Enterprise Systems and Architecture - CA2

Paul Good – C12397836

“Continuous Integration and Continuous Delivery - what are they, are they new, how can they be utilised by the Enterprise and to what end?”

## Continuous Integration

Continuous Integration is a software engineering approach in which a team of developers would add code to a shared space regularly. This is then checked and tested before fully added to the main code base/repo. If the code is error free it is added to the main code base. All errors are corrected and detected easily due to these small merges to the main code base. This practice isolated errors straight away.

Code can be added to the shared space or repo every week or every day or even several times per day using this approach.

Unit tests must pass before sending to the server repo. Integration tests are usually performed on the CI server.

In the early 90s, the software development process was hugely different. A large software project would use some form of Waterfall or Big Bang approach. A large amount of time would go into planning to an extremely low level and these plans were followed vigorously. Upon arriving at the integration stage everything would be set in stone in the detailed plans and requirements could not be modified or changed. Integration would be a long and unpredictable process and could take months to years to complete. Integration was a huge and complicated process for any software project. [1]

With continuous integration, this process is changed and improved drastically. It is must less risky when it comes to the actual integration as it consists of integrating already implemented and tested mini versions of the systems or modules in incremental steps. [2] This splits the project into several builds which are integrated individually reducing the integration and testing overhead for the project. These increments can be planned and deadlines can be put in place and development teams can be split into sub teams to work on individual components for the system, and upon completions of a module, it can be integrated into the full system and the final product of the overall project can be clear and visualized after each increment. Developers can also identify often complex integration risks easily in the project lifecycle. “*Java programmers have found that keeping a continuously integrated system using full specifications and stubbed bodies decreases the integration time and costs dramatically*”. [2]

Martin Fowler is said to have coined the term [3] continuous integration as he worked for ThoughtWorks in the late 90s. Fowler credits Dave Farley and Jez Humble as well as other colleagues for helping to push Continuous Integration out to many organizations and helping to cement it as a common practice in the world of software engineering. Fowler also mentions Ken Beck as a predominant role in the development and inception of the term. [4]

The origin of continuous integration can be found in the Extreme Programming world and came about as a result of clear issues with integration testing and the overhead it incurred on projects with regards to time and as a result, money spent on the project during the integration stage. Extreme programming is said to be a precursor to continuous integration and was used to improve software quality while remaining flexible to changing user requirements.

There are numerous things needed to implement Continuous Integration (CI) into an Enterprise Environment. In order to achieve CI the developers must have an automated build process, some kind of automated testing suite, a dedicated source code repository and a continuous build server.

To achieve automated deployment an automated build script sends source code from the repository to the dedicated build server and this is then sent to the development team, an integration subsystem and a staging area to be prepared for actual production.

CI tools manage and provide the solutions required for CI mentioned above. There are a large number of CI tools and servers available to organizations such as Jenkins, Drone.io, Travis CI, Bamboo, CircleCi, GitLab, CodeShip, Go, and TeamCity to name a few. These tools provide a variety of tools to improve the effectiveness of such a CI solution and improve the entire process such as Build Triggers, Source Code Management tools, Build Tools, Build Wrappers, Build Notifiers and Build Reports.

The process of a CI system is as follows. Changes are made to code and sent to the shared repo upon completion of unit tests. This code is submitted to a subversion system and this triggers a build to run on the CI server. The CI server can then send a report of issues back to the developer to be addresses. Upon a successful build and all tests passed, the new code is integrated into the subsystem.

A build agent listens for changes to the repo. This build agent runs on a build script which can be run from a command line interface, on some web application server via a servlet container or on the CI server itself. This build agents usually handles triggering the CI system to run a build when there is a code check-in on the source code repo.

The actual build pipeline overview can be described as fast tests, slower tests, acceptance tests, code quality metrics generation, a deploy to test stage and an eventual deploy to production stage. [5]

The inner workings of a build, is a complex and vital one. The build deals with many different components of the continuous integration paradigm. The build deals with compilation in which it ensures that the code is compiled on every platform the enterprise is dealing with and for all code that is committed. It then deals with the execution of the testing in which it must ensure that the product functions as expected and it does this through repeated testing within its testing suite. The build must also provide database integration and ensure that the database and constantly updating code stays communicating and in sync with each other. The CI server automates the creation and updating of the database and test data. The build also performs code inspection ensuring the code conforms to best practices and identifying code issues immediately. As previously discussed, a key to CI is Automated Deployment in which a product can be ready to use or sent to the production stage whenever required without the developer needing to intervene. Automated deployment also solves the problem of the application or code working on one developers or consumers machine and not on another’s. This is all handled within the build and the build ensures automated deployment as part of its pipeline. The build also keeps documentation up to date and generated as much as possible to reduce the burden of documentation on the developer. It does this within living documentation which is constantly changing. It also produces the build reports and metrics for the current build. [6] [7]

This complex build pipeline fixes many of the problems of other integration solutions. As a result, CI avoids overwritten code within a project, broken APIs, problems presented within developer environments not using CI servers such as updating core packages presenting integration malformities and broken outdated and non-synchronized tests. It shows how CI avoid the long waterfall process. And these large builds ensure project visibility is global due to singular versions and accessible and calculatable due dates for these projects.

In the world of CI, the industry best practices are concerned with several key CI issues. The CI server and build script should trigger to build after every code check-in or any time a dependency changes. There should not be any commits of broken code and the developer should fix any build failures immediately. Commits to the repository should be done as soon as possible and as often as possible. The CI project should have a single source code repository. The build and test process should be automated when possible. Every developer on the project should have a set time period to commit changes such as every day. The build and build process should be kept fast to avoid project downtime and slow code compilation and feedback from the CI server. The CI server and build process should allow every developer to see what is happening and when as this is a key idea behind the CI process and practice.

## Continuous Delivery

Continuous Delivery is a software engineering approach in which software is available to be released whenever needed and production is done in short software cycles. This approach improves the reliability of a software project as it is ready to be released whenever needed as building, testing and releases are done frequently and can be developed to adapt to any changes presented to the project. This creates a low-risk, agile and adaptive software development process. Software is deployable at any stage of its lifecycle. This in turn reduces cost and time of incremental updates and changes to requirements resulting in a straightforward process which shifts to changing markets and business strategies. The code becomes in essence ‘on demand’ software and these small releases provide quick feedback. [8] [9]

Continuous Delivery (CD) is a natural progression in the software engineering practices to satisfy the move toward agile development practices and all this involves. CD has roots in Continuous Integration (CI) and uses CI to achieve its goals as CI is a core practice within CD and CD is seen as an evolution of CI. CD is, along with CI, the brain child of a team at ThoughtWorks and the term really came to life in the book Continuous Delivery by Jez Humble and Dave Farley. CD is by their definition a form of build pipeline.

CD is a practice with many applications and can be used to deliver value to a project or organizations operations much sooner than traditional methods. CD is applied and created in a sense at the deployment pipeline of a project. This pipeline has three key components which are visibility, feedback and continuous deployment. Visibility is the characteristic of being visible to all developers and members of the team during every area of the development process such as build, testing and push to production. Feedback is the characteristic of the team’s ability to lean of potential bottlenecks and problems and fix them as efficiently and effectively as possible as fast as possible. Continuous Deployment is the characteristic of being able to fully automate the entire process to release at any time to any system.

The key categories involved in CD are building, deploying, testing and reporting and are mentioned in all aspects of the CD paradigm.

There are some key guiding principles to get from CI to CD. Development follows an agile process as previously mentioned and uses agile practices such as scrum. A key to achieving CD is to start small with manageable goals and this mean that results can be easily measured and acted upon. To achieved CD in an organization new tools are required and there are operational costs to obtaining and maintaining these as CD is largely different to native traditional methodologies such as the waterfall methodology. An organization must ensure all teams and fellow sub-organizations are on board to ensure a full buy-in to this new technological approach. To achieve CD, CI must be fully implemented as a given and all tests and builds must be autonomous. Other aspects must also be autonomous from testing to environment builds to database updates. This is achieved using automation rules and triggers. The entire application must always be in a deployable state and any of its version must be deployable also at the push of a button. Best practices must be enforced for the best results and the deploy to production pipeline must be fully automated. Any difficult or frequent operations must be automated. [8] [10] [11]

CD works on the overall high-level architecture of teams interacting with a CD server which is then operating with a CD pipeline agent.

A key to any organization implementing the CD process is the Maturity Model. The Maturity Model is used by organizations to consider all aspects of CD, some of which may have been overlooked or misunderstood and this provide the organization with the ‘big picture’ of the CD process. Larger organizations may be overwhelmed with the consideration of implementing CD due to the large steps that must be taken to adopt this development paradigm. Many worries an organization may be experiencing can be due to large internal activities within a complex organizational structure involving many departments and potential obstacles. The Maturity Model highlights the key aspects needed to be considered for successfully CD implementation across the entire organization. [12] [13] [14] [15]

The structure of the Maturity Model consists of five separate levels, Base, Beginner, Intermediate, Advanced and Expert. These levels dictate to what level CD is implemented into the system. To achieve Base level, the organization must implement a build script and build machine and also have deployment scripts and some form of test automation. To achieve Beginner level, an organization must implement self-service build, conduct nightly builds and all build artifacts must be stored. It must implement self-service deploy to test scripts and automatic deploy of build with standardized deploys. There must also be significant test execution at build time. To achieve Intermediate level, an organization must have builds run on every commit, implement a dependency repository and a secured configuration. It must have self-service deploy to test and production scripts and it must be a standard process across all of its environments. There must be some analysis reports and automated functional testing. To achieve Advanced level, an organization must implement triggered builds and build clusters. It must have database deployment automation and coordinated multi-tier deploys. There must be expansive tests covering all manner of code requirements and environments. There must be security scans and risk based testing reports. To achieve Extreme level, the build must run form virtual machine snapshots with gated commits. There must be continuous deployment to production and 100% test coverage.

The four CD categories of build, deploy, measure and communicate were formed and held together with culture within the CD world. A culture is “*a pattern of shared tacit assumptions that was learned by a group as it solved its problems of external adaptation and internal integration, that has worked well enough to be considered valid and, therefore, to be taught to new members as the correct way to perceive, think, and feel in relation to those problems*”. [16]

There are many key benefits to using CD. An organization can deliver value much quicker and delivery of modules is done incrementally in a controlled, fully-tested environment. As it is incremental it is easier to fully test all of the small regular changes quickly and fix the efficiently and effectively. This is turn reduces the overall risks presented to the project and lowers the impact of change which can be daunting to an organization. Another benefit is that the entire process is automated reducing workflow and downtime. All of this combines to create a greater software standard as well as deployment quality is ensured. This process also promotes collaboration within the project lifecycle. [17]

# References

|  |  |
| --- | --- |
| [1] | M. Fowler, 25 April 2017. [Online]. Available: https://www.martinfowler.com/articles/continuousIntegration.html#FurtherReading. |
| [2] | Carnegie Mellon University, 23` April 2017. [Online]. Available: http://www.sei.cmu.edu/productlines/frame\_report/softwareSI.htm. [Accessed 23 April 2017]. |
| [3] | C. Mountford, Atlassian Developers, 12 February 2016. [Online]. Available: https://developer.atlassian.com/blog/2016/02/journey-to-ci/. [Accessed 23 April 2017]. |
| [4] | M. Fowler. [Online]. Available: https://www.martinfowler.com/books/continuousDelivery.html. [Accessed 2017 April 19]. |
| [5] | “Jenkins,” [Online]. Available: https://jenkins.io/user-handbook.pdf . [Accessed 21 April 2017]. |
| [6] | P. M. Duvall, Continuous Integration - Improving Software Quality and Reducing Risk, The Addison-Wesley Signature Series, 2007. |
| [7] | M. Fowler, 01 May 2006. [Online]. Available: http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10\_Fowler\_Continuous\_Integration.pdf. [Accessed 25 April 2017]. |
| [8] | M. Fowler, 30 May 2013. [Online]. Available: https://martinfowler.com/bliki/ContinuousDelivery.html. [Accessed 24 April 2017]. |
| [9] | ThoughtWorks, [Online]. Available: https://www.thoughtworks.com/continuous-delivery. [Accessed 22 April 2017]. |
| [10] | P. Swartout, Continuous Delivery and DevOps - A Quickstart Guide, Packt Publishing, 2014. |
| [11] | J.-P. Briend, “SlideShare,” 19 April 2017. [Online]. Available: https://www.slideshare.net/Mudak1/continuous-delivery-why-where-to-start-how-to-scale?qid=df0d7fcb-c5ef-426a-be05-6c4c6148f66c&v=&b=&from\_search=7. [Accessed 25 April 2017]. |
| [12] | Zend. [Online]. Available: https://www.mightyguides.com/wp-content/uploads/Zend\_ContinuousDelivery\_MiniBook2.pdf. [Accessed 24 April 2017]. |
| [13] | A. Rehn, T. Palmborg and P. Bostrom, “InfoQ,” 06 February 2013. [Online]. Available: https://www.infoq.com/articles/Continuous-Delivery-Maturity-Model. [Accessed 20 April 2017]. |
| [14] | Forester Research and ThoughtWorks, “Continuous Delivery: A Maturity Assessment Model,” March 2013. [Online]. Available: http://info.thoughtworks.com/rs/thoughtworks2/images/Continuous%20Delivery%20\_%20A%20Maturity%20Assessment%20ModelFINAL.pdf. [Accessed 20 April 2017]. |
| [15] | J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Addison-Wesley Signature Series (Fowler)), The Addison-Wesley Signature Series, 2010. |
| [16] | E. H. Schein, The Corporate Culture Survival Guide, Jossey-Bass, 2009. |
| [17] | P. Duvall, “Continuous Delivery Patterns and Antipatterns,” [Online]. Available: http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/lecturas/10\_Continuous\_Delivery\_Dzone\_Refcardz.pdf. [Accessed 27 April 2017]. |