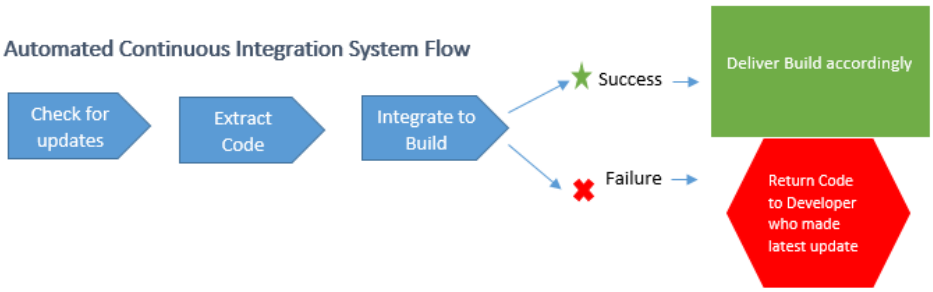
Implementing Continuous Integration

Choosing the right Continuous Integration Tools

Choosing the right **Continuous Integration** tools for your particular requirements may take some experimentation. Different tools have different attributes, so project necessities will determine which is best suited for your solution.

The **CI** approach was created to save time, and automation is crucial to attaining this goal. Each team should be checking in code frequently so it can be tested in the most recent build. The tools you choose will depend on the priorities of your current workflows, and which complexities integration will entail.



Setting up your **CI** environment will depend on the answers to some important questions:

- *Will you be using Open Source or Proprietary Software?*
- *Do your service requirements entail customization?*
- *Is security a high priority for your current and future projects?*
- *Does the tool to have the functionality to build, test, integrate, and release code?*

Complex solution setups are going to vary for each organization. Currently, these solutions can be enhanced with **CI** tools, apps, and features. Once the above questions have been answered, you can better discern which combination works best for your team.

Code Repository: which one is right for your project?

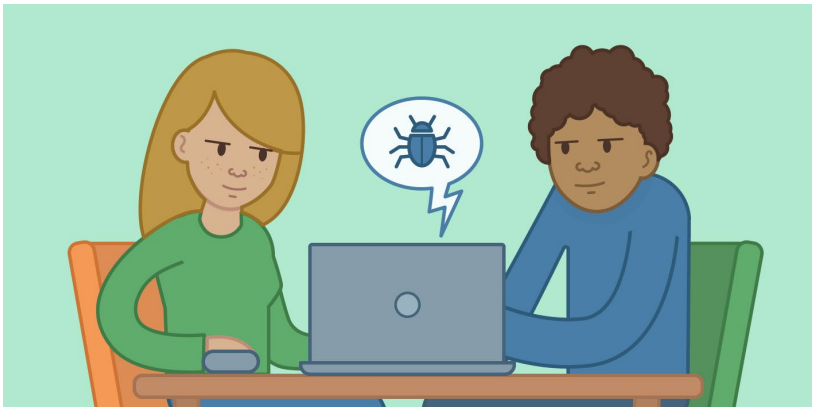
When development teams choose a repository for their environment, there are a number of different factors that will need to be considered. Requirements must be created and prioritized, and pros & cons will need to be weighed for using a hosted service, an institutional repository, or to run the infrastructure internally.

Many options exist for repositories, but the one that your organization chooses will depend on your unique circumstances. Management, integration, and functionality requirements will determine which direction to go, and with what solution.

- *How much effort will need to be exerted for services?*
- *What is the industry standard for your environments that are similar to yours?*
- *How many outside entities will be contributing to decisions that affect the development process?*
- *Does the product enable Easy Installation?*
- *Will the product work in conjunction with your future solution and integrate upgrades easily?*

It is extremely important to remember that a repository serves its purpose in the present. Provisions will need to be reviewed consistently, in case of the advent of migration to a different service. If your software project involves creators and committers that are spread across more than one institution, it is likely that you will lean toward hosted services.

For an **Agile** Development team, it will be an absolute necessity to find a host that is tailored to the most common forms of code. The team will need to prioritize features such as code review, issue trackers, patches, bug reports, and other management tools that developers find necessary when coding. So, what is the best way to ascertain the host that is going to offer the features that are necessary for your project needs?



Factors to consider when choosing a repository

- *Will your Code be publicly available?*
- *Will you be keeping your current version control system?*
- *Will code commit be local only?*
- *How easy will it be to integrate systems that are run separately?*
- *How easy will it be to upgrade additional functionality?*
- *What functionality do you need now?*
- *What additional social networking functionality will you need?*
- *Where are similar projects hosted?*

Most **Agile** development teams depend on bug & issue tracking, customized software package hosting & publishing, analytics reporting, access control, and release management. Without these assets, it is nearly impossible to competently implement **CI** workflows. Fortunately, there are many tools and services that assist developers with code maintenance to ensure that it is ready to publish upon request.

Popular CI server tools

- *Jenkins*
- *Visual Studio (Team Foundation Server*
- *Buildbot*
- *Travis CI*
- *GitLab*
- *TeamCity*

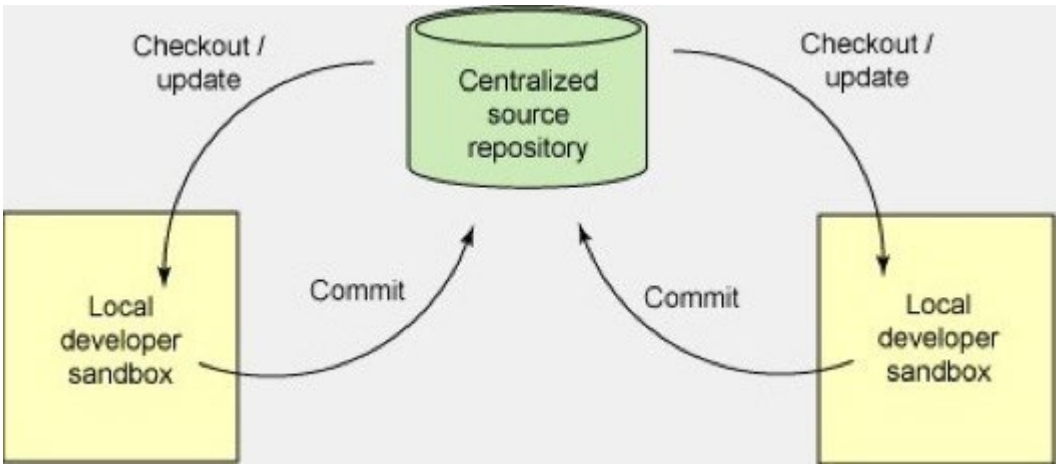
Setting up the ideal **CI** environment for your server structure requires direct communication with your developers to get critical information. Do they use a build server, and if so which brand? Do they use a deployment tool, and if not, how do they deploy new software to different environments? What changes would they make to their current system to help deploy database changes in a more efficient manner?

Continuous integration tools provide similar services but don't be fooled because the more expensive options aren't necessarily better (or worse) than open-source and low-cost tools. The difficulty of integration, the overall cost, and the final results of the solution will be the priority factors to consider when deciding which tools an organization should choose.

Creating & Managing a Source Code Repository

A **source code repository** is a file archive and web hosting facility where large amounts of source code is stored, recorded, and reported. They can be public or private and are often used by open-source projects and other multi-developer environments.

Your team's repository will enable developers to maintain a stable foundation of code through the use of a revision control system. All features, upgrades, and updates should be merged into a fresh build whenever they are checked in. Generally, it is recommended that all changes are integrated into the main branch of the build as soon as they are checked in. If the build is broken, developers can address the issues immediately, but the build will not have been corrupted by broken code. Revision control ensures that changes are rolled back if they do not integrate properly, which is critical for allowing editors to track each other's edits, correct mistakes, and collaborate on priority tasks.



Agile development teams will need to archive and organize their source code. Archiving code in a **source code repository** will decrease the average time and energy that is spent on troubleshooting errors. Small updates of code get logged as soon as they are committed, so it can be retrieved later on if necessary. When working in a collaborative environment, smaller more manageable snippets of code are extremely beneficial for teams or large groups.

Source code repository hosts are available for open source projects, single developers, private teams, or multi-developer projects. Features and tools that exist in each hosting facilities services such as mailing lists, release management, version control, bug tracking, and documentation will be especially important to measure and prioritize.

Running an infrastructure internally

If a team decides to run their infrastructure internally will need to be prepared to commit a fair amount of time to set up and maintain the installation. This can benefit companies who are in certain situations because it gives them control over the repository and its inevitable customization. If a team is already running other infrastructure for their current project flow, it will be worthwhile, because most likely the organization will be hosting more projects in the future.

Teams who utilize an internally controlled infrastructure will find that it is easier to distinguish which developers can collaborate with others to complete projects faster. The size of a team and the scale of their distribution will be important factors, because at a certain point it may not make sense to be tied to a specific institution.

Generally speaking, it is no longer a benefit to run an infrastructure internally if the driving motivation to do so is based on the fear of losing your code because a third-party provider goes out of business. Teams can protect their content by maintaining a good backup regime through distributed revision control systems, so this should not be an issue. Using these **DRCS** systems will retain a full copy of your code and commit history, so moving your repository will be a basic task, as opposed to a stressful ordeal.

Choosing the right Repository tools for your Organization

What factors determine the correct type of repository, and who makes the final decision? Anybody who is involved in the software development process should be involved in determining the expected criteria of a **source code repository**. It is recommended that developers perform their assessments of code quality, usability, and overall sustainability through experimentation. Obviously with so many options there is no set of hard and fast rules, so a variety of factors will apply in certain situations and some will not.

For some, an approach that measures sustainability, usability, and maintainability should be used. Others may want developers to evaluate the software by creating a reproducible record of experiences. This way, they can assess any potential technical barriers that can hinder adoption. They will want to answer these questions:

- *Will software get released as binary?*
- *Will users need to access an online portal?*
- *Will the software be released as a source with users must build?*
- *Issue Tracker?*
- *User documentation*

Who is your software being evaluated for? The results of the evaluation will be reported to somebody, so the development team should have an agreement with the scope of each evaluation, including software and project resources. A realistic synopsis requires user involvement, because it will by necessity need to be looked at from their perspective. Consequently, they will need to help choose the priority requirements from the basis of the evaluation.

- *What type of user are they?*
- *Will their workload include downloading, installing, or configuring system environments?*
- *Is there a web portal, and who needs access to it?*
- *Do the users write code?*
- *Do the users update software, fix bugs, or extend the functionality of the current solution?*
- *Will they work on legal tasks - policy upgrades, licensing, copywrites, and coding standards?*
- *Will they be expected to support new software components that are created?*

When developers report their evaluation findings, the analytics should cover all of the basic facts. Reports should include the evaluation completion date, a brief synopsis of the software functionality, and the software producer's credibility rating. If possible, final findings should be prioritized, grouped into sensible categories and cross-referenced to the relevant requirements of the agreed-upon scope.

The versions of software that were used should be recorded along with any questions or concerns that could change the quality of the proposition. Estimate the impact of addressing the most pressing issues that were encountered, so that a cost-benefit analysis can be conducted with confidence.

When gathering useful information out of evaluation, teams will spend an ideal period of 1-2 weeks in duration, and/or 3-5 days of effort. Determining factors will depend on the complexity of the software and the nature of the evaluation requirements.

Separating requirements & dependencies

Business owners and project managers should not try to describe the requirements of an interface before the concerns of the end-users have been identified and explored. Generally speaking, it is critical for a system to satisfy multiple properties, perform multiple functions simultaneously, and address numerous purposes. The user interface will reflect and identify the concerns that need to be understood by developers so that they create a product that improves the experience.

Requirements analysis can be approached in many different ways, most of which will be catered to a particular user type. Scenarios or User Cases are anticipated, so interactions between users and the system can be addressed and coded accordingly.

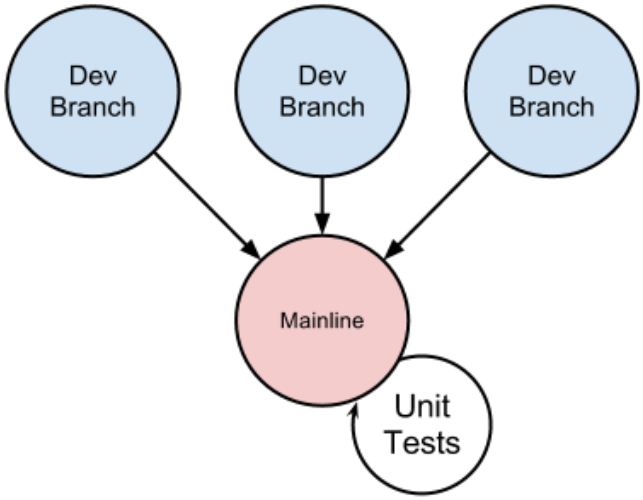
Accessing overall effectiveness can be measured more realistically against the experience of users. The behaviors of the users the most important data that is available, because it ensures that the requirements interface properly with the expectations of the client. If this is not done, the result will likely be a software product that does not properly address requirements and has unnecessarily complicated code.

To use branches or not?

The bigger and more complex projects are virtually guaranteed to experience broken builds on the path of development. Larger teams, legacy code base, outside vendors, and other factors are important issues that need to be addressed. They should not, however, take up time for developers, because developers are paid to write code and create software. The longer it takes to address integration issues, the more stressful it will be for developers to be in a state of constantly working on defects, and trying to keep up with deadlines.

Business concerns do not necessarily reflect the importance of fixing errors, primarily because it is difficult to quantify the time and money that is lost because of efforts to re-write broken code. Eventually, the team will be incapable of fixing all errors in time. Often, the business side of the organization has priorities that do not coincide with “clean builds”, and technical leaders can’t buy time for fixing unit tests.

Branching can work alongside **Continuous Integration**, providing that everyone who is on the development team is working out of the same tree. Every time a developer checks code into the main tree, they create a separate local branch of development. Every time code is committed, it is merged into the main build. It is a balance that needs to be struck between integration and isolation, and each developer will need to apply a certain degree of each in a branch to accomplish anything. What's needed is a proper balance of integration and isolation.



Centralized Continuous Integration

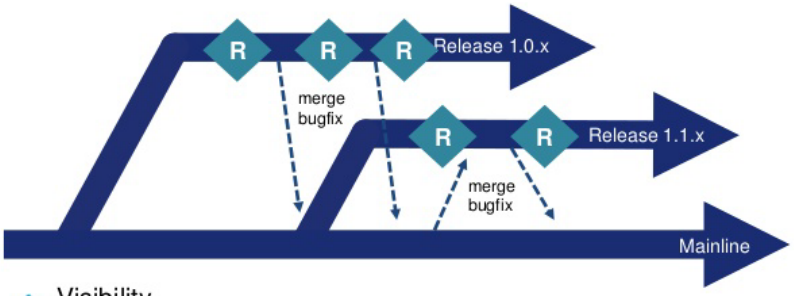
Developers can work on branches without breaking any of the rules of **Continuous Integration**. As long as developers are prohibited from merging bad code into the master/ tree trunk, the integration cycle will not be negatively affected. Programmers commit incremental changes to the branch often, and they can be easily reviewed by peers in real-time. The last thing an organization wants is bad code infecting the entire solution, so unit testing will guarantee that the entire branch will be rejected without exception.

Development teams need to have the ability to isolate non-related problems as required. Inadequate branching strategies will likely cause more work and a bigger headache with lengthy integration and release cycles. The appropriate question is not necessarily if developers should utilize branches, it's more a decision of how many branches will be necessary, and for what purpose? Code commits will still be tracked, and they will still know if their code updates adversely impacted somebody else.

There is often pushback when the concept of **CI** is introduced because initially, this approach slows down the development. When developers have to write cleaner code, it takes more effort, but the results pay off. Code quality is increased, and troubleshooting time & effort is minimized.

Tests are going to fail, there is simply no way around this reality. When they fail, the common phrase states that the "build is broken". Broken builds are an excellent quality control mechanism, because errors are dealt with as soon as they are discovered, and they are never committed so that each new build remains clean. The latest build should always be clean. Always.

With branching, developers can make changes and run tests without the fear of infecting the main codebase. Coding languages can be modified, developed, and tested in isolation with parallel development practices.



- Visibility
- Always deployable
- Continuous Integration, only create branches for releases
- Manage what gets into mainline
  - Continuous Integration
  - Code Review

Some have referred to **Continuous Integration** as a tool when in actuality it is a practice. Team members check in and commit code multiple times per day, and each upgrade is verified by a build server. when every team member is committing to a Trunk multiple times a day, Continuous Integration is achieved.

Now that we have covered that, let's look at a situation that would not necessarily entail Branching. The good thing about abstaining from using developer branches, they always push their changes to the master. When this method is used, pair programming will be important, because it will be the only time that collaboration will happen in the pre-commit stage. We do not want to periodically integrate code; we want to integrate updates as often as possible.

The more often everyone commits to the master, the smaller the batch size, which results in a number of substantial benefits. Remember, pre-commit collaboration is preferential to any kind of post-commit collaboration because it eliminates rework.

One of the main points is this: **Branches** do make it possible for developers to retrieve and review each other's code before it gets merged, but it can also enable them to write software in isolation from each other, which defeats the purpose of branching in the first place. This way, if somebody creates a new function or feature, all of the other developers can start using it, and improving it as soon as it is committed. Utilizing these practices will enhance opportunities for communication and collaboration between the development team.

Configure Automated Deployment

For an **Agile** development team, building and testing code daily is not enough, because this would leave too much room for defects to creep their way in while developers continue to add layers of code. By the time a bug is discovered, more code has been committed on top of broken code - making it harder and more expensive to fix.

Testing upgrades as soon as they are committed dramatically reduces the cost of addressing defects, so kick each commit should be guarded by gated check-in. Automated builds should also run at scheduled intervals throughout the day.

Continuous Integration fueled by automation is the cornerstone of effective an **Agile** development process. Immediate feedback is unavoidable because each commit is a small, incremental change that is easily tested and integrated into the build.

Automation eliminates the effort and time that used to be dedicated to troubleshooting bugs and code collision. Features are tested and integrated early and often throughout the entire development cycle. It also reduces human interaction, which speeds up the process, and makes it less likely to create user errors. Simply put, automation increases the productivity of a development team by reducing time that is spent on low-value tasks.

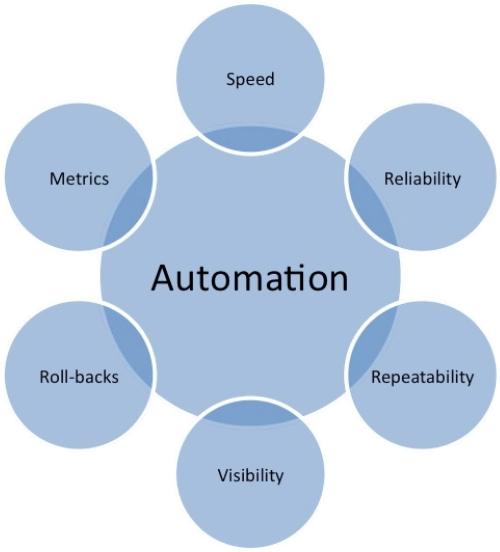
When **unit tests** are automated, they can be automatically triggered with every commit, recorded, and reported on multiple times per day. Most of the assets available can assist developers with build tools that automate tasks like packaging, compiling, and deployment.

When testers are stuck with repetitive manual testing, it can dull their senses and increase the probability of user error. Consequently, defects slide through and become expensive to fix because they are not detected until later. The good news is, **code refactoring** doesn't have to be an ordeal anymore. Automated testing ensures that each new code commit is rigorously tested.

The need for developers to refactor is heavily diminished when **unit tests** are thoroughly run against it. Unit tests will catch errors immediately, so the developer be alerted and able to fix them before they will pass the test. Some of the most popular automation tools include:

How frequently do teams need to verify their existing backup strategy and assess the necessary time and effort necessary to recover? Software development teams will be well served to adopt the "*Driven by customer Demand*" philosophy to keep business expectations aligned with software requirements. Development & operation departments should work and plan in conjunction with each other to ensure that the expected behavior of the software is verified before it is demonstrated to first time users.

Code-based infrastructure enables organizations to justifiably treat the development of automated testing as a priority discipline. When the development process encourages and enforces an automated deployment process, teams are slated to create code that is deployable in any working environments at the end of each iteration. Business owners are happy, and developers can to be flexible with changing requirements from user feedback upon request. The final result is software that is efficient, stable, and easy to get into production reliably and quickly.



The Benefits of Automated Deployments

A build is not ready to be released into production until all tests and integration issues have been tested and proven to be work at sufficient quality to call done. Let's look at the result of what happens when you don't automate your testing and integration efforts. To be truly Agile, all deployments need to be fully automated, thus enabling the release of software products in a repeatable and reliable push-button solution.

Without automated **unit testing**, the development process will:

- *Be less efficient, and consistently unreliable across multiple environments*
- *Be slower, non-repeatable, and subject to user error*
- *Bog down developers tracking extensive documentation required for continuous improvement*
- *hinder collaboration between developers, business owners, and users*

Why do teams initially avoid Automated Testing?

It seems like it would be an obvious process to build into daily workflows, but many development teams are slow to adopt these practices, because of the initial "perceived decrease" in productivity. It is easy to sympathize with this, being that the perceived overhead of creating, configuring, integrating, and maintaining automated testing mechanisms will hamper process in its first stage of adoption. By tracking and measuring test results, developers will be able to see and demonstrate the benefits in a cost-benefit analysis.

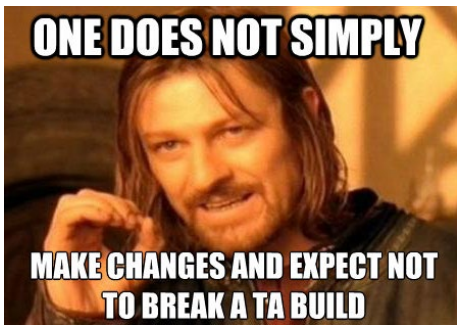
- *"Our development process is already complex enough"*
- *"Creating all of those tests will take time away from writing code"*
- *"It's not worth the effort, and business doesn't think it will hinder deadlines."*

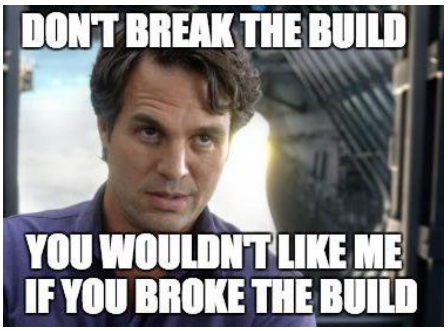
Development teams who have never created and utilized automation testing are likely to view the perceived extra work as a sticking point. Undefined work tends to look massive and insurmountable when it hasn't been broken down, plus they probably don't know where to start when initializing the endeavor. When practiced in combination with **Agile** development practices and automated testing, continuous integration allows you to minimize time and effort by reducing waste. Speed to market is improved because the software is developed in a manner that ensures it is consistently functional in any real-world environment.

When each patch of code is checked in, tracked and analyzed, it is much more manageable to assess any areas that need refactoring or might be at risk. When developers don't need to guess, efforts to fix bugs are focused, purposeful, and consequently more manageable.

When code enters the commit stage, the build automation server should test it against the software repository by using a virtual environment that does not affect existing software adversely. During the acceptance test stage, automated scripts should mimic real-world runtime environments, and specifications based on anticipated user actions. Automated smoke tests and acceptance tests will be executed and verify the application conforms to business-critical customer acceptance criteria. Beyond the business level, automated capacity tests, or load tests need to be run to verify the system won't bend under heavy production conditions. After having passed all of the necessary stages, a successful build is made available in the repository, from where it can be released into production upon requests.

Every software development team that is **Agile** should have a fully automated test, build, and deployment process. There is no room to debate this point, it is a mandatory pre-requisite. Many development teams utilize processes that are partially automated, and partially manual. The small percentage that sees the greatest benefit from Continuous Integration will have a hands-off, fully automated process.





There are quite a few automated software deployment tools that can help organizations reduce the workload of developers. The amount of effort required is not as dire as it may look at first glance. Choosing the tools that are easily integrated into your existing infrastructure will reduce the overhead of setting up the automated processes.

What is a unit test?

Unit testing can be defined as the testing of a small "unit" of code. Typically, unit tests are performed by the developer of the code that's being tested. For **unit tests** to be executed for peak performance, the tests themselves should be coded so that they can be replicated and run again through an automated process.

Traditionally, software developers design, code, and then test their product. With the **Test-Driven Development** portion of **Continuous Integration**, the approach is more iterative. With this practice, developers don't need to wait until the entire design is completed to start writing code. Designing, testing, development, and deployment are completed in small pieces and small iterations. This helps designs evolve according to necessity and helps to avoid getting off track and experiencing code drift.

With **TDD**, **unit tests** are required for the implementation of **Agile** methodology to the development process. First, the developer should determine the desired outcome of a small segment of code, or functionality. The testing drives the development of the code because it needs to fulfill the conditions of a pre-written requirement. In other words, testing for the code is written before the application code is created. The code will fail in testing until the test pre-requisites are satisfied. Another benefit of this practice is that the tests become the documentation of each requirement. These records can be extremely useful when future issues arise. Automated tests should be used as often as possible.

During the development process, database changes should be integrated as frequently as possible, so that it makes changes easier, creates more flexible code, and identifies code clashes and dependencies as early as possible. When these test results are recorded, problems are visible while they are still manageable, giving developers a blueprint with which they can compare future integrations.

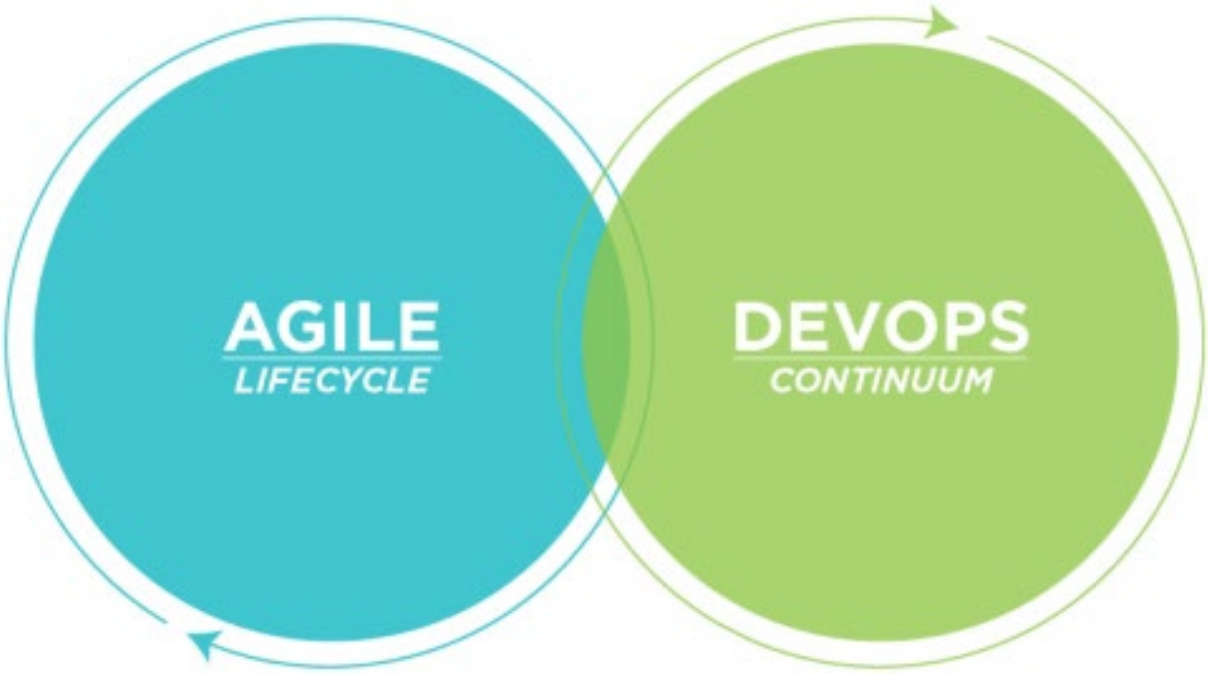
There is no way to get rid of bugs, they are just an unfortunate fact of life. With **Continuous Integration** and Agile practices, they are dramatically easier to locate and remove. If the team knows that the version with most recent integration is solid, they have a solid blueprint that helps know what to look for when issues arise. and developers, management, and testers will find that the **Database Lifecycle** is more streamlined and easier to navigate. If the software fails, the entire team knows immediately, and further integrations are prohibited with a gatekeeper until the broken code is fixed.

Unit Tests ensure that individual units of code can be tested in isolation, function according to expectations, and integrate successfully into existing infrastructure. Database requirements that are defined by automated tests ensure quality and save time and effort.

Why is Continuous Integration so important?

Generally speaking, most applications are made up of *customized procedural code, third-party components, libraries, frameworks, user interfaces, and databases*. The usual suspects are usually around too, materializing in a mashup of off-the-shelf applications. Software components are likely to need upgrades and improvements along the way, especially during the development process. With technology moving at such a rapid pace, new resources and solutions are popping up every month, and organizations who do not adapt will not do well in this competitive environment. **Continuous Integration** hinges on developers embracing new requirements with flexible software whenever necessary. When technology changes, applications should be able to change with it, therefore integration should be continuous. **CI** is integral because it forces new components to be tested and prove that nothing has been broken as the result of a change. **Code commits** trigger the **automated testing,** and if methods within the system perform as expected, then they will be added to the build. Each update prompts **integration testing** and a fresh build.

As time goes on, applications will tend to expand, and developers will continuously refactor the schema and database code objects to adhere to the needs and desires of the business. The **Agile** development process proceeds in small steps, as developers integrate updated code into a shared version control repository several times a day. Tests are written to drive development in the correct direction, and when the code passes, this small piece of functionality is integrated into the mainline, or "trunk". Each new object that is checked in is automatically verified by an automated database build, and subsequent testing, helping developers to detect problems early.



Eventually, development teams will find value in breaking builds up into many smaller chunks that can run in parallel to one another. Depending on the size of each project, test suites should be broken into smaller batches so that they can be run simultaneously. The more recent the code was written, the easier it is to troubleshoot, and this will be the fastest way to get feedback to developers. This is integral, because continuous integration builds may need to run dozens of testing jobs in parallel to function efficiently. Environments that favor parallelization will need to at least match, but preferably exceed the number of test jobs. Before all is said and done, most developers will have tested against different browsers, operating systems, databases servers, and so on.

The application of **Agile** & **DevOps** methodologies can be the main factor that determines whether a software team fails or succeeds. These methodologies are heavily reliant upon Continuous Integration, which provides a lifeline for code improvement that needs to be initiated at a moment's notice.

Conclusion

When **Continuous Integration** is implemented correctly, it creates more time for developers to focus on creative tasks, because issues are detected early, and dealt with immediately. The new paradigm requires results upon demand because clients want to know what they are paying for in real-time. Teams that practice the evolved form of **CI** will be able to deploy examples whenever necessary, and they will see a decrease in disruption, unproductive downtime, and stress on those who are involved with the development process. Once these results become a reality, word will spread, and your company's product will experience a significant brand lift. Testimonials will abound, and those who require examples will be provided with analytics reports that prove the success and value of your establishments' quantified services.