

Table of Contents

Adopting Continuous Testing as Part of the CI/CD Pipeline 3

The Fundamentals of Continuous Integration, Deployment, and Delivery 4

Shrinking the Feedback Loop 6

Automation of Error-Prone or Sensitive Processes 7

Considerations When Adding UI Testing to CI/CD 8

Extra Infrastructure 8

Leveraging the Cloud for Flexibility 8

Managing Dependencies and Third-Party Tools 9

Duration 9

Implementing UI Automation in Your Organization 10

Who Builds the Tests? 11

Support Frameworks 11

Continuous Improvement 11

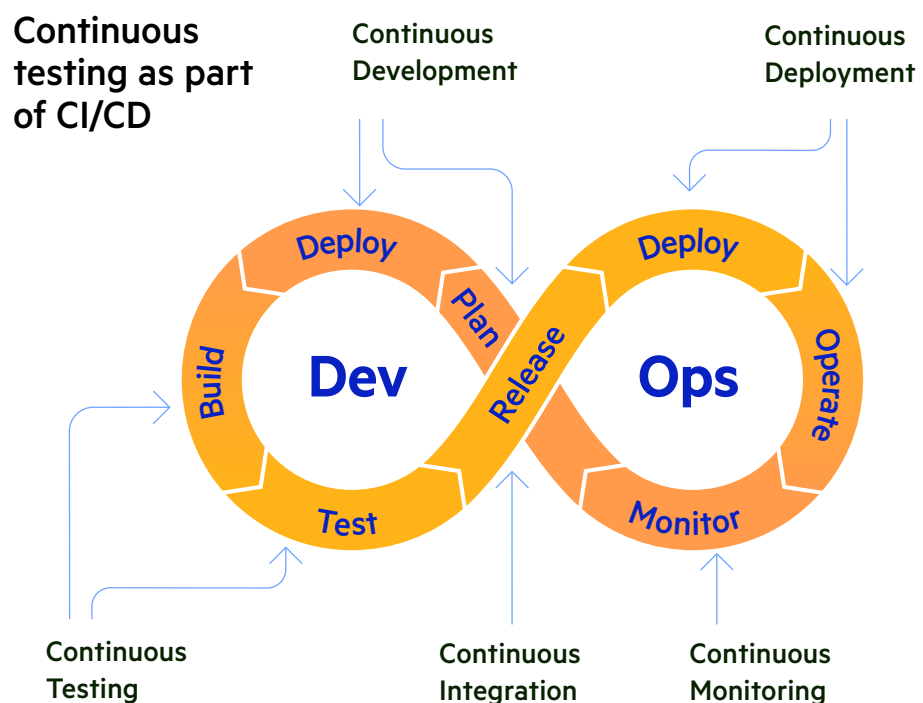
Improve Your Pipeline's Value With UI Test Automation 11

Adopting Continuous Testing as Part of the CI/CD Pipeline

Most organizations have grasped the value of Continuous Integration (CI) and Continuous Deployment/Delivery (CD). The benefits of frequent, if not continuous builds and integrations have already been proven across the industry for a number of years. Organizations are continuing to mature by including additional quality gates as part of their automated processes.

The value of automatically executing unit and integration checks, static code analysis, and other similar actions has been long an integral part of the CI/CD flow. What's been missing for too long is the inclusion of User Interface (UI) testing as part of the CI/CD pipeline.

Thankfully software delivery is embracing advances in thinking around testing and is using a "push left" mindset to bring testing earlier into the delivery cycle. Organizations are seeing tremendous benefits by adopting a continuous testing culture and practices which involve testing activities early in the projects. Solid conversations early in the game lead to better understanding of what's to be built, and ensure the team has appropriate coverage for all forms of automated and exploratory tests. Forming this solid base ensures the best possible value of continuous testing in a CI/CD environment.



 Progress® Telerik® Test Studio®

Understanding CI/CD Benefits

Continuous Integration has been around for decades, and most organizations understand its benefits. Continuous Delivery and Deployment are much newer in the industry, and as such many organizations still haven't realized their critical value to a smoothly running, value focused organization.

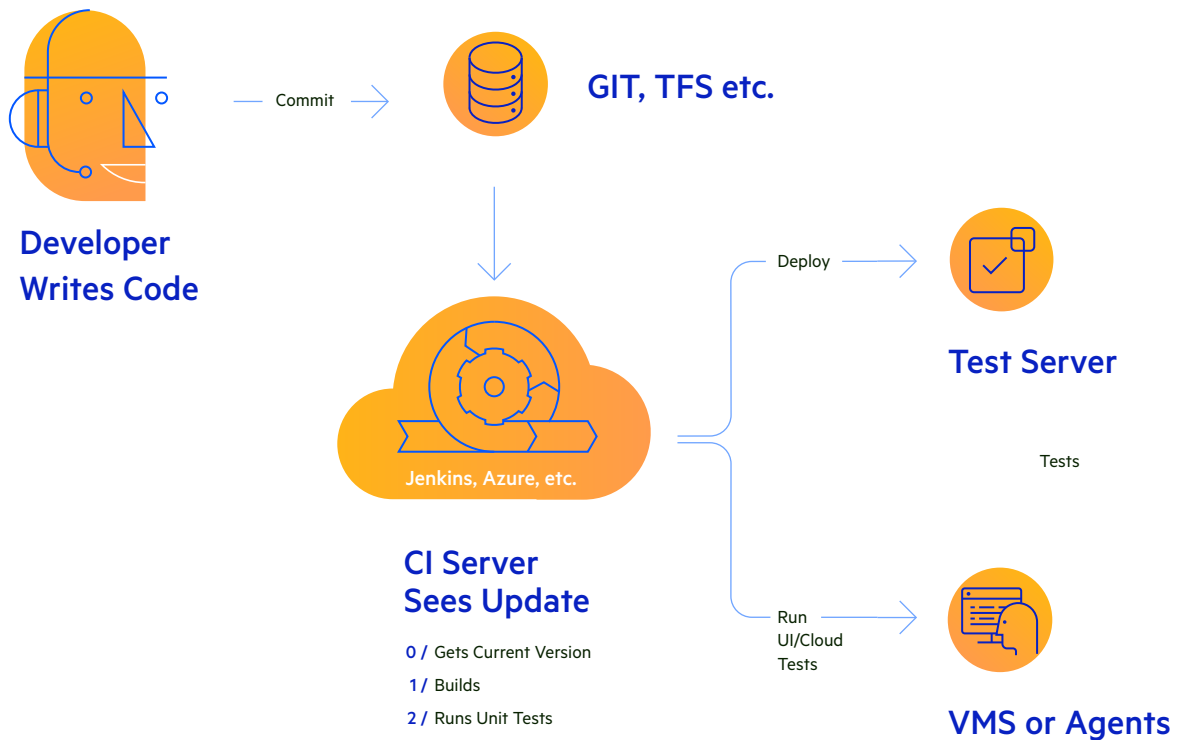
The Fundamentals of Continuous Integration, Deployment, and Delivery

Continuous Integration, Continuous Deployment, and Continuous Delivery are often misunderstood and conflated, so it's important to understand how the terms are used in this article.

At its most basic form, Continuous Integration (CI) uses a specialized build server to frequently pull source code from the repository. The server runs the most basic compile/link steps needed to fully build the software. CI was created as an attempt to mitigate problems in environments where developers or teams wouldn't merge or integrate separate work for extended periods—months in the worst cases. This led to the most basic of build problems: errors due to a failure to integrate fundamental changes by source code changes from sources other than the developer's own system. Continually building the source code ensures that at least no breaking errors at the syntax or linking stages have been introduced by other members of the team.

In larger environments these builds often can't be run after every single push to the source code repository. Instead the CI server is configured to build once after every X number of commits, or once every X minutes. These values are typically adjusted down to the smallest practical increments in order to provide the fastest possible feedback.

CI servers have evolved to handle far more tasks than just the build process. CI servers will often orchestrate a number of other quality-related tasks which can be run without deployment to an actual environment. Examples including running unit tests (unit tests require no dependent systems such as databases), static code analysis, etc.



Continuous Delivery takes this notion a step further by automating processes to ensure any given commit or check-in to source control is ready to be delivered to production at any time. Continuous Delivery takes the frequent builds from CI, and pushes them to appropriate deployable environments where further automated checks can be performed—another round of unit tests, integration tests checking services, user interface or end-to-end tests validating business-level features, security and accessibility testing, and often at least perfunctory performance testing.

This series of tasks and escalating environments is often referred to as a delivery pipeline.

A critical concept in Continuous Delivery is that code which doesn't pass any of the various quality gates (build, automated testing, static code analysis, etc.) does not move further down the delivery pipeline. The process is immediately stopped and appropriate error notifications are publicized.

Continuous Delivery often also addresses audit and regulatory compliance issues. As such, artifacts from the build and test processes are stored as a group to provide traceability where needed.

Continuous Deployment closes the last step in this evolution by deploying production-ready packages to the organization's final environment. While this may sound trivial, in many organizations deployment to production is forced to comply with company or regulatory rules. As such, the auditing factors mentioned earlier become even more prominent.

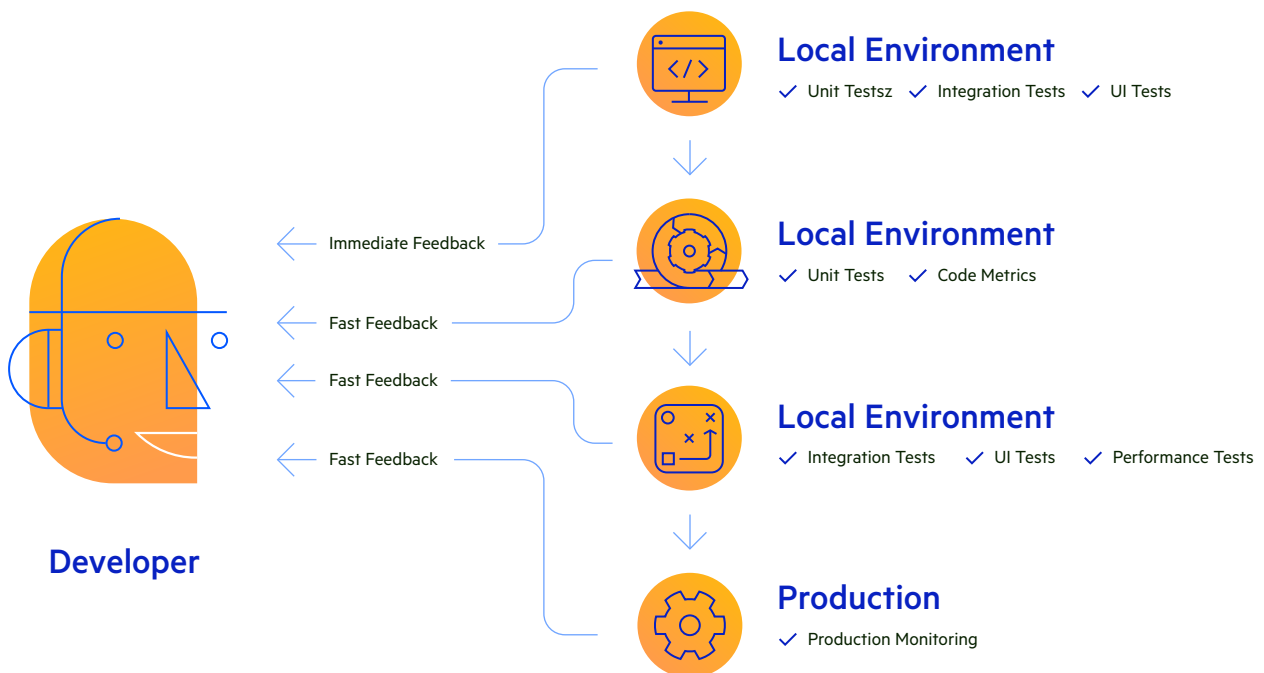
Shrinking the Feedback Loop

One of automation's primary benefits is speeding up the feedback loop.

Continuous Integration, as first named by Grady Booch in 1991, has always been about minimizing delays in understanding when critical errors exist in a codebase. Learning about integration issues immediately, versus days or weeks later, has been CI's focus since its first implementations.

Projects that don't get rapid feedback on the state of code and system quality are at significant risk. Missing critical errors until late in the game will likely cause costly rework and problematic delays. Shortening the feedback loop through automated processes is a proven way to reduce risk and smooth out project delivery.

CI servers like Jenkins, TravisCI, TeamCity, or Visual Studio Team Systems, allow teams to create multiple tasks in an automated pipeline. These same servers handle orchestration for continuous delivery/deployment. These tasks will accomplish specific granular operations like pulling from source repositories, building the system, deploying to staging environments, running appropriate quality checks, etc. Failure at any of these gates will result in notifications being pushed out to team members and various dashboards.



Contrast this to “old-school” approaches where complex quality gate tasks were run only once a night, once a weekend, or worse yet at some longer, arbitrarily determined period. Critical errors were masked for far too long. Numerous other changes were likely piled on top of the initial error, either hiding it or making it worse.

On the other hand, a well-constructed pipeline ensures all team members know immediately when any part of the pipeline has failed.

Automation of Error-Prone or Sensitive Processes

Humans are prone to make errors. The error rate gets far worse when the task at hand is repetitious because we humans don’t do well at focusing on detail when we’re bored by doing something for the 1,542 time. Wikipedia and YouTube have thousands of examples of cringe-inducing errors where lack of attention costs huge amounts of money, injury, or worse.

We see this regularly in all aspects of Information Technology work. Manual deployments break due to missed configuration options. Teams store sensitive or secret data in public locations by manually copying a sensitive file to an inappropriate area. Artifacts are pushed to production yet never stored in an archival location, breaking all audit chains.

Moving to an automated CI/CD process eliminates the risk of such errors by creating hands-off tasks that run the same every time. That’s not to minimize the thought, care, and testing needed to build these tasks; it’s just that the human factor can be vastly mitigated.

Moreover, unlike with human actions, all CI/CD servers offer a complete audit trail of what tasks were executed by whom at what time. Security issues can be controlled and documented by applying well-known, highly understood security controls for each of the various tools used.

Automated tasks offer audit trails, compliance coverage, and risk-elimination for all tasks involved in building, testing, and deploying software.

Considerations When Adding UI Testing to CI/CD

Unit and integration testing have long been a part of CI/CD environments. User interface, or end-to-end testing, hasn't normally been a part of such pipelines due to its complexity.

Extra Infrastructure

User Interface testing requires significantly more infrastructure and tooling in place than “just” unit or integration testing. Servers hosting the application require a full environment with most, if not all dependent services in place. A reasonable, if not full data source is required. Many similar dependencies add to the problem.

Extra infrastructure is required to solve this problem by providing places for multiple environments to be provisioned and hosted during the pipeline's execution.

A critical aspect of these environments is that they be built dynamically upon need. Dynamic environment creation ensures the highest flexibility for deploying to different parts of the pipeline (development, test, staging, etc.) without manual intervention. Dynamic construction also ensures teams can avoid manual intervention in the CD pipeline by holding all configuration as part of the build scripts paired with data pulled from source control during the build.

All this work so far is just on the server side. UI testing requires infrastructure from which to run the test clients (browsers, agents, etc.). UI test automation complexity gets even more intense when mobile is brought into the picture because mobile platforms require infrastructure for both simulators and real devices.

In all practicality this client test execution infrastructure will need to support parallel execution. It's not feasible to run even modest-sized UI test suites in series—multiple execution is a near necessity.

Leveraging the Cloud for Flexibility

Managing UI client infrastructure brings its own set of headaches. Some organizations chose to maintain their own internal infrastructure, while others look to service providers like Microsoft Azure, SauceLabs, etc.

Using external cloud providers can be a huge boost for organizations looking to scale out their automated testing. Azure, Amazon, SauceLabs, and all major cloud platforms provide the out-of-the-box ability to instantly spin up multiple execution agents dynamically based on the needs of each test run.

Such effort does require up-front work to provision and configure the infrastructure; however, tools such as pre-configured Selenium WebDriver docker images can tremendously ease the effort.

Additionally, organizations need to be sensitive to cost-management of cloud-based infrastructure of any sort. Cloud test infrastructure is no different, so care must be taken to ensure the CI/CD build pipelines properly shut down all assets after each test run is completed. A sharp eye needs to monitor asset utilization reports and the associated financial impact.

Managing Dependencies and Third-Party Tools

As mentioned earlier, system-level dependencies must be addressed as part of a robust, practical automation pipeline. As such organizations need to address how to incorporate external services such as databases, workflow engines (think Tibco or BizTalk), security providers, etc. External content providers may need to be addressed, and even small third-party tools like CAPTCHA must be considered.

Often in lower environments (development, testing, early staging) many of these services can be simulated via various mocking or virtualization tools. While these approaches can greatly improve speed and flexibility when dealing with dependent services, they're still engineering tasks that require time and effort.

Feature flagging or switching can also be used to manage dependencies by enabling configuration options to turn off entire swathes of functionality, bypassing unneeded dependencies or portions of the system not needed for testing. This approach is often used to cut CAPTCHA out of an automated test flow, for example.

Duration

Without a doubt UI or end-to-end tests are an order of magnitude slower than other forms of automated tests. Keeping your entire test suite, all types of automated tests,

running fast must be a high priority when fleshing out a delivery pipeline. Slow execution of tests breaks one of the major benefits of the entire concept—fast (or as fast as possible) feedback.

Understanding what to test via the UI and what not to test is the most critical aspect of good UI test automation, regardless of whether you're running those tests in a delivery pipeline or not. Far too many teams automate far too many tests at the UI level. Instead, carefully determine what's already covered via unit and integration tests. UI tests should never check validations, algorithm correctness, or error handling for example. Those should be handled by unit or integration tests. Instead, UI tests should focus on user-level flows such as 'Can an item be deleted from my shopping cart?', 'Can I update a contact on my customer list?'

Ensuring only the most valuable, appropriate functionality is checked with UI automation is the most crucial part of keeping a fast-running suite.

Addressing slow-running test suites is an exercise in priorities and can bring pleasant results when teams look at the system itself, versus simply throwing hardware at an ever-growing parallel grid of client-agent systems.

Emanuil Slavov gave a presentation at ISTACon 2015 on his team's efforts to speed up their test suite's execution time. When they started, their test execution was three hours. They approached the problem with a very thoughtful, measured approach working through constraints and bottlenecks. They brought their test execution time down to three minutes by dramatically reworking their system architecture and design. They only added in parallel test execution after they'd hit five minutes.

You can find a video of Slavov's talk on the ISTA Conference's [YouTube channel](#).

Implementing UI Automation in Your Organization

Creating and managing automated delivery pipelines is an engineering effort. It requires time and work from the entire team, and management must be willing to ensure support for the work. Rolling UI tests into your pipeline requires an understanding of the principles already laid out earlier, coupled with a number of practical issues around team responsibilities.

Who Builds the Tests?

Coming to an agreement on who writes UI tests is critical to a smooth-running workflow. Creating solid, valuable, maintainable UI tests is best done as a whole-team effort that starts as early as possible and is tied to work item completion. UI testing should never be left as a separate task accomplished after the feature or user story is finished. This leads to disconnects in flow and can result in UI test automation falling far behind the actual work.

When possible, developers and testers should pair up to create the UI tests while work on the feature/user story/task is being done. This allows developers to quickly modify the system in order to make it more easily testable—think better UI locators, flags for complex asynchronous actions, etc.

Moreover, pairing developers and testers together helps ensure the best organized, most maintainable test code possible. Good developers intrinsically understand and use clean code principles to help eliminate duplication and complexity in tests. Developers can help testers create and maintain things like Page Objects for UI representation, Domain Objects for handling business logic, etc.

Some organizations aren't mature enough to place developers and testers in combined teams. Other organizations may not have the IT staff or skills to write full-on coded UI tests. In such cases high-quality commercial test automation tools like Progress Telerik Test Studio can be used to fill the need for building high-value, easily maintainable UI tests. Commercial tools should be evaluated for their ability to quickly create tests that are adaptable and flexible. All UI test suites will require some form of code modification, so it's critical that any commercial tool selected supports the record-modify-playback/execute philosophy. Additionally, such tools must provide extensions or command line tools to properly integrate into the pipeline. Output format for test execution runs is critical in order to consolidate UI test status in the CI/CD server's dashboards. Support for common test report format such as NUnit or JUnit makes for the easiest incorporation into existing pipelines.

Support Frameworks

All test automation projects (other than the most trivial ones) will require creating customized frameworks to support critical tasks as part of a test automation suite. Custom frameworks enable UI tests to quickly set up test prerequisites, create test data, configure systems, provide hooks into the system for oracles or heuristics, and other tasks custom to the specific system under test.

These custom frameworks are critical in a delivery pipeline. Frameworks like this give the automation suite important flexibility to adapt to the various environments systems will be deployed to in a CD flow. Well-designed frameworks decouple the test suites from specific databases or other services and help ensure tests run stably and consistently no matter where they are currently being executed.

Custom frameworks of this nature are generally created by developers familiar with the system. These frameworks are best implemented when they leverage existing APIs and features of the system to perform various tasks.

For example, creating a new contact for testing in a Customer Relation Management (CRM) system should not reach directly into the system's database. Instead, the custom test framework could hit an existing web service in use by the system. This approach ensures required business logic is respected, and any resulting actions are properly triggered. Re-writing such calls within the test framework would add significant complexity and duplication. Moreover, there would be a significant risk of problems if the actual system behavior ever changed—the odds are high the test framework code wouldn't be updated at the same time.

Continuous Improvement

No software organization succeeds without committing to continually improving. The same is true for delivery pipelines, and especially true for UI automated testing suites. Organizations need to commit time and effort to updating and maintaining their delivery effort in order to keep it valuable and smooth-running.

Improving pipelines is especially important for organizations that are new to CI/CD, software craftsmanship, or test automation. Teams will learn an incredible amount about how to work efficiently and smoothly. These learnings normally come through hard lessons learned and outright failure. It's to be expected, although good organizations try to minimize rework and failures as part of those learnings.

There are many areas ripe for continuous improvement in test automation running as part of a pipeline:

- **More effective test scripts.** Cutting back on low-value UI tests is always a good thing. Refactoring or rewriting existing tests to make them more stable, faster, and more easily maintained is also a crucial task.
- **Better use of Page and Domain Objects.** New teams often struggle initially with these concepts. Ensure time is made available to refactor or rework as needed.

- **Improving Custom Frameworks.** Invest time building better test data helpers, oracles, or configuration options. Make use of new system features that could improve testability.
- **Level up software craftsmanship principles.** Teams don't start out as great coders. They learn it through practice and experience.

Improve Your Pipeline's Value With UI Test Automation

Continuous Delivery / Deployment pipelines provide terrific value to organizations creating software systems. Adding UI automated suites to a pipeline takes careful planning and execution, but it's a natural evolution for organizations wanting to improve the quality and value of systems they ship.



About the Author

Jim Holmes is a modernization strategist and coach for Centric Consulting. He helps organizations improve their software delivery processes by working closely with executives, delivery teams, and individual team members. He's worked with organizations in many industries and has spent time with organizations from small startups to global Fortune 10 companies.



Try Telerik Test Studio

About Progress





Progress(NASDAQ: PRGS) offers the leading platform for developing and deploying strategic business applications. We enable customers and partners to deliver modern, high-impact digital experiences with a fraction of the effort, time and cost. Progress offers powerful tools for easily building adaptive user experiences across any type of device or touchpoint, leading data connectivity technology, web content management, business rules, secure file transfer and network monitoring. Over 1,700 independent software vendors, 100,000 enterprise customers, and two million developers rely on Progress to power their applications. Learn about Progress at www.progress.com or +1-800-477-6473.

© 2020 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.
Rev 2020/05 RITMxxxxxx

Worldwide Headquarters

Progress, 14 Oak Park,
Bedford, MA 01730 USA
Tel: +1-800-477-6473

www.progress.com

-  facebook.com/progresssw
-  twitter.com/progresssw
-  youtube.com/progresssw
-  linkedin.com/company/progress-software