Continuous Testing for DevOps
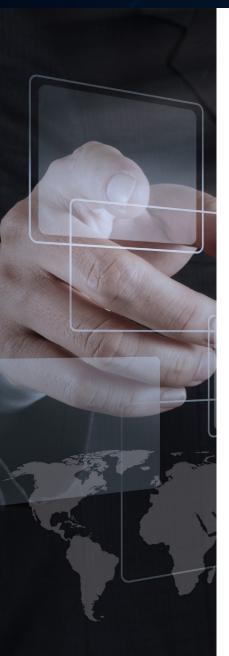## Evolving Beyond Simple Automation

INTRODUCTION

DevOps represents a cultural shift that stresses collaboration between the business, developers, and IT professionals. Software test automation can enhance these connections and help organizations achieve desired SDLC acceleration, but it's important to recognize that automation is just one piece of the DevOps puzzle.

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 (more than just automated tests running regularly) helps promote all of the core pillars of DevOps: Culture, Automation, Lean, Metrics, and 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.
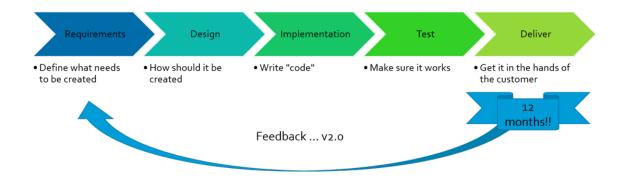
DEVOPS PRINCIPLES

There are several key pieces to understanding DevOps revolutions and they are often brought about by a compelling event at an organization, such as a shift to agile. As organizations start to move into an agile development methodology, they start to uncover other processes that can be accelerated, such as delivery by DevOps and testing by Continuous Testing. The acceleration that is set in motion via agile makes it necessary to accelerate the release schedule. In order to ensure a successful release, an organization must adopt continuous testing to make sure the conveyer belt does not break down. The modernization maturity model has these three distinct phases:
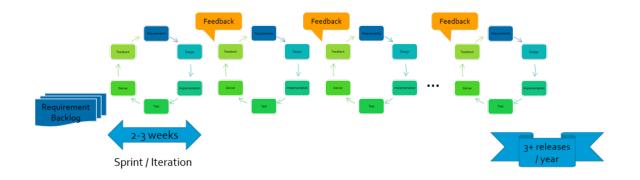
Agile → DevOps → Continuous Testing

AGILE

Agile software development is a different way of thinking about approaching the challenge of development time. Traditionally, applications are developed over a 12 month period and they go through distinct phases:

PARASOFT®
Automated Software Testing

The challenges with this traditional approach often termed "waterfall" all stem from the feedback loop. Traditionally in waterfall development, the feedback cycle takes up to 12 months, and in today's software industry, that would be certain death for any product trying to be competitive. There are programs such as beta testing, that allow for customer feedback, but these programs are limited. To really understand the customer's desires, you have to get the application to market much faster. Additionally, after 12 months of isolated development, the business expectations may change and no longer synch with development efforts. With these constraints apparent, a faster release cycle cadence is apparent, thus giving rise to agile development:

With agile, the same SDLC basics apply but the cycle is compressed and iterative. This has many advantages such as quicker feedback, the ability to react to the market, the ability to release whenever you want, and most importantly, more accurate delivery.
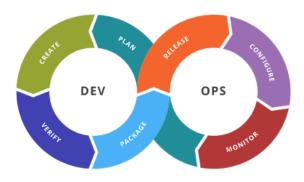
There are cons to agile and they are all centered around acceleration. Change is coming much faster and you have many additional developer check-ins to contend with, as well as a requirement to get releases into the hands of business and your customers faster. This need for an accelerated delivery pipeline becomes a catalyst for DevOps.

DEVOPS

The ultimate challenge for businesses is ensuring that the IT infrastructure is in place and working as expected prior to new application deployment. There is a requirement for a constant state of readiness. Since agile speeds up the development process, there are more frequent code check-ins, which collide when they are deployed into the environments. This causes a state of constant evolution and revolution for an environment and achieving the state of readiness required for consistent application deployment is almost impossible.

DevOps is aimed at automating all of the necessary steps required to take application code and deliver it to the end-user. DevOps works like a conveyor belt to move application code through the distinct phases.

These seven distinct phases are often referred to as the DevOps toolchain where a series of processes or "tools" are strung together in order to manage the delivery pipeline. The phases are as follows:

1. Plan
2. Create
3. Verify (Test)
4. Package
5. Release
6. Configure
7. Monitor

One of the most important phases of the DevOps toolchain is the testing phase. You need to validate that your application has an acceptable level of risk before moving it to the next phase, and build in automatic mechanisms to verify functional and nonfunctional behavior of your application. Since DevOps helps the IT side of the house start to build "infrastructure as code," an organization's ability to quickly deploy applications becomes a reality. But without thorough, multi-channel, end-to-end validation of your application in the context of the ecosystem, what you end up deploying into this accelerated ecosystem can break in ways that were unexpected and slow down the entire DevOps process. This will delay releases, decrease profitability, and ultimately put you back into the place you were before you decided to adopt DevOps.

Continuous testing is a step or a phase in the DevOps toolchain that will allow you to mitigate this risk, not only ensuring thorough end-to-end validation of the application but automating the process of testing the application.

CONTINUOUS TESTING: IT'S NOT WHAT YOU THINK

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 isn't 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

Continuous Integration, Continuous Deployment, Continuous Release, and Continuous Delivery are key DevOps enablers. 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.

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. 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.
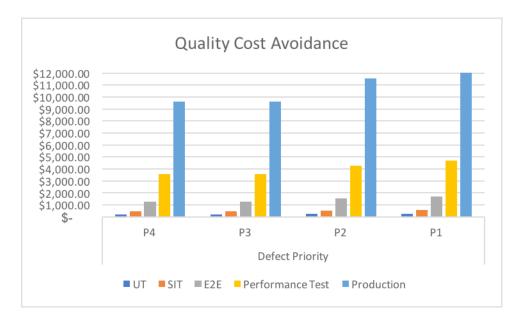
WHAT'S NEEDED FOR CONTINUOUS TESTING

Now it's time to discuss the individual factors that make up continuous testing. How do you actually accomplish it, from beginning to end?

Automated Code Scans

Continuous testing should start in the earliest phases of the SDLC. Arguably this is where an organization can actually take the best advantage of acceleration because the early phases of application development are traditionally where the worst defects are introduced. By introducing an automated testing strategy at the very beginning, an organization can rapidly identify most process defects as they are introduced into the code base,

and provide information as to how to remediate those defects before they make their way into an integrated system.

Identifying a defect in the early stages of testing is clearly beneficial to cost. The chart above represents the cost incurred in dollars per defect identified across different environments. The cost incurred of finding a priority 1 defect in a development environment is around $260, but if you let that defect slip into production, the cost goes up to $12,000. That's about a 4700% increase in cost, for 1 defect!

Early-stage testing is essential to not only build a solid foundation of the right kind of tests, but also to set a process in motion for finding the root causes when dealing with defects as opposed to reacting later on. The numbers tell the whole story:

Cost Incurred in $ per defect identified

Phase	Defect Priority			
	P4	P3	P2	P1
UT	$ 195.00	$ 195.00	$ 260.00	$ 260.00
SIT	$ 455.00	$ 455.00	$ 520.00	$ 585.00
E2E	$ 1,300.00	$ 1,300.00	$ 1,560.00	$ 1,690.00
Performance Test	$ 3,575.00	$ 3,575.00	$ 4,290.00	$ 4,680.00
Production	$9,620	$9,620	$11,570	$12,480

Early-stage testing can be executed as a part of the CI pipeline. As a developer checks their code into source control, automated scans kick-off and perform static analysis and unit tests. Within your CI orchestration software, you can pull in the results and pass/fail the build based on your policy gates. For example, you can set a threshold in which as long as 80% code coverage is achieved and no more than five level 3 or higher findings are detected, the build is considered a success, but if the coverage threshold drops below 80% or any high-priority defects are detected, then fail the build, reject the integration, and notify the development team.

One of the great benefits of this approach is that you can allow your development teams to be mostly autonomous. They can use whatever development or testing tools they prefer and can be mostly self-managed, while the business is putting a series of policy gates in place, through continuous testing, to make sure that these teams
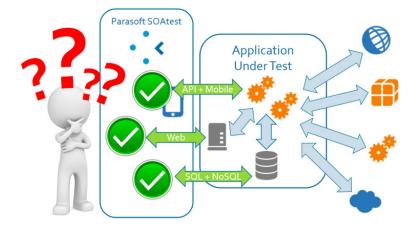
stay within "risk boundaries" defined by the business. Should they ever fall out of line, you have the data to back up whatever necessary remediation you must take to realign the team.

Automated Test execution

Once an application is built, things start getting tricky. You now have many interfaces for your application, i.e. mobile, web, API, database, etc. All of these different interfaces must be tested and adhere to the same policy gates as the code. The good thing is that if you have stuck to a rigorous automated code scan policy, the quality of the applications coming down the pipeline will be significantly increased, thereby taking some of the effort off of functional and nonfunctional testing. However, one of the oldest adages in the testing realm is that the testing team will never find fewer defects. They always find bugs, but what this ultimately results in is a much more rigorously tested application because you're doing that testing upfront.

Omni/Multi-Channel Testing with Parasoft SOAtest

But how do you do the type of complex, end-to-end, multichannel testing required for today's modern, distributed, microservice rich applications? How do you achieve they type of quality@speed required for continuous testing? It really comes down to understanding the contracts that exist between different applications and taking advantage of those contracts to ensure not only API coverage but the impact of change.

API coverage simply means; "Did I test all of the resources described by my service?" Your typical contract, whether it's a WSDL, Swagger or RAML document, will indicate what available services/ resources exist for this application. Your API testing tool simply needs to consume that contract and ensure that it's tested across all of those different interfaces. From that, you will be able to tie together the different API tests with the components that they are testing as well as the requirements that are driving this test initiative. We'll talk about requirements in the next section.

You can then start to create different test scenarios by bringing together all of the different types of testing that you require, and keeping a tight association with the application components to which these tests belong. This mapping is critical to understand which tests touch which code and more importantly understanding: when the code changes, which tests need to be re-run?

This is an excellent opportunity to start creating nonfunctional tests (performance, load, security, penetration, etc.) because you have the underlying "framework tests" in place. Simply leverage the tests you just created to validate API unit level as well as API scenario and convert them into security and performance tests either at the team/application level or at the organizational/LOB level. All of these tests then get checked into source control and brought into your library of test cases.

At this point, the story is the same as the automated code scans. Once a developer checks code into source control, that code is associated with a particular application. An application can then be built by the CI orchestration

software, such as Jenkins, Team City, Bamboo, etc. Once the application is built, you can reach into your test library and execute the required tests that need to validated at the unit > API > scenario > Integration levels, all the way up to nonfunctional, load, security and performance.

These tests can be executed directly from the CI orchestration system and the results from these tests get pulled back into the same build. The same policy gates can be applied to the results so that, in an automated way, you can decide whether this application can go to the next stage of testing. An example of one of these policy gates would be: an API coverage rate of 80% of all application services, a 90% Performance SLA agreement, and all their test results succeeding. The build will be automatically promoted to the next environment and more rigorous testing be run. If not, it would fail the build and reject the deployment.

This is true continuous testing.

Requirements Traceability

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 also 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 reporting and analytics dashboard 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 reporting and analytics engine should 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.

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 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?
- Now that my application has changed, which tests do I need to automatically rerun

The goal is to have a single view of how 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.

---

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

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.

More importantly, you then have the ability to rerun the associated test if the underlying application code changes. Since earlier you set up the automation framework required for running these tests, you can simply link the code to the test mapping in the automation system and execute the associated tests when the code changes.
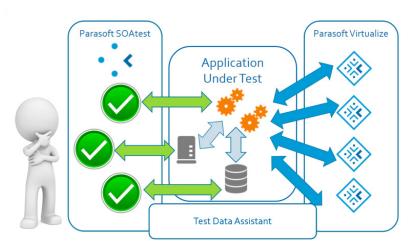
Test Data

Access to realistic test data can significantly increase the effectiveness of a continuous testing strategy. 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 next) 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.

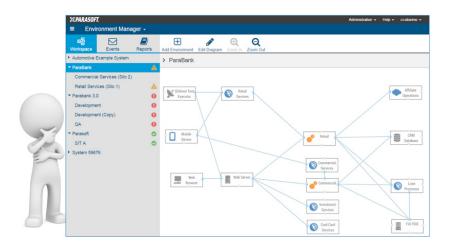## Total control of the Test Environment

Additionally, and most importantly, data generation becomes an essential part of continuous testing. At the most basic level, there is an assumption that comes with infrastructure as code that says "I know what my ecosystem looks like **right now.**" Service contracts help us to define how we should be communicating and what the response should look like but **data is ever-changing.**

Data-generation underpins continuous testing because you can continuously generate data appropriate for the type of scenario you're trying to execute instead of trying to rely on production data sources and hoping that all the right data is in the right place. Combining data generation with simulation will allow you to inject the right data in the right place at the right time.

TEST ENVIRONMENT ACCESS AND SIMULATION (SERVICE VIRTUALIZATION)

With the convergent trends of parallel and iterative development, increasing system complexity/interdependency, and Microservices, it has become extremely rare for a team to have ubiquitous access to all of the dependent applications required to execute a complete test. Simulation solves this problem.

Environment based approach to testing

You have highly-distributed development and test teams that need simultaneous on-demand access to a release candidate in order to continuously test throughout the software lifecycle, as well as its myriad of Microservice APIs and dependencies that must be present in the test environment. Using a conventional onpremise 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 or more importantly do not exist in the test environment.

To eliminate these constraints, teams can leverage service virtualization, which allows you to leverage innovative system cloning and simulation technologies to rapidly configure, provision, scale, and reproduce complete dev/test environments. Service virtualization empowers you to simulate the behavior of dependencies you can't trust (e.g., third-party services, databases, mainframes, not-yet-implemented APIs, etc.) or want to stabilize for test coverage purposes. Dynamic environment deployment techniques are becoming more ubiquitous within DevTest organizations, yet most organizations are not yet using service virtualization to its full extent.

By leveraging service virtualization in conjunction with DevOps, teams can remove environmental constraints, enabling the organization to gain full access to (and control over) the test environment, further enabling Continuous Testing to occur as early and often as needed.

Within the construct of continuous testing, teams can create virtual assets that represent their dependencies. These virtual services can then be associated with the test cases that have been identified for execution of a particular application during the requirements and traceability phases. At this point, you have a template defining

what is required in order to isolate an individual component within an ecosystem, and the required test cases that will be executed to validate it. What you are building is a dynamic environment or container that your application will live in temporarily during continuous test execution. This environment template can then be deployed on demand from the CI orchestration software.

Dynamic test environments

This **environments-based approach to continuous testing** is only possible with service virtualization. An environments-based approach to testing is the best way to achieve true dynamic continuous testing. Through this, QA and performance testers can test earlier, faster, and more completely. Organizations that do this successfully validate interconnected 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.

CONTINUOUS FEEDBACK

At the end of the day, it all comes down to results. By leveraging an environments-based approach. Organizations can realize the possibilities of true continuous testing by having a large suite of test cases that execute continuously or as a function of code check-in, that not only automates the go or no-go decision but also helps an organization understand their current level of risk associated with any application. By also leveraging a powerful results/analytics dashboard, the business can, at any moment, take the temperature of an application and immediately re-prioritize resources if necessary. This agility truly enables a team to adopt a DevOps strategy and to make the most advantage at of their agile initiative. For teams to benefit tremendously from the agile initiative, they must adopt a DevOps strategy to enable true agility.

ABOUT PARASOFT

Parasoft helps organizations perfect today's highly-connected applications by automating time-consuming testing tasks and providing management with intelligent analytics necessary to focus on what matters. Parasoft's technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software, by integrating static and runtime analysis; unit, functional, and API testing; and service virtualization. With developer testing tools, manager reporting/analytics, and executive dashboarding, Parasoft supports software organizations with the innovative tools they need to successfully develop and deploy applications in the embedded, enterprise, and IoT markets, all while enabling today's most strategic development initiatives — agile, continuous testing, DevOps, and security.

www.parasoft.com

Parasoft Headquarters:
+1-626-256-3680

Parasoft EMEA:
+31-70-3922000

Parasoft APAC:
+65-6338-3628

PARASOFT®
Automated Software Testing