Continuous integration

In software engineering, **continuous integration** (**CI**) is the practice of merging all developer working copies to a shared mainline several times a day.[1] Grady Booch first proposed the term CI in his 1991 method,[2] although he did not advocate integrating several times a day. Extreme programming (XP) adopted the concept of CI and did advocate integrating more than once per day – perhaps as many as tens of times per day.[3]

Contents

Rationale

The main aim of CI is to prevent integration problems, referred to as "integration hell" in early descriptions of XP. CI is not universally accepted as an improvement over frequent integration, so it is important to distinguish between the two as there is disagreement about the virtues of each.

In XP, CI was intended to be used in combination with automated unit tests written through the practices of test-driven development. Initially this was conceived of as running and passing all unit tests in the developer's local environment before committing to the mainline. This helps avoid one developer's work-in-progress breaking another developer's copy. Where necessary, partially complete features can be disabled before committing, using feature toggles for instance.

Later elaborations of the concept introduced build servers, which automatically ran the unit tests periodically or even after every commit and reported the results to the developers. The use of build servers (not necessarily running unit tests) had already been practised by some teams outside the XP community. Nowadays, many organisations have adopted CI without adopting all of XP.

In addition to automated unit tests, organisations using CI typically use a build server to implement *continuous* processes of applying quality control in general — small pieces of effort, applied frequently. In addition to running the unit and integration tests, such processes run additional static and dynamic tests, measure and profile performance, extract and format documentation from the source code and facilitate manual QA processes. This continuous application of quality control aims to improve the quality of

software, and to reduce the time taken to deliver it, by replacing the traditional practice of applying quality control *after* completing all development. This is very similar to the original idea of integrating more frequently to make integration easier, only applied to QA processes.

In the same vein, the practice of continuous delivery further extends CI by making sure the software checked in on the mainline is always in a state that can be deployed to users and makes the deployment process very rapid.

Workflow

When embarking on a change, a developer takes a copy of the current code base on which to work. As other developers submit changed code to the source code repository, this copy gradually ceases to reflect the repository code. Not only can the existing code base change, but new code can be added as well as new libraries, and other resources that create dependencies, and potential conflicts.

The longer a branch of code remains checked out, the greater the risk of multiple integration conflicts and failures when the developer branch is reintegrated into the main line. When developers submit code to the repository they must first update their code to reflect the changes in the repository since they took their copy. The more changes the repository contains, the more work developers must do before submitting their own changes.

Eventually, the repository may become so different from the developers' baselines that they enter what is sometimes referred to as "merge hell", or "integration hell",[4] where the time it takes to integrate exceeds the time it took to make their original changes.

Continuous integration involves integrating early and often, so as to avoid the pitfalls of "integration hell". The practice aims to reduce rework and thus reduce cost and time.[5]

A complementary practice to CI is that before submitting work, each programmer must do a complete build and run (and pass) all unit tests. Integration tests are usually run automatically on a CI server when it detects a new commit.

History

The earliest known work on continuous integration was the Infuse environment developed by G.E. Kaiser, D.E. Perry and W.M. Schell.[6]

In 1994, Grady Booch used the phrase continuous integration in Object-Oriented Analysis and Design with Applications (2nd edition)[7] to explain how, when developing using micro processes, "internal releases represent a sort of continuous integration of the system, and exist to force closure of the micro process."

In 1997, Kent Beck and Ron Jeffries invented Extreme Programming (XP) while on the Chrysler Comprehensive Compensation System project, including continuous integration.[8] Beck published about continuous integration in 1998, emphasising the importance of face-to-face communication over technological support.[9] In 1999, Beck elaborated more in his first full book on Extreme Programming.[10] CruiseControl, one of the first open-source CI tools,[11] was released in 2001.

Common practices

This section lists best practices suggested by various authors on how to achieve continuous integration, and how to automate this practice. Build automation is a best practice itself.[12][13]

Continuous integration – the practice of frequently integrating one's new or changed code with the existing code repository – should occur frequently enough that no intervening window remains between commit and build, and such that no errors can arise without developers noticing them and correcting them immediately.[14] Normal practice is to trigger these builds by every commit to a repository, rather than a periodically scheduled build. The practicalities of doing this in a multi-developer environment of rapid commits are such that it is usual to trigger a short time after each commit, then to start a build when either this timer expires, or after a rather longer interval since the last build. Note that since each new commit resets the timer used for the short time trigger, this is the same technique used in many button debouncing algorithms [ex:[15]]. In this way the commit events are "debounced" to prevent unnecessary builds between a series of rapid-fire commits. Many automated tools offer this scheduling automatically.

Another factor is the need for a version control system that supports atomic commits, i.e. all of a developer's changes may be seen as a single commit operation. There is no point in trying to build from only half of the changed files.

To achieve these objectives, continuous integration relies on the following principles.

Maintain a code repository

This practice advocates the use of a revision control system for the project's source code. All artifacts required to build the project should be placed in the repository. In this practice and in the revision control community, the convention is that the system should be buildable from a fresh checkout and not require additional dependencies. Extreme Programming advocate Martin Fowler also mentions that where branching is supported by tools, its use should be minimised.[14] Instead, it is preferred for changes to be integrated rather than for multiple versions of the software to be maintained simultaneously. The mainline (or trunk) should be the place for the working version of the software.

Automate the build

A single command should have the capability of building the system. Many build tools, such as make, have existed for many years. Other more recent tools are frequently used in continuous integration environments. Automation of the build should include automating the integration, which often includes deployment into a production-like environment. In many cases, the build script not only compiles binaries, but also generates documentation, website pages, statistics and distribution media (such as Debian DEB, Red Hat RPM or Windows MSI files).

Make the build self-testing

Once the code is built, all tests should run to confirm that it behaves as the developers expect it to behave.[16]

Everyone commits to the baseline every day

By committing regularly, every committer can reduce the number of conflicting changes. Checking in a week's worth of work runs the risk of conflicting with other features and can be very difficult to resolve. Early, small conflicts in an area of the system cause team members to communicate about the change they are making.[1] Committing all changes at least once a day (once per feature built) is generally considered part of the definition of Continuous Integration. In addition performing a nightly build is generally recommended. These are lower bounds; the typical frequency is expected to be much higher.

Every commit (to baseline) should be built

The system should build commits to the current working version to verify that they integrate correctly. A common practice is to use Automated Continuous Integration, although this may be done manually. Automated Continuous Integration employs a continuous integration server or daemon to monitor the revision control system for changes, then automatically run the build process.

Keep the build fast

The build needs to complete rapidly, so that if there is a problem with integration, it is quickly identified.

Test in a clone of the production environment

Having a test environment can lead to failures in tested systems when they deploy in the production environment because the production environment may differ from the test environment in a significant way. However, building a replica of a production environment is cost prohibitive. Instead, the test environment, or a separate pre-production environment ("staging") should be built to be a scalable version of the production environment to alleviate costs while maintaining technology stack composition and nuances. Within these test environments, service virtualization is commonly used to obtain on-demand access to dependencies (e.g., APIs, third-party applications, services, mainframes, etc.) that are beyond the team's control, still evolving, or too complex to configure in a virtual test lab.

Make it easy to get the latest deliverables

Making builds readily available to stakeholders and testers can reduce the amount of rework necessary when rebuilding a feature that doesn't meet requirements. Additionally, early testing reduces the chances that defects survive until deployment. Finding errors earlier can reduce the amount of work necessary to resolve them.

All programmers should start the day by updating the project from the repository. That way, they will all stay up to date.

Everyone can see the results of the latest build

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

Automate deployment

Most CI systems allow the running of scripts after a build finishes. In most situations, it is possible to write a script to deploy the application to a live test server that everyone can look at. A further advance in this way of thinking is continuous deployment, which calls for the software to be deployed directly into production, often with additional automation to prevent defects or regressions.[17][18]

Costs and benefits

Continuous integration is intended to produce benefits such as:

- Integration bugs are detected early and are easy to track down due to small change sets. This saves both time and money over the lifespan of a project.
- Avoids last-minute chaos at release dates, when everyone tries to check in their slightly incompatible versions
- When unit tests fail or a bug emerges, if developers need to revert the codebase to a bug-free state without debugging, only a small number of changes are lost (because integration happens frequently)
- Constant availability of a "current" build for testing, demo, or release purposes
- Frequent code check-in pushes developers to create modular, less complex code

With continuous automated testing benefits can include:

- Enforces discipline of frequent automated testing
- Immediate feedback on system-wide impact of local changes
- Software metrics generated from automated testing and CI (such as metrics for code coverage, code complexity, and feature completeness) focus developers on developing functional, quality code, and help develop momentum in a team

Some downsides of continuous integration can include:

- Constructing an automated test suite requires a considerable amount of work, including ongoing effort to cover new features and follow intentional code modifications.
  - Testing is considered a best practice for software development in its own right, regardless of whether or not continuous integration is employed, and automation is an integral part of project methodologies like test-driven development.
  - Continuous integration can be performed without any test suite, but the cost of quality assurance to produce a releasable product can be high if it must be done manually and frequently.
- There is some work involved to set up a build system, and it can become complex, making it difficult to modify flexibly.[19]
  - However, there are a number of continuous integration software projects, both proprietary and open-source, which can be used.
- Continuous Integration is not necessarily valuable if the scope of the project is small or contains untestable legacy code.
- Value added depends on the quality of tests and how testable the code really is.[20]
- Larger teams means that new code is constantly added to the integration queue, so tracking deliveries (while preserving quality) is difficult and builds queueing up can slow down everyone.[20]
- With multiple commits and merges a day, partial code for a feature could easily be pushed and therefore integration tests will fail until the feature is complete.[20]
- Safety and mission-critical development assurance (e.g., DO-178C, ISO 26262) require rigorous documentation and in-process review that are difficult to achieve using Continuous Integration. This type of life cycle often requires additional steps be completed prior to product release when regulatory approval of the product is required.

See also

- Application release automation
- Build light indicator
- Comparison of continuous integration software
- Continuous design
- Continuous testing
- Multi-stage continuous integration
- Rapid application development

References

1. "Continuous Integration" (https://www.thoughtworks.com/continuous-integration).
2. Booch, Grady (1991). *Object Oriented Design: With Applications* (https://books.google.com/?id=w5VQAAAAMAAJ&q=continuous+integration+inauthor:grady+inauthor:booch&dq=continuous+integration+inauthor:grady+inauthor:booch). Benjamin Cummings. p. 209. ISBN 9780805300918. Retrieved 18 August 2014.
3. "Embracing Change with Extreme Programming". CiteSeerX 10.1.1.617.9195 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.617.9195).
4. Cunningham, Ward (5 August 2009). "Integration Hell" (http://c2.com/cgi/wiki?IntegrationHell). *WikiWikiWeb*. Retrieved 19 September 2009.
5. "What is Continuous Integration?" (https://aws.amazon.com/devops/continuous-integration/). *Amazon Web Services*.
6. G. E. Kaiser, D. E. Perry and W. M. Schell, "Infuse: fusing integration test management with change management," [1989] Proceedings of the Thirteenth Annual International Computer Software & Applications Conference, Orlando, FL, USA, 1989, pp. 552-558. doi: 10.1109/CMPSAC.1989.65147
7. Booch, Grady (December 1998). "Object-Oriented Analysis and Design with applications (2nd edition, 15th printing)" (http://www.cvauni.edu.vn/imgupload_dinhkem/file/pttkht/object-oriented-analysis-and-design-with-applications-2nd-edition.pdf) (PDF). *www.cvauni.edu*. Retrieved 2 December 2014.
8. Fowler, Martin (1 May 2006). "Continuous Integration" (http://martinfowler.com/articles/continuousIntegration.html). *martinfowler.com*. Retrieved 9 January 2014.
9. Beck, Kent (28 March 1998). "Extreme Programming: A Humanistic Discipline of Software Development" (https://books.google.com/books?id=YBC5xD08NREC&lpg=PA1&ots=MSoAk8Bl14&dq=%22Extreme%20Programming%3A%20A%20Humanistic%20Discipline%20of%20Software%20Development%22&pg=PA4#v=onepage&q=%22Extreme%20Programming:%20A%20Humanistic%20Discipline%20of%20Software%20Development%22&f=false). *Fundamental Approaches to Software Engineering: First International Conference, FASE'98, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, 28 March – 4 April 1998, Proceedings, Volume 1*. Lisbon: Springer. p. 4. ISBN 9783540643036.
10. Beck, Kent (1999). *Extreme Programming Explained* (https://books.google.com/?id=G8EL4H4vf7UC&pg=PA97&dq=continuous+integration+intitle:Extreme+intitle:Programming+intitle:Explained#v=onepage&q=continuous%20integration%20intitle%3AExtreme%20intitle%3AProgramming%20intitle%3AExplained&f=false). ISBN 978-0-201-61641-5.
11. "A Brief History of DevOps, Part III: Automated Testing and Continuous Integration" (https://circleci.com/blog/a-brief-history-of-devops-part-iii-automated-testing-and-continuous-integration/). *CircleCI*. 2018-02-01. Retrieved 2018-05-19.
12. Brauneis, David (1 January 2010). "[OSLC] Possible new Working Group – Automation" (http://open-services.net/pipermail/community_open-services.net/2010-January/000214.html). *open-services.net Community* (Mailing list). Retrieved 16 February 2010.
13. Taylor, Bradley. "Rails Deployment and Automation with ShadowPuppet and Capistrano" (http://blog.railsmachine.com/articles/2009/02/10/rails-deployment-and-automation-with-shadowpuppet-and-capistrano/). *Rails machine* (World wide web log).
14. Fowler, Martin. "Practices" (http://martinfowler.com/articles/continuousIntegration.html#PracticesOfContinuousIntegration). *Continuous Integration* (article). Retrieved 29 November 2015.
15. "Arduino - Debounce" (https://www.arduino.cc/en/Tutorial/Debounce).
16. "Reaching true agility with continuous integration" (https://www.atlassian.com/agile/continuous-integration).
17. Ries, Eric (30 March 2009). "Continuous deployment in 5 easy steps" (http://radar.oreilly.com/2009/03/continuous-deployment-5-eas.html). *Radar*. O'Reilly. Retrieved 10 January 2013.
18. Fitz, Timothy (10 February 2009). "Continuous Deployment at IMVU: Doing the impossible fifty times a day" (http://timothyfitz.wordpress.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/). Wordpress. Retrieved 10 January 2013.
19. Laukkanen, Eero (2016). "Problems, causes and solutions when adopting continuous delivery—A systematic literature review" (http://ac.els-cdn.com/S0950584916302324/1-s2.0-S0950584916302324-main.pdf?_tid=8ea1852c-f79d-11e6-9404-00000aab0f6c&acdnat=1487616855_e837d2062fa783ce6f4348d7be9c638c) (PDF). *Information and Software Technology*. **82**: 55–79. doi:10.1016/j.infsof.2016.10.001 (https://doi.org/10.1016%2Fj.infsof.2016.10.001) – via Elsevier Science Direct.

20. Debbiche, Adam. "Assessing challenges of continuous integration in the context of software requirements breakdown: a case study" (http://publications.lib.chalmers.se/records/fulltext/220573/220573.pdf) (PDF).

Further reading

- Duvall, Paul M. (2007). *Continuous Integration. Improving Software Quality and Reducing Risk*. Addison-Wesley. ISBN 978-0-321-33638-5.

External links

- Fowler, Martin. "Continuous Integration" (http://www.martinfowler.com/articles/continuousIntegration.html).
- "Continuous Integration" (http://www.c2.com/cgi/wiki?ContinuousIntegration) (wiki) (a collegial discussion). C2.
- Richardson, Jared. "Continuous Integration: The Cornerstone of a Great Shop" (http://www.methodsandtools.com/archive/archive.php?id=42) (introduction).
- Flowers, Jay. "A Recipe for Build Maintainability and Reusability" (http://jayflowers.com/joomla/index.php?option=com_content&task=view&id=26).
- Duvall, Paul (2007-12-04). "Developer works" (http://www.ibm.com/developerworks/java/library/j-ap11297/).
- "Version lifecycle" (http://www.mediawiki.org/wiki/Version_lifecycle). MediaWiki.
- "Continuous Integration in the Cloud" (http://crosstalkonline.org/storage/issue-archives/2016/201605/201605-Dustin.pdf) (PDF). CrossTalk. 2016.
- Bugayenko, Yegor. " Why Continuous Integration Doesn't Work" (https://devops.com/continuous-integration-doesnt-work/)