inflectra

CONTINUOUS TESTING

Shift Left & Shift Right, Get it Right

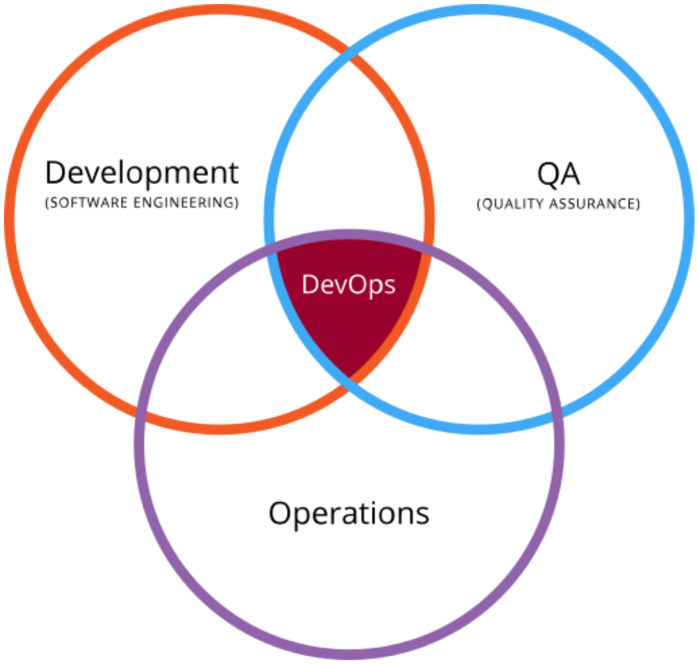
Inflectra 2021

Table Of Contents

What is DevOps Anyway?

Traditionally the words of **software development** , testing (also known as **Quality Assurance**), and the **IT infrastructure** needed to support such activities (often called Operations) were separate worlds. The developers would write code based on the requirements they were given, testers would test the features based on the same requirements (hopefully?!), and the IT staff would provide the computers, networks, and software needed by the two other groups to perform their activities. They would also be in charge of providing different environments (development, test, staging, production) that could be used by the development and testing teams.

With the rise of agile methodologies such as Scrum, XP, and Kanban , these three separate "stove-piped" worlds could no longer exist. The term Application Lifecycle Management (or ALM) was the unification of development and testing into a single process, and the logical next step has been the unification of all three disciplines into a **single integrated process** called DevOps:



(By Devops.png: Rajiv.Pant derivative work: Wylve - derived from Devops.png: , Link, originally by Gary Stevens)

The goal of DevOps is to automate as many of the steps as possible between an idea being formed and the finished code being released into production. This shrinks the time between someone coming up with the idea of a new product or business and the new product being available in the marketplace. This means that concerns such as provisioning servers and other infrastructure as well as scaling a successful application need to be as automated and seamless as the software development build process.

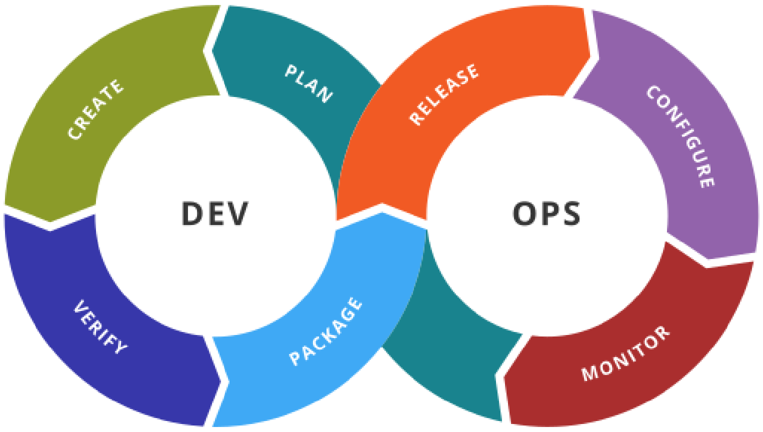
What are the Elements of DevOps?

There are many different ways of categorizing tools that improve DevOps, however in general, it is recognized that the following seven areas need to be considered when looking for different tools that make up what is usually known as the **DevOps Toolchain**:

- **Plan -** Plan is composed of two things: "define" and "plan". This activity refers to the business value and application requirements requirements
- **Code / Build** - code design and development tools, source code management tools, continuous integration / build servers
- **Test / Verify** – continuous testing tools and processes that provide feedback on business risks
- **Package** – artifact repository, application pre-deployment staging
- **Release** – change management, release approvals, release automation
- **Configure** – infrastructure configuration and management, Infrastructure as Code tools
- **Monitor** – applications performance monitoring, end-user experience

Now the relative importance of each of these seven items will vary. It will depend on:

- the type of application (web-based, mobile, legacy desktop, micro-services, AI, data warehouses)
- the methodology being used (continuous build and integration usually require an agile methodology)
- whether the applications are in MVP, early adoption, mainstream adoption, or support
- maintenance mode.



However, one of the criticisms of the term DevOps is that on the surface, it doesn't contain a reference to testing or quality. This has led to increasingly complicated portmanteau such as DevTestOps, DevSecOps, DevTestSecPerfOps, each one trying to add different types of testing or quality to the phrase. However, in reality when you actually start to look at each of the DevOps elements above, you quickly realize that in fact, testing is everywhere.

We call this idea Continuous Testing, and changes the traditional notions of what testing is and when testing should be done!

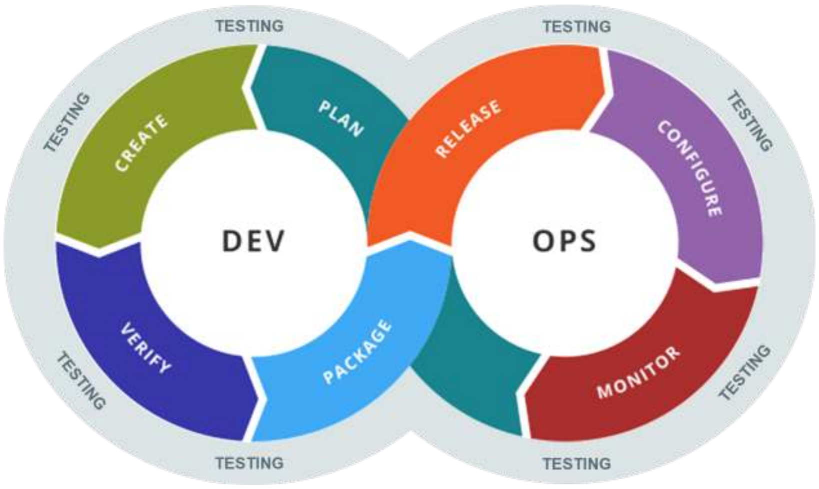
What is Continuous Testing?

Continuous testing is the process of executing tests as part of the software delivery pipeline to obtain immediate feedback on the business risks associated with a software release candidate.

1. Originally limited to automated tests in the CI portion
2. Originally limited to automated tests in the CI portion

For Continuous testing, the scope of testing extends from validating bottom-up requirements or user stories to assessing the system requirements associated with overarching business goals, and goes all the way to monitoring the system in production to find problems that need to be corrected.

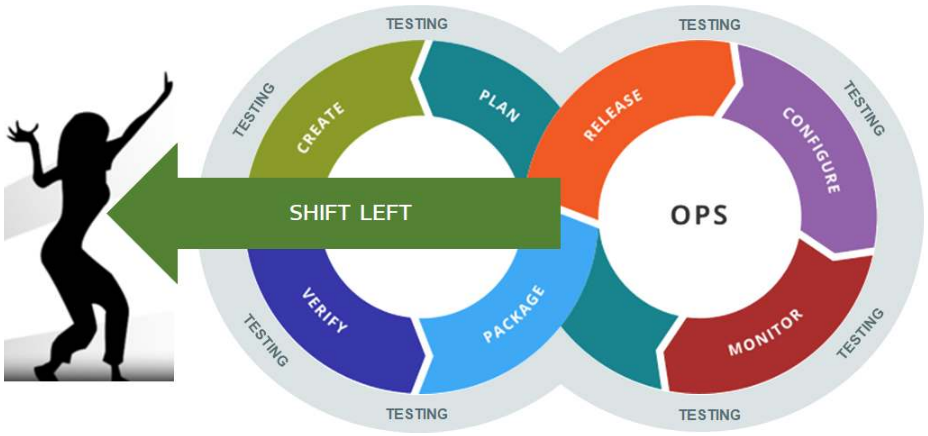
Adding Continuous Testing to our DevOps diagram gives us this new way of thinking about software testing and how it first into the DevOps lifecycle:



As you can see Testing is not limited to one specific phase of the pipeline, but is an integral discipline throughout all the DevOps stages.

Shift Left: Testing Early, in the CI Pipeline

The first change in our thinking occurs when we look at the "left-side" of the DevOps diagram. The concept of "Shift Left" testing means to increase the amount of testing you're doing on the left-hand side of the diagram. In this section, we will discuss how you can improve the testing being done during development in CI by adding early testing for functionality, usability, performance, security, and accessibility.



Shifting Left is about removing downstream blockers and finding and fixing defects closer to where they are introduced. When you are looking at ways to add testing to your CI pipeline, you should focus on the following general aspects:

- Rapid, automated testing to prevent previously known issues from reappearing
- Identifying areas of risk earlier to understand where to focus testing later
- Do just enough of each type of testing early in the pipeline to determine if further testing is justified
- Early exploration of features to understand business issues/risks

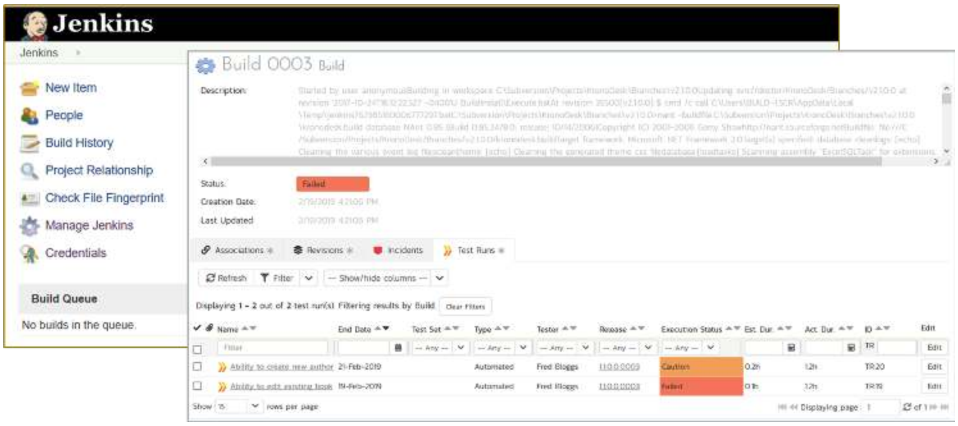
The areas of testing that should be covered early in the CI pipeline typically include:

- Functionality
- Performance
- Security
- Integration
- Infrastructure
- Usability

In the following sections, we will cover these different areas in more detail.

Automated Functional Checking

In your DevOps Continuous Integration (CI) pipeline, we recommend that you have multiple layers of automated tests (often called checks) that verify that the application is working as expected from a functionality perspective. These are often called 'checks' rather than tests at this stage because they are mainly checking that the system behaves the way it is expected, vs. testing for new behaviors.



Description: Started by user anonymous Building in workspace C:\Subversion\Projects\[illegible]

Status: Failed

Creation Date: 2/19/2019 4:21:05 PM

Last Updated: 2/19/2019 4:21:05 PM

Displaying 1 - 2 out of 2 test run(s). Filtering results by Build

Name	End Date	Test Set	Type	Tester	Release	Execution Status	Est. Dur.	Act. Dur.	ID	Edit
Ability to create new author	21-Feb-2019		Automated	Fred Bloggs	1.1.0.0.0003	Caution	0.2h	1.2h	TR:20	Edit
Ability to edit existing book	19-Feb-2019		Automated	Fred Bloggs	1.1.0.0.0003	Failed	0.1h	1.2h	TR:19	Edit

Show 15 rows per page — Displaying page 1 of 1

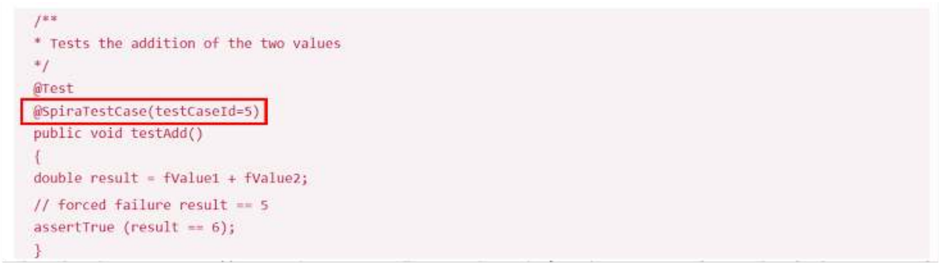
The different types of automated tests include:

- Unite Tests
- Api & Integration Tests
- User Interface Testing

Unit Tests

These are the initial building blocks of your automated testing infrastructure. Suppose you are genuinely following an agile methodology. In that case, you should be using Test Driven Development (TDD) to ensure that all your code has unit tests written by developers **before** it is even written. However, even in cases where this is not feasible, we recommend ensuring that all public methods and functions have at least one coded unit test associated with them.

We recommend that you connect the different unit tests to a test management tool such as SpiraTest to make sure you get accurate real-time reporting of the test coverage with every CI build:



```
/**
* Tests the addition of the two values
*/
@Test
@SpiraTestCase(testCaseId=5)
public void testAdd()
{
double result = fValue1 + fValue2;
// forced failure result == 5
assertTrue (result == 6);
}
```

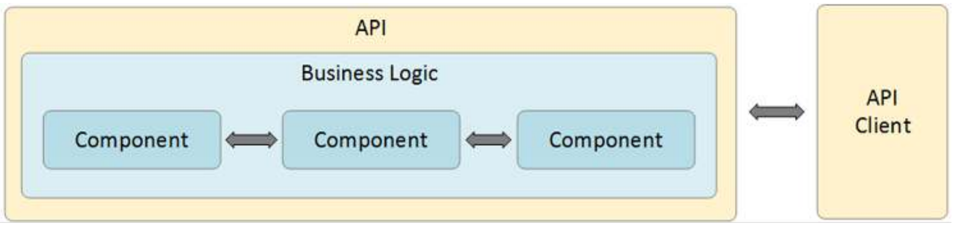
There are different xUnit testing frameworks for different languages, so make sure you choose the appropriate framework for your platform (NUnit, jUnit, PyTest, etc.) and that your developers are managing the testing code in the same SCM tool (e.g., Git) as your software code. The same best practices for having frequent code reviews and adhering to documented coding standards apply to your test code. As many leaders in the testing community advocate – “automated testing code” is “code” and should be treated with the same provenance and importance as the application code itself.

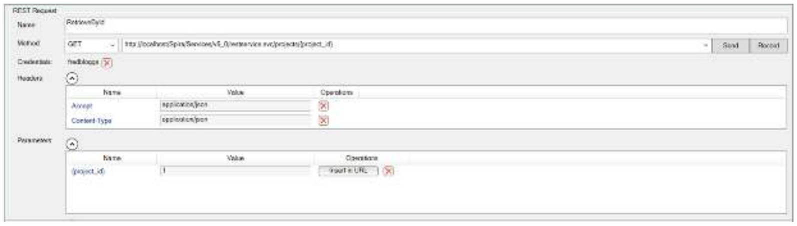
One of the dangers of unit testing code written by the developers is that they will only test the ‘happy path’ flows in the program, so we recommend that the primary developers write the initial unit test, which acts as a source of documentation for the code. Then have other developers / QA resources review the code and expand the coverage of the code in the unit test. Some standard techniques for making sure your unit tests have good coverage include:

- Fuzz Testing
- Mutation Testing

API & Integration Tests

Since APIs are crucial interfaces that different parts of your system may rely on to communicate, and also are often used heavily by external clients, having a repeatable, automated robust set of API tests in your CI pipeline makes a lot of sense. Such API tests need to cover all the various versions and formats (REST, SOAP, ODATA, GraphQL, etc.) that are supported and be fast enough to be executed in every CI build.





REST Request

Name	RetrieveById
Method	GET http://localhost/Spira/Services/v5_0/restservice.svc/projects/{project_id}
Credentials	fredbloggs

Headers

Name	Value	Operations
Accept	application/json	
Content-Type	application/json	

Parameters

Name	Value	Operations
{project_id}	1	Insert in URL

In keeping with the philosophy of continuous testing, if you have any slower, long-running API tests, they should be kept outside the CI pipeline and run later in the DevOps cycle, potentially used only when one of the other earlier API tests indicates an area of risk to investigate.

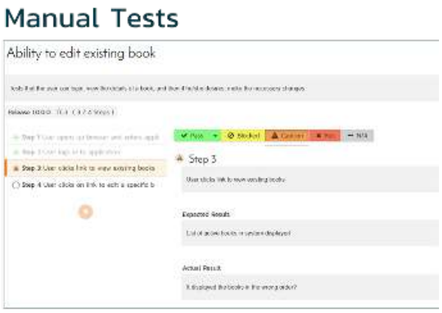
In all cases, have the API tests integrated with your test management tool (such as SpiraTest) to ensure that any failures in the test are reported back against the appropriate release and requirements. This allows the impact of the failures to be visualized in real-time.

User Interface (UI) Testing

Typically, UI testing is something that is thought of as being an activity that should happen later in the DevOps lifecycle because the risk of application changes invalidating the testing (or breaking the tests in the case of automated UI tests) is greater at this stage.

Following the philosophy of Continuous Testing, we'd recommend that you conduct **some UI testing** in the early stages of the DevOps lifecycle, using that information to prioritize and focus later, deeper testing activities that happen later on.

For automated tests, we recommend using a tool such as Rapise that employs a codeless, model-based approach that makes writing initial tests and changing tests later easier. That way, when changes happen, you can adapt your tests more easily. In addition, for the "shift left" activities, we'd recommend writing automated tests that focus on simple high-level, repeatable tests that save time (e.g., logging in with every user, role, permission, and web browser) over deep, complex scenarios (creating a purchase order and receiving goods) that should be done later, when the application is more stable.



Ability to edit existing book

[illegible]

Step 3 User clicks link to view existing books

Step 4 User clicks on link to edit a specific b

Step 3

User clicks link to view existing books

Expected Result

List of active books in system displayed

Actual Result

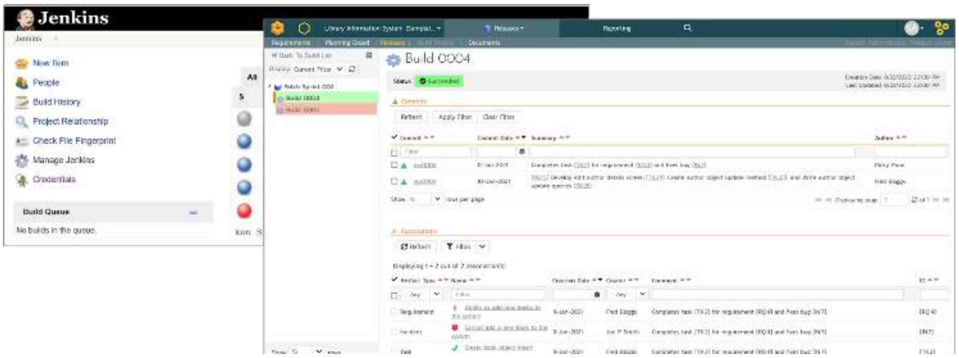
It displayed the books in the wrong order?

For manual testing, we'd recommend using rapid, exploratory techniques such as **exploratory testing** , **session-based testing** , where testers can follow high-level project charters to uncover potential issues and areas of risk. Later on, these insights can be used to focus deeper, more structured scenario manual tests on the areas that were uncovered.

For example, when doing exploratory testing, it was identified that the permissions checks seemed to be unreliable in some of the screens. Later on, in the DevOps cycle, these early insights could be used to have the testing team perform some more exhaustive testing on the authorization system. This early, rapid feedback lets you find the Rumsfeldian "unknown unknowns" that will otherwise come back to bite you later. We will cover this more later in the section on Exploratory Testing.

Continuous Integration (CI)

One of the assumptions we have made in the effort to “Shift Left” our testing is that we’ll be using CI automation tools such as Jenkins, TeamCity, or other DevOps pipeline platforms.

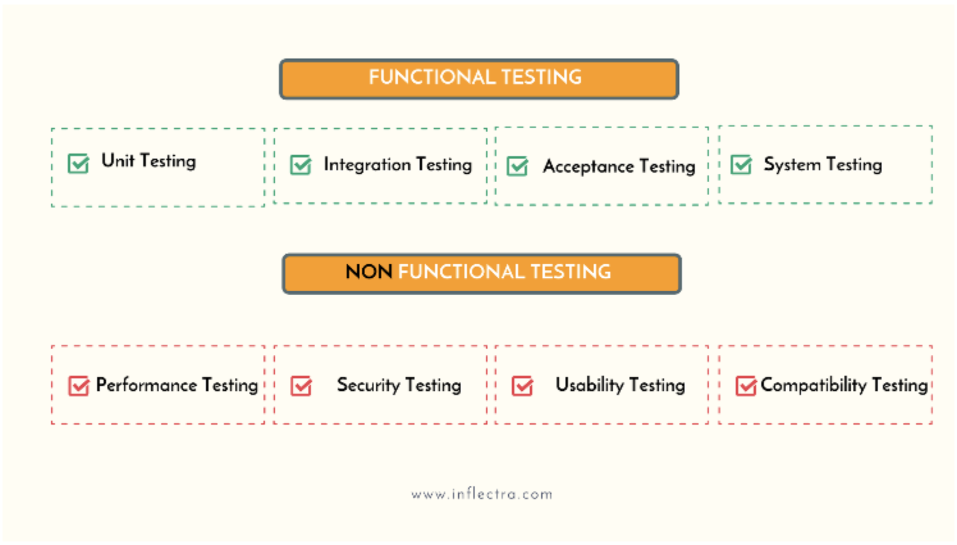


We recommend that you use the CI pipelines to orchestrate the various automated checks and have the CI pipeline report back into the same enterprise test management tool (SpiraTest) as the various testing frameworks that have been integrated.

This approach of early feedback to inform later testing also applies to the realm of “non-functional” tests that traditionally have been done later in the lifecycle of the release. In adopting Continuous Testing practices, we need to understand how we can also “Shift Left” these activities as well.

Non-Functional Testing

Unfortunately, although great strides have been made with agile and DevOps practices to move testing further "to the left", with unit tests and automated functional checks, the area of non-functional tests has tended to remain an activity done at the last minute, close to release.



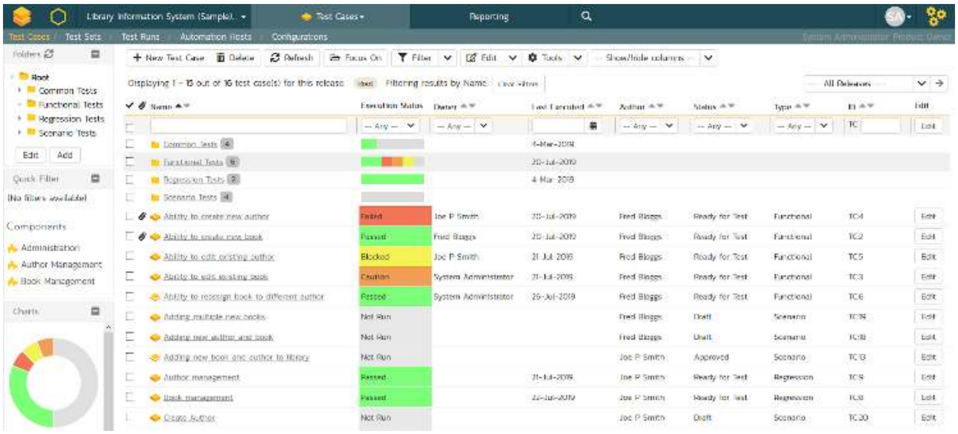
With the adoption of continuous testing, we recommend the following approach be taken:

- Don't leave these to the end of the process, before deployment
- Do just enough of each type of testing early in the pipeline to determine if further testing will be needed later.

Testing Early to Identity Risk

The key to being successful with non-functional testing is to avoid the two extremes: doing none of these tests until the system is developed and ready for deployment and doing all of these tests so that you don't have enough time in the various sprints.

It is probably best to perform lightweight, rapid testing in each of these areas so that you can determine early which ones will need deeper testing later.



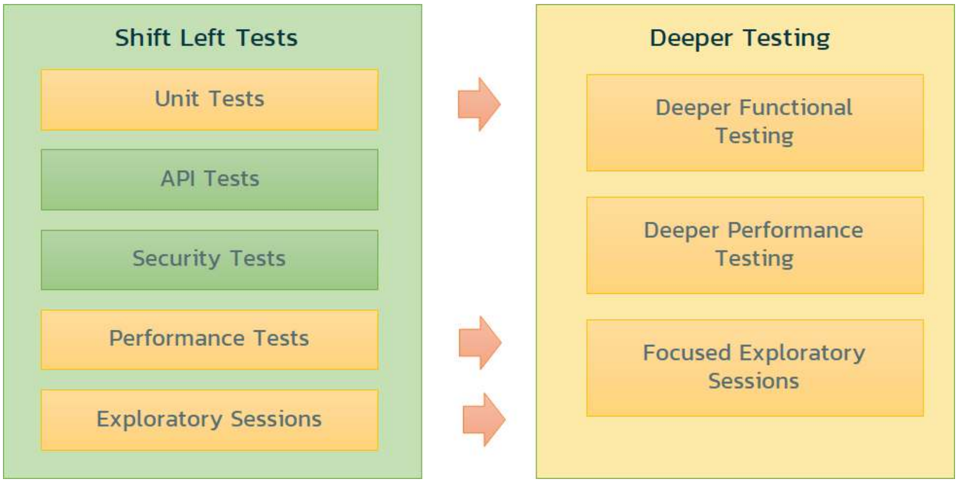
Displaying 1 - 15 out of 16 test case(s) for this release. Filtering results by Name. Clear Filters

Name	Execution Status	Owner	Last Executed	Author	Status	Type	ID	Edit
Common Tests 4			4-Mar-2019					
Functional Tests 6			20-Jul-2019					
Regression Tests 2			4-Mar-2019					
Scenario Tests 4								
Ability to create new author	Failed	Joe P Smith	20-Jul-2019	Fred Bloggs	Ready for Test	Functional	TC4	Edit
Ability to create new book	Passed	Fred Bloggs	20-Jul-2019	Fred Bloggs	Ready for Test	Functional	TC2	Edit
Ability to edit existing author	Blocked	Joe P Smith	21-Jul-2019	Fred Bloggs	Ready for Test	Functional	TC5	Edit
Ability to edit existing book	Caution	System Administrator	21-Jul-2019	Fred Bloggs	Ready for Test	Functional	TC3	Edit
Ability to reassign book to different author	Passed	System Administrator	25-Jul-2019	Fred Bloggs	Ready for Test	Functional	TC6	Edit
Adding multiple new books	Not Run			Fred Bloggs	Draft	Scenario	TC19	Edit
Adding new author and book	Not Run			Fred Bloggs	Draft	Scenario	TC18	Edit
Adding new book and author to library	Not Run			Joe P Smith	Approved	Scenario	TC13	Edit
Author management	Passed		21-Jul-2019	Joe P Smith	Ready for Test	Regression	TC9	Edit
Book management	Passed		22-Jul-2019	Joe P Smith	Ready for Test	Regression	TC10	Edit
Create Author	Not Run			Joe P Smith	Draft	Scenario	TC20	Edit

For example, if your early security scans determine that no material changes have been made to the system and the underlying libraries have no new vulnerabilities, maybe you can avoid a deeper penetration test later.

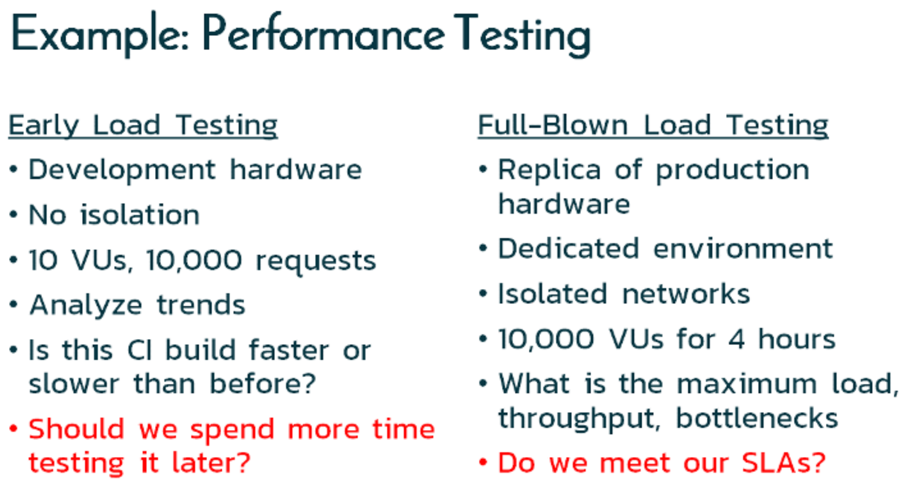
If you look at this approach as a general guideline, use each of your "shift left" tests to identify the areas of most risk, and schedule deeper testing (for later in the process) for only these areas:

Continuous Testing - Shift Left, Shift Right



Performance Testing

As another example, if we consider performance / load testing, we can compare the difference between what we might try and do in our "shift left" tests vs. what we'll do later on:



Example: Performance Testing

Early Load Testing
- Development hardware
- No isolation
- 10 VUs, 10,000 requests
- Analyze trends
- Is this CI build faster or slower than before?
- Should we spend more time testing it later?

Full-Blown Load Testing
- Replica of production hardware
- Dedicated environment
- Isolated networks
- 10,000 VUs for 4 hours
- What is the maximum load, throughput, bottlenecks
- Do we meet our SLAs?

For the early tests, we may not even need to use a traditional load testing tool, maybe just recording the build durations in our test management tool will be sufficient. If we see the build time on the same infrastructure take > 5% longer than the previous version, it's time to do a load test later on with a more powerful tool such as NeoLoad or JMeter.

Security Testing

We have already mentioned this earlier, but in the same vein, for our initial early tests, we can run some simple, fast tools to check common issues:

- Dependency Checking Tools
- Dependency Tracking Tools
- Library managers (Nuget, Maven, NPM, etc.)
- Code Analysis (SonarQube, PyLint, Visual Studio, etc.)
- Vulnerability Scanning (OWASP ZAP)

These will find common issues like out-of-date libraries, OWASP top 10 vulnerabilities such as XSS, SQL Injection, CSRF, etc., that can be easily addressed during development. These tools are also relatively fast and so can be incorporated into your CI pipeline.

In addition, they will also indicate where there are weaknesses that you can test for in your post-development testing:

Use a PCI-DSS scanning tool to find any vulnerabilities once deployed on the production (or replica) environment.

Schedule a manual penetration test (pen-test) using the results from the earlier automated tests to focus the pen-test on specific areas of weakness.

Usability Testing

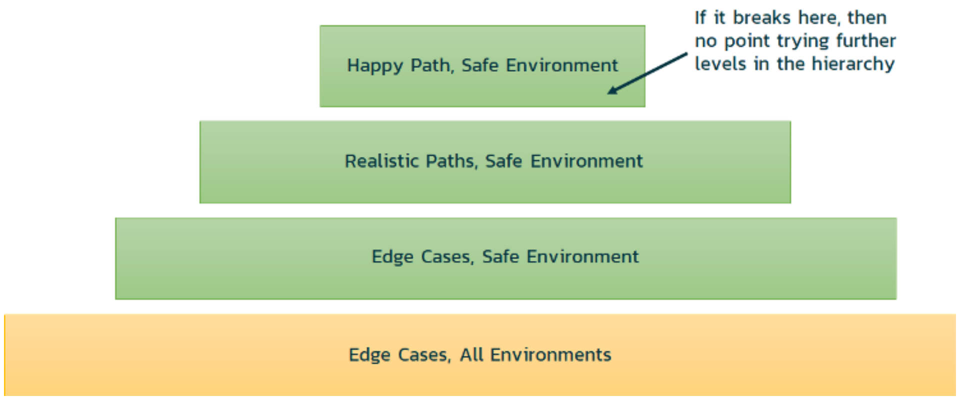
Normally usability testing is something that is done towards the end of development once the UI is stable. However, we'd recommend conducting some usability testing on early builds, even if some of the data and screens have to be faked. That way, you can get early feedback on core UX elements such as navigation, screen layouts, etc., while there is still time to make changes without a lot of rework.

In accordance with the principles of continuous testing, this early UX feedback doesn't mean you don't do additional UX testing later on; it just means you can get some feedback earlier (before people get too attached to the design!).

Compatibility Testing

Another area of testing that can benefit from continuous testing is the testing across different environments.

For example, it is tempting to take the latest CI build and try running it on the worst environment (IE11 for example) with an eye to breaking it by trying lots of crazy tests. However, if there are major bugs, it will be hard for the developers to distinguish if it's the latest change or the environment that caused the problem.


If it breaks here, then no point trying further levels in the hierarchy
Happy Path, Safe Environment
Realistic Paths, Safe Environment
Edge Cases, Safe Environment
Edge Cases, All Environments

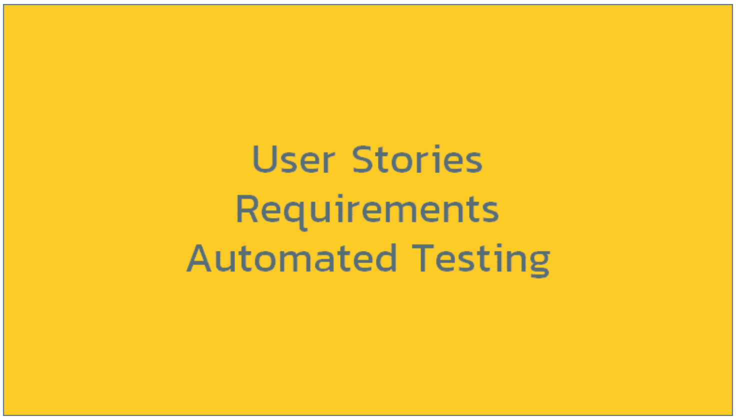
So, we would recommend that you focus initially on testing the simplest use cases (aka the "happy path") and use the same environment that developers were using (e.g., the Google Chrome web browser). That way any bugs in functionality are truly due to code changes and not the environment.

You should also do some limited exploring using the other environments, perhaps try the “worst” environment (e.g., IE11) and see if there is anything different. In those cases, don’t log bugs, simply add them to the test plan for the later, exhaustive compatibility testing that you will be doing.

Exploratory Testing

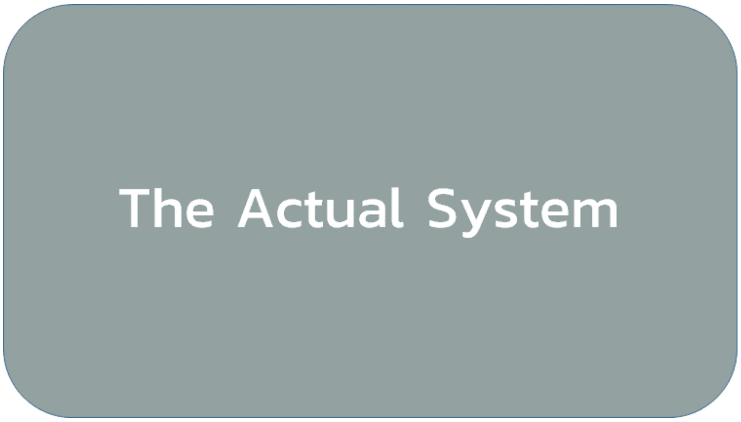
It is an often under-appreciated fact that great testers find issues that no one knows about, the so-called Rumsfeldian (named after Donald Rumsfeld, Secretary of Defense) “unknown unknowns”. So even though you may have a great library of automated checks and manual test scenarios in your test plan, you need to always allow time for exploratory and other forms of unstructured testing.

This is because, when you are defining a system, you have the “box” of what are the understood and defined features:

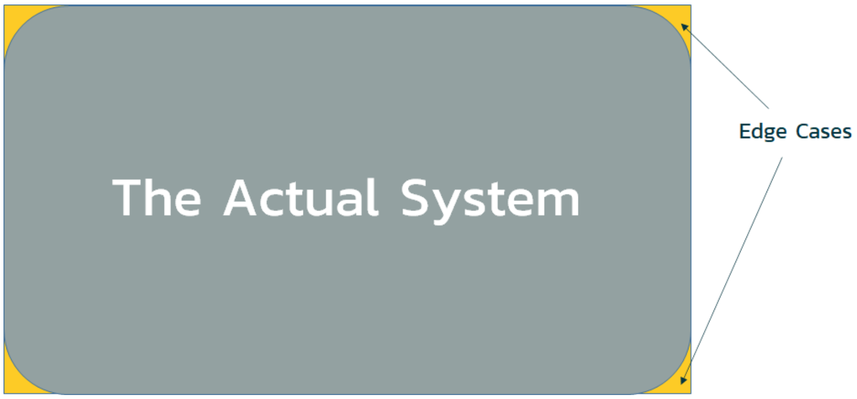


However, the actual system will have many small design choices, coding decisions, and other unappreciated changes that were made along the way.

Agile methodologies avoid the use of long, prescriptive, “set in stone” requirements so that the team has the flexibility to make smart decisions in developing the system as they go along. That means the actual system looks like this:

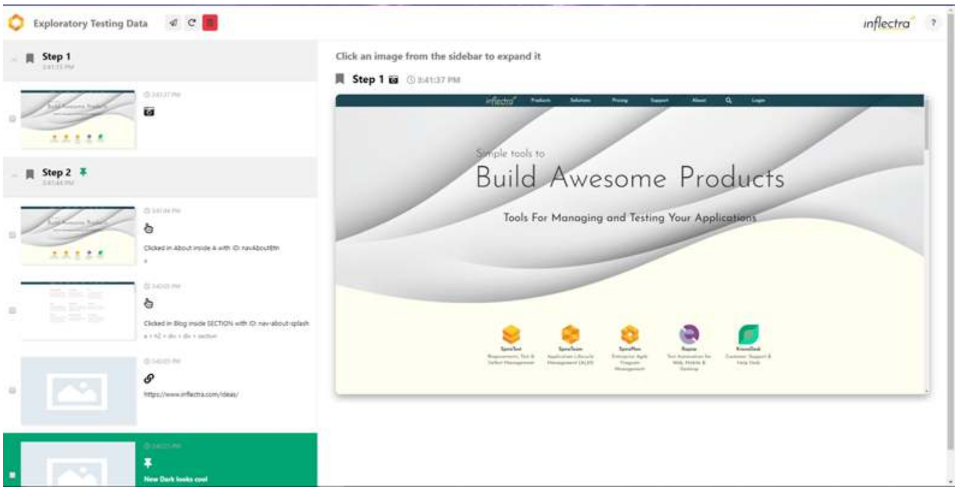


Conceptually that means the actual system will not be a 100% reproduction of what was designed, so inevitably, even if you have the best test plan in the world, with 100% test coverage of all your requirements, there will always be ‘edge cases’ that need to be discovered and tested.



One of the benefits of exploratory testing is that it maximizes the cognitive abilities and human testers instead of treating them like machines (following a script). Exploratory testing is a great way of finding those edge cases, especially early in the sprints when functionality is being created/updated iteratively.

Exploratory testing should be done using a test management tool that has built-in support for exploratory testing, where you can document your findings in real-time, in conjunction with a capture tool that captures your sessions automatically.


Exploratory Testing Data
inflectra
Step 1
Step 2
Click an image from the sidebar to expand it
Step 1
Simple tools to
Build Awesome Products
Tools For Managing and Testing Your Applications

Clicked in About inside A with ID navAboutBtn

Clicked in Blog inside SECTION with ID nav-about-splash

https://www.inflectra.com/ideas/

New Dark looks cool

The one recommendation we have for these early testing activities is to avoid labeling the findings ‘bugs’; typically, these early explorations may find observations that could be beneficial as well as harmful. You are looking for areas of risk and providing feedback to the development team. Much of what you may find may be known or intended, or if not, may be “better than what was intended.” We recommend you use project tasks to track these rather than bugs or issues.

Tools Recap

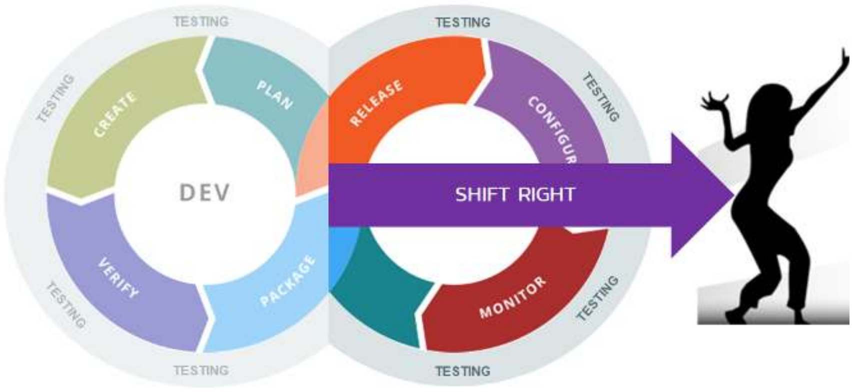
We have mentioned some tools in the previous sections, but just to recap, here are some ideas:

- Dependency checkers – DependencyCheck
- Static Code Analysis – Parasoft, SonarQube
- Unit Tests – xUnit (jUnit, Nunit, PyTest, etc.)
- Performance Tests – JMeter, Gatling, NeoLoad
- API Testing – Postman, Rapise, SoapUI
- Security Testing – OWASP ZAP
- UI Testing – Rapise, Selenium
- Exploratory Testing – SpiraCapture

Shift Right: Monitoring in Production, after CD

In continuous testing, “Shifting Right” refers to continuing to test the software after it is released into production. Also known as “testing in production,” the process can refer to either testing in actual production or a post-production environment.

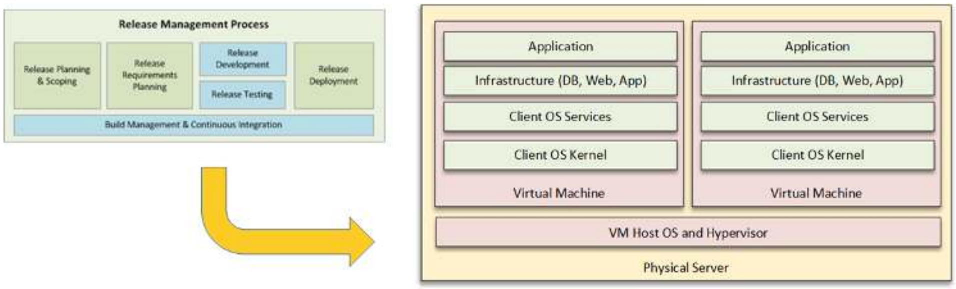
The key aspect is that Shifting Right is all about leveraging the insights and usage of real customers and access to production data to test effectively and support feedback loops.



Shift Left testing relies on finding problems early with fast, repetitive testing that uncovers areas of risk and tries to prevent problems from occurring. Shift Right testing, on the other hand, involves monitoring user behavior, business metrics, performance and security metrics, and even deliberate failure experiments (chaos testing) to see how resilient the system is to failures.

Continuous Deployment (CD)

In agile projects, Continuous Integration (CI) means that you are continuously integrating and testing the code so that it could be theoretically be deployed into production at any time. However, Continuous Deployment (CD) takes this one step further and automates the process of actually deploying the code into production in an automated fashion. Usually, with CD environments, because you are reducing or eliminating the manual "go/no-go" decision process, there should be a way to seamlessly roll back the update just as easily.


Release Management Process
Release Planning & Scoping
Release Requirements Planning
Release Development
Release Testing
Release Deployment
Build Management & Continuous Integration
Application
Infrastructure (DB, Web, App)
Client OS Services
Client OS Kernel
Virtual Machine
Application
Infrastructure (DB, Web, App)
Client OS Services
Client OS Kernel
Virtual Machine
VM Host OS and Hypervisor
Physical Server

With CD, a lot depends on the type of application being deployed (SaaS, PaaS, on-premise, etc.) and the infrastructure in place (physical, virtual, containerized, cloud, etc.), however in general, CD is all about automating the process from being release-ready (CI), tests passed, to being released into production, with post-release checks in place.

Post Release Testing

Once you have released the system into production, testing does not end. You will need to perform a variety of post-release tests:

- Post-Release Automated Checks
- Usability & Experience Testing
- A/B and Canary Testing

Post-Release Automated Checks

Once the code changes have been deployed into production, you should automatically run a set of non-destructive, automated functional tests that test that the code released into production behaves functionally the same as what was tested in development. This may sound obvious, but it is amazing how often different configurations used in production (higher security, higher availability, etc.) can cause the system to behave differently.

You can usually take your existing automated unit tests and run a subset of them against production. You have to be careful because you are dealing with live customer data, so some automated tests will have to be modified or excluded because of how they would change live data.

Usability & Experience Testing

Shift-right testing also is concerned with the business value and usability of the updated version, i.e., do the new user stories make the system better for the users. Even though the system is now live, you can still perform usability and experience testing to make sure that users can understand any UI changes and that the system has not degraded the experience. Two techniques for measuring this are A/B testing and Canary Testing.

A/B and Canary Testing

In digital marketing, A/B testing is a widespread technique that is now being “borrowed” in the world of software development and DevOps. You divide up your customers into two groups (“A” and “B”). Half of the users get the old version of the system (“A”), and the other half get the new version of the system (“B”). This means your CD system needs to be able to deploy to specific regions or instances at different cadences. You can then survey or measure users’ interactions with both versions. For example, in an e-commerce system, you could measure what % of new prospects purchase a product using the two versions of the system.

A variation of “A/B testing is “Canary Testing” so-called because it mirrors the idea of the canary in the coal mine that alerts miners to the first presence of the poison gas. With Canary Testing, you push the latest update to just a small community of users that you know will react quickly to any changes. For example, you might have an “early adopter” update channel to which motivated users can subscribe.

Monitoring

In addition to post-release testing (which tends to happen immediately after pushing code into production), continuous monitoring is the other aspect of "Shift Right" testing that needs to be put in place, both system, and business metric monitoring.

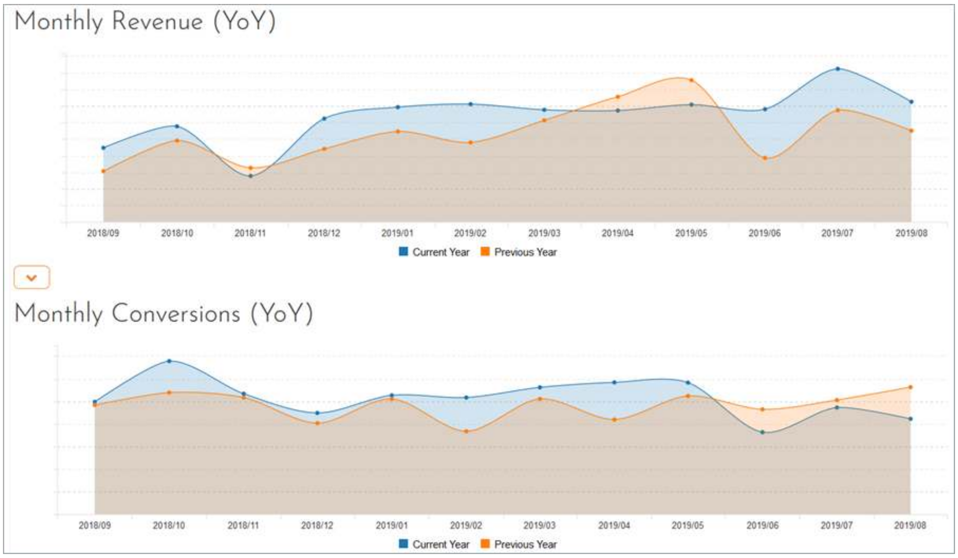
System Monitoring

Many of the non-functional testing tools used in Shift Left testing (load, performance, security, etc.) have similar tools that are used for monitoring (performance monitoring, security monitoring, etc.) the same metrics in production.

So once the code has been pushed into production, you should ideally have a system monitoring dashboard that lets you see the key metrics (security, performance, etc.) from the live system in real-time. There are subtle differences; for example, performance monitoring focuses on checking the system response time, CPU utilization, etc., under the current load, vs. generating load to see when problems occur.

Business Monitoring

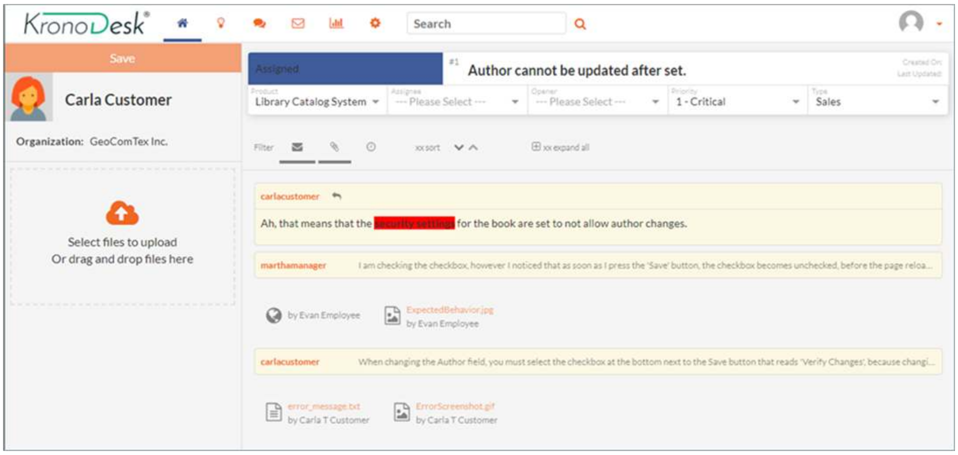
In addition to system metrics, you can learn a lot about a system by monitoring the various business metrics and correlating them together. For example, if you have a website that tracks signups (conversions) and actual sales (revenue/orders), you can measure the ratio of the two metrics (sales conversion rate), and if you were to get a sudden drop in this rate after a code push, you would quickly know to check the user flow to make sure a bug has not been introduced that prevents a user completing a sale.



In this way, business metrics can be a helpful indicator of a problem that system monitoring may have missed! In addition, monitoring different business metrics can be used to “test” the business value of system changes. For example, a change that makes the application faster but generates less valuable orders might be desirable for users, but not for the company!

User Monitoring & Feedback

Another important source of feedback on a system release is users. Therefore, your customer support system (help desk, service management solution (ITSM), etc.) is an important source of feedback data. If you have accidentally introduced a new defect or functionality regression, your users will let you know quickly.



Carla Customer

Organization: GeoComTex Inc.

#1 Author cannot be updated after set.

Product	Assignee	Owner	Priority	Type
Library Catalog System	--- Please Select ---	--- Please Select ---	1 - Critical	Sales

carlacustomer

Ah, that means that the security settings for the book are set to not allow author changes.

marthamanager I am checking the checkbox, however I noticed that as soon as I press the 'Save' button, the checkbox becomes unchecked, before the page reloa...

by Evan Employee

ExpectedBehavior.jpg by Evan Employee

carlacustomer When changing the Author field, you must select the checkbox at the bottom next to the Save button that reads 'Verify Changes', because changi...

error_message.txt by Carla T Customer

ErrorScreenshot.gif by Carla T Customer

If you start seeing a trend from trusted users raising support tickets about a specific issue, you can immediately have a 'tiger team' do an investigation and get a hotfix into the CI/CD pipeline, or if serious enough, you can potentially rollback the change before it affects other users.

Chaos Engineering & Self-Healing

Finally, one relatively innovative approach taken by some companies (most notably Netflix with its Chaos Monkey) is to have an automated agent that deliberately (and randomly) terminates instances in production to ensure that engineers implement their services to be resilient to instance failures.

If you decide to implement Chaos Engineering, you will need to ensure you have developed real-time monitoring and "self-healing" services that detect the failures and can have the system adapt to them in real time. For example, you could have the infrastructure spin up additional resources, route traffic around the degraded instances, and take other protective and reactive measures.

Tools Recap

We have mentioned some tools in the previous sections, but just to recap, here are some ideas:

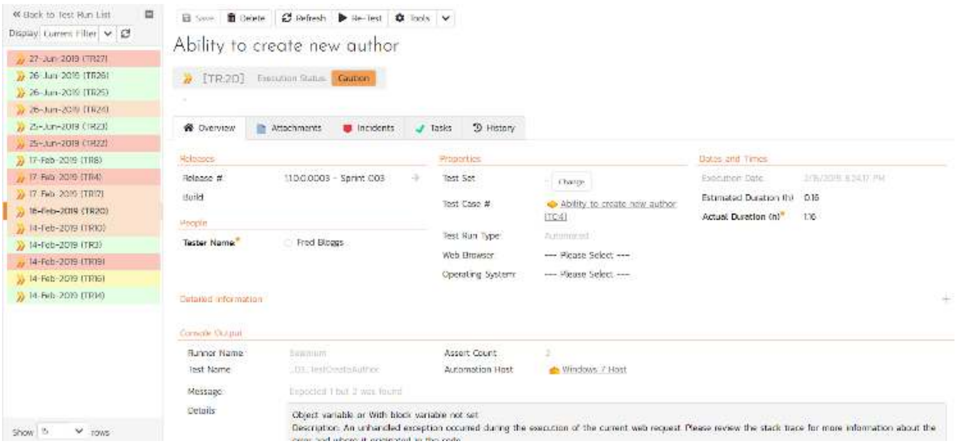
- **System Monitoring**
- Performance – Dynatrace
- Security - Cloudwatch
- Customer Help Desk – ZenDesk, KronoDesk, Remedy, etc.

- Customer Behavior – Google Analytics, Splunk, Matomo

- **Business Monitoring**
- Traditional BI Platforms – PowerBI, Splunk, Cognos, Domo, etc.
- AI-Driven Data Analytics – Qlik, Tellius, etc.
- Data Visualization – Tableau, etc.
- Customer Satisfaction – Delighted, Promoter, AskNicely, etc.

Assessing Risk and Deciding Whether to Go Live

The final part of Continuous Testing is the part in the middle: in between rapid testing that identifies risks and uncovers areas to test further and released into production is the part where you evaluate all of the test results and decide whether to go live.



27-Jun-2019 [TR27]
26-Jun-2019 [TR26]
26-Jun-2019 [TR25]
26-Jun-2019 [TR24]
25-Jun-2019 [TR23]
25-Jun-2019 [TR22]
17-Feb-2019 [TR8]
17-Feb-2019 [TR4]
17-Feb-2019 [TR17]
16-Feb-2019 [TR20]
14-Feb-2019 [TR10]
14-Feb-2019 [TR3]
14-Feb-2019 [TR19]
14-Feb-2019 [TR16]
14-Feb-2019 [TR14]

Ability to create new author

[TR:20] Execution Status: Caution

Releases

Release #: 1.1.0.0.0003 - Sprint 003
Build

People

Tester Name*: Fred Bloggs

Properties

Test Set
Test Case #: Ability to create new author [TC:4]
Test Run Type: Automated
Web Browser: --- Please Select ---
Operating System: --- Please Select ---

Dates and Times

Execution Date: 2/15/2019 8:24:17 PM
Estimated Duration (h): 0.15
Actual Duration (h)*: 1.10

Detailed Information

Console Output

Runner Name: Selenium
Test Name: _03_testCreateAuthor
Message: Expected 1 but 2 was found
Details: Object variable or With block variable not set
Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the

Assert Count: 2
Automation Host: Windows 7 Host

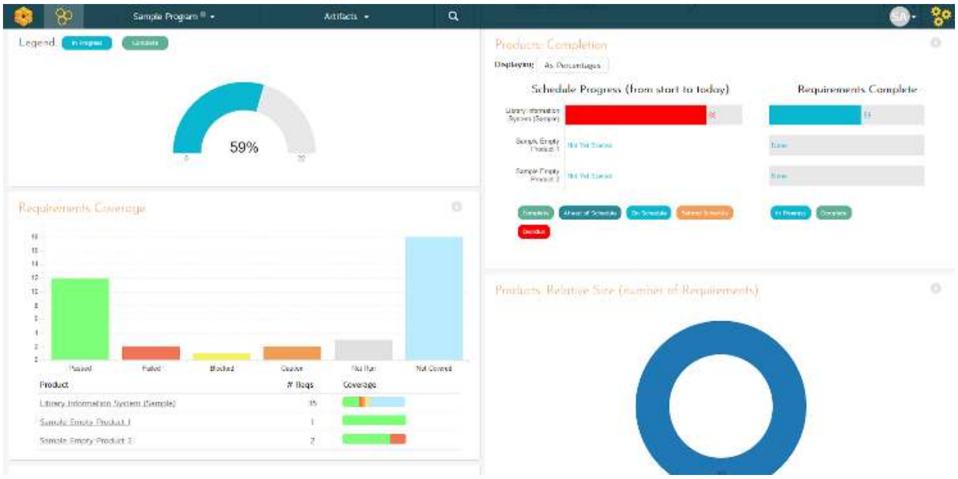
In a true Continuous Deployment (CD) system, you may have automated the part where the DevOps infrastructure pushes code into production. Still, even so, you will need to have codified the rules to determine when the system is “stable enough” for release. In a traditional “manual” release process, you have a human manager reviewing the testing information and deciding to go live.

In either case, you are going to have to answer the following two questions:

- How Do We Know if We Can Release into Production?
- How Do We Know We Have Tested the Right Areas?

We Don't Have Enough Time to Test Everything!

The reality is that whether manual or automated, you will not have enough time in an agile, CI/CD environment to retest every part of the system 100% with all your tests. If you use a combination of Shift-Left techniques to know where to focus, deeper testing in the middle to explore those areas, and Shift-Right to be able to identify and recover quickly from a problem, then the key is to understand at what point do you have enough data to determine that the risk from releasing is less than the risk from not releasing.



That does not mean there is zero risk from a bug, just that the risk of a bug occurring is less critical than the risk that existing bugs remain unfixed, and that the business risk of delaying that feature is greater than the business risk of an issue that impacts users.

Conclusion

In conclusion, when you are looking to incorporate Continuous Testing practices into your DevOps process, these are the three areas you will need to prioritize and plan for:

- Integrate Shift-Left techniques into your CI Pipeline. Address risk areas identified early to focus deeper testing
- Add Shift-Right techniques into your post-deployment infrastructure so that testing doesn't end after live
- Have a risk-based approach to deciding when to release functionality into production

by Adam Sandman,
The founder and CEO,
Inflectra
adam.sandman@inflectra.com

Founded in 2006, **Inflectra** is a market leader in software test management, test automation, application lifecycle management, and enterprise portfolio management space with its **SpiraPlan, Rapise and KronoDesk** Platforms. The company is headquartered in the USA but has offices in over 10 countries. Known globally for its legendary customer support, Inflectra makes turn-key solutions that address many challenges in software testing and QA, test automation, and product lifecycle management. Its methodology agnostic software tools are used in regulated industries where portfolio management, requirements traceability, release planning, resource management, document workflow, baselining, and enterprise risk analysis are required. The company uses a concurrent pricing model for all its tools with unlimited products, projects, sprints, tests, API calls, included in a single price. All Inflectra products have a 30-day free trial.