Continuous Testing: An IBM point of view

Execute tests when *needed*, not when *possible*

Marianne Hollier

June 16, 2016

Testing takes time and it is costly. Especially across multiple systems and platforms in the complex, hybrid cloud world. Teams that improve their test efficiency and effectiveness significantly reduce their expenses and the time it takes to get high-quality innovative solutions to end users. You can't test everything, and you can't automate everything, so it is critical to find the right subset of tests in the riskiest areas. Also key is service virtualization, which allows testing to begin as soon as a build is made, by mimicking missing dependencies. By combining test automation and service virtualization, teams test earlier, or "shift left," and continuously, so they gather feedback faster than ever.

Testing is costly

Software development and delivery is evolving from complex, monolithic applications, whose many dependencies are resolved at build-time, toward a more distributed, service-centric architecture whose dependencies can be resolved at runtime. Most enterprise applications are a combination of existing applications originally designed for a pre-cloud environment (also called systems of record) and new "systems of engagement" applications developed in the cloud. Their architectures tend to be complex due to their many dependencies, and they use APIs to bridge between the existing systems of record and the new systems of engagement. They leverage API management and cloud integration technologies to enable integration while addressing the organization security requirements. Their workloads can run across multiple environments: on-premises, private cloud, public cloud - the combination of which is an architecture also referred to as hybrid cloud.

> *“ Continuous testing enables a project team to execute tests when needed, not when possible. ”*

Hybrid cloud architecture is becoming the norm for both cloud-enabled and cloud-native applications. Hybrid cloud provides flexibility in deployment, enabling organizations to choose the right platform to run their workloads (DevOps for hybrid cloud: An IBM point of view, February 2016). IDC predicts that 80% of enterprise IT companies will commit to hybrid cloud architectures by 2017 (IDC FutureScape: Worldwide Cloud 2016 Predictions — Mastering the Raw Material of Digital Transformation, November 2015).

The IBM view is that testing improvements afford companies significant savings, and a competitive edge. Testing is an area ripe for better processes — as the average spending on quality assurance as a percentage of the total IT budget is projected to rise to 40 percent by 2018. Costs are aggravated by the fact that many organizations spend more than a third of their testing budget on test environments (World Quality Report 2015-16: Capgemini, Sogeti, HP).

A recent survey conducted by ADT found that testing is by far the number one reason for delays when deploying to production (ADT Study Whitepaper: Organizations Increasingly Look to Continuous Deployment to Recover). Using analytics and testing insight, IBM focuses on optimizing testing and the related deployment operations so that more resources remain available for innovation.

Why strive for Continuous Testing?

For businesses to deliver high-quality innovative solutions to the market quickly, the whole delivery team needs to encourage and embrace all feedback. These feedback channels need to occur not just between development and operations teams, but across the entire delivery ecosystem – including business analysts, developers, designers, architects, testers, release managers, third-party vendors, etc. – and the business stakeholders. The business stakeholders ask testers to validate that defined processes and transactions will operate as expected. The testers look for ways to minimize the cost and impact of testing activities. Continuous testing enables the trust of the whole team by providing immediate feedback on the quality of the solution. For the business stakeholders, continuous testing increases the confidence that a delivery can be relied upon and that there is minimum risk on business impact. For a project delivery team, continuous testing techniques and tools minimize the impact of testing which can reduce project costs, enable faster delivery of the solution and most importantly, ensure that a high quality, reliable solution is delivered.

The need for quality and speed

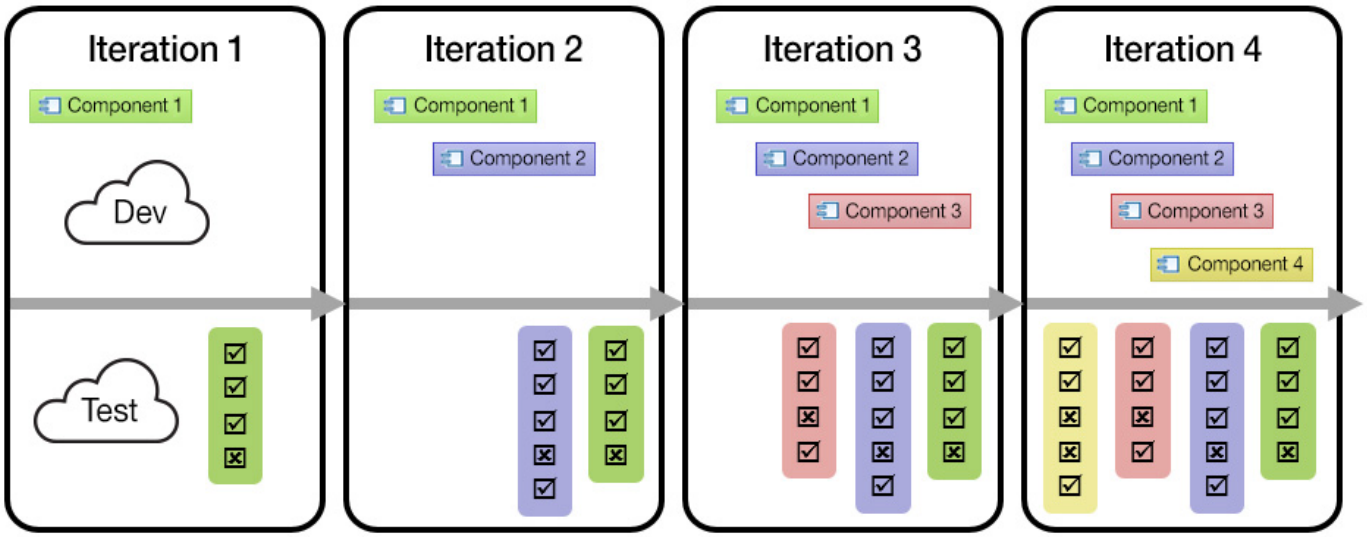
Business stakeholders ask project teams to deliver new applications, integrations, migrations and changes as quickly as possible. In turn, project teams ask their testers to validate that the:

- Technical environment works as required
- Business processes and transactions run as expected
- Solution scales to the expected usage
- Application is secure and user data is protected

Regardless of the industry or technologies involved, there is a careful balance between speed and quality. With the increased acceptance of cloud technologies, software teams have more tools than ever at their fingertips to make their work more efficient. However, while software and system developers do the best they can to avoid making mistakes, human errors are inevitable, and are why we test. The risk created by NOT testing greatly outweighs the cost of performing even a small number of tests.

With the increasing adoption of agile and DevOps practices, re-executing manual tests in every iteration is not a sustainable pattern. There just isn't enough time and adding more personnel to

execute manual regression tests results in diminishing returns. Even more importantly, the slow speed of feedback to the developers decreases productivity significantly. Test effectiveness is a critical aspect to keeping up with the faster-paced development lifecycles. It is optimized by running the fewest number of tests that find the largest number of problems. Even teams working in more traditional development lifecycles, where all the testing is performed in a single phase, have found that they cannot keep up with the regression testing each time they get a new build – with defect fixes, changes to existing features and even new functionality all bundled into the new build. The illustration below shows that after just a few iterations, the quantity of new features, and therefore number of tests, has increased significantly.



The only way to keep up with the needed regression testing is to automate the right set of tests. To accomplish this balance, mature DevOps teams use a combination of test automation and manual exploratory testing, both running in a continuous pattern.

Find the right set of tests

Attempting to automate all tests is impossible, and the cost of trying to do so eventually outweighs the benefit. For any reasonably complex application, you cannot test every possible path through the system because even if there is only a single loop in the application, the number of possible paths becomes infinite. If you then add in the test data permutations, you quickly realize that attempting to test everything is just not feasible. The key is to identify the most important subset of tests – the ones where you:

- Typically find issues
- Have seen regressions
- Have customer complaints
- Know the occurrence of a failure would be significant or even catastrophic

Impact analysis of the new code changes is a critical aspect in identifying which regression tests to run. But without good change set input, the data and analysis of the code changes can be misleading. This analysis of what are the *right* tests to automate should involve the entire team,

from business to development, to test, to operations, and support. Each role brings a different perspective on where things can and do go wrong, so it's important to include everyone.

Some examples of where test automation can be applied are:

- Data-driven tests where the data sets are complex and cover critical aspects
- Business logic validation to ensure the correct results are achieved
- Integration with a third-party system, where developers and testers might not have full access to that third-party system
- Low-intensity performance testing. Running small-scale stress, load, volume or memory leak checks across builds to identify degradations early, well before conducting formal load testing
- Installation and upgrade of customer-installed software across the many platforms and operating systems for commercially available software
- Scanning for security vulnerabilities, especially when financial or personal information is at risk

Test automation implementation comes in all shapes and forms, some key examples are:

- Unit testing
- Functional testing at the user interface (UI) layer
- Functional testing via APIs
- Performance testing, via the UI or API
- Security testing

In addition, there are tests that are extremely valuable when conducted manually. Continue these tests in parallel with test automation:

- **Exploratory testing.** Unscripted tests where the tester analyzes different aspects of the system without a prescribed end result. This helps find new scenarios that carry defects, but are not yet covered by automated tests.
- **Usability testing.** End-users are asked to test specific aspects of the system and give verbal feedback as they progress. This allows the team to better understand what users are thinking when they use the system.

Test smarter and "Shift Left"

Have you ever been told, "Test smarter, not harder!"? Unfortunately, test and development managers send this message to their teams frequently. But, they don't provide guidance on how to accomplish this *smarter* testing. Teams are so busy trying to keep up with their existing testing that they don't have time to figure it out for themselves.

A key aspect of testing smarter is to test earlier and more often, or *shifting test left*, in the delivery lifecycle. This way, the team can test the riskiest elements early, and those tests can then be continuously reused. Providing early, iterative feedback on code quality directly to development teams to ensure that fewer problems are found late in the lifecycle where they are more expensive to fix is a smarter approach.

Imagine a situation where a live application has low quality and bad reviews. Here is a scenario that describes how the project team can improve their ability to test and perform test automation when needed, not just when possible.

First, all stakeholders (business, development, test, operations, etc.) work together to understand the root causes of post-production defects. Using defect data and analysis, the team discovers that the most significant defects arise from two areas:

- Integration with other systems
- Response time delays

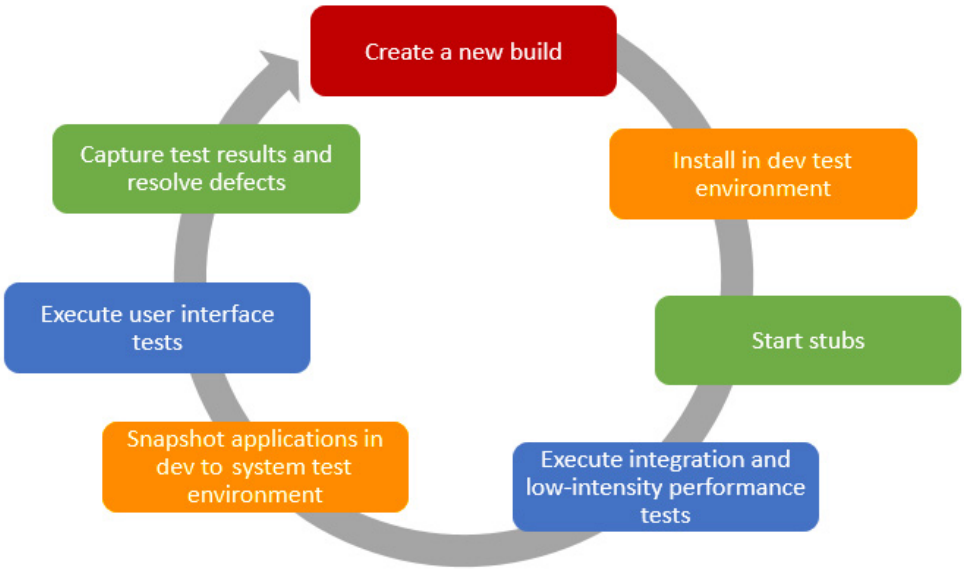
The team decides that they need to test the high-risk integrations earlier in the development process, and not wait until just before deployment to production, so they collaborate and capture the agreed-upon interface and data definitions as part of their system architecture.

They also decide to conduct some low-intensity performance testing as soon as a build is available, so critical performance issues can be identified and fixed much earlier. They'll track these performance tests across each build so they can immediately see if there is any degradation in performance.

Based on the interfaces, the developers and testers work together to create virtual services, or stubs, for each high-risk interface. The interface stubs mimic the other systems and allow the developers and testers to test the high-risk areas earlier in the development process – as soon as the code is written and built.

The team then automates many of the functional integration tests, and key response time tests, since those are where most of the significant production defects came from.

Now whenever a new build is made, the successful build process triggers the automated activities shown below:



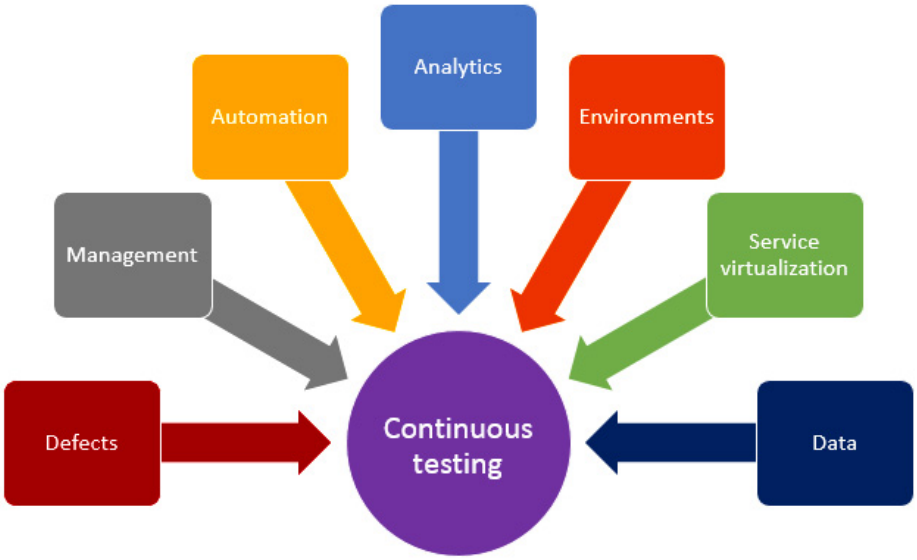
1. The new application build is installed in the automatically provisioned development cloud-based test environment.
2. The stubs for missing dependent services are started.
3. An automated integration test suite is triggered for execution, followed by the low-intensity performance tests.
4. The test results are captured and feedback is made available to the entire team. The team analyzes the results for any regressions, especially around performance.
5. Provided the integration and performance tests pass, a snapshot of the applications in the development test environment is deployed to the system test environment.
6. The automated user interface based test suites are triggered and executed in the system test environment.
7. Again, the test results are captured, feedback is provided, defects are resolved, and a new build is created.

After the dependent systems are available, the team runs those same automated integration and performance tests again, against the real systems, to confirm their application still behaves as expected.

This scenario can be applied to any pattern of defects. It enables whole teams to work together to not only start testing earlier in the lifecycle, but also to automate the most important tests first, so that those tests are executed repeatedly. This ensures good test coverage of the riskiest parts of the application.

Achieve Continuous Testing

Building a continuous testing culture requires people, practices, tools, and time. The following diagram shows the practices that are typical in creating a continuous testing process.



When applying a traditional testing approach, defects are the initial communication channel between testers and developers. Testers think "I found a bug!" and developers then agree or disagree that there is a problem with the code. Many times the defect isn't a problem with the code itself, but with the requirements, design or even the test. Sometimes, the problem lies outside the application itself – in the test environment, the test data, the test script itself, or some combination of these things. Having a collaborative environment to log and track defects is essential in any software development lifecycle, but tracking defects is just the tip of the iceberg when it comes to adopting the right set of practices.

One of the next practices that test teams typically adopt is test management – planning the testing effort, identifying needed tests, creating manual tests, gathering existing tests, executing the tests, and tracking and reporting on the progress of testing. This test management practice helps both teams and management understand at a glance if tests are passing, failing, or blocked. It also answers the question: *Is the test effort on schedule, or behind, or even ahead?*

Test automation practices typically come next, when teams discover that running all of their tests manually is inefficient, ineffective, and in many cases downright impossible. Successfully creating a robust and maintainable test automation framework can be difficult if not approached as a software development project itself. There needs to be a shared vision, requirements, architecture, design, even coding in some instances, and finally validation that the automation does what was intended. Without these aspects, test automation frameworks tend to be fragile, brittle, difficult to maintain and costly to refactor, and are frequently abandoned.

Alongside test automation, test analytics and insights practices start to grow to answer questions like: *Which tests should we run, and when? Why are we running them?* Code change impact analysis can be a difficult task, especially if the developers are not rigorous and consistent with their code change sets. Understanding what has changed from the last build is critical for selecting the right sets of tests to run. Without that analysis, over testing – or re-testing everything is a tempting but costly answer.

Another critical aspect of testing is creating the test environments. Automating the provisioning of test environments has huge benefits. Automated creation and configuration of test environments decreases the time it takes to start testing a new build from hours, or even days or weeks, to minutes. This automation also reduces the number of false errors due to test environment issues, incorrectly installed dependent software, and other manual processes that introduce problems. In the IBM DevOps story, test automation goes hand-in-hand with automated deployment.

In conjunction with the test execution and test environment automation, service virtualization, or *stubbing*, greatly increases the effectiveness of the test effort. By creating virtual services from known and agreed-upon interfaces, developers and testers write code and test against the same interface, even when that dependent system is not available. Executing tests against these known interfaces enables testing to begin much earlier and to run much more frequently – on every build. Service virtualization also allows testing of scenarios that might not be readily tested with a live system – for example:

- Exceptions and errors
- Missing data
- Delayed response times
- Large volumes of data or users

Incorporating service virtualization into the overall test effort enables teams to build and effectively test the riskiest parts of the system first, instead of just building and testing the easy-to-test parts of the system. When you then automate tests for those risky parts of the system and execute them on each build, you get better coverage of those parts. This ensures regressions are identified as quickly as a new build is made.

A typical scenario might be that mobile and web development teams work on cloud applications and mainframe teams work on premises. With service virtualization, IBM decouples the environment dependencies in these hybrid cloud scenarios. Teams then build and test at their own speeds and adopt the right DevOps approach that fits with their culture.

Another aspect of testing that increases test effectiveness is having the right sets of test data. Using test data that is as production-like as possible provides better coverage with more scenarios. Extracting data from a production environment and masking it for security purposes provides a realistic set of test data. A good set of test data also enables the team to create exception or error situations that might be difficult to test otherwise.

Working together, a set of effective testing practices enables whole teams, not just testers, to improve quality and decrease the time it takes to deliver new capabilities. These practices remove dependencies and bottlenecks to allow testing earlier in the lifecycle and enable continuous testing -- without the prohibitive investments typically required by traditional testing environments.

Conclusion

The desire for "quality at speed" has never been more relevant to companies adopting a hybrid cloud strategy – with some projects developed in the cloud, and some developed on premises, and both needing to work seamlessly together when deployed.

We call the combination of test automation and service virtualization "Continuous Testing." Moving the testing earlier and more often in the lifecycle – "shifting left" – enables teams to build, deploy, test, and gather feedback – all continuously. Providing early, iterative feedback on code quality directly to development teams helps ensure that fewer problems are found late in the lifecycle where they are more expensive to fix. And when problems are discovered in production, not only are they extremely costly to resolve, but they can significantly harm a company's reputation, and even have a lasting impact on customer loyalty. Without timely testing and feedback, companies cannot truly deliver quality at speed.

Additional contributors:

- Glyn Rhodes, Offering Management Lead – Continuous Test
- James Hunter, Offering Lead – Industry Solutions and SAP
- Roger LeBlanc, Sr. Offering Manager – Performance Testing and UI Automation
- Matt Tarnawsky, Offering Manager – API Testing and Service Virtualization

Related topics

- IBM Continuous Testing
- "Shift Left" for higher quality at greater speed
- Test early test often Continuous testing as part of the DevOps lifecycle