UNIT 4

COMPUTER SCIENCE ENGINEERING (Jawaharlal Nehru Technological University, Kakinada)

UNIT- IV

CI/CD: Introduction to Continuous Integration, Continuous Delivery and Deployment, Benefits of CI/CD, Metrics to track CICD practices

Introduction to CI/CD

CI/CD is a way of developing software in which you're able to release updates at any time in a sustainable way. When changing code is routine, development cycles are more frequent, meaningful and faster.

"CI/CD" stands for the combined practices of Continuous Integration (CI) and Continuous Delivery (CD). Continuous Integration is a prerequisite for CI/CD, and requires:

- Developers to merge their changes to the main code branch many times per day.
- Each code merge to trigger an automated code build and test sequence. Developers ideally receive results in less than 10 minutes, so that they can stay focused on their work.

The job of Continuous Integration is to produce an artifact that can be deployed. The role of automated tests in CI is to verify that the artifact for the given version of code is safe to be deployed.

In the practice of **Continuous Delivery**, code changes are also continuously deployed, although the deployments are triggered manually. If the entire process of moving code from source repository to production is fully automated, the process is called Continuous Deployment.

Continuous Integration (CI):

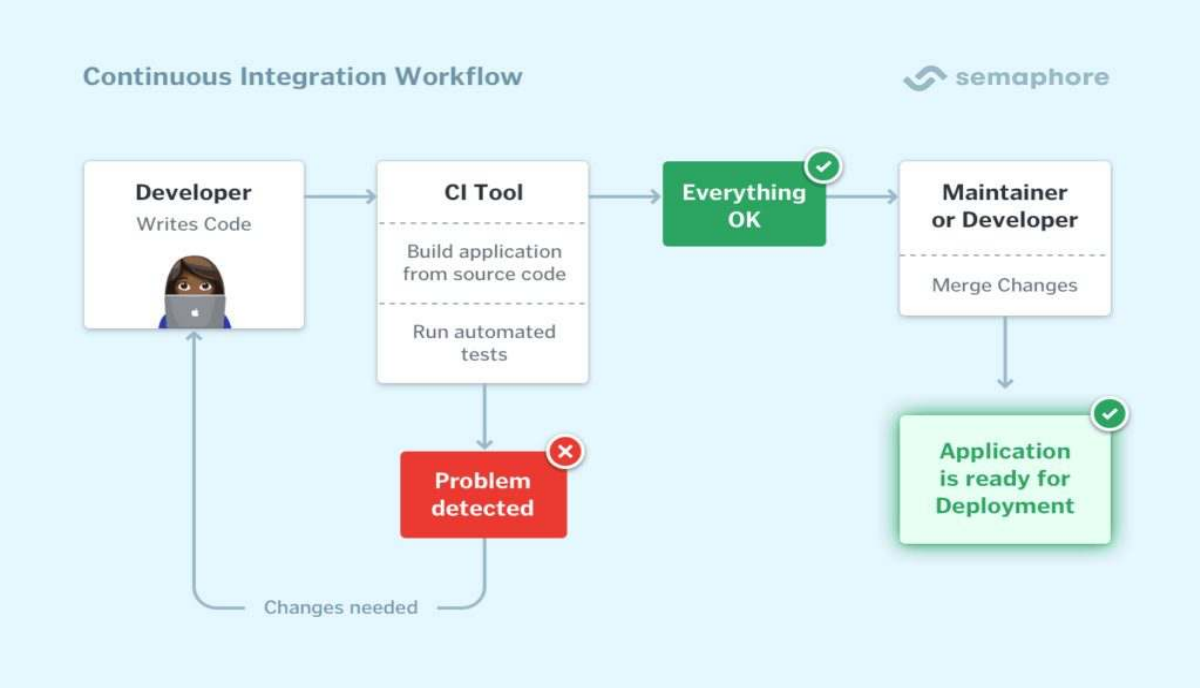
Continuous integration (CI) is a software development practice in which developers merge their changes to the main branch many times per day. Each merge triggers an automated code build and test sequence, which ideally runs in less than 10 minutes. A successful CI build may lead to further stages of continuous delivery.

If a build fails, the CI system blocks it from progressing to further stages. The team receives a report and repairs the build quickly, typically within minutes.

All competitive technology companies today practice continuous integration. By working in small iterations, the software development process becomes predictable and reliable. Developers can iteratively build new features. Product managers can bring the right products to market, faster. Developers can fix bugs quickly and usually discover them before they even reach users.

Continuous integration requires all developers who work on a project to commit to it. Results need to be transparently available to all team members and build status reported to developers when they are changing the code. In case the main code branch fails to build or pass tests, an

alert usually goes out to the entire development team who should take immediate action to get it back to a “green” state.



Why do we need Continuous Integration?

In business, especially in new product development, we often don’t have time or ability to figure everything upfront. Taking smaller steps helps us estimate more accurately and validate more frequently. A shorter feedback loop means having more iterations. And it’s the number of iterations, not the number of hours invested, that drives learning.

For software development teams, working in long feedback loops is risky, as it increases the likelihood of errors and the amount of work needed to integrate changes into a working version software.

Small, controlled changes are safe to happen often. And by automating all integration steps, developers avoid repetitive work and human error. Instead of having people decide when and how to run tests, a CI tool monitors the central code repository and runs all automated tests on every commit. Based on the total result of tests, it either accepts or rejects the code commit.

Extension with Continuous Delivery

Once we automatically build and test our software, it gets easier to release it. Thus Continuous Integration is often extended with Continuous Delivery, a process in which code changes are also automatically prepared for a release (CI/CD).

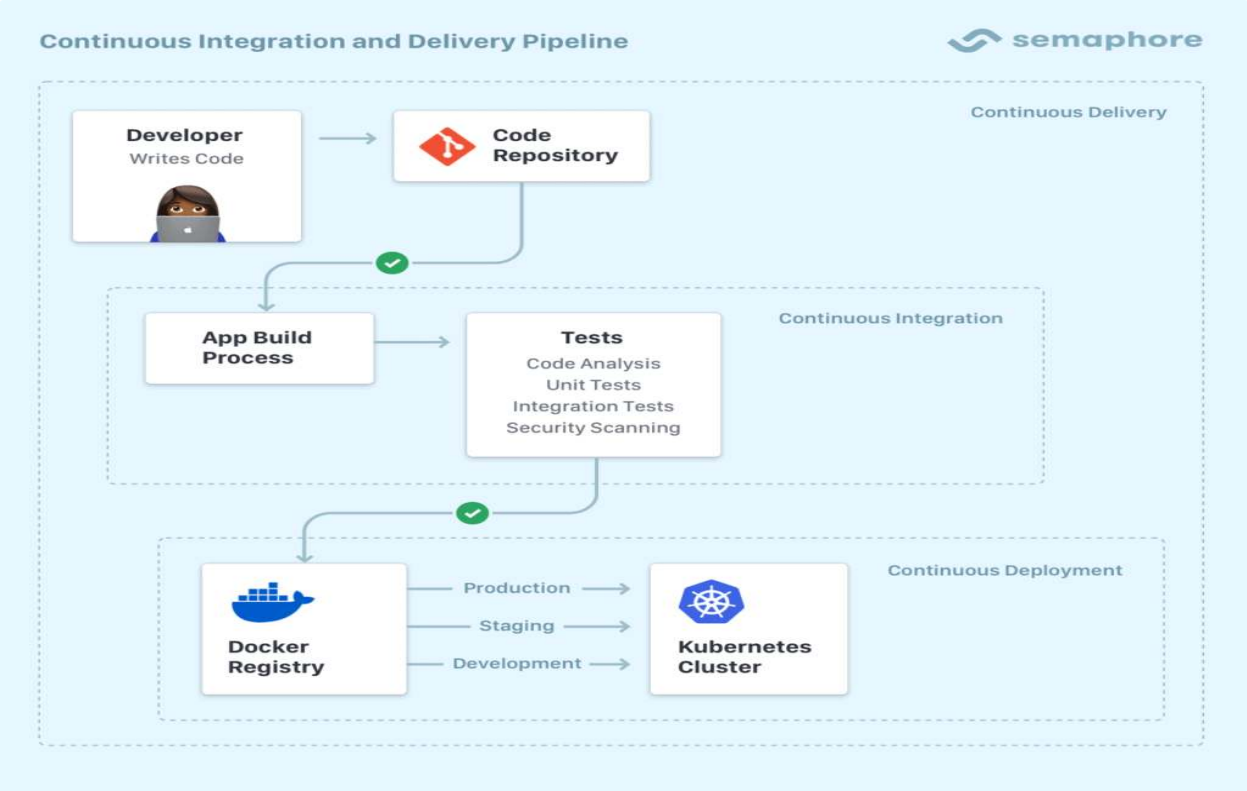
In a fine-tuned CI/CD process, all code changes are being deployed to a staging environment, a production environment, or both after the CI stage has been completed.

Continuous delivery can be a fully automated workflow. In that case, it's usually referred to as Continuous Deployment. Or, it can be partially automated with manual steps at critical points. What's common in both scenarios is that developers always have a release artifact from the CI stage that has gone through a standardized test process and is ready to be deployed.

CI and CD pipeline

CI and CD are often represented as a pipeline, where new code enters on one end, flows through a series of stages (build, test, staging, production), and published as a new production release to end users on the other end.

Each stage of the CI/CD pipeline is a logical unit in the delivery process. Developers usually divide each unit into a series of subunits that run sequentially or in parallel.



A Continuous Integration pipeline, extended with Continuous Delivery

For example, we can split testing into low-level unit tests, integration tests of system components working together, and high-level tests of the user interface.

Additionally, each stage in the pipeline acts as a gate that evaluates a certain aspect of the code. Problems detected in an early stage stop the code from progressing further through the pipeline. It doesn't make sense to run the entire pipeline if we have fundamental bugs in code to fix first. Detailed results and logs about the failure are immediately sent to the team to fix.

Because CI/CD pipelines are so integral to the development process, high performance and high availability are paramount for developer productivity.

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.

Find bugs earlier, fix them faster

The automated testing process can include many different types of checks:

- Verify code correctness;
- Validate application behavior from a customer's perspective;
- Compare coding style with industry-standard conventions;
- Test code for common security holes;
- Detect security updates in third-party dependencies;
- Measure test coverage: how much of your application's features are covered by automated tests.

Building these tests into your CI pipeline, measuring and improving the score in each is a guaranteed way to maintain a high quality of your software.

A CI tool provides instant feedback to developers on whether the new code they wrote works, or introduces bugs or regression in quality. Mistakes caught early on are the easiest to fix.

Continuous integration tools

Continuous integration is, first and foremost, a process derived from your organization's culture. No tool can make developers collaborate in small iterations, maintain an automated build process and write tests. However, having a reliable CI tool to run the process is crucial for success.

Semaphore is designed to enable your organization to build a high-performing, highly available CI process with almost unlimited scale. Semaphore provides support for popular languages on Linux and iOS, with the ability to run any Docker container that you specify.

With the advantages of tight integration with GitHub and ability to model custom pipelines that can communicate with any cloud endpoint, Semaphore is highly flexible. Embracing the serverless computing model, your CI process scales automatically with no time spent on queues. Your organization has nothing to maintain and pays based on time spent on execution.

Whatever CI tool you choose, we recommend that you pick the one which maximizes your organization's productivity. Feature-wise, your CI provider should be a few steps ahead of your current needs, so that you're certain that it can support you as you grow.

Build and testing tools

You can consider all tools used within your build and test steps as your CI value chain. This includes tools like code style and complexity analyzer, build and task automation tool, unit and acceptance testing frameworks, browser testing engine, security and performance testing tools, etc.

Choose tools which are widely adopted, are well documented and are actively maintained.

Wide adoption is reflected by a large number of package downloads (often in millions) and open source contributors. Googling common use cases returns a solid number of examples from credible sources.

Well documented tools are easy to get started with. A searchable documentation website provides information on accomplishing more complex tasks. You will often find books covering their use in depth.

Actively maintained tools have had their latest release recently, at least within the last 6 months. The last commit in source code happened less than a month ago. The best tools evolve with the ecosystem and provide timely bug fixes and security updates.

Continuous Integration best practices

The following is a brief summary of CI best practices.

Treat master build as if you're going to make a release at any time. Which implies some team-wide don'ts:

- Don't comment out failing tests. File an issue and fix them instead.
- Don't check-in on a broken build and never go home on a broken build.

Keep the build fast: up to 10 minutes. Going slower is good but doesn't enable a fast-enough feedback loop.

Parallelize tests. Start by splitting by type (eg. unit and integration), then adopt tools that can parallelize each.

Have all developers commit code to master at least 10 times per day. Avoid long-running feature branches which result in large merges. Build new features iteratively and use feature flags to hide work-in-progress from end users.

Wait for tests to pass before opening a pull request. Keep in mind that a pull request is by definition a call for another developer to review your code. Be mindful of their time.

Test in a clone of the production environment. You can define your CI environment with a Docker image, and make the CI environment fully match production. An alternative is to customize the CI environment so that bugs due to difference with production almost never happen.

Use CI to maintain your code. For example, run scheduled workflows to detect newer versions of your libraries and upgrade them.

Keep track of key metrics: total CI build time (including queue time, which your CI tool should maintain at zero) and how often your master is red.

Continuous integration metrics and KPIs

Key performance indicators (KPIs) are used in practically every industry to show data around goals. In enterprises, KPIs are often applied to business functions as well as individuals to measure performance and progress towards certain initiatives. Like any business investment, teams need to be able to show how their continuous integration solution is meeting their needs.

Continuous integration metrics can range significantly, depending on a team's priorities or even the industry. Successful CI strategies can look different from team to team, but there are metrics that can highlight potential problems or areas of opportunity for any team.

1. Cycle time

Cycle time is the speed at which a DevOps team can deliver a functional application, from the moment work begins to when it is providing value to an end user. In GitLab, we call this value stream analytics and it measures how long it takes your team to work in each stage of the developer workflow. By answering the question "How long does it take us to create something?" teams create a baseline that can then be revisited and improved upon.

2. Time to value

Once code is written, how long before it's released? While cycle time measures the process, as a whole, time to value (TTV) focuses on the release process. This delay from when code is written to running in production is a bottleneck for many organizations. Having robust continuous delivery can help to overcome this barrier to quick deployments.

3. Uptime, error rate, infrastructure costs

Uptime is one of the biggest priorities for the ops team. Uptime is simply a measure of stability and reliability – how often is everything working as it should? With the right CI/CD strategy that automates the development lifecycle, ops leaders can focus more of their time on system stability and less time on workflow issues. If uptime and error rates seem high, it can illustrate a common CI/CD challenge between dev and ops teams. Operations goals are a key indicator of process success.

4. Team retention rate

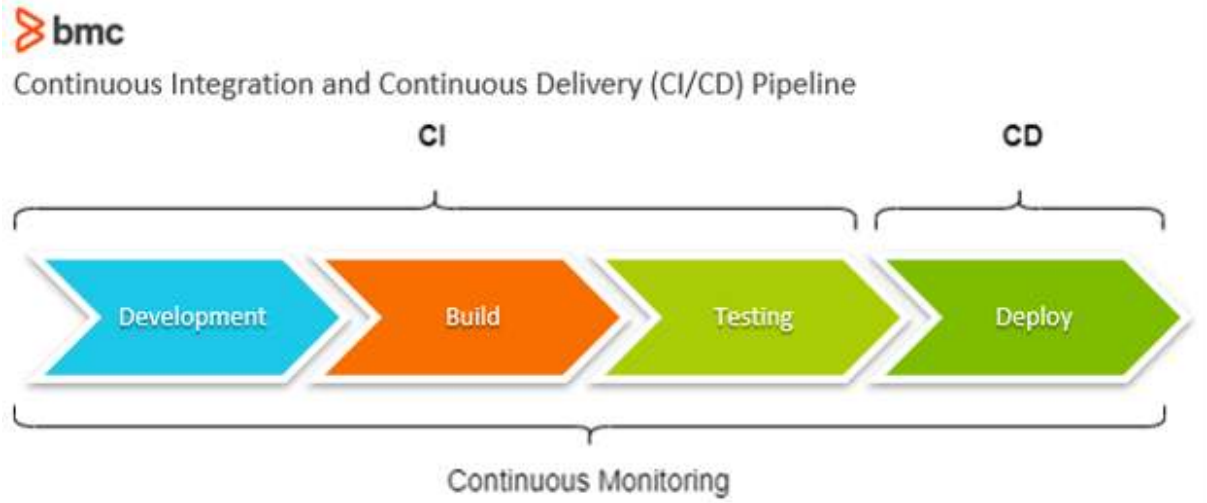
While happiness is a metric that's nearly impossible to measure, happy developers do tend to stick around. Retention rates can't measure happiness, but it can shed some light on how well processes and applications are working for the team. It might be tough for developers to speak up if they don't like how things are going, but looking at retention rates can be one step in identifying potential problems.

DevOps organizations monitor their CI/CD pipeline across three groups of metrics:

- Automation performance
- Speed
- Quality

With continuous delivery of high-quality software releases, organizations are able to respond to changing market needs faster than their competition and maintain improved end-user experiences. How can you achieve this goal?

Let's discuss some of the critical aspects of a healthy CI/CD pipeline and highlight the key metrics that must be monitored and improved to optimize CI/CD performance.



Metrics for optimizing the DevOps CI/CD pipeline

Now, let's turn to actual metrics that can help you determine how mature your DevOps pipeline is. We'll look at three areas.

Agile CI/CD Pipeline

In regard to delivering high quality software, infusing performance and security into the code from the ground up, developers should be able to write code that is QA-ready.

DevOps organizations should introduce test procedures early during the SDLC lifecycle—a practice known as shifting left—and developers should respond with quality improvements well before the build reaches production environments.
DevOps organizations can measure and optimize the performance of their CI/CD pipeline by using the following key metrics:

- **Test pass rate.** The ratio between passed test cases with the total number of test cases.
- **Number of bugs.** The number of issues that cause performance issues at a later stage.
- **Defect escape rate.** The number of issues identified in the production stage compared to the number of issues identified in pre-production.
- **Number of code branches.** Number of feature components introduced into the development project.

Automation of CI/CD & QA

Automation is the heart of DevOps and a critical component of a healthy CI/CD pipeline. However, DevOps is not solely about automation. In fact, DevOps thrives on automation adopted strategically—to replace repetitive and predictable tasks by automation solutions and scripts.
Considering the lack of skilled workforce and the scale of development tasks in a CI/CD pipeline, DevOps organizations should maximize the scope of their automation capabilities while also closely evaluating automation performance. They can do so by monitoring the following automation metrics:

- **Deployment frequency.** Measure the throughput of your DevOps pipeline. How frequently can your organization deploy by automating the QA and CI/CD processes?
- **Deployment size.** Does automation help improve your code deployment capacity?
- **Deployment success.** Do frequent deployments cause downtime and outages, or other performance and security issues?

Infrastructure Dependability

DevOps organizations are expected to improve performance *without*disrupting the business. Considering the increased dependence on automation technologies and a cultural change focused on rapid and continuous delivery cycles, DevOps organizations need consistency of performance across the SDLC pipeline.
Dependability of infrastructure underlying high performance CI/CD pipeline responsible for hundreds (at times, thousands) of delivery cycles on a daily basis is therefore critical to the success of DevOps. How do you measure the dependability of your IT infrastructure?

Here are a few metrics to get you started:

- **MTTF, MTTR, MTTD: Mean Time to Failure/Repair/Diagnose.**These metrics quantify the risk associated with potential failures and the time it takes to recover to optimal performance. Learn more about reliability calculations and metrics for infrastructure or service performance.
- **Time to value.** Another key metric is the speed of Continuous Delivery cycle release performance. It refers to the time taken before a complete written software build is released into production. The delaying duration may be caused by a number of factors, including infrastructure resources and automation capabilities available to test and process the build, as well as the governance process necessary for final release.
- **Infrastructure utilization.** Evaluate the performance of every service node, server, hardware, and virtualized IT components. This information not only describes the computational performance available for CI/CD teams but also creates vast volumes of data that can be studied for security and performance issues facing the network infrastructure.

Continuous Delivery metrics you should be tracking before you start:

1. Lead time to production

The clock starts ticking on this one at the point where enough is known about the work to begin and ends when the work is live in production. On this metric, Bowler points out that if lead time is "excessively long then we might want to track just cycle time." This is because, "When teams are first starting their journey to continuous delivery, lead times to production are often measured in months and it can be hard to get sufficient feedback with cycles that long." Measuring cycle time in the interim, or the time between when work starts on an item and that work meets the team's definition of 'done,' "can be a good intermediate measurement while we work on reducing lead time to production."

2. Number of bugs

"Shipping buggy code is bad and this should be obvious," writes Bowler, "Continuously delivering buggy code is worse." If there are quality issues with the code then continuous delivery is only going to get more bugs out into the world faster, causing more headaches later on.

3. Defect resolution time

Again, quality code is a prerequisite to successful Continuous Delivery. One way to measure quality, along with the team's commitment to quality, is to track the life of bugs. "I've seen teams that had bug lists that went on for pages and where the oldest was measured in years," Bowler observes, adding that, "Really successful teams fix bugs as fast as they appear."

4. Regression test duration

Tracking the time it takes to complete a full regression test "is important because we would like to do a full regression test prior to any production deploy." For teams with manual testing practices, this metric will be measured in "weeks or months," and "minutes or hours" for teams who primarily use automated tests. The argument that Bowler is putting forth here is that regression testing helps to reduce the risk of a failed deployment, and shortening this cycle down as much as possible will help increase the probability that continuous delivery will be successful.

5. Broken build time

"We all make mistakes," writes Bowler, but "the question is how important is it to the team to get that build fixed?" This is, again, a measure of the team's commitment to quality as a prerequisite to Continuous Delivery. If a team lets a build stay broken "for days at a time," continuous delivery won't be an achievable goal.

6. Number of code branches

Continuous Delivery advocates for main trunk development so that everything is maintained in the same pipeline. Tracking the number of branches you have to the code "now and tracking that over time will give you some indication of where you stand" in your preparedness to begin continuous delivery. Bowler adds that "if your code isn't in version control at all then stop taking measurements and just fix that one right now." No advanced practice like continuous integration or continuous delivery is possible without managing and tracking code in version control.

7. Production downtime during deployment

"Some applications such as batch processes may never require zero-downtime deploys," writes Bowler, stipulating that, "Interactive applications like web apps absolutely do." There is a cost to downtime for customer-facing or revenue-producing applications, and for those, the goal should be zero downtime during deployments. Bowler writes that "If you achieve zero-downtime deploys then stop measuring this one."