What do CI and CD mean?

The close relationship between continuous integration, continuous delivery and continuous deployment can sometimes be confusing, especially when combined in the cyclical process known as CI/CD. It's important to understand the differences between each approach.

Continuous integration (CI) focuses on the early stages of a software development pipeline where the code is built and undergoes initial testing. Multiple developers work on the same codebase simultaneously and make frequent commits to the code repository. Build frequency can be daily or even several times per day at some points in the project's lifecycle.

Continuous delivery (CD) picks up where CI leaves off. It focuses on the later stages of a pipeline, where a completed build is thoroughly tested, validated and *delivered* for deployment. Continuous delivery can -- but does not necessarily -- deploy a successfully tested and validated build.

The adoption of a CI/CD platform that automates testing, builds and alpha/beta/production deployment is a key indicator of DevOps maturity. However, a survey conducted in mid-2020 by GitLab — one of the leading CI/CD solutions — shows that only 38% of organizations have fully incorporated CI/CD workflows and tooling into their DevOps implementations.

How CI/CD Captures Business Value

The following is a brief overview for business stakeholders of what CI/CD can contribute to their organization:

1. **Code is easier to manage and of better quality**: Managing smaller chunks of code on an ongoing basis makes it easier and faster to isolate problems (shorter MTTR) and reduces the risk of unintended consequences in the production environment. Automated tests can fix bugs immediately, often reducing the backlog of even non-critical

issues. Also, end-users will be grateful for not being made involuntary members of your QA team.

2. **Shortens time-to-market for new products and features**: CI/CD lets the organization achieve maximum business velocity without compromising quality. Evolving end-user requirements can be addressed quickly, including on-demand releases. Both the accelerated time-to-value and the enhanced end-user satisfaction are significant competitive advantages.
3. **Creates fast feedback/failure loops:** A key CI/CD principle is that if something is going to fail, it should fail fast. With CI/CD, companies get immediate feedback from end-users on new functions and features and can respond accordingly. CI/CD also lends itself to rapid A-B testing, feature toggles and blue-green deploys of new production features prior to full release. If an update turns out to be problematic, it can be automatically rolled back with little to no downtime.
4. **Creates new communications channels within the organization:** CI/CD fosters a culture of agility and innovation throughout the entire organization. It also provides a platform for communications among developers, product managers, testers and operations admins. These teams can now collaborate seamlessly, with a sense of shared responsibility and within an environment that values responsiveness and initiative.
5. **Enhanced employee productivity and satisfaction**: Development, IT and operations teams welcome the automation of tedious repetitive tasks that often contribute to employee burnout. Thus, employees have more time for strategic initiatives that can move the business forward. Also, their enhanced productivity can reduce costs.
6. **Data-driven business decisions**: CI/CD produces a lot of valuable metrics across the entire software development life cycle, including continuous monitoring and observability data from the production environment. These metrics support data-driven business decisions regarding product roadmaps, infrastructure requirements, team performance and more.

Benefits and challenges of a CI/CD pipeline

In a CI/CD pipeline, everything is designed to happen simultaneously: Some software iterations are being coded, other iterations are being tested and others are heading for deployment. Still, CI/CD involves important tradeoffs between benefits and drawbacks.

The benefits of a CI/CD pipeline include the following:

- **Efficient software development.** Smaller iterations (steps) allow for easier and more efficient testing. The limited scope of code in each new iteration, as well as the scope to test it, makes it easier to find and fix bugs. Features are more readily evaluated for usefulness and user acceptance, and less useful features are easily adjusted or even abandoned before further development is wasted.
- **Competitive software products.** Traditional software development approaches can take months or years, and formalized specifications and requirements aren't well suited to changing user needs and expectations. CI/CD development readily adapts to new and changing requirements, which enables developers to implement changes in subsequent iterations. Products developed with CI/CD can reach market faster and with more success.
- **Freedom to fail.** CI/CD's rapid cyclicality enables developers to experiment with innovative coding styles and algorithms with far less risk than traditional software development paradigms. If an experiment doesn't work out, it probably won't ever see production and can be undone in the next rapid iteration. The potential for competitive innovation is a powerful driver for organizations to use CI/CD.
- **Better software maintenance.** Bugs can take weeks or months to fix in traditional software development, but the constant flow of a CI/CD pipeline makes it easier to address and fix bugs faster and with better confidence. The product is more stable and reliable over time.
- **Better operations support.** Regular software releases keep operations staff in tune with the software's requirements and monitoring needs. Administrators are better able to deploy software updates and handle rollbacks with fewer deployment errors and needless troubleshooting. Similarly, IT automation technologies can help speed deployments while reducing setup or configuration errors.

Despite these compelling benefits, business leaders and development teams must consider some of the potential pitfalls of CI/CD pipelines:

- **Dedication to automation.** CI/CD relies on the consistency of an established tool set and strong automation to build, test and deploy each build. This demands a serious intellectual investment to implement and manage the automation, which can involve a steep learning curve. Changes to the development process or tool set can profoundly impact the CI/CD pipeline, so CI/CD is often employed in mature and active development environments.

What is an example of a CI/CD pipeline?

There is no single *right* way to build a CI/CD pipeline. Every pipeline can use different tools and include variations or alternate pathways to support project types with different scopes and sophistication. Regardless of the approach, an ideal CI/CD pipeline should meet three fundamental goals:

- Improve the software's quality.
- Make software development faster and more agile.
- Boost confidence in software deployment to production.

Let's examine a typical CI/CD pipeline, consider the activities within each stage and note several possible tools to tackle them.

Source

Developers write code using editors or IDEs. A development team may employ several editors or IDEs to support multiple languages for different projects.

Tools: Examples of software IDEs include Atom, Cloud9 IDE, Microsoft Visual C++ or Visual Studio, PyCharm and Xcode.

The source code is typically stored in a common shared *repository*, or *repo*, where multiple developers can access and work on the codebase at the same time. Repos also generally hold other parts of the software development process, such as artifacts (of compilation and linking), libraries, executables, modules, test scripts and suites. Repos provide a comprehensive *version control system*, which ensures developers work on the latest codebase and integrate the latest components in the build process.

Tools: GitHub, based on Git, is a popular repo; other examples of repositories include GitLab, Cloudsmith Package, Docker Hub for container projects, JFrog Artifactory, NuGet and SVN.

Once a developer commits changes to the codebase, those changes are saved to the version control system in the repository, which automatically triggers a new build.

Build

The build process typically involves multiple steps: Fetch the source code components from the repo, compile code, and link libraries and other modules. The result is a simple executable file, or a more complex assembly such as a deployable container or VM. All of the artifacts involved in the build process are typically retained in the repository. If there are problems or errors in the build, the process stops and issues are reported back to the developers for remediation. Typical problems include functional errors, such as a divide-by-zero math error, or missing components -- for example, a required library or module is not present in the build manifest.

Tools: Many IDEs include build tools natively tailored to the selected programming language. Alternatively, standalone build tools include Ant, Gradle, Make, Maven, Meister, Phing and Rake. Jenkins is a popular CI engine*, but there are*

Test

While some testing is indigenous to the build process, most happens after the build is successfully completed. This complex phase involves numerous steps and goals, including the following:

- Unit testing validates new features and functions added to the build.
- Dynamic application security testing (DAST) scans the build for security flaws, such as weak passwords or missing credentials.
- Interactive application security testing (IAST) analyzes traffic and execution flow to detect security issues, including those in third-party or open source components.
- Regression testing verifies that changes or additions do not harm previous features.
- Integration testing ensures the build operates with other applications or services.
- User acceptance testing assesses whether users are able to use new features and functions as intended.
- Performance testing ensures the build operates as required under load.

Deploy

A build that successfully passes testing may be initially deployed to a test server; this is sometimes called a test deployment or pre-production deployment. A script copies a build artifact from the repo to a desired test server, then sets up dependencies and paths.

Once on a test server, the build can be configured to simulate a production environment; for instance, access to test databases and other applications may be enabled for "real-world" functional and performance evaluations. Much of this relies on

automation but may involve human testing to shake down nuances of the build. This is sometimes called an *alpha* or development release, and involves only a small base of well-informed testers and users.

Conclusion

The CI/CD process is both flexible and fast, allowing developers to improve and deliver their code in a short timeframe and simplify implementation by the operations group. The methodology used is identical for all of the different software used, with some minor differences in terminology.

It also helps to decrease the separation between the developers and the operations group, enabling greater cooperation. CI/CD can be easily managed using standard frameworks

References:

https://www.gbnews.ch/continuous-integration-recommendations-for-implementation/

https://www.techtarget.com/searchsoftwarequality/CI-CD-pipelines-explained-Everything-you-need-to-know

https://thenewstack.io/a-business-perspective-on-ci-cd-pipelines/