Continuous Testing

for IT Leaders

PARASOFT®

Continuous Testing

For IT Leaders

Table of Contents

Preface: Evolving from Automated to Continuous Testing for Agile and DevOps

As agile development practices mature and DevOps principles begin to infiltrate our corporate cultures, organizations realize that there is a distinct opportunity to accelerate software delivery. However, when you speed up any process, immature practice areas and infrastructure roadblocks become much more pronounced. It's the difference between driving over a speed bump at 5 MPH versus 50 MPH ... at 50 MPH, that speed bump is going to be quite jarring.

Accelerating any business process will expose systemic constraints that shackle the entire organization to its slowest moving component. In the case of the accelerated SDLC, testing has become the most significant barrier to taking full advantage of more iterative approaches to software development. For organizations to leverage these transformative development strategies, they must shift from test automation to Continuous Testing. Drawing a distinction between test automation and Continuous Testing may seem like an exercise in semantics, but the gap between automating functional tests and executing a Continuous Testing process is substantial.

The most fundamental shift required in moving from automated to continuous is aligning "test" with business risk. Especially with DevOps and Continuous Delivery, releasing with both speed and

confidence requires having immediate feedback on the business risks associated with a software release candidate. Given the rising cost and impact of software failures, you can’t afford to unleash a release that could disrupt the existing user experience or introduce new features that expose the organization to security, reliability, or compliance risks. To prevent this, the organization needs to extend from validating bottom-up requirements to assessing the system requirements associated with overarching business goals.

Introduction

Executive Summary

Leading companies will differentiate themselves with innovative software that bonds customers with the company, its products, and its services. With software as the primary interface to the business, companies must reassess the risk and cost of quality and react strategically.

No matter what industry you're in, software is increasingly becoming the interface to your business. Organizations that are able to increase the speed and quality of innovative software releases will capitalize on differentiable competitive advantages; those that cannot will languish behind competitors. Although the preceding statement can seem worn, the number of companies that become headline news due to software failure is on the rise.

Not surprisingly, many enterprises have begun flirting with the idea of accelerating the SDLC to drive innovation through software. However, it's critical to realize that there's an optimal balance between speed and quality with software delivery—as with all engineered products. We're in an era in which leading organizations must reassess the true "cost of quality" for software. Remember: the cost of quality isn't only the price of creating quality software—but also (and more importantly) it's the penalty or risk incurred by failing to deliver quality software.

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 some time 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. Unfortunately, at the time of the critical "go/no-go" decision for a software release candidate, few organizations release with confidence.

As the release date looms, development teams have become accustomed to asking: "Are we done testing?" Fundamentally, this is the wrong question. It ties the concept of "quality" to static tests that produce multiple, independent, and primarily binary data points of pass or fail. This approach results in a lot of data points, but not the information needed to help the business understand the real impact to the end user experience. Understanding the specific risks associated with each release candidate becomes mission critical as organizations attempt to accelerate the release cycle. Without this visibility and knowledge of the impacts to the business, managers are unable to make the appropriate trade off decisions or timing decisions for releasing software.

Instead of "Are we done testing," we should be asking: "Does the release candidate have an acceptable level of business risk?" This new question is much more complex than it seems at the surface. It carries a few critical assumptions:

1. The inherent business risks associated with a given application and the particular release candidate are well defined.

2. There is an understanding of how to measure each of these defined business risks.

3. A baseline and thresholds are established for defining what constitutes

an acceptable level of risk. Some business risks might have zero tolerance and no thresholds for acceptance.

4. Automation is in place to continuously assess the state of the application versus these defined risks.

This is why the concept of Continuous Testing is so critical. It balances the traditional bottom-up tasks associated with software development and testing with a top-down approach focused on safeguarding the integrity of the user experience while protecting the business from the potential impacts of application shortcomings.

Continuous Testing is Not a Tool

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.

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 now presents the single greatest risk to the organization.

We begin this book by exploring how Continuous Testing accelerates the SDLC, promotes innovation, and helps mitigate business risks. Next, we look at how to bridge the gap between business expectations and development/testing activities. Finally, we explain what's involved in establishing a system of decision that collects essential data across the SDLC and transforms it into actionable risk mitigation tasks.

The Value of Continuous Testing

Executive Summary

Continuous Testing creates a central system of decision that helps you assess the business risk each application presents to your organization. Applied consistently, it guides development teams to meet business expectations and provides managers visibility to make informed trade-off decisions in order to optimize the business value of a release candidate.

You cannot fully appreciate the value of Continuous Testing without understanding the concept of business risk. We've mentioned the term "business risk" a few times so far; let's take a moment to define it before proceeding.

What is "Business Risk"?

In terms of software, a business risk is any application shortcoming that impairs the end user's (or customer's) expected experience and ultimately erodes confidence in the business. A software business risk can manifest itself as a headline news event such as a reservation system outage which strands holiday travelers—damaging brand equity. Or it could be a series of user-experience hiccups that eventually drive customers to a competitor—directly impacting revenue or a subscriber base.

The most infamous business risks associated with software are tied to application security, which is a multi-million dollar annual initiative for IT organizations. The loss of personal or private information due to data theft, data

breaches, or hackers not only erodes brand equity but also brings distinct financial penalties.

Many other risks pose an equally formidable threat to the business, but garner far less attention. For example, risks can fall into categories such as application resiliency, accessibility, availability, reliability, and testability...to name just a few. Due to the extremely varied nature of software development, the top risks will inevitably vary across organizations, applications, and releases. For instance, security could be absolutely critical in the context of a banking application, but be considered trivial in a public web service that reports a weather observation.

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 risks before they progress to the next stage of the SDLC. The goal is to eliminate meaningless activities and produce value-added tasks that drive the development organization towards a successful release.

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

First, Continuous Testing results in clearly-delineated business risks associated with each application in the organization's portfolio—including measurement standards for assessing the level of risk. It guides business and technical teams to collaboratively close the gap between business risk and development activities.

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 the related application components and 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 in order to meet the stated business objective. By providing a holistic understanding of the risk of release, Continuous Testing helps to optimize the business outcome.

Fourth, when teams are continuously executing a broad set of tests via "sensors" placed throughout the SDLC, they collect metrics regarding the quality of the process as well as the state of the software. The resulting metrics can be used to re-examine and optimize the process itself, including the effectiveness of the tests. This information can be used to establish a feedback loop that helps teams incrementally improve the process. Frequent measurement, tight feedback loops, and continuous improvement are all key DevOps principles.

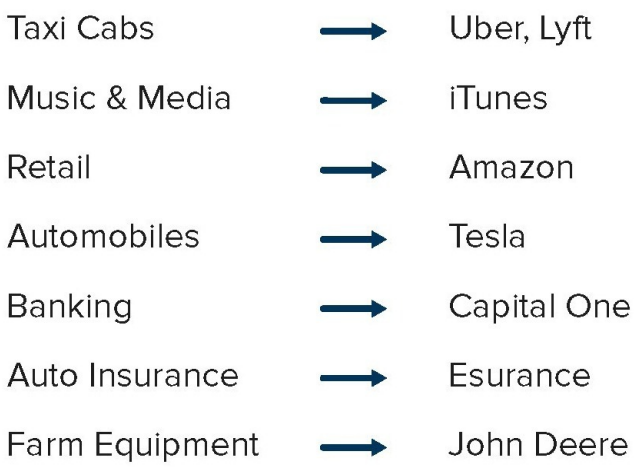
Re-Evaluating the Cost of Quality

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—driving new deposits.[1] Getting a secure, reliable mobile check deposit 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.

With a focus on connectivity and a seamless end-user experience, every business segment is being redefined—and in many cases re-invented—with software:



In some cases, the industry incumbents have created the new go-to market paradigm with forward-thinking research and development. In most cases, we have seen aggressive challengers come into markets and challenge the status quo. For example, consider the meteoric rise in popularity of Uber and Lyft versus a stagnant and heavily-fragmented taxi and livery services.

This sea change focused on the customer experience also comes with much greater expectations of software quality. 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 from 2012 through 2015. In 2014, each incident initiated an average -3.75% decline in stock price, which equates to an average of negative $2.35 billion loss of market capitalization. This is a tremendous loss of shareholder value. Tracking the same type of software failure events in 2015, our findings suggest that the market is punishing news of software failures even more aggressively. In 2015, each incident initiated an average -4.06% decline in stock price, which equates to an average of negative $2.55 billion loss of market capitalization. In a single year, the penalty for software failures increased by over 8%.

Impact on Market Capitalization

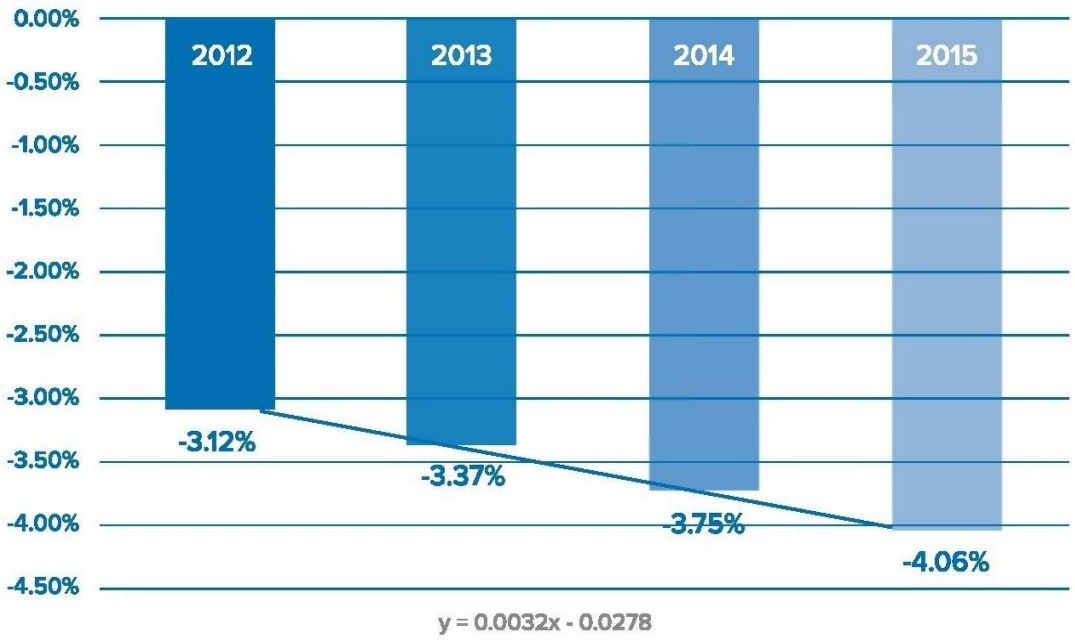


Figure 1 - From 2012 through 2015, software failures that made headline news had an increasing average loss of market capitalization

Additionally, looking at organizations that endured multiple newsworthy software failures in 2015, it is clear that the market punishes repeat offenders even more acutely. 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.[2]

Impact on Market Capitalization
for Software Failure Repeat Offenders - 2015

Figure 2 – In 2015, public companies that had two software failures within 12 months were faced with steeper losses to market capitalization

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.

---

1 http://www.mybanktracker.com/news/2012/03/13/bank-mobile-check-deposit-next-gen-standard/
2 From Parasoft equity analysis of the most notable software failures of 2012-2015.

Establishing Business Expectations

Executive Summary

Today, there is a gap between how the business defines risk and how development addresses these risks in software. Given the growing importance of software, we must ensure that development and testing efforts are focused on mitigating the organization's stated business risks. This is accomplished by expressing objectives in policies that are clearly defined, readily accessible, automatically measured, and managed by exception.

There's no doubt that the daily concerns of the CEO are different than the daily concerns of developers and testers. Yet the software development team that writes and tests code for customer-facing applications could actually have a greater impact on customer satisfaction and loyalty than the day-today activities of the CEO. Unfortunately, due to the detailed and technical nature of developers' and testers' jobs, it's very likely that they become divorced from the overarching business drivers that concern the CEO and executive management.

Closing this gap between business expectations and technical implementation will not only reduce business risk, but also minimize the negative business impacts of faulty software. When development has a firm grasp of business expectations and how to translate them into the technical implementation, business risks are significantly reduced.

There are several critical requirements for bridging the gap between business expectations and technical implementation:

- Business expectations or risks must be clearly communicated to

development as policies. A policy converts management expectations into actionable, measurable tasks. This helps the organization ensure process consistency while agilely adapting to evolving market trends, regulatory environments, and customer demands.

- A real-time infrastructure must give developers feedback on whether they're meeting expectations. For this to work, business expectations or risks must be mapped to non-functional requirements that are automatically measured and monitored. If it's not fully automated and completely unobtrusive (managed by exception with zero impact on productivity), it simply won't be feasible—especially for teams who have adopted agile and/or DevOps.

- Executive sponsors must be clear that satisfying these expectations is non-negotiable. Executives must also be able to automatically monitor compliance and assess the level of business risk for each project or release candidate.

- Training must ensure that developers truly understand what's expected and how it translates to the technical level (i.e., how the application is developed and tested).

The Gap Is Vast.. . and Growing

All too often, when the team is in the throes of developing and testing a release candidate, they hyper-focus on the specific technical aspects of the functional requirement or user story in scope. What gets lost in the shuffle is a holistic perspective of the user experience. For example, a team tasked with implementing a more secure login mechanism might inadvertently degrade application performance across critical transactions. Without a proactive attempt to consistently align their work with clearly-defined business requirements, the team is likely to run fast—but not necessarily in the expected direction. If the development and test teams' hard work does not yield the expected business result, you run the risk of significantly hampering productivity and demoralizing the team.

This acute focus on validating bottom-up requirements or user stories can be further exacerbated by the time-boxes introduced by agile or more iterative development methodologies. In a recent survey, Parasoft discovered an inverse correlation between more iterative development methodologies and the

likelihood for teams to measure compliance to system-level (non-functional) requirements.

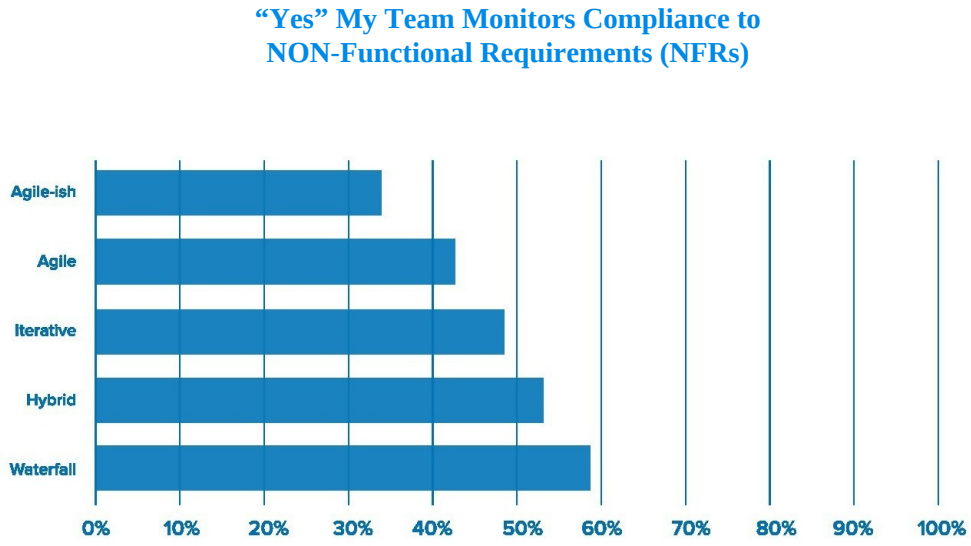


Figure 3–In a recent Parasoft survey, respondents who identified themselves as Agile or agile-ish were less likely to monitor compliance to system-level (non-functional) requirements

Agile and agile-ish teams had a 38% likelihood of monitoring compliance to system-level (non-functional) requirements. Compare this to waterfall teams, which had a 58% likelihood of measuring compliance to system-level (non-functional) requirements. Based on these results, as well as interviews with customers, it seems that the time constraints associated with short development iterations compel teams to focus their resources on validating the user stories in scope.

When getting each story "done done" within constrained agile timelines is already a challenge, it's hard to justify spending time validating whether the modified application satisfies broader system-level expectations. And this problem is certain to escalate as DevOps adoption increases. After all, if this level of checking is not feasible when you're working on two week sprints, how could it possibly work when you start releasing multiple times a day?

What's Needed to Bridge the Gap?

Especially in light of agile and DevOps, if you truly want to assess the business risks associated with a release candidate, it's essential to have an automated, unobtrusive way to continuously assess the overarching business expectations in the context of an evolving application. This mechanism needs to be based on clear and convincing business expectations, driven by strong executive sponsorship, and supported by an effective training infrastructure.

Let's start by looking at what's needed to take a policy and put it into practice.

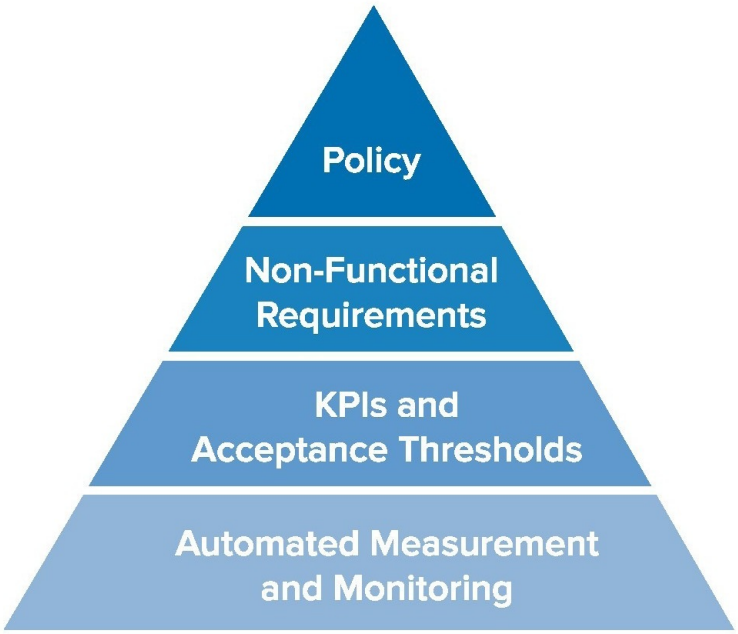


Figure 4 - A policy cannot just be a declaration; it must be dissected into actionable requirements, supported by acceptance thresholds, and automatically monitored.

Policy: The primary element is a definition of the business expectation: what we refer to as a "policy." Policy definition, training, and monitoring is covered in more detail later in this chapter.

Non-Functional Requirements: Each policy is supported by an array of non-functional requirements (NFRs). Whereas functional requirements define what the system should do, non-functional requirements describe how the overall system should behave. Non-functional requirements could include application resiliency, accessibility, availability, reliability, and testability—to name just a

few. Policy has a one-to-many relationship with non-functional requirements. In other words, multiple NFRs might be needed to assess the exposure to a specific business risk that's defined in a policy.

KPIs and Acceptance Thresholds: Once NFRs are established, the business and development team must collaborate to define the key performance indicators. Furthermore, if the organization is exploring exception-based workflows or automated decision nodes, then the teams need to work together to establish acceptance thresholds: criteria for triggering a notification and/ or stopping a release from progressing through the delivery pipeline.

Automated Measurement and Monitoring: A policy and accompanying NFRs without an automated method to measure and monitor can only be considered a guideline.

For example, an organization with a mobile shopping application could have a policy associated with the customer experience as related to network performance. Under certain latency conditions, the business would like to inform its users that performance degradation is caused by network issues rather than the application itself. The organization would construct a policy for network performance and associate the NFRs for application performance and resiliency to the policy. The organization would then establish the expected performance as well as set the threshold that should trigger a warning about network performance. The development team would need access to a test environment that could simulate a broad range of network performance conditions and continuously test the evolving release candidate as part of the Continuous Integration process.

Defining Business Expectations

Given that the ultimate goal of the above pyramid is to automatically assess whether release candidates satisfy business expectations at any given point in time, let's take a closer look at how to best craft and enact policies for meeting those expectations.

The key for any software development manager is to ensure that the team truly understands the business impact of the application they're working on, as well as the potential business risks associated with application failure.

Quantifying risk is an important step in achieving a credible and compelling reason for action. For example, this might involve quantifying the cost of an outage or understanding the impact to brand equity in quantifiable terms. Far too often, the concept of software quality is addressed in a "fluffy" manner of fear, uncertainty, and doubt rather than of known quantifiable impacts. With an understanding of business demands, development teams can then focus their efforts on the aspects of the application that are truly most important to the business.

Demonstrating Executive Sponsorship

The lack of executive sponsorship is the single biggest failure point in relation to quality initiatives. Without an executive manager establishing the importance of the tasks or activities associated with quality, testing practices run the risk of being deemed unfavorable and will rapidly decay. In other words, you end up with elective guidelines rather than policies. "Wash your hands after using the restroom" and "Look both ways before crossing the street" are both guidelines: they're great suggestions, but unless they're mandated and monitored, compliance will be highly variable. The lack of a clear policy is also exacerbated by highly-distributed development teams or teams that utilize third-party contributors. It's very easy for directions that were intended as requirements to be interpreted as guidelines. For example, "You should do peer code review" is typically understood to mean "Do peer code review if you feel you have time" while the intent is quite the opposite.

Communicate the Impact of Failure

After the true business impact is assessed and quantified, the executive sponsor needs to communicate this with developers and testers. It's important to focus on the actual impact of failure rather than the theoretical impact of failure. Tangible stories are key for achieving this purpose. Also, it's invaluable to have the executive personally communicating this to the team, placing a name and face to the risks and concerns. It is one thing to read about the potential for risk in a training manual; it's another to have executive management stop by to highlight its importance. This truly humanizes the impact of business failure.

The impact of failure should also be detailed within the description of a policy. For example:

- “A data breach is estimated to cost our organization about $250 per record—not including the impact to brand and shareholder value.”
- “A news event associated with faulty software in our industry has an average of a negative $2.5 billion decline in shareholder value.”
- “A production outage equates to a $66,000 per minute loss of revenue.”
- “The cost to recall and fix an embedded software component runs the company $1,750 per vehicle.”
- “The physical cost of a medical device recall would run over $18.5 million—not including the damages associated with inevitable lawsuits.”

Provide Visibility into Process Adherence

The executive sponsor and direct reports must have sufficient insight into policy compliance to be able to identify emerging issues and know when it’s appropriate to step in and ask questions. To provide this level of visibility, a central platform must aggregate data and deliver warnings when desirable thresholds are exceeded. Ultimately, what the executive sponsor needs is enough information to make optimal process and resources decisions. For example:

- Why won’t we meet the expectation associated with the policy?
- Is there something wrong in terms of resources?
- Is there something wrong in terms of tooling?
- Did we underestimate the level of effort?
- Are we missing critical process steps?
- Are certain process steps not delivering the expected outcome?

Training on Business Expectations

Ensuring that developers understand business demands and feel compelled to satisfy them is one thing; preparing them to actually meet those expectations is

another. That’s why training is a critical component.

The first aspect of effective training is to provide a formal venue for outlining policies and training on how to meet the expectations encapsulated in those policies. The new habits will eventually become second nature to your long-term team members as time goes on. This is great; however, when senior team members are mentoring new ones, core concepts will inevitably be overlooked. To ensure that new hires receive the same level of training that was provided when the policy was first introduced, training needs to be a formalized, continuous process.

Second, training must also involve a repository that centralizes access to all relevant artifacts as well as provides objective, real-time feedback on whether activities are meeting expectations. This ties back to the policy pyramid presented at the beginning of this chapter. The least disruptive way to help team members ensure that they’re on the right track is with an exception-based notification system. If developers perform the expected actions (as defined by the policy), then the system remains passive and does not engage them. Notifications are generated only when deliverables don’t align with policy definitions. The result is that experienced team members who understand and execute the company’s policies have the freedom to write code and test without interruption, while those who are new to the team can be gently nudged in the right direction.

A Platform to Assess Business Risks

Executive Summary

Continuous Testing requires an infrastructure to apply policies consistently across individuals, teams, projects and divisions. A Development Testing Platform translates policies into prioritized tasks in order to mitigate the defined business risks. It also provides managers insight and control over the process of creating quality applications.

SDLC acceleration requires that distinct quality objectives are automatically validated at each stage of the SDLC. With Continuous Testing, the team needs to always be aware of the state of the application versus the actual business objectives that either define “quality” or mitigate risks. This is key for ensuring that the team meets the expected objectives before progressing to the next stage —significantly reducing the need for manual intervention or late-stage functional or non-functional requirement validation.

If speed is the primary definition for team success, quality will inevitably suffer unless you have established expectations that are automatically monitored for compliance. Making quality expectations non-negotiable sets the boundaries for acceleration while reducing the risks associated with project, application, or business failure.

A Development Testing Platform is a Central “System of Decision”

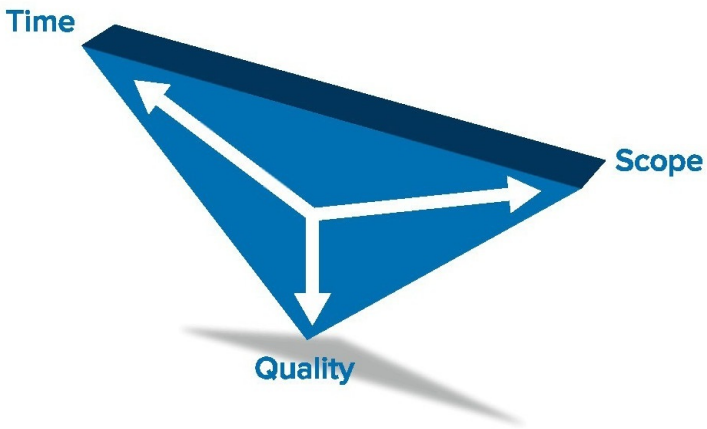
Being able to automatically assess whether a release candidate meets the organization’s specific definition of software quality requires a method to

federate quality information from multiple infrastructure sources (source code management, build management, defect management, testing, automated analysis, etc.). A Development Testing Platform is this central "system of decision"; it transforms policies into prioritized tasks as well as delivers insight and control over the process of creating quality applications.

A Development Testing Platform assists the organization to work smarter—limiting the nature and degree in which business-critical quality tasks can be discounted. A Development Testing Platform also assists business managers to balance the three project variables which always seems to be at odds with one another: time, scope, and quality.

More on the Time-Scope-Quality "Trade-off"

It's important to note that the current trends in SDLC optimization (e.g., DevOps, lean, bi-modal, agile) all advocate the optimization of time, scope, and quality—not the traditional trade-off among them.



What's the reason for this shift? It's the manifestation of software as the core method to reach and retain customers. It means that software quality is no longer optional—and the definition of software quality in context of specific business applications will need to become as tightly-defined as accounting principles or human resource policies.

A Development Testing Platform is like a navigation/mapping application that discovers an optimal route predicated on multiple, automated inputs. You have a much greater chance of reaching your destination faster if you're using today's mapping applications like Waze, Google Maps, and Apple Maps than you would by relying on news radio's traffic reports. Given that navigating a release cycle is more like driving in a crowded city center like Los Angeles than cruising through the country side, advanced navigation is critical if you want to avoid bottlenecks.

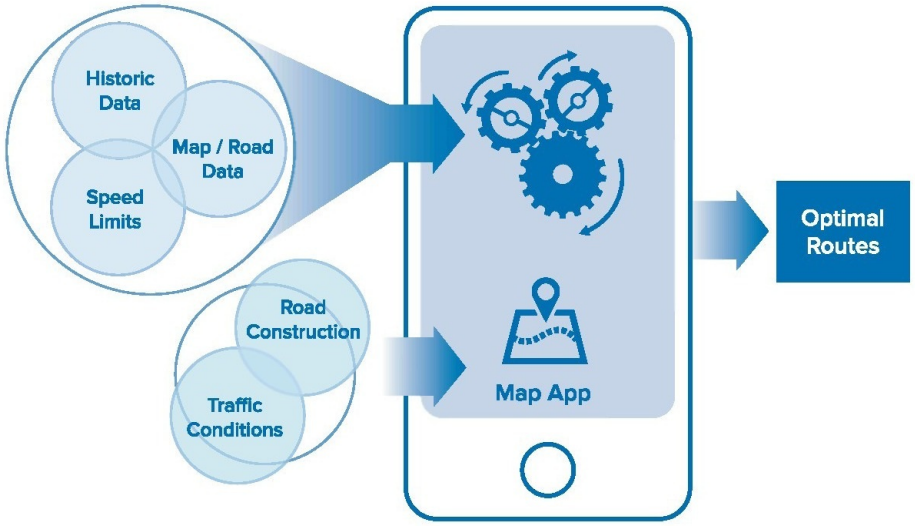


Figure 5 – A Development Testing Platform is like a mapping application; multiple inputs from various data sources are leveraged to present an optimal route to the given destination

Development Testing Platform: Managing SDLC "Sensors"

As they begin evolving to Continuous Testing, every development team will inevitably have a disparate collection of tools providing data on a wide array of measures ranging from defect trends, to performance monitoring metrics, to code optimization opportunities, to unit test suite effectiveness. This is a great foundation: these tools all collect key observations about the current state of the software. However, most organizations tend to adopt and deploy tools in an ad-hoc manner, which compromises the consistency and accuracy of the findings.

Moreover, the configuration and execution of these tools is typically divorced from business expectations—so their results do not provide the needed insight on whether a release candidate is meeting business expectations.

The goal of a Development Testing Platform is to take all of these tools and place them in the context of a larger system that:

- Drives consistent deployment and adoption—ensuring consistency and accuracy
- Aligns execution with business expectations—ensuring business-relevant results
- Performs advanced multivariate analysis across different tools, test runs, and over time—identifying application hotspots that harbor hidden defects

The tools serve as “sensors” placed throughout the SDLC. The more sensors the better, the more data the better—as long as we have an automated method to collect raw observations and process the raw observations into valuable findings. We will go into more detail about managing this data later.

With tools collecting data via automated analysis or from the output of an artifact like a test, we can aggregate raw observations. The aggregation of raw observations would make little sense to a human observer, given both the scope and volume of data. At this point, we need a mechanism to cull through the raw observations and give SDLC practitioners valuable and actionable findings that will help them prevent software defects and meet corporate compliance objectives.

This where policy truly shines. In conjunction with a post-analysis engine (process intelligence engine), a policy allows the system of decision to filter the noise from raw observations and highlight valuable and prioritized findings.

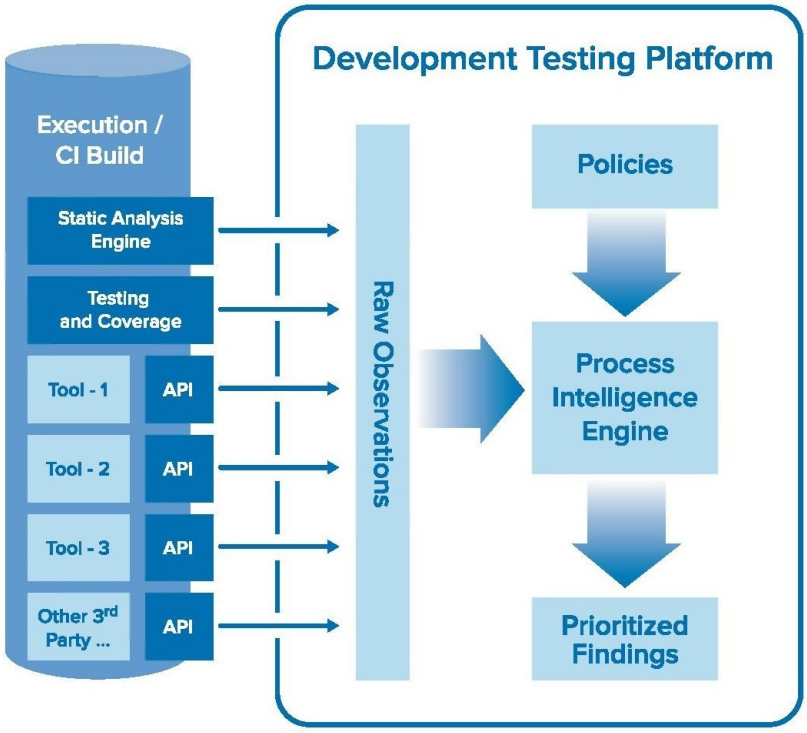


Figure 6 – SDLC domain based APIs collect data as raw observations; policies and a Process Intelligence Engine convert raw observations into actionable, prioritized findings

Development Testing Platform: Key Capabilities

The following sections outline, from a more technical perspective, what characteristics a Development Testing Platform needs in order to drive this process of converting sensor observations into prioritized, actionable findings.

Openness and Ease of Integration

Leveraging a Development Testing Platform requires openness: the platform should furnish well-defined APIs that allow information and data from associated software development infrastructure systems to be consumed and published with ease. The ease of integrating data from disparate systems will be the key to truly establishing a system of decision.

Furthermore, the APIs that drive the integration should enable various infrastructure domains, making canonical data elements associated with various

SDLC systems, quality practices, or artifact types readily available. This eases integration and allows for much more flexible downstream data transformation, analysis, and processing. Software development domain-specific APIs are a core differentiator between a Development Testing Platform and a general Business Intelligence (Bl) tool. Native integrations with well-known software infrastructure systems (defect, source code, build, static analysis, unit testing, code review, IDEs...) speed system configuration and management.

One of the primary considerations for adopting a commercial Development Testing Platform versus building one yourself should be access to an ecosystem of add-ons or value-added plugins to the platform. A marketplace for plugins significantly reduces the time and effort required to either customize data filtering or integrate with niche tools.

Driven by Policy

As we have alluded to in previous sections, a policy is a business expectation translated for the development and testing staff. A Development Testing Platform is the central repository for putting those policies into action consistently across the organization. A Development Testing Platform correlates policy with automated analysis techniques and testing practices that assess the level of policy adherence on a continuous timeline. The platform also generates notifications and tasks by exception, guiding the team to achieve the stated policy objectives.

Although a policy is established to mitigate business risks, a policy can cause tension with developers and testers if the intent of the policy is not clearly understood. All policies need to be continuously reviewed and improved upon, but policies that cause tension need to be seriously re-evaluated. There are four root causes of policy failure:

1. The policy is not properly aligned to a business objective.

2. There is a lack of understanding or training about the policy’s true business intention.

3. The policy is superfluous and causes unnecessary re-work.

4. There is insufficient automation to achieve or measure the implementation of the policy.

In broader terms, you can think of a policy as a container for one or more non-functional requirements. For example, you could have the “Company X- Secure Coding Policy.” This policy would define the minimum criteria for how code should be constructed to prevent and/or eliminate potential application vulnerabilities. The policy must be enforced automatically; in this case, it could be enforced via static code analysis (to prevent vulnerabilities) and application penetration testing (to root out any vulnerabilities that slipped through your prevention efforts and reached the built application).

A policy must have (at least) three components:

1. **Human Readable**. It must be human-accessible, readable, and understandable. A business expectation should be associated with each policy. A sample policy could read, “Company X is a 125 year old financial institution that bases its success on earning the trust of our clients. Part of that trust includes information security and privacy. A single security breach could erode the trust that we have built with our clients. Additionally, a breached security vulnerability has a physical cost of $250 per record as well as severe negative impacts to stock price and brand equity. This is why our secure coding guidelines have been formally defined and supported by our CEO...”

2. **Automatically Enforced**. It must be enforceable via an automated, exception-based notification system. Managing the policy itself should not impede productivity. A process that forces developers and testers to manually report on policy adherence is not sustainable.

3. **Measureable**. It must be measurable and visible to management. Furthermore, the volumes of detailed data generated from development and test teams needs to be filtered and translated so that the business impacts of the data is readily understandable by both senior technical managers and business managers. Through a simple, intuitive reporting interface, managers must be able to rapidly assess policy compliance—and, more importantly, determine what actions to take to address non-compliance.

Execution

To ensure speed and accuracy when executing specific analyses or tests, it’s

imperative to have a platform that offers flexibility—from execution options natively available within the platform, to an API that's specifically designed for executing test artifacts over distributed resources. First, the API must be callable by popular build management, continuous integration (Cl) and DevOps tools. Second, the API must provide appropriate operations that orchestrate the execution of test artifacts at the desired stage of the SDLC. This flexibility is key for ensuring speed as well as achieving actionable outcomes that can ultimately mitigate business risks prior to release.

Considering the myriad execution scenarios that could transpire, having the flexibility to run the right tests at the right time becomes the critical path. The flexibility to execute specific sets of tests also requires access to a complete test environment. This is where simulated test environments (via Service Virtualization) become an indispensable component of your development and test infrastructure.

Process Intelligence

Process control throughout the SDLC requires the ability to observe and synthesize data across systems, analysis techniques, and testing practices. "Siloed" or one-off reports generated by single analysis types provide only a small fraction of the process story. Ultimately, the aggregation and intelligent interpretation of the data generated from various sub-systems should deliver suggestions for optimizing the process and mitigating the risks prioritized by the business.

As the central system of decision for SDLC quality, the Development Testing Platform must readily manage multiple data inputs from various infrastructure sources. The collection of raw observations across systems is the first step in transforming compartmentalized data points into process intelligence. The second step is the ability to process raw observations through correlation, advanced analysis, and the application of patterns. At the final step, observations are filtered based on the organization's policies to more accurately pinpoint findings that represent the highest risks associated with the specific stage of the SDLC.

The ability to perform domain-specific advanced analysis is a core differentiator between a Development Testing Platform and a general Business Intelligence (Bl) tool. Being able to rapidly apply specific analysis tools (e.g., pattern

recognition tools, multivariate analysis, inference engines, correlation analysis, etc.) allows for systemic risks to be rapidly identified and continuous improvement opportunities to be highlighted.

The effective application of advanced data analysis will enable the organization to systematically prevent defects. Process intelligence also assists the organization to detect inefficiencies or waste in the SDLC that can hamper acceleration and advanced automation.

Prioritized Findings

The primary challenges associated with adopting development testing tools are managing expected outcomes and presenting information in a way that's valuable (and actionable) to managers and practitioners.

Throughout the SDLC, there are numerous opportunities to collect raw observations; however, there is usually very limited time to investigate, research, and remediate potential defects. A Development Testing Platform must automatically deliver prioritized findings that directly correlate to the reduction of risk. Additionally, it must be flexible enough to deliver the findings as actionable remediation tasks at the optimal stage of the project. Systems that triage results through a human reviewer cannot scale sufficiently for Continuous Testing to be effective.

Advanced analysis and testing is critical for success—yet, without a centralized process to systematically generate prioritized tasks in order to fix the discovered defects, quality practices will typically disintegrate and then resurface when the organization faces a painful or highly-publicized failure. A Development Testing Platform must make defect remediation achievable by prioritizing the actionable findings and automatically distributing tasks to the correct resource. The tasks should be accessible not only within the Development Testing Platform, but also via an open API that provides access to tasks within workflows of other, complementary process tools.

In addition to driving a central process for defect remediation, the Development Testing Platform should also offer actionable information to managers. Rows of data do not deliver readily accessible analysis about risk. Data must be converted to manager-friendly dashboards that help the team make optimal trade-off decisions.

Teams looking to apply advanced automation throughout the SDLC must consider that there is a vast difference between data presented in a dashboard and actionable findings that are correlated to a release candidate. If go/no-go decisions are predicated on predetermined policies and thresholds, a dashboard that aggregates data is far too passive of a technology for extreme automation. A dashboard requires human interpretation as well as all the inefficient human-to-human negotiation that comes with compelling an individual or team to take action.

A system of decision is much different than a dashboard. Generally speaking, once a finding makes its way onto a dashboard or report, it becomes one of many things to do—overwhelming management, developers, and testers. Prioritized findings, driven by policies, must become part of a go/no-go punch list with full transparency across the team.

The “Continuous” in Testing: What’s Involved?

Executive Summary

Continuous Testing does not mean do more of the same “bottom-up” tasks with greater automation. To achieve a real-time, objective assessment of the business risks associated with a release candidate, organizations must consider the efficacy of test artifacts and analysis techniques that ultimately drive the assessment of quality or risk.

Consider this: if software quality efforts have traditionally been a “time-boxed” exercise, then we can’t possibly expect that accelerating the SDLC will yield better results from a testing perspective. If organizations want to accelerate software releases, they must reassess the current testing practices in order to keep quality as status quo. However, in order to improve software quality in conjunction with SDLC acceleration, organizations will have to truly consider re-engineering the process of creating quality software.

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

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.

Figure 7 – Continuous Testing is synonymous with continuous improvement; it requires constant re-evaluation of risk and the infrastructure in place to mitigate prioritized risks

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.

Technical Debt

The concept of technicaI debt has gained popularity over the past few years. Its measurement has become core to the assessment of the SDLC and it can be an effective practitioner-level metric.

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 practitioner’s dashboard.

Risk Mitigation Tasks

All quality tasks requested of development should be 100% correlated to a policy or an opportunity to minimize risk. A developer has two primary jobs: implement business requirements (or user stories) 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.

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, component, service, application, 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. Coverage analysis delivers great data that should be used in conjunction with other SDLC "sensors." For example, coverage data in conjunction with rich application component metadata that profiles risk can establish parameters for exploratory testing or expanded simulation conditions. Coverage analysis in conjunction with cyclomatic complexity can highlight an application hotspot that must be investigated.

Test Quality Assessment

Processes and test suites have one thing in common: overtime, 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 out of context of the time or user story that drove them.

Continuous Testing requires reliable, trustworthy tests. When test suite results

become questionable, there is a rapid decline in how and when team members react to test failures. This leads to the test suite becoming out-of-sync with the code—and application quality ultimately out of control.

With this in mind, it is just as important to assess the quality of the 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 or sensors are not reliable, then we must consider the process to be out of control.

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 that reduce business risk.

The primary business objectives of policy analysis are:

- Expose trends associated with 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.

Requirements Traceability—Determine if you are “Done-Done”

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.

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 Develop-ment 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.

Advanced Analysis—Expose Application Risks Early

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.

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 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."[3]

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.

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 not only help teams identify these higher-priority tests, but also assist them in selecting the appropriate set of tests for various stages of the release timeline.

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 list for test creation or maintenance 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[4] 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.

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.

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.

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.

Construction and Testability

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.

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 chapter) 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.

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

Test Environment Access and Simulation (Service Virualization)

With the convergent trends of parallel and iterative development, increasing system complexity/interdependency, and DevOps, it has become extremely rare for a team to have ubiquitous access to all of the dependent applications required to execute a complete test. The ability to accurately assess the risk of a release candidate for today's composite applications is becoming a tall order.

You have highly-distributed development and test teams that need simultaneous on-demand access to a release candidate—as well as its myriad APIs and dependencies that must be present in the test environment—in order to continuously test throughout the software lifecycle. Using a conventional on-premise infrastructure to build out complete test environments that closely resemble production is typically slow, technically challenging, extraordinarily expensive, and infeasible due to dependencies that can't be reproduced in the test environment.

To eliminate these constraints, teams must leverage innovative system cloning and simulation technologies to rapidly configure, provision, scale, and reproduce complete dev/test environments. The application stacks that are under your control (cloud-ready) can be imported and imaged via an elastic Environment-as-a-Service (EaaS) in a cloud. Service Virtualization then allows you to simulate the behavior of those dependencies you cannot easily image (e.g., third-party services, SAP regions, mainframes, not-yet-implemented APIs, etc.), or those you want to stabilize for test coverage purposes. EaaS environments are becoming more ubiquitous within DevTest organizations, yet most organizations are just now discovering Service Virtualization.

By leveraging Service Virtualization or simulation 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 in order to bring higher quality functionality to the market faster and at a lower cost.

---

3 http://www.fda.gov/medicaldevices/deviceregulationandguidance/guidancedocuments/ucm085281.htm#_To

4 PCI DSS is the Payment Card Industry Data Security Standard

Conclusion: From Testing to QA; From Automated to Continuous

Executive Summary

We're at a strategic inflection point when it comes to defining software quality and building a process to achieve it. To release engaging software faster, we need to evolve from a world where QA is focused on constructing and executing bottom-up tests to a paradigm where the entire organization plays a role in defining and mitigating business risks through an end-to-end quality process.

Compared to the rigor of the quality process for discrete or manufactured products, software has a ways to go. A past generation of end users have become accustomed to restarts, shut downs, and Task Manager "end tasks." However, this acquiescent attitude towards faulty software has run its course—millennials have significantly different expectations for software quality, and future generations will likely be even less tolerant of disruptions to the user experience. With the ease of integrating software at an all-time high and the cost of switching applications at an all-time low, it's easier now than ever to replace applications or move to another subscription service.

The Impetus for Change

As we referenced in Figure 1, the penalty for exposing faulty software is at an all-time high. Public companies with a "software glitch" that made headline news experienced a -4.08% drop in their stock price. This should be enough of

an incentive to place organizations on notice that software quality matters.

Although it's not quality related, consider the impact of the Volkswagen emissions cheating scandal in relation to the evolving SDLC. The Volkswagen scandal was a pure act of questionable ethics—carried out via software. As of the writing of this book, the total financial impact of the scandal is yet to be determined, but the impact to software development shops will be indelible. The event has pushed software into the spotlight of compliance—irrevocably highlighting software as a substantial business risk.

From Testing to QA

We cannot expect that the software testing practices of the past will suffice for the modern software development methodologies and SDLC processes that are being evolved today.

Across the array of roles and responsibilities for the post-agile (or more iterative) development methodologies, the job of QA has experienced the most profound change. At the same time that the defined window for the task of testing disappeared, the primary method for executing tests became obsolete.

Even though the term "QA" is derived from "quality assurance," the QA role on software development teams has been more or less focused on tactical testing. For the more modern collaborative process initiatives (DevOps, lean, agile...) to take hold, the role of QA must shift back to quality assurance. In this case, QA is responsible for defining and enabling a continuous, proactive process that identifies and prevents business risks throughout the software lifecycle.

If you accept the above definition, then the idea of QA being focused on creating and managing functional test scripts will seem strange; this task is neither preventative nor process oriented. This leads us to one of our primary conclusions for Continuous Testing: organizations must make a concerted effort to separate the activity of testing from the concept of quality. The concept of quality and how it is defined is an organizational and business responsibility that should be reflected in the company's culture. Testing is just one of many activities that ensure the organizational quality targets are being achieved.

From Automated to Continuous

There is a vast schism between automated testing and Continuous Testing—and this schism will be bridged over time as the process of delivering software matures. Both internal and external influences will drive the evolution of Continuous Testing. Internally, agile, DevOps, and lean process initiatives will be the main drivers that generate the demand for change. Externally, the expense and overhead of auditing government and industry-based compliance programs will be the primary impetus for change. Any true change initiative requires the alignment of people, process, and technology—with technology being an enabler and not the silver bullet. Yet there are some basic technology themes we must explore as we migrate to a true quality assurance process. In general, we must shift from a sole focus on test automation to automating the process of measuring risk. To begin this journey, we must consider the following:

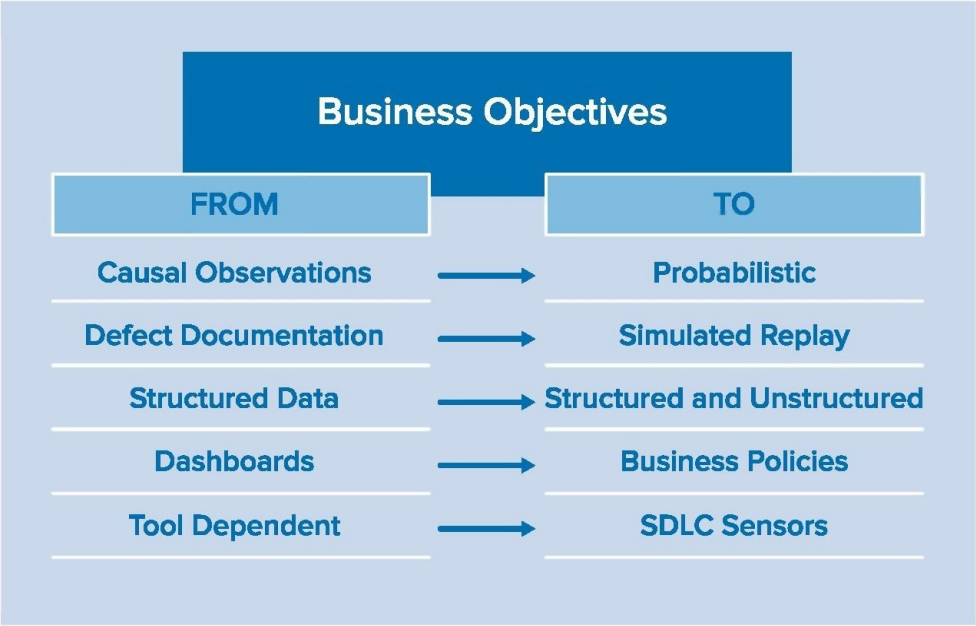


Figure 8 – Driven by business objectives, organizations must shift to more automated methods of quality assurance and away from the tactical task of testing software from the bottom up

From Causal Observations to Probabilistic Risk Assessment

With QA traditionally executing manual or automated tests, the feedback from the testing effort is focused on the event of a test passing or failing—this is not enough. Tests are causal, meaning that tests are constructed to validate a very

specific scope of functionality and are evaluated as isolated data points. Although these stand-alone data points are critical, we must also use them as inputs to an expanded equation for statistically identifying application hotspots.

The SDLC produces a significant amount of data that is rather simple to correlate. Monitoring process patterns can produce very actionable results. For example, a code review should be triggered if an application component experiences all of the following issues in a given Cl build:

- Regression failures greater than the average
- Static analysis defect density greater than the average
- Cyclomatic complexity greater than a prescribed threshold

From Defect Documentation to Simulated Replay

The ping-pong between testers and developers over the reproducibility of a reported defect has become legendary. It's harder to return a defect to development than it is to send back an entree from a world-renowned chef. Given the aggressive goal to accelerate software release cycles, most organizations will save a significant amount of time by just eliminating this back and forth.

By leveraging Service Virtualization for simulating a test environment and/or virtual machine record and playback technologies for observing how a program executed, testers should be able to ship development a very specific test and environment instance in a simple containerized package. This package should isolate a defect by encapsulating it with a test, as well as give developers the framework required to verify the fix.

From Structured Data to Structured and Unstructured

The current tools and infrastructure systems used to manage the SDLC have made significant improvements in the generation and integration of structured data (e.g., how Cl engines import and present test results). This data is valuable and must be leveraged much more effectively (as we stated above in the "From Causal Observations to Probabilistic" section.

The wealth of unstructured quality data scattered across both internal and

publicly-accessible applications often holds the secrets that make the difference between happy end users and unhappy prospects using a competitor's product. For example, developers of a mobile application would want constant feedback on trends from end user comments on:

- iTunes app store
- Android app store
- Stackoverflow
- Twitter
- Facebook
- The company's release announcements
- Competitors' release announcements

This data is considered unstructured since the critical findings are not presented in a canonical format: parsing and secondary analysis are required to extract the valuable information. Although these inputs might be monitored by product marketers or managers, providing these data points directly to development and testing teams—in terms that practitioners can take action on—is imperative.

From Dashboards to Business Policies

In a Continuous Everything world, quality gates will enable a release candidate to be promoted through the delivery pipeline. Anything that requires human validation clogs the pipeline. Dashboards require human interpretation—delaying the process.

Dashboards are very convenient for aggregating data, providing historical perspectives on repetitive data, and visualizing information. However, they are too cumbersome for real-time decision making because they do not offer actionable intelligence.

Business policies help organizations evolve from dashboards to automated decision making. By defining and automatically monitoring policies that determine whether the release candidate is satisfying business expectations,

quality gates will stop high-risk candidates from reaching the end user. This is key for mitigating the risks inherent in rapid and fully-automated delivery processes such as Continuous Delivery.

From Tool Dependent to SDLC Sensors

Let's face it—it's cheap to run tools. And with the availability of process intelligence engines, the more data observations we can collect across the SDLC, the more opportunities will emerge to discover defect prevention patterns.

Given the benefit of a large and diverse tool set, we need to shift focus from depending on a single "suite" of tools from a specific vendor (with a specific set of strengths and weaknesses) to having a broad array of SDLC sensors scattered across the software development lifecycle. And to optimize both the accuracy and value of these sensors, it's critical to stop allowing tools to be applied in the ad hoc manner that is still extremely common today. Rather, we need to ensure that they are applied consistently and that their observations are funneled into a process intelligence engine, where they can be correlated with other observations across tools, across test runs, and over time. This will not only increase the likelihood of identifying application hotspots, but will also decrease the risk of false negatives.

Final Thoughts on Continuous Testing (For Now)

Over the past decades, business initiatives that focused on software quality for the sake of improving software quality have yielded underwhelming results. There are many reasons why these types of initiatives failed:

- The business felt powerless negotiating with "techies"
- There was a perception that a software failure did not have extenuating business impacts
- The development team had greater organizational power over QA, enabling development to resist shift-left
- The QA organization was historically (mis)aligned with development, rather than with the business

- The initiatives lacked executive management sponsorship

In other words, software quality initiatives isolated to the development and testing teams lacked a compelling business driver to promote organizational change. However, now we are in a new era of software. The business is expecting more—and existing software processes are not meeting demands for quality and speed. The time is ripe for true process change.

So why invest in the shift to Continuous Testing today? We are in the midst of a rapid paradigm shift from the age of the vendor to the age of the customer. If your business leverages software to attract, enable, or retain customers, then you are witnessing some unprecedented market shifts:

- **Prospects are judging you before you interact with them:** Today's prospec-tive customers enjoy abundant access to information, with blogs, reviews, reports, and customer reviews universally accessible from an array of devices. Before you actually get in touch with a prospect, they have probably already researched you (and your competitors) and developed a bias based on their unguided exploration and your online reputation.

- **Lower switching costs:** Switching costs for software are at an all-time low and dropping. Gone are the days of multi-million dollar system integration projects. With APIs being ubiquitous and easy to use, the cost of switching software applications or devices is at an all-time low. Think of a banking application or a mobile phone—in the past five years, it has become significantly easier to transfer data. This form of vendor lock-in has been eroding over the past decade and it will continue to erode as more devices, applications, and back-end systems are able to interconnect. The ubiquity of data access will be challenged or halted only by security and data privacy issues.

- **Increased demand for compliance:** For most large industries that hold data which could be considered private or confidential, the need to comply with government or industry standards will inevitably increase. Inability to demonstrate compliance or inability to comply with standards in a cost-effective manner will erode customer and prospect confidence.

- **Increased cost of quality:** The cost of software quality is on the rise

> and will continue to escalate as industry-leading companies rely on software for more and more core interactions. The cost of quality is the penalty or risk incurred by failing to deliver quality software—and this is relative. In other words, if an entire industry is delivering an average user experience via software, then the cost of software quality is not as high in that market. However, if one organization manages to deliver an exceptional user experience, this could be a valuable competitive differentiator.

The undeniable truth is that meeting your customers' expectations with quality software drives brand loyalty. Without a software quality process that is well defined and continuously improved, an organization will become laggards among its competitors and the brand will slip from the market.

In the chapter The Value of Continuous Testing, we presented a few examples of companies that transformed markets via software. Market transformation does not stop with those examples—every industry is under fire to deliver an exceptional user experience via software or face extinction.

Today, every organization is entrusting their software development team to deliver an exceptional end user experience—just like an airplane pilot is entrusted with transporting passengers to their destination. Trying to ensure a positive user experience without a constant awareness of the business risks inherent in each release candidate is like trying to land a jet at a busy airport without air traffic control. If you have immediate and continuous feedback on the nature and severity of the risks you're facing, you'll have a much better likelihood of landing safely.

About Parasoft

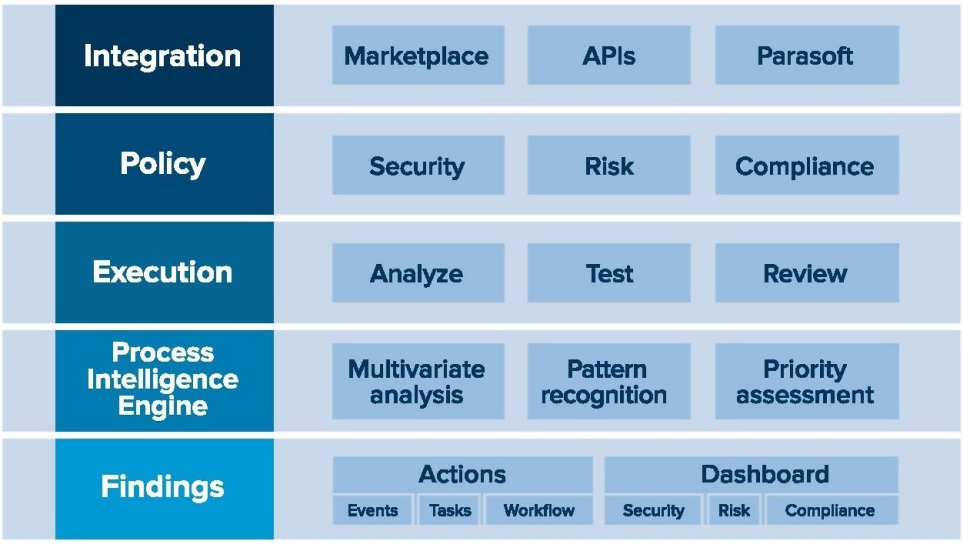
Overview

Parasoft develops automated software quality solutions that prevent and detect risks associated with application failure. To help organizations produce top-quality software consistently and efficiently as they pursue agile, lean, DevOps, compliance, and safety-critical development initiatives, Parasoft offers a Development Testing Platform and Continuous Testing Platform.

Development Testing Platform

Parasoft Development Testing Platform (DTP) enables Continuous Testing. Leveraging policies, DTP consistently applies software quality practices across teams and throughout the SDLC. It enables your quality efforts to shift left-delivering a platform for automated defect prevention and the uniform measurement of risk.

Parasoft DTP helps organizations:



- Leverage policies to align business expectations with development activities
- Prevent software defects and eliminate rework-reducing technical debt
- Focus development efforts on quality tasks that have the most impact
- Comply with internal, industry, or government standards
- Integrate security best practices into application development
- Leverage multivariate analysis to discover application hotspots that harbor hidden defects

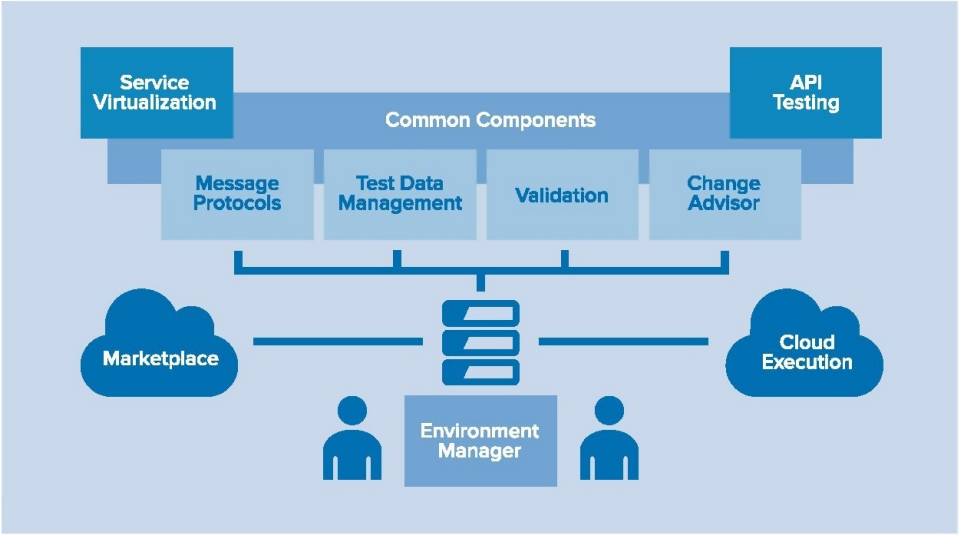
Continuous Testing Platform

Today's DevOps and "Continuous Everything" initiatives require the ability to assess the risks associated with a release candidate—instantly and continuously. Parasoft Continuous Testing helps organizations rapidly and precisely validate that their applications satisfy business expectations around functionality, reliability, performance, and security.

Parasoft Continuous Testing Platform features the following core capabilities:

- **Service Virtualization:** Provides on-demand access to complete, realistic test environments by simulating constrained dependencies (APIs, services, databases, mainframes, ERPs, etc.)

- **API Testing:** API/service unit testing, end-to-end functional testing, load/performance testing, and security testing



- **Test Environment Management:** On-demand provisioning of complete test environments in order to rapidly evaluate a release candidate; allows your automated tests to run continuously versus complete test environments

- **Test Data Management:** Centralized creation and management of secure test data that can be applied across all solutions and integrated tools (including open source tools), as well as across team roles and test types (unit, integration, performance, security...)

About the Authors

Wayne Ariola, Chief Strategy Officer, leads the development and execution of Parasoft's 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 numerous patents for his inventions. A recognized leader on topics such as the SDLC, Service Virtualization, SOA and APIs, quality policy governance, and business strategy, Ariola is a frequent contributor to business publications—as well as a sought-after speaker at key industry events. Ariola brings more than 20 years strategic consulting experience within the technology and software development industries. Prior to joining Parasoft, Ariola was the Senior Director of Strategy at Fasturn, Inc., a company he helped to start. Previously, Ariola was an Associate Director for PricewaterhouseCoopers, where he was recognized as a leader in the Strategic Change practice. Ariola joined Parasoft in March of 2003. He has a BA from UCSB and an MBA from Indiana University.

Cynthia Dunlop, Lead Technical Writer, authors technical and marketing communications for Parasoft—currently specializing in Service Virtualization, API testing, DevOps, and Continuous Testing. She has over 15 years of experience writing for the software development industry, including numerous articles for industry publications and books for Wiley and Wiley-IEEE. Dunlop holds a BA from UCLA and an MA from Washington State University.