Continuous Integration:

Patterns and Anti-Patterns

UPDATED BY **DAVID POSIN**
ORIGINAL BY **PAUL DUVALL**

CONTENTS

ABOUT CONTINUOUS INTEGRATION

Continuous Integration (CI) is about minimizing code conflicts and maximizing efficiency. It describes an automated process designed to build a project whenever the codebase changes. Continuous Integration starts with developers committing code to a shared repository one or more times a day. It ends with the CI system successfully building the project from scratch.

CI has been around for a long time, and during that time, the communities around it have developed best practices. Explaining these best practices can be done by describing patterns (i.e., a solution to a problem in a particular context) and anti-patterns (i.e., generally ineffective approaches to solving a problem). Anti-pattern solutions tend to produce adverse effects.

WHAT IS CONTINUOUS INTEGRATION?

Conventionally, the term "Continuous Integration" refers to the "build and test" cycle. Individual developers merge their code into one shared project master branch. CI servers rebuild the project from scratch every time a code merge happens.

Conversations about Continuous Integration tend to couple CI with Continuous Delivery (CD). CD differs from CI by referring to the movement of code from one environment to another, such as development to QA to User Acceptance Testing. CI servers will usually do CD as well. The two concepts are inexorably linked since they work hand in hand. There are other DZone Refcardz that you should consider downloading if you wish to understand Continuous Delivery better, such as, "Preparing for Continuous Delivery" or "Continuous Delivery: Patterns and Anti-Patterns in the Software Lifecycle". This Refcard is going to focus specifically on CI patterns, although there will occasionally be some overlap since the two are so closely related.

WHY DOES CONTINUOUS INTEGRATION MATTER?

Continuous Integration benefits any organization that implements it correctly. Some of the key benefits are:

- **Better Quality Code** - Code that makes it into the project's master branch is of a consistent quality. The automated CI system will perform code checking and code linting as part of the merge and build process.
- **Better Tested Code** - Unit tests, end-to-end tests, and code coverage reports can be run automatically to ensure all tests pass and code coverage does not slip.
- **Production Snapshot** - Building from scratch means that the build on QA is going to be the same on UAT, which is going to be the same as production.
- **Early Error Detection** - Code that passes all the linting, unit tests, and e-2-e tests can still fail to build for unexpected reasons. A build failure is identified immediately after the code is merged in, making it easy to identify the broken commit(s).
- **Project Confidence** - All of this adds up to increased confidence in the product from developers, managers, and customers.

CI needs to be implemented properly to reap these benefits. Everything starts from the point of change. Every time a change is merged into the master branch, then a build job must be run.

PATTERNS AND ANTI-PATTERNS

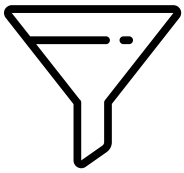
Patterns, and corresponding anti-patterns, are discussed in the following sections. According to Google trends, the most searched for version control tool is Git and the most commonly searched CI software is Jenkins.

Continued on next page

The Modern Solution to Infrastructure Monitoring and Troubleshooting

No complex setup. No waiting. Just answers.

Rapid7 InsightOps combines log management with IT asset search for a new approach to infrastructure and application monitoring and troubleshooting. It takes only minutes to setup and is free to try.



Centralize

Easily centralize data across your infrastructure, from system logs to IT assets.

Monitor

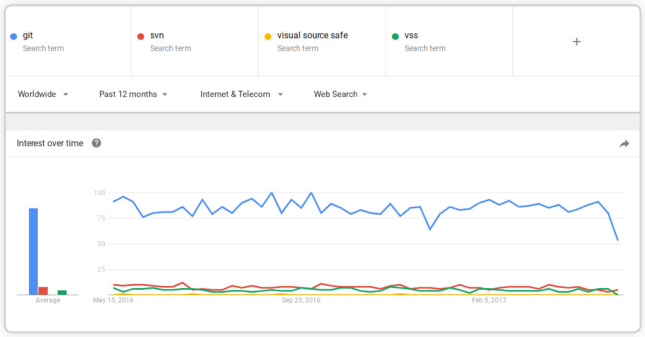
Monitor your IT environment for anomalies, key metrics, and critical events.

Answer

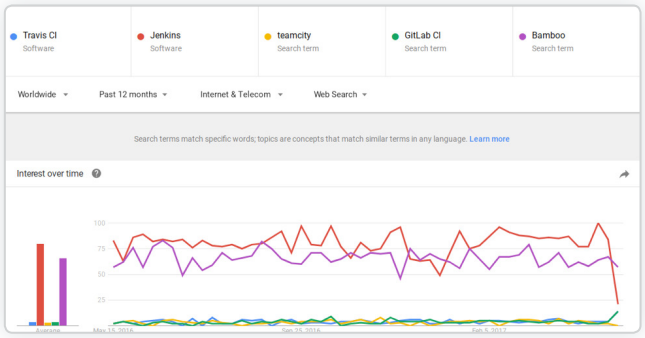
Get the answers you need, when you need them, to resolve critical issues and maintain uptime.

Start a Free Trial

Google Trends



Google Trends



Where examples are provided, we will use the Git Command Line Interface commands or Git Plugin hooks for Jenkins. This Refcard will use those two technologies for examples based solely on search popularity.

BUILD SOFTWARE AT EVERY CHANGE

One of the most important patterns to remember when talking about CI is to build the project every time a change is merged into master. Traditionally, the project is built on an arbitrary schedule, such as nightly or every weekend. Scheduled builds are an anti-pattern for CI.

One of the goals of building after every code merge is to immediately identify when problems occur. If the newly merged code causes the build process to fail, then developers know which section of code to examine. If builds are scheduled and an error occurs, the exact code change might not be obvious, and fixing it could require significant digging.

A Continuous Integration system should be set up with:

- A shared version control repository (i.e. Git).
- An automated build script or CI server configuration to run when the repository has changed.
- Some sort of feedback mechanism (such as e-mail or chat software).

Pattern	Run a software build with every change applied to the Repository.
Anti-patterns	Scheduled builds, nightly builds, building periodically, building exclusively on developer's machines, not building at all.

VERSION CONTROL [1]

Versioning is one of the central pillars of CI. A good version control system (VCS) will maintain a core functioning codebase. Developers can then build off that main codebase by creating code branches to add features, fixes, and patches without affecting anyone else. Developer branches can then be merged into the main code branch, called mainline or master, when complete.

Here are some important best practices to consider when working with a version control system:

- **Private Workspace** - Developers should be working on their own machines (real or virtual) with local copies of the repository. Developers should not be working over file systems that allow them to share the same code files, or on the machine serving as a repository host (if the repository is not cloud-based).

- **Repository** – In line with the point above, all code should be hosted in a repository. No successful Continuous Integration plan will work with a file system hosted project.

- **Master** - The main branch from which builds are run should be the master, or mainline, branch. This branch should be heavily protected. Developers should be able to merge code into the master branch but should not be able to commit code directly. The master branch will host your project's main history and milestone builds. No code should ever be added to it directly.

- **Branching Policy** - All teams working on a CI process should have an agreed-upon branching policy. Developers should branch off of master using a naming convention agreed upon by the team. Code should be merged into master from individual developer branches through pull requests, or some other agreed-upon mechanism. Branches should be pushed to the repository just like master so they can be shared if needed.

TASK-LEVEL COMMIT

Most modern version control software integrates this best practice into its core functionality. A developer moves code into the VCS when they have made enough progress to consider saving it. Usually the process involves adding the changed code files to a commit, and then committing them with a relevant message.

For example, in Git, a task-level commit uses the following commands:

```
git add -A
git commit -m "message" (i.e., git commit -m "added file object class")
git push
```

[1] Addison-Wesley, Software Configuration Management Patterns, 2003, Berczuk and Appleton.

Pattern	Organize source code changes by task-oriented units of work and submit changes as a Task-Level Commit.
Anti-patterns	Keeping changes local to development for several days and stacking up changes until committing all changes. This often causes build failures or requires complex troubleshooting.

LABEL BUILD

When master has reached an important development or release milestone, give it a name to mark that code state. Typically, this will be, or will incorporate, the release version number using semantic versioning (major.minor.patch, i.e. 1.10.23).

Git uses the term "tagging" instead of "labeling". To create a tag for the current build, use the following commands:

```
git tag -a annotation -m "message" (i.e., git tag -a
v1.2.12 -m "version 1.2.12")
git push remote --tags
```

Pattern	Tag or Label the build with unique name so that you can refer to run the same build at another time.
Anti-patterns	Not labeling builds, using revisions or branches as "labels."

BUILD MANAGEMENT

Code being merged into master should cause the CI server to automatically build the project, including the new changes. There should be no developer or DevOps engineer involvement required.

AUTOMATED BUILD

There should be hooks in the version control system or polling in the CI system to force a new build when master changes. It is important that builds happen after every merged code change so a breaking commit can be identified immediately. Developers should not be expected to kick off CI builds manually to avoid impacting development progress.

Setting up an automatic build between Git and Jenkins uses the following procedure. Inside the project's .git directory, update the hooks/post-receive file to send the repository to your CI tool (in this case Jenkins). Your CI tool should be installed with a Git plugin to accept the curl.

```
curl http://yourserver/git/notifyCommit?url=<URL of
the Git repository>[&branches=branch1[,branch2]*]
[&sha1=<commit ID>]
```

Pattern	• Automate all activities to build software from a source without manual configuration. • Create build scripts that will be executed by a CI system so that software is built at every change.
Anti-patterns	Not labeling builds, using revisions or branches as "labels."

BUILD PRACTICES

PRE-MERGE BUILD

The automated build discussed above occurs on the CI server using shared resources. VCS systems can be configured to perform a fast, stripped-down build locally first to pre-check the code.

The procedure in Git works using the project's .git directory. Update the hooks/pre-push file to run the CLI command for your project's build/compilation/bundling tool. For example, a JavaScript or Node project might use a tool called Gulp:

gulp <my project>

Pattern	Verify that your changes will not break the integration build by performing a pre-merge build—either locally or using Continuous Integration.
Anti-patterns	Checking in changes to a version-control repository without running a build on a developer's workstation.

CONTINUOUS FEEDBACK

The results of a build are of special importance to the developer submitting the revised code. It is important that they are aware of the results as soon as they are available. Positive and negative results are both important and developers should be trained to watch for all feedback before moving on to other tasks. The method(s) of providing feedback will vary depending on your organization's infrastructure. Methods to consider include:

- Email
- Hipchat
- Slack
- SMS
- Web Push Notifications
- Campfire
- Any infrastructure tool with extensions that your organization uses

Pattern	Send automated feedback from the CI server to development team members involved in the build.
Anti-patterns	Sending minimal feedback that provides no insight into the build failure or is non-actionable. Sending too much feedback, including to team members uninvolved with the build. This is eventually treated like spam, which causes people to ignore messages.

EXPEDITIOUS FIXES

Mistakes will happen and occasionally code will make it into master that breaks the build. The responsible pattern is to be

ready to handle and resolve the problem immediately. The worst possible way to handle a failure in a build is to ignore it, or expect it to be resolved in a future build. You should consider taking all the following steps:

- **Fix Broken Builds Immediately** - Although it is the team's responsibility, the developer who recently committed code must be involved in fixing the failed build. It is possible the problem was a result of a lack of knowledge, so it is a good idea to have a seasoned developer available to assist if needed.
- **Always Pull Master and Build** - Developers should pull the latest code into their branch from master before pushing committed code. After pulling master into their own branch, they should run unit tests and build locally to ensure nothing pulled from master breaks their code. This also allows the developer a chance to fix conflicts that result from the merge before the merge gets to master.
- **Don't Pull Broken Code** - If master is broken, notify the team. Developers should avoid pulling master into their own branch while it is broken. Development time could be wasted by other developers struggling with bad code that will be changed shortly.

Pattern	Fix build errors as soon as they occur.
Anti-patterns	Allowing problems to stack up (build entropy) or waiting for them to be fixed in future builds.

DEVELOPER DOCUMENTATION

The build process is an excellent opportunity to generate documentation for your source code. Developers tend to dislike writing documentation manually, and keeping documentation up to date manually can be time-consuming. The preferred approach is to incorporate documentation into your code, and then having the build process generate documentation from the code. This keeps documentation up-to-date and does not create more work for the development team.

Pattern	Generate developer documentation with builds based on checked-in source code.
Anti-patterns	Manually generating developer documentation. This is both a burdensome process and one in which the information becomes useless quickly because it does not reflect the checked-in source code.

BUILD CONFIGURATION

INDEPENDENT BUILD

Builds should happen the same way on all machines that run them. A build on a developer machine should run the same procedure as the CI server. Therefore, train developers to not use the IDE build process. Instead, the IDE can be configured to run the required build scripts so that building can still happen from the IDE. Every project should include its own build scripts so it can be built from anywhere it is being worked on.

Pattern	Create build scripts that are decoupled from IDEs, but can be invoked by an IDE. These build scripts will be executed by a CI system as well so that software is built at every change.
Anti-patterns	Relying on IDE settings for Automated Build. Build cannot run from the command line.

SINGLE COMMAND

Running a project build should be as simple as possible. It is best to have a simple CLI command that can run everything required for a build in the correct order. This ensures that both developers and servers use the exact same code in the exact same order. A single command-invoked build script can also be kept up to date with the current state of the project.

Build, compile, and testing phases can be time consuming for a developer. In order to support the development team, provide flags on the CLI command to limit the build process to fit their needs. For example, a developer might be updating a class and only needs to compile the code. They are not at a point where they need to test and build the whole project. The script could take a flag, such as --compileonly, to only perform the compilation process, but this way there are not individual commands for developers to know. Everything goes through the same single CLI command.

Pattern	Ensure all build processes can be run through a single command.
Anti-patterns	Requiring people or servers to enter multiple commands and procedures in the deployment process, such as copying files, modifying configuration files, restarting a server, setting passwords, and other repetitive, error-prone actions.

DEDICATED RESOURCES

CI builds of master should be performed on servers (real or virtual) that are only tasked with building the project. These dedicated machines should have sufficient resources to build the project smoothly and swiftly to limit developer downtime. Performing builds on a dedicated machine ensures a clean environment that doesn't introduce unexpected variables. Clean builds give a certain degree of reassurance that the project will build successfully when being deployed to other environments.

Pattern	Run master builds on a separate dedicated machine or cloud service.
Anti-patterns	Relying on existing environmental and configuration assumptions (can lead to the "but it works on my machine problem").

EXTERNALIZE AND TOKENIZE CONFIGURATION

Configuration information that is specific to a machine or deployment environment should be a variable in any build and configuration scripts. These values should come from the build process so they can be build or environment specific. Files that use this information should use tokens so that the build process can replace them with actual values. For example, a build might be supplied a hostname, then any configuration file that needs hostname should use $hostname for that value. The build process will go through all config files and replace $hostname with the correct value every time the server is built. This lets the build process create the project on as many different machines as possible.

Pattern	• Externalize all variable values from the application configuration into build-time properties. • Use tokens so the build process knows where to add variable values.
Anti-patterns	Hardcoding values in configuration files or using GUI tools to do the same.

DATABASE

Databases are the cornerstones of all modern software projects. No project of any scale beyond a prototype can function without some form of database. For this reason, databases should be included in the Continuous Integration process. They should be treated with the same extensibility and care as project software code.

SCRIPTING DATABASE CHANGES

All changes made to a database during development should be recorded via database scripts. The CI process can then run scripts as the project is built. It is an anti-pattern to expect any manual manipulation of a database during the build process. A database for the project should be able to be migrated to new changes regardless of timing or platform.

Pattern	All changes made to a database during development should be recorded into database scripts that can be run on every database on every platform hosting the project (including developer machines, see below).
Anti-patterns	Expecting database administrators to manually compare databases between platforms for changes, or to create on-off scripts that are only good for updating a single platform.

DATABASE SANDBOX

Every instance of the project should have its own version of the database with a relevant set of data. This should include development machines, build machines, testing machines, testing servers, and QA machines. No individual or server working with the project should have to worry about the integrity of their data, or the integrity of some other entity's data, while coding, building, and testing.

This is true of schema, but not necessarily data. The data for each environment should be a subset of the whole, and scrubbed of sensitive information. For example, a developer should have access to a lightweight version of the database with a very small subset of records. However, the data they do have shouldn't have real social security numbers, addresses, etc. Use discretion and ahere to any relevant regulations when deciding what data to use for database population.

The CI process should include a way to import the data correctly into the database. Any data import or manipulation should be scripted so it can be performed via command line. Developers, testers, and build machines shouldn't have to know the intricacies of the data. The build process in particular will need a command line command to call during the build.

Pattern	• Create a lightweight version of your database (only enough records to test functionality) • Use a command line interface to populate the local database sandboxes for each developer, tester, and build server • Use this data in development environments to expedite test execution
Anti-patterns	Sharing development database.

UPDATE SCRIPTS STORED IN THE VCS

All scripts to perform database and data operations should be stored in the version control system being used by the codebase. Scripts should be named and/or annotated to refer to their appropriate version number to simplify automation. Keeping old versions is useful for doing multiple scripts if necessary, and to maintain a history of changes.

Pattern	Store the scripts for updating the database and its data in the version control system with the code and annotate appropriately
Anti-patterns	Storing update scripts in an alternative location, i.e. a shared file server

TESTING AND CODE QUALITY

The sometimes rapid pace of Continuous Integration can feel daunting in the beginning, but it becomes comfortable very quickly. One of the reasons for that is testing. Testing and code quality validation are a massively important part of CI. Tests running with every build ensures that nothing has been broken

and that the code is maintaining superior quality. CI cannot be successful without a robust testing method. Without testing, CI can become chaotic.

AUTOMATED TESTS

Tests should run with every build. The build scripts described in the sections above should include running tests against all code in the project. Tests should be as comprehensive as possible and can include unit tests, end-to-end tests, smoke tests, or UI tests. Testing should be done on all new code. Tests for new code should be a requirement of a successful build. All code being merged into the project should be required to meet an appropriate code coverage level. All modern test running suites will have some form of coverage reporter that can be tied into the build process. Any code submitted without sufficient test coverage should fail.

Pattern	• Notify team members of code aberrations such as low code coverage or the use of coding anti-patterns. • Fail a build when a project rule is violated. • Use continuous feedback mechanisms to notify team members.
Anti-patterns	• Deep dive reviews of every code change. • Manually calculating or guesstimating code coverage.

AUTOMATED SMOKE TEST

Smoke tests are a subset of tests used to confirm the functionality of the most important elements of a project. They function as gatekeeper to confirm that building, full testing, or QA can continue. A suite of well-designed smoke tests can save QA personnel time and effort by checking the most likely candidates for failure first.

This is also useful for Continuous Deployment. Smoke tests can be designed to check functionality most sensitive to changes in the environment. It is an easy way to confirm that the deployment will work.

Pattern	Create smoke tests that can be used by CI servers, developers, QA, and testing as a pre-check to confirm the most important functionality as they work, or before committing resources to a full build.
Anti-patterns	• Manually running functional tests. • Forcing QA to run the full suite before every session • Manually checking deployment sensitive sections of the project

ABOUT THE AUTHOR

DAVID POSIN has been involved in the Information Technology Industry for two decades. Fifteen years of that time was spent consulting with many companies in a wide range of industries to build solid technology stacks and robust application architectures. David has watched the Cloud and the World Wide Web grow from their infancy, and now spends every day fully entrenched in those worlds. Currently, David builds high-performance web applications and offers professional technical writing services.