<https://martinfowler.com/articles/continuousIntegration.html>

Continuous Integration Best Practices:

1. Maintain a code repository
2. Automate the build
3. Make the build self-testing
4. Everyone commits to the baseline every day
5. Every commit (to baseline) should be built
6. Keep the build fast
7. Test in a clone of the production environment
8. Make it easy to get the latest deliverables
9. Everyone can see the results of the latest build
10. Automate deployment

Best Practice 1: Maintain a Code Repository

The Vision

There should be a revision control system for the project’s source code. All artifacts required to build the project should be placed in the repository. The system should be buildable from a fresh checkout and not require additional dependencies. Branching should be minimized – the mainline / trunk should be the place for the working version of the software, and all changes should preferably be integrated there.

The Reality – Avg. Score: A (Good)

This is a principle the community has managed to implement consistently. Most developers today commit to a central repository.

However, with regard to branching, the theory does not hold up to reality. In giants like Google, Facebook or Amazon, everyone commits to the main branch and there is ongoing development of one main codebase. But in most organizations, it’s simply not possible to perform all changes on one branch:

* It’s often necessary to stop work on a release, stabilize and ship it, while another part of the team starts working on the next release.
* Customers will typically demand that you support, and fix bugs for old versions of the software, or they will not accept new versions of the software “pushed” to them.

These typical scenarios, and many others, often force software companies to develop on several branches in parallel.

What Can We Do to Improve?

Having seen so many CI scenarios in which branching was unavoidable, we can’t agree that “branching is evil”. Branching is often necessary in real-life development, and recognizing this, we need tools and processes that will help us manage it correctly.

The classic challenge of branching is “merge hell” in which much development work goes into synchronizing the changes of the new release branch with the main branch. This problem can be minimized by making releases shorter, as has become customary in agile.

Another issue is that multiple branches require multiple builds, sometimes with different automatic build configurations. Merging the release branch back into the trunk is not just a matter of merging the code – you also need to make sure that new build artifacts, changes to build scripts, etc. are merged. This is a weak spot, which often results in broken builds and unanticipated integration issues.

Best Practice 2: Automate the Build and Deployment

The Vision

A single command should have the capability of building the system. Automation of the build should include automating the integration, and preferably running a script after the build ends, which deploys the software into a production-like environment.

The build script should not only compile binaries, but also generate documentation, website pages, statistics and distribution media (such as Red Hat RPM or Windows MSI files).

The Reality – Score B (Satisfactory)

Today’s build systems are capable of building complex software components automatically, generating documentation, distribution media etc., and also “deploying”, but only in the sense of placing the files on a designated network location. Does this pass as an “automatic build” according to CI best practices? Maybe, but only for an application running on one machine. As soon as additional machines are involved, the paradigm starts to break down.

Take a simple client-server architecture, with a server and two flavors of a client, one running on Linux and one on Windows. One way to automate this build is to use a tool like Jenkins to build all three components (server, client A and client B), and place them somewhere on the network. Then, someone has to manually take each of the three tiers and install it on the relevant server – but this breaks the requirement for automation.

A more sophisticated method is to use three instances of a build server like Jenkins to build each of the components, and save each of them to the correct machine in your testing or production environment. But this introduces more challenges:

* Now you have three different build configurations to maintain. Given that branching is common, this could quickly multiply to dozens of builds, even in this small setup. There is a cost to managing dozens of build configurations. But beyond that cost, because of the complexity and the typically large number of artifacts (some of which are shared between the builds and some of which are unique), there will inevitably be errors that lead to broken builds – again requiring manual intervention.
* Deployment is not as trivial as placing files in a directory. For example, if your server component is based on the .NET framework, it will need to be pre-installed on the machine before the server itself is installed. And what if the target machine has a previous version of .NET? Or if the previous version of the server has not been uninstalled? And how can you make sure that the target machine for each of the clients actually has the right version of Windows or Linux? Some of this can be automated with tools like Chef, but very often, manual labor still creeps in.
* The build script is expected to run all tests automatically. What if you want to test each of the clients (Windows and Linux) against the server? Presumably the same test script will run on each client. If you deploy both clients and run the tests automatically, chances are that each client will make changes on the server, and these changes can create possible conflicts for the other client, which expects a certain initial state. What is needed in this case is to run a test on the first client, restore server to initial state, then manually run a test on the second client. This is not supported by today’s build tools, and so it will typically be done manually.

We’ve seen **numerous “points of failure”** in which even a simple 3-part build will not be fully automated. Now imagine a web application with 3 web servers, a load balancer, an app server and a cluster of 3 database servers – in this slightly larger setup, all these issues are made exponentially worse. With today’s build tools it will be almost impossible to automate this 8-part build, with full integration, deployment and testing, except with a heavy investment in scripting and manual maintenance, that most organizations will not wish to undertake.

What Can We Do to Improve?

A typical way to deal with this complexity is to “do less stuff”: run less automatic testing, install less components on CI and test environments, and handle production deployments manually, to make sure all components play well together. But all of these compromises take us further away from the original vision of CI.

To eliminate the manual work that creeps into the build process, automated build systems need to learn a few new tricks:

* Build systems need to be able to manage both variations of the same build process, and shared components between different build processes. Otherwise you quickly end up with dozens or hundreds of build configurations, many of which are close duplicates or are inter-dependent.
* Build systems need to be aware of deployment semantics – what are the characteristics of the machines we are deploying to, and what does each component need in order to run properly.
* Build systems need to orchestrate deployments and testing in a more sophisticated manner – to enable scheduling and a sensible sequence of deployment and testing across multiple software components.

*Does your build system do all of that?*

Best Practice 3: Make the Build Self-Testing

**The Vision**

Once the code is built, all tests should run automatically to validate that the software behaves as the developers expect it to.

**The Reality – Score B (Satisfactory)**

In most CI setups, builds are run at several stages, for different types of builds: development builds are ran by developers while making changes to the code; CI builds run frequently to synchronize changes between developers; UAT/staging builds are ran to test an entire system on pre-production; production builds; and more.

In reality, running “all tests” in **all**these types of builds is impractical:

* Many automated tests require deployment of the software into a production-like environment, which is labor-intensive to deploy numerous times a day, because deployment is complex and test environments are scarce.
* Some tests take a long time to run – browser/UI automation tests for example. Therefore, it’s very common that teams resort to nightly, or even less frequent builds, for the more complex testing.
* Some parts of the system are rarely touched and so will not be tested on a regular basis – for example, tests of the database schema are probably not needed if you’re not changing the schema in the current version.

So while running “all tests” in all builds would be ideal, reality dictates that we **prioritize testing** and run certain types of tests in certain stages of the dev/test cycle. There are a few ways to achieve prioritized testing:

* Having one automated build process that runs only the basic set of tests, and trigger additional tests manually.
* Having several build configurations: one for a basic set of tests, one including some more tests, up to the full production build which includes all tests. These sets of tests create numerous builds that are using almost the same artifacts, creating duplication and room for error. Not to mention maintenance-hell. Also, this approach limits flexibility to some extent. For example, what if a developer wants to execute a specific automated UI test to see if a change broke the interface? That will have to wait for later in the dev/test cycle.

What Can We Do to Improve?

Given that some tests simply take a long time, or require complex setup, it’s inevitable that we prioritize tests and rarely run a build with “all tests”. The question is how difficult it is to set up this prioritization, and is it flexible enough to support decision making at different development stages.

To improve our ability to prioritize tests, and thus enable us to run close to all the tests required in the current stage of development, the following innovations are needed:

* Build systems need to recognize that one build might have different testing configurations – so you should be able to run the build for System A with “development tests”, “CI tests”, “UAT tests”, “Production tests” etc.- without having to duplicate build configurations numerous times.
* Build systems need to provide flexibility to run tests as needed – a developer or build engineer should be able to “make an exception” and run additional tests, or less tests, balancing the need for information with the acceptable speed for the current build.

Best Practice 4: Fast Builds with Most Recent Changes

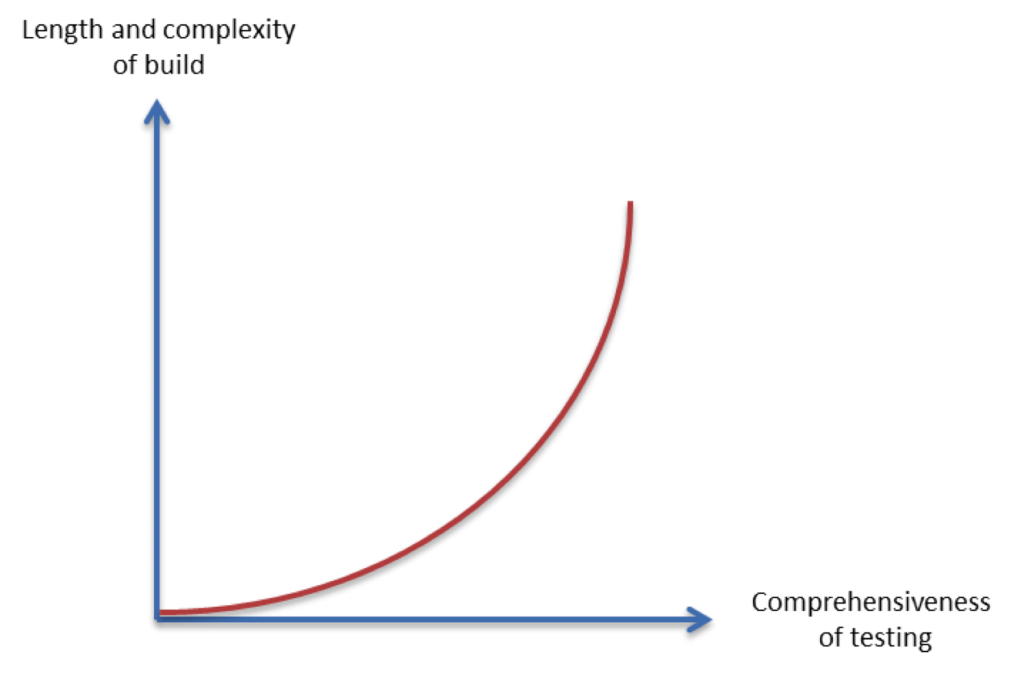
The Vision

To reduce the number of conflicting changes, developers should commit to the baseline every day. Checking-in a week’s worth of work runs the risk of conflicting with others’ code and can be very difficult to resolve. By committing at least once a day, conflicts are quickly discovered and typically focus on a limited part of the system, and team members can communicate about their recent changes.

As part of this process, every commit should be built, to review the impact of the changes on the system by the entire team. Builds should run at least nightly, preferably more frequently. To facilitate this, builds should run fast.

The Reality – Score B (Satisfactory)

The idea behind this principle is that the development team requires rapid feedback: what is the impact of the changes we have made, and have they broken anything? The problem is that in order to really know if something is broken, you need to run a full battery of tests on production-like environments. The more tests you run, and the more production-like your environment setup is, the longer it will take the build to run, and the more manual work you’ll need in order to run it on that environment.

  
As the graph shows, the length and complexity of the build increase exponentially with the comprehensiveness of tests you want to run. This places severe limitations on your ability to get high quality feedback frequently, as the best practice requires.

We all run automated tests in almost all our builds. But those are typically the tests that are the **easiest to automate**. In our experience, tests that are harder to automate – and therefore are done much less frequently – are the ones that more closely simulate the real production environment, and so are most valuable for team. The earliest in the development stages that the team can receive this feedback- the better..

What Can We Do to Improve?

This one is pretty obvious – automate much more of the testing process and make it run faster. If we could run the complex tests, or even some of them, several times a day during a dev or CI build, the feedback provided to the dev and QA teams would improve dramatically. This can be done by a heavy investment in scripting – but those scripts will require tedious, and complex, maintenance every time your software or deployment environment change. There is a need for automated systems that are easier to set up, will execute tests and deployments on demand with no manual intervention, and easily adapt to changes.

Best Practice 5: Test in a Clone of the Production Environment

The Vision

Testing environment that differs from the Production one could lead – and often does lead – to software that tested successfully in QA but fails in Production. However, building a replica of a production environment is cost prohibitive. Instead, the pre-production environment should be built to be a scalable version of the actual production environment, to both alleviate costs while maintaining technology stack composition and nuances.

The Reality – Score C (Poor)

The concept of a “pre-production environment” is a useful one, but it breaks down in all but the simplest systems. A typical example is the database: in production you run with an Oracle database on a dedicated server, but in early dev and test everyone works with a MySQL database running locally. If you’ve heard the phrase “it worked on my machine” – that means someone is developing or testing in a different environment from the one used in production, which is almost always the case.

The problem is further compounded when the software runs on several platforms, or customers use different platforms to access it (e.g. various mobile devices, operating systems and browsers). To discover defects effectively we would need to simulate **all** the production environments used by end users, requiring multiple build configurations and complex deployments – introducing manual effort. If they can fully test the customer’s production environments, most organizations will do so only very close to shipping the product.

What Can We Do to Improve?

It will never be possible to fully simulate the production environment in all stages of the dev/test cycle, if it has any degree of complexity. But we can take steps to close the gap from “my machine” to that of the end user. CI systems need to be able to automatically deploy on more complex environments. Imagine that any developer could push a button and create three VMs on the fly, that have the exact same configuration as the production environment, with their three-tier web application installed on those VMs and integrated correctly.

Best Practice 6: Make it Easy to Get Latest Deliverables

The Vision

Builds should be readily available to stakeholders and testers, to reduce the amount of rework necessary when rebuilding a feature that doesn’t meet requirements. Early testing should be done to reduce the chances that defects survive to deployment; early discovery of errors can reduce the amount of work necessary to resolve them.

The Reality – Score A (Good)

Most development teams that have adopted CI use a central Artifact Repository. Builds are pushed to the repository automatically, and all team members have access to the latest builds. In our experience this works well – as long as the build runs relatively fast and subject to the extensiveness of tests that are run in the build. Artifact repositories make it possible for developers and testers to easily access the builds, discover problems early and react to them quickly.

An important consideration is what happens in response to an issue with the build. When a build breaks, does everyone stop what they are doing and focus on solving the problem? As teams adopt CI, they gradually learn to adapt work processes to the feedback received from the build system. It’s important to recognize that CI is not software in a box, but incorporates a way of thinking and eorking that people need to adapt to. It sometimes takes time until an entire team learns to react immediately, and in a coordinated manner, to issues discovered in the latest build.

Best Practice 7: Everyone Can See the Latest Build

The Vision

It should be easy to find out whether the build breaks and, if so, who made the relevant change.

The Reality – Score A (Good)

Most development teams have measures in place to push build feedback to team members so developers are aware if the build is red or green. Many organizations use email alerts, instant messaging, monitors in central locations that show the results of the latest builds, and so on.

One important issue to note is that, as previously discussed, the more complex testing and deployments often tend to be done manually. If something is done manually, it’s not part of the build feedback loop. The QA team who ran the test will notify dev about the results, but it likely won’t be done automatically or be part of the closed-loop process for monitoring, notifications and shared visibility.

CI Best Practices Made Easier: Electric Flow

[ElectricFlow](http://electric-cloud.com/products/electricflow/) is a suite of applications for build automation, build/test acceleration and automated deployment. ElectricFlow, powered by ElectricCommander automation platform, helps solve many of the gaps between the vision of Continuous Integration and the de-facto practice of CI in most organizations, that stem from the difficulty in managing, automating and orchestrating complex build processes.

CI Best Practice:  Automate the Build and Deployment

**Limitations of Current Build Systems:**

* Multiple build configurations
* Limited deployment tools
* Unable to orchestrate multi-component tests

**With ElectricFlow:**

ElectricFlow allows complete control over software delivery pipeline right from developer check-ins to end-user using the product. It allows automating your build processes, orchestrating test automation, deploying to various environments.

It offers unique features like:

* Pre-flight builds for gated checkins to source code systems. Pre-flight allows developers to do CI locally before checking in code. This significantly improves your chances of getting green builds.
* Tight integrations with many source code systems
* Orchestration across tool-chain like defect tracking systems, unit testing systems, code analysis tools, test automation suites, configuration management systems, cloud providers, monitoring systems, service management tools, etc.
* Workflow automation regardless of development methodology or process
* Centralized management of software artifacts
* Visibility and reporting for project predictability and fewer process errors
* Security and scalability to support geographically distributed teams

CI Best Practice:  Make the Build Self-Testing

**Limitations of Current Build Systems:**

* Cannot automatically test on production-like environments
* Tests that take a long time are run infrequently
* No flexibility to request additional tests, or remove tests, on demand

**With ElectricFlow:**

ElectricFlow allows to deploy applications automatically across various environments and then run appropriate test suite to automate testing. In fact the success rate of the builds could be significantly improved by using the pre-flight capability which allows developers to run CI locally even before checking in the code and only on successful build check the code in. In addition with integrations with various unit testing and test automation suite it is possible to get complete control over automated testing.

CI Best Practice:  Fast Builds with Most Recent Changes

**Limitations of Current Build Systems:**

* Only the simplest tests are automated
* Complex tests involve manual work and are done less frequently
* Important feedback from production-like testing occurs close to shipping

**With ElectricFlow:**

ElectricFlow allows to checkout only the changed files instead of checking out the entire set of files which can improve the build times significantly.

CI Best Practice:  Test in a Clone of Production

**Limitations of Current Build Systems:**

* Difficult to create a “pre-production environment” automatically
* Difficult to test on numerous production environments

**With ElectricFlow:**

ElectricFlow allows to model various environments and applications. With the model based deployment philosophy the system allows to define deployment processes such that deployments could be carried out across various environments consistently and repeatedly. This ensures that both the code as well as process to deploy are tested well before they get into production.

CI Best Practice:  Everyone Can See the Latest Build

**Limitations of Current Build Systems:**

* Manual deployment and testing is not part of build feedback loop

**With ElectricFlow:**

ElectricFlow has heavy emphasis on visibility and governance. ElectricFlow allows to consolidate statistics from various tools, build outputs etc all in a centralized place. This ensures complete visibility of the pipeline and various metric associated. In addition it also has a built-in artifact repository to store all build outputs for easy access.