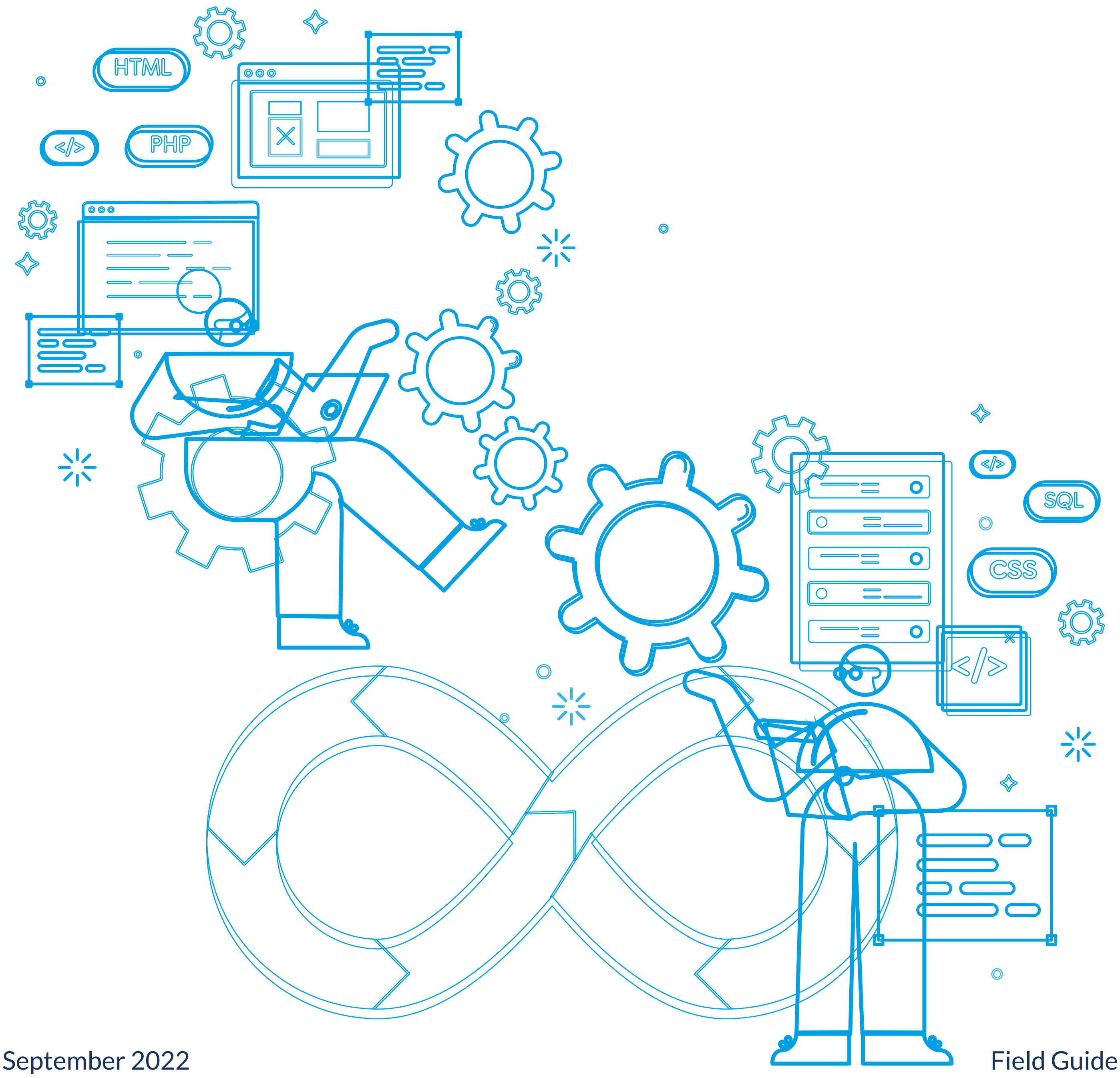
CODIFICATION

What are CI/CD Pipelines?



September 2022

Field Guide

Table of Contents

Introduction

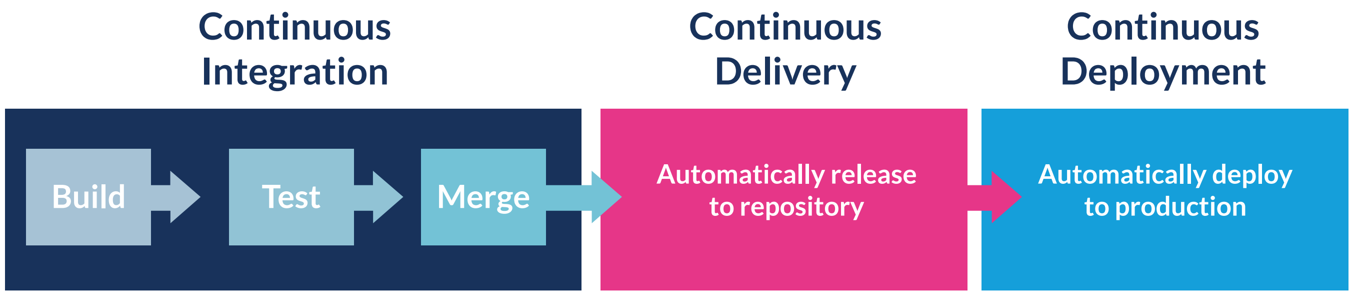
Under the DevOps banner, continuous integration and continuous delivery (CI/CD) are two methodologies (the combining of development and operations). The infrastructure provisioning, build, testing, and deployment processes that were formerly necessary to get new code from a commit into production are now mostly or entirely automated by CI/CD. Using a CI/CD pipeline, developers can update code, and the updated code is provided, tested, and deployed right away. Code releases occur more quickly and downtime is decreased with proper CI/CD implementation.

What are CI/CD Pipelines?

A flexible DevOps workflow centred on a regular and dependable software delivery procedure is the continuous integration/ continuous delivery (CI/CD) pipeline. DevOps teams may work together and in real-time to write code, integrate it, run tests, deliver releases, and distribute modifications to the program thanks to the iterative process used instead of a linear approach.

The use of automation to guarantee code quality is an important component of the CI/CD workflow. Test automation is used to deploy code changes to various environments, deliver applications to production environments, and uncover dependencies and other issues earlier as the software changes move through the pipeline. The automation's task in this situation is to perform quality control, evaluating factors like performance, API usage, and security. This guarantees that the changes made by every team member are thoroughly incorporated and work as intended.

Development teams can increase quality, productivity, and other DevOps KPIs by automating some steps of the CI/CD process.



Continuous Integration (CI)

Continuous integration is a DevOps software development process that involves developers consistently merging their code changes into a common repository, which is then followed by automated builds and tests. CI refers to the build or integration stage of the software release process, which comprises both an automated component (such as a CI or build service) and a culture component (e.g. learning to integrate frequently). The main goal is to detect and fix issues more quickly, improve software quality, and reduce the time it takes to validate and publish new software upgrades.

Previously, developers on a team would typically spend a lot of time working on their own projects, and only merge their changes to the master branch after they were done. This resulted in defects building up, which can take a long time to fix, making it harder to provide clients with timely updates.

Developers routinely commit to a shared repository with CI using a version control system such as Git. They have the option to run local unit tests on their code before every commit as an additional layer of verification before merging. A CI service then automatically executes unit tests on the changes.

Continuous Delivery (CD)

Continuous integration and continuous delivery are software development practices that automate the process of setting up infrastructure and releasing applications.

CD takes over during the last stages after code has been tested and built as part of the CI process to make sure it can be packed with all it needs to deploy to any environment at any time. Everything from infrastructure provisioning to application deployment in a testing or production environment can be covered by CD.

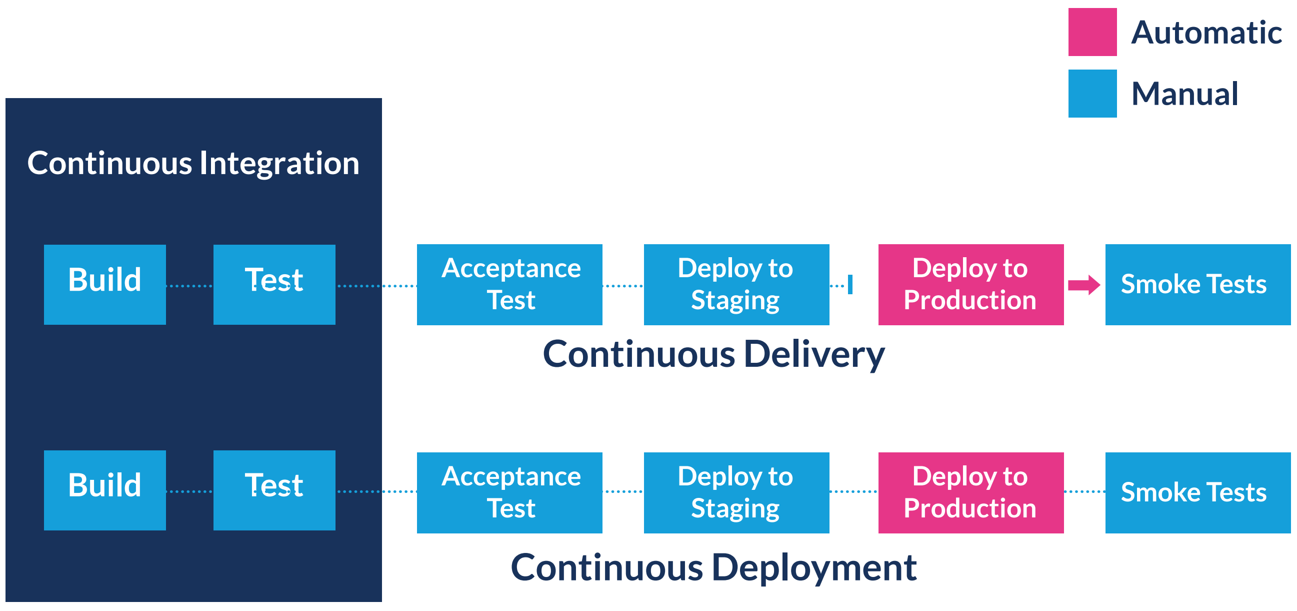
The program created using CD can be delivered to production at any moment. Then you have the option of manually starting the deployments or switching to continuous deployment, where the deployments are also automated.

Continuous Deployment

Organisations can deploy their applications automatically with the help of continuous deployment, doing away with the requirement for human interaction. With continuous deployment, DevOps teams predetermine the requirements for code releases; if those requirements are satisfied and verified, the code is released into the production environment. Organisations can be more flexible and deliver new services to users more quickly as a result of this form of automation.

While continuous integration can be carried out without continuous delivery or deployment, CD actually can't be done without CI already in place. That's because if you aren't using CI essentials like merging code to a shared repo, automating testing and builds, and doing it all in tiny batches every day, it would be very challenging to be able to release to production at any time.

Simply expressed, both continuous delivery and continuous deployment are components of continuous integration. Additionally, automatic releases make continuous deployment similar to continuous delivery.



Relationship with DevOps

CI/CD is a crucial component of DevOps and any contemporary software development methodology. By boosting an organisation's output, boosting efficiency, and optimising workflows through integrated automation, testing, and collaboration, a purpose-built CI/CD platform may maximise development time. The CI/CD features can aid in reducing the complexity of application development as they get bigger. Other DevOps approaches can be used to assist eliminate development silos, scale securely, and maximise the benefits of CI/CD. These strategies include shifting left on security and establishing tighter feedback loops.

Why are CI/CD Pipelines Important?

Dev and Ops experts can work as effectively and efficiently as possible thanks to CI/CD, which is crucial. It reduces time-consuming manual development effort and outdated approval procedures, allowing DevOps teams to develop software more creatively. Process automation reduces the chance of human mistakes by making procedures predictable and repeatable. To lower the risk of build-breaking changes, DevOps teams can integrate smaller changes more frequently and receive faster feedback. Software development lifecycles are shortened by making DevOps processes continuous and iterative, enabling businesses to release more products that consumers appreciate.

Fundamentals of CI/CD

Eight key components of CI/CD help guarantee your development lifecycle is as effective as possible. They cover both deployment and development. To enhance your DevOps workflow and software delivery, incorporate these essentials into your pipeline:

1. Single source repository
2. Recurring check-ins with the main branch
3. Automated builds
4. Self-testing builds
5. Recurring iterations
6. Stable testing environments
7. Maximum visibility
8. Always available predictable deployments

1. Single source repository

The source code management (SCM) system that stores all the files and scripts required to produce builds All the components required for the build should be in the repository. Source code, database architecture, libraries, properties files, and version control are all included. Additionally, test scripts and scripts for creating apps must be included.

2. Recurring check-ins with the main branch

Incorporating new code frequently and early into your trunk, mainline, or master branch - also known as trunk-based development. Work exclusively with the main branch and avoid any sub-branches. As regularly as you can, merge little sections of code into the branch. Only one change should be merged at a time.

3. Automated builds

Scripts should have all the components you need to create something from a single command. Web server files, database scripts, and application software are all included in this. The code should be automatically packaged and compiled into a usable application by the CI processes.

4. Self-testing builds

Testing scripts should make sure that when a test fails, the build also fails. Static pre-build testing scripts can be used to verify the integrity, calibre, and security of the code. Only let code into the build that passes static tests.

5. Recurring iterations

There are fewer hiding places for disputes in the repository as a result of several commits. Instead of making large adjustments, make quick, modest modifications. If there is an issue or controversy, this makes it simple to roll back modifications.

6. Stable testing environments

A copy of the production environment should be used to test the code. The operational production version cannot be used to test new code. Make a replica of the real world that is as accurate as you can. To find and track down bugs that evaded the first pre-build testing phase, use thorough testing scripts.

7. Maximum visibility

Every developer should have access to the most recent executables and be able to observe any repository modifications. The repository's information ought to be accessible to everyone. Manage handoffs using version control so that developers are aware of the most recent version. Everyone can monitor progress and spot possible problems when there is maximum visibility.

8. Always available predictable deployments

The team feels comfortable carrying out deployments whenever they like because they are so common and low-risk. Processes for CI/CD testing and verification should be exacting and trustworthy. The team now feels confident deploying upgrades whenever they are needed.

CI/CD Pipelines in Action

A good pipeline moves quickly and consistently. Fast, secure, and repeatable pipelines are even better

Before code reaches an end user, a great pipeline will finish in under an hour and find 95% of anomalies and regressions. You may want to reevaluate your CI/CD pipeline design and strategy if it takes more than an hour for your code to reach production or if more than two out of every ten deployments fail.

Objectives

A CI/CD pipeline may represent a variety of objectives. A CI/CD pipeline's organisational structure typically mirrors the objectives that it is motivated by.

Motivated by environments

The number of locations a service must be deployed to grows as systems become more dispersed. Your CI/CD pipelines will typically be more deployment-centric, preferring the orchestration of all the environments a service must pass through, if your primary objective is to deploy to many environments/locations.

Motivated by services

As microservices have grown in popularity, deployments now frequently involve many services. The deployment of many services simultaneously (or sequentially) is required if the pipeline is being utilised for service orchestration. These pipelines are frequently used to orchestrate a variety of services, maintaining consistency across all deployments.

Motivated by services

People were very interested in moving deployments along before there were pipes. These pipelines typically run for a long time and are used to monitor human workflow if the deployment process is still largely manual with numerous permission gates (i.e., driven by people). These anti-patterns are avoided by contemporary pipelines.

Motivated by tests

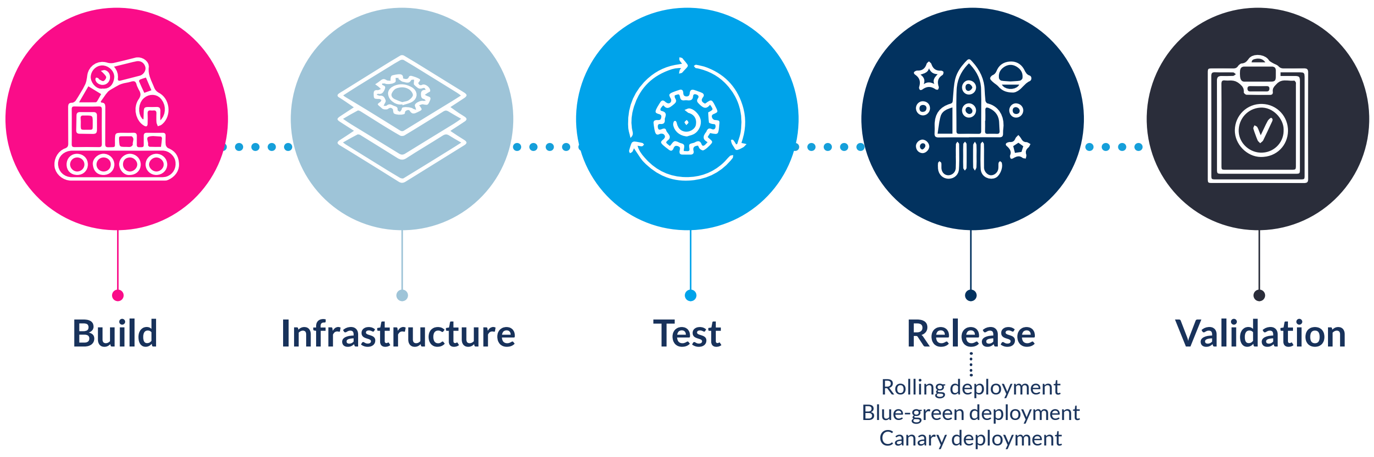
The usage of CI/CD pipelines for test automation and orchestration is widespread. A logical place for the automation that advances the testing is in your pipeline because you have to link together numerous different testing approaches. Longer "time per stage" occurs when the pipeline comes closer to production as testing rigour grows.

Motivated by outcome

The feature must eventually correspond to the expectation. Pipelines that prioritize results typically include lengthier final stages and the capacity to continue after the deployment is finished. If SLAs, SLOs, or SLIs are violated, the pipeline serves as a conduit to restore MTTR. This is an example of an outcome. Hours or days following a deployment, there may be a regression or a change in the result.

Elements of a CI/CD pipeline

The range of typical CI/CD pipeline building blocks extends from the deployment of source code to production.



Build

For possible deployment, source code must be created and packaged. To automate this step of the CI/CD process, a wide variety of Continuous Integration technologies are available. The deployable units need calling and executing the build tools of the languages being used because they are language-dependent. To create a JAVA distribution, for instance, Maven or Gradle must be called. A packing process may also include the automated build components. Continuing with the Java example, if a Docker image of the Java application needs to be created, the necessary Docker Compose stages must be called. Tests that are build-specific, including unit tests and dependency scanning, can be run in the build elements.

Infrastructure

Modern CI/CD pipeline generations are aware of their infrastructure. With the rise of infrastructure-as-code, infrastructure is now provisioned during pipeline execution as opposed to pipelines of the past where infrastructure was waiting before application deployment. The development of the CI/CD pipeline is controlled by the success or failure of infrastructure provisioning. Infrastructure provisioning, such as running a Terraform Script or Cloud Provider Script, is used to prepare the following environment as an item moves through it.

Test

Instilling confidence is one of the main objectives of most pipelines. Running tests is the standard method for fostering trust in software. Test components come in a variety of sizes and shapes. CI/CD pipelines are logical locations to run the tests as quality gates when test methodology change. Tests that require the full application, such as integration tests, soak tests, load tests, and regression tests, in addition to build-centric tests, are a perfect fit. Infrastructure-level testing can also be done using contemporary testing techniques like Chaos Engineering.

Release

Release elements are the actual deployment process that aids in continuous deployment. Do you need to deploy in a rolling, canary, or blue-green manner? Your CI/CD pipeline's release components will handle that orchestration.

Rolling deployment

Running instances are updated sequentially as part of a rolling deployment. To elaborate on that, until all nodes in the sequence have been replaced, the old program version is shut down, and a new version is then installed in its stead.

Blue-green deployment

A release strategy made for safety is known as blue-green deployment. The stable version (green) will continue to function until it is deemed safe to repurpose or decommission it. With two parallel versions of production operating, the new release (blue) will replace the stable version (green). Rollbacks are simpler when blue-green deployments are used. On the other hand, setting up and maintaining the necessary infrastructure (two copies of production) might be expensive.

Canary deployment

An incremental release approach called a “canary deployment” gradually replaces the stable version with the new change (the canary). Canary deployments are carried out through several stages. For instance, if the first phase is successful, the node swap could grow to 50% of the nodes, 100% of the nodes, and finally 10% of the nodes. The safety they offer during a release and the fact that they use fewer resources than a blue-green deployment is the key justifications for implementing canary deployments. On the other hand, because canaries must undergo validation before being promoted, canary deployments can be difficult.

Validation

Validation components can be used as decision points to advance along the pipeline, keep promoting artefacts, and keep track of deployed systems. Numerous monitoring and observability tools are best in class when it comes to their capacity to detect regression. These monitoring and observability tools are only a few of the signals that modern CI/CD pipelines can use to assess if advancement or rollback should take place. The health of what is or has been deployed can also be routinely validated by modern CI/CD workflows. It’s crucial for continuous testing objectives that testing and validation continue after a deployment.

Benefits of CI/CD pipeline

Businesses and organisations that embrace CI/CD frequently observe several beneficial effects. Here are some advantages of CI/CD implementation that you might anticipate:

- ✓ **Better customer satisfaction:** Your users and customers will have a better experience because fewer defects and problems are introduced into production. Customer satisfaction, confidence, and reputation all increase as a result.
- ✓ **Better value on time:** You can advertise new features and goods faster when you can deploy them at any moment. A quicker turnaround lowers your development costs and frees up your staff for additional tasks. Customers benefit from quicker outcomes and competitive advantage.
- ✓ **Better synergy:** Fire drills can be significantly reduced by testing code more frequently, in smaller batches, and earlier in the development cycle. This leads to a seamless development cycle and reduced team tension. Results are more consistent, and errors are simpler to locate and address.
- ✓ **Reliability:** By reducing deployment bottlenecks and making deployments predictable, the uncertainty surrounding fulfilling deadlines can be eliminated. When work is broken down into smaller, manageable portions, it is easier to complete each stage on time and monitor progress and deadlines.
- ✓ **More time for developers:** Because more of the deployment process is automated, the team has more time to reward initiatives and increase their productivity. Developers spend between 35 and 50 per cent of their effort validating, debugging, and testing their code.
- ✓ **Less context switching:** When developers receive real-time feedback on the code they commit, it is easier to focus on one task at a time and reduce cognitive load. They can debug code faster and improve their productivity. Finding bugs is easier when there is less code to review.
- ✓ **Less stress on teams:** According to research, continuous delivery significantly lessens team fatigue and deployment discomfort. When using CI/CD methods, developers feel less stress and annoyance. Employees who are happier, healthier, and less burned out as a result.
- ✓ **Faster recovery:** Fixing problems and recovering from incidents is made simpler with CI/CD (MTTR). The continuous deployment process enables frequent and minor software upgrades, making it easier to identify defects when they occur. The customer can resume working if developers want to roll back the modification or resolve bugs.

What Does a Good Pipeline Look Like?

A robust CI/CD pipeline is quick, dependable, and precise.

Speed

There are various ways that speed renders:

How quickly do we receive feedback on how well our work is done? Pushing code to CI is akin to inviting a developer to a meeting while they are in the middle of addressing an issue if it takes longer than it takes to grab a coffee. Developer productivity will suffer as a result of constant context switching.

How much time does it take to develop, test, and deliver a straightforward code commit? For instance, if CI and deployment take an hour in total, the engineering team's daily deployments are strictly limited to no more than seven. Due to this, developers choose less frequent but riskier deployments rather than the quick change that businesses today require.

Do our CI/CD pipelines scalable to accommodate changing development demands? In the past, CI/CD pipelines had a restricted capacity, which limited the number of pipelines that could operate simultaneously. As a result, resources are typically idle, and developers must queue up for CI/CD to become available during peak hours. Auto-scaling and a pay-as-you-go pricing mechanism, a "serverless" operating philosophy that enhances developer productivity, are two of the key enhancements in the recently launched Semaphore 2.0.

How quickly can a new pipeline be established? The friction that results from increasing CI/CD infrastructure or from reusing the current setup slows down development. The ideal way to use the cloud infrastructure of today is to write software as a collection of microservices, which necessitates frequently starting new CI/CD pipelines. A programmable CI/CD solution that integrates with the existing development workflows and stores all CI/CD configurations as code that can be reviewed, versioned, and restored solves this problem.

Reliability

For a given input, a reliable pipeline consistently generates the same output with no runtime fluctuations. Developers are extremely frustrated by sporadic failures.

It takes a lot of work to maintain and scale CI/CD infrastructure that offers a growing team of on-demand, spotless, identical, and isolated resources. When the team, the number of projects, or the technological stack change, what initially seems to work well for one project or a small group of engineers frequently fails. Unreliable CI/CD is one of the main reasons that new users, frequently coming from a self-hosted solution, cite for switching to Semaphore.

Accuracy

Automation is a good thing in any amount. The CI/CD pipeline must accurately run and visualise the entire software delivery process before the work is considered finished. This calls for the use of a CI/CD solution that can model both straightforward and, if necessary, complicated workflows, making it virtually impossible for a manual error to occur during recurring activities.

For instance, it's not unusual to have the CI phase entirely automated yet leave out deployment as a manual task that is frequently completed by a single team member. This barrier can be eliminated if a CI/CD solution can simulate the required deployment workflow, for instance using secrets and multi-stage promotions.

Best Practices

When the master is broken, drop what you're doing and fix it. Broken master can prevent the execution of a continuous deployment/delivery pipeline, in which deployment jobs are carried out after the test stage in master pipelines. So maintain the pipeline with a "nothing broken should exist" policy.

Perform quick and basic tests initially. It is usual practice to parallelize a big test suite to speed up the execution time. However, if any code quality tests have failed, it's best to set up a pipeline with quick and basic tests—like security scanning and unit tests—run initially, and the pipeline only moves on to integration or API testing once they pass, and then UI tests.

Utilise the same environments at all times
A CI/CD pipeline cannot be trusted if one pipeline run changes the environment for the following pipeline. Every workflow ought to begin in the same tidy, isolated environment.

Internal quality control. For each major programming language, there are open source tools that offer static code analysis, encompassing everything from code style to security scanning. Utilise these tools as part of your CI/CD workflow to free up thought for original problem-solving.

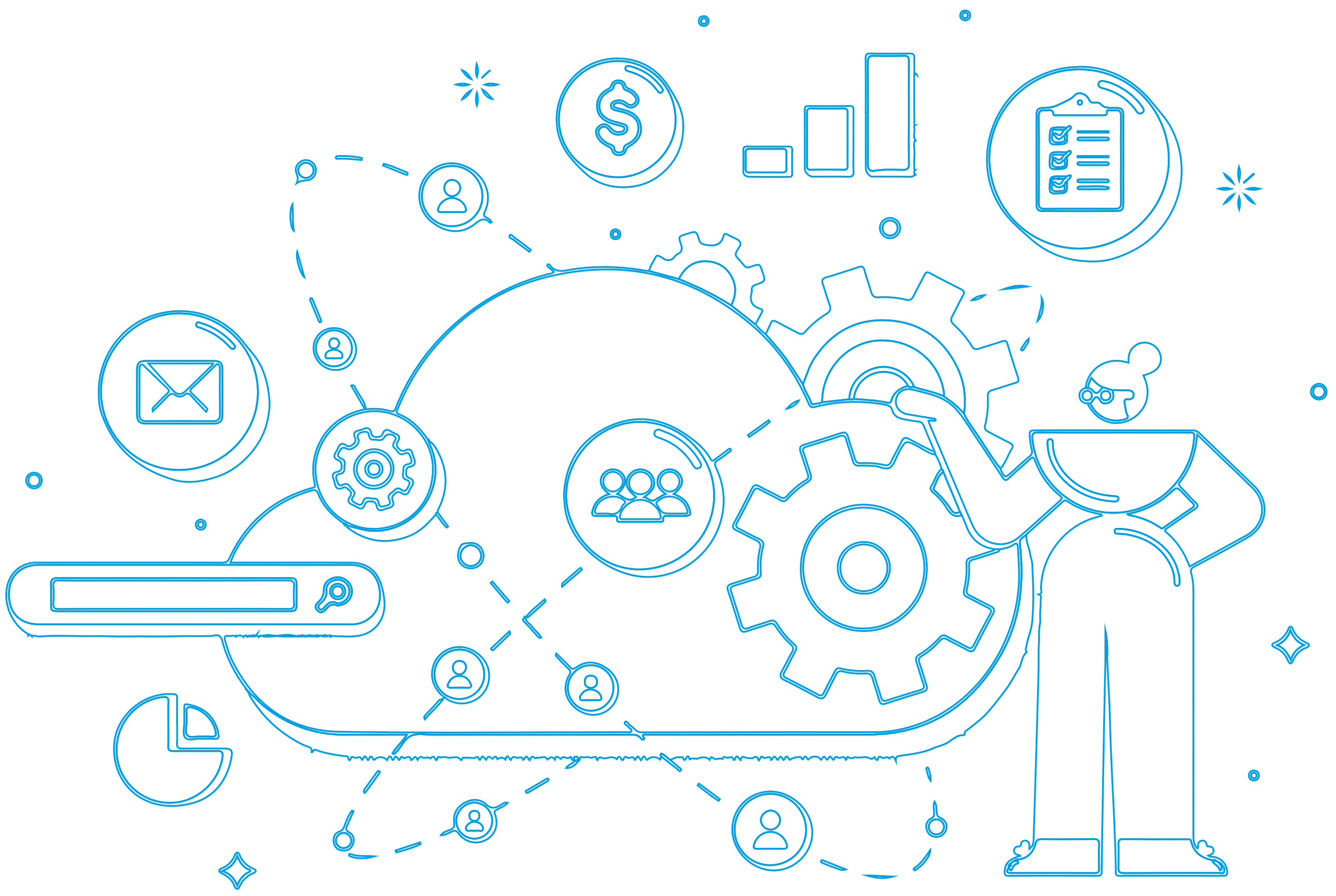
Pull requests must be included. A CI/CD pipeline shouldn't be constrained to just one or a few branches. Contributors can find problems before peer review or, worse yet, before the point of integration with the master branch by running the standard set of tests against each branch and the accompanying pull request. As a result, merging any pull request becomes routine.

Each pull request should be reviewed. No CI/CD pipeline can replace the requirement for new code reviews. There are instances when the change is so minor that peer review is pointless. Nonetheless, it is best to set the rule that every pull request requires a peer review and make exceptions only when necessary. Building an engineering culture of teamwork through peer code review is essential.

Conclusion

Since the inception of CI in 1991, CI/CD has evolved from a rather specialised technique to the de facto industry norm. Along with it, there has been an explosion in software artefacts that cover a seemingly limitless range of tools and technologies as a result of the combined and mutually reinforcing effects of the rise of open-source, containerization, and distributed applications.

The advantages enterprises can gain from using CI/CD pipelines, both technologically and commercially, are exciting. Implementing CI/CD pipelines will benefit your company much in the long term, even though there will undoubtedly be some initial difficulties. This is why working with a reputable solution provider like Codification may help.



Get in touch with Codification

Visit our website to learn more: www.codification.io/services

About Codification

Codification is a Cloud Native transformation consultancy, with a team of over 100 engineers, consultants and business professionals distributed across the world. We were founded in 2019 in the United Kingdom. We have grown since then to have a presence in Europe, the Middle East and Asia, serving leading multinational corporations, government institutions, global banks, and industry giants with our consultancy and expertise.

Through our experience, we have noticed that visionary leaders want to transform their organisations into technology companies to thrive in the new digital-first economy. Here, businesses want to release software faster, improve quality and build a continuous improvement culture where the best ideas win. At Codification, we establish the direction of a company's technological transformation journey and help implement new technologies and processes, resulting in a modernised digital-ready organisation.

CODIFICATION

Codification United Kingdom
The Core
Bath Lane
Newcastle upon Tyne
NE4 5TF

Phone: +44 01670 223994

Web: www.codification.io/