

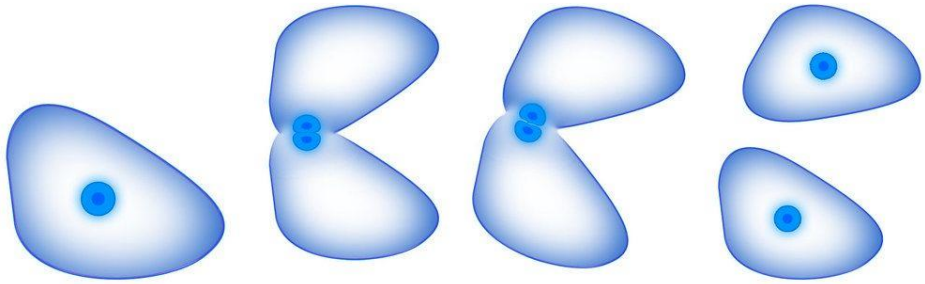
- 

Colin Chartier

Colin is co-founder & CEO at Layer. He previously worked as CTO at ParseHub and as a software design lecturer at the University of Toronto. You can reach him at colin@layerci.com

More posts by Colin Chartier.

COLIN CHARTIER
29 MAR 2021 • 2 MIN READ

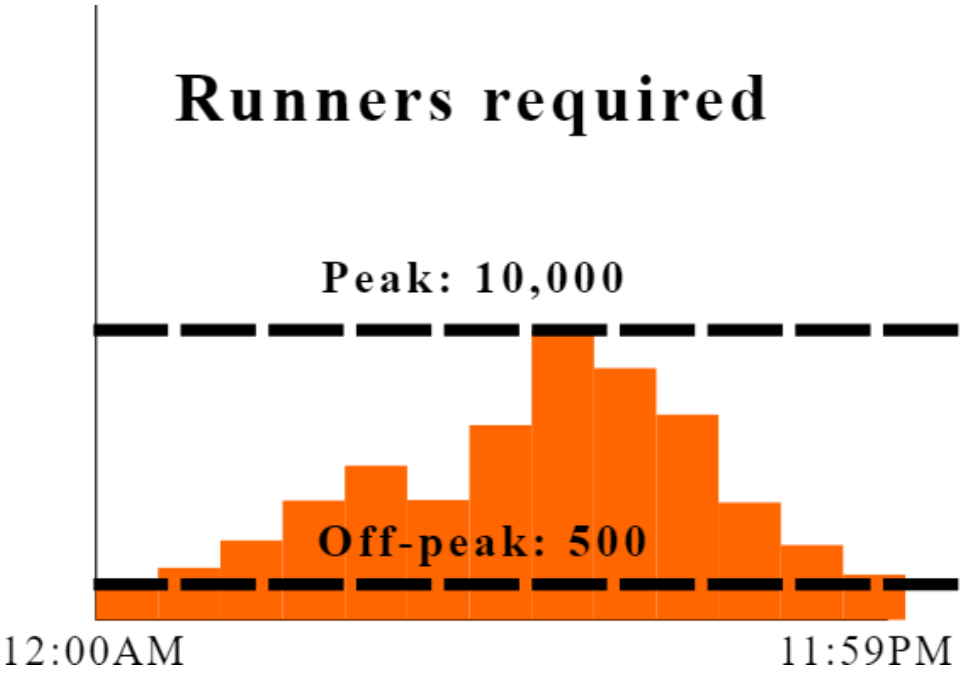


Let's see if we can clarify things with a more concrete example.

Example: A CI system

Let's say you were building a CI system (what is CI?) - your users would push code, and you'd have to spin up runners to run tests against that code. You'd see bursts of traffic during your users' business hours, and then significantly less traffic outside of business hours.

For a peak load of 10,000 concurrent CI runs, you'd need a at least 10,000 runners provisioned to avoid queuing. However, at night (off-peak hours) these 10,000 runners would mostly sit idle.



In an ideal world, you'd be able to create or destroy runners as necessary - during peak hours you'd be able to create new ones, and at off-peak hours you'd be able to destroy them. This is the idea for autoscaling.

Autoscaling in cloud providers

It's only possible to create/destroy workers because of cloud providers. At their enormous scale it's possible to offer servers for cheap on small (~1 hour) leases. The most popular technology at the writing of this post is AWS EC2 spot which acts exactly like cloud-hosted VMs, but with large discounts if you provision them for short periods of time.

Another popular technology for autoscaling is Kubernetes' Horizontal Pod Autoscaling, which can be consumed by cloud providers that provide support for it such as Google Kubernetes Engine.

Here are some additional references for autoscaling being provided by cloud providers:

- https://azure.microsoft.com/en-ca/features/autoscale/ (Azure VMs)
- https://docs.microsoft.com/en-us/azure/aks/concepts-scale (Azure containers)

- https://aws.amazon.com/getting-started/hands-on/ec2-auto-scaling-spot-instances/ (AWS VMs)
- https://docs.aws.amazon.com/eks/latest/userguide/horizontal-pod-autoscaler.html (AWS containers)
- https://cloud.google.com/compute/docs/autoscaler (Google VMs)
- https://cloud.google.com/kubernetes-engine/docs/how-to/horizontal-pod-autoscaling (Google containers)

Autoscaling vs serverless

Autoscaling is usually discussed on the timeline of ~1 hour chunks of work. If you took the concept of autoscaling and took it to its limit, you'd get serverless: Define resources that are quickly started, and use them on the timeline of ~100ms.

For example, a webserver might not exist at all until a visitor first requested a page. Instead, it would be spun up specifically for the request, then the page served, and then shut back down.

Serverless is primarily used for services that are somewhat fast to start, and stateless. You wouldn't run something like a CI run within a serverless framework, but you might run something like a webserver or notification service.

Autoscaling is primarily used for services that are slower to start or require state. You'd likely run a CI job within an autoscaled VM or container, not within a serverless container.

As of 2021, the distinction between the models is becoming quite blurred - serverless containers are becoming popular, and they often run for upwards of 1 hour. Within a few years it's likely that serverless and autoscaling will converge into a single unified interface.