[[1]](#footnote-1)

Continuous Integration: Processes and APIs (December 2016)

Kyle S. Healy

*Abstract*—This paper is to discuss continuous integration, it’s processes, tools, and APIs. Continuous integration is the process of continually developing, testing, running, and interweaving new features into software. The benefits and limitations of continuous integration will be analyzed in great detail. Continuous Integration in industry and the up and comings in academia will be described. After, we’ll look at the actual processes of continuous integration, how they were created, and the best practices currently. Continuous integration is in itself a process; Every process has tools and APIs. Some of industries current tools include Jenkins, Buildbot, Travis CI, Strider, Go, and Integrity. Each of these tools will be compared and contrasted to find the best and worst pieces about them and what they can bring to the table to help a software team develop faster, efficiently, and ultimately usable maintainable code.

*Index Terms*—APIs, Automation, BuildBot, Continuous Delivery, Continuous Deployment, Continuous Integration, Jenkins, Nexus, Parallelization, Processes, Python, Ruby

# INTRODUCTION

C

ontinuous integration is not a new concept to the software industry, but it is quickly being adopted by a lot of programs. Continuous integration is a concept to get software developers to consistently commit code to the central repository for building and testing. Amazon Web Service describes continuous integration as “a DevOps software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run” [1]. ThoughtWorks® states “Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. Each check-in is then verified by an automated build, allowing teams to detect problems early” [2]. As the multiple companies above described, continuous integration is becoming a key piece to the software development toolbox.

Continuous integration is a process with multiple different steps and APIs. Continuous Integration is used to help find bugs early and notify developers if the baseline has been broken. The process of how to set and use continuous integration in development will be discussed in detail. Continuous integration is already used in industry. There are many software companies that have already fully integrated continuous integration into their development cycle. Besides industry, academia has picked up continuous integration, and we will take a look into any new developments trending in academia as well as in industry. Next, a comparison to integration tools and APIs will be conducted. Each tool and API brings something different to the table. Having a better understanding of how these tools work and what they can and cannot do will be hugely beneficial.

# Continuous Integration: What it is?

Continuous integration is a key piece to software development cycles. Continuous integration is used much more in agile development cycles, but it is making its way into the waterfall method as well. Continuous integration is used to help software developers build and test their code before committing each and every time they commit. The build/test/commit cycle greatly increases the software developers’ effectiveness, because they can immediately know if the new code builds and passes all the current tests. This is helpful because each new commit has the possibility of breaking the current build. Most developers focus on adding a new piece of code or “feature” to the baseline. When they focus on just that feature, they may not understand what else they are changing, and having a continuous integration process that builds and tests the code on each commit will immediately let the software developer know the build is broken. This is ideal because than new code is not being built on top of code that has a bug in it.

## What is Continuous Integration

Continuous integration can be broken down into multiple steps: 1. Building new code changes, 2. Testing new changes, 3. Report results to the software developer, and 4. Committing new changes to baseline repository. These are the four basic steps of continuous integration, but this is just the high level steps. Figure 1 shows a diagram of the continuous integration cycle occurring. The diagram shows continuous integration as a cycle. A cycle is a good way to describe continuous integration because the cycle is continually iterating. Each step is extremely necessary and a building block to get to the next step. The diagram equates to our steps as follows. Step 1, building new code changes is ‘Initiate CI Process’ in the diagram. Step 2, Testing new changes is ‘Test’ in the diagram. Step 3, report results to the software developer, is ‘Report’ in the diagram. Lastly, Step 4, Committing new changes to the repository is ‘commit’ in the diagram. Now, we will go into each step in more detail.

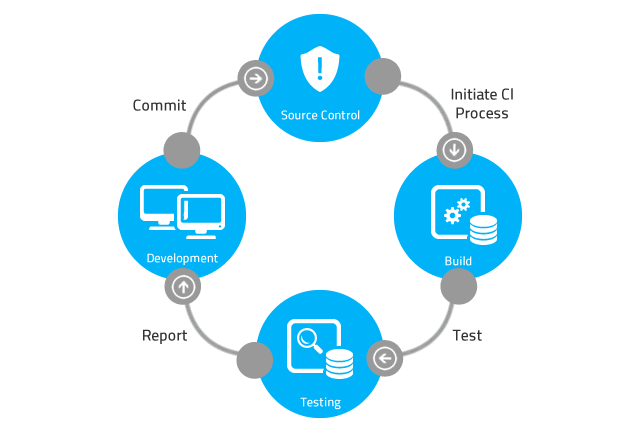


Fig. 1. Image of a basic continuous integration process [3].

### *Building new code changes*

### New code changes are the very beginning process of initiating the continuous integration process. Each developer is generally developing code separately for a different feature. This new code that the developer creates is continually locally built and tested by the developer on his or her own local machine. Once the developer is happy with the code changes, they will begin the continuous integration process with the build phase. This phase includes some changes to the developer side of the house as well. Developers must now get used to building code more consistently and committing it up to the server multiple times a day. The purpose of multiple commits a day is to keep the baseline the most up to date and make small changes. As the new smaller changes get created, they will be tested and errors can be caught earlier. Maintaining a working baseline is extremely important to continuous integration. Teams will continually make sure they are committing on a regular, daily basis. This testing will be discussed next.

### *Testing new changes*

### The testing phase of continuous integration is a baseline of tests that continually run against the code to confirm the integrity of the baseline with each new commit. The baselined tests may continually change by either growing or decreasing. The baseline tests should cover all paths that the code may handle. This is not always realistic, but it is the goal of this phase. The purpose of the testing phase is be able to test each new build over a larger variety of tests to confirm the changes are not subsequently breaking a different feature within the baseline of code. New tests need to be added to handle the new features as well. If new tests are not added consistently with new features, the tests will not cover the whole baseline. If the whole baseline isn’t covered, the ability to continually confirm that new changes don’t break the baseline is compromised. This testing phase is crucial to the continuous integration process.

### *Report results to the software developer*

### After the continuous integration software builds and tests the baseline, the software will report the results back to the development team. These results are extremely beneficial to development teams as they rely on a baseline being working when they commit. The most common way this phase is handled is to only report when the baseline breaks. This means that when something builds incorrectly or a test case fails, the software developers find out, otherwise no action is taken. The reporting procedure can occur in multiple ways, most commonly, an email is sent to the entire development team. When a development team receives this email, it is a show stopper for the team. It doesn’t make sense to commit new changes to a broken baseline, so the baseline must be fixed before new efforts can be added. The entire team receives an email because the broken code or tests are a priority, so everyone must know to not commit new code on top of it and that it needs to be fixed immediately. Most tools do have a GUI or web interface to keep track of the builds and tests that occur over time. These tools help to see trends in the baseline to see if it is stable or continually breaking. These trends are extremely helpful to teams to be able to verify the robustness of the baseline.

4) Committing new changes to baseline repository

The last step in the continuous integration process is to commit the changes to the repository. The continuous integration process tries to be a blocker between the software developer and the configuration manager when it comes to bad code commits. The continuous integration software uses the building and testing phases head off problems of poor code being committed to the configuration management repository. Even if bad code gets committed to the repository, most continuous integration tools are very good at being able to point out directly where the issue occurred and what changes were made to cause the problem. All changes to the repository happen through the continuous integration process.

## What is the Purpose of Continuous Integration?

The purpose of continuous integration is to be able to have multiple developers successfully and efficiently be able to commit code multiple times a day, have it fully tested, and report any failures if they occur. The continuous integration process is extremely useful in finding new bugs and helping mitigate the introduction of errors from new code. Each new piece of code has the possibility of introducing bugs directly in the code written, or in subsequent code that relies on the code changes.

Continuous integration tries to automate a lot of the tedious processes that are “best practices” but in theory are rarely done in a development environment. An example of these “best practices” is regression testing on the whole baseline after each new change is made. Often local testing on that specific new feature is done by the developer, and perhaps they baseline that test. That is a decent situation than, but did that developer run a full regression test on the old features of the baseline before the commit? Most likely they did not. Continuous integration works to automate that process and consistently run those regression tests on a server before the new changes are committed. Taking that testing process and automating it on a server gives the developer the chance to move forward to the next tasking without having to spend time to regression test the baseline with each new feature. The developer gets an email if something breaks, than they can jump back to the previous code and find the issue that occurred. This compartmentalizing and automation help speed up the process of development and helps build and maintain a strong, robust baseline to be deployed rapidly.

In summary, the purpose of continuous integration is to maintain the integrity of a baseline through robust, automated building and testing of a baseline before committing the changes to the configuration manager. The major change in thought process comes from the developers. Developers must work on continually committing smaller changes more often. Being able to commit changes more often minimizes the differences between the developers’ stream and the configuration management stream. Smaller changes are easier to debug and find the problems that exist.

## Current Use Cases of Continuous Integration

Continuous Integration has many use cases in the software realm now. Continuous Integration can be seen in the software industry in the agile development cycle as well as the waterfall development cycle. It is always being taught now in an academic environment and not just learned on the job anymore. The benefits of continuous integration are very strong and making the field applicable to software developers of all kinds. Below are a few break downs of where continuous integration is currently being used. It is not a complete all-inclusive list, but it does touch on some of the key areas that are currently developing within the continuous integration framework.

### *Continuous Integration Uses in Industry*

### The continuous integration framework is currently being used throughout the software industry. Continuous integration has been the push for multiple companies to help keep a stable baseline that developers can continually build off of without fear of corrupting a whole system. The examples come from the defense industry. The defense industry has many companies in it. A few will be discussed below.

### Raytheon is a defense contractor that is currently using continuous integration on multiple programs. Raytheon engineers created a next generation Air and Missile Defense Radar (AMDR) [4]. Raytheon engineers have implemented continuous integration into their agile software environment. Mike Meservey, chief product owner for AMDR, said “In one case, we found a minor issue that we recognized as a very low probability event. It was a latent defect that in a traditional development environment might have gone undetected for months or years. But with our automated test suite and continuous integration, we identified and corrected it in a matter of days.” Raytheon is saying they are using continuous integration on a daily basis to help find bugs and fix problems in there baseline. They also have programs that have moved to an agile environment. An agile environment and continuous integration work well together and push productivity and efficiency to heighten levels.

### Northrop Grumman is another company that uses continuous integration in their development environment [5] [6]. Northrop Grumman has created a specific tool called Virtual System Integration Lab or VSIL [6] that couples nicely with continuous integration. The VSIL is an integration tool to let developers test code and find faults before moving to expensive hardware. The VSIL is a tool that provides capabilities to automate tests and the VSIL adds to the continuous integration process. Northrop Grumman is working with the US government to create a “continuous delivery and continuous integration pipeline” [5] to deploy their GEOINT services. The US government has a demand to implement continuous integration, and Northrop Grumman is helping to lead that process.

### *Continuous Integration Uses in Academia*

### Continuous integration is being used in industry a lot more today. It is being used so much that the colleges and universities are starting to actually teach continuous integration. Many free open courseware classes are also teaching continuous integration to help spread the knowledge to all software developers about its benefits. CloudAcademy [7] has an online course that teaches continuous integration. It describes continuous integration as “the first step towards a completely automated development, deployment and operations pipeline” [7]. Other online courses and on campus courses are beginning to teach continuous integration. The concepts behind continuous integration are necessary for all software developers. Learning them early can greatly increase the productivity and efficiency of a new software developer in the field.

### *System set ups*

### Continuous integration can be set up multiple different ways. One way continuous integration can be set up is through a server set up. This set up includes using a server to host the repository and run all the builds and tests. This set up maximizes the efficiency of the software developer because it takes the building and testing away from the programmer and puts it on a remote server. This is how most continuous integration systems are set up. If the server is starting to back up because too many jobs are coming in, the system can be upgraded to have slave nodes as well. These slave nodes can be tasked by the master server to run builds or tests concurrently, reducing the load on the master server. These parallel builds help to decrease time in between builds and bugs and errors can be discovered faster. Finding bugs and errors faster helps to produce better code that can be delivered whenever it is needed to be.

* Another way systems can be set up is to have the work be distributed between users PCs. This case is seen less often, but if a small team doesn’t have a server, it’s a good way to start up the continuous integration process. This process has been successfully seen in industry and works when necessary. It is less reliable because computers must be left on and connected to be able to run jobs when they are queued up, but it is better than not running any type of automated build and test suite.

## Benefits of Continuous Integration

The benefits of continuous integration include automated building, consistent testing, quick reporting and easy traceability when it comes to the baseline. There are more benefits to continuous integration as well, but discussing all of the benefits would take up massive amounts of space and time, so we will stick to the few stated above.

The automated building is a major benefit to continuous integration. The ability to build a baseline while continuing to develop new code is a great time saver. Software developers can spend more time thinking creatively to find solutions to problems that exist instead of spending time in idle while they wait for the baseline to build the changes over and over again. Each build is automated and run on servers or individual machines to minimize the impact to each individual developer at their work stations.

Consistent testing is a large benefit because it is how a software developer knows if they’re code works or not. It also tells the developer that the code they wrote or changes they made didn’t break another piece of code. Having some standard regression tests helps keep the baseline in a stable state. The tests are automatically run on the server after each new build. These auto-run tests help the developers keep their code up to date and working correctly. All software should have accompanying unit tests that are performed to verify the integrity of the baseline.

Quickly reporting build or test failures back to the development team helps to fix issues significantly earlier. A stable baseline is a happy baseline. Having a reporting system that informs the whole team if the baseline breaks is essential to maintaining the integrity of the baseline. Software developers learn to not commit to a broken baseline. Adding changes to a baseline that is not working only masks the problem if the new commit is anything but a fix. Reporting failures gives the software developer the confidence to continue onto the next feature without having to wait for every single build to complete to determine of the code’s going to work.

Tracing the problem, when it exists, can become a huge hassle. Having an integration tool that can tell you exactly where the build broke or test failed is useful. Tracing the problem helps the software developer spend less time debugging and spend more time creating new additional features. Also, continuous integration helps to enable greater communication by increasing visibility of problems as they occur and displaying those problems to the end user.

Continuous integration minimizes the painful process of the long intense integrations of new features when they are added to the baseline in any other framework. If each developer is continually committing to the baseline, the changes are small and minor. Changes can be easily traced and debugging takes significantly less time. Also, having a reliable baseline adds confidence to each software developer that they are building on a strong foundation. The same is true for a robust suite of unit, integration, and regression tests.

Continuous integration is extremely cheap. There is open source free software out there for almost every language when it comes to continuous integration. It is cost effective and integration often saves time. The longer between integrations, the better chance of not catching all the bugs and fixing the problems. Larger integrations also take longer, and time directly translates into money in most cases.

## Limitations of Continuous Integration

Continuous integration isn’t a catch all solution. The build needs to be automated before continuous integration can occur. Unit tests must exist and reliably pass before being integrated into the framework. New tests need to be created and added by the user to keep the test cases up to date and completely covering the baseline. If developers never update the test cases, the new code is not consistently tests and bugs may begin to fester inside that untested code.

A major limitation of continuous integration is the culture of the software team to adopt it. Continuous integration is not nearly as effective as it could be if all it does is build and test the code. The developer needs to make smaller commits more often on a regular basis. This is a major culture change for some software developers who are used to making large commits and integrating very infrequently. Continuous integration tools also build, test and report metrics back. If software teams are not doing anything with these metrics, (ie trying to improve them) the use of a continuous integration tool and process becomes minimal. The whole software team has to embrace the continuous integration process to reap the most benefits from the process.

# Continuous Integration Processes

The continuous integration process most often occurs as seen in figure 2, although this is not the only set up for it. Usually, there are multiple developers that are all developing locally on their own PCs. Multiple times a day, preferably, they would commit those changes to the repository. The source control server than gets polled by the continuous integration server. The continuous integration server pulls in the new baseline and builds it. If there is a failure, the continuous integration server reports that failure back to the developers. If the build is successful, it begins unit tests on the new baseline. Again, if any tests fail, the continuous integration server reports a failure, which test failed, where it failed etc. back to the developer. After the unit tests all pass, the continuous integration server will attempt to run any integration tests and regression tests against the new baseline. After this point, the developers are notified regardless of a success or failure to complete. This process can be seen in figure 2.

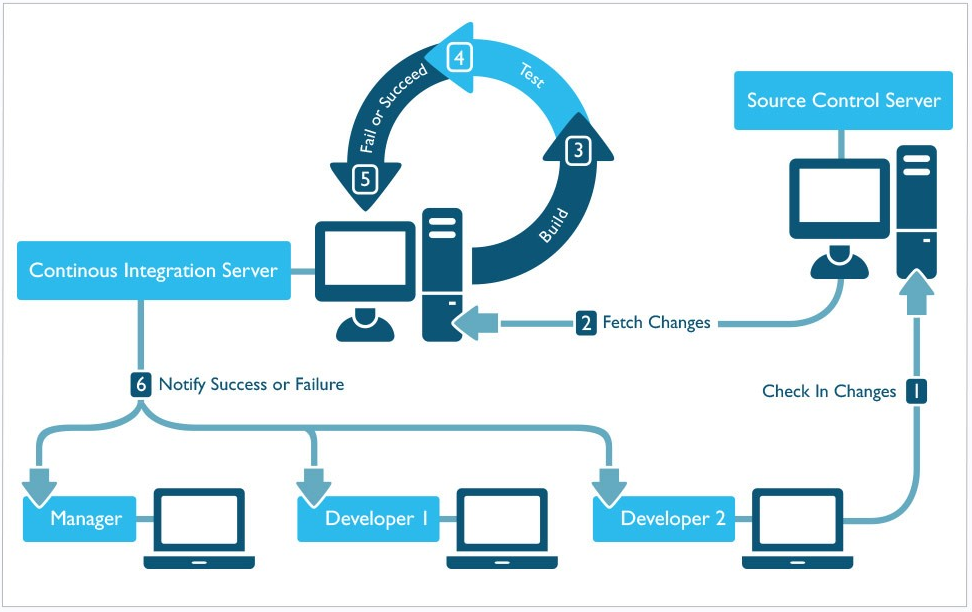


Fig 2: Continuous Integration Process [8].

# Continuous Integration Tools and APIs

There are many tools and APIs out there for continuous integration. Each tool provides a different interface and hook into the code. Each separate tool may do things slightly differently, but they all provide the continuous integration process as described above. Some tools may seem superior in certain situations, but all the tools have served a purpose and may be better in different scenarios.

Tools, such as Jenkins, Buildbot, Travis CI, Strider, Go, and Integrity, will be described and analyzed here. These tools all provide different benefits to their customers, but they also all have different limitations as well. Next we’ll take some time to explore these benefits and limitations of each tool in greater detail.

## Jenkins

Jenkins is an open source automation server that provides continuous integration solutions to its customers. Jenkins is written in Java. Jenkin’s was originally developed under the Hudson [10]. Jenkins helps developers by automating as much of the non-human part as possible. Jenkins has expanded on the continuous integration framework and now enables the development teams to implement continuous delivery. Continuous delivery is the next step from continuous integration. Continuous delivery allows software teams to build, test, and stage their software for release much faster and more often. Continuous delivery is the next step after continuous integration, but it is outside the scope of this paper.

Jenkins is cross-platform and free to download. It runs as a server-based system in a servlet container (Ex. Apache Tomcat). Jenkins supports many software configuration management tools. These software configuration management tools that Jenkins supports include AccuRev, CVS, Subversion, Git, Mercurial, Perforce, Clearcase, and RTC. Jenkins can execute shell scripts, Windows batch commands, Apache Ant and Apache Maven based projects.

Jenkins uses software configuration management tools to help it automate the build process. Jenkins can be triggered by various means. One way to trigger a Jenkins build is to commit a version to the control system. Jenkins will use a scheduling tool to poll the source control server. This polling of the source control server kicks off a Jenkins integration build, and the whole process begins.

Jenkins is set up as a continuous integration server. It polls the source control server for new changes at a set rate. Whenever a new change is found it fetches the changes and begins a full build on the new baseline. If the build passes, it begins to unit test this new build. After the unit tests pass, Jenkins will notify the developers that it successfully passed the test for the new baseline. If anything goes wrong, either in the build or the unit tests, Jenkins will immediately notify the developers as well. That is the basic life cycle of a regular commit inside Jenkins.

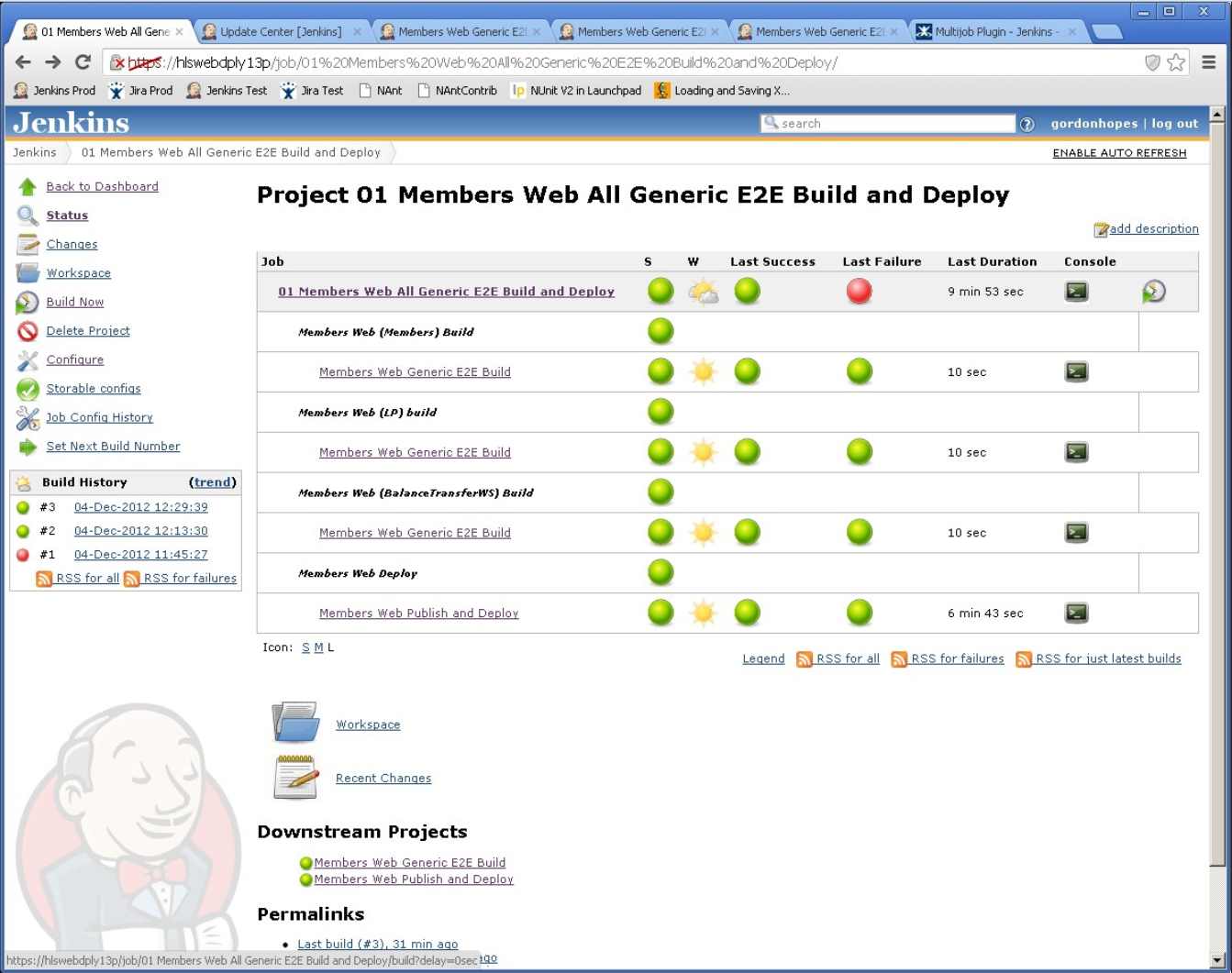


Fig. 3: Screen shot of a Jenkins home page [11]

Jenkins has the ability to run nightly builds with more in depth testing. Jenkins can run at a set time each day a full build and full test suite that doesn’t get run on every commit. These nightly builds run longer, more in depth tests on the code baseline. These tests may incorporate the whole system and stand up more pieces than the unit tests do. The purpose of the nightly builds is to take away the long testing process at each build so the build and test after individual commits is sped up. At the same time, it provides a daily full suite of tests that are run after hours to test the integrity of the baseline with the new changes that day.

On top of the underlying functionality that Jenkins provides, it also provides a GUI that is easy to use and intuitive. The GUI is updated in real time against the building projects and gives a lot of feedback to the development team. The GUI shows if the baseline is broken or intact. It also shows historical data of the baseline. It keeps track of which builds passed and which builds failed. This is extremely helpful to verify that the baseline is robust enough and the build has been stable for a while. The longer a build is stable the more reliable it is, and developers can trust it. Jenkins’ GUI also tracks which tests are run, passed, failed, and skipped. If an error occurs, Jenkins makes it easy to drill down into the broken project and find exactly where the error occurred. This speeds up debugging and helps developers find problems quicker. Jenkins runs tests consistently, automating the process so developers no longer have to run each test.

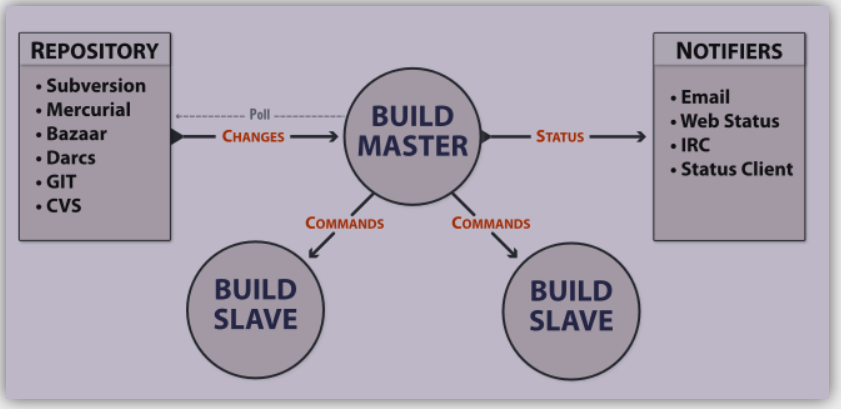


Fig. 5: Buildbot basic set up [15]

Jenkins has a lot of plugins that can be utilized to help improve a software development teams’ experience. Plugins are created by the Jenkins community and open to all. There are over 1000 plugins currently [9]. Some plugins include “Mailer” which allows the development team to configure email notifications regarding the build results. This is an extremely helpful tool when the team needs updates on who the build is doing. Other plug ins include Junit, Javadoc, Ant, SSH Slaves. Each plugin brings a different capability to the Jenkins framework and helps each user be able to customize Jenkins to their specific needs.

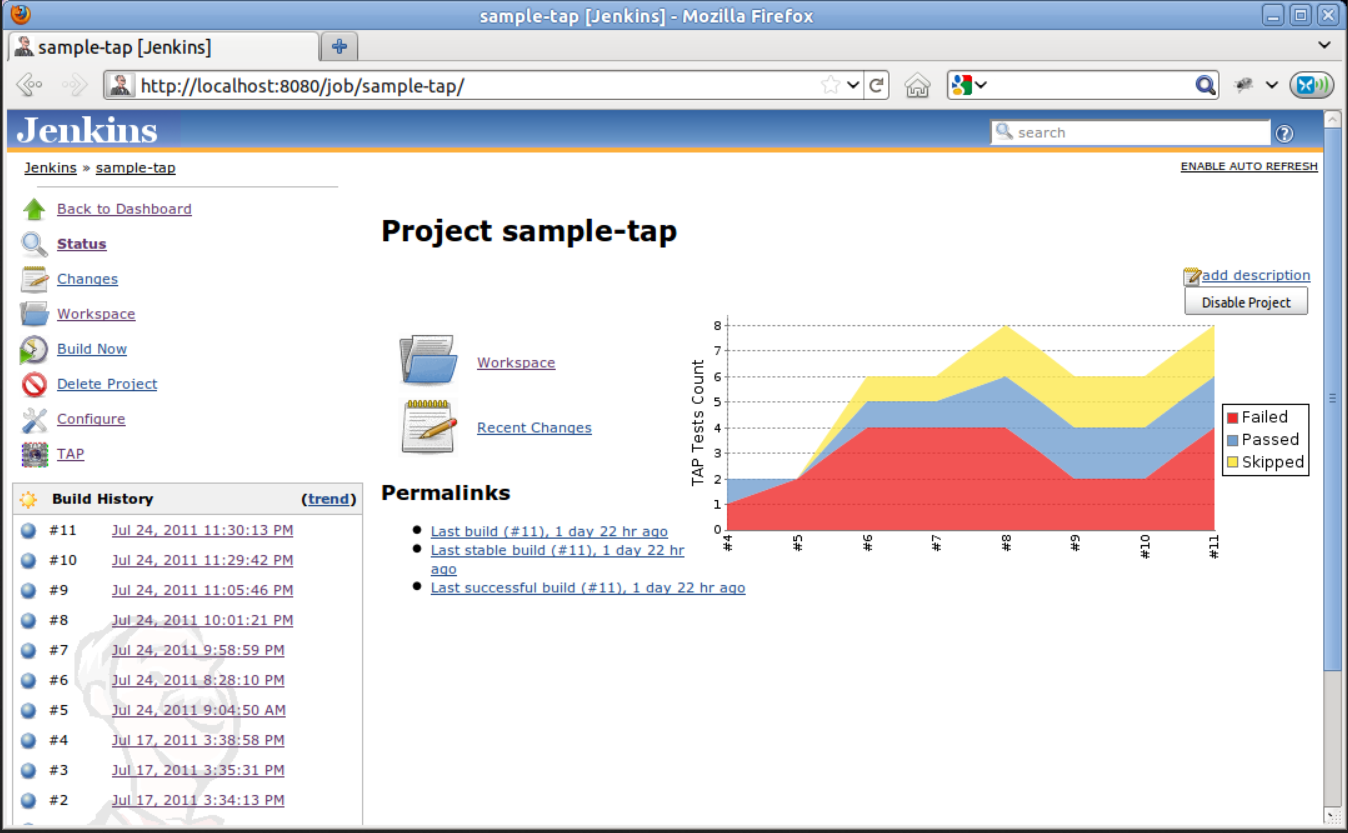


Fig. 4: Screen shot of a Jenkins project page [12]

Jenkins makes it possible to be able to develop in most main stream languages. Current supported languages by Jenkins include Java, C/C++, Python, and Ruby. Support for more languages exists. Jenkins supports other tools, such as Android Development, GitHub, Docker and much more. Each of these tools adds to the appeal of Jenkins and helps software development teams become more open to adopting Jenkins as a continuous integration tool.

## Buildbot

Buildbot is a software continuous integration tool that helps to automate the compile/build and test cycle required in each individual build. Each individual build and test cycle is used to validate the baseline of the project. Buildbot is written in python and was initially released April 29, 2003 [13].

Buildbot support multiple source control management tools. These tools include CVS, Bazaar, Darcs, SVN, Perforce, Mercurial, Git, Montotone, Repo, and BitKeeper [14].

Buildbot describes their basic functionality as follows, “ Buildbot supports distributed, parallel execution of jobs across multiple platforms, flexible integration with version-control systems, extensive status reporting, and more.” Buildbot is first and foremost a job scheduling system. It’s primary purpose is to manage resources and schedule jobs for execution when resources become available. It also reports the results of these jobs after they run.

Buildbot can have one or more masters and a collection of slave nodes. The build masters do most of the managing and coordinating. The build master polls the repository for changes and manages the resources of the build slave nodes. The build master also sends statuses back to the development team as necessary. These statuses are sent by email, web status, IRC or Status Client. The build master commands the build slaves. Each build slave can be run on a variety of operating systems to help fully test code compatibility with every operating system.

Buildbot is highly configurable. The Buildbot build master uses a python configuration script to configure the whole system. Because this script is in python, it can either be a very simple configuration file or it can take full advantage of all the libraries and built-in components python has available. The build slaves are also highly configurable. The build slaves are controlled by the build master. The build slaves are the work horses of the Buildbot system. They build the baseline, run it, and test it. They can do this in multiple operation systems and in multiple languages, depending on how they are configured. Buildbot is a continuous integration tool that is highly configurable and manages its resources extremely well.

## Travis CI

Travis CI is a continuous integration service. It is hosted and distributed. Travis CI is extremely easy to set up and is targeted at building and testing projects on GitHub [16]. Travis CI has a free version for open source projects and a fee based cost for private projects. Travis CI provides custom deployments for customers on the customer’s hardware.

Travis CI is extremely easy to hook into and use. The configuration set up for Travis CI is adding a .travis.yml file to the root directory of the repository. This configuration file details some of the set up used such as the programming language, desired build and test environment and a variety of other parameters that may be helpful. The downside is that all dependencies on the build and test environment must be installed before the software can build and/or test the baseline.

Travis CI is extremely robust when it comes to languages. Travis CI is written in Ruby but has support for a plethora of other languages including Android, C/C++/C#, Clojure, Erlang, Go, Groovy, Java, Javascript, Objective-C, Perl, Python, Ruby and a lot more [16]. Travis CI also makes it easy to deploy your code to Heroku, AWS Code Deploy, or OpenShift.

Travis CI is built to utilize GitHub as its source control management software. Travis CI, like other continuous integration tools, can poll GitHub for changes and fetch the changes, create the build and test it. Other build flow options exist as well in Travis CI. One option is that GitHub can trigger Travis CI to build after a push has been made to GitHub [16]. This is beneficial because the continuous integration server, Travis CI, doesn’t have to continually spend resources polling GitHub to find updates. It’s a small save in overhead, but it always means your testing the most up to date code. Up to date to the minute or even second. Another build flow Travis CI utilizes, which is unique, is the pull request. A pull request is created. GitHub than tells Travis CI the build is mergeable. If the build passes, Travis CI updates the pull request to say it passed and sends that back to the developer. When the developer gets the response back that everything passed, they merge in the pull request. This makes merging significantly easier on the software developer because all the heavy lifting, building and testing, are automated and done for them.

Lastly, Travis CI supports integration external tools such as static analyzers and coverage analyzers. One example is Coverity. Travis CI can build the baseline, than submit that baseline to Coverity for analysis. This is just extra support Travis CI provides to their customers.

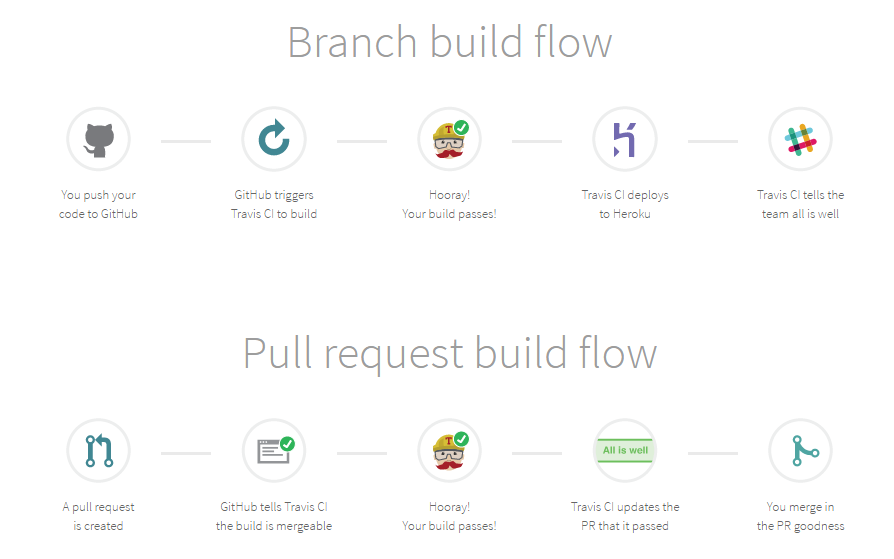


Fig. 6: Travis CI build flows [16]

# Best Practices in Continuous Integration

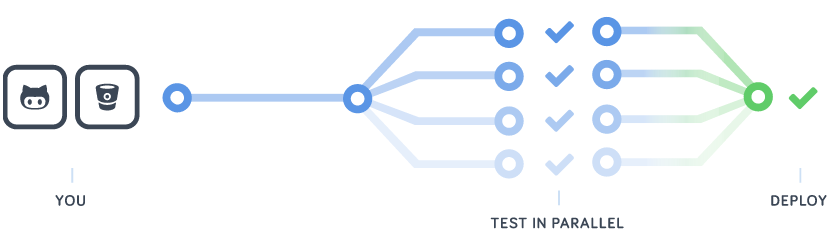


Fig. 7: Data flow of tests run in parallel [18]

Continuous integration has made its way into most major companies at this point. It is a good goal for most software development teams to shoot for. The benefits are worthwhile for a medium to large team, or a team with high turnover rate. Resource management, parallelization, and build flow all stand out as best practices and goals all cutting edge teams have in their continuous integration pipeline. [1, 9, 15, 16]

## Resource Management Inside Continuous Integration

Resource management is a huge piece of continuous integration tools. A good solid continuous integration tool will boast about its resource management. Continuous integration tools need to have a good resource management tool to be able to parallelize the build, speed up the build, manage slave nodes, and compile metrics efficiently. Resource management will make or break a continuous integration server. The overhead of running multiple builds, tests, metric collection, reports etc at once, is enough to bog down any baseline, especially a large company.

### *Build Master Node*

### The build master node is the key piece of resource management in a continuous integration tool. There can be one build master node or multiple build master nodes. The build master is required to reach out to other servers to fetch changes, or notify developers of updates. It is also required to command build slaves. These aren’t just ‘send and forget’ commands. Responses are needed to monitor the build slave nodes to the build master can know how to spread the load of the whole system out. The build master node or nodes is required to have a status on everything it can to manage the system load better and more efficiently run the whole integration process [17]. The entire process becomes more complicated when multiple build masters are involved. In this case, there is a more overhead processing power to deal with the resource management, but you are now also managing resources between your build master or resource managers. Each scenario is a little different, but exploring an option of cost vs time vs processing power is necessary to determine if one or multiple build masters are needed.

### *Build Slave Nodes*

### The build slave nodes are the work horses of the continuous integration tools. Build slave nodes take commands from the build master. The build slaves can build baselines on multiple different operating systems and in different languages [15]. Build slaves are each to configure and simple to use which makes them very diversifiable. Adding more build slaves is easy and they are just plug and play if the build master is set up right. Each build slave has no concept of other build slaves. The only thing the build slave knows is that it’s getting commands from a build master, and it needs to execute those commands.

## Parallelization

Parallelization is a concept in continuous integration that is not absolutely necessary, but massively increases productivity, efficiency and throughput if it can be integrated into the tool. Certain tools out there, such as Codeship [18], provide frameworks to parallelize tests to speed up the process. Having tests run in parallel greatly decreases the amount of time the entire continuous integration process takes. Test errors may occur in a single threaded test scenario when one test sets registers and another test reads that same register expecting a value that may have not been set in that test. When each test is run in parallel, it removes this error and each test is run start to finish on its own. This removes the ability of left over variables or memory holding values that could affect the outcome of the test.

## Fast Builds with Most Recent Changes

Builds need to run extremely fast. This isn’t just arbitrary. The concepts of continuous integration are commit often and get updates if you broke the build. Commit changes often requires a lot of building and testing. Getting that consistent feedback of something breaks is necessary to keep the baseline building and testing correctly. The more building and testing you do the longer it takes, but a more comprehensive and through testing has occurred. The automated tests that are implemented first are generally the easy ones to automate that don’t really simulate an operating environment. It’s the manual tests that are harder and longer to implement that better simulate an actual operating scenario. These tests need to be automated as soon as possible to give developers realistic feedback consistently on each build. Figure 8 shows the relationship between the length and complexity of the build and the comprehensiveness of the testing. As you can see it’s an exponential increase, therefore having a fast build is key to getting developers to actually consistently run these tests to confirm the integrity of a baseline with each build.

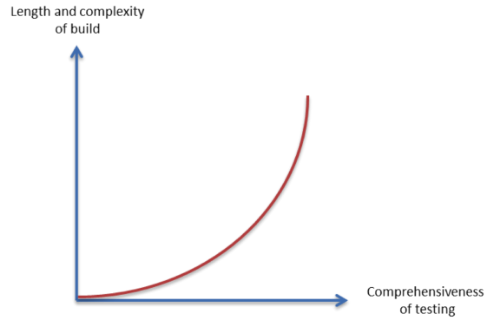


Fig. 8: Exponential growth of building vs testing [18]

On top of just speeding up the build process, automating more tests is key to a stronger robust baseline. The more in depth tests, integration and regression tests, which can be automated and added to the check in process or nightly builds, the faster bugs will be found within the code. The best way to combat this issue is to invest heavily in scripting [19]. Scripting can help development teams add more test cases, and more complex tests, to the continuous integration build. The downside to a scripting all of these tests is the maintenance needed during environment changes inside your software or while deploying to servers.

## Build Flow

Build flow is the actual flow of information through the continuous integration tool. Build flow encompasses the build, tests, overhead and the path of the data and messages that are sent throughout the continuous integration tool and all the servers and machines involved in the process. Each build flow works to build and test the baseline in one form or another. Each build moves the development team one step closer to getting a feature implemented into the baseline and out to the customer. The more efficient the build flow, the faster code gets delivered. The more comprehensive the testing, the more bugs that are caught before delivery occurs, keeping your baseline up and running more often.

### *Branch Build Flow*

### The branch build flow triggers off of a source control commit. The software developer pushes to their code to a source control repository [16]. The source control repository triggers a build within the continuous integration tool. If the build passes, the continuous integration tool delivers the build to a deployment server and stages it for deployment. Lastly, the continuous integration tool informs the team that the build was successful and that it is staged and ready for deployment. The main point of the branch build is that continuous integration cycle is being kicked off by the source control repository.

### *Pull Request Build Flow*

### The pull request build flow triggers off pull request to the source control repository server. The source control repository tells the continuous integration tool that the build is mergeable [16]. When the build passes, the continuous integration tool updates the pull request to say it passed. The user than merges in the pull request. The main purpose of this build flow is to test that pull requests are up to date and to confirm the build being pulled in is up to date and builds correctly.

The branch build flow and the pull request build flow are the two build flows that are most useful. These two build flows should be in every continuous integration tool. Not all tools include them, but most are working towards integrating both. Now the best practices are known, it’s time to move on to proposed applications for how a continuous integration system could be bettered.

# Proposed New Applications In Continuous Integration Systems

Continuous integration systems continue to expand and become more complex. Although they all aim to achieve similar results, no two integration tools are exactly the same. Each tool caters to a different need and each has different strengths and limitations. New applications can be created that maximize the current continuous integration features while implementing some changes to take better address the shortfalls within the current continuous integration tools. Proposed applications for new features to continuous integration systems include easy set up for automated systems, system easily adapts to changes, auto test generation, continuous delivery and deployment, parallelization, and updated report procedures. All of these will be touched on in the coming sections throughout the rest of this paper.

## Comprehensive Automated System Updates

Automating the system is the part of the underlying tasks of continuous integration. The tough part is really digging in and finding the most bang for your buck when deciding what to automate. The updates that would be beneficial to the continuous integration process aren’t anything completely revolutionary in this subsection, but creating a tool that could do all of them would be ground breaking and currently doesn’t exist. Currently no tool exists that can create an automated system that is easy to set up, will execute tests and deployments on demand with no manual intervention and is easily adapted to changes.

### *Easy to Setup Automated Build*

### The automated build set up needs to be easy. It needs to be almost so easy that all developers and almost anyone else can set up an automated build. This is necessary because if setting up the automated build is extremely difficult, the initial set up cost may not be worth it. As for continuous integration tools, set up should be extremely easy to get some new baseline integrated into the new tool. This set up process should be as easy as calling out the root directory to your baseline, and passing a simple configuration file with some key information broken out inside. The configuration file should consist of a simple structure with information such as programming language, and some metadata. Everything else should be automated at this point. The tool should be able to interpret the rest of the information as needed. One example of this would be the environmental variable. The continuous integration tool should be able to check some basic operating system calls to discover the operating system and fill in the environmental variables from there. There exists current continuous integration tools that exist that are easy to set up, but they don’t also contain the next few steps, making them incomplete for this portion.

### *Execute Tests and Deployments on Demand with No Manual Intervention*

### Continuous integration tools need to be able to automate a significant amount of their work, if not all of it. Every build and test should not include any user interaction after it starts. The only pieces that involve user interaction are kicking off the whole process (and sometimes that’s not even true), and where the reports based off the results go. After a build is kicked off and passes, the continuous integration tool should hand off this new built baseline to the test section and let the testing commence. Tests should go through a breadth of testing, unit tests, integration tests, regression tests and full scale end-to-end system tests. This is all on the continuous integration server though. When a developer is testing locally before committing changes, most likely they will have a few sanity check tests and the one or two “new” functionality tests. All of these tests need to be automated and not include any human interaction. One example of human interaction is if a program writes a value to a file, a human goes and opens the file and confirms the file is what was expected. That whole process can be automated by having a scripting language open the file, parse the file, and compare the value in the file versus an expected value saved off somewhere else. After all tests are automated and removed the human from them, the next piece is getting the system to do continuous deployments. This is a step a few steps past continuous integration, but it’s a step that makes sense for most software teams that don’t need full large scale releases. Continuous deployment will go through the full continuous integration process and then stage the code, test it on a DevOps or near Ops environment and then automatically deploy the code if it passes. This is a full scale end to end automated process that starts with a developer committing and ends with the new code deployed to the end user after all tests have passed. Figure 9 shows how continuous deployment expands on continuous integration and takes it a few steps farther to get the code to the customer at a much faster rate.

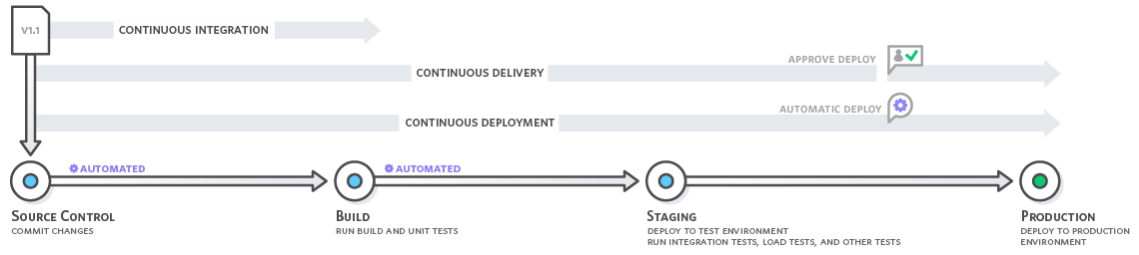


Fig. 9: Steps of Continuous Integration, Delivery, and Deployment [1]

### *Adapt to Changes Easily*

### Automating all of these builds requires build scripts. The maintenance on updating these build scripts with each new test, production, and build environment is extremely time consuming. Being able to have an integration tool that can automatically adapt to the environment and changes would greatly reduce the amount of down time due to maintenance of these scripts. Each build script is specifically designed for one operating environment with one set of configuration files. This process would become significantly smoother if an integration tool could just build these scripts on their own to utilize automation in this sense as well. Other issues that occur in this realm are build dependencies. If a build dependency is introduced into the code, the build script needs to be robust enough to change the order of the build. If a script isn’t written correctly, it could require maintenance to go in and update the build script in that case. This is more down time and that directly equates to a loss in dollars. No company wants to see that. The build script would need to be agnostic to build dependencies or be able to create them on the fly, similar to how Ant does it [20].

Creating a new tool is the only way to be able to get an easy setup, execute tests and deployments on demand with no manual intervention, and have the tool easily adapt to changes in the environment. Automated tools in the continuous integration systems today are very robust and bring a lot to the table, but no one tool covers all aspects inside this section. This is why creating a tool that does all of these could fill a whole inside the market that currently exists.

## Auto Test Generation

Continuous integration tools are just that; Tools! They are only as useful as their developers make them. This is specifically referring to the test modules that are run against the baseline. If a software developer or a systems tester doesn’t create a comprehensive test that covers edge cases, the test coverage of that new feature is minimal. Each time the code is changed or a new feature is added, a test should be updated or added to cover that code. It’s a lot of work to maintain tests on top of writing code. One would assume all software engineers and developers creates tests with every piece of code they write, but that’s not always the case. Some baselines are documented really well and have tests that cover a large portion of their baseline. Some baselines have no tests at all.

Testing is a large piece of the continuous integration system and it is essential that comprehensive tests exist, or at least that’s the goal of the baseline with time. Walking into a baseline with no tests will take some time to build them up, but it is possible. One option to building tests up is to use an auto test generation tool. One tool that does this is Parasoft [21]. Parasoft is a software testing and software quality assurance tool. Parasoft can scan code and initial unit tests and test stubs. The majority of these tests need to be updated to the correct values, but it a large amount of brute work is taken away from the developer. This doesn’t fully automate the tests perfectly 100% of the time, but it does ease the load significantly when trying to get a baseline up and running with a decent amount of tests to cover the baseline.

The next necessary piece in this auto test generation is an update to the API for the continuous integration tool. The API needs to have some type of history of the tests being run against the baseline. Being able to track trends is extremely useful and helpful when debugging code. As a developer having an understanding if something has never broken in the past 20 years and now all of the sudden did, or if it was broken 2 builds ago, are both useful information. A good API with history and trends of test data would be absolutely essential in this new continuous integration tool.

Known tests that cause problems would need to be highlighted in the API. Showing trends is not always enough. Being able to group tests together would also be beneficial to developers. If all the tests in output methods are giving problems, it would be good to know in case it’s an underlying base class that all the output classes extend from. Grouping tests and highlighting trends would be possible with some python scripting.

## Parallelization of Build and Tests

Parallelization has to occur in any future operations of continuous integration. Builds can be parallelized as well as tests. Being able to parallelize both the build and the test should have huge increases in performance and a decrease in time it takes to execute.

Parallelizing a build can greatly increase the performance of the build, but can also add complexity to the build script and an increase in overhead. Finding the right mix is key when abstracting this process out to make it common to a continuous integration tool. One way to do this would be to break apart the build into a lot of smaller projects. After they are broken apart, one option is to use Ant [20] to help work through dependency issues and be able to build the smaller projects in parallel.

Another piece of this parallelization continuous integration puzzle is the Nexus [21] framework. Sonatype’s Nexus is a repository management tool, similar to GitHub or SVN. What makes Nexus different from GitHub or SVN is that it is a package management system. Package management systems are used to store of binary executables as well. This is beneficial if you are trying to build in 3rd party dependencies. The dependency binary can be saved off in Nexus and pulled down whenever it is needed. This becomes even more useful when a program is very large and has multiple teams. As one team finishes up code and delivers it to ops, it also stores a copy on Nexus. When another team needs that new feature, instead of checking out the other baseline and building the whole baseline to gain access to that feature, a binary executable or jar file can be pulled from Nexus with the recently deployed code from the other team. This allows teams to build separate features concurrently. This allows projects to be built concurrently without dependency issues. The projects are just linked together before testing to finalize the hooks.

Being able to parallelize the build isn’t just the autonomous building of code portion, it also include the development of the software by teams. A large software company isn’t going to have every developer work the same team. The team would be too large and check-ins updates would be too cumbersome. Large software companies are broken down into development teams that work on different projects, either in the same baseline or in separate baselines. These projects have their own unit tests and integration tests internally. The company may have end-to-end tests that uses multiple projects, but that is a higher level of testing that occurs during continuous delivery.

The last piece of the continuous integration puzzle to get parallelized is the most obvious, the tests. Tests, if written correctly, are all entirely separate. They do everything from stand up the environment to instantiate the variables necessary to perform the tests. Tests should be self-contained and not reliant on any other test to perform correctly. When these tests are self-contained and modularized, they can be easily parallelized. The speed up from running tests in parallel can be orders of magnitude better, if there are a lot of tests. A master build node can distribute tests to multiple slave nodes with a working baseline to perform the tests. It is on the master build node to be able to manage all those resources and know when all the tests have returned successfully or if one failed.

Parallelization does have negative effects though. Overhead becomes more of an issue with more parallelization. One or more master build nodes need to be able to keep track of all the resources available and where every test was sent. More parallelization means more message passing and more overhead. This becomes significantly more complicated for the master build nodes. There is a tipping point in all parallelization systems that use this master/slave method where the overhead becomes too much and causes inefficiencies in the system. These inefficacies could be too many messages circulating, or lost packets of data. Finding that happy median between parallelization to speed up the build and minimizing overhead to not bog down the resource management of the master is where continuous integration will find its best performance.

## Multiple Baselines – Build and Test

Creating multiple baselines for each commit and building and testing them individually is a new aspect to this system. Builds get pulled at time intervals, so multiple check ins can get wrapped up into one build. If something breaks, it could be any of those check ins. One proposition is to create a new build after each check in. Take that new build, break it off into a slave build node and run it there. The master can than move to the next check in and begin the same process. This creates an easy history of check ins vs builds/test. If something breaks in build 4782, than we can see that the matching check in 4782 has a problem. This massively cuts down the amount of debugging involved because the chain set is minimized to that single check in. After that build passes, the continuous integration tool will remerge the check ins after that fix and build/test each check in from there on out. Another benefit/problem is that the following check in, includes the error in the broken build. If the fix comes from the second build, the continuous integration tool will report the issue and tell a developer to look into it. The development team, may have already realized the issue and checked in the fix, which they would mark as ‘OK’ or they would know a bug was created and almost hidden. Having multiple builds occur at the same time helps speed up the process of debugging and fixing errors when they occur. One down side is there is a lot of processing power going on from constant builds and tests occurring. This set up would not be for a small team, as the cost for running and maintaining these servers would be more suited for a large scale team.

Each baseline that is built will run through a range of tests. Not every single commit needs to run full integration and regression tests. If developers are making 10 to 20 commits a day, there would just be too many tests and builds going on to every fully accomplish anything productive with this system. Saying that, the continuous integration tool needs a solid API that lets developers click which tests or test suites they want to run when they check in. It can be a GUI API or even by command line. Being able to declare which tests need to be run against new code or changes to code would minimize the amount of tests that are absolutely necessary to run on each commit. That being said, hourly, or at least twice a day, a full scale build with unit tests, integration tests, regression tests and end-to-end tests need to run to verify the integrity of the overall system. Moving the culture to a “commit small changes, commit often” type strategy has ups and downs, and all of these changes may introduce a problem that isn’t foreseen at a higher level. Having these hourly or daily build/test helps bring confidence to developers and managers on how the baseline is really holding up against all of these changes.

## Pull Request Build Flow

Using a “Pull Request Build Flow” [16] is beneficial locally when pulling in new changes from the master. This pulls in the new changes, kicks off a local build and test to determine if the baseline is working. After the build passes it merges in the pull request to the local developers work station. The pull request is key to the proposed continuous integration tool because it verifies the baseline given to each developer is intact and working correctly before they start making changes. This is good because any changes that cause the build to break locally were almost always created by that developer. Pull request build flows are essential to keeping developers confident that the build worked before they got it, so it should work if they are confident in their changes. Pull requests also help ease the nerves of developers and get them into the process of working with working code more often instead of building bad habits of committing to broken builds.

## Continuous Delivery and Continuous Deployment

Continuous Delivery and Continuous Deployment are two rather large topics that take everything in this paper a step farther. They are both worth mentioning and expanding on though because both stem from continuous integration. The presented continuous integration tool would require at a minimum continuous delivery, and depending on the application hopefully continuous delivery. A high level understanding of continuous delivery and continuous deployment will be described next, followed by use cases of how they would best be implemented into this cutting edge framework being developed. Continuous Delivery and Continuous Deployment can be seen in Figure 10 on the next page.

### *Continuous Delivery*

Continuous Delivery builds on top of continuous integration. Continuous integration is the starting few steps to continuous delivery. After all the building and testing of continuous integration finish, the executable is moved to a DevOps environment. This environment mirrors an actual operation environment. Everything starts from a clean slate to avoid bugs from old memory problems. Once the executable is moved to this clean DevOps environment, a whole suite of application level automated tests kick off to begin testing the executable at a full system level integration test. End to end tests are run, load bearing tests are run to stress the system. After all the application tests are finished, the executable stays in this DevOps environment. This executable is fully ready to deploy at the click of a button [23]. It has been tested thoroughly and has the confidence of the development team and the management team that it will run smoothly when deployed. This is where continuous delivery breaks the automated cycle. All executables that get to this point are ready to be deployed, but they need someone to manually deploy them to the operational servers or states. After the executable gets deployed to production, automated deploy tests start running and the process is complete. The main take away from continuous delivery is that the automated process ends when an executable is fully built, tested, staged and ready for deployment. It just needs a human to manually deploy it to production.

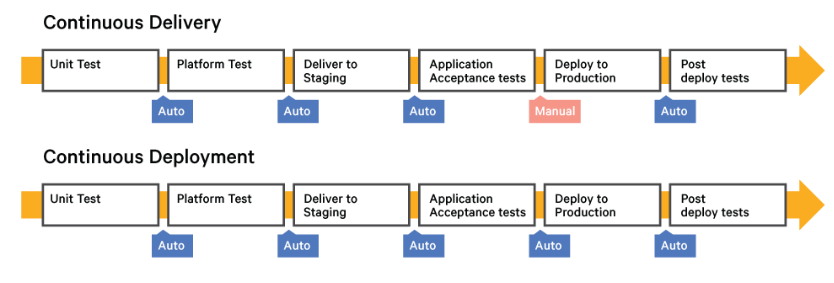


Fig. 10: Difference between Continuous Delivery and Continuous Deployment [23]

### *Continuous Deployment*

Continuous Deployment builds on continuous delivery, just like continuous delivery built on continuous integration. Continuous deployment utilizes both the continuous integration and continuous delivery steps. Continuous deployment takes full advantage of a fully automated system from start to finish. It starts with the developers commit, and if everything succeeds, ends with a new working executable or product running on the operational system. Continuous deployment takes the manual process of deploying the code to ops and automates that as well. This is what most software companies strive for, but it doesn’t always make sense to run continuous deployment. Some situations don’t warrant continuous deployment. If certain requirements need to be met and QA/QC’d by an actual person, it doesn’t make sense to automate everything. Also, if certain patches occur that IT wants to release at certain times due to 3rd party updates, having an executable staged, makes the most sense. Continuous deployment is the cutting edge, and end goal that most teams shoot for, it just takes time to get there.

Continuous delivery would be absolutely necessary for this new tool being researched. Having that ability to be able to deploy working code everyone is comfortable with is essential to how fast pace business is now moving. Continuous deployment is a good goal, but not an absolute necessity. Support for continuous deployment would be essential, but integration into every customer’s baseline is not necessary. Continuous delivery and continuous deployment both extended continuous integration. The concepts are not brand new, but the applications haven’t been worked out entirely to make them efficient and user friendly. Most tools are user friendly, or highly configurable, but the learning curve takes some time. The processes are all well-defined, and the implementation will come with the correct tools.

## Reporting Procedures of Continuous Integration Systems

Resource

### *Broken Build*

### The build master node is the key piece of resource management in a continuous integration tool. There can be if one or multiple build masters are needed.

### *Broken Unit Test*

### The build sla commands. management in a continuous integration tool. There can be if one or multiple build masters are needed.

### *Broken Integration/Regression Test*

### The build sla commands. integration tool. There can be if one or multiple build masters are needed.

### *Broken Delivery or Deployment*

### The build sla commands

This is similar to a requirement of the system having real world knowledge.

# Conclusion

## Embedded software verification and model checking for growth and increase in technology, the servers, algorithms, and computing power have caused major increases in complexity of the system. Again with information systems, a bad outcome could be a life or death situation.

References

1. Amazon Web Service (2016). What is Continuous Integration?. [Online]. Available by web page. <https://aws.amazon.com/develops/continuous-integration/>
2. ThoughtWorks (2016). “Continuous Integration” [Online]. Available by web page. <https://www.thoughtworks.com/continuous-integration>
3. Khan, Aman (August 14, 2015). The Existing challenges of Continuous Integration (CI). [Online]. Available by web page. <http://www.360logica.com/blog/2015/08/the-existing-challenges-of-continuous-integration-ci.html>
4. Raytheon Corporation (2014). Engineering on the Fast Track. Raytheon’s Agile Engineering Creates Next-Gen Radar [Online]. Available by web page. <http://www.raytheon.com/news/feature/amdr_agile.html>
5. Northrop Grumman (2016). GEOINT Symposium. [Online]. Available by web page. <http://www.northropgrumman.com/MediaResources/MediaKits/GEOINT/Pages/default.aspx>
6. Northrop Grumman (2016). Virtual System Integration Lab (VSIL). [Online]. Available by web page. <http://www.northropgrumman.com/Capabilities/VSIL/Pages/default.aspx>
7. CloudAcademy (2016). Introduction to Continuous Integration. [Online]. Available by web page. <https://cloudacademy.com/cloud-computing/introduction-to-continuous-integration-course/>
8. Cois, C. Aaron (2015, January). Continuous Integration in DevOps *Software Engineering Institute Carnegie Mellon University* [Online]. Available by web page. <https://insights.sei.cmu.edu/devops/2015/01/continuous-integration-in-devops-1.html>
9. Jenkins (2016). Jenkins “Build great things at any scale.” [Online]. Available by web page. <https://jenkins.io/>
10. Kawaguchi, Kohsuke. “Hudson” (PDF). <https://www.java.net//blog/kohsuke/archive/20070514/Hudson$20J1.pdf>
11. Wsource (2016). Jenkins Screen Shot. [Online]. Available by web page. <https://issues.jenkins-ci.org/secure/attachment/22920/screenshot-1.jpg>
12. Kinoshita, Bruno (2016, Oct). TAP Plugin. [Online]. Available by web page. <https://wiki.jenkins-ci.org/display/JENKINS/TAP+Plugin>
13. Sourceforge (2005, September). Buildbot Release 0.3.1 (29 Apr 2003). [Online]. Available by web page. <http://sourceforge.net/project/shownotes.php?release_id=155929>
14. Mitchell, Dustin J. (2014 June). Builtbot / MAINTAINERS.txt [Online]. Available by web page or git pull. <https://github.com/buildbot/buildbot/blob/v0.8.7p1/MAINTAINERS.txt>
15. Buldbot (2016). Buildbot The Continuous Integration Framework. [Online]. Available by web page. <http://buildbot.net/index.html>
16. Travis-CI (2016). Travis-CI. Test and Deploy with Confidence. [Online]. Available by web page. <https://travis-ci.org>
17. Gorbatchev, Alex (2015 Janguary). StrongLoop. Continuous Integration in the Cloud: Comparing Travis, Circle and Codeship. [Online]. Available by web page. <https://strongloop.com/strongblog/node-js-travis-circle-codeship-compare/>
18. Codeship (2016). ParallelCI: Get Faster Tests. Run Your Tests in Parallel. [Online]. Available by web page. <https://codeship.com/features/parallelci>
19. ElectricCloud (2016) Continuous Integration Best Practices: Vision and Reality. [Online]. Available by web page. <http://electric-cloud.com/plugins/continuous-integration/>
20. Apache Ant (2016) The Apache Ant Project. [Online]. Available by web page. <http://ant.apache.org/>
21. Parasoft (2016). Parasoft. [Online]. Available by web page. <https://www.parasoft.com/>
22. Nexus (2016). The Nexus [Online]. Available by web page. <http://www.sonatype.org/nexus/>
23. Caum, Carl (2013, August). Puppet. “Continuous Delivery Vs. Continuous Deployment: What’s the Diff?” [Online]. Available by blog post. <https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff>

Biography of author

1. This work was supported solely by the author. It was used for a Research in Computer Science Course at Lewis University.

   Kyle S. Healy is working this paper for Lewis University, though he works full time for Northrop Grumman Corporation, Rolling Meadows, IL 60008, USA (email: healykys@gmail.com). [↑](#footnote-ref-1)