**Jenkins Fundamentals**

**Overview**

*Jenkins is an open source automation server that enables developers around the world to reliably build, test, and deliver their software.*

In the not-so-good old days, the Development, Quality Assurance, and Packaging/Deployment groups were often siloed and the process was more like this:

* A large group of developers coded for months
* Developers handed the "completed" code to Quality Assurance
* Quality Assurance ran tests and handed bugs back to developers to fix
* Developers fixed the bugs then handed the code back to Quality Assurance,  
  who reran the tests
* Quality Assurance handed the tested code to Packaging/Deployment,
* Packaging/Deployment released the product

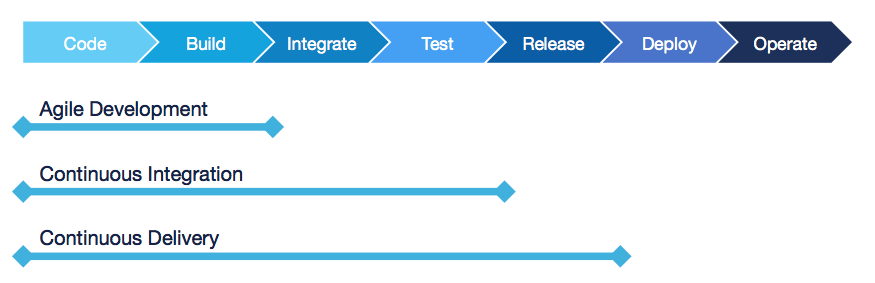
**Modern development philosophies**

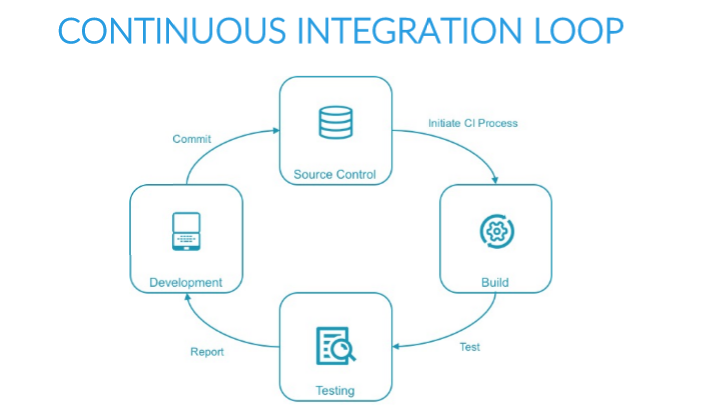
The philosophies of modern development emphasize collaboration between different teams, flexibility in planning and development, and shorter development cycles. Three major philosophies are interrelated:

* **Agile** - emphasizes adaptive planning and evolutionary development. Work is planned and completed in "sprints" (usually 1-2 weeks of work), with frequent (usually daily) "scrums" where all team members report progress and plan their next steps. See the [Agile Manifesto](http://agilemanifesto.org/).
* **DevOps** - extends the Agile philosophy into operations and production by advocating for the automation and monitoring of all steps in the development cycle. See [What is Devops?](https://theagileadmin.com/what-is-devops)
* **Continuous** - implements Agile and Devops philosophies with tools that standardize the steps in the process and thoroughly test each code modification before it is integrated into the official source.

**Agile and continuous philosophies**

Agile mostly applies to the earliest steps of the process. Continuous applies to all stages through deployment.





## **CONTINUOUS PHILOSOPHY**

* Integrate code often, at least daily, so that integration is a non-event
* Builds are triggered automatically based on **commit** and **merge** actions  
  and the success of upstream builds
* Each integration is verified by an **automated** build (including test)
* Automate the complete build-test-deploy cycle
  + Activities always run in the same order
* Build and test each code modification
  + Find problems sooner, when they are easier to fix
* *Continuous Integration does not get rid of bugs, but it does make them dramatically easier to find and remove.*

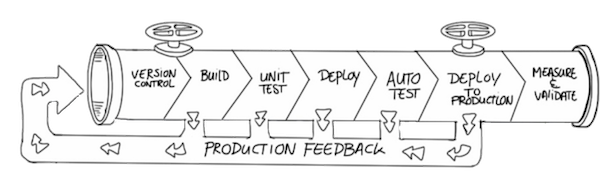
## **CONTINUOUS INTEGRATION, DELIVERY, AND DEPLOYMENT**

**Continuous Integration (CI)** is the frequent, automatic integration of code. All new and modified code is automatically tested with the master code.

**Continuous Delivery (CD)** is the natural extension of CI. It ensures that the code is always ready to be deployed, although manual approval is required to actually deploy the software to production.

**Continuous Deployment** automatically deploys all validated changes to production. Frequent feedback enables issues to be found and fixed quickly.

To successfully implement continuous delivery, it is essential to have a collaborative working relationship with everyone involved. You can then use **Delivery Pipelines**, which are automated implementations of your product’s lifecycle.



**Jenkins Workflow**

Jenkins automatically performs all the activities required to deliver your software. You specify how to build and test your software as well as when, where, and how to deploy it using these guidelines:

* Define a Jenkins Pipeline to run each activity in the same order every time.
* Pipeline is glue for the activities defined. Do not code build actions directly in the Pipeline! Instead, use shell scripts or a tool such as [Apache Maven](https://maven.apache.org/), [Gradle](https://gradle.org/), [npm](https://www.npmjs.com/get-npm), [Apache Ant](https://ant.apache.org/), or [make](https://linux.die.net/man/1/make) to define the specific actions required at each step and use the pipeline to define the execution order.
* The pipeline runs each time the code is modified.

**Continuous Integration**

* Continuous Integration, in its simplest form, involves a tool that monitors your version control system for any changes. Whenever a change is detected, this tool automatically compiles and tests your application.If something goes wrong, the tool immediately notifies the developers so that they can fix the issue immediately.
* Continuous Integration can also help you to monitor the code quality and code coverage metrics, this encourage developers to take pride in the quality of their code and strive to improve it.
* It has the potential to enable and trigger a series of incremental process improvements, going from a simple scheduled automated build right through to continuous delivery into production.
* A good CI infrastructure can streamline the development
* process right through to deployment, help detect and fix bugs faster, provide a useful project dashboard for both developers and non-developers, and ultimately, help teams deliver more real business value to the end user.
* In essence, Continuous Integration is about reducing risk by providing faster feedback. First and foremost, it is designed to help identify and fix integration and regression issues faster, resulting in smoother, quicker delivery, and fewer bugs.

**Software Testings**

Testing is a critical component of the software development cycle. Include a good set of tests as part of your pipeline so your team can:

* Validate that the software meets its goals.
* Search for defects that can be fixed to improve software quality.
* Facilitate refactoring and upgrades by validating that everything is still working after the changes are applied.

This section provides an overview of the role and structure of testing in the build cycle. It does not attempt to teach you how to create a complete and excellent set of tests for your project; if you want to learn more about good testing practices, the end of this section suggests some resources that are available.

Good testing involves many different types of tests:

* Unit testing, integration testing, smoke testing
* Functional testing
* Non-regression testing
* Acceptance testing
* Code quality and static analysis
* Performance and security testing

The build cycle should report and perhaps archive test results

## **Automated testing**

Testing should be automated as much as possible, based on the following principles and practices:

* Tests can be run frequently and always in the same order.
* Running tests frequently means that problems are found early and you usually know which small piece of code caused the problem.
* Automated tests consume machine resources but require little human time beyond what is required to review the test results.
* Tests should be independent from each other as much as possible.
* Many tests can be run in parallel—​especially tests that validate your code for different operating systems or JDK versions.
* Define different tests to run at different stages of the build chain.

## **Categories of testing**

The testing field has identified different categories of test types; you can find long discussions about the proper definitions of all these types of tests.

Test types can be categorized by how quickly they run. Faster automated test types include:

* **Unit tests** test a small piece of code (a function, method, or command). They run the fastest and are often written by the person who writes the code.
* **Integration tests** validate integration between multiple subsystems, including external subsystems such as a database.
* **Smoke tests** (also known as **sanity checking**) validate basic functions of the system.

Slower automated test types include:

* **Functional tests** validate the normal software behaviors against the expectations and requirements.
* **Non-regression tests** validate that the system still produces the same result.
* **Acceptance tests** test the full product from the perspective of the end user use cases and sentiment. These tests usually include manual testing.

Manual testing should be performed rarely, and only on software that has passed all automated tests. It is appropriate when the test result is subjective, such as user experience testing, and when the cost of automation is excessive.

## **The testing portfolio**

Jenkins enables you to run large numbers of tests frequently and at appropriate stages in the build cycle.

Your testing portfolio should have more low-level tests than high-level tests.

* Unit tests usually run every time you compile the code.
* You can define whether functional and non-regression tests run if the unit tests fail.
* Large, broad tests can be set up to run periodically (for example, during non-work hours) rather than being run each time new code is committed.

The Testing Pyramid is a visual representation of these principles:



Therefore, the following principles should be reflected in your testing portfolio:

* The low-level tests at the bottom of the pyramid run quickly and inexpensively and should be run very frequently.
* The higher-level tests at the top of the pyramid take more time to run and are expensive; they should be run less frequently and only on software that has passed all the tests that are lower on the pyramid.
* When low-level tests fail, it seldom makes sense to run higher-level tests before fixing the problems detected by the low-level tests.
* When a higher-level test fails, consider that it may have detected a defect in the lower-level tests as well as a defect in the code.
* The stackoverflow [What are unit tests, integration tests, smoke tests, and regression tests?](http://stackoverflow.com/questions/520064/what-is-unit-test-integration-test-smoke-test-regression-test) discussion introduces the types of testing you can perform.