Ranorex® | White Paper

WHAT IS CONTINUOUS TESTING?

What is Continuous Testing?

Conventional **software testing** has often meant the separate, manual, "end-to-end" exercise human specialists perform toward the end of a development cycle, largely after developers finish all or most of their work. **Continuous testing (CT)** is a contrast in nearly every aspect. CT generally:

- Is automated
- Happens as a result of programmer actions
- Occurs throughout the development cycle, from the time the first lines of source are written
- Returns results to programming staff through a dedicated and automated channel.

While there is a significant difference between the "**what**" of CT and conventional testing, the "**why**" and "**how**" differ even more.

Why Adopt Continuous Testing?

Conventional testing's business role often is a "gate" or filter: programmers write an application, then a testing department has the opportunity to report why the application isn't ready for the marketplace. It's easy for testing to be hurried and compromised, as other departments focus on the milestones outside testing. The organization as a whole often, if inadvertently, pressures Testing to hurry. Communication of Testing results generally occurs through "human" channels: a written report or tabulation of individual findings.

CT plays a more positive, integrated, reproducible, objective, and sustainable role: rather than just costing time between the end of programming and release to customers, CT generates information DevOps teams immediately apply to improve the product. CT simultaneously generates information about the business risk of that product. As CT begins with the very first programming, its measurements are available as quality metrics to help estimate when to complete and release a version of the product. Objective fulfillment of requirements largely replaces subjective speculation about how long testing will take to finish.

CT has a wide span. CT potentially plays a role from far to "the left"--white-box test-driven development of all programming--to "the right", with black-box system-level testing-in-production and asynchronous non-functional testing.

Shift left: Get quick results from testing

A unifying CT theme is left-shifting automation. Because it's automated, CT results and benefits are more-or-less immediately and routinely available. There's no need to coordinate schedules and ensure Testers aren't vacationing or double-booked or so on when developers need test results.

With a properly-configured integrated development environment (IDE), CT starts at the first keystrokes. A good IDE reports syntax errors and suggests improvements or completions as a programmer types; there's no need to wait for an often time-consuming compilation.

This represents an enormous **ergonomic** gain. Unaided programmers are prone to mistyping and mis-capitalizing names, misplacing punctuation, and so on. When compilation takes many minutes, as it often does for even small units of mature products, those errors are only identified many minutes or more later, when the programmer's attention has already moved on to other matters.

Here's a concrete example in JavaScript: it's easy for a programmer to mistakenly type

```
global_threshold = ( => 42;
```

By the time this has been entered, though, the IDE can identify that a closing parenthesis is missing, and even suggest

```
global_threshold = ( => 42;
```

A good IDE thus compacts the fundamental

```
code-compile-understand diagnostics-repair errors
```

cycle down to

```
code-correct
```

Programmers keep their attention on higher-order programming abstractions, and trust the CT to handle most of the **syntactic** and **semantic** validation quickly and with a minimum of distraction.

A next level of CT is functional: good unit tests in a CT scheme mean that functional errors are quickly identified. While correction of these pragmatic errors takes a little deeper comprehension than fixing syntax errors, their early identification encourages good programmers to "get in a flow" that keeps the overall state of development always either "green", or just one step away. An example at this level, in Python, might be

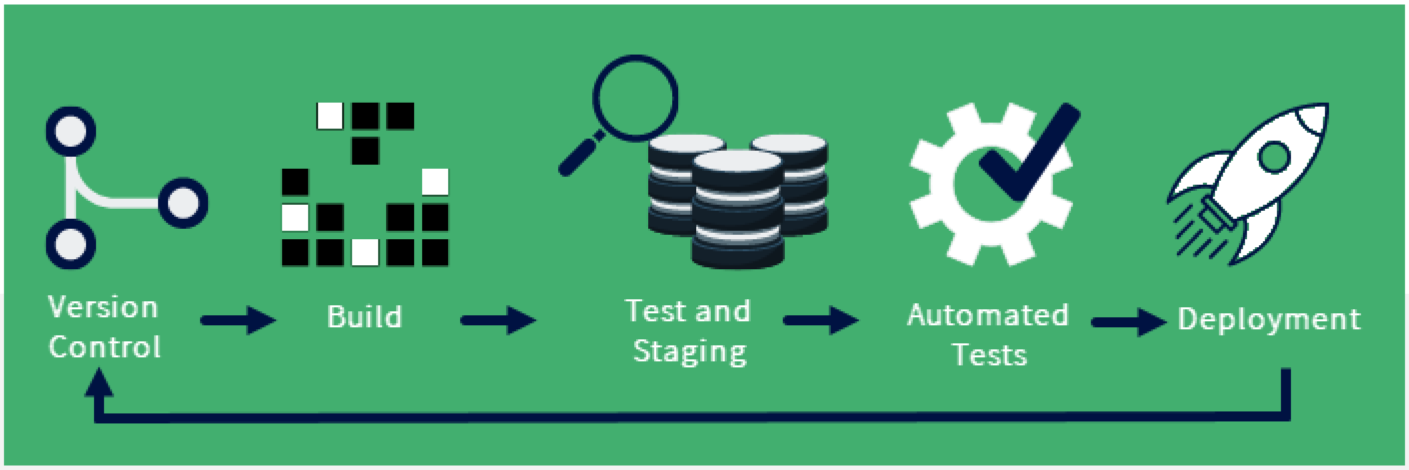
```
for depth in range(1, no_elements):
  clean item(depth)
```

This represents an off-by-one error that minimal unit tests quickly detect. It's usually just one additional quick step to correct

```
for depth in range(1, no_elements + 1):
  clean item(depth)
```

Many programming teams configure CT at multiple development points: with IDEs, for example, but also on **commits** of source to a **source-code control system** (SCCS). CT applied on commits of source is probably the single most common element of an identified continuous integration (CI) **pipeline.**

CI/CD PIPELINE DIAGRAM



Notice the **programmer experience** of CT: the system tells the individual its findings. There's no particular subjective element, at this level. Just as with integrated spell-checkers in word processors, any finding is about the source. It's not that a human programmer typed a wrong character, and a different human adversary found the error and reported it through channels. Instead, the source simply flags an exception, often with a recommendation for a fix, and the programmer implements the fix and continues, without further interruption or distraction or emotional involvement. Too many developers report source review and QA to be nightmares that they try to avoid. Good CT helps turn these stressors into predictable daily professional habits.

Shift right: Extend the range of testing

While left-shifting is a natural first step for CT, right-shifting is also rewarding. Testing in production has long had a bad reputation: it was done as a last resort, and earned its association with a crisis.

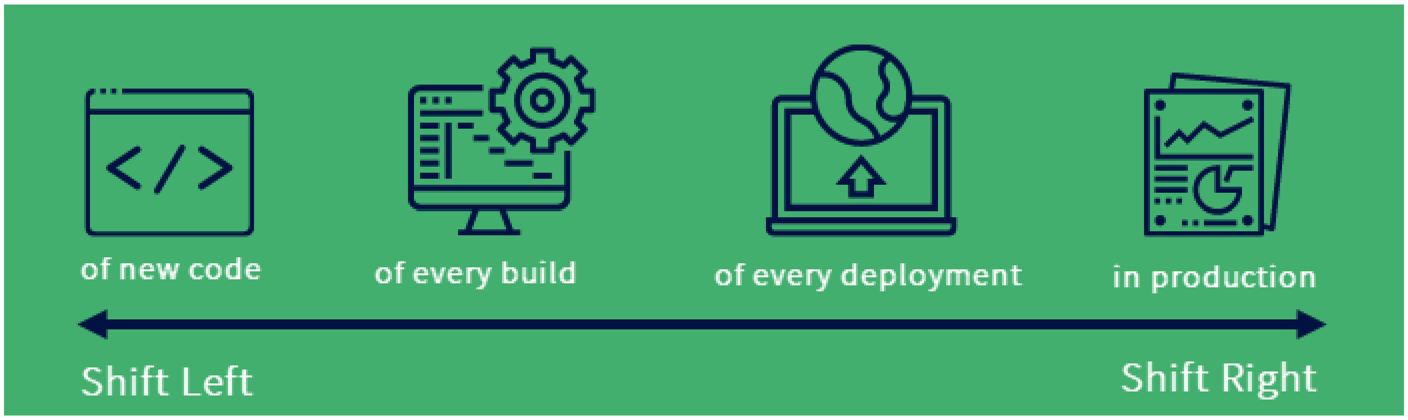
One of the great lessons of CI and CT, though, is how best to achieve **velocity** and **quality.** Older software development practice delayed integration and testing until the last possible moment, to defer those time expenses as long as possible. CI and CT teach that we're better off to make individual updates small, inexpensive, routine, and quick. This attitude wrings much of the drama out of software development: from the very beginning of a project, quality is high, tests are green, **branch conflicts** are few, and it's much easier to estimate how long the next milestones will take to complete.

In this perspective, testing-in-production is a habit to cultivate, rather than an emergency to avoid. With the right tooling and architectures, production-time testing creates no hazards for customers. Instead, a good testing-in-production practice enables at least two great symptoms of healthy service delivery:

> The provider detects any errors as soon as customers do. Customer support proactively corrects faults, rather than reacting to customer complaints;
>
> When errors arise, tooling is already in place to analyze those errors to diagnose root causes. There's little or no need to invent new mechanisms on the spot to understand the production environment adequately.

Consistent CT, including testing-in-production, simultaneously keeps problems small and quickly detected, and also practices the team in the techniques to remediate any problems that do turn up.

CONTINUOUS TESTING DIAGRAM



Expand the scope of testing

That's the basic operation of CT, then: execution of automated tests throughout the pipeline of software delivery. Note that "delivery" has multiple parts. CI, for instance, concentrates on the reproducible integration of software updates to a "test" or "**stage**" instance of usable software. A later stage of the pipeline **deploys** software to the "**production**" hosts or customers where it can usefully run. A comprehensive product perspective often speaks in terms of continuous **delivery** (CD) of service to ultimate customers. From this base of focusing on automation throughout the pipeline, a number of extensions are possible. In the IDE and in CI, CT happens **synchronously**: a programmer acts, and the CT quickly returns a go-no go (or green-red) response.

Certain automated tests are important, but so expensive in time or other resources that they have to be scheduled. While such tests don't return immediate results, they're still part of CT in being automated and in acting on current sources. "Current" here might refer either to what's active in an IDE, or committed to a branch of the SCCS, or perhaps a combination.

Consider a few concrete examples: unit tests for correct parsing of results delivered by a database lookup are good candidates to be mocked. They'll typically finish in a few milliseconds, and are worth running on every commit. Query performance results might also be important: if a subtle algorithmic change causes a search to take three minutes, rather than three seconds, we want the programmer who made that change to know as soon as possible. Performance tests are inherently expensive, though: rather than launching on each commit, it might be necessary to schedule certain performance tests to run only once an hour. That's a compromise: scheduling in this way doesn't give the programmer immediate feedback, but it's far better than waiting three weeks for its discovery at the end of a **sprint**.

Another activity point to apply certain tests is in production; here, the goal is to run tests that identify and even correct errors before end-users realize they're happening. To "drill down" or expand a report of a production-level symptom into more detailed descriptions of what relates specifically to that symptom is another crucial CT capability.

Recognize that CT also has the potential to do far more than just validate syntax and launch unit tests. More comprehensive CT verifies many dimensions of source style, including the contents of documents and headers. Automations also exist for measurement of **accessibility,** suitability for (human-language) **localization**, security and privacy hazards, and much more. Recall the fundamental workflow behind all these: each time a DevOps team member creates or updates or corrects anything about the product under consideration, automations spring into action to report any problems as quickly as possible. That's CT, and it greatly elevates the productivity of a development team and the quality of the software that team develops.

CT is like other software in requiring active management. Teams typically prototype a CT that automates just one aspect of quality control. Some teams even get "stuck" at that first step, with such an overwhelming load of features to implement or errors to correct that they're unable to enjoy CT's wider "wins". In general, though, as the team gains experience and familiarity with CT's operation and outcomes, it's natural to extend the CT to deliver more and more benefits.

Enhance teamwork

CT doesn't eliminate a Quality Assurance department, manual testing, or specialized testing responsibilities. Even organizations with the most advanced CT installations still have plenty of work for testers to do. CT is a bundle of practices around test automation that help get the most value from tests that can be automated, drive down the cost of those tests, enhance developers' productivity, and allow professional testers to concentrate on other aspects of testing that repay their expertise.

How to Succeed with Continuous Testing

At an important level, continuous testing is about how team members experience *what other people do*: CT's automations give confidence that any code checked into a branch, say, *even by someone else*, has passed specified tests. In such a system, all software assets achieve a specified quality level. That kind of confidence gives individual practitioners the freedom to concentrate entirely on the work before them, with a minimum of doubt that errors elsewhere in the system will confound their efforts.

Programmers often experience continuous testing in terms of the tools that implement CT:

> Configuration of a **repository** to define a particular CI pipeline;
>
> A testing framework that facilitates expression of functional tests;
>
> Any IDEs involved;
>
> Review managers to expedite source review;
>
> Supplementary validators for style, resource use, performance measurement, and so on; and
>
> Reporting dashboards that capture the instantaneous health of the development cycle.

Ideally, all these tools "play nicely" with each other and interface easily. The greatest success is when the testing and development teams rarely think about the tools: most of their time should simply go to their productive use, rather than having to study how to achieve particular effects.

Keep in mind that any organization's view of continuous testing will change through time. It's entirely appropriate to start with even a single automated test for a single activity point, say, a unit test which runs each time a developer commits to a branch. Over time, as the organization gains **fluency** with CT and understands more of what it can achieve, it's natural to extend the range of tests, shift them left, shift them right, and so on. Over time, CT helps the organization progress on shrinking release cycles from years to weeks or even hours, elevation of quality scores closer and closer to 100%, and competence in repair of customer problems in minutes or days rather than months. CT minimizes surprises, and makes software development more **manageable.**

Does your development team have confidence that minor errors--misspelled function names, off-by-one oversights, Web elements that don't render in the visible window--will be caught and corrected right away? If so, continuous testing is making you a winner; if not, it's time to plan how to make continuous testing work for *you*.

Ranorex®

US Headquarters

10801 N Mopac Expressway
Building 1, Suite 100
Austin, TX 78759
USA

+1 727 835 5570

sales@ranorex.com