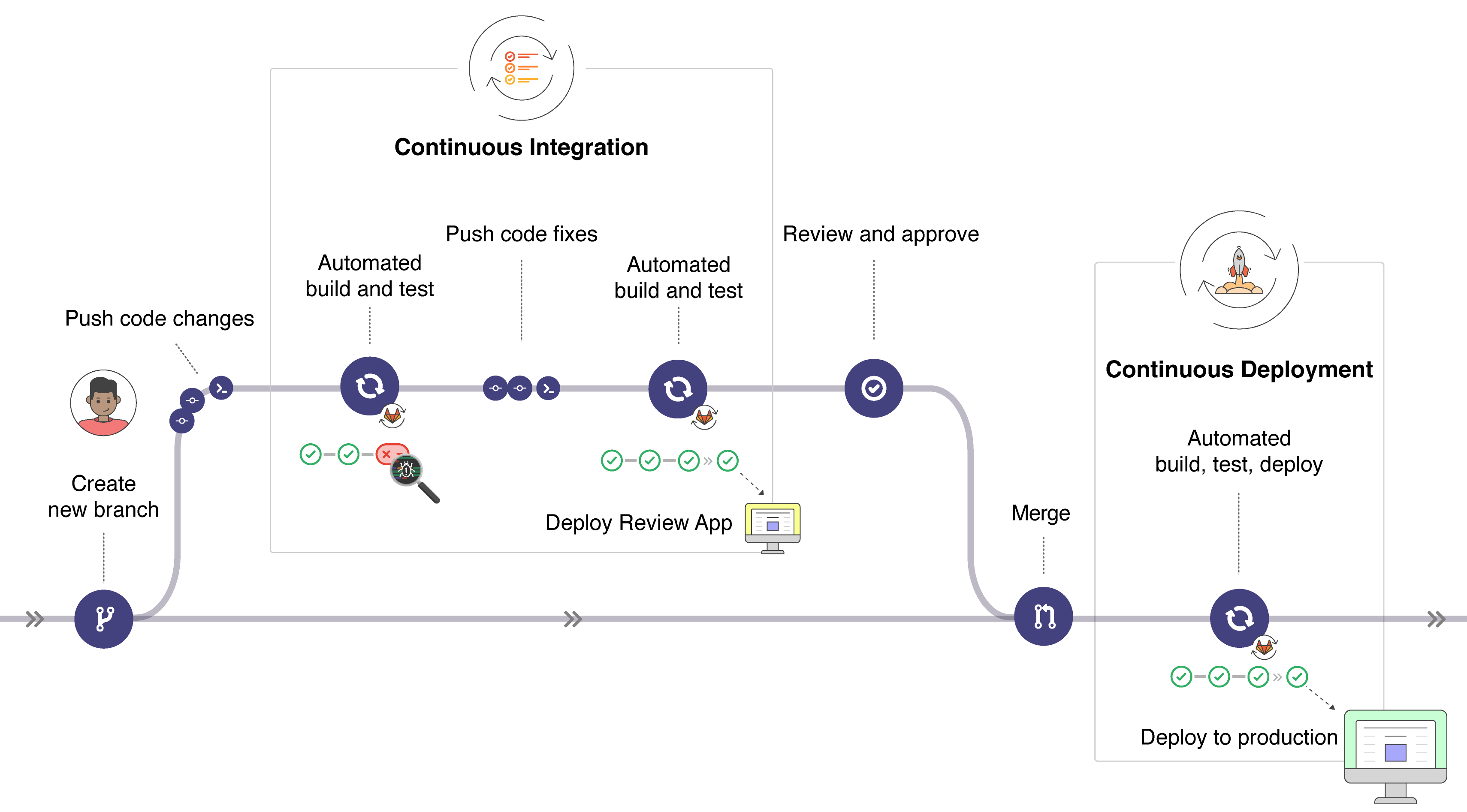
# Lecture 06 – Continuous Integration / Continuous Deployment (CI/CD)



# Basic CI/CD Workflow

Let’s take a moment and place ourselves in the prospective of a front-end developer. We’ve been working on a new feature request on our local development environment and it’s not come time to ship the change off for Quality Assurance by one of our analyst teammates.

We know that with Continuous Integration (CI) integrated into our workflow, once we trigger a push to our remote repository containing a source code change, it’s going to kick start a CI process for the work to be integrated on a remote server for testing. The CI process will handle an automated build and any custom test cases defined will execute in order to ensure integrity of our new code change. In an ideal world, the change that we qualified with our own due diligence as a developer prior to having our PR reviewed and triggering the CI process, should reflect on the QA server for an analyst to confirm that it meets the requested requirements for the feature.

Once the feature is merged into our default branch, or however the branching strategy is set up, the CI/CD process takes over. Depending on the CI/CD workflow, our change is automatically deployed to a development, QA or production server. **In a typical set up, a CI/CD process will not be automated for production deploys. The change will live on a QA server until a release is scheduled.** If something went awry, like our feature working locally but for some reason the QA server is unable to build the application, or the server starts throwing 5xx error codes, a member of your team can easily roll back the changes in event that something went wrong.

## A More Granular View of a Basic CI/CD Workflow

**With a basic CI/CD workflow, it can be broken down into 3 core stages. These stages are the ability to verify, package and eventually release** software development. Let’s break down each one of these core tenants

1. Verify:
   * After you feel that the requirements of the feature request have been met on your local development environment, a PR is made for your teammates to review
   * Once the required number of approvals are obtained, it’s considered best mannered for the submitter of the PR to make the code merge.
   * After the code change is merged by the developer, CI processes are triggered to automatically build and test the application with the newly introduced change
   * The pipeline will analyze the quality of the source code to ensure that no code conventions are broken with the change. Linters are usually used for this to ensure standards and usually can remove things like additional line breaks or adding missing things like semicolons in the case of a language with Auto Semicolon Insertion.
   * Tests are now ready to be performed. This can come in the form of Unit Tests or Code Coverage tests to ensure that the set percentage of coverage is met and not degraded by a lack of Unit Tests
   * Scanners can also be run to ensure that newly introduced dependencies do not contain any vulnerabilities
2. Package:
   * The application can be packaged with tools like Docker for containerization to meet architectural demands
   * Project dependencies can be stored in a private registry. Some companies will often host their own private registry to catalog proprietary libraries/packages that have been built. Changes or additions of dependencies are stored accordingly
   * Application artifacts can also be stored in a similar fashion as dependencies. Example; If you’re working in a Java shop, Maven is often used as a repository to manage these items
   * Bundling of static assets can be distributed to CDN’s to optimize content delivery for global traffic
3. Release:
   * We have now entered the domain of Continuous Deployment. Depending on the strategy defined by the DevOps team, the application will be deployed to its respective server. As previously mentioned, code changes usually live on a QA server for a scheduled release
   * **Manual deploys, also known as “one-click deploys”** can be leveraged for finite control of the deployment process. This can be utilized for different deployment strategies which we’ll touch on next.
   * If you’re working with a static website like a SPA, it can be deployed directly to something like GitHub pages, like we’ll be doing in our lab
   * Add release notes associated with any Git tag
   * Update and view health checks and the status of each CI environment (like QA or staging servers)
   * A/B testing release to a designated user group to test the efficacy of the newly implemented feature

## Release Strategies

When Software is in a release-ready state, meaning it’s had adequate testing perform in a production-like environment, frequent releases are made possible. This is why it’s a very strongly considered best practice to maintain a server environment which mirrors what’s on Production in a practically identical state. Consider it an unusual business practice if you wind up in a workplace that doesn’t offer QA or developers with an environment which closely resembles what’s found on their production application.

When it comes to commonly used release strategies, here are a few that are helpful to know of:

* Feature Flag deployment
  + Feature flag deploys otherwise known as “Feature Toggling” allows teams to deliver new functionality to users rapidly but safely. It provides a product to be released for manual exploratory testing with the ability for the new featured to be toggled on or off. It also allows the feature to live in a production environment where it’s seen as off (non-existent) to the general public but on (available for a specific IP subset like an internal subnet) for companies to play with. **Feature flags are used for exploratory testing completing by company staff in a Production environment without it reaching the general public.**
* Kubernetes cluster deployment
  + One nice thing about deploying a Kubernetes cluster is its ability to count on stability regardless of the environment it’s shipped to. You can deploy locally, cloud-based, to a datacenter or a managed cluster. Given then Kubernetes is an orchestration tool for Docker containers, you can count that the container will run the same in any environment. It’s one of the definite benefits of using Docker, only under rare circumstances will you find a container won’t run on another environment if adequate testing was done before deploying.
* Blue green deployment
  + Through the use of two identical production environments one referred to as blue and the other green, you drastically reduce the risk of downtime. Only one of the environments is live at any given time. Example; Blue can be currently live and green would be considered idle. A router is used to switch traffic from blue to green if something goes awry.
* Canary deployment
  + This pattern allows for a roll releases out to a subset of users or servers. The strategy is to release to a small subset, test it and if all goes well, the release is the rolled out to the rest of the servers. **Canary deployments can be used for early warning indicators with less impact on downtime**. If the deployment fails, because it was only released to a small subset, the risk is minimized.
* A/B Testing
  + A similar flavor of blue green deploys but not to have the two tastes confused. A/B testing offers a solution of testing feature specific deploys within the application for things like usability. It allows for insights to be collected on the success of the new feature or two compare two separate designs to see which one is the better performer when combined with some sort of analytics tool like Google Analytics.

## Deciding What to Automate

**It’s often beneficial to start automating from the origin of source code and working your way through the SDLC**, determining which pieces can either be scripted or handled via tooling. Reason being – all products and features start with developers requiring the ability to commit code daily. Since an engineer's role is to optimize workflow processes, we want our developers working in an environment where they can fail quickly and recover fast in order to achieve rapid product delivery. From the CI process, things like Unit tests and linting is integrated to reduce a developer's workload.

Functional testing can be automated followed by UI testing. Functional tests do not usually require frequent updates in the automation script while UI changes happen at a higher frequency which may require more attention. One principle is to think about all the possible dependencies and make an impact evaluation to priotize automation in a sensible manner.

**Fundamentally, the primary goal of a CI/CD pipeline is to automate out the process of application builds, testing and the release of software.** Using the DevOps tools, we looked at in our previous lecture, they can help simplify the process of automation. With that comes the ability to get a better sense of progress in product software through monitoring performance metrics throughout the SDLC and quickly raise alerts for rapid recoveries when something goes wrong in a release or deployment.

One thing to remain cognizant of is preventing a condition commonly referred to as “analysis paralysis” when deciding which tools to implement and what to automate. Even in organizations where there has been a strong DevOps culture and well-defined foundation, this is a common syndrome that engineers can encounter if they are working in isolation. Having a team of individuals who are tasked with the same objective of solving a problem to bounce ideas off and challenge solutions is an ideal environment to be working in. If you find your tasked with automating a process whether it’s improving a current implementation or designing a new one, start from the prospective of the developer and work your way through right till the end of the SDLC process where the product is expected to be in the hands of consumers.

## Recapping the CI/CD Process

When thinking about the process in its entirety, it can usually be grouped in three separate categories. With each category representing a different facet of the SDLC, we can draw distinctions as to which category should be responsible for what. Let’s look at each segment, get a quick high-level overview so the process is really engrained in our foundational knowledge.

Continuous Integration

This is the automation of common development tasks specifically related to a developer work and how they are using version control to get code to an environment where it can be confirmed to working as expected.

1. Application build
2. Application tests
3. Code merge

The goal here is to enable a workflow where multiple developers can work simultaneously in the same source code without stepping on each other's toes. Developers should be committing code and making PR’s often. If things are set up where merges aren’t happening frequently enough, the version control headaches can leave an individual feeling like things are in an irrecoverable state and time can be wasted at an exponential rate. Developers shouldn’t be working in isolation as if they are and not merging frequently, transparency is lost, and teammates will start feeling like distant members. This results in something known as merge conflicts, where merges can’t be made as the source you are merging does not contain the latest changes.

The intentions of CI is to provide developers with an adequate landscape where code changes can be made rapidly and with a high level of transparency so your peers know what’s being worked on and what needs doing. With CI, different levels of automated testing, typically in the form of both unit and integration tests, to ensure the changes haven’t resulted with the application left in a broken state when the build is automated. If automating testing discovers a conflict between new and existing code, CI makes it easier to fix those bugs both quickly and oftenly.

Continuous Delivery

Next in line after the CI process completes, **continuous delivery is responsible for an automated release of the validated code change to a staging/QA environment.** F**or an effective continuous delivery process in place, it’s imperative that CI is already built into the development pipeline**. Continuous delivery allows for a codebase that is always ready for deployment to a strategically chosen environment. One designed either of testing or release that ideally, mirrors the production copy of the application.

With continuous delivery, every stage from the merge of code changes to the delivery of production-ready-builds involves test and code release automation. The goal is to have a process that at its end, the ops team can deploy an app to production quickly and easily with low risk.

Continuous Deployment

The automated deployment process, providing a production-like or production environment with the newly created changes.

**Considered to be the final stage of a mature CI/CD pipeline is continuous deployment**. As an extension of continuous delivery, continuous deployment automates releasing an application to an environment for serving up the final product ready for use. Assuming that a developer’s hard work has made it this far through the pipeline without triggering any build or test failures, in theory an application could go live within minutes at this stage. At this stage of the process, all the pipeline pieces should allow for safe deployment of an application.