Understanding CI / CD

With the buzz around Continuous Delivery (CD) these days it might seem like an antiquated notion to even speak of, let alone write about Continuous Integration (CI). However, in many organizations looking to implement CD or in the midst of an implementation (and stumbling), it has become rather clear to us that there exists a wide chasm that needs to be bridged on the foundational aspects that support an organization’s desire to adopt CD. CI is a key supporting construct whose principles are still not understood/implemented with sufficient clarity, and hence worth a revisit.

**Continuous…**

**Integration** is the process of eliciting fast, automated feedback on the *correctness* of your application every time there is a change to the code.

**Delivery** builds upon the earlier concept by providing fast, automated feedback on the*correctness* and *production readiness* of your application every time there is a change to*code, infrastructure, or configuration*.

The premise behind CD is that software is always deployable. Hmm. Sounds familiar. Anyone glance at the principles behind the Agile Manifesto lately? The very first principle states:

*“Our highest priority is to satisfy the customer – through early and****continuous delivery****– of valuable software.”*

It seems CD is less a newfangled notion and more about realizing a long-standing promise. And, the key prerequisites toward realizing this ideal are:

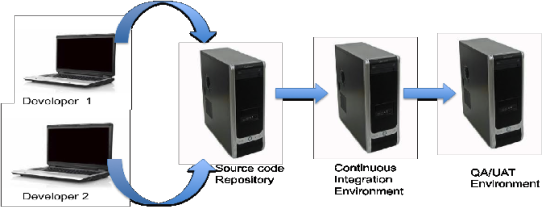
* Continuous Integration
* Comprehensive Configuration Management
* Multi-level Automated Testing

The rest of the post delves a little deeper into the CI aspect of CD.

**Continuous Integration**

CI is the process of executing the software Build-Deploy-Test (BDT) cycle frequently with minimal manual intervention with the underlying principle as one of constant communication and feedback among team members. Martin Fowler, a well-known author and speaker on software development topics, describes it as:

*“…a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily – leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.”*

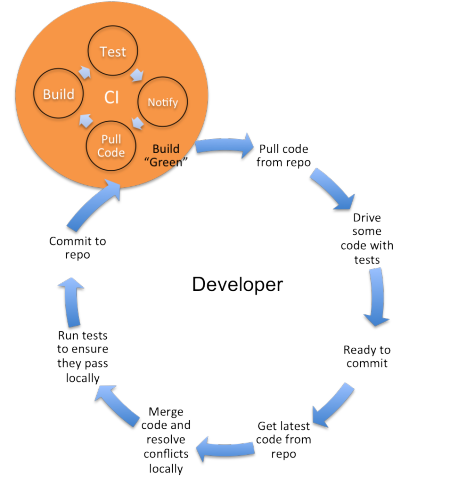
[](http://blogs.versionone.com/agile_management/wp-content/uploads/sites/3/2015/03/CI_image1-e1426983415571.png)

**If It Hurts, Bring the Pain Forward**

CI provides a framework for detecting software integration errors quickly, allowing teams to develop unified software more rapidly while ameliorating the dreaded “big bang” integrations and associated code merge misery that plague many software teams. CI aims to eliminate integration uncertainty by encouraging developers to make frequent integration of their work part of the daily development activity. It does not eliminate the challenging activity of code merge and conflict resolution – it sharply reduces the pain associated with infrequent merges by demanding small, incremental merges daily. It trades debilitating pain later for some pain today.

**How does it change what the developer does daily?**

CI is a mindset, and while the process can be executed in manual fashion, most teams use a tool to enforce the discipline. The figure below depicts the typical integration cycle within the context of using such a tool – a CI server – to help implement the practice.

[](http://blogs.versionone.com/agile_management/wp-content/uploads/sites/3/2015/03/CI_image2-e1426983470698.png)

* The day begins with the developer pulling the latest codebase from the source control repository
* Then there is the typical development activity (guided by automated unit tests preferably)
* Developer is ready to commit code into the repository and might need to communicate the intent to the team
* Updates local working copy with code from the repository (since others may have updated the code stream)
* Merges code and resolves any conflicts locally
* Builds and ensures tests pass locally
* Commits code into the repository
* The CI server “listens” for changes in the code repository. Any change triggers the build and test cycle automatically, and the results communicated to the team members in an automated fashion. A broken or “red” build (either due to compilation or test failures) is indicative of an integration failure and should be tended to immediately. Bringing the CI environment to “green” should become the top priority of the developer/team.

**Where do we begin?**

As is evident from the description of the integration cycle, the adoption of CI mandates certain practices that are necessary to make CI possible and effective.

**Single Source Repository**

All source code files, database scripts, dependent libraries, property files needed to build the deployable software artifacts need to be version controlled in some sort of a source control system since the CI server monitors changes to the code repository to trigger the BDT cycle. As rudimentary as this constraint might seem, there remain some organizations that haven’t yet reached that stage of maturity where centralized source control and version management is a given.

**Build Automation**

Even with centralized source control in place, many organizations still rely upon manual processes to creating deployable software artifacts that involve coordination across organizational silos and team members. Since the premise of CI is to kick off a build without human intervention, it necessitates the creation of a single-click build capability in order to automate the BDT cycle. The rule of thumb prescribed for build automation is to be able to stand-up a running system from scratch.

**Test Automation**

While not a precondition, test automation plays a key role in providing reliable feedback to the team that the software components are not only integrated but also working correctly. Don’t let test automation get in the way of moving towards CI, but its best not to wait too long to introduce automated tests into the mix to truly harness the benefits of the practice.

**What are some of the “best practices”?**

* Frequent commits to a common code stream
* Disallow commits into a “broken” build
* A “broken” build on CI should be attended to immediately and its resolution should be of utmost priority
* Address long-running builds. 10-15 minutes is ideal; more than 20-30 min is the outside limit for meaningful frequent integration. The shorter the cycle the better. The most likely culprit could be integration tests masquerading as unit tests. Stage the builds if necessary.
* Rebuild the database (build from zero)
* Build/deploy/test in production-like platform
* Provide capability to QA to deploy targeted builds to higher-level environments

## Top Continuous Integration Tools to Build Scalable CI

**Jenkins:**It is developed in Java and can be installed using the simple java command:  
–jar Jenkins.war. It is mainly used for building and testing software projects continuously and monitoring externally run jobs. Jenkins supports a set of SCM tools like Git, Mercurial, Subversion, etc. You can also build through Apache Ant and Apache Maven based projects.

**Buildbot:**It is built based on twisted framework and a Python configuration script that needs to be provided to the master for Buildbot configuration. It is highly useful for mixed language applications or complex release tasks and allows installation to grow with your requirements.

**Travis CI:**This is one of the simplest CI server to get started, and is available as open source and free to host on your server. It is easy to link your Github account after providing relevant permissions.

**Go:**It helps distribute builds across different systems and monitor them all at one place. It is easy to perform regular tasks as they are added as pipelines.

**Strider:**It is written in Java and JavaScript and uses MongoDB as backing store. Strider is extremely customizable through plugins.

**What are some of the misconceptions?**

**CI is the Same as the Nightly Build**

Not true. A nightly build produces software artifacts for external consumption i.e., QA functional testing, product review, etc. XP took the notion of a nightly build and pushed it to an extreme by suggesting frequent intra-day code synchronization check points as a means for a developer to converse with the system to ensure that he/she has done his/her part, at least for the time being. It is not to say that there isn’t value in the nightly build, and in fact CI could be used to generate it in a staged manner.

**We Are Not Agile, So We Don’t Do CI**

While the term “Continuous Integration” was introduced by Extreme Programming (XP) the underlying concept of more frequent integration predates XP. In fact, CI is one of the least controversial of the practices espoused in XP and its guiding principles are relevant whether an organization has embraced agile or not.

**Frequent Deployment Disruptive**

Adoption of CI doesn’t necessarily mean an automatic deployment of each every “green” build to higher-level environments whether requested or not. A good CI implementation should provide the capability such that QA or other team members can deploy targeted builds on the CI server into other environments as needed.

**Conclusion**: The greatest benefit of CI is that of reduced risk. Too often the issues with delayed and infrequent integration manifest during the latter part of the project, when the stakes are high and pressure to deliver is greatest. Further, it paves the way for automation at many levels, starting with build automation, test automation and more recently continuous delivery/continuous deployment, all of which aim to foster the creation of higher quality software, faster.

These are some of the common and less common problems that I have come across:

* Huge core dumps filling up a partition.
* A temporary folder filling up with hundreds of thousands of zero byte files.
* Opening a folder with Windows Explorer which contained hundreds of thousands of files. CPU jumped to 100% and the server became almost unusable.
* A folder containing debug information filling up with over 32,000 folders containing debug information.
* A log monitor sending huge amounts of log data over the network to a log server.
* A tcp port monitor bringing down an application which didn't like it's ports probed outside of the client application.
* A RAID drive that failed but had no email address set in the notification tool. Only discovered after the mirrored drive also failed.
* A monitoring program failed because it was sensitive to the clock being adjusted backward in time. Gave a false alert
* Internal SSH Certificates set to a one year expiry and forgotten about.
* A firewall sending a snmp trap to the monitoring server which passed through the said firewall triggering an snmp trap to the monitoring server.
* Running a script on server which deleted all files under a folder with the root folder and sub folder both specified by unset environment variables.
* A vendor setting up a service account with their own name and password. Once the vendor left and their account was deleted from LDAP/AD the Production batch stopped when it tried to authenticate with the web service.
* A user hosted the web contents folder in their home directory. It was deleted when their account was de-provisioned.
* Hardware failures (multiple disks, PSUs).  
  Contractors in the DC  
  Switch failures  
  Bad code rollouts.  
  Bad configuration rollouts  
  Cascading failures from databases going down.

IMO, the worst is when a node has had packages installed/updates applied many times only to fail to start clean at the next reboot. Ex: borked kernel package install.   
  
Designing services to be able to withstand the temporary failure of any particular node in that service and then regularly rebooting or rebuilding nodes which compose said service ("phoenix server" model) is the best way to solve that problem. I fall very much in the "run toward the pain" philosophy of infrastructure management though.  
  
The above is a large part of the excitement around containers, btw. "Screw it. We'll just deploy a new one rather than fix the old one!"

* Network Issues with respect to ISP
* DNS related Changes
* Deployment related (In case bug was not identified in Staging/Pre Prod)