









CONTINUOUS TESTING IN PRACTICE

Completing the Continuous Delivery Process



CONTENTS

	INTRODUCTION: CONTINUOUS DELIVERY IS THE NEW NORMAL	(1)
	PART 1: WHY CONTINUOUS DELIVERY?	(2)
	PART 2: TESTING: THE FINAL BARRIER FOR CONTINUOUS DELIVERY	(3)
	PART 3: CONTINUOUS DELIVERY WITHOUT CONTINUOUS TESTING	(4)
	PART 4: HOW CONTINUOUS TESTING FACILITATES CONTINUOUS DELIVERY	(6)
	PART 5: CONTINUOUS TESTING IN PRACTICE: REQUIREMENTS CHECKLIST	(9)
	CONCLUSION	(11)
	REFERENCES	(12)





INTRODUCTION

CONTINUOUS DELIVERY IS THE NEW NORMAL

Continuous Delivery (CD) is rapidly emerging as the 'new normal' in software development, with approximately 80% of SaaS companies and 51% of non-SaaS companies adopting this practice¹. Dominant online companies like Amazon, Facebook, and Google have been implementing aggressive Continuous Delivery workflows for years, and deploying release to production numerous times a week or even multiple times a day. As such, companies failing to make the leap to Continuous Delivery are falling behind and putting themselves at a significant competitive disadvantage.

Despite the widespread, rapid, and perpetually increasing adoption of this practice, it is apparent that there are deep-rooted barriers preventing companies from the actualization of a 'True' Continuous Delivery process. Today, the toughest and most challenging barrier is testing.

This whitepaper covers the rise of Continuous Delivery, the key barriers and pain points leading companies towards inevitable failure, and how to overcome these challenges through Continuous Testing.





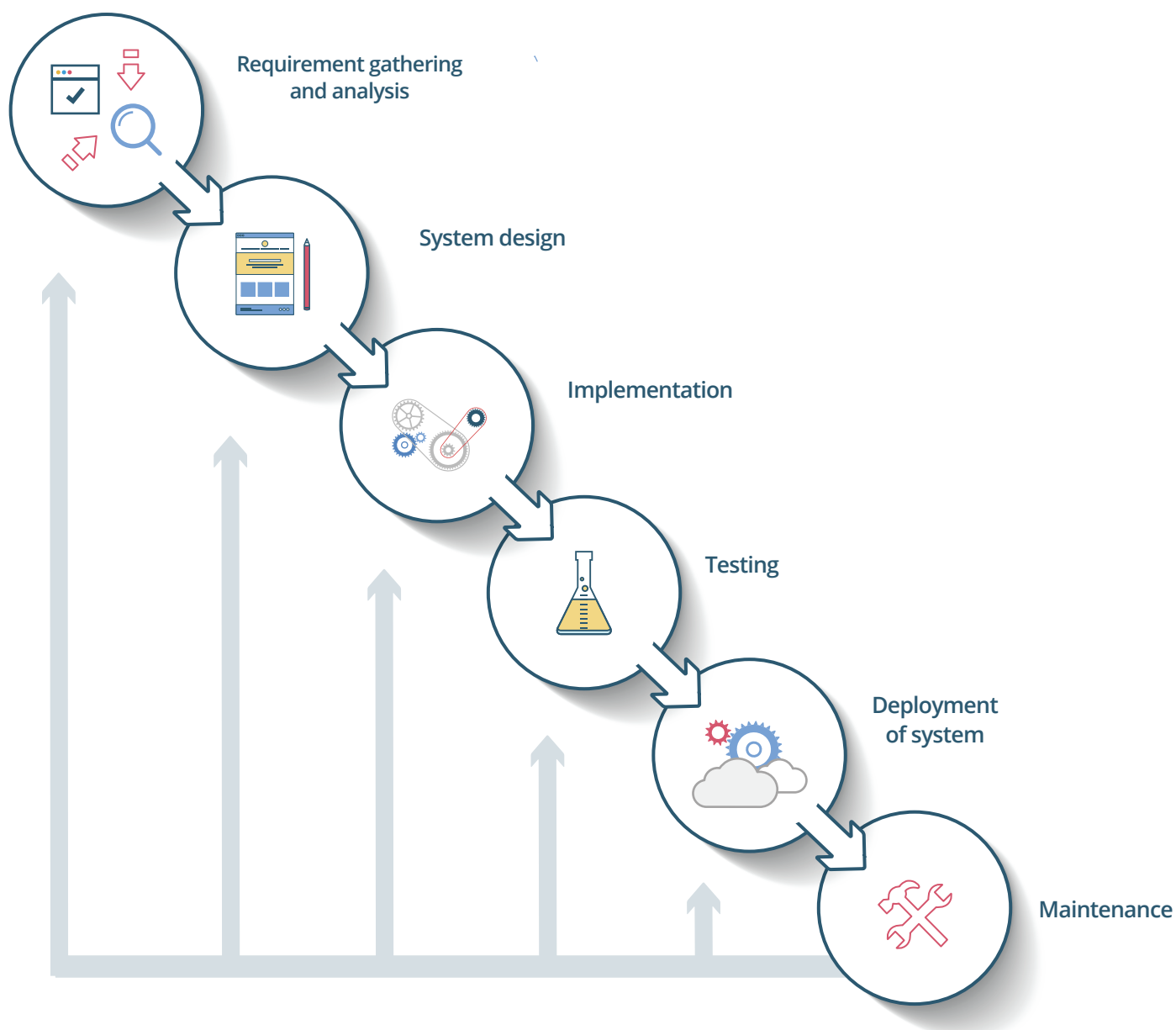
PART 1: WHY CONTINUOUS DELIVERY?

Jez Humble, co-author of the Jolt Award winning book on Continuous Delivery, describes the practice as follows:

“Continuous delivery is a set of principles and practices to reduce the cost, time, and risk of delivering incremental changes to users.”²

Continuous Delivery is the “Agile Age’s” answer to traditional waterfall methodologies - the dominant mode just a few years ago.

In the ‘Waterfall Age’, product iterations were linear and sequential, moving from product marketing specification, through to design, development, QA/testing, and finally deployment and maintenance. As the name “Waterfall” indicates, the process can only move forward. Once a phase in the lifecycle is completed, it proceeds to the next phase and it’s almost impossible to turn back.



There are numerous flaws with this approach. By the time an application gets to testing, it's very difficult to go back and resolve an issue that was caused in the design or implementation stage. As such, delays are common and application releases and updates usually take many months to complete.

In today's 'Agile Age', such timeframes are unacceptable.

Continuous Delivery plays a critical role in today's Agile Age. Without it, there is no agility. In an ideal Continuous Delivery process, the test, support and development teams work closely and collaboratively to automate and streamline the build, test and release process. Iterations are short, manual handoffs are eliminated, risks are minimized, and releases are fast and frequent.

As a result, a successful Continuous Delivery process can reduce the time-to-market for new releases from several months to a matter of days or even hours.



PART 2: TESTING: CONTINUOUS DELIVERY'S FINAL BARRIER

The case for Continuous Delivery is undisputable.

However, a smooth, error-free and automated Continuous Delivery process is difficult to actualize.

Key barriers have been overcome through the widespread adoption of Continuous Integration (CI) and Continuous Deployment. Continuous Integration, enabled by tools like Jenkins, TeamCity, and Travis, gave IT organizations the first component of Continuous Delivery. Developers integrate code into a shared repository and verify each check-in with an automated build, enabling the rapid detection and resolution of problems.

The second component, Continuous Deployment, could be considered as an extension of Continuous Integration. IT teams automate the deployment process, greatly improving deployment time and reducing human error, while minimizing the need for manual work.

Despite these considerable developments, a crucial component in the Continuous Delivery process is diminishing its effectiveness. To date, software testing remains a major source of friction. It has failed to match other advancements made in the software development lifecycle, yet its importance just keeps on increasing.

As software changes over time, the number of tests required to ensure its quality increases significantly. IT organizations do not have the time or manpower to run all the necessary tests every time a new feature is added. As a result, most companies only execute a small fraction of these required tests – resulting in software quality compromises or an increased time-to-release.

As such, any Continuous Delivery process is ultimately bound to fail.





PART 3:

WITHOUT CONTINUOUS TESTING: WHERE CONTINUOUS DELIVERY FAILS

Testing should be a cross-functional operation which involves the entire technical team throughout the product lifecycle.

To demonstrate the pain that testing can bring at every step of the Continuous Delivery process, this paper examines the product pipeline of a mid-sized organization adopting the practice.

A Continuous Delivery Scenario

Engineering teams develop software modules and commit their code regularly into a version control tool like GitHub.

“Devop” teams use a Continuous Integration solution such as Jenkins to run daily builds, including various post-build tests, to validate the quality of the build.

QA testers validate release candidates prior to their deployment in production with a handful of automation tools (e.g. JMeter, Loadrunner).

Operations validate releases that were just deployed in production, and monitor the application further in a post-production environment.

The above process is operated by numerous roles within the organization and require many test points, including:

- Post-build tests through unit/module tests
- QA tests through performance, functional, UI, soak, and regression tests
- Deployment acceptance tests
- Post-production monitoring/testing



Below is a typical example of how these tests points are executed by various roles throughout the organization.



As the example above illustrates, different tools are used by each role in the organization to perform a wide range of tests. When it comes to testing, the cohesiveness and consistency between each role is little to none. Each tool needs a script to perform a test, and the script usually requires the application to be “reverse-engineered”. For example: it may need to record the operation and convert this recording into a script the testing tool can understand and execute.

This issue is exacerbated when the application is updated regularly, as the scripts also need to be updated. This workflow is extremely hard to maintain and will ultimately result in inadequate testing and a longer time to release.

The above process is incompatible with a successful Continuous Delivery Process.



PART 4:

HOW CONTINUOUS TESTING FACILITATES CONTINUOUS DELIVERY

Continuous Testing involves applying the methods and concepts of agile development to the testing and QA process, resulting in a more efficient testing process.

For Continuous Testing to succeed, it requires an automated end-to-end testing solution that integrates into the existing development process, eliminating errors and facilitating true continuity throughout the development lifecycle.

This paper has already covered the roles and test points in a typical Continuous Delivery process.

Now, we present these same roles and test points when Continuous Testing is properly integrated into Continuous Delivery.

Developers

Testing must start with the developers, testing the performance and functionality of their own code by creating and running tests using open source tools such as JMeter or Selenium, commercial tools like Perfecto mobile or code, i.e Ruby or Python. Tests include API, Functional, Performance, UI, and more. Regression tests can also be added to the set to ensure functionality is not compromised and specific scenarios are tested.

Test configurations need to be committed with every code commit. The test configuration references all tests associated with a certain version of a software module.

To ensure this will be a widely adopted practice, the creation and running of such tests should be easy, fast, and require minimum prior education and/or experience. Developers can successfully create and execute these tests without help from consultants, professional services and support development workflows, such as version control, bug tracking systems etc.

DevOps (Continuous Integration)

DevOps do not need to understand the internals of each software module. As such, they only need to configure the testing element of any project once. Any post-build test should consider the module's test configuration and be able to run the right set of tests associated with the build in the CI environment. Tests run in parallel and end as soon as possible with a clear "pass"/"fail" indication, allowing the process to continue smoothly even when the number of tests increase.

Once the CI environment is configured to use the right set of tests, developers receive an automatic message whenever a build fails. They receive sufficient test artifacts to identify the cause of failure and the tools to be able to keep running the tests until a "pass" indication is given. This process should be as fully automated as possible.



Another important item to consider is the speed in which execution needs to happen. Not only is it important to ensure comprehensive coverage when testing, but it needs to be accomplished within a reasonable timeframe. Of course, the definition of a “reasonable” timeframe is purely subjective. Therefore, it's best to attempt to complete the CI process as fast as possible.

QA (Pre-Production)

QA needs to validate a release candidate prior to production. As such, they need to run comprehensive tests to ensure a release candidate is ready for deployment. There are numerous tests, which include: regression, functional, load, soak, and more. Today, QA testers build scripts and use tools according to a predefined test plan.

As there are sets of tests associated with the release candidates, the QA tester doesn't need to understand the internals of each release. Any pre-production test should consider the set of module test configurations and be able to run the right set of tests associated with the release candidate in the pre-production environment. Again, tests should run in parallel and end as soon as possible with a clear “pass”/“fail” indication, allowing the process to continue smoothly as the number of tests increase. When configured correctly, they can run continuously and without human intervention. When there's a failure, the developer should receive a message with the artifacts and ability to keep running the tests until a “pass” indication is given.

Operations (Continuous Deployment)

Every release needs to pass an acceptance test upon deployment. Once more, operations managers can leverage an existing set of tests which have already been created, used and, most importantly, are associated with a specific release.

Operations (Post-Production Monitoring)

In a production environment, operations managers use a variety of monitoring tools to test the application on an ongoing basis. As such, this process is often simply referred to as “monitoring”.

In a Continuous Delivery process, monitoring is yet another form of Continuous Testing. In a post-production environment, Operations can leverage the set of tests associated with a specific release to monitor the health of the application on an ongoing basis. There is no need to reverse engineer, create a script or adapt the script with every change. A comprehensive set of tests can run 24/7 and send automatic alerts on “failures”.

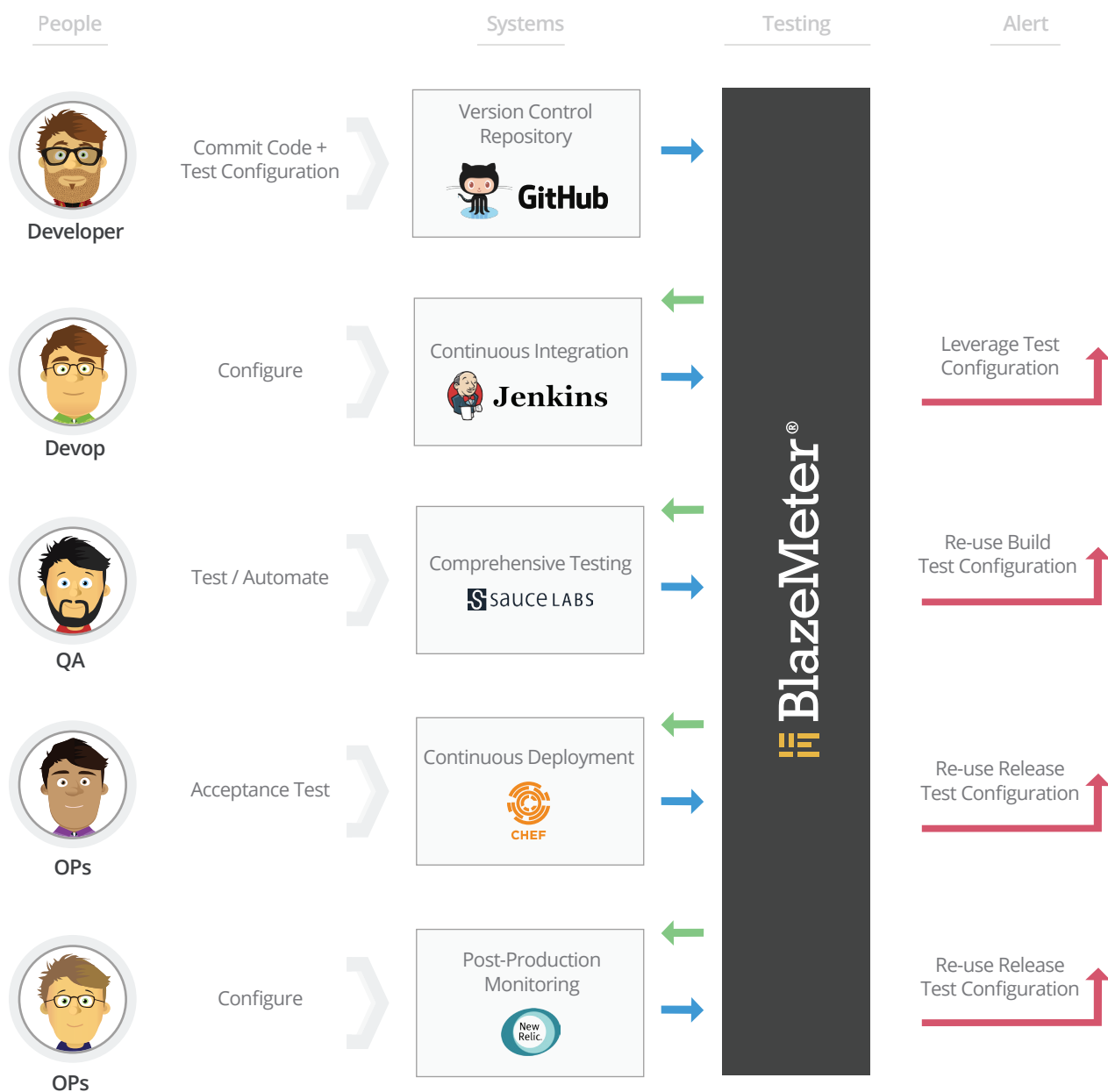
End to End Continuous Testing

When implemented correctly, Continuous Testing facilitates true continuity in the delivery cycle. Even tasks previously considered as ‘management tasks’, such as multi-geographical load tests, can now be performed as just another automated step in the delivery process.



Testing that was once cumbersome, complicated and time consuming can be completely streamlined, ensuring a predictable time-to-release and superior software quality.

The Testing Process - from Development to Operations



Red Arrow = Test Failed Green Arrow = Test Succeeded Blue Arrow = Sent to Testing





PART 5:

CONTINUOUS TESTING IN PRACTICE: REQUIREMENTS CHECKLIST

A successful continuous testing process requires the right tools and processes to be adopted across the various teams within the organization. Without the proper tools in place, the process will fail.

BlazeMeter developed the following checklist based on its experience helping clients adopt continuous testing. The checklist provides the features and functionality required for the continuous testing process to succeed, enabling teams to run tests at scale while also providing smooth integrations with various CI environments.

Tools, Scripting and Overall Test Creation

- Support for both manual and automated testing
- The provision of libraries and APIs to create “homegrown” tests using common DSLs.
- Ability to activate commercial tools (such as Perfecto Mobile)
- Support of JMeter and API tests (BlazeMeter)
- The ability to create browser-based tests (Selenium Builder, BlazeMeter Chrome Plugin)
- Support of Selenium and Appium tests (Sauce Labs)

Provisioning & Resources

- The option to run the same sets of tests with different provisioning (i.e. a developer can run a module test from behind the firewall with local resources, and the same test can run in production from multiple geographical locations on the public cloud)
- The ability to easily adopt a new provisioning scheme when a test configuration is executed in a different environment (e.g. Dev, CI, Pre-Prod, Post-Prod). These schemes should include:
 - The developer’s laptop for debugging and troubleshooting
 - A private cloud of resources available on-premise and protected behind the firewall
 - A public cloud of unlimited resources available on multiple geographical locations
- Sufficient resources in the public and private cloud, enabling organizations to run multiple tests in parallel with zero time to test
- The option for users to define which OS and Browser combinations they want to execute the test against
- The ability to start testing as part of the CI process (i.e. Jenkins, Travis, TeamCity, CircleCI etc.)

Automation

- The ability to run any test configuration, a test, or a set of tests using simple API calls
- The option to run as many tests as required in parallel
- Availability of all test data via a REST API



On-Demand Testing and Re-Running Failed Use Cases

- The ability to run any type of test on-demand. This is critical for test development, debugging, troubleshooting and identifying the root cause of a failure
- The option to automate test executions and run the same test on-demand

Modules, Builds, Release Candidates, Releases and Production

- The ability to combine different configuration fragments into one test configuration in order to comprehensively test build, release, and production snapshots.

Version Control Friendly Incremental Testing

- The ability to support version control, incremental testing and associate test configurations, sets of tests, and tests with versions (e.g code, build, RC, releases).

Automatic 'Failure' Alerts

- Automatic indications of failures, with an alerting scheme per developer, module and project
- The ability to gather all test artifacts and immediately send to relevant people
- The option to run the failed tests again to identify the root cause

Reporting

- A seamless integration with existing reporting solutions (e.g. Jenkins Performance Trend)
- The option to group tests by builds
- The ability to give jobs a 'pass' or 'fail' status and group accordingly
- Pass/fail trend reports
- "Deep dive" reports for diagnostics and analysis, which can overlay data from third party systems (e.g. NewRelic, CloudWatch) for a comprehensive picture.





CONCLUSION:

To compete in today's environment, it's critical to act and react fast in order to keep up with the rapid pace of change. As such, Continuous Delivery is more than just a new working process, it's a strategic capability that provides organizations with a strong competitive advantage. This 'advantage' is rapidly becoming the new benchmark; companies that don't adopt agile processes like Continuous Delivery risk falling beneath today's standards of acceptability.

However, the path to a successful Continuous Delivery process is still incomplete. Continuous Integration paved the way, rapidly followed by Continuous Deployment. Yet testing has remained the roadblock in Continuous Delivery's path, thwarting attempts to reduce time-to-market without quality compromises.

The only viable solution for IT organizations is to integrate a fully automated Continuous Testing process into their software development lifecycle. There is no room in agile processes for manual handoffs and quality compromises - they must be fully automated and predictable. As such, Continuous Testing will become increasingly critical as an inherent piece of any successful Continuous Delivery process.

Find Out More:

To learn more about how BlazeMeter can help you with Continuous Testing, contact us at: info@blazemeter.com, [request a demo](#), call us on 1.855.445.2285, at or visit www.blazemeter.com





REFERENCES:

1. Perforce Software Study
2. Thoughtworks - The Case for Continuous Delivery
3. TechTarget - Waterfall Model
4. Wikipedia - Waterfall Model
5. StickyMinds - Testing in the Agile Age
6. BlazeMeter
7. Sauce Labs

