

Automation for the people: Continuous Integration anti-patterns, Part 2

Make your life with CI easier by learning what not to do

Skill Level: Introductory

Paul Duvall (paul.duvall@stelligent.com)
CTO
Stelligent Incorporated

04 Mar 2008

While Continuous Integration (CI) can be extremely effective at reducing risks on a project, it requires a greater emphasis on your day-to-day coding activities. In this second installment of a two-part article on CI anti-patterns, automation expert and co-author of *Continuous Integration: Improving Software Quality and Reducing Risk*, Paul Duvall, continues laying out CI anti-patterns, and more importantly, demonstrates how to avoid them.

In the [first part](#) of this two-part article, I described the following six CI anti-patterns:

- Infrequent check-ins, which lead to delayed integrations
- Broken builds, which prevent teams from moving on to other tasks
- Minimal feedback, which prevents action from occurring
- Receiving spam feedback, which causes people to ignore messages
- Possessing a slow machine, which delays feedback
- Relying on a bloated build, which also delays feedback

About this series

As developers, we work to automate processes for end-users; yet, many of us overlook opportunities to automate our own development processes. To that end, *Automation for the people* is a series of articles dedicated to exploring the practical uses of

automating software development processes and teaching you *when* and *how* to apply automation successfully.

These anti-patterns delay or prevent the benefits you can experience with CI. In this second part, I'll cover five more equally deceiving practices:

- Waiting until the end of the day to commit changes, leading to *Bottleneck Commits*, which typically cause broken builds and frustrated developers
- A build consisting of minimal automated processes, which results in builds that never fail, leading to *Continuous Ignorance* and delaying integration problems
- Hindering build fixes through a preference for *Scheduled Builds*, rather than frequently building software with every code change
- Believing that code *Works on My Machine*, only to discover problems later in other environments
- Failing to remove old build artifacts, which leads to a *Polluted Environment* causing false positive and false negative errors

Once again, to receive the manifold benefits of CI, it pays to understand these anti-patterns — and avoid them.

Stop choking on bottleneck commits

Name: Bottleneck Commits

Anti-pattern: Developers commit code changes prior to leaving for the day, causing integration build errors and preventing team members from going home at a decent time.

Solution: Check-in code frequently throughout the day.

What's an integration build?

An *integration build* is a type of build that checks out source files from a version control repository and is run on a *separate* machine, rather than just a developer workstation.

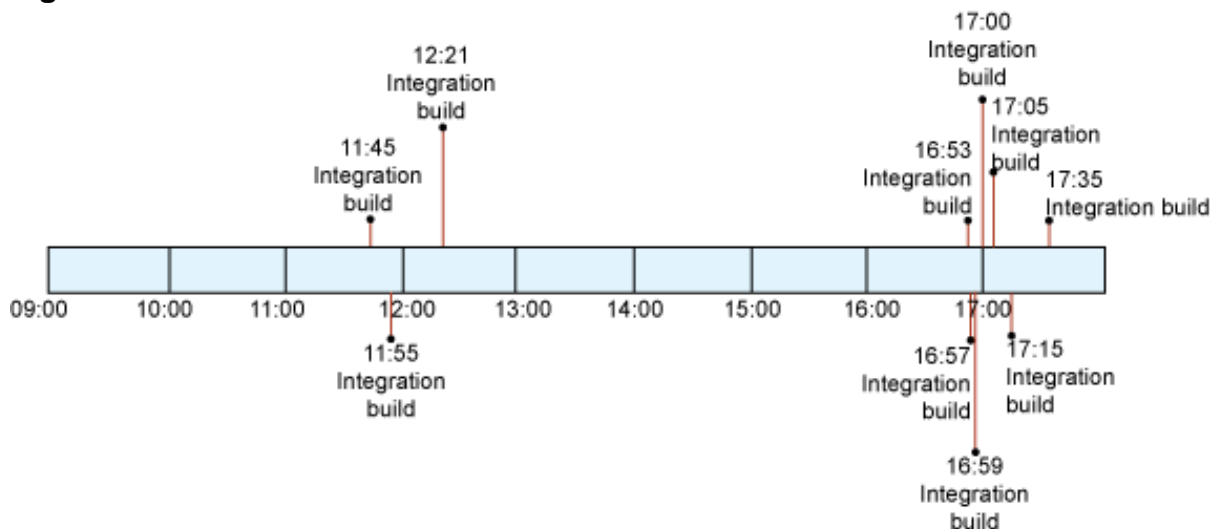
Bottleneck commits are a variation of the *Infrequent Check-in* anti-pattern. It assumes you are checking-in code at least once a day. However, the problem is that everyone is checking in *at the same time*. In a CM Crossroads article, Slava Imeshev describes how the vast majority of build failures occur between 5 p.m and 8 p.m. (and to a lesser extent, the lunch hour). In the "Five-O'Clock Check-In", Imeshev associates this occurrence to a tendency for developers to check-in their code changes toward the end of the day (see [Resources](#)).

It's five o'clock somewhere

The *Five-O'Clock Check-in* is a quick way to lose friends if your team follows the rule that no one goes home until changes are committed and a build is run. Just the wait to submit code can be tedious. Imagine what happens, then, when the build breaks. I'm thinking that there will be a lot of calls home to explain why you'll be late ... again.

Figure 1 illustrates a typical timeline of check-ins for a software development team. Note that repository commits are bunched around meal time and prior to leaving for the day.

Figure 1. Bottleneck commits



The solution to this headache is, of course, to commit changes frequently! With this approach, you'll have smaller, but more frequent, integration builds, and when a build error does occur, it'll likely be small and much simpler to fix.

Five o'clock should have a good connotation, so do yourself and your team a favor by checking-in code frequently and keep five o'clock a happy time for the masses.

Ignorance is not bliss

Name: Continuous Ignorance

Anti-pattern: When a build was successful, everyone assumes the resulting software is working fine. In reality, the limited build consists of compilation and a few unit tests.

Solution: A full integration build is run with every change to the version control repository.

Sometimes teams can be lulled into a false sense of security with one successful build after the other. In fact, an almost sure sign that your team may be applying the *Continuous Ignorance* anti-pattern is when build failures seldom occur. If your integration builds never fail, you probably aren't verifying or validating enough of your code! The less your build is doing, the less *informed* you become about whether the software is actually working as expected.

Test Ignorance

A variation on the Continuous Ignorance anti-pattern occurs when you choose to comment-out a broken test in an effort to force a build to succeed, rather than fixing the source of the issue. Although practicing this technique results in a successful integration build, it is just another delaying tactic used to defer what will likely become a bigger issue later.

Comprehensive builds offer enlightenment

The stack on the left in Figure 2 illustrates an integration build that doesn't do much more than compile source code, package classes into a binary, and deploy software into an operating environment. Of course, this is better than never performing an integration build at all. However, take a look at the stack on the right. Comparing the two stacks should provide insight into problems that can be uncovered through running more processes as illustrated on the right, such as tests and database changes.

Figure 2. Comprehensive builds are your friend



Streamline your builds

By including additional processes like database integration, developer tests (unit, component, functional, and so on), automated code inspections (such as coding standard adherence, cyclomatic complexity, and code duplication checks), and installation distributions, you are better able to determine if your software is actually working earlier in the development cycle. Keep in mind, however, that the more you add to an integration build, the slower feedback will be. Therefore, you'll probably

want to consider creating build pipelines to run slower-running processes after the initial commit build — doing so will create opportunities for quicker feedback and provide for a more flexible mechanism for software validation.

Reschedule your scheduled builds

Name: Scheduled Builds

Anti-pattern: Builds are run daily, weekly, or on some other schedule, but not with every change.

Solution: A build is run with *every* change applied to a source code repository.

Continuous Integration is about integrating software assets often — and by often, I mean *any time code changes*. This is, as far as I know, the absolute quickest way to spot issues early. Spotting issues early is a good thing. For one, it saves you money. For another, it actually results in the ability to release better code more often.

The problem with a scheduled build is that it's run regardless of whether you committed a change to a repository. This means that builds could be run that provide no value, because either nothing changed since the last build, or there were so many changes that any resulting problems become difficult to untangle.

Performing CI effectively requires a proactive emphasis — when a build fails, it's imperative that problems are fixed immediately. The nature of a scheduled build discourages proactive action and tends to lead to an "I'll fix it when I can get to it" approach, which is the antithesis of CI.

Changing a scheduled build to be more frequent is as easy as properly configuring a CI server. For instance, Listing 1 demonstrates a CruiseControl script that polls a version control repository every two minutes. If a change is discovered, CruiseControl runs a build.

Listing 1. Build with every change

```
...
<schedule interval="120">
  <ant anthome="${cc.ant.dir}" buildfile="build-${project.name}.xml"/>
</schedule>
...
```

Don't get me wrong — builds that run on a scheduled basis can be useful in certain scenarios. For instance, running load and performance tests could be done nightly because of the length of time the build takes to run. As a general rule, however, if all builds are run on a less frequent basis than with every code change, you are limiting your chances of discovering issues early.

But, it works on my machine!

Name: Works on My Machine

Anti-pattern: You run a private build that works on your machine, only to discover later that the changes don't work in other environments.

Solution: The team uses an integration build machine that runs a build with every change committed to a version control repository.

Imagine you make a code change and run your build (through an IDE or Ant), and after all is working as expected you commit your changes to a version control repository. A few days later, someone deploys the code into another environment and you're informed that your changes aren't working. You then launch the software application on your workstation, and, lo and behold, everything works perfectly leading you to exclaim "but, it works on my machine!"

Has this ever happened to you? If it hasn't, get ready, it probably will. There can be many reasons for this wicked occurrence, but the typical culprits are that you forgot to add a new file into a version control repository or some specific configuration on your machine isn't configured on another environment.

The world beyond your desktop

Name: IDE-only Build

Anti-pattern: You run a private build using an IDE (that works locally) on your workstation only to discover things don't work in another environment.

Solution: Create a build script and commit it into a version control repository. Run this same build script with every change.

There's nothing wrong with using an IDE to write code or to create build scripts. IDEs make your job more efficient. However, if build processes are so tightly coupled to the IDE that you're unable to run an integration build without having the IDE installed in the deployment environment, you've got a problem. This is because it's unusual that operations teams will install an IDE in staging and production environments. If they were to do this, it could require a manual configuration (which could lead to inconsistencies across environments and build errors) of the IDE in order to run an integration build. What's more, having the build *depend upon the IDE* could delay finding configuration problems until late into the development process as dependencies are removed from the IDE in the staging and production environments.

To solve the IDE-only build problem, all you must do is create a build script (you can still easily use it through your IDE). This same build script can then be the master

mechanism for integration builds — regardless of environment.

Myopia and the developer

Name: Myopic Environment

Anti-pattern: Assuming that because a build works in one environment, it'll work in any environment.

Solution: Define build behavior in build targets; moreover, externalize environment-dependent data into `.properties` files.

The Myopic Environment anti-pattern is reflective of a developer's mentality of falsely assuming that if their code works in one environment, their job is done. Therefore, to combat this mind-set, CI systems should assume nothing (within reasonable constraints). Reducing assumptions is as easy as removing platform-specific constraints and making them replaceable, such as through property files.

If a build needs to run in both Windows® and Linux® environments, it stands to reason that it'll fail in Linux if there are references to the `C` drive. If a build references a machine-dependent environment variable (for example, `GLOBUS_LOCATION`), it'll fail if these variables have not been set up in another environment. In these cases, all you must do is replace those references to variables that can be replaced at build time.

The Ant XML snippet in Listing 2 demonstrates how to include a `.properties` file that contains environmental values:

Listing 2. Referring to environment properties in a build script

```
<property file="${basedir}/TEST.properties" />
```

The properties in Listing 3 show data that is particular to a deployment environment. Note how this file reduces assumptions in the build itself by externalizing information like database connection values, Web container locations, host names, and authentication information, for instance.

Listing 3. Defining data in a properties file

```
database.host.name=integratebutton.com
database.username=myusername
database.password=mypassword
database.port=3306
jboss.home=/usr/local/jboss/server/default
jboss.temp.dir=/tmp
jboss.server.hostname=integratebutton.com
jboss.server.port=8080
jboss.server.jndi.port=1099
```

All build behavior should be contained in targets (such as Ant targets). Any data that is referenced more than once in the same build script should be defined in a property. Any data that varies from one machine to the next must be externalized into a .properties file. If you follow this simple advice, you will achieve maximum flexibility with minimal headaches.

Clean up your environment

Name: Polluted Environment

Anti-pattern: An incremental build is run to save time; however, an older artifact (from a previous build) produces a false positive (or false negative) build.

Solution: Clean previously built artifacts prior to running a build. Baseline server and configuration information.

Cleaning up before yourself

There are few more frustrating things than running a build that fails, only to discover that it's because artifacts from a previous build are still resident. Such artifacts may include recently deployed WAR files, incorrect JAR versions, or database updates. On the flip-side, and even more troubling, is when you're duped into a false sense of security because previously built artifacts are in the build environment causing a successful build.

I can't stress enough that it's important to put your environment into a known state prior to any relevant build activities. In fact, I like to run what I call a *scorched earth* policy when building software. Listing 4 is a simple example of removing previous log directories, distributions, and reporting files using Ant's `delete` task. This drastically reduces the chances of false positives or false negatives.

Listing 4. Removing directories via Ant

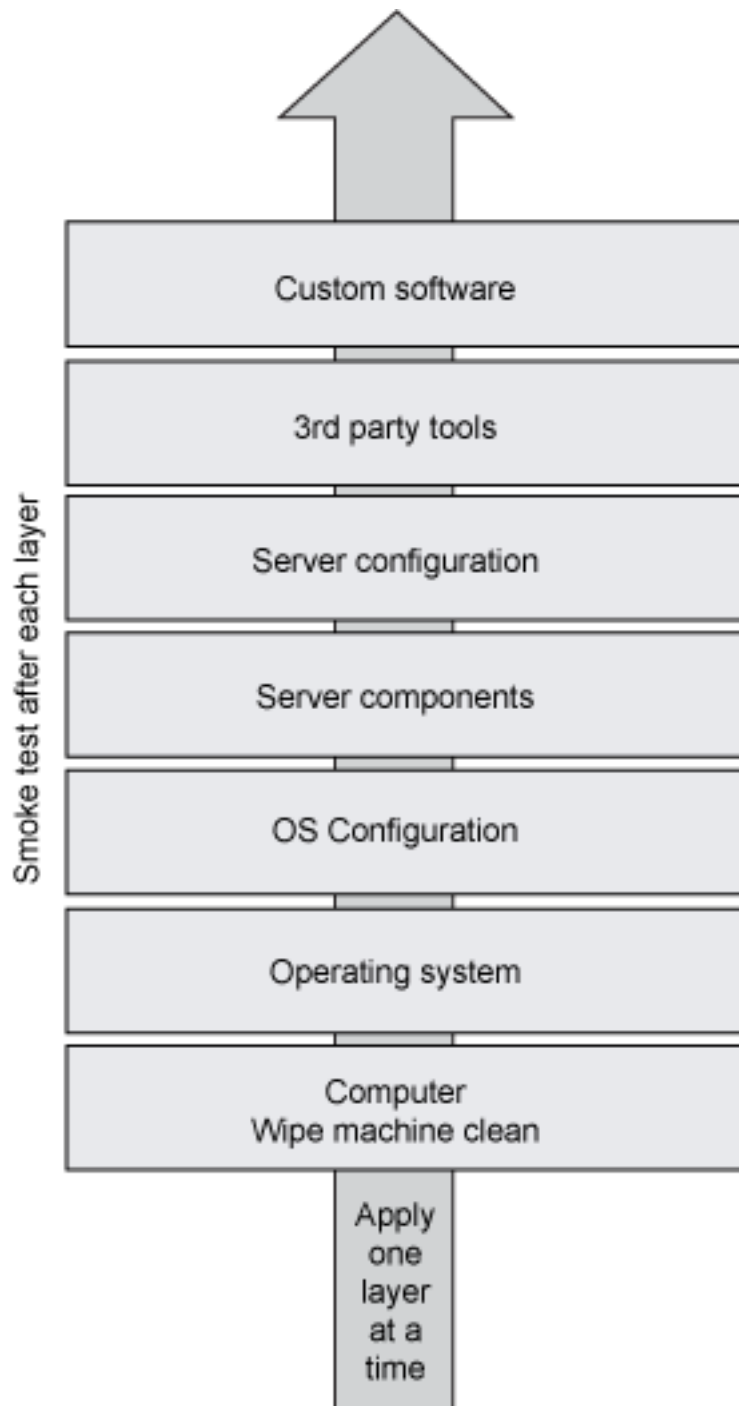
```
<target name="clean">
  <delete dir="${logs.dir}" />
  <delete dir="${dist.dir}" />
  <delete dir="${reports.dir}" />
  <delete file="cobertura.ser" />
</target>
```

The code above is simple — scorching an environment may also include removing old class files and previously deployed EAR/WAR files, putting the database into a known state (for instance, dropping and recreating database tables), reinitializing a classpath, and avoiding the use of environment variables.

Baseline deployment environments

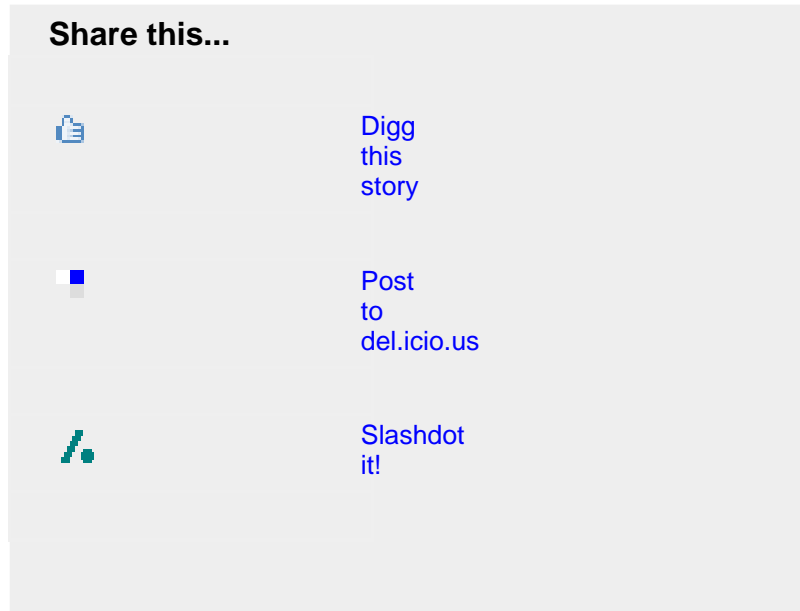
"Scorching" can lead you to consider more advanced techniques such as setting the state of the operating environment where you're running integration builds. To baseline these environments, each component and configuration item is removed and re-applied using automation. As compared to simply removing old files, this can be a more difficult undertaking depending on the environment complexity. However, it can be incrementally applied. A component could be something like a database, Web or file server, or some proprietary software. A configuration item may be an environment variable or a change in the database configuration, such as its memory allocation. Consistency is key. Each environment you run your builds in should have a similar configuration. By baselining deployment environments, you are better equipped to find the cause of a particular problem. Figure 3 illustrates some of the components that may need to be scorched and re-applied in a similar manner in each deployment environment.

Figure 3. Scorching a deployment environment



In my experience, one of the most significant benefits with scorching your environment is to provide a way to troubleshoot problems more expeditiously. Baseline comparative environments gives you the ability to compare apples to apples, so to speak, and ultimately enables more rapid fixes should problems arise between environments.

Succeed with Continuous Integration



I hope you've seen that while Continuous Integration is a great practice to use in your development projects, you can enjoy more of its benefits by avoiding certain anti-patterns. There are good reasons why development teams use some of these practices, but they can lead to the use of anti-patterns. For instance, sometimes there are good reasons to run a scheduled build. Further, the effective practice of committing code often can actually lead to bottlenecks, but that doesn't make frequent commits a bad practice. Remember, anti-patterns aren't bad practices, per se, but they can be bad approaches in certain situations.

Resources

Learn

- ["Remove the smell from your build scripts"](#) (Paul Duvall, developerWorks, October 2006): Refactor your build to make it easier to maintain.
- [Continuous Integration: Improving Software Quality and Reducing Risk](#) (Paul Duvall et. al, Addison-Wesley Signature Series, 2007): Read the reference on everything related to Continuous Integration.
- ["Avoiding Continuous Integration Build Breakage Patterns"](#) (Slava Imeshev, CM Crossroads, November 2005): Keep the five o'clock hour a fun one!
- ["Continuous Integration"](#) (Martin Fowler, martinowler.com): Fowler's seminal article on Continuous Integration.
- ["Is Pipelined Continuous Integration a Good Idea?"](#) (infoq.com, September 2007): Leading CI advocates weigh in on the build pipeline.
- [AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis](#) (Brown et. al, Wiley, 1998): An excellent book describing anti-patterns related to software development.
- ["Continuous Integration anti-patterns, Part 1"](#) (Paul Duvall, developerWorks, December 2007): Read about the first six anti-patterns.
- [Automation for the people](#) (Paul Duvall, developerWorks): Read the complete series.
- [developerWorks](#): Hundreds of articles about every aspect of Java programming.

Get products and technologies

- [Ant](#): Run your Java™ build with Ant.

Discuss

- [Improve Your Code Quality discussion forum](#): Regular developerWorks contributor Andrew Glover brings his considerable expertise as a consultant focused on improving code quality to this moderated discussion forum.
- [Accelerate development space](#): Andrew Glover also hosts this one-stop portal for all things related to developer testing, continuous integration, code metrics, and refactoring.

About the author

Paul Duvall

Paul Duvall is the CTO of [Stelligent Incorporated](#), a consulting firm and thought leader in helping development teams optimize Agile software production. He is the co-author of the Addison-Wesley Signature Series book, [Continuous Integration: Improving Software Quality and Reducing Risk](#) (Addison-Wesley, 2007). He also contributed to the [UML 2 Toolkit](#) (Wiley, 2003) and the [No Fluff Just Stuff Anthology](#) (Pragmatic Programmers, 2007).