Continuous Integration and Build Management Server Evaluation Guide

by Maciej Zawadzki

Urbancode, Inc.
2044 Euclid Avenue • Suite 600
Cleveland, Ohio • 44115
P: 216.858.9000 • www.anthillpro.com

Contents

Devilish Details

Evaluating Continuous Integration (CI) and Build Management (BM) servers may appear like a straight forward proposition, but that can be deceiving. Having responded to well over 50 RFPs in the last 12 months alone, we recognize that the marketing checklist is one thing, and the operational needs of the team and enterprise are quite another. If you will soon be looking at CI and/or BM (perhaps beyond) for your organization, this paper can help you structure the right requirements for your need set, as well as help you produce an RFP that insures a quick ROI.

The paper is organized around three feature sets that we typically see in evaluation scenarios. The first focuses on CI and provides some of the basic functionality required for an implementation of either CI and/or BM. In this way, the CI feature set lays the foundation for the BM feature set that follows. The BM feature set focuses on providing the traceability that is critical to a successful implementation of the practice. And the Enterprise Class feature set focuses on the attributes and deliverables that we have found are indispensable for implementing either a CI or BM server at an enterprise scale.

Depending on the goals of a particular project/evaluation, the emphasis of any of the three feature sets may change. For example, a small team of developers evaluating a CI server may not assign much weight to the Enterprise Class feature set. Whereas a Fortune 100 enterprise looking for an enterprise wide BM solution to help aid with compliance requirements may not assign much value to the CI feature set.

We hope you find this guide to evaluating CI and/or BM servers useful. It is intended to provide some background and educational material regarding the features that a potential solution should contain, along with explanations as to why such characteristics are important. The paper is written by people that have been there from the beginning; having released their first CI and BM server back in July of 2001. Since then, the AnthillPro team has helped over 350 organizations adopt CI, implement BM, and introduce Application Lifecycle Automation.

At the end of this paper, a summary checklist of all the features covered in the text is included. The text is annotated to correspond to items on the checklist. For example, the section on **SCM Integration** is marked with **[1]** and the **Distributed Builds** section is marked with **[12]** in the left hand column. So when using the checklist, it is possible to easily find detailed information about any particular feature. We do not provide an evaluation methodology, such as rating each solution under evaluation on a scale of 0 to 5. That scoring varies by organization, depending on your devilish details, but the framework will hopefully be helpful.

Continuous Integration Servers

As Continuous Integration (CI) proves its value to enterprise organizations, a new breed of servers has emerged to address the needs of this growing practice. A quick search for enterprise CI servers reveals a host of product offerings from numerous vendors ... each with its own take on what a CI server is. The sheer amount of information can be overwhelming when an organization is looking to integrate CI into their development environment, and may warp expectations and requirements. In order to understand what features are necessary, or even sufficient to implement CI, it is essential to have a thorough understanding of CI. Before discussing the CI feature set, a brief definition of the practice and its goal should prove helpful.

Foundations of Continuous Integration

Martin Fowler describes CI as a software development practice in which team members integrate their work frequently (at least once a day), and that each integration is verified by a build process that includes testing. For our purposes, we can make this widely accepted definition a bit more general. CI is a practice made up of two components: 1.) team members integrating their work frequently; and 2.) each successful integration does not degrade code quality.

To a large extent, CI is based on the observation that the longer developers go without integrating their work the more painful the eventual integration. It follows that the more frequently developers integrate their work the less painful the integration will be is also true. So let us push this observation to the limit and integrate continuously (e.g., with each change). However, frequent integration is only one part of the practice, and is not sufficient on its own to constitute CI. In order to practice CI, the second part of the definition must also be met: Each successful integration should not degrade code quality. This is exactly where a CI server enters the picture. It is the purpose of the CI server to determine whether an integration degrades the code quality. It is the CI server that decides whether a set of integrated changes constitute a successful integration. And if not, then the CI server equips the development team with the needed information to either back out the offending integration or correct the problem.

There are two implicit questions that have been raised: how does the CI server determine whether an integration is successful; and what is the information that needs to be provided back to the development team in order to address the problem. The standard means of determining code quality is testing. This is why CI and testing go hand in hand; and it is why a CI server needs to integrate with testing tools. If tests determine that an integration is unsuccessful, developers need to be notified regarding both the changes that were integrated and what test(s) failed. In actuality, more information may be provided to the development team, but this simple notification is the minimum.

While the concept of CI is simple and intuitive, there are many nuances in its practice. The implementation of CI raises many issues that you might not have thought of ahead of time, such as: how to ensure that a CI server obtains a consistent set of sources if a developer uses multiple commits to submit a single change; what happens if the duration of the build exceeds the duration of the build loop; or how to keep the builds from queuing up.

CI is not simply about performing builds on a tight loop. This paper seeks to point out some of the pitfalls that are frequently encountered in practice. The point of the paper is to arm you with the knowledge about such pitfalls and inform you of the features that can (and should) exist within the CI server that you select to address such pitfalls. Knowledge is power.

Pure Continuous Integration Features

To successfully practice CI, the server must provide a minimum set of features:

[1] SCM Integration

The practice of CI takes for granted (and is reliant on the fact) that the development team uses a source code management (SCM) system. Naturally, a CI server needs to be able to interact with the SCM repository. The extent of this interaction may be surprising as it goes well beyond the simple ability to retrieve the latest sources. In addition, there are changelog reports, triggers, and the quiet period.

[1.1 to 1.1.2] **Support for your SCM.** This is one of those hurdles that you need to get through just to get to the more challenging and fun stuff. The first questions here are: how well does the CI server support your SCM out-of-the-box. If it has a built-in integration, then that is ideal. If there is no built-in integration, the question "how much effort is it going to take to develop the integration and then to maintain it?" arises. The nuance on the above is whether the tool supports your usage pattern for your SCM. For example, you may always build from a branch or stream ... does the tool support that? Or you may build from a label or snapshot ... is that supported? The more "enterprise-ready" (read complicated) your SCM tool is, the more potential ways to use it there may be. You need to determine whether the CI tools you are evaluating are going to support the way you interact with your SCM.

The CI server needs to be able to produce a changelog report based on data in the SCM. While this item perhaps belongs in the "Feedback" section, its implementation is reliant on integration between the CI server and your SCM. To satisfy this criterion, the CI server needs to be able to produce a report identifying all source code changes included in the CI build. This is critical to a proper implementation of CI, a point that will be discussed in further detail (see the "Feedback" section).

[1.2 to 1.2.2] **Triggers.** Triggers are what tell the CI server that some new source code changes are available in the SCM and should be verified to determine whether they degraded the code quality. Triggers fall into two categories: polling and event-based.

Polling triggers run on a tight schedule such as every 15 or 30 minutes. Every time the schedule fires, the trigger runs an SCM changelog command to determine whether there have been any new changes committed to the repository since the last build. This is a very simple and effective system but with several drawbacks. One problem is that polling triggers exert a load on the SCM server in a periodic pattern, even if no changes have been made to the source code. This can impact developers trying to interact with the SCM server at the same time. Another problem with the polling trigger is that it does not actually produce "continuous" integration, as changes get batched together based on the polling interval. This means that developers may be waiting for a long time before getting feedback on their integrated changes. There is an upside to polling triggers: they will work in virtually every environment. For this reason, you should ensure that any CI server supports polling triggers.

Event-based triggers are tightly integrated with the SCM and usually consist of a script that your SCM invokes every time there is a commit. The script calls out to the CI server and informs it of the commit. Event-based triggers are more efficient than polling triggers as they do not tax the SCM server resources unnecessarily. They also produce a more "continuous" behavior, since the CI server is notified of every commit as soon as it happens. The down side to event-based triggers is that they require the SCM to run a script which may not be an "approved" integration with the SCM, and thus not every organization may be permitted to make use of event-based triggers. When evaluating CI servers, make sure to dig into the details of how SCM triggers are implemented. Many CI servers make it seem like they provide event-based triggers even though they actually implement polling triggers underneath. Keep in mind that terms such as "polling" and "event-based" are not necessarily used in the same manner by all vendors. You have to dig under the surface to find out what is actually going on.

[1.3 to 1.3.1]

Quiet Periods. Quiet periods ensure that the source code used for a CI builds is consistent. There are two well known situations that can produce an inconsistent code base for a build. The first is the situation where a developer uses multiple commits to submit a single change. For example, the developer may have modified two files in one directory and commits those together before committing a second set of files in another directory. If a build grabs the sources after the first commit but before the second commit, the code base will be in an inconsistent state.

The second scenario that can produce an inconsistent code base is a non-transactional SCM. Unfortunately there are still non-transactional SCMs in use today, and they range from open source ones to some of the most popular enterprise-level systems. When a non-transactional SCM is used in conjunction with a polling trigger, the trigger can fire in the middle of a multi-file commit. When that happens, since a set of related changes to multiple files is not treated as a single commit but as a series of individual file commits, the CI server can grab source code from the middle of the commit. Fortunately, if your server implements the quiet period feature both of the above mentioned scenarios can be overcome.

The quiet period feature assumes that a single cohesive set of changes may be split into a sequence of multiple commits. The challenge is in the grouping of commits into cohesive changes. The strategy is to use periods devoid of commits (the quiet periods) to demarcate the boundaries between cohesive sets of changes. CI servers that implement the quiet period feature will be able to effectively handle the scenarios discussed above; however, a server without this feature runs the risk of using an inconsistent set of sources for builds.

There is a common mistake or short-cut that can all too often be found in a quiet period implementation. We can use **Fig. 1** to illustrate the mistake. Let us assume that our quiet period is configured

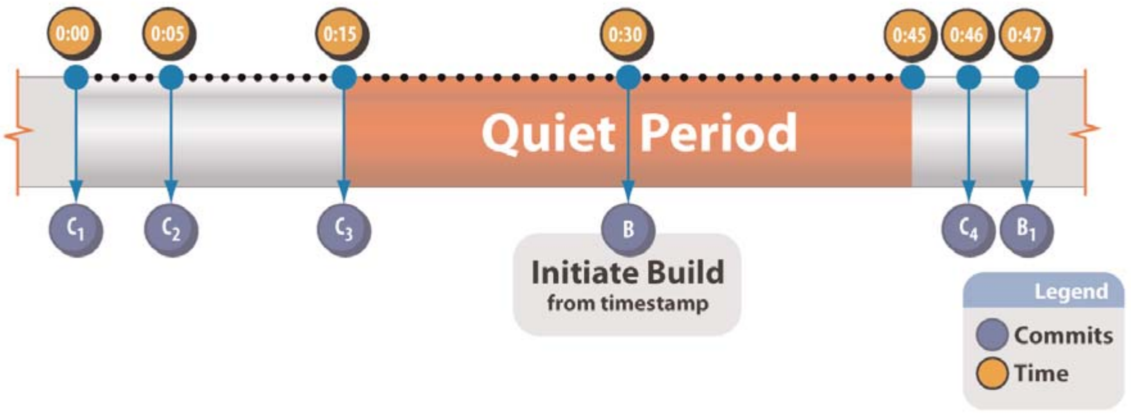

Fig. 1

to be 30 seconds and that there is a commit (C1) at 0:00, followed by a commit (C2) at 0:05, and a commit (C3) at 0:15. A CI server will notice that the quiet period expired at 0:45 and will then proceed to get the source code. If the CI server obtains the latest source code from the SCM after determining that a quiet period passed, then there is a race condition. Computers are fast, but they are not instantaneous. There is some period of time that passes between the determination that the quiet period has been met (at 0:45) and the retrieval of the source code from the SCM (B1 at 0:47). And this period of time is enough to let a commit (C4) at 0:46 sneak in and produce an inconsistent set of sources for the build. The proper implementation of this feature would not obtain the latest sources from the repository after ensuring a quiet period passed. Instead, it would obtain source from the middle of the quiet period (B at 0:30). As with the event-based triggers, the devil is in the details here. You must dig under the surface to determine whether the quiet period implementation is flawed or not.

[2 TO 2.2]

Build Tool Integrations

It is easy to fall into the trap of thinking that build production is the main purpose of a CI server, since CI servers build the source code obtained from the SCM. But the build is an ancillary part of CI. (To learn about the CI build and how it differs from the BM build, please refer to our earlier white paper "Drawing the Line: Continuous Integration and Build Management"). Nevertheless, CI servers do build source code and thus they must have integrations with build tools. When evaluating CI servers, you should ensure that the CI server does in fact integrate with the build tools that are used within your organization. Perhaps the only confusing part of this evaluation criterion is the determination of what type of build tools we are talking about, since the CI server itself can be viewed as a build tool. The best way to clear up this confusion is to provide examples of build tools that the CI server should integrate with. They are: Ant, Maven, Nant, MSBuild, Make, and the list can go on. In some cases, a CI server will not have out-of-the-box integrations with all the build tools in your organization. In those cases, you need to ensure that the CI server will be able to invoke any non-integrated build tools via some other means.

[3]

Quality Determination

It can be said that the purpose of the CI build is to determine whether there has been degradation in code quality due to an integration. But before we detect a degradation in code quality, we need some means to determine code quality. Current technology allows us to do this through testing. Therefore, the CI server must integrate with testing and coverage tools. But be forewarned that there is a lot of snake oil out there regarding tool integrations. Some vendors will claim to integrate with every tool under the sun simply because they allow the user to manually configure a step to call the tool's command-line interface. Since determining code quality and whether it has degraded is the primary purpose of the CI server, the integrations with testing and code-coverage tools have to be meaningful.

[3.1 to 3.1.4]

Testing Tool Integrations. Virtually any tool that calls itself a CI server can call out to a testing tool. The difference is in what happens next: does the CI server have a way to capture the testing tool output and reports? In order to do that, the test reports need to be placed somewhere like an integrated build portal. Without an integrated build portal that includes testing reports, the value proposition of the CI server is severely diluted as developers will have to hunt through additional web sites, FTP sites, or directory trees to locate reports that should automatically be associated with a build.

Does the CI server parse the output of the testing tool and store it in a data warehouse? In order to do this, the CI server needs to understand the output of the testing tool. Storing data in a data warehouse makes it available for various metrics, analysis, trending, and that key point of it all: determining whether the quality of code in the most recent CI build was lower than the code quality in the previous build.

Being able to parse the output of testing tools allows one other piece of functionality; the ability to make logical decisions based on the test results. For example, in development it may be completely expected that 5 percent of the unit tests fail at any one time. But, if that ratio goes up to 6 percent, then perhaps the project manager and/or the lead developer should be automatically notified.

[3.2 to 3.2.4]

Test Coverage Tool Integrations. As with testing tool integration, the same points raised regarding testing tool integrations apply to code coverage tool integrations. Can the CI server capture the output and reports and place them in an integrated build portal? Can the CI server parse the output and store it in an integrated data warehouse? Can logical decisions be made based on the output of the code coverage tool? Can the data stored in the data warehouse be consumed as metrics?

[3.3] **Means to Integrate Additional Reports.** The CI server should allow reports from additional tools (such as static code analyzers, and other tools) to be integrated into the build portal so that developers can quickly and easily locate them.

[4] Feedback

CI is concerned with integrated source code changes, the effect those changes have on quality, and communicating that information to the development team. It is critical that the CI server be timely and targeted in its communications. Remember that the longer the developers go in their separate directions, the more difficult the eventual integration will be. Likewise, the longer the development team goes down the wrong path, such as working on top of changes that degraded code quality, the more painful it will be to correct that mistake. That is why timely feedback is key, and it is why CI practitioners often refer to the feedback loop and the quest to shorten said loop. Features such as event-based triggers (discussed above) shorten feedback loops. In addition to being timely, feedback must be effective so that it will be acted upon. In this section we focus on features that make feedback effective.

[4.1 to 4.1.2] **Notifications.** Effective feedback is woven into the set of tools that developers already utilize. Therefore, the CI server should be flexible in the choice of communication medium that is utilized to send feedback to developers. Some developers may prefer to receive e-mails; while others may prefer instant messages; and yet others may prefer a tray icon to provide a visual cue. Of course, a notification system integrated with an IDE via a plug-in also makes a lot of sense. So make sure that the CI server you are selecting supports the notification medium that your developers would prefer to use.

In addition to the notification medium, the format of the notification message should be taken into consideration. Is the CI server flexible enough to give you control over the message itself? Some teams like to receive short direct messages that are to the point, along with a link to the build portal where additional information is available. Yet other teams would prefer to receive all the pertinent information right in the notification message.

The "blame developers" feature allows a quality-degradation-feedback message to be sent directly to the developer or developers that are responsible for the code changes. This is a very effective feature, as it ensures that developers do not ignore or "tune out" the feedback messages from the CI server, as they might do if they receive every failure notification.

[4.2 to 4.2.2] **Build Portal (Dashboard) with Reports.** The build portal, sometimes called the dashboard, should be integrated into the CI server; and should provide a single location where all the information about the CI build is accessible via a browser. Code quality reports (mentioned earlier) as well as the changelog and any error logs all need to be readily available in the build portal. In that pragmatic and fast-paced world of Agile development, a tool that does not place all the pertinent information in a single, logical, and easy to find location is simply not going to provide the needed value.

One of the crucial pieces of information for a CI server to make available in the build portal is the changelog. This piece of information is indispensable in the situation where integrated code changes have to be backed out. In that case, the changelog will identify exactly what source code changes need to be rolled back. Any tool incapable of producing an easy to consume changelog detailing the source code changes integrated and verified by the build cannot be considered to be a CI server.

Sometimes it is easier to fix an offending piece of code than to back it out. In order to do that, developers need to have immediate access to any error logs and other information that would help pin point the problem causing the quality degradation. The pertinent piece of information may be a build-log that shows off a compilation problem. Or it may be a test report that details a particular test failure. Whatever the pertinent piece of information is, it should be included in the build portal so that it can be quickly and easily accessed by the development team.

[4.3] **IDE Plugins.** IDE plugins have already been mentioned above as a communication medium; however they are important enough to warrant their own subsection and evaluation criterion. Because the CI server is mainly a developer tool, it needs to be integrated with the developer tool set. The reality is that a vast majority of developers spend most of their time working in their favorite IDE. Being able to interact with the CI server from within their IDE provides developers with additional value.

[5] Smart Build Queue

This is one of those features that separates hardened CI servers from the impostors. Let's start by describing the situation that makes this feature so important. Suppose you have a polling trigger that polls for changes every 15 minutes. Your build takes about 13 minutes to run, so your server is almost always busy producing a build. But as your project grows, your build time grows as well, all the way up to 20 minutes per build. Now, when the trigger fires in an attempt to kick off a build to verify the integration of some new code changes, the previous CI build is still running.

There are several things that can happen at this point. You could allow multiple builds of the same project to run concurrently. This may work for a while if you have enough machines to handle all these concurrent builds. If the problem gets bad enough, or if you are supporting multiple projects, it is just a matter of time before you have builds that are queuing up because the system cannot handle the build throughput ... no matter how many build machines you have. And once builds start queuing up, your feedback loop gets longer and longer. Soon, your development team will be waiting hours for the feedback on their changes.

The solution to the above problem is to use a CI server that includes a smart build queue that automatically adjusts the granularity level of builds by coalescing build requests generated by triggers. If the build queue contains a request to build project A with changes from 9:45 when another request to build project A with changes from 10:00 comes in, the smart build queue will consolidate the two requests. The result will be a single request to build project A with changes from 10:00 since the changes from 9:45 will be automatically included if we take the code from 10:00. Of course, you need to make sure that the build request coalescing implementation takes into account the quiet period; however, that may take a bit of probing.

[6] Dependency Management

With the trend toward more complex software, the popularity of SOA and composite applications, as well as the availability of numerous open source and third-party libraries, the ability to provide dependency management has become almost indispensable. Treatment of dependency management by CI servers ranges from none at all; to simplistic event-based schemes; to integrated, sophisticated dependency management solutions. And while it may seem that a dependency management system should exist independently from a CI server, the impact of a dependency tree on build scheduling suggests otherwise.

[6.1 to 6.1.2] **Inter-project Dependencies.** Inter-project dependencies allow projects to depend on the artifacts produced by other projects. For example, project A may be dependent on project B, thus project A may require artifacts produced by a build of project B to be available to a build of project A. In such situations we typically say that project A is dependent on project B. Notice, however, that the statement that project A is dependent on project B is ambiguous since it says nothing about what build of project A depends on what build of project B. Presumably successive builds of project A and of project B include source code changes and some of those source code changes are incompatible.

Removing the ambiguity without placing a heavy burden on the developer is the key to a good dependency management system, and can be accomplished with the aid of flexible dependency relationships. A simplistic strategy for resolving this ambiguity in a flexible manner may be to create a rule that states that the latest build of A is dependent on the latest build of B. A more sophisticated dependency relationship would allow A to depend on the latest build of B that has passed not only the unit test but also the functional test suite. These are just two examples of flexible dependency relationship rules that should be supported by the CI server.

Of course, any dependency management system needs to handle transitive dependencies. While on the surface this may look like a simple feature set, once different types of artifacts enter the equation, robust handling of transitive dependencies can be difficult to find.

[6.2] **Pulling Builds.** Consider a simple dependency scenario: project A is dependent on project B. Lets say that a source code change spanning both project A and B is committed at 2:01. At 2:05 we request a build of project A. The CI server could simply get the sources of project A and use the artifacts produced by the latest build of project B (lets say from 1:30). But that would produce a failed build of project A and would incorrectly indicate that the code quality of project A has degraded.

Instead the CI server should know that project A depends on project B and should determine whether project B needs to be built before it proceeds to build project A. We call this "pulling builds". The CI server should include support for this feature if you plan to use dependency management at all. But our example does not end there.

Let us say that our CI server correctly checks the dependency (project B) and determines that it should be built prior to the build of project A. Let us further say that at 2:05 the build of project B commences and lasts 10 minutes. Thus, at 2:15 the CI server is ready to start a build of project A. In order to properly handle this situation, the CI server needs to obtain the source code for the build of project A using the exact same timestamp that was used to build project B. If the CI server chooses to use the source code for project A as it exists after the build of project B, meaning at 2:15, then it again can lead to a failed build, since there could have been a change to both projects A and B committed at 2:10. This example points out at least some of the intricacies that must be handled by the CI server when scheduling builds in a dependency graph.

[6.3] **Pushing Builds.** Pushing builds works in the opposite direction of pulling builds. In the same scenario as above (A dependent on B), rather than a build request for A pulling a build of B, a build of B would push a build of A. The reason that you may want a build of B to push a build of A is to ensure that the changes made to B do not break or degrade the functionality in project A.

Most CI servers, if they include any dependency management features, support pushing builds. However, as we have seen, a seemingly simple feature set has many intricacies. For example, just as with pulling builds, the CI server should ensure that the timestamp used to retrieve the source code for the build of A is the same as the timestamp used for the build of B. Very few CI servers ensure this ... even if they claim support for pushing builds.

Another complication with pushing builds occurs if there are multiple paths between nodes in the dependency graph. For example, take project A that is dependent on B and C, with project B also dependent on project C. In this scenario there are two paths from C to A: directly from C to A; from C to B and from B to A. In this scenario, a build of project C should result in a build of project B, which should result in a single build of project A. Simplistic implementations of this feature set will invoke two builds of project A, one build for each path from project C to project A.

[6.4] **Dependencies on Third-party Artifacts.** The integrated dependency management system should support dependencies on third-party libraries in addition to supporting inter-project dependencies. In today's world, most software projects rely on either open source libraries or commercial components. Managing these relationships through various releases and upgrades provides a lot of value.

[6.5] **Traceability of Artifact Usage.** A dependency management system should be able to provide information necessary to conduct impact analysis and full traceability of artifacts. Let us say that you found a bug with a build of a library that is used by several projects throughout your organization. Would it not be nice to know whether the buggy build is used by any dependents? Needless to say, traceability the other way, the ability to see the exact builds of every component included in a build of a particular project, is also invaluable.

[6.6] **Provisioning Artifacts to Developers.** A dependency management system not only needs to be integrated with the CI server, but also with the developer's ecosystem. If the project is going to use a dependency management system (DMS), the DMS not only needs to provide artifacts during builds at the CI server but also provide artifacts for developer builds as well. This means that the DMS should have IDE plugins that allow developers to hook into the DMS from within the IDE. It also means that the DMS should have integrations with existing build tools such as Ant, Maven, Nant, MSBuild, etc. And, the DMS should have a command line interface as well.

[7 to 7.3] Ease of Configuration

A modern CI server will provide a web interface through which all configuration can be performed. The days of logging into a remote machine and manually editing flat files or XML file are over. In the leading CI servers, the fight is over ease of use and flexibility. The two goals may at times seem to be contradictory, but there are several approaches employed to help reconcile them.

The first approach is to allow users to pre-configure a pertinent piece of a process and then reuse that piece within multiple projects. This feature allows users to effectively create a library of reusable processes. Some CI servers make the 80-percent case simple and easy, forsaking the remaining 20 percent. Other products focus on the general case and thus require some extra work for every scenario. Yet other servers try to be smart about configuration: if the user is configuring a common process such as a CI build, the CI server can imply common steps such as source code check-out and allow the user to fill in the remaining pieces. This approach attempts to marry the ease of use for the 80-percent case with the flexibility required for the remaining 20 percent.

[8] Build Purging Rules

One of the repercussions of implementing CI is that your active projects start to accumulate quite a few builds. For a moderately active project, 10 to 20 builds per day is very common. As we have seen, the amount of data attached to each build is less than trivial: there are changelog reports, test reports, coverage reports, various additional reports ... not to mention the artifact which you may be keeping around. The problem with accumulating all these builds is that they take up space. Thus, the CI server you are evaluating should have some built-in mechanism to purge unneeded builds, or at least to reduce the amount of space taken up by such builds. Ideally, the determination of what is an "unneeded" build is flexible enough to allow you to customize it to your needs. Perhaps you only want to hold onto the builds from the last 5 days; or you want the latest 10 successful builds and the last failed build. The possibilities are nearly endless, and a flexible system of determining what is no longer needed is required here.

Build Management Servers

While it may be surprising that this paper combines the evaluation of CI servers with the evaluation of Build Management (BM) servers, the reality is that when it comes to products, supporting a CI as well as a BM feature is very natural; the difference in features is quite small. The big difference is in the practices, but the features required to implement the practices are very similar. Further, the distinction between CI features and BM features is rather arbitrary and very inexact. For example, SCM integration is required for BM just as much as it is for CI. The same goes for most of the features that we have listed for CI ... perhaps with the exception of the quiet period.

To some extent, the features that we list for BM may also be applicable to CI. For example, CI builds do create artifacts and therefore an integrated artifact management system may make sense. However, since artifact production is not the main point of CI, and it is the main point of BM, that feature set finds its home under the umbrella of BM. The reality is that many organizations use the same product for CI builds as they do for authoritative BM builds. Thus, from that point of view, it makes a lot of sense to provide a unified evaluation of products aimed at both practices.

A Brief Summary of Build Management

When evaluating BM servers, it is helpful to keep in mind that while the role and approaches to BM have shifted, the basic principles are still the same. BM, as the practice of producing an authoritative build, can be roughly described by the activities of a build master or release engineer once they have received code from a development team. The authoritative build, then, can be seen as any build prepared and executed with the intent to deliver the artifacts to an environment other than development. For example, a release build can be prepared and sent for QA testing; for a demonstration; or for beta testing. In this view of BM, builds are not scheduled. The development team decides when the code is ready for an authoritative build. Of course, there is the question of whether a scheduled nightly build is part of CI or BM, as it does not fit neatly under either heading.

Build Management Feature Set

Critical to the success of a BM server is its ability to provide traceability. We can decompose the traceability problem into three subparts: traceability between the build and source code; traceability between the build and the artifacts; and traceability between the build and consumers of the build. Consequently, in addition to many of the features already covered for CI servers, the BM server feature set should also include source traceability, artifact traceability, and build identifiers.

[9 to 9.3]

Source Traceability

Source traceability provides the connection between the build and the source code that went into the build. In this section, we need to evaluate whether the BM server has the required features to guarantee this aspect of traceability and whether there are any drawbacks to how the traceability mechanisms are implemented. It is critical that we understand how the BM server establishes a link between the build and the source code used. Given a build, the BM server should make it very explicit what source code was used. There are several ways to accomplish this. One approach is to let the user specify the source code to be built by providing a snapshot or label. If that is how your organization performs authoritative builds, then that provides traceability as long as your SCM guarantees that the contents of the snapshot or label will not change over time. Another approach to establishing traceability between a build and source code is for the BM server to manage the creation of labels, snapshots, or baselines with the contents of the source code used for the build. The BM server should provide this ability our-of-the-box. This is one of those features that require an integration with the SCM, and thus you need to ensure that the BM server includes the required integration features. One caveat with this approach is that it may not be very efficient. In some SCMs, the creation of a label, baseline or snapshot is a very long running process that can take up to several hours for modestly sized projects. Thus, while this approach does provide traceability it may not always be the optimal approach to take.

An approach to traceability that is very effective is to have the BM server obtain the source code from the SCM based on a timestamp. This approach does assume that the SCM allows source code to be retrieved based on a timestamp and that the SCM will in fact produce the same source code given the same timestamp no matter when the request is made. Fortunately, these assumptions hold true for most SCMs currently in use. The benefit of this approach is that it guarantees traceability without the costs associated with using labels, baselines, or snapshots.

[10 TO 10.3]

Artifact Traceability

Because BM deals almost exclusively with the production of build artifacts, the BM server should include an integrated artifact management system that ensures traceability between a build and the generated artifacts. A server that does not have integrated artifact management makes it difficult to trace the artifacts back to a particular build. In such cases artifacts must be placed in an external storage medium and a link between the build and the artifacts must be made using a naming convention, a spreadsheet, etc. An integrated artifact management system can provide additional benefits, such as artifact hashing at the place and time the artifact is created. The artifact hash can then be used at any time that the artifact is consumed to verify that the artifact has not been corrupted during transmission (transport) and that it has been tampered with while in storage. Solutions that do not provide an integrated artifact management system need to provide these missing features via additional configurations and integrations with scripts or third-party tools.

The artifact management system should be integrated with the dependency management system to provide a single, simple interface to the artifacts produced by a build, whether those artifacts are being consumed as dependencies by another build or simply as artifacts. This type of integration can eliminate the need to integrate with multiple external tools using scripts and manually configured steps that complicate every build execution.

[11 TO 11.2]

Build Identifiers (Stamps)

Build identifiers (stamps) provide a way to unambiguously identify a build using a human friendly string. This is the piece that provides traceability between the build and its consumers. This is the piece that allows you to ask questions such as: what build is currently running in production? Without build identifiers, even if we did have perfect traceability between source code and artifacts, we would still have a lot of ambiguity when we try to use a build or their artifacts.

The build identifier (stamp) typically follows a format that has been adopted by the organization. This format may be something as simple as “x.y.z” or it may be more complicated such as “WIN-32-2008-02-18-01”. There is quite a bit of variation as to how organizations use stamps. For starters, some organizations only use build numbers, others use only version numbers, and yet others use a combination of version numbers with build numbers. When evaluating BM servers, you need to make sure that your specific requirements for the build identifier are going to be met.

Keep in mind that the build identifier may be a piece of information that is provided by another system. For example, the SCM may have a unique identifier for the source code being provided to a build based on change list number or the global repository revision number. The BM server should include features that allow the build identifier to be based data coming from other systems.

Enterprise Build Servers

Thus far, the focus has been on evaluating features that apply to teams, workgroups, and enterprises; however, with trends in today's software development space, the build server, either for CI or BM, plays an increasingly important role to the enterprise. With this, a new breed of enterprise-class BM servers, building on the core feature set, has emerged to address the needs and requirements of the large, distributed organization. To meet demand, vendors now offer enterprise-class servers; however, some offerings are scaled up versions of team and workgroup servers, while others do not offer the rich feature set that the enterprise requires. The repercussions of this can be great: Without a thorough investigation of available products, and without an understanding of what features to look for, the enterprise risks choosing a build server that will not perform up to expectation.

While there is a lot of noise about "enterprise" readiness, our experience has led us to believe that there are essentially four sets of features required at the enterprise level. These are in addition to all the features that we have covered thus far and they include: distributed builds, data storage, IT governance, and scalable user interface.

[12] Distributed Builds

One of the main differences between team or workgroup server and enterprise ones is scale. While a team server may be required to handle a load of 20 builds per day, an enterprise server may need to handle several thousand builds in the same time period. The typical approach to solving this scalability problem is to adopt a distributed model under which work (builds) can be farmed out to agents running on additional machines. The typical architecture uses a hub and spoke pattern where a central server provides features such as the configuration and build portals and the distributed agents provide the CPU and other resources.

[12.1] **Agent Discovery.** In large installations with tens and even hundreds of agents, features that focus on management of the agents begin to take on importance. For example, can the server discover agents, or does an administrator have to configure each agent manually. While this is not a huge point, it certainly does help when additional agents need to be brought online quickly.

[12.2] **Agent Upgrade.** A much bigger feature handles the propagation of automatic upgrades to agents. When faced with manually upgrading agents across your entire build farm (e.g., when a new version of the software comes out), the importance of this feature becomes acute. Does the server you are evaluating support automated agent upgrades, or are you going to have to manually upgrade each agent?

[12.3] **Agent Selection.** The server should allow agents to be aggregated into groups, and the selection of agents for use by builds should be flexible enough to allow dynamic allocation of agents to build jobs at run-time. Such dynamic allocation means that build jobs do not have to be configured to run on a specific agent but can be configured to run on any agent that meets the selection criteria. There are numerous benefits of such a dynamic approach, as it allows the server to balance the load among agents sending build jobs to agents that meet all the prerequisites and have the least load on them, thus achieving faster turn around time for builds.

Also, the dynamic agent selection allows the server to work around any agent outages. If an agent goes down and is no longer available, the server can run a build job on the next available agent that meets all the selection criteria.

You need to ensure that the agent selection mechanism in the server you are evaluating is going to support the type of behavior that you need. Can you select agents based on the type of operating system? Can you select agents based on the version of a library or SDK installed? Can you use multiple selection criteria in aggregate?

[12.4] **Split Build Across Multiple Agents.** It is only natural that as we introduce distributed build systems, our quest for faster turn-around times make us ask the question whether we can speed up the build by using multiple machines. This is a very natural and a very good question. Unfortunately, there is a particularly high amount of disinformation on this topic. A lot of vendors claim speed improvements that are only achievable in selected situations, and they are not explicit in letting you know that it is likely not possible to speed up your build significantly in most other instances.

[12.5] **Concurrent Builds.** There are several factors that affect the potential for a speed improvement in your build resulting from distributed builds. The primary factor is the programming language that you are using. Some languages benefit more from concurrent or distributed compilation than others. For example, C/C++ can benefit greatly from distributed compilation, while Java and .Net will not see any impact on compilation speed from distribution. This does not mean that your Java build cannot be sped up by using a distributed build farm; it just means that the compilation portion of the build will not be impacted.

It is common to run unit tests in a distributed manner and gain almost linear performance increase from that strategy. Also, other portions of the build, such as producing automated documentation, can be run in parallel to the compilation. So performance improvements are possible, and you should ensure that the server you are evaluating will enable you to achieve enhanced speed. You should definitely dig into any claims being made on this point, as there are a lot unsubstantiated assertions in this area.

[12.6] **Supported Platforms.** The last main concern when managing a build farm is perhaps the most fundamental. Can the tool actually run on all the machines that you need it to run on? What are all the platforms, and operating systems used in your organization? What about the test machines? Are any of them 64 bit? These concerns are fundamental but often ignored until late in an evaluation process. Understand what you will demand of a potential tool early in the process.

[13 to 13.2] Data Storage

Any enterprise-class solution is going to store configuration as well as run-time data in a relational database. You should ensure that a database back-end is in fact used and that it is used efficiently. Efficient usage of a relational database uses the database for storing the type of data for which the relational database technology has been optimized and should avoid storing other types of data in the database. Therefore, while it may be appealing to store large log files in the database, doing so may severely impact performance.

Another point that needs to be kept in mind about database storage is that the choice of database should be flexible, and allow the server to utilize the database of choice for your team. If a server ships with an embedded version of MySQL and your organization is an Oracle shop, is that a risk you are willing to take?

[14] IT Governance

While IT governance plays a major role within the enterprise, the control that needs to be exacted over the development process needs to be tempered with flexibility, allowing teams to work in the most productive manner. Because the enterprise has hundreds, or thousands, of developers and projects, access to administrative functions (e.g., the ability to change source code or build another team's project) must be controlled. But if the build server does not provide the flexibility to determine individual user permissions, teams may be restricted in how they do business.

During the evaluation process, the enterprise that seeks governance and flexibility will be presented with many options for consideration ... some better than others. Asking a vendor if their build server provides role-based security, auditability, and integrations with authentication and authorization tools will increase the chances that the enterprise selects the build server that best meets their needs.

[14.1] **Role-based Security.** Role-based security has become the standard way of implementing flexible authorization systems. Users are assigned roles and roles are assigned permissions to resources such as projects, build configurations, and other resources. For example, a developer may be permitted to build a project his or her team is working on, but only view a related project. This allows teams to know what others are doing while protecting the code base from inadvertent or worse, even nefarious changes.

[14.2] **Traceability and Auditability.** Compliance has always been an important topic at the enterprise level, and more so recently. In addition to the separation of roles provided by role-based security, traceability and auditability enables an organization to determine what process was invoked, by whom, when, why, and with what parameters. The server you are evaluating should provide the feature set required to answer the questions above.

When a build is requested, the server should capture request information such as the user requesting the build, the date and time of the request, as well as any properties that were provided to the build process. Notes can shed some light on the reason for the build. In addition, integrations with tools such as issue tracking tools and change management tools can produce reports that list the issues or change requests associated with the build; thus helping to provide an answer to the why as well.

[14.3] **Integration with Authentication System.** Because each enterprise has a significant investment in existing tools, choosing a build server that integrates into the ecosystem will maximize both existing investment as well as that of the new build server. Though most servers integrate with authentication and authorization tools such as LDAP, Kerberos, SSO, etc., many enterprises often utilize homegrown solutions. When evaluating servers, it is important to understand if and how the build server integrates with these tools, and if the server has the capability to integrate with tools developed inhouse.

[15] Scalable User Interface

A user interface that scales to enterprise needs presents yet another criterion during the evaluation process. One area of the user interface that typically fails first at the enterprise level is the presentation of projects. Most products provide a simple list of projects to the user. But when that list includes hundreds and even thousands of projects, such a list becomes a very ineffective user interface device. Vendors have taken several approaches to solve this problem. The basic approach is to provide user configurable filters for the list. Using this approach the user would enter a regular expression to select and view only the project whose name matches the regular expression. While effective at reducing the number of projects, this approach requires a very strict naming convention that needs to include every aspect that could be included in the selection filter. This solution leads to project names that are long and rather cryptic, as they need to encode a lot of information that can be used by the filters.

Another solution is to organize the projects into a hierarchy using a project tree. In using this approach the enterprise may create a hierarchy that models their physical locations, or business units, or any other type of organizational structure. Projects can be placed within folders that make up the hierarchical tree. Permissions to various portions of the tree can be assigned to roles representing project teams, business units, etc. This approach does not require elaborate or cryptic naming conventions and provides a familiar metaphor for organizing large numbers of projects.

Conclusion

While the list of features provided by this paper is not exhaustive, it should provide some indication of the depth of attributes that a successful CI or BM server should contain. We focused on highlighting the most important features in each category, and on explaining the implications regarding the quality of implementation for each feature. Armed with this information, you should be able to sift through the noise and focus on information that is meaningful and will ultimately lead a successful deployment. If key words such as Service Oriented Architecture, Agile, Continuous Integration, Build Management, Deployment Automation, Release Management, Dependency Management, and Compliance start to show up in your business lexicon, we hope this paper will be a good reference.

We would like to receive any feedback that you may have, as well as any relevant experiences that you can share. You can contact us at info@urbancode.com.

About Urbancode

Urbancode has been providing Application Lifecycle Automation solutions to customers in the financial, banking, insurance, software, and high-tech industries since the 2001 release of Anthill. With AnthillPro 3.4 ALA, Urbancode offers unique innovations in Continuous Integration, Build and Dependency Management, Deployment Automation, Test Orchestration, and Release Management for organizations using .NET, Java, and/or native technologies. Urbancode's products integrate with SCM, issue tracking, change management, test automation and IDE tools, and are designed with a distributed architecture that scales to enterprise levels.

Urbancode's commitment to complete lifecycle automation, management, visibility, and traceability, along with AnthillPro's technical depth, form our comparative advantage. Savvy, market leading companies, including 25 of the Fortune 100, rely on Urbancode for Continuous Integration, Build Management, Deployment Management, and Application Lifecycle Automation.

Founded in 1996, Urbancode is headquartered in Cleveland, Ohio.

Continuous Integration

1 SCM Integration	
1.1 support for SCM	
1.1.1 support for your usage pattern	
1.1.2 changelog report	
1.2 triggers	
1.2.1 polling trigger	
1.2.2 event trigger	
1.3 quiet period	
1.3.1 race condition free	
2 Build Tool Integrations	
2.1 integrates with your build tools	
2.2 flexibility to run additional commands	
3 Quality Determination	
3.1 testing tool integrations	
3.1.1 aggregating results	
3.1.2 results in data warehouse	
3.1.3 results available for logical decisions	
3.1.4 metrics	
3.2 test coverage tool integrations	
3.2.1 aggregating results	
3.2.2 results in data warehouse	
3.2.3 results available for logical decisions	
3.2.4 metrics	
3.3 means to integrate additional reports	
4 Feedback	
4.1 notifications	
4.1.1 mediums	
4.1.2 blame developer	
4.2 build portal (dashboard) with reports	
4.2.1 changelog	
4.2.2 errors clearly identified	
4.3 IDE plugins	
5 Smart Build Queue	
6 Dependency Management	
6.1 inter-project dependencies	
6.1.1 flexible dependency relationships	
6.1.2 transitive dependencies	
6.2 pulling builds	
6.3 pushing builds	
6.4 dependencies on third-party artifacts	
6.5 traceability of artifact usage	
6.6 provisioning artifacts to developers	
7 Ease of Configuration	
7.1 web-based configuration	
7.2 reuse existing configuration in new projects	
7.3 smart configurations	
8 Build Purging Rules	

Build Management

9 Source Traceability	
9.1 specifying source for build	
9.2 tags, baselines, snapshots	
9.3 using timestamps	
10 Artifact Traceability	
10.1 integrated artifact storage	
10.2 artifact hashing	
10.3 integration with dependency management	
11 Build Identifiers (Stamps)	
11.1 flexible pattern	
11.2 based on other systems	

Enterprise Features

12 Distributed Builds	
12.1 agent discovery	
12.2 agent upgrade	
12.3 agent selection	
12.4 split builds across multiple agents	
12.5 concurrent builds	
12.6 supported platforms	
13 Data Storage	
13.1 database back-end	
13.2 choice of databases	
14 IT Governance	
14.1 role-based security	
14.2 traceability and auditability	
14.3 integration with authentication systems	
15 Scalable User Interface	

Using the Checklist

The preceding text is annotated to correspond to items on this checklist. For example, the section on **SCM Integration** is marked with **[1]** and the **Distributed Builds** section is marked with **[12]** in the left hand column. So when using the checklist, it is possible to easily find detailed information about any particular feature. We do not provide an evaluation methodology, such as rating each solution under evaluation on a scale of 0 to 5. That scoring varies by organization, depending on your devilish details, but the framework will hopefully be helpful.

urban{code}

Urbancode, Inc.
2044 Euclid Avenue • Suite 600
Cleveland, Ohio • 44115
P: 216.858.9000 • F: 213.858.9602
www.anthillpro.com