What do CI and CD mean?

CI, short for *Continuous Integration*, is a software development practice in which all developers merge code changes in a central repository multiple times a day. CD stands for *Continuous Delivery*, which on top of Continuous Integration adds the practice of automating the entire software release process.

With CI, each change in code triggers an automated build-and-test sequence for the given project, providing feedback to the developer(s) who made the change. The entire CI feedback loop should run in less than 10 minutes.

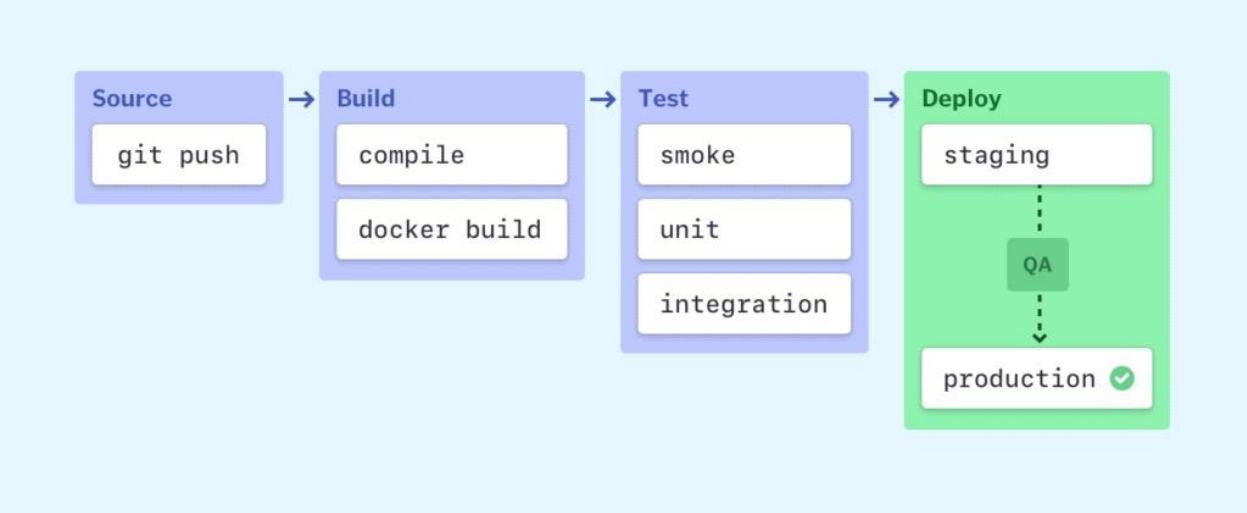
Continuous Delivery includes infrastructure provisioning and deployment, which may be manual and consist of multiple stages. What's important is that all these processes are fully automated, with each run fully logged and visible to the entire team.

Learn more here: CI/CD: Continuous Integration & Delivery Explained

Elements of a CI/CD pipeline

A CI/CD pipeline may sound like overhead, but it isn't. It's essentially a runnable specification of the steps that any developer needs to perform to deliver a new version of a software product. In the absence of an automated pipeline, engineers would still need to perform these steps manually, and hence far less productively.

Most software releases go through a couple of typical stages:



Stages of a CI/CD pipeline

Failure in each stage typically triggers a notification—via email, Slack, etc.—to let the responsible developers know about the cause. Otherwise, the whole team receives a notification after each successful deployment to production.

Source stage

In most cases, a pipeline run is triggered by a source code repository. A change in code triggers a notification to the CI/CD tool, which runs the corresponding pipeline. Other common triggers include automatically scheduled or user-initiated workflows, as well as results of other pipelines.

Build stage

We combine the source code and its dependencies to build a runnable instance of our product that we can potentially ship to our end users. Programs written in languages such as Java, C/C++, or Go need to be compiled, whereas Ruby, Python and JavaScript programs work without this step.

Regardless of the language, cloud-native software is typically deployed with Docker, in which case this stage of the CI/CD pipeline builds the Docker containers.

Failure to pass the build stage is an indicator of a fundamental problem in a project's configuration, and it's best to address it immediately.

Test stage

In this phase, we run automated tests to validate our code's correctness and the behavior of our product. The test stage acts as a safety net that prevents easily reproducible bugs from reaching the end-users.

The responsibility of writing tests falls on the developers. The best way to write automated tests is to do so as we write new code in test- or behavior-driven development.

Depending on the size and complexity of the project, this phase can last from seconds to hours. Many large-scale projects run tests in multiple stages, starting with smoke tests that perform quick sanity checks to end-to-end integration tests that test the entire system from the user's point of view. An extensive test suite is typically parallelized to reduce run time.

Failure during the test stage exposes problems in code that developers didn't foresee when writing the code. It's essential for this stage to produce feedback to developers quickly, while the problem space is still fresh in their minds and they can maintain the state of flow.

Deploy stages

Once we have a built a runnable instance of our code that has passed all predefined tests, we're ready to deploy it. There are usually multiple deploy environments, for example, a "beta" or "staging" environment which is used internally by the product team, and a "production" environment for end-users.

Teams that have embraced the Agile model of development—guided by tests and real-time monitoring—usually deploy work-in-progress manually to a staging environment for additional manual testing and review, and automatically deploy approved changes from the master branch to production.

Examples of CI/CD pipelines

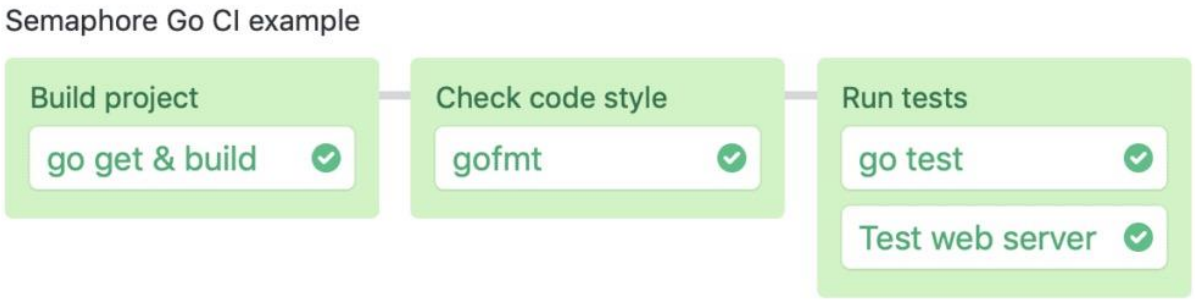
Pipelines reflect the complexity of a project. Configuring even the simplest pipeline with one job that runs on every code change will save a team many headaches in the future.

On Semaphore, pipelines can easily be extended with multiple sequential or parallel blocks of jobs. Pipelines can also be extended using promotions that are triggered manually or automatically, based on custom conditions.

A pipeline for a simple program

A pipeline can start very simple. Here's an example of a Go project pipeline that:

- Compiles the code,
- Checks code style, and
- Runs automated tests in two parallel jobs:



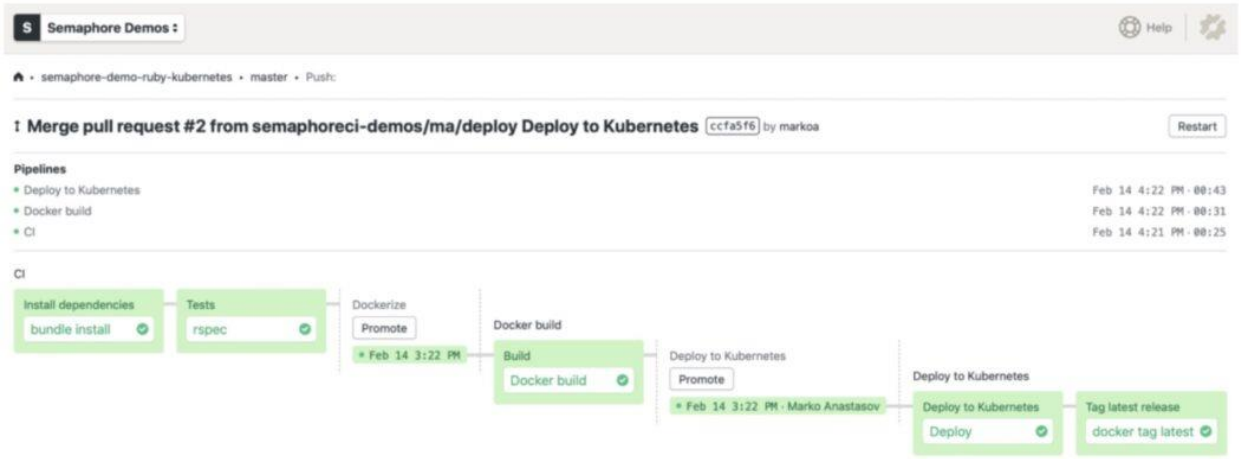
A simple CI pipeline for a Go project

A continuous delivery pipeline with Docker and Kubernetes

Using Docker increases the complexity of CI/CD pipelines by requiring developers to build and upload a large container image in each code build.

The benefits of easy developer onboarding and standardized deployment, for many teams, outweigh the cost.

Here's a pipeline that builds, tests and deploys a microservice to a Kubernetes cluster:



a CI/CD pipeline with Docker and Kubernetes

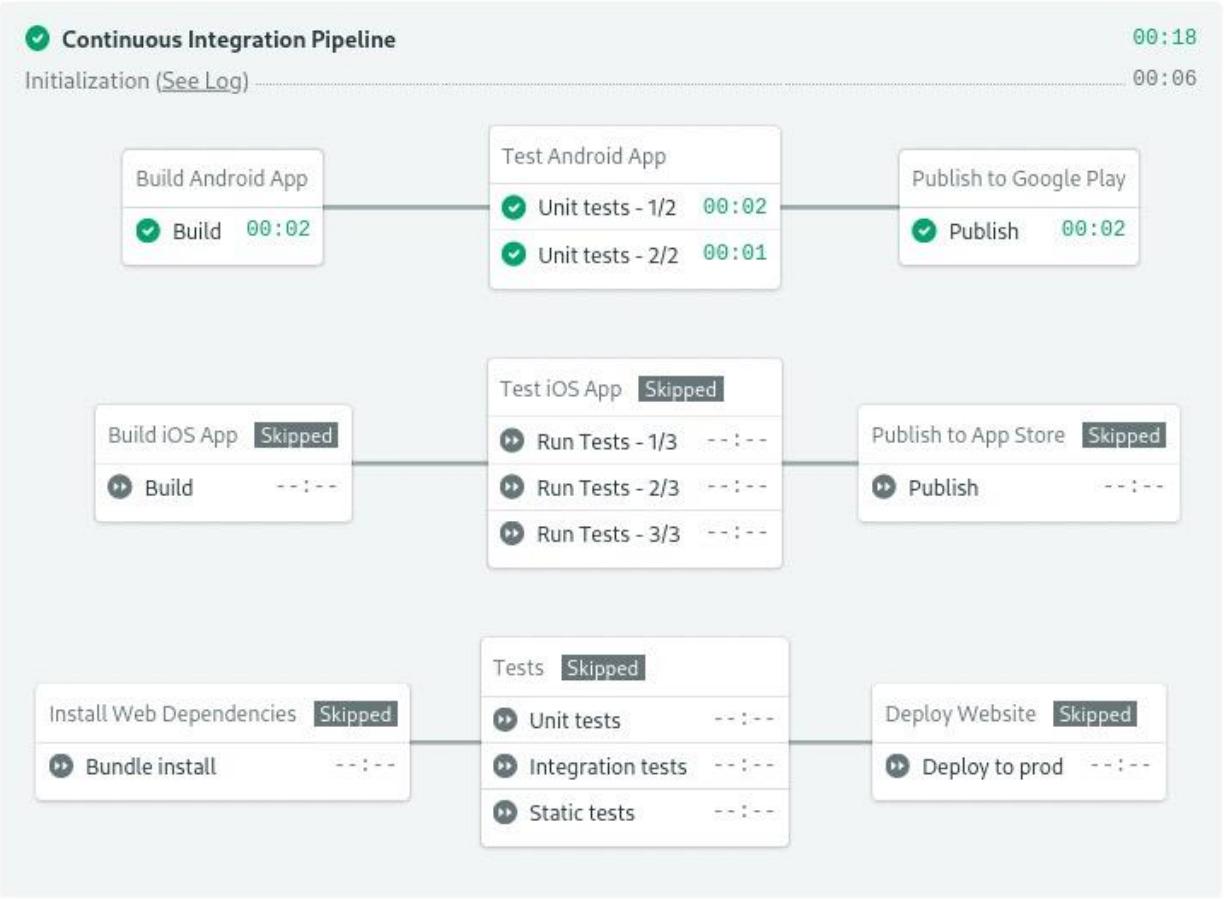
Using a private Docker registry on Semaphore (available on Enterprise Cloud plan), developers can get large performance boosts in their pipelines as they avoid the overhead of communicating with an external cloud registry in every CI/CD workflow.

CI/CD pipeline for a monorepo

Developing with a monorepo means having a single, version-controlled code repository that contains many projects. The projects can be related, but are often logically independent and typically run by different teams.

Doing CI/CD with a monorepo is a challenge. By default, every code change would launch a pipeline that would build, test, and deploy every single project included in the monorepo. This is time-wasting, costly, and a risky.

Semaphore automatically detects what has changed in a monorepo and lets developers specify what parts of the pipeline to run based on that:



Monorepo continuous integration pipeline

Additional benefits of pipelines

Having a CI/CD pipeline has more positive effects than only making an existing process a little more efficient:

- Developers can stay focused on writing code and monitoring the behavior of the system in production.
- QA and product stakeholders have easy access to the latest, or any, version of the system.
- Product updates are not stressful.
- Logs of all code changes, tests and deployments are available for inspection at any time.
- Rolling back to a previous version in the event of a problem is a routine push-button action.
- A fast feedback loop helps build an organizational culture of learning and responsibility.

Learn more here: 7 ways in which continuous delivery helps build a culture of learning

What makes a good pipeline?

A good CI/CD pipeline is fast, reliable, and accurate.

Speed

Speed manifests itself in several ways:

- **How quickly do we get feedback on the correctness of our work?** If it's longer than the time it takes to get a coffee, pushing code to CI is equivalent to asking a developer to join a meeting in the middle of solving a problem. Developers will work less effectively due to inevitable context switching.
- **How long does it take us to build, test, and deploy a simple code commit?** For example, a total time of one hour for CI and deployment means that the entire engineering team has a hard limit of up to seven deploys for the whole day. This causes developers to opt for less frequent and more risky deployments, instead of the rapid change that businesses today need.
- **Do our CI/CD pipelines scale to meet development demands in real time?** Traditionally CI/CD pipelines have limited capacity, meaning that only a certain number of pipelines can run at a given time. As a result, resources sit idle most of the time, while developers wait in a queue for CI/CD to become available at busy periods of the day. One of the biggest changes in the recently released Semaphore 2.0 is auto-scaling and a pay-as-you-go pricing model, a "serverless" operating principle that supports developer productivity.
- **How quickly can we set up a new pipeline?** Difficulty with scaling CI/CD infrastructure or reusing existing configuration creates friction, which stifles development. Today's cloud infrastructure is best utilized by writing software as a composition of microservices, which calls for frequent initiation of new CI/CD pipelines. This is solved by having a programmable CI/CD tool that fits in the existing development workflows and storing all CI/CD configuration as code that can be reviewed, versioned, and restored.

More info here: Why cloud-native success depends on high velocity CI/CD.

Reliability

A reliable pipeline always produces the same output for a given input, and with no oscillations in runtime. Intermittent failures cause intense frustration among developers.

Operating and scaling CI/CD infrastructure that provides on-demand, clean, identical and isolated resources for a growing team is a complex job. What seems to work well for one project or a few developers usually breaks down when the team and the number of projects grow, or the technology stack changes. When we hear from new users, unreliable CI/CD is one of the top reasons why they move to Semaphore, often from a self-hosted solution.

Accuracy

Any degree of automation is a positive change. However, the job is not fully complete until the CI/CD pipeline **accurately runs and visualizes the entire software delivery process**. This requires using a CI/CD tool that can model both simple and if needed, complex workflows, so that manual error in repetitive tasks is all but impossible.

For example, it's not uncommon to have the CI phase fully automated but to leave out deployment as a manual operation to be performed by often a single person on the team. If a CI/CD tool can model the deployment workflow needed, for example with use of secrets and multi-stage promotions, this bottleneck can be removed.

Good things to do

When the master is broken, drop what you're doing and fix it. Maintain a "no broken windows" policy on the pipeline.

Run fast and fundamental tests first. If you have a large test suite, it's common practice to parallelize it to reduce the amount of time it takes to run it. However, it doesn't make sense to run all the time-consuming UI tests if some essential unit or code quality tests have failed. Instead, it's best to set up a pipeline with multiple stages in which fast and fundamental tests—such as security scanning and unit tests—run first, and only once they pass does the pipeline move on to integration or API tests, and finally UI tests.

Always use exactly the same environment. A CI/CD pipeline can't be reliable if a pipeline run modifies the next pipeline's environment. Each workflow should start from the same, clean, and isolated environment.

Build-in quality checking. For example, there are open source tools that provide static code analysis for every major programming language, covering everything from code style to security scanning. Run these tools within your CI/CD pipeline and free up brainpower for creative problem-solving.

Include pull requests. There's no reason why a CI/CD pipeline should be limited to one or a few branches. By running the standard set of tests against every branch and corresponding pull request, contributors can identify issues before engaging in peer review, or worse, before the point of integration with the master branch. The outcome of this practice is that merging any pull request becomes a non-event.

Peer-review each pull request. No CI/CD pipeline can fully replace the need to review new code. There may be times when the change is indeed so trivial that peer review is a waste of time; however, it's best to set the norm that every pull request needs another pair of eyes and make exceptions only when it makes sense, rather than vice versa. Peer code review is a key element in building a robust and egoless engineering culture of collaborative problem-solving.

Top 10 Benefits of Continuous Integration and Continuous Delivery

1. Smaller Code Changes

One technical advantage of continuous integration and continuous delivery is that it allows you to integrate small pieces of code at one time. These code changes are simpler and easier to handle than huge chunks of code and as such, have fewer issues that may need to be repaired at a later date.
Using continuous testing, these small pieces can be tested as soon as they are integrated into the code repository, allowing developers to recognize a problem before too much work is completed afterward. This works really well for large development teams who work remotely as well as those in-house as communication between team members can be challenging.

2. Fault Isolations

Fault isolation refers to the practice of designing systems such that when an error occurs, the negative outcomes are limited in scope. Limiting the scope of problems reduces the potential for damage and makes systems easier to maintain.

Designing your system with CI/CD ensures that fault isolations are faster to detect and easier to implement. Fault isolations combine monitoring the system, identifying when the fault occurred, and triggering its location. Thus, the consequences of bugs appearing in the application are limited in scope. Sudden breakdowns and other critical issues can be prevented from occurring with the ability to isolate the problem before it can cause damage to the entire system.

3. Faster Mean Time To Resolution (MTTR)

MTTR measures the maintainability of repairable features and sets the average time to repair a broken feature. Basically, it helps you track the amount of time spent to recover from a failure.

CI/CD reduces the MTTR because the code changes are smaller and fault isolations are easier to detect. One of the most important business risk assurances is to keep failures to a minimum and quickly recover from any failures that do happen. Application monitoring tools are a great way to find and fix failures while also logging the problems to notice trends faster.

4. More Test Reliability

Using CI/CD, test reliability improves due to the bite-size and specific changes introduced to the system, allowing for more accurate positive and negative tests to be conducted. Test reliability within CI/CD can also be considered Continuous Reliability. With the continuous merging and releasing of new products and features, knowing that quality was top of mind throughout the entire process assures stakeholders their investment is worthwhile.

5. Faster Release Rate

Failures are detected faster and as such, can be repaired faster, leading to increasing release rates. However, frequent releases are possible only if the code is developed in a continuously moving system.

CI/CD continuously merges codes and continuously deploys them to production after thorough testing, keeping the code in a release-ready state. It's important to have as part of deployment a production environment set up that closely mimics that which end-users will ultimately be using. Containerization is a great method to test the code in a production environment to test only the area that will be affected by the release.

6. Smaller Backlog

Incorporating CI/CD into your organization's development process reduces the number of non-critical defects in your backlog. These small defects are detected prior to production and fixed before being released to end-users.

The benefits of solving non-critical issues ahead-of-time are many. For example, your developers have more time to focus on larger problems or improving the system and your testers can focus less on small problems so they can find larger problems before being released. Another benefit (and perhaps the best one) is keeping your customers happy by preventing them from finding many errors in your product.

7. Customer Satisfaction

The advantages of CI/CD do not only fall into the technical aspect but also in an organization scope. The first few moments of a new customer trying out your product is a make-or-break-it moment.

Don't waste first impressions as they are key to turning new customers into satisfied customers. Keep your customers happy with fast turnaround of new features and bug fixes. Utilizing a CI/CD approach also keeps your product up-to-

date with the latest technology and allows you to gain new customers who will select you over the competition through word-of-mouth and positive reviews.
Your customers are the main users of your product. As such, what they have to say should be taken into high consideration. Whether the comments are positive or negative, customer feedback and involvement leads to usability improvements and overall customer satisfaction.
Your customers want to know they are being heard. Adding new features and changes into your CI/CD pipeline based on the way your customers use the product will help you retain current users and gain new ones.

8. Increase Team Transparency and Accountability

CI/CD is a great way to get continuous feedback not only from your customers but also from your own team. This increases the transparency of any problems in the team and encourages responsible accountability.
CI is mostly focused on the development team, so the feedback from this part of the pipeline affects build failures, merging problems, architectural setbacks, etc. CD focuses more on getting the product quickly to the end-users to get the much-needed customer feedback. Both CI and CD provide rapid feedback, allowing you to steadily and continuously make your product even better.

9. Reduce Costs

Automation in the CI/CD pipeline reduces the number of errors that can take place in the many repetitive steps of CI and CD. Doing so also frees up developer time that could be spent on product development as there aren't as many code changes to fix down the road if the error is caught quickly. Another thing to keep in mind: increasing code quality with automation also increases your ROI.

10. Easy Maintenance and Updates

Maintenance and updates are a crucial part of making a great product. However, it's important to note within a CI/CD process to perform maintenance during downtime periods, also known as the non-critical hour. Don't take the system down during peak traffic times to update code changes.
Upsetting customers is one part of the problem, but trying to update changes during this time could also increase deployment issues. Make sure the pipeline runs smoothly by incorporating when to make changes and releases. A great way to ensure maintenance doesn't affect the entire system is to create microservices in your code architecture so that only one area of the system is taken down at one time.

Conclusion

There are many tools that can help enable a smoother transition to a CI/CD process. Testing is a large part of that process because even if you are able to make your integrations and delivery faster, it would mean nothing if was done so without quality in mind. Also, the more steps of the CI/CD pipeline that can be automated, the faster quality releases can be accomplished.

To enhance your CI/CD pipeline, take this **free** self-assessment to learn where your team is at in the continuous testing maturity roadmap and get expert tips for percentage of automated tests, test reports, and more.