Continuous Testing for DevOps: Evolving Beyond Automation

The demand for faster application delivery is driving a surge of interest in DevOps (including lean, Agile, etc.). Organizations successfully leveraging DevOps are indeed delivering innovative new software to the market faster and more frequently. However, you can't achieve these results by simply automating and speeding up your existing processes (remember the famous "I Love Lucy" chocolate factory scene where the conveyer belt starts running faster and faster and Lucy and Ethel struggle to keep pace?).

DevOps represents a cultural shift that stresses collaboration between the business, developers, and IT professionals. Automation can certainly assist to enhance these connections at the same time that it helps organizations achieve the desired SDLC acceleration. However, it's important to recognize that automation is just one piece of the DevOps puzzle. *Simply accelerating most organizations' existing SDLC processes will not only introduce I-Love-Lucy-like chaos—it will also increase the risk to the business.*

If your organization isn't ready to bear the risks of releasing software that fails to meet expectations around security, reliability, and performance, the transition to DevOps is a prime opportunity to re-examine your quality processes. Since testing is often one of the greatest constraints in the SDLC, optimizing quality processes to allow testing to begin earlier—as well as shrink the amount of testing required—can have a marked impact on acceleration. Moreover, adopting a bona-fide Continuous Testing process (NOT just automated tests running regularly) helps promote all of the core pillars of DevOps: Culture, Automation, Lean, Metrics, Sharing.

In this paper, we'll explore why and how Continuous Testing's real-time objective assessment of an application's business risks is a critical component of DevOps.

DevOps Principles

The basic DevOps framework was introduced by Damon Edwards and then refined by Jez Humble into the acronym we now know as CALMS:

Culture – Drive collaboration and communication between silos

Automation – Remove manual steps from your value chain

Lean – Apply lean principles to enable higher cycle frequency

Metrics – Measure everything and use data to refine cycles

Sharing – Share experiences (good and bad) to enable others to learn

As you can see, automation is a core component—but automation alone won't be sufficient for achieving your DevOps objectives.

Continuous Testing: It's Not What You Think

Before we delve into how Continuous Testing promotes DevOps, let's start by defining what we mean by Continuous Testing.

Today's DevOps and "Continuous Everything" initiatives require the ability to assess the risks associated with a release candidate—instantly and continuously. Continuous Testing provides an automated, unobtrusive way to obtain immediate feedback on the business risks associated with a software release candidate. It guides development teams to meet business expectations and helps managers make informed trade-off decisions in order to optimize the business value of a release candidate.

Continuous Testing is NOT simply more test automation. 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—safeguarding the integrity of the user experience while protecting the business from the potential impacts of application shortcomings.

Continuous Testing and DevOps

It's become widely-accepted that Continuous Integration, Continuous Deployment, Continuous Release, and Continuous Delivery are key DevOps enablers. And although some level of *Automated* Testing is commonly embedded within many software delivery toolchains, the potential of *Continuous* Testing to revolutionize DevOps efforts is just now gaining attention.

Automated testing involves automated, CI-driven execution of whatever set of tests the team has accumulated. However, if one of these tests fails, what does that really mean: does it indicate a critical business risk, or just a violation of some naming standard that nobody is really committed to following anyway? And what happens when it fails? Is there a clear workflow for prioritizing defects vs. business risks and addressing the most critical ones first? And for each defect that warrants fixing, is there a process for exposing all similar defects that might already have been introduced, as well as preventing this same problem from recurring in the future? This is where the difference between *automated* and *continuous* becomes evident.

Admittedly, moving from automated testing to the level of business-driven Continuous Testing outlined above is a big leap. But it is one that yields significant benefits from a DevOps perspective. Through objective real-time validation of whether software meets business expectations at various "quality gates," organizations can automatically—and confidently—promote release candidates through the delivery pipeline. The benefits of this automated assessment include:

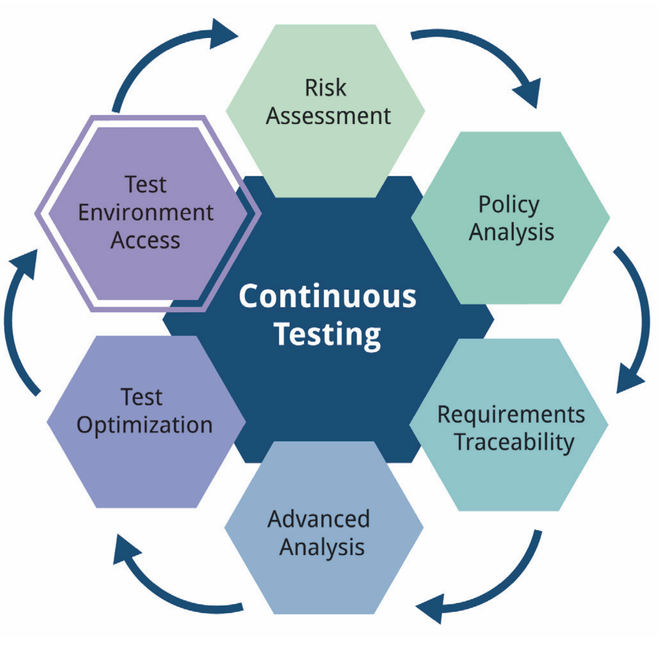
- Throughout the process, business stakeholders have instant access to feedback on whether their expectations are being met, enabling them to make informed decisions.
- At the time of the critical "go/no go" decision, there is an instant, objective assessment of whether your organization's specific expectations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

With a broad set of automated defect prevention and detection practices serving as "sensors" throughout the SDLC, you're continuously measuring both the product and the process. If the product falls short of expectations, don't just remove the problems from the faulty product. In the spirit of the "sharing" component of DevOps CALMS, consider each problem found an opportunity to re-examine and optimize the process itself—including the effectiveness of your sensors. This establishes a defect-prevention feedback loop that ultimately allows you to incrementally improve the process.

For example, if a certain security vulnerability slipped through your process, you might want to update your policy to include ways to prevent this vulnerability, then adjust your sensors to check that these preventative strategies are being applied from this point forward. If feasible, you'd also want automated tests to check for this vulnerability at the earliest feasible phase of the SDLC.

What's Needed for Continuous Testing

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



Risk Assessment—Are You Ready to Release?

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 technical 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: over time, they grow in size and complexity until they reach a breaking point when they are deemed "unmanageable." Unfortunately, test suite rationalization is traditionally managed as a batch process between releases. Managing in this manner yields to sub-optimal decisions because the team is forced to wrangle with requirements, functions, or code 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 Development Testing Platform will also optimize the prioritization of tasks.

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

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."[1]

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[2] audit. Although code within the online transaction component might not be changing, test metadata associated with the component could place it in scope for testing. Furthermore, a policy defined for the

[1] http://www.fda.gov/medicaldevices/deviceregulationandguidance/guidancedocuments/ucm085281.htm#_Toc517237928

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

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 Virtualization)

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.

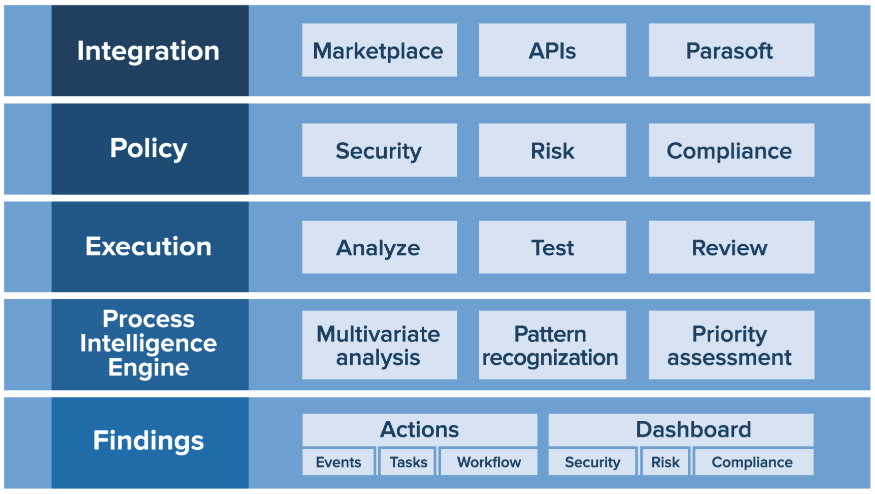
With Service Virtualization, organizations can access simulated test environments that allow developers, QA, and performance testers to test earlier, faster, and more completely. Organizations that rely on interconnected systems must be able to validate system changes more effectively—not only for performance 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.

Parasoft: quality@speed for DevOps

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.

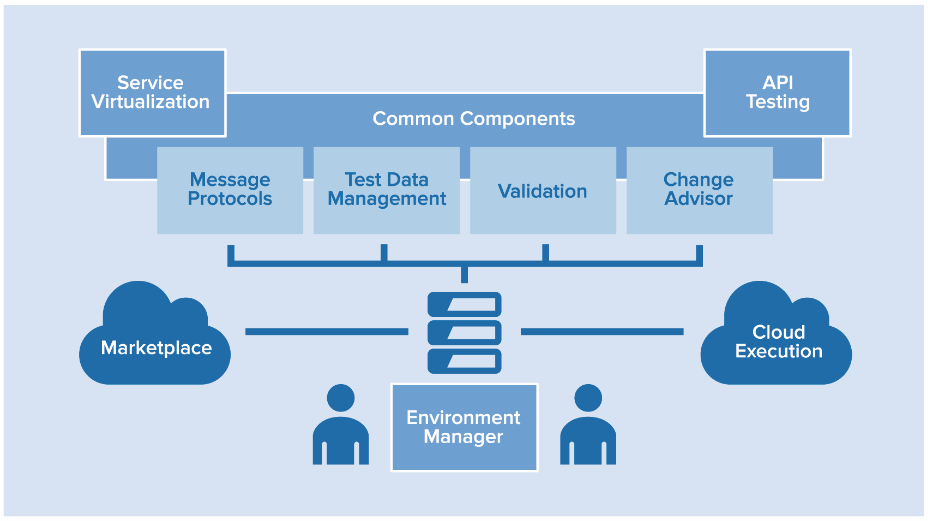


Parasoft DTP helps organizations:

- 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 Parasoft

Parasoft researches and develops software solutions that help organizations deliver defect-free software efficiently. We reduce the time, effort, and cost of delivering secure, reliable, and compliant software. Parasoft's enterprise and embedded development solutions are the industry's most comprehensive—including static analysis, unit testing, requirements traceability, coverage analysis, functional & load testing, dev/test environment management, and more. The majority of Fortune 500 companies rely on Parasoft in order to produce top-quality software consistently and efficiently as they pursue agile, lean, DevOps, compliance, and safety-critical development initiatives.

Contacting Parasoft

USA Headquarters - Phone: (888) 305-0041, Email: info@parasoft.com

APAC | France | Germany | Italy | LATAM | Poland | UK | More - http://www.parasoft.com/contacts

Author Information

This paper was written by:

- Wayne Ariola (wayne.ariola@parasoft.com), Chief Strategy Officer at Parasoft
- Cynthia Dunlop (cynthia.dunlop@parasoft.com), Lead Technical Writer at Parasoft