SPECIAL REPORT

Testing in the Age of Agile and DevOps

PARASOFT®
Automated Software Testing

As agile and DevOps move further into the mainstream, testing must evolve to keep pace. Agile's frequent releases and DevOps' continuous deployment leave little time for testers to ensure the product they deliver is secure and error-free.

This special report addresses common challenges testers face in this fast-paced environment and offers tips to kick start the test automation and API testing initiatives you must implement to succeed in today's market.

In this API Testing Special Report

How Do You Choose the Right API Testing Tool?

Key Capabilities to make your Organization Successful in Today's Testing Environments

When it comes to API testing, where do you start?

There's no question that API testing is integral for identifying defects at multiple layers of your application and ensuring a seamless customer experience. But there are many different approaches and tools available on the market. How do you get the ROI you're looking for to achieve the automation necessary to deliver high quality software at the speed of Agile and DevOps initiatives?

Use this guide to gain confidence choosing a tool to help you address today's key challenges, such as omni-channel testing, test maintainability, microservices, difficulties reproducing defects, and the impact of rapidly changing APIs. It is more important than ever to choose the solution that is right for you and your organization.

What Is API Testing?

API testing enables developers and testers to test the series of components that make up their application. Without API testing, they would be stuck manual testing, just like a user, testing the application at the UI level, waiting until the entire application stack is built before being able to start testing.

You can instead perform automated API testing by testing the application at the API level, designing test cases that interact directly with the underlying APIs, and gaining numerous advantages, including the ability to test the business logic at a layer that is easy to automate in a stable manner. Unlike manual testing, which is limited to validating a specific user experience, API testing gives you the power to bulletproof your application against the unknown.

Testing earlier, at the API level, helps you "fail fast and fail early," catching defects early at their source, rather than later in the software delivery process. API tests include service tests, contract tests, component tests, scenario tests, load/performance tests, security tests, and omni-channel tests.

API tests can all be automated and run continuously, so you can ensure that your application is aligned to business expectations while also functionally precise. Since API tests work at a much lower level than UI tests, you know that you will have consistency and the tests that you are building will last for a long time to come.

The API Testing Tool Landscape

There are two main types of API testing solutions, broadly categorized into lightweight and enterprise tools.

Lightweight Tools

Free or open-source tools (i.e. Postman) are great because they help you get going quickly, create very simple smoke tests, apply basic validations, and we often see them used by development teams to test in an ad-hoc way. A lot of these tools are free in the short term (although become more expensive when it comes to vendor support and maintenance).

Enterprise Tools
More powerful, vendor-supported tools enable daily testing of APIs, supporting a broader range of technologies. These comprehensive tools support complex DevOps workflows and easily scale across teams and organizations to enable continuous testing.

So What Do You Need?

You want to be able to do everything required for your API testing requirements, but still have a solution that's easy to use, adopt, and scale throughout the organization. Continue reading this guide to understand the key capabilities you need in your API testing solution to get the best of both worlds.

Ease-of-Use and Core Capabilities

- ✓ VISUAL AND SCRIPT-LESS FUNCTIONALITY
- ✓ A CUSTOM EXTENSIBILITY FRAMEWORK
- ✓ AUTOMATED ASSERTIONS AND VALIDATIONS
- ✓ DATA-DRIVEN TESTING
- ✓ TEST RE-USABILITY
- ✓ ABILITY TO RAPIDLY CREATE TESTS BEFORE A SERVICE IS AVAILABLE
- ✓ AUTHENTICATION

✓ VISUAL AND SCRIPT-LESS FUNCTIONALITY

Your API testing tool should not require you to have any experience writing code. A visual testing tool with an intuitive and scriptless user interface will empower a large body of testers (at a variety of experience levels) to use the tool productively. API testing can be overlooked when developers push it to QA, and QA focuses on manual testing. Having an API testing tool that is visual and script-less will enable testers to adopt this critical testing practice without having to spend lots of time on training and enablement.

✓ A CUSTOM EXTENSIBILITY FRAMEWORK

Should you need to write code to accomplish tasks such as generating a proprietary token or a unique identifier, your API testing tool should have the capability of using the scripts, but it should not be limited to a single language. Different testers and developers use different scripting languages depending upon the level of expertise and preference, so at a minimum, your API testing tool should support Java, Jython, JavaScript, and groovy.

Your API testing tool should also include a framework that allows you to extend the tool's capabilities, built-in transports, and protocols. This will enable you to support any transport or protocol that your organization is using, whether it is an industry standard or custom tailored to you. Examples of these might be DDS, plain socket, TCP, file-based messaging, etc.

✓ AUTOMATED ASSERTIONS AND VALIDATIONS

It is important for your API testing tool to help you define success criteria for response validation. This process should enable the tests to run automatically as a batch, validate messages, and eliminate the need for you to manually inspect your traffic. Your tool should be able to automatically compare responses to those expected, in bulk, as well as enable you to surgically validate a single element. Additionally, your tool should be able to validate the schema of a request or response payload to ensure the service is complying to it service definition.

✓ DATA-DRIVEN TESTING

For maximum flexibility when working with your API testing tool, it should be able to data-drive your test cases. This means that your framework should enable you to swap out static values in your API calls with dynamic values derived from data sources. It should support a wide range of data sources including CSV, Excel, JSON, as well as simple in-project tables.

You should be able to leverage dynamic data sources as well. An example of this would be connecting to a live database and pulling dynamic data at runtime, as well as a writable data source that can be updated on the fly. Additionally, you should be able to aggregate data sources together and have a mechanism for rapidly switching between them.

✓ TEST RE-USABILITY

Your API testing solution should be able to re-leverage any API test you have previously created. This gives you the ability to define scenarios such as web UI logins, complex authentication, or a repeated set of actions once and then bring them into subsequent test cases as a reference. This will help you maintain your library of test cases by providing a location where you can update critical paths and have child test suites inherit that information.

✓ ABILITY TO RAPIDLY CREATE TESTS BEFORE A SERVICE IS AVAILABLE

It's important to ensure that your API testing solution helps you rapidly create tests early in the software development process. This is critical because in order to adequately take advantage of the benefits of agile, you will need to focus on new features and functionality introduced into each Sprint. Often this takes the form of modifications to UIs and APIs. Being able to rapidly test this functionality before it's available ensures that you will be able to maximize your test coverage and ensure your critical use cases.

To do this your API testing solution must be able to consume service definitions such as Open API/Swagger, WSDL, RAML, as well as schema definitions.

✓ AUTHENTICATION

Your API testing solution also needs to work with authentication, encryption, and access control. A large number of your services will be deployed via an encrypted protocol such as SSL, as well as having a security policy such as Oauth, Basic auth, Kerberos, payload encryption, SAML, Signatures, etc.

It is important to be able to communicate using these authentication mechanisms, so your API tool must support all of the common standards. Additionally, you will need to validate your security is working properly, so your API testing tool should have a mechanism to ensure that the standards are implemented properly and work flawlessly.

Optimized Workflows

- ✓ TEST FLOW LOGIC
- ✓ AI-POWERED TEST CREATION
- ✓ TEST DATA MANAGEMENT / GENERATION
- ✓ EVENT MONITORING
- ✓ BDD SUPPORT (CUCUMBER)

✓ TEST FLOW LOGIC

Your API testing tool should have a mechanism for controlling test flow based on conditions. Not all test scenarios will execute in a linear fashion, so you may need to make automatic decisions at runtime that will affect how your test executes. An example of this might be ensuring that a response contains a specific element prior to moving to the next test step. Additionally, you may want to pause execution and poll a web service for a while to ensure a process has taken place, so your API testing solution must have the ability to analyze responses for key criteria and then use that information to control the rest of the test execution.

✓ AI-POWERED TEST CREATION

Your API testing solution must be able to work in an agile context. A key capability for success here is intelligent test creation, that brings significant efficiency gains. Your API testing tool should help you build test cases by understanding actions that you do repeatedly and learning how to do them for you. By leveraging artificial intelligence, your API testing solution will be able to monitor actions you're already testing manually with your applications, and automatically infer the relevant API calls from these actions. By streamlining the activities you do often and learning how to optimize the test creation process, you will be able to achieve a more successful API testing rollout.

✓ TEST DATA MANAGEMENT / GENERATION

Testers can spend a lot of time gathering adequate test data. Your API testing tool should support you in this activity by providing workflows for connecting to various data sources as well as generating test data itself. Your solution should have the ability to understand the types of data you require for given scenarios, and build on that test data with additional use cases so that your test cases can be as flexible as possible.

✓ EVENT MONITORING

To enable end-to-end testing, your API testing solution must be able to monitor events as they flow through your system, so you can validate inputs and expected outputs, and understand how transactions transform as they move through your application. With multi-step validation by plugging into your application internals via JMS messaging, database monitoring, etc., your solution will be able to provide greater levels of test coverage.

✓ BDD SUPPORT (CUCUMBER)

To support BDD, your API testing solution should give you the foundation for Cucumber step definitions that your QA team needs to build the required test steps for business analysts to leverage in their BDD. This will broaden the usage of your solution across key stakeholders and ensure that your testing is done at the earliest stage possible.

Supported Technologies

- ✓ REST API TESTING
- ✓ SOAP API TESTING
- ✓ MQ / JMS TESTING
- ✓ IOT AND MICROSERVICE TESTING
- ✓ DATABASE TESTING
- ✓ WEB-BASED TESTING
- ✓ PERFORMANCE TESTING
- ✓ TESTING NON-STANDARD MESSAGE FORMATS

✓ REST API TESTING

Your testing tool must be able to interface with Representational State Transfer protocol (REST) APIs. This includes support for service definitions such as Open API OAS3, Swagger, or RAML. Your tool must be able to send URL, Method, Path, Parameter, Query, and JSON payload information, as well as Headers, Mime Types, Attachments, and so on. Additionally, it needs to be able to consume and validate request and response for simple regression or schema validation.

✓ SOAP API TESTING

Simple Object Access Protocol (SOAP) is still extremely relevant in most applications, so your API testing solution must be able to interface with SOAP APIs. This includes support for service definitions such as WSDL and schema XSD. It must be able to send SOAP-compliant requests that include SOAP action, attachments, WS policy, and relevant SOAP headers. Additionally, it needs to be able to consume and validate SOAP XML responses for simple regression or schema validation.

✓ MQ / JMS TESTING

Your API testing solution must support queuing technology such as MQ and JMS, so you can simulate various patterns including point-to-point and publish/subscribe. This will allow you to do complete end-to-end testing, and validate message systems that leverage these technologies.

✓ IOT AND MICROSERVICES TESTING

IoT and microservices are bringing a host of new testing challenges. While still largely REST/JSON based, the future of IoT and Microservices will have interfaces deployed on nonstandard protocol (i.e. Websockets, MQTT, AMQP/Rabbit MQ, Kafka, and Protocol Buffers). Your API testing tool must be future-proof and have the ability to communicate via these new protocols, so that as they begin to be implemented at your organization, you are ready to test them.

A second testing need is the ability to isolate individual Microservices that are deployed via an asynchronous protocol. Your API testing tool must be able to emulate not only a provider to the queue or topic but also

have the capability of consuming messages off of the topics, so you can isolate and test individual services.

✓ DATABASE TESTING

Your API testing tool must have the ability to communicate with databases. This will give you the ability to validate the contents of the database as you are interacting with your system via the APIs. This will enable end-to-end testing and ensure that your messages traverse through your system properly. Additionally, by being able to connect with your databases you will be able to pull data out of the database prior to API test execution. This will make your API test scenarios more dynamic by being able to use the most relevant and up-to-date data.

✓ WEB-BASED TESTING

Your API testing tool must have an integrated capability for testing your web UIs. This does not need to be your only web UI testing solution (i.e. Selenium), but should complement your API testing so that you can test across multiple interfaces exactly the same way your customers will interface across your business. An example of this would be using your web UI capability to log into an application and begin a transaction. You could then leverage your APIs to hit individual components to validate the transaction has seated into your application properly, and then finally you would be able to pull from the database to ensure the relevant information was stored appropriately.

✓ PERFORMANCE TESTING

Your API testing solution should allow you to shift left performance testing by leveraging the API tests you have been creating for individual component, smoke, and regression tests as a part of your nonfunctional load and performance testing strategy.

Your API testing solution should allow you to generate load via a built-in mechanism for controlling the number of transactions or number of virtual users. By having this performance testing capability integrated into your API testing solution, you will be able to conduct more performance tests earlier in the software delivery lifecycle.

✓ SUPPORT FOR NON-STANDARD MESSAGE FORMATS

Your API testing solution must be able to communicate over non-standard message formats and protocols, such as mainframe (copybook), fixed length messaging, or Electronic Data Interchange (EDI, which is often found in the exchange of computer to computer business documents), as well as industry standard message formats and protocols such as FIX and SWIFT. It is also important for your API testing solution to be extensible, to include additional proprietary message formats and protocols that your business demands. The extension of these protocols should be scriptless and modular.

Automation

- ✓ CI INTEGRATION
- ✓ BUILD SYSTEM PLUGINS
- ✓ COMMAND-LINE EXECUTION
- ✓ OPEN APIS FOR DEVOPS INTEGRATION

✓ CI INTEGRATION

Your API testing tool should have the capability to integrate into your existing CI process, so you can execute test cases via a command line interface or a series of open REST APIs, and select which type of test cases you want to execute with a specific configuration. Additionally, you should be able to retrieve the results and process them back into your CI pipeline to make an automated go or no-go decision in your deployment activities.

✓ BUILD SYSTEM PLUGINS

Your API testing solution should have built-in plugins for common CI systems such as Jenkins, Microsoft VSTS, Atlassian bamboo, and Jet Brains Team City. There are other build systems available, so your API testing solution should have an extensibility platform to allow you to build connectors for all future CI systems.

✓ COMMAND-LINE EXECUTION

Your API testing solution should have the ability to execute your API tests in batch via a command line interface. The command line interface should be easy to use so that you can set up a variety of batch execution scenarios across multiple interfaces without having a heavy reliance on scripting.

Your command line interface should be dynamic so that you can swap out variables, data sources, and test environments by simply modifying flags. This will ensure the command line interface is readily used by your organization.

✓ OPEN APIS FOR DEVOPS INTEGRATION

Your API testing solution should have open APIs that allow you to programmatically generate, configure, and execute test cases. This will allow you to set up a client/server configuration for your DevOps platform and your API testing platform. A series of open APIs will allow you to set up a scalable infrastructure and reduce overall licensing costs by programmatically making calls to the API testing server from multiple parts of your organization as needed to execute the proper test cases.

Management and Maintenance

- ✓ INTEGRATION WITH REQUIREMENTS MANAGEMENT SYSTEMS
- ✓ BASIC AND ADVANCED REPORTING
- ✓ TEST ORCHESTRATION
- ✓ A PROCESS FOR MANAGING CHANGE
- ✓ ON-PREMISE AND BROWSER-BASED ACCESS

✓ INTEGRATION WITH REQUIREMENTS MANAGEMENT SYSTEMS

Your API testing solution should integrate with requirements management systems such as ALM, Bugzilla, and Jira. You should be able to identify to which requirements specific API tests are associated, and have a mechanism to understand how the results of that API test affect the individual requirements.

✓ BASIC AND ADVANCED REPORTING

Your API testing solution should have a rich reporting framework that allows you to understand individual API test results, as well as the entire project health. Your reporting framework should allow you to generate shareable documents that can be customized to individual stakeholders needs, and be machine-readable so they can be processed by your build systems and provide a simplified integration into your CI pipeline.

✓ TEST ORCHESTRATION

Your API testing solution should provide the ability to bundle together API testing scenarios for execution in multiple environments. This will enable you to create a test run that is customized to an application in an environment. Overall this will make your test cases easier to manage by allowing you to provide dynamic information into the scenarios such as data sources, environment endpoints, user access control, etc. while reusing the same test cases.

✓ A PROCESS FOR MANAGING CHANGE

One of the most critical capabilities of your API testing solution is a change-management process.

First and foremost, your API testing solution must natively integrate with your source control system. This will allow you to maintain several versions of your API tests, for forwards and backwards compatibility.

Additionally, your API testing solution must be able to understand the various versions of your APIs via service definitions, or schemas, and should your API service definitions change, your library of test cases should have an automated re-factoring process. This

should be automatic in the case of small changes, or have an easy-to-understand process when dealing with larger changes.

✓ ON-PREMISE & BROWSER-BASED ACCESS

Your API testing solution must be broadly available. This means that you should be able to access your solution either on Prem or through a cloud provider, and your API testing solution must have a desktop application as well as a browser-based application. This flexible deployment will help broaden access and adoption of the solution, as well as help teams collaborate and reuse artifacts. As a part of enabling browser-based access, your API testing solution should have the proper user access controls in place so that your company's information is secure.

About Parasoft

From development to QA, 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.

Powerful reporting and analytics help users quickly pinpoint areas of risky code and understand how new code changes affect their software quality, and groundbreaking technologies that add artificial intelligence and machine learning to software testing make it easier for organizations to adopt and scale an efficient software testing practice across development and testing teams.

www.parasoft.com

5 Key Factors to Achieve Agile Testing in DevOps

By Denise Rigoni

In recent years, many organizations have been impacted by DevOps.

Some have implemented DevOps end to end by changing people's mindsets, automating deployment and build processes by implementing appropriate tools and processes, increasing test automation, breaking up silos between development and operations, and automating monitoring and reporting. Most organizations, though, have just started on their DevOps journey or are somewhere in the middle of the transformation process.

Part of the path to DevOps requires adoption of agile methodologies. What does it mean for testing when you switch from the traditional waterfall model, with a few long release cycles per year, to the agile model, with changes occurring every two weeks, or possibly even more quickly?

In most organizations, the world of legacy systems collides with the world of modern applications. However, the two need to coexist. In the past, we were executing regression tests for legacy systems over several weeks—manually as well as automated—but we don't have the same amount of time for testing anymore.

Switching from a traditional waterfall model to an agile one requires new testing approaches.

Optimizing test automation is an excellent way to bridge this gap. With automated tests in place, it is possible to allocate scarce testing resources to high-value activities, reduce time spent on test execution, and increase the number of test cycles possible in a shorter amount of time. The impact of these changes can be realized immediately with reduced efforts, cost savings, and dramatically improved time-to-market readiness.

My organization has implemented a lot of automation in the last few years—and made many mistakes in the process! But we also learned a lot and are now able to adapt better to a world that's becoming more and more agile. Automation only works if you have robust tests that can run unattended from the beginning to end.

Here are five key factors to achieve agile software testing in DevOps.

1. Test Data Management

Having the right test data is the first important step in automation. If your test data is not stable, you will never succeed. According to software industry statistics, nearly 30 percent of test execution failures are due to improper test data.

For each of your test cases, define what data is needed in order to execute it. Keep it flexible, describing the attributes your test data needs to fulfill rather than the test data itself. For example, "John Doe from UK" is the test data itself, and the correct definition would be "Male natural person with nationality UK."

Once you have defined the required attributes, where is the best place to find test data? To be completely flexible, and with the new GDPR legislation especially, it makes sense to build up required test data synthetically. This is a good option if you don't need historical data (even though that is also possible to build) and if your data setup is not too complicated or distributed across too many different systems.

Next to missing test data, unstable, unavailable, or incomplete test environments are one of the biggest time-consumers in testing.

If you require more complex data, you might use existing customer data that has been anonymized. In this case, queries are needed to find data in the databases matching the exact combinations of attributes.

Usually, a combination of both approaches works best. Creating synthetic test data and running queries on a database to attain highly suitable test data can and should be automated.

Proper test data management is not only a must for automation; it's required for manual testing as well. According to some surveys and statistics, manual testers spend 50 percent to 75 percent of their effort on finding and preparing appropriate test data. Test data must be independent from people maintaining it for you to get a return on investment.

2. Flexible Steering

When you design your test cases, use steering parameters instead of copying the same test case several times to fulfill certain criteria.

Let's assume you need to run your test case first in a development environment and later on in a production-like environment. You would need to log in with role X and role Y. It's inefficient to copy the test case four times for each combination of environment and role when you can enter these parameters only once in the test case as a steering element. This reduces the overall maintenance of this test case by 75 percent.

Whenever possible, it's advisable to work with reusable test step blocks. Frequently and repetitively used test steps can be defined once and reused in different test cases. For example, with a login for an application or generation of a new client, which are used over and over again for subsequent test cases, there will no longer be the need for repetition, which saves valuable time and effort.

A failing test case often results in an end to the automated execution; therefore, the next test case cannot run, because the application was left in an "undefined" state. Imagine starting the automated execution in the evening to run the test cases overnight. The next morning you would find your test cases were not executed because the first test case that had failed and prevented all the others from running.

This can be avoided by defining recovery scenarios for each test case or making them completely independent of each other. Instruct the test case what to do in case of failure.

3. Test Environment and Service Virtualization

Next to missing test data, unstable, unavailable, or incomplete test environments are one of the biggest time-consumers in testing.

Consider this scenario: Your tests are planned, testers organized and blocked their time on a certain day, and, finally, when you want to start testing the application, it is not available. Or the application itself is available, but another dependent application or dependent service delivering the data you needed is not.

The more people and applications involved, the more complex the test environment becomes, and the risk of unavailability increases.

However, in our modern agile world, the days of

waiting to test everything until all components are developed and all systems and services are completed are gone. Instead, you need to simulate those components or systems that are not yet connected or developed, so testing can be started earlier. This is possible with service virtualization. You need to know the parameters a service expects and the data it returns.

A good service virtualization testing tool can react to changing circumstances and switch from virtualized services to the real service if it becomes available, then back to virtualized services if the service is not available for whatever reason.

Bear in mind you can't do all your testing based on virtualized services; at a certain point, you'll have to start real, end-to-end testing. But many critical defects will be found before your end-to-end tests if you start testing much earlier. This saves a significant amount of rework and debugging costs later.

Additionally, a well-managed test environment helps to improve stability. This can be achieved with organizational measures (e.g., a centralized test environment management team that coordinates and controls the test environments) or defined test environment management processes.

Many organizations do not want too many processes put in place because they believe agile development doesn't need processes. However, certain processes—or, rather, rules of engagement—are necessary to improve test environment stability.

These processes don't need to be very time-consuming, at least for more advanced test stages, such as end-to-end tests or business verifications:

- If several applications and services from different teams are involved, define time slots when deployments should happen—testers will then know when everyone is done and they can start testing
- Actions like restarts of applications and services, database refreshes, or other outages should be planned during time slots where no testers will test
- If testers communicate when they plan to test—either manual or automated test cases—such as by calendar, a test environment management team can better coordinate and communicate
- The more detailed new changes are communicated to the testers, the better they can adapt their test cases

If scripted properly, DevOps automation can also help manage environments.

4. Test Users

When automating test cases, it's best to never include your personal credentials in the automation. If you're out sick, your colleague won't be able to execute the test cases you've designed. Also, using another person's personal user account is typically a security breach in most companies.

The correct approach is to set up special test users that cover the different roles you require for testing your applications. Test users will persist if someone leaves the company or changes their role. Just make sure you take into account roadblocks like single sign-on, remote desktop connections, firewalls, cross-border access, or other barriers.

5. Continuous Testing and Continuous Integration

Integrate your automated test cases into the continuous software development cycle. If the test cases are stable, why not run them every night or after every build to quickly get results about the status of the software? Ensure long-running tests are only integrated in test automation that runs during night builds, when enough time is available to finish execution before the next build starts.

Some automation tools integrate with standard development tools that provide continuous integration capabilities. Use distributed execution to run automated test cases on several machines in parallel to reduce testing time and get the results as early as possible.

Performance Testing for Our Modern, DevOps World

By Paola Rossaro

Performance testing used to be an optional aspect for most applications—something you did at the end, if you had time and you really cared. Now, it's a must.

User expectations today are much more sophisticated (you compete not only within your industry; your app is compared with everything else users experience online), and software is becoming more complex, with many third-party components deployed over cloud infrastructure.

And this is not limited to the performance of your app. Your development process needs to be fast and agile to guarantee a faster time to market and rapid response to customers' needs.

Enter DevOps, a new approach to the whole development lifecycle, made possible by the introduction of modern approaches such as continuous integration and continuous deployment. The need for automation is critical to DevOps, not only for development and deployment, but also for testing. You don't want testing to become the bottleneck in your development process.

Continuous testing is possible thanks to new solutions that allow for quick test creation and automation. Testing that used to take weeks to cycle through QA can now be accomplished within hours—no bottleneck.

Consequently, testing now can and must be an integral part of the application lifecycle. In the early stages of development, you can use new techniques based on test-driven development. During integration, functional testing solutions automate most if not all the required tests. And in preproduction, acceptance and performance tests complete the QA phase.

Performance testing is no longer an add-on. It is now a required component to guarantee applications can scale rapidly and reliably.

Performance testing solutions make it easier to quickly identify problems before the customers see them. The key is automation and iteration. It is important to be able to test frequently, rapidly, and efficiently.

As the cloud enables distributed architectures, software is also moving in that direction. There has been an increase in adoption of third-party cloud services and, more recently, containers and microservices. These new technologies have helped conclude the era of the monolithic application and made it possible to create modular architectures, where logical entities can be developed and tested separately.

The boundaries of each logical component are well defined, so there is a built-in resilience against failures from other components. The change process gets simplified and pieces can be tested and deployed separately. The identification of issues and problems also becomes faster, as each component is first tested separately.

Containers and microservices also have an impact on performance. It makes it easier to test the performance of individual components in isolation, especially when changes affect only some areas. Containerized environments make it easier to optimize the application by validating changes to the environment or system configuration.

As DevOps-based methodologies are more broadly adopted, the software testing process will increasingly move to a continuous testing model. While this may be a significant adjustment in the near term, the use of modular software architectures and continuous testing will have a positive impact on the overall application performance, empowering DevOps teams to deliver better products faster.

The Shift-Left Approach to Software Testing

By Arthur Hicken

The campaign for agile and DevOps teams to shift left is about moving critical testing practices earlier in the development lifecycle.

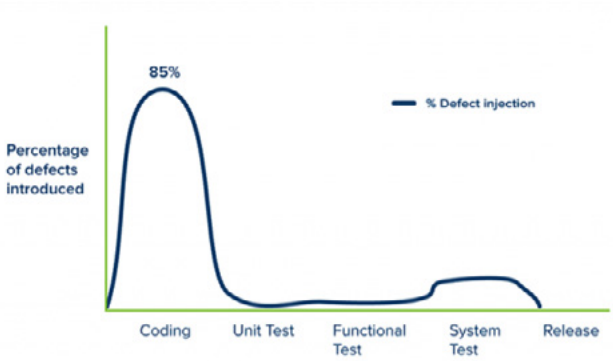
Many testing activities occur late in the cycle, where it takes longer to figure out problems and costs more to fix them. When you wait to perform testing practices later in the development cycle, your nonfunctional business requirements in particular, such as security and performance testing, are so fundamentally ingrained in your code that all you can really do is patch them up rather than fix them properly.

Shifting left is about doing this identification and prevention of defects sooner.

Detecting and Fixing Software Defects

The shift-left testing strategy is well illustrated in the somewhat famous graph from Capers Jones, below, which shows the increasing cost of bugs and defects as they are introduced into the software at each phase of software development.

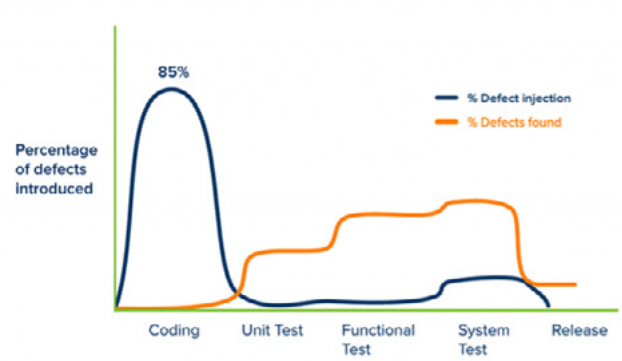
The first part of the graph shows that the vast majority of bugs come in during the coding phase, which is to be expected.



Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*

Whether they make mistakes, misunderstand the requirements, or don't think through the ramifications of a particular piece of code, developers introduce defects as the code is produced. Defects are also introduced into the application when it's time to fit the pieces together, especially if multiple teams are involved (and as modern architectures like microservices get more complex).

Now let's overlay onto the same graph the line that shows when defects are *found*. Notice that it is basically an inverse of the first line:

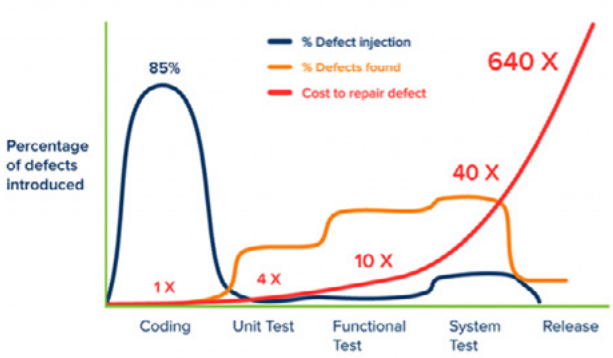


Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*

This also isn't surprising, because typically you find bugs when you start testing, and it can be difficult without a proper infrastructure to begin testing before everything is ready.

But what we also see here is that while bugs are mostly introduced during coding, they are almost never found at that phase. What does it cost to fix these bugs?

It becomes important to understand the difference it costs to fix defects at each phase of development. This is represented with a third line:



Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*

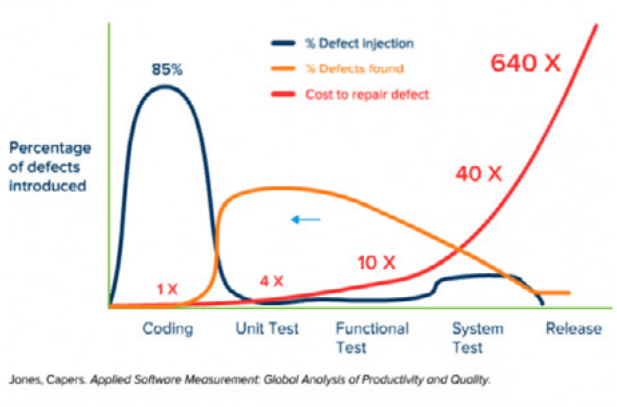
Now it starts to get really interesting, as we see a nasty progression of cost that dramatically increases the later the defect is found. Letting a bug sneak through to system testing is forty times the cost of finding it during coding, or ten times more costly than finding that same bug during unit testing. And it gets ridiculously expensive when you look at the numbers for letting bugs slip through to the actual deployment.

There are a few reasons for this cost escalation:

- The time and effort it takes to track down the problem. The more complex the test case is, the more difficult it is to figure out which part of it is the real troublemaker
- The challenge of reproducing defects on a developer's desktop as dependent systems like databases or third-party APIs are brought in (it's common for organizations to experience a lag of several weeks between defect detection and defect remediation in these situations)
- The impact of the change that is needed to fix a defect. If it's a simple bug, it doesn't matter so much, but if you have it in many places, you've used the wrong framework, or you've built code that isn't scalable enough for the expected load or that can't be secured, it's a larger problem

The Reasons behind Shifting Left

Now look at the orange line on the graph below, as it illustrates a proposed defect detection cycle that is based on earlier testing. You can see the orange detection curve growing larger on the cheap side of things and smaller on the expensive side, giving us a pretty significant cost reduction:



Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*

This shift left relies on a more mature development practice, such as one based on the software testing pyramid—developers create a set of unit tests that cover the code reasonably well, and functional testers and API testers do as much as they can and minimize reliance on late-cycle testing, so you have just enough manual and UI tests to prove that everything is working. This way, the late cycle tests are there to prove functionality, not to find bugs. "Test early, test often" is the mantra of teams shifting left.

Some organizations stop at this point. But you get even more value when you push even further left, into coding itself. After all, this is where bugs are introduced, so let's start looking for them while development is still working.

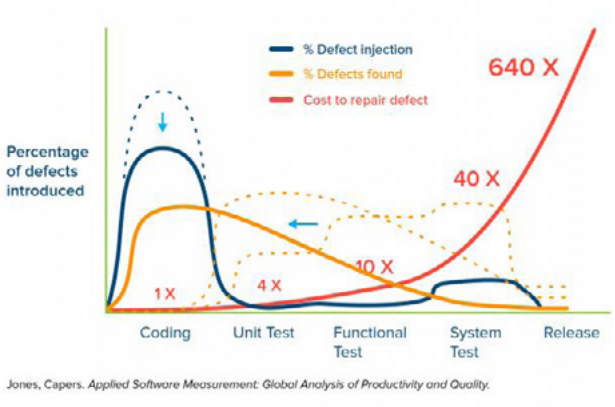
This is where we benefit from static code analysis. You can start finding bugs during the actual coding phase, when the cost of finding bugs is as low as it can get.

Finding defects before testing begins is not only the most cost-effective, but the most time-effective as well, because it doesn't leave developers with any issues trying to reproduce bugs or understand the failures. Being able to shrink a defect remediation cycle from days or weeks to hours or minutes is tremendously helpful.

Applying the Shift-Left Approach

So, how do you shift left? For the sake of brevity, the shift left testing approach breaks down into two main activities: applying development and testing best practices, and leveraging service virtualization to enable continuous testing.

Doing earlier-stage development practices, such as static code analysis and unit testing, helps you identify and prevent defects earlier in the process. It's important to remember that the goal is not to *find* bugs, but to *reduce* the number of bugs, especially those that make it into the release. Ultimately, creating fewer bugs in the first place is far more valuable than finding more bugs—and it's a lot cheaper. See the graph below, with the lovely reduced bubble on the left.



Jones, Capers. *Applied Software Measurement: Global Analysis of Productivity and Quality.*

Coding standards are the software equivalent of engineering standards, and they are key to reducing the volume of bugs (in addition to finding bugs earlier) and getting the most value out of your shift-left initiative. Coding standards help you avoid bad, dangerous, or insecure code through static code analysis.

For software security, it is especially important to harden your software. You want to build security into your code, not test it in. Coding standards let you build a more secure application from the beginning (i.e., making it secure by design), which is both a good idea and a requirement, if you're subject to regulations like GDPR.

Next, you must take the tests that were created at all stages of the development process, including the later stages, and execute them continuously moving forward. It is critical for teams that are adopting agile development practices to provide continuous feedback throughout the development process. Unit tests can easily be executed continuously, but shifting left the execution of later-stage functional tests is often difficult due to external system dependencies. This is where you can leverage service virtualization to enable continuous testing.

Service virtualization enables you to simulate dependent systems that might have limited availability, such as mainframes, third-party services, or perhaps systems that just aren't ready yet. By simulating them, you can perform functional testing without having the whole system available, and you can shift test execution left all the way to the development desktop.

In terms of performance testing, service virtualization enables you to test before everything is ready, and without having a complete lab of everything in the system. You can even run all kinds of what-if scenarios, like what if the appserver is fast and the database is slow (something difficult to make happen in the real world)? Or what if my server starts throwing funny errors, like a 500 error—how will that affect system performance? You can push the system as hard as you like and beyond, and do it as early as possible.

Similarly, you can start doing your security testing earlier. Decoupling from physical systems allows you to do something even more interesting: make the simulated systems behave in an evil fashion. Instead of just poking at your system for tainted data and distributed denial-of-service (DDoS) attacks, you can have a system flood you with packets, send malformed data, or any of the many other exploits commonly used by attackers. So not only can you test earlier, but you can also test much deeper than is possible with a test lab or production system.

Avoiding Mistakes and Traps

One danger in shifting defect detection into the coding phase is accidentally putting too much testing burden on the software developers. The important thing to remember as you look at the graph is that while the cost of defect remediation gets drastically higher as you go right, the resources on the left have possibly the highest cost of any in the software lifecycle—not to mention that you are taking them away from focusing on developing functionality.

You don't just want to find defects earlier; you want to *decrease the number of defects you're putting into the application in the first place.*

And there's another trap: If you were rewarding people for finding and fixing bugs, now they will find fewer—which is actually what you want, but only if you did reduce the number of bugs you're introducing. Measuring the number of defects that make it into the field is probably a more useful metric.

Improving Your Process and Product

By leveraging modern software testing technologies, you can achieve software that is safe, reliable, and secure. By shifting testing left along the software development lifecycle, you can reduce the cost of testing by finding bugs earlier, when it's cheaper, while also reducing the number of bugs you put into the code in the first place. Try this approach to save time, money, and headaches.

> ***By leveraging modern software testing technologies, you can achieve software that is safe, reliable, and secure.***

Getting Your New Web Test Automation Up and Running

By Daniel Garay

You're excited about implementing a new web test automation initiative, but suddenly, it hits you: Where the heck do I start? Do I just starting writing tests? What automation tool will I use? Should I set up some kind of infrastructure? Do I write tests locally on my machine and then port the environment over to some staging environment? What hurdles should I consider before I move forward? There are so many factors to think about.

Before you take a step forward, let's just take a step back and consider what, exactly, you want to accomplish.

Test automation is not a new concept. There are numerous resources out there that discuss its pros and cons, as well as many different approaches to achieve successful test automation infrastructure.

Let me walk you through a scenario I went through when I was given the responsibility of a new team and getting the web automation test infrastructure up and running. The end goal was defined, but it was completely up to me to decide what path I would take to get there.

Here, I'll discuss the hurdles, pitfalls, and successes I encountered on my journey to build a new web automation test infrastructure at my company, along with how we migrated our existing manual tests. Hopefully when it's all said and done, you can use my experience to streamline your process a bit more efficiently.

These are the milestones I defined to measure success and the path my team took to establish our new web test automation initiative.

Do Your Research

Like any other big task, you always want to start by doing your due diligence and researching all of the tools necessary.

The first issue to consider was what tool we would use. Is it scalable? How's the maintenance? Is it some-

thing that can fit into the team's existing ecosystem? What would be the learning curve for those who would maintain the automated tests? What about the existing development team's infrastructure—does it integrate with that? And what are we going to do about reporting? We had to consider the team's familiarity with existing tools within the company, as well as who would maintain the test, both short- term and long-term. We went over the same points regarding what scripting language we'd be writing in.

After considering many factors, we decided to use an API testing tool for web automation testing and an analytics platform for reporting purposes. It addressed the majority of our questions, it was easy to use, and it didn't require any prior knowledge of any programming languages.

Every company, every team, and even every individual will have a different set of questions to answer before moving forward, but the main point is to try to get as many of your questions answered up front rather than later in order to reduce the bottlenecks you may encounter up ahead.

Define the Scope and Coverage of the Tests

Next, what should you define as the scope of tests to automate? Don't be that person who tries to automate everything. These are web functio-nal tests, so you have to focus on the high-traffic areas and most commonly used parts of the application's web interface to get the most value out of your automated tests.

Because the application under test (AUT) was new to me, I had to work with both developers and QA to understand the current test cases and manual smoke test procedure. Their existing manual test cases were at a higher level, for exploratory testing, so the QA engineer couldn't just point me to obvious test cases for automation. It was a constant collaboration in every sprint—and even, at times, in our daily stand-ups—to make sure we had the coverage we wanted to automate. Once the scope was defined, we then prioritized the coverage areas so I knew exactly what to work on first.

It was a constant collaboration in every sprint—and even, at times, in our daily stand-ups—to make sure we had the coverage we wanted to automate.

This is a good rule of thumb: Even if you know the application, you should always work in collaboration with the existing team when defining the scope.

Create and Maintain Automation Tests

With the infrastructure set up and both scope and priorities defined, I could finally begin creating the automated tests. I was excited to get to write my first set of automated tests.

For this project, I started with using the browser playback feature to get a good understanding of my API testing tool, then easily migrated to editing existing browser playback tests and creating my own. I'm not embarrassed to say that my first few tests were not implemented in an ideal manner. But that's how we all learn—by trial and error.

My initial tests were very dependent on the environment, where they could only be executed in a specific sequence. There was no setup or teardown as part of my test. This obviously made it harder to maintain and troubleshoot for other team members.

We started using the tool's built-in capabilities to set up and tear down tests, reuse existing tests (such as shared tests as subsets of another test), and parameterize them so they could be portable in different environments. It was easy to integrate REST API tests

within our web automation functional tests, which made our life a heck of a lot easier to populate any prerequisite data. Single sets of tests were executed against different browsers seamlessly.

Occasionally we ran into a browser-specific issue, such as being unable to perform a click action where an element is not visible. But the tool's powerful built-in feature for different wait conditions, capability of executing arbitrary JavaScript, rich documentation, and active user forum became a savior to us.

Publish the Results Transparently

The last goal I had identified was the reporting aspect of the test results. Here, it's all about visibility. This wasn't some secret formula I concocted and wanted to keep to myself. On the contrary, I wanted everyone to be aware of the results so that the whole team was responsible for maintaining the tests.

Your automated tests are like a living organism that has to be looked after on a daily basis.

I set up test results to be reported into a reporting and analytics platform. I was able to easily create a dashboard with multiple gadgets to display the test results, so they could be displayed on the big TV screen within our development department. This way the truth was clearly visible.

The only way we would benefit from this method of reporting down the line was if we kept the test results at 100 percent passing. Otherwise, it would just be noise that no one cares about. Before I even started, I had established with development that this would be a team goal to keep tests maintained. They all agreed, and now, when I walk into the office every morning, I can easily look up and see where we are with the test results from the previous run.

Everyone Succeeds Together

Getting everything completed was never meant to be a one-person job—nor did I want it to be. It took a lot of collaboration and support from the team, including management.

One thing I learned is that you must stay on top of the tests. Keep them maintained and passing at 100 percent. Your automated tests are like a living organism that has to be looked after on a daily basis.

Do your research first before diving into the project and you'll be able to address some of the bottlenecks ahead of time, and do not hesitate to optimize your tests. All in all, this was a great learning experience for me, and I look forward to getting thrown into another team and repeating the same procedures to establish an effective, transparent, and collaborative web test automation initiative.

Insight from around the Industry

On Getting Buy-In for Test Automation

"When you're proposing automation, consider what the team needs are, then determine how you can include automation in a way that balances those needs. You're more likely to get support if the benefits are more easily felt, so ask your product owner or developers what's important to them."

~ Angela Riggs

On Effective Performance Test Reporting

"In theory, reporting performance test results should not be a problem at all. Just present the results and indicate a pass or fail. But again, we don't only want to know the result; we want to get an idea of how the result relates to the target. Crafting a report that is not overly complex but still delivers a complete picture of the status is a balancing act."

~ Michael Stahl

On the Risks of Shifting Testing Right

"To ensure testing in production works, more effort is needed than in the other gears. Testing processes have to be defined close to perfectly and all work should be automated, favoring the API layer. Whatever you forget to check will be present in production, so an area that wasn't covered in test could lead to real damage."

~ Maximilian Bauer

On the Need for Structured API Testing

"It is wise to test more important functions sooner and to spend more time on crucial aspects of a product. That calls for having a structured model and for having risks and priorities identified and monitored as the software development project unfolds."

~ Albert Gareev

On Automating Tests for Continuous Testing

"If your automated tests are meant to support continuous testing, they should be able to run continuously or on demand. Whether you're running them once a day or once a minute, your tests should be ready to go and provide the required feedback whenever you need it. Unfortunately, this is not always the case."

~ Bas Dijkstra

On the Importance of Coherent Test Documentation

"Well-documented test cases become even more important when someone other than the originator needs to use them, like a test automation engineer who might not know the product features as well as you, or another tester who might need to regression test the feature and has not worked with it much in the past. You should ease the burden for the next person down the line by providing coherent test documentation."

~ Steven Penella

Additional Resources

MORE INFORMATION FOR SOFTWARE PROFESSIONALS

BLOGS

Accelerate Software Testing by Sharing Test Assets Across Dev and Test Teams

To Make API Testing Easier, Add Machine Learning to Your AI

How to Move Beyond Record-and-Replay for Better Automated API Testing

What Is API Testing, and Are You Doing It Right?

WHITE PAPERS

Continuous Testing for DevOps: Evolving Beyond Automation

Testing Microservices

