DevOps: Are You Pushing Bugs to Your Clients Faster?

Wayne Ariola

wayne.ariola@parasoft.com

Abstract

Now that rapid delivery of differentiable software has become a business imperative, software development teams are scrambling to keep up. In response to increased demand, they are seeking new ways to accelerate their release cycles—driving the adoption of agile or lean development practices such as DevOps. Yet, based on the number of software failures now making headlines on a daily basis, it's evident that speeding up the Software Development Lifecycle (SDLC) opens the door to severe repercussions.

Organizations are remiss to assume that yesterday's practices can meet today's process demands. There needs to be a cultural shift from *testing an application* to *understanding the risks associated with a release candidate*. Such a shift requires moving beyond the traditional "bottom-up" approach to testing, which focuses on adding incremental tests for new functionality. While this will always be required, it's equally important to adopt a top-down approach to mitigating business risks. This means that organizations must defend the user experience with the most likely use cases in the context of non-functional requirements—continuously.

In order for more advanced automation to occur, we need to move beyond the test pass/fail percentage into a much more granular understanding of the impact of failure: a nuance that gets lost in the traditional regression test suite. Continuous Testing is key for bridging this gap. Continuous Testing brings real-time assessments, objective go/no-go quality gates, and continuous measurements to refine the development process so that business expectations are continuously met. Ultimately, Continuous Testing resets the question from "are you done testing?" to "is the level of risk understood and accepted?" This paper is designed to help software development leaders understand how a new perspective on "test" can help them accelerate the SDLC and release with confidence.

Biography

Wayne Ariola, Chief Strategy Officer at Parasoft, leads the development and execution of Parasoft's long-term strategy. He leverages customer input and fosters partnerships with industry leaders to ensure that Parasoft solutions continuously evolve to support the ever-changing complexities of real-world business processes and systems. Ariola has contributed to the design of core Parasoft technologies and has been awarded several patents for his inventions. He has a BA from UC Santa Barbara and an MBA from Indiana University.

Cynthia Dunlop, Lead Technical Writer at Parasoft, authors technical articles, documentation, white papers, case studies, and other marketing communications—currently specializing in service virtualization, API testing, DevOps, and continuous testing. She has also co-authored and ghostwritten several books on software development and testing for Wiley and Wiley-IEEE Press. Dunlop holds a BA from UCLA and an MA from Washington State University.

1 Introduction

In response to today's demand for "Continuous Everything," the software delivery conveyer belt keeps moving faster and faster. However, considering that testing has been the primary constraint of the software delivery process, it's unreasonable to expect that simply speeding up a troubled process will yield better results. (I Love Lucy fans: Just think of Lucy and Ethel at the candy factory, struggling to keep pace as the conveyer belt starts putting out chocolates faster and faster.)

Now that rapid delivery of differentiable software has become a business imperative, software development teams are scrambling to keep up. In response to increased demand, they are seeking new ways to accelerate their release cycles—driving the adoption of agile or lean development practices such as DevOps. Yet, based on the number of software failures now making headlines on a daily basis, it's evident that speeding up the SDLC opens the door to severe repercussions.

Organizations are remiss to assume that yesterday's practices can meet today's process demands. There needs to be a cultural shift from *testing an application* to *understanding the risks associated with a release candidate*. Such a shift requires moving beyond the traditional "bottom-up" approach to testing, which focuses on adding incremental tests for new functionality. While this will always be required, it's equally important to adopt a top-down approach to mitigating business risks. This means that organizations must defend the user experience with the most likely use cases in the context of non-functional requirements—continuously.

In order for more advanced automation to occur, we need to move beyond the test pass/fail percentage into a much more granular understanding of the impact of failure: a nuance that gets lost in the traditional regression test suite. Continuous Testing, which involves automatically executing a set of tests designed to verify whether business expectations for functionality, security, reliability, and performance are satisfied, is key for bridging this gap. It shifts the paradigm from a pure bottom-up approach to a hybrid model in which top-down testing is leveraged to protect the expected user experience from changes introduced as the application evolves. Ultimately, Continuous Testing resets the question from "are you done testing?" to "is the level of risk understood and accepted?"

2 Testing: The Elephant in the Room

As organizations begin to accelerate the SDLC, process bottlenecks will become evident. One of the bottlenecks that continues to plague SDLC acceleration is testing. At best, testing has been considered inevitable overhead—a "time-boxed" event that occurs sometime between code complete and the target release date. At worst, organizations have pushed quality processes to post-release, forcing a "real-time customer acceptance test."

Testing has always been the elephant in the room. Psychologically, the malleable nature of software has given organizations an excuse to defer investment in testing. However, this deferral results in technical debt. Over time, technical debt compounds, escalating the risk and complexity associated with each release.

Another obstacle to SDLC acceleration is the lack of a coordinated, end-to-end quality process. If trustworthy and holistic quality data were collected throughout the SDLC, then more automated decision points could optimize downstream processes.

3 Continuous Testing: How it Helps

Continuous Testing provides a real-time, objective assessment of the business risks associated with an application under development. Applied uniformly, it allows both business and technical managers to make better trade-off decisions between release scope, time, and quality.

Continuous Testing is NOT simply more automation. Rather, it is the reassessment of software quality practices—driven by an organization's cost of quality and balanced for speed and agility. Ultimately, Continuous Testing can provide a quantitative assessment of risk and produce actionable tasks that will help mitigate these risks before progressing to the next stage of the SDLC.

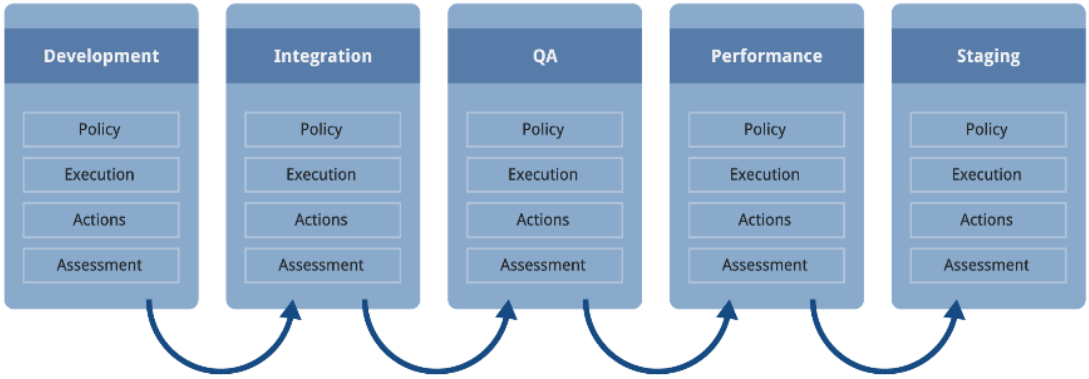

Figure 1: Mitigate risks before they progress to the next stage of the SDLC

For example, assume that a retailer knows that very specific user experience factors related to the application make the consumer more likely to add items to their e-commerce shopping cart. These metrics, which are defined in business language and measured automatically, are the exact same benchmarks used to accept a software release candidate. The question that the test suite should answer is how application modifications impact this set of business metrics. The goal is to bring the business expectations and business language to the forefront of the validation process—ultimately, protecting the user and the brand.

When it comes to software quality, we are confronting the need for true process re-engineering. Continuous Testing is not a "plug and play" solution. As with all process-driven initiatives, it requires the evolution of people, process, and technology. We must accommodate the creative nature of software development as a discipline, yet we must face the overwhelming fact that software permeates every aspect of the business—and software failure presents the single greatest risk to the organization.

4 What's the Business Value of Continuous Testing?

Given the business expectations at each stage of the SDLC, Continuous Testing delivers a quantitative assessment of risk as well as actionable tasks that help mitigate these risks before they progress to the next stage of the SDLC. The goal is to eliminate meaningless tests and produce value-added tasks that truly drive the development organization towards a successful release.

Continuous Testing—when executed correctly—delivers three major business benefits.

First, continuous tests are the artifacts that drive a central system of decision for the SDLC. Continuous tests place a bracer around core business functionality as expressed and implemented in software. The assessment of the validity or stability of these core business artifacts provides a real-time, quantifiable assessment of the application's health.

Second, Continuous Testing establishes a safety net that allows software developers to bring new features to market faster. With a trusted test suite ensuring the integrity of dependent application components and related functionality, developers can immediately assess the impact of code changes. This not only accelerates the rate of change, but also mitigates the risk of software defects reaching your customers.

Third, Continuous Testing allows managers to make better trade-off decisions. From the business' perspective, achieving a differentiable competitive advantage by being first to market with innovative software drives shareholder value. Yet, software development is a complex endeavor. As a result, managers are constantly faced with trade-off decisions between time, functionality, and quality. By providing a holistic understanding of the risk of release, Continuous Testing helps to optimize the business outcome. We'll cover this in greater detail in section 6.1 of this paper.

5 Re-Evaluating the Cost of Quality

A critical motivator for the evolution towards Continuous Testing is that business expectations about the speed and reliability of software releases have changed dramatically—largely because software has morphed from a business process enabler into a competitive differentiator.

For example, APIs represent componentized pieces of software functionality which developers can either consume or write themselves. In a recent Parasoft survey about API adoption, over 80% of respondents said that they have stopped using an API because it was "too buggy." Moreover, when we asked the same respondents if they would ever consider using that API again, 97% said "no." With switching costs associated with software like an API at an all-time low, software quality matters more than ever.

Another example is mobile check deposit applications. In 2011, top banks were racing to provide this must-have feature. By 2012, mobile check deposit became the leading driver for bank selection (Zhen, 2012). Getting a secure, reliable application to market was suddenly business critical. With low switching costs associated with online banking, financial institutions unable to innovate were threatened with customer defection.

This sea change associated with the quality expectations of software is also reflected in the manner in which software failures are reported. Today, software failures are highlighted in news headlines as organizational failings with deep-rooted impacts on C-level executives and stock prices. Parasoft analyzed the most notable software failures in 2012 and 2013; each incident initiated an average -3.35% decline in stock price, which equates to an average of negative $ 2.15 billion loss of market capitalization. This is a tremendous loss of shareholder value. Additionally, looking at organizations that endured multiple news-worthy software failures, it is clear that the market punishes repeat offenders even more aggressively. Repeat offenders suffered an average -5.68% decline in stock price, which equates to an average of negative $ 2.65 billion loss of market capitalization.[1]

The bottom line is that we must re-evaluate the cost of quality for our organizations and individual projects. If your cost of quality assessment exposes a gap in your quality process, it's a sign that now is the time to reassess your organization's culture as it relates to building and testing software. In most organizations, quality software is clearly the intention, yet the culture of the organization yields trade-off decisions that significantly increase the risk of exposing faulty software to the market.

6 What's Needed for Continuous Testing?

As you begin the transformation to Continuous Testing, the following elements are necessary for achieving a real-time assessment of business risks.

[1] From Parasoft equity analysis of the most notable software failures of 2012-2013.

Figure 2: Elements of Continuous Testing

6.1 Risk Assessment—Are You Ready to Release?

As we review the elements of Continuous Testing, it's hard to argue that one element is more important than the rest. If we present our case well enough, it should become obvious that each element is critical for overall process success. However, we need a place to start— and establishing a baseline to measure risk is the perfect place to begin as well as end.

One overarching aspect to risk assessment associated with software development is continuously overlooked: If software is the interface to your business, then developers writing and testing code are making business decisions on behalf of the business.

Assessing the project risk upfront should be the baseline by which we measure whether we are done testing and allow the SDLC to continue towards release. Furthermore, the risk assessment will also play an important role in improvement initiatives for subsequent development cycles.

The definition of risk cannot be generic. It must be relative to the business, the project, and potentially the iterations in scope for the release candidate. For example, a non-critical internal application would not face the same level of scrutiny as a publically-exposed application that manages financial or retail transactions. A company baseline policy for expectations around security, reliability, performance, maintainability, availability, legal, etc. is recommended as the minimum starting point for any development effort. However, each specific project team should augment the baseline requirement with additional policies to prevent threats that could be unique to the project team, application, or release.

SDLC acceleration requires automation. Automation requires machine-readable instructions which allow for the execution of prescribed actions (at a specific point in time). The more metadata that a team can provide around the application, components, requirements, and tasks associated with the release, the more rigorous downstream activities can be performed for defect prevention, test construction, test execution, and maintenance.

6.1.1 Technical Debt

Core to the management of SDLC risk is the identification of technical debt. A Development Testing Platform will help prevent and mitigate types of technical debt such as poorly-written code, overly-complex code, obsolete code, unused code, duplicate code, code not covered by automated tests, and incomplete code. The uniform measurement of technical debt is a great tool for project comparison and should be a core element of a senior manager's dashboard.

6.1.2 Risk Mitigation Tasks

All quality tasks requested of development should be 100% correlated to an opportunity to minimize risk or technical debt and related to a defect prevention policy (discussed in section 6.2). Policies can be formed around any aspect of the development process. For example, policies might cover non-functional requirements related to security, reliability, performance, compliance, etc. To be effective, policies must be definable, enforceable, measureable and auditable.

A developer has two primary jobs: implement business requirements and reduce the business risk associated with application failure. From a quality and testing perspective, it is crucial to realize that quality initiatives generally fail when the benefits associated with a testing task are not clearly understood.

A risk mitigation task can range from executing a peer code review to constructing or maintaining a component test. Whether a risk mitigation task is generated manually at the request of a manager or automatically (as with static code analysis), it must present a development or testing activity that is clearly correlated with the reduction of risk.

6.1.3 Coverage Optimization

Coverage is always a contentious topic—and, at times, a religious war. Different coverage techniques are better-suited for different risk mitigation goals. Fortunately, industry compliance guidelines are available to help you determine which coverage metric or technique to select and standardize around.

Once a coverage technique (line, statement, function, modified condition, decision, path, etc.) is selected and correlated to a testing practice, the Development Testing Platform will generate reports as well as tasks that guide the developer or tester to optimize coverage. The trick with this analysis is to optimize versus two goals. First, if there is a non-negotiable industry standard, optimize based on what's needed for compliance. Second (and orthogonal to the first), optimize on what's needed to reduce business risks.

Coverage analysis is tricky because it is not guaranteed to yield better quality. Yet, coverage analysis can certainly help you make prioritization decisions associated with test resource allocation.

6.1.4 Test Quality Assessment

Processes and test suites have one thing in common: over time, they grow in size and complexity until they reach a breaking point when they are deemed "unmanageable." Unfortunately, test suite rationalization is traditionally managed as a batch process between releases. Managing in this manner yields to sub-optimal decisions because the team is forced to wrangle with requirements, functions, or code when the critical details are no longer fresh in their minds.

Continuous Testing requires reliable, trustworthy tests. When test suite results become questionable, there is a rapid decline in how and when team members react. This leads to the test suite becoming out-of-sync with the code—and ultimately out of control.

With this in mind, it is just as important to assess the quality of the test itself as it is to respond to a failing test. Automating the assessment of the test is critical for Continuous Testing. Tests lie at the core of software risk assessment. If these risk monitors are not reliable, then we must consider the process to be out of control.

6.2 Policy Analysis—Keep up with Evolving Business Demands

Policy analysis through a Development Testing Platform is key for driving development and testing process outcomes. The primary goal of process analysis to ensure that policies are meeting the organization's evolving business and compliance demands.

Most organizations have a development or SDLC policy that is passive and reactive. This policy might be referenced when a new hire is brought onboard or when some drastic incident compels management to consult, update, and train on the policy. The reactive nature of how management expectations are expressed and measured poses a significant business risk. The lack of a coordinated governance mechanism also severely hampers IT productivity (since you can't improve what you can't measure).

Policy analysis through a Development Testing Platform is the solution to this pervasive issue. With a central interface where a manager or group lead defines and implements “how,” “when,” and “why” quality practices are implemented and enforced, management can adapt the process to evolving market conditions, changing regulatory environments, or customer demands. The result: management goals and expectations are translated into executable and monitor-able actions. One such example of a Development Testing Platform is the Parasoft Development Testing Platform, which monitors policies using a number of software testing practices (e.g., static analysis, unit testing, coverage analysis, runtime error detection, etc.) across distributed teams and throughout the SDLC.

Policy analysis is the measurement of the policy itself. For example, if a particular security policy such as "validate every input as soon as it is received" is enforced via static analysis, measuring that policy might involve considering:

- How is the static analysis rule violated?
- How many times is it violated?
- Is there quantifiable evidence that following the related coding standard actually helps the application meet business expectations associated with security?

Ultimately, policy analysis is part of an infrastructure that promotes continuous process improvement.

The primary business objectives of policy analysis are:

- Expose trends associated with the injection of dangerous patterns in the code
- Target areas where risks can be isolated within a stage
- Identify higher risk activities where defect prevention practices need to be augmented or applied

With effective policy analysis, “policy” is no longer relegated to being a reactive measure that documents what is assumed to occur; it is promoted to being the primary driver for risk mitigation.

As IT deliverables increasingly serve as the “face” of the business, the inherent risks associated with application failure expose the organization to severe financial repercussions. Furthermore, business stakeholders are demanding increased visibility into corporate governance mechanisms. This means that merely documenting policies and processes is no longer sufficient; we must also demonstrate that policies are actually executed in practice.

This centralization of management expectations not only establishes the reference point needed to analyze risk, but also provides the control required to continuously improve the process of delivering software.

6.3 Requirements Traceability—Defending the Business Objective

All tests should be correlated with a business requirement. This provides an objective assessment of which requirements are working as expected, which require validation, and which are at risk. This is tricky because the articulation of a requirement, the generation or validation of code, and the generation

of a test that validates its proper implementation all require human interaction. We must have ways to ensure that the artifacts are aligned with the true business objective—and this requires human review and endorsement. Continuous Testing must promote the validation of testing artifacts via peer review.

A Development Testing Platform helps the organization keep business expectations in check by ensuring that there are *effective* tests aligned to the business requirement. By allowing extended metadata to be associated with a requirement, an application, a component, or iteration, the Development Testing Platform will also optimize the prioritization of tasks.

During "change time," continuous tests are what trigger alerts to the project team about changes that impact business requirements, test suites, and peripheral application components. In addition to satisfying compliance mandates, such as safety-critical, automotive, or medical device standards, real-time visibility into the quality status of each requirement helps to prevent late-cycle surprises that threaten to derail schedules and/or place approval in jeopardy.

6.4 Advanced Analysis—Expose Application Risks Early

6.4.1 Defect Prevention with Static Analysis

It's well known that the later in the development process a defect is found, the more difficult, costly, and time-consuming it is to remove. Mature static analysis technologies, managed in context of defined business objectives, will significantly improve software quality by preventing defects early.

Writing code without static code analysis is like writing a term paper or producing a report without spell check or grammar check. A surprising number of high-risk software defects are 100% preventable via fully-automated static code analysis. By preventing defects from being introduced in the first place, you minimize the number of interruptions and delays caused by the team having to diagnose and repair errors. Moreover, the more defects you prevent, the lower your risk of defects slipping through your testing procedures and making their way to the end-user—and requiring a significant amount of resources for defect reproduction, defect remediation, re-testing, and releasing the updated application. *Ultimately, automated defect prevention practices increase velocity, allowing the team to accomplish more within an iteration.*

At a more technical level, this automated analysis for defect prevention can involve a number of technologies, including multivariate analysis that exposes malicious patterns in the code, areas of high risk, and/or areas more vulnerable to risk. All are driven by a policy that defines how code should be written and tested to satisfy the organization's expectations in terms of security, reliability, performance, and compliance. The findings from this analysis establish a baseline that can be used as a basis for continuous improvement.

Pure "defect prevention" approaches can eliminate defects that result in crashes, deadlocks, erratic behavior, and performance degradation. A security-focused approach can apply the same preventative strategy to security vulnerabilities, preventing input-based attacks, backdoor vulnerabilities, weak security controls, exposure of sensitive data, and more.

6.4.2 Change Impact Analysis

It is well known that defects are more likely to be introduced when modifying code associated with older, more complex code bases. In fact, a recent FDA study of medical device recalls found that an astonishing "192 (or 79%) [of software-related recalls] were caused by software defects that were introduced when changes were made to the software after its initial production and distribution"(FDA, 2002).

From a risk perspective, changed code equates to risky code. We know that when code changes, there are distinct impacts from a testing perspective:

- Do I need to modify or eliminate the old test?

- Do I need a new test?
- How have changes impacted other aspects of the application?

The goal is to have a single view of the change impacts from the perspective of the project as well as the perspective of the individual contributor. Optimally, change impact analysis is performed as close to the time of change as possible—when the code and associated requirements are still fresh in the developer's or tester's mind.

If test assets are not aligned with the actual business requirements, then Continuous Testing will quickly become unmanageable. Teams will need to spend considerable time sorting through reported failures—or worse, overlook defects that would have been exposed by a more accurate test construction.

Now that development processes are increasingly iterative (more agile), keeping automated tests and associated test environments in sync with continuously-evolving system dependencies can consume considerable resources. To mitigate this challenge, it's helpful to have a fast, easy, and accurate way of updating test assets. This requires methods to assess how change impacts existing artifacts as well as a means to quickly update those artifacts to reflect the current business requirements.

6.4.3 Scope and Prioritization

Given a software project's scope, iteration, or release, some tests are certainly more valuable and timely than others. Advanced analysis techniques can help teams identify untested requirements, tasks, and code. Advanced analysis should also deliver a prioritized list of regression tests that need review or maintenance.

Leveraging this type of analysis and acting on the prioritized test creation or maintenance tasks can effectively prevent defects from propagating to downstream processes, where defect detection is more difficult and expensive. There are two main drivers for the delivery of tasks here: the boundaries for scope and the policy that defines the business risks associated with the application.

For example, the team might be working on a composite application in which one component is designed to collect and process payment cards for online transactions. The cost of quality associated with this component can be colossal if the organization has a security breach or fails a PCI DSS[2] audit. Although code within the online transaction component might not be changing, test metadata associated with the component could place it in scope for testing. Furthermore, a policy defined for the PCI DSS standard (as well as the organization's internal data privacy and security) will drive the scope of testing practices associated with this release or iteration.

6.5 Test Optimization—Ensure Findings are Accurate and Actionable

To truly accelerate the SDLC, we have to look at testing much differently. In most industries, modern quality processes are focused on optimizing the process with the goal of preventing defects or containing defects within a specific stage. With software development, we have shied away from this approach, declaring that it would impede engineering creativity or that the benefits associated with the activity are low, given the value of the engineering resources. With a reassessment of the true cost of software quality, many organizations will have to make major cultural changes to combat the higher penalties for faulty software. Older, more established organizations will also need to keep up with the new breed of businesses that were conceived with software as their core competency. These businesses are free from older cultural paradigms that might preclude more modern software quality processes and testing practices.

No matter what methodology is the best fit for your business objectives and desired development culture, a process to drive consistency is required for long-term success.

[2] PCI DSS is the Payment Card Industry Data Security Standard

Test optimization algorithms help you determine what tests you absolutely *must* run versus what tests are of lower priority given the scope of change. Ideally, you want intelligent guidance on the most efficient way to mitigate the greatest risks associated with your application. Test optimization not only ensures that the test suite is validating the correct application behavior, but also assesses each test itself for effectiveness and maintainability.

6.5.1 Management

Test optimization management requires that a uniform workflow is established and maintained associated with the policies defined at the beginning of a project or iteration. A Development Testing Platform must provide the granular management of queues combined with task workflow and measurement of compliance. To achieve this:

- The scope of prescribed tasks should be measurable at different levels of granularity, including individual, team, iteration, and project.
- The test execution queues should allow for the prioritization of test runs based on the severity and business risk associated with requirements.
- Task queues should be visible and prioritized with the option to manually alter or prioritize (this should be the exception, not the norm).
- Reports on aged tasks should be available for managers to help them determine whether the process is under control or out of control.

6.5.2 Construction

With a fragile test suite, Continuous Testing just isn't feasible. If you truly want to automate the execution of a broad test suite—embracing unit, component, integration, functional, performance, and security testing—you need to ensure that your test suite is up to the task. How do you achieve this? Ensure that your tests are…

- **Logically-componentized**: Tests need to be logically-componentized so you can assess the impact at change time. When tests fail and they're logically correlated to components, it is much easier to establish priority and associate tasks to the correct resource.
- **Incremental**: Tests can be built upon each other, without impacting the integrity of the original or new test case.
- **Repeatable:** Tests can be executed over and over again with each incremental build, integration, or release process.
- **Deterministic and meaningful**: Tests must be clean and deterministic. Pass and fail have unambiguous meanings. Each test should do exactly what you want it to do—no more and no less. Tests should fail only when an actual problem you care about has been detected. Moreover, the failure should be obvious and clearly communicate what went wrong.
- **Maintainable within a process:** A test that's out of sync with the code will either generate incorrect failures (false positives) or overlook real problems (false negatives). An automated process for evolving test artifacts is just as important as the construction of new tests.
- **Prescriptive workflow based on results:** When a test does fail, it should trigger a process-driven workflow that lets team members know what's expected and how to proceed. This typically includes a prioritized task list.

6.5.3 Test Data Management

Access to realistic test data can significantly increase the effectiveness of a test suite. Good test data and test data management practices will increase coverage as well as drive more accurate results. However, developing or accessing test data can be a considerable challenge—in terms of time, effort, and compliance. Copying production data can be risky (and potentially illegal). Asking database administrators to provide the necessary data is typically fraught with delays. Moreover, delegating this

task to dev/QA moves team members beyond their core competencies, potentially delaying other aspects of the project for what might be imprecise or incomplete results.

Thus, fast and easy access to realistic test data removes a significant roadblock. The primary methods to derive test data are:

- Sub-set or copy data from a production database into a staged environment and employ cleansing techniques to eliminate data privacy or security risks.
- Leverage Service Virtualization (discussed later in this resource) to capture request and response traffic and reuse the data for subsequent scenarios. Depending on the origin and condition of the data, cleansing techniques might be required.
- Generate test data synthetically for various scenarios that are required for testing.

In all cases, it's critical to ensure that the data can be reused and shared across multiple teams, projects, versions, and releases. Reuse of “safe” test data can significantly increase the speed of test construction, management, and maintenance.

6.5.4 Maintenance

All too often, we find development teams carving out time between releases in order to "clean-up" the test suites. This ad-hoc task is usually a low priority and gets deferred by high-urgency customer feature requests, field defects, and other business imperatives. The resulting lack of ongoing maintenance typically ends up eroding the team’s confidence in the test suite and spawning a backlog of increasingly-complex maintenance decisions.

Test maintenance should be performed as soon as possible after a new business requirement is implemented (or, in the case of TDD-like methodologies, prior to a requirement being implemented). The challenge is to achieve the optimal balance between creating and maintaining test suites versus the scope of change.

Out-of-sync test suites enter into a vicious downward spiral that accelerates with time. Unit, component, and integration tests that are maintained by developers are traditionally the artifacts at greatest risk of deterioration. Advanced analysis of the test artifact itself should guide developers to maintain the test suite. There are five primary activities for maintenance—all of which are driven by the business requirement:

- Delete the test
- Update the test
- Update the assertions
- Update the test data
- Update the test metadata

6.6 Service Virtualization—Eliminate Test Environment Access Issues

With the convergent trends of parallel development and increasing system complexity/interdependency, it has become extremely rare for a team to have ubiquitous access to all of the dependent applications required to execute a complete test. By leveraging Service Virtualization to remove these constraints, an organization can gain full access to (and control over) the test environment—enabling Continuous Testing to occur as early and often as needed.

Want to start testing the component you just built even though not much else is completed? Don't have 24/7 access to all the dependencies involved in your testing efforts—with all the configurations you need to feel confident that your test results are truly predictive of real-world behavior? Tired of delaying

performance testing because access to a realistic environment is too limited (or too expensive)? Service Virtualization can remove all these constraints.

With Service Virtualization, organizations can access simulated test environments that allow developers, QA, and performance testers to test earlier, faster, and more completely. Organizations that rely on interconnected systems must be able to validate system changes more effectively—not only for performance and reliability, but also to reduce risks associated with security, privacy, and business interruption. Service Virtualization is the missing link that allows organizations to continuously test and validate business requirements. Ultimately, Service Virtualization brings higher quality functionality to the market faster and at a lower cost.

7 Conclusion

Admittedly, moving from automated testing to the level of business-driven Continuous Testing outlined above is a big leap. But it is one that yields significant benefits from a DevOps perspective. Through objective real-time validation of whether software meets business expectations at various "quality gates," organizations can automatically promote software from one phase of the SDLC to the next.

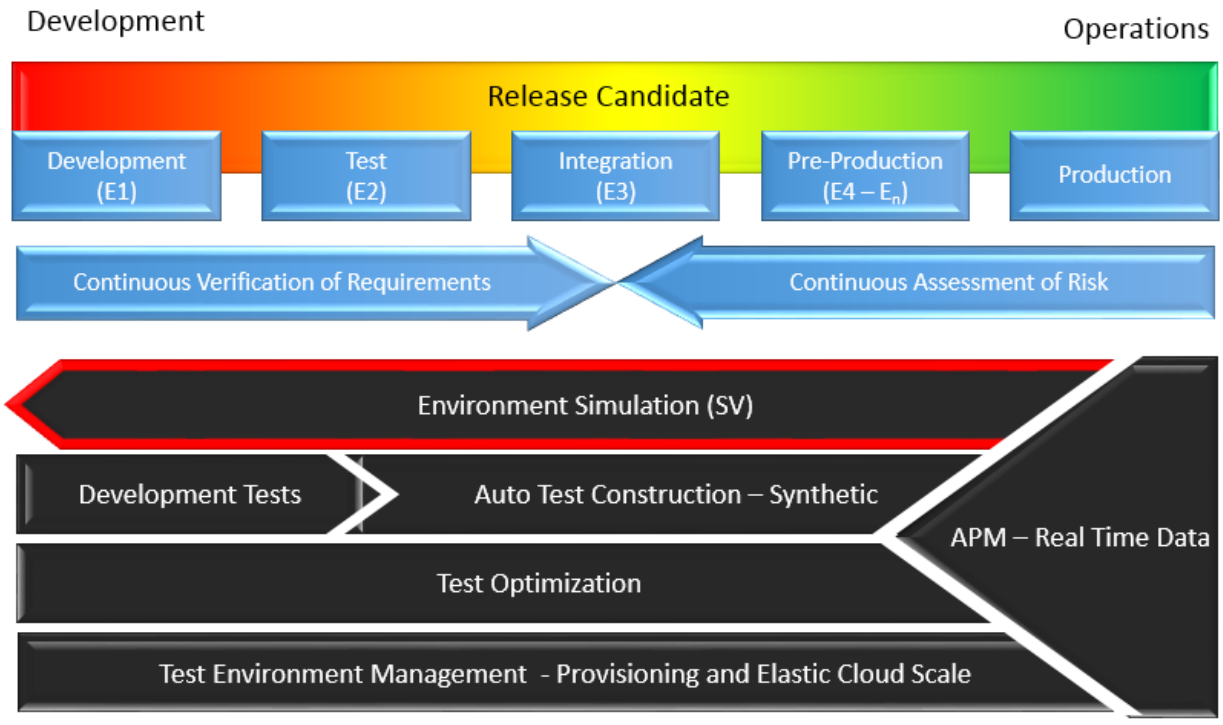

Figure 3: The Next Generation Software Quality 'System'

The benefits of this automated assessment include:

- Throughout the process, business stakeholders have instant access to feedback on whether their expectations are being met, enabling them to make informed decisions.
- At the time of the critical "go/no go" decision, there is an instant, objective assessment of whether your organization's specific expectations are satisfied—reducing the business risk of a fully-automated Continuous Delivery process.
- Defects are eliminated at the point when they are easiest, fastest, and least costly to fix—a prime principle of being "lean."

- Continuous measurement vs. key metrics means continuous feedback, which can be shared and used to refine the process.

References

FDA. "General Principles of Software Validation; Final Guidance for Industry and FDA Staff." http://www.fda.gov/RegulatoryInformation/Guidances/ucm085281.htm (accessed June 8, 2015).

Zhen, Simon. "Take it to the Bank: Mobile Check Deposit is the Next-Gen Standard," MyBankTracker, entry posted March 13, 2012, http://www.mybanktracker.com/news/2012/03/13/bank-mobile-check-deposit-next-gen-standard/ (accessed June 8, 2015).