Continuous Testing in Hewlett Packard Enterprise:

Increasing delivery speed & quality for Agile and DevOps teams

Malcolm Isaacs

Hewlett Packard Enterprise Software

EuroSTAR Software Testing

ABSTRACT

Technological revolutions bring real challenges for software development, testing and operations teams. These influences include: the Internet of Things, Software as a Service, and modern delivery practices such as DevOps. Organizations are being asked to release new application features to market rapidly, even continuously, without sacrificing quality. This task is achieved by testers and developers collaborating to achieve consistently high quality in their lean, Agile and DevOps delivery processes. They are testing continuously throughout the entire lifecycle – from development all the way though to production.

Continuous Testing means running tests at each stage in the pipeline, giving the team feedback to improve quality and increase their velocity. Tests are triggered automatically by events such as code check-in. The faster defects are found, the faster they can be addressed. Continuous testing doesn't happen by itself. Instead it requires a significant shift for development teams to succeed.

This eBook explores the concept of continuous testing and what it involves. It describes how Hewlett Packard Enterprise Software development teams adopted Continuous Testing throughout the development process to achieve faster delivery and higher levels of quality.

3 Key Takeaways

- Understand the concept of Continuous Testing and the benefits it can bring to your organization
- Discover how HPE Software implements Continuous Testing to improve quality and increase their velocity
- Learn how you can get started with implementing Continuous Testing in your organization

EuroSTAR Software Testing

BIOGRAPHY

@MalcolmIsaacs

Malcolm Isaacs is a senior researcher in Hewlett Packard Enterprise Software's Application Delivery Management (ADM) team, with a particular interest in Agile, DevOps, and software quality. During the course of his 25+ years career in software development, Malcolm has held various positions, including software engineer, team leader, and architect. In 2003, he joined Mercury (later acquired by HP), where he worked on a number of products specializing in supporting traditional and agile software development life cycles, from planning through deployment. You can follow @MalcolmIsaacs on Twitter, reach out to him on LinkedIn, and read his articles on TechBeacon and on the Hewlett Packard Enterprise blogs.

Contents

Introduction

Have you ever watched a chef prepare a dish? You might have noticed that as they work, they examine the ingredients in detail – tasting, smelling, feeling, looking, and even listening to them. Their knives are always ready for action, exceptionally sharp. At each stage of the preparation, they taste the food—even though the final result will look, taste and feel totally different. If something isn't quite right, they'll stop immediately, and fix the problem. There's no way they'll serve their customers food that doesn't come up to their demanding standards.

This form of 'continuous tasting' is emulated by the software industry in the form of 'continuous testing'. It's been around for a long time, but started to become popular as part of eXtreme Programming, formalized by Kent Beck in the 1990s. XP, as it became known, said that if a little testing can eliminate a few flaws, then a lot of testing can eliminate more flaws. Developers would write unit tests to validate small pieces of code, then testers wrote and ran acceptance tests to ensure that the requirements have been implemented correctly.

As Agile development started to become more popular during the 2000s, testers and developers were encouraged to work together as part of the same team. They would collaborate with the product owner to define the functionality of a feature, and deliver the software incrementally. Rather than waiting for the developers to finish the coding phase, testing is conducted in parallel with development, so that working software can be delivered at the end of the sprint.

Today, the new buzzword is 'DevOps' – bringing the development team and the operations team together to deliver software quickly without sacrificing quality. This can only be achieved by keeping the source code free of defects by detecting problems as soon as they are introduced, so that they can be weeded out before they get closer to the customer.

Whether you're using waterfall, agile or DevOps methodologies, Continuous Testing will help you deliver better software.

This book explains the concept of continuous testing, what it involves, and describes how HPE Software adopted continuous testing throughout the development process and achieved faster delivery and higher levels of quality.

Important

This eBook is a work-in-progress, and is constantly evolving. To get the latest edition, subscribe to TechBeacon.com, and look for updates in TechBeacon's Free Downloads section.

The evolution: Waterfall to Agile to DevOps

Waterfall

The Waterfall development methodology includes a well-defined testing phase as part of the software development lifecycle. It follows the development phase, often at the 'Feature Complete' milestone. Defects would be found and logged during the testing phase. When testing was complete, the developers would fix the defects, and the testers would get another build to test at the 'Code Complete' milestone. This model worked, but it was far from perfect:

- **The 'touch point' between testers and developers, when they start to collaborate, is at the handover**. The developers would explain the feature and what needs to be tested, but there was no context to allow the testers to focus their efforts.
- **A change to code would only be delivered at the next drop to QA, which could be weeks or even months later.** If the change introduced a problem, it would be discovered very late in the process. This would take even longer if most of the testing was performed manually.
- **Only defects deemed a 'blocker', preventing the tester from being able to test the feature, would be fixed immediately.** Other defects would have to wait till the developer goes into the bug-fixing phase. Of course, by that time, the developer had moved on to other things, and would need to do an expensive context-shift to remember the feature and what needs to be done to fix it.
- **Development usually takes longer than estimated, but milestones tend to stay fixed.** The only variable left is quality, which invariably suffered as serious defects would be reconsidered as 'medium priority' in order to reduce the number of essential fixes for the release.
- ***I*ntegration was very messy.** Pieces would only come together during the latter phases of a project, and they would often be developed according to different specs and understandings. Like two jigsaw pieces that don't quite fit together, the parts would be forced together, and both of them would become distorted in the process.
- **Load testing and security testing were performed late in the development cycle**. In many cases, poor performing code would be released to production because there wasn't time to re-architect a badly designed infrastructure.
- **The customer would only get to see the product once it was released.** This is way past the point where their feedback could influence the product's development. In some cases, by the time a product or feature was released, the customer no longer had a need for it.

Agile

Many organizations have adopted a different, more responsive process, known as Agile development. Agile development practices were created as a response to the rigid, bureaucratic, and linear approach that was typical of legacy waterfall development. Agile emphasizes delivering working software as quickly as possible to enable customer feedback and interaction.

- **Developers and testers collaborate on what they're working on from the very start of development.** In Agile organizations, they are part of the same team, working with the product owner to understand and refine what each feature will look like, and what the criteria will be that determines when each user story is done.
- **The team aims to deliver the Minimally Viable Product (MVP), which is the smallest functionality that provides value to the end user.** Then the team releases additional functionality at regular intervals. The goal is to release working software as quickly as possible, maintaining a consistently high level of quality in order to rapidly get user feedback.
- **Developers regularly check code in to the main source code repository, even if a feature isn't completely implemented.** Only code that compiles—and also passes a specific battery of tests—is allowed to make it into the source code trunk. If a bug is found, it is fixed immediately.
- **Different components of the application are integrated regularly.** The integration is tested frequently.
- **They prefer automated testing to manual testing.** Although manual testing is important, particularly for new functionality, teams aim to write automated tests that can be run as part of the continuous integration with the mains source code.
- **Testing isn't the only thing that is automated.** Many organizations are automating their infrastructure, allowing them to quickly and repeatedly deploy and configure testing and production environments.

The customer, or end-user, is involved in the product's development from the start. They are able to provide feedback, and the development team is able to make changes in response to that feedback, resulting in a product that meets the end-users needs.

DevOps

Today, organizations are increasingly adopting a DevOps approach, by bringing the development team and the operations team closer together to build high-quality software that can be released to production at any time. DevOps attempts to remove artificial impediments to delivering software and empowers the delivery team to be responsible and accountable for the software they build and ship. In order to lower the checkpoints between development and production, there are several critical considerations:

- DevOps teams depend on robust communication and collaboration to ensure that the team is not introducing technical debt.
- Automation of manual steps is essential. Typically, manual steps in delivering software often introduce defects, latency, and rework. Automation of everything from compiling and building a code change, to deploying and configuring software, to testing the software is essential.
- Providing and acting on feedback at every stage.

DevOps teams gain a number of benefits, including:

- **Time to market is reduced.** DevOps encourages teams to expose the end-user to the software as early as possible in order to get feedback.
- **They build the right product.** Because the end-user experiences the software from an early stage, the DevOps team can correct their course immediately.
- **Releases are more reliable.** The DevOps team must be able to release the software at any time. By investing in automated deployment, there are fewer surprises when deploying to production.
- **The product's quality is higher.** The emphasis on being able to release at any time means that the DevOps team has to ensure consistently high quality.

Just as a chef uses all of their senses to examine their food at each stage of preparation, today's software development team tests the software at each stage of the development process. This is known as continuous testing.

Confidence through continuous testing

Today's software development team often treats software as a form of experiment, where they depend on feedback to prove their hypothesis. Development teams strive to release software updates to their customer as quickly as possible so they can iterate and improve. To achieve this, they adopt mechanisms that enable new functionality to flow as quickly as possible from the developer's keyboard to the end-user's screen. This path from developer to end-user is usually termed the 'pipeline', and in its ideal form, it is a fully-automated process that includes continuous integration and building of code, testing, deployment, and monitoring of the system in production. The pipeline is typically implemented by a Continuous Integration server, such as Jenkins, Bamboo, TeamCity, CircleCI, etc, and delivery automation platforms such as Chef, Puppet, Ansible, Codar, etc.

The goal of the pipeline is to rapidly and efficiently deliver changes to the system and then validate that the system works as expected through testing. The complete pipeline will move and validate a change from the developer's workstation through the QA environments—all the way to production. Anything that impedes the goal of rapid delivery must be removed. Any code that enters the pipeline is rigorously tested to ensure that it doesn't add technical debt—through defects or instabilities—to the product. Code should flow smoothly through the pipeline, and any bottleneck that impedes that flow should be fixed. If a defect is found in the code, the flow of that code is halted, and the defect must be fixed in order for it to continue its journey.

Many organizations adopt a process called 'gated check-in', which performs a quick check on any code being checked into the source code repository. If the code doesn't compile, or doesn't pass a basic set of tests, the code changes are rejected and not added to source control. As a result, the code is not allowed to enter the pipeline. Of course, this is no guarantee that the code doesn't have defects, but it does help ensure that the pipeline always has code that compiles and meets the minimum basic standards for entering the pipeline.

The knowledge that the system will prevent you from checking in code that doesn't compile, or will reject code that doesn't pass the tests at each stage in the pipeline gives developers confidence. Now they can check code in frequently, without being 'that guy' who broke the build and prevented others from checking in their work.

Triggered builds versus scheduled builds

Continuous Testing means running tests at each stage in the pipeline, which are triggered automatically by events such as code check-in. But at each of these stages, you can also configure tests to run periodically—even if no known changes were introduced. This type of scheduled testing alerts you to changes that might have been introduced inadvertently. An example of an inadvertent change may be a manual change to configuration that bypassed the automatic deployment mechanism. The aim of Continuous Testing is to alert the team to problems as quickly as possible, and running tests all the time is another mechanism to ensure prompt feedback.

Enabling continuous testing

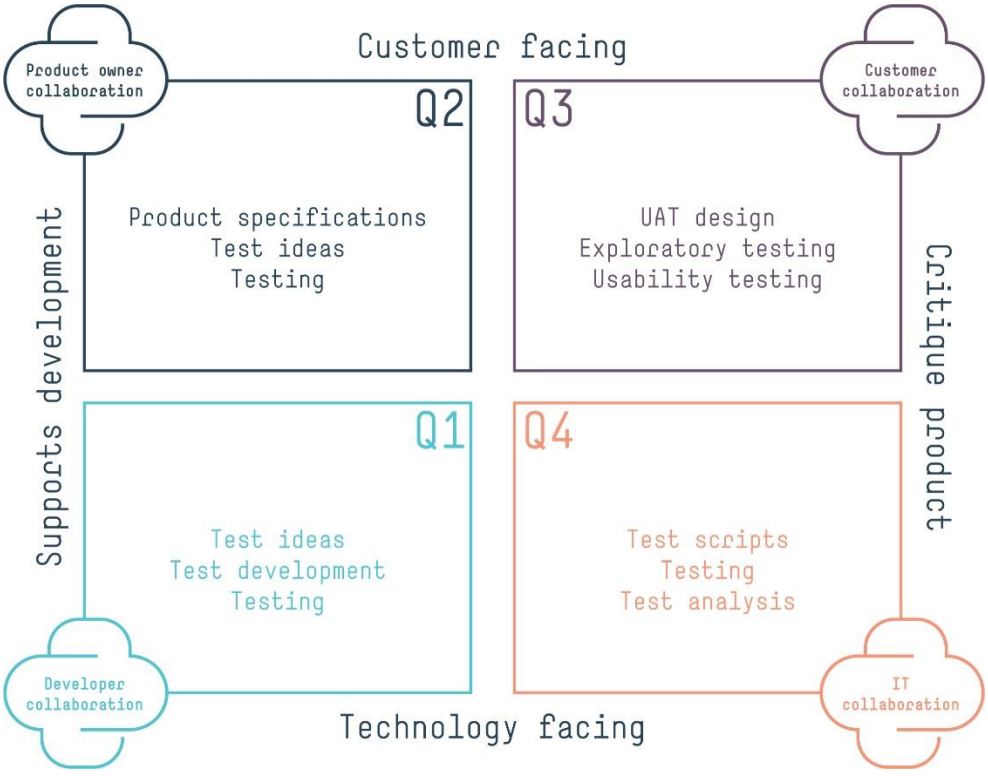
The ultimate goal of Continuous Testing is to give the team feedback to improve quality and increase their delivery velocity. The more quickly defects are found, the faster they can be addressed. Continuous testing doesn't happen by itself, but instead requires a significant shift for development teams to succeed.

Continuous testing involves four essential practices: collaboration, continuous integration, automation and continuous assessment.

Collaboration

Sharing and communication are crucial if a team is attempting to accelerate delivery and improve quality through continuous testing. Teams need awareness of issues and status changes in order to work in concert—as opposed to creating re-work and overhead for each other. This is especially true as each member of the team plays their specific role and works toward the overall goal. The importance of communication and collaboration is fundamental in agile development techniques.

If you've adopted agile techniques, you will understand the importance of collaboration between different teams, and the people within each team. But even if your developers and testers aren't part of an agile team, you can still ensure effective communication between the different people involved. Declan Whelan illustrates this in his presentation "Agile Testing: The Role of the Agile Tester":



(Reformatted from original image, licensed under Creative Commons Attribution 4.0)

Collaborating with the product owner and the customer will ensure that the testers and the developers understand and plan in detail exactly what the product, its features, and its user stories are supposed to do. It will aid them in adopting a test-first approach. It's not a monologue though; testers and developers ask questions for clarification, and bring up any obstacles that might prevent a feature from being implemented in the way that the product owner initially proposed. One of the goals of the collaboration is to agree on the **definition of done** that will determine what tests need to be written, executed, and of course, passed, in order for a feature or a user story to be considered complete. It's also imperative that the testing is performed on the target platforms and environments, so the team should be working with IT and Operations in order to understand how the product will be deployed and secured, and how it'll be monitored in production.

Collaboration is not a one-off event. There should be continuous collaboration throughout the lifecycle of the product. For example, testers can perform ad-hoc testing on developers' machines so that they can give feedback even before the code is checked in.

In agile and DevOps environments, it's common to find testers and developers working together on feature teams, taking end-to-end responsibility over features from the UI all the way through to back-end. This ensures that the right people are talking to each other and reducing risk. The team as a whole takes ownership of quality, rather than leaving it to the QA team.

Continuous Integration

Continuous Integration means that each team member frequently checks in and shares their changes in the source code repository—often many times per day. This way, they 'integrate' their changes with each other very frequently. The use of a common repository is vital for continuous integration and the repository should be used for ALL the development artifacts including source code, test scripts, deployment scripts/instructions and configuration scripts/instructions. Typically, the act of checking code into the source code repository will trigger a number of actions to be performed. For example, when someone checks code in, the code will be merged with the main code base, and automatically built on a build server. If the build completes successfully, a series of tests will be run. These are typically short-running tests, so that the developer will get feedback quickly if any of the tests fail. In some cases, this is actually performed 'on the fly' by a gated check-in mechanism, which will build the code and run tests before actually committing the code to the main repository. This helps ensure that the main source code trunk is kept clean, and will always compile and pass the basic tests.

As we described in the previous section, the pipeline is usually implemented through a Continuous Integration server. In simple terms, a CI server is an automation hub that interacts with key development systems, typically source repository, testing tools and reporting systems. The CI server guides a change through the pipeline, effectively shepherding the change through different stages. The server guides changes from initial check-in, to testing, deployment, and even post-deployment testing and monitoring. Continuous testing is almost always implemented via a Continuous Integration server.

Automation

Most applications today must support multiple platforms, particularly on the client side. Laptops and desktops often run different versions of Windows, Linux, and Mac OS. Each of these operating systems support multiple browsers, such as Chrome, Firefox, Safari, Internet

Explorer, and Edge. On top of that, today's applications must have a presence on mobile devices such as Android, iOS, and Windows Mobile. Keep in mind that each of these mobile operating systems can be installed on a plethora of different devices and form-factors. Manually testing even one combination of device, operating system and browser is a huge task, and takes a long time. But in a continuous testing environment, there are many different combinations to test, and you need to get the results of the tests quickly.

Unit tests are automated by nature, and are typically configured to run automatically as part of a build. Similarly, system and integration tests should be automated, because it takes too long to execute the tests and get the results if you do it manually. This includes testing non-functional requirements as well, such as: testing performance, security and regulatory compliance. Some manual tests are still performed, even in a continuous testing environment, but this is usually limited to exploratory testing of new features. In order to get feedback quickly, the vast majority of tests must be automated.

Setting up the different environments that the tests require takes time. You'll need to set up your application's servers with the correct operating system and support software such as a web server. Then you'll need to deploy the application itself and configure it—including populating the system with data for testing. You'll need to set up your client machines and devices with their operating systems, browsers, and any configuration they require. This preparation process can take a lot of time, and is error-prone when done manually. By treating your infrastructure as code, the testing and production environments can be deployed and configured without any manual intervention. Once the environment is established, the application can be deployed and configured on top of it automatically, and without any manual intervention. This ensures consistency of testing environments and production environments.

Continuous assessment for fast, high fidelity feedback

Automated building, testing and deploying is essential, but it's not particularly valuable if you don't know the results and the status of the pipeline. If tests fail, you must understand which tests failed and where they failed. This knowledge allows you to investigate why they failed, and how to remediate the failure. Similarly, if the build fails, you need to know who to contact to fix it. If the deployment fails, you'd better know about it before your customers do. Once the application is running in production, you'll need to monitor it to detect and correct problems as soon as they occur, and identify usage patterns so that you can optimize your development and testing to actual usage scenarios.

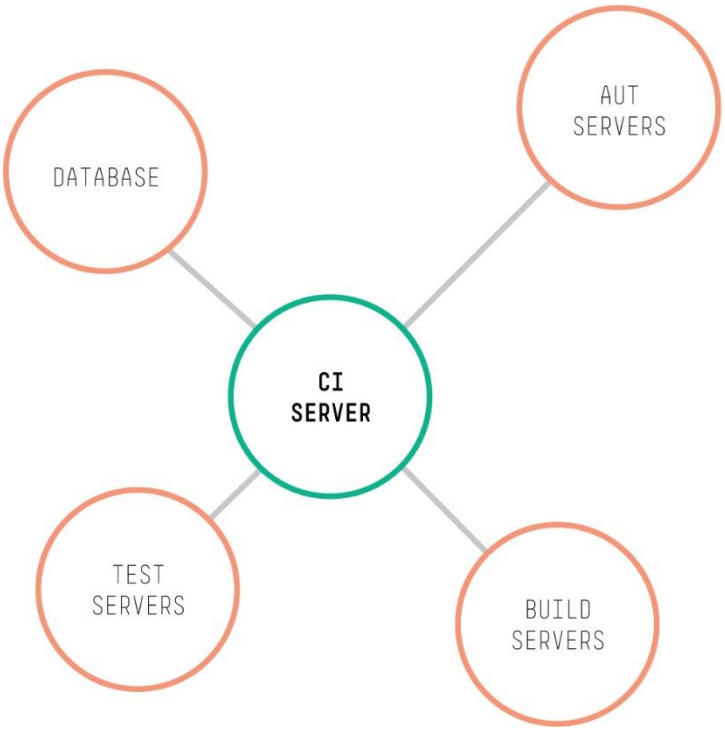
And updated dashboard is the most common way of providing insight into the pipeline. This dashboard needs to be visible to the whole team. At its simplest, the dashboard is a green/red indication of whether or not there are problems. But when the dashboard is red, it must provide a way of drilling down into the source of failure, and assigning someone to fix the problem.

The dashboard can provide way more information than just a pass or fail status for tests. It can show indicators such as the team's velocity. It also shows trends over time, such as the software's performance improvements or deterioration, and other information that can be used to help make decisions. Some systems today are implementing predictive algorithms to help you estimate the amount of time it will take to fix defects.

Continuous testing in HPE Software

In Hewlett Packard Enterprise Software, we have many business units, each developing a large number of products. Each product has its own idiosyncrasies and quirks, and there are many different requirements to take into consideration too. In recent years though, all of our product teams have gone through an agile transformation, with some product teams working with DevOps.

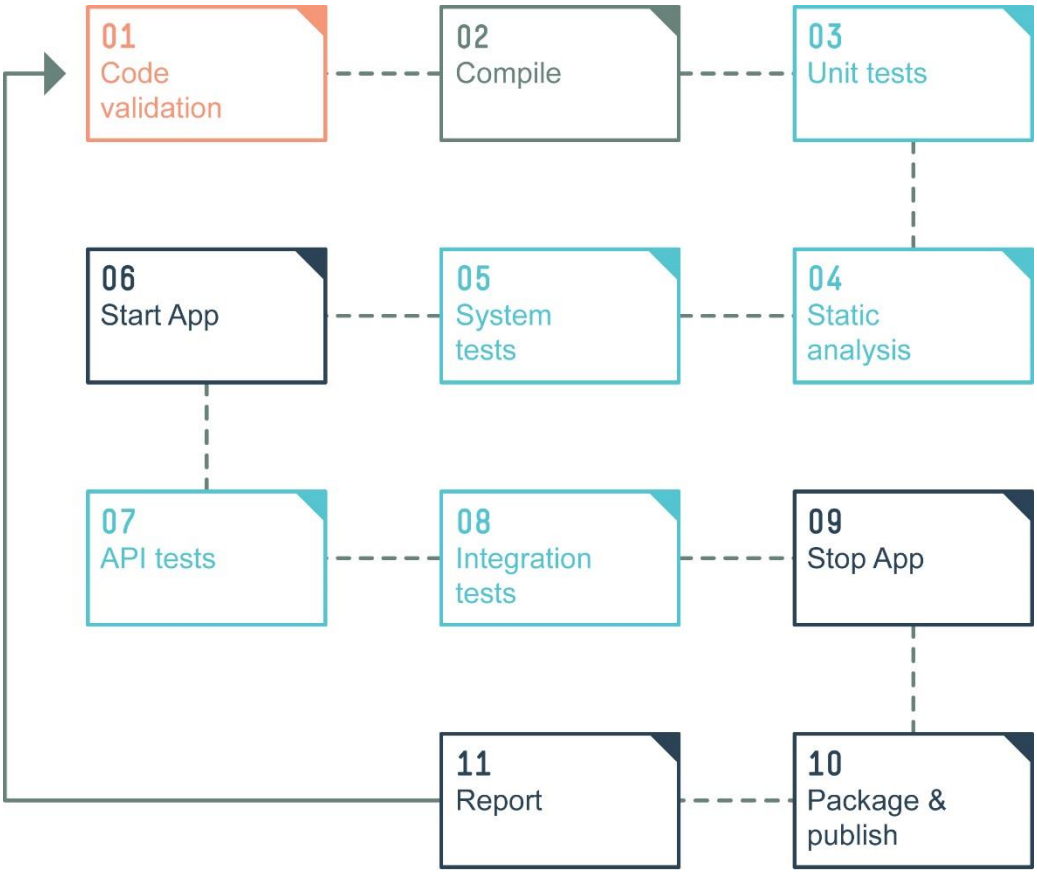
Although there are differences between each product's implementation, each of these teams have several common factors. Most notably, they all work with a delivery pipeline and Continuous Integration server, and they all implement continuous testing. This is how the continuous integration server deploys the various servers for one of our products:



The servers involved are:

- **CI (Continuous Integration) Server** – All of our delivery pipelines are implemented with Jenkins.
- **AUT (Application Under Test) Servers** – This is where the application to be tested is deployed.
- **Build Servers** – The builds are performed here.
- **Test Servers** – The AUT is tested by running tests from the test servers, typically by issuing API calls, or driving a browser on the test server that accesses the AUT.
- **Database** – The database(s) that most applications require.

Here's the journey through that product's pipeline:



1. **Code validation** – When code is checked in to the source code repository, a quick check is first run to make sure that the code compiles and passes some tests. If the code doesn't compile or fails a test, it will not be checked in. This is the concept of gated check-in.
2. **Compile** – The code is compiled.
3. **Unit tests** – We run a lot of unit tests after every compilation.
4. **Static analysis** – We also run static analysis on the source code to find issues with code, and with front-end code such as HTML and CSS
5. **System tests** – Component tests are run to ensure that the different parts of the application are working. This product's pipeline doesn't require the whole system to be deployed in order to perform the system tests.
6. **Start App** – Once the system tests have completed, the application is deployed and started, so that more extensive testing can take place.
7. **API tests** – Test the API layer of the application.
8. **Integration tests** – Test the end-to-end functionality of the application, typically through the GUI.
9. **Stop App** – Stop the application.

10. **Package and Publish** – Create a deployable package that can be installed on the customer's site.
11. **Report** –Report the results of the build and the tests, and update the various dashboards.

In the next section, we'll look at the process within a typical HPE Software pipeline.

Flowing through the pipeline

The developers and testers will usually start a task by meeting with the Product Owner to define and refine the requirements. By the end of the session, everyone understands what they need to do to complete the task. Most importantly everyone understands the criteria by which the task will be judged to be complete. The developers start writing code and will integrate it with the source code repository. This is where code enters the pipeline. In parallel, the testers work on automating the testing from the beginning. Let's look at what happens when the developer attempts to commit some code.

Committing code to the Source Code Management system

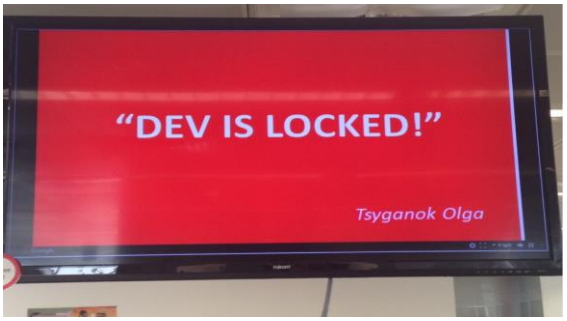
Following informal and ad-hoc testing that the developers perform on their local workstations, the developer runs a quick validation suite consisting of basic and complex validation tests that are required to pass before the code can be checked into GitHub. Within HPE Software, we use GitHub as our source-code management (SCM) system. To automate this preliminary validation, we use the open source Verigreen plug-in to prevent 'bad' code from getting into the main trunk. If there's a problem, it alerts the developer who attempted the commit and the developer can fix the problem without impacting the rest of the team.

If the code passes this preliminary testing, it is checked into the main source trunk.

Successful commit triggers Jenkins to do a build

When code is successfully committed, it triggers a comprehensive build, causing the entire code base on the main trunk to be compiled. Because a pre-check has already run on the code, the compilation usually succeeds. In the rare event that the compilation fails, the developer who performed the last check-in is notified, and must fix the code as soon as possible. When the build fails, the trunk is locked, and no more changes are allowed in until the problem has been resolved. This prevents problems from accumulating, and ensures that the build and trunk is kept GREEN.

Teams take the status of their build and code base very seriously and have dedicated monitors always displaying the status of the source code repository. Here's an example of a monitor, showing when a senior test engineer, Olga, locked her product's trunk because of a bad build. The monitor continues to display the message, and no code can be checked in, until the problem has been resolved, when she'll open it back up:



In parallel to the monitor, emails and other alerts are sent to make sure that everyone understands what is happening in more detail, and that someone has been assigned to fix the problem.

Unit and System Tests are run

As soon as the build has completed, Jenkins runs a suite of automated tests that have been pre-configured to run. Typically, the set of tests is designed to complete execution quickly so that the team gets feedback on the build's quality as soon as possible.

The test suite consists primarily of API level unit and functional tests along with some performance and security tests as well—although they must complete quickly as rapid feedback is a priority. The tests are run on a test environment which is automatically deployed after the build.

As with a failed build, if the tests fail the trunk is locked. If a test fails and the trunk is locked, it is usually the result of a serious problem that could not have been picked up on the developer's environment.

The goal is to complete the process and report the results within 15 minutes of the commit that triggered the build.

Full build

If the tests succeed, Jenkins performs a full build on all of the software. It deploys a test environment, and then deploys the compiled and built code to the test environment. In some cases, this might require the approval of someone from the team, but whether someone has to approve this step manually or it happens automatically, the deployment itself is done automatically, without having to manually copy or modify files.

Integration tests and regression tests are run

Once the test environment is ready and configured, with the application installed, configured, populated with test data, and running, more extensive end-to-end tests are run, and their results are reported back to the dashboard in HPE ALM Octane.

We design the full build, deployment, and tests to take no longer than an hour and a half to run to completion.

Nightly Build

In addition to the builds that are triggered by a developer committing source code to the repository, we also run a more comprehensive, long-running test each night. Some products with lengthier tests might do this on a weekly basis. These tests are more detailed, and often involve intensive load and performance testing, full regression testing, upgrade testing, and testing across the full product availability matrix (PAM) of supported operating systems and environments.

We don't limit the amount of time it takes for the nightly build and its tests, although realistically, we try to keep it to eight hours or less, so that the results are ready by the time the team starts work the next morning. We monitor the progress continuously through our dashboards.

Deploy to production

If the previous batch of tests succeed, the software *can* in theory be deployed to production. In practice, most of our products do not actually deploy immediately after the tests have succeeded. The most common scenario is that we first deploy the product to a staging area that

is identical to the production environment, where we run extensive tests. If we're satisfied with the overall quality, we deploy to production at a time that causes the least disruption to the end-users. As with the deployment to the test environment that we saw previously, the process might need to be triggered manually. But because we've adopted the concept of treating our infrastructure as code, the process of deploying to production is completely automatic, as is the configuration of the application and its environment.

Production tests run

Testing doesn't stop once a product is deployed to production. One of the first things that happens after deployment is to run a series of tests, to ensure that the deployment actually succeeded. These tests include end-to-end functional tests, API tests, security tests, and performance tests. Many of these tests are automated, but we also do manual end-to-end testing in production.

Application is monitored

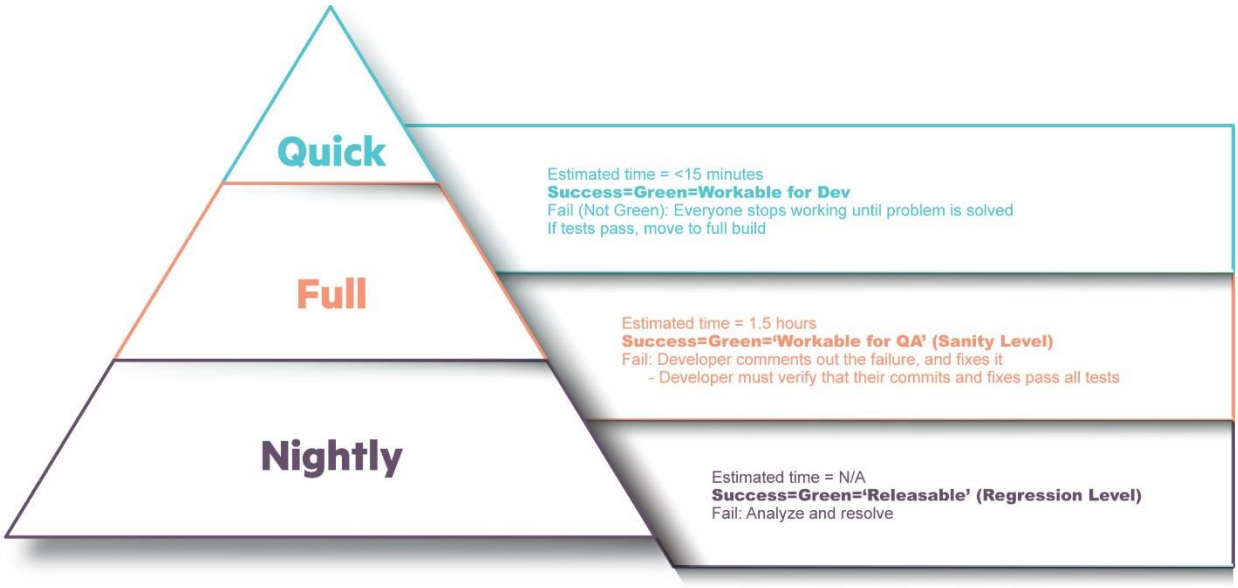
As long as the application is live, it is constantly monitored for performance, security and user experience. Many of our products make extensive use of the HPE AppPulse Suite, which provides a continuous picture of the end-user experience with application monitoring. The AppPulse Suite also provides the information and context required to isolate, identify and resolve problems before the users experience them. The information gathered by AppPulse helps us understand how our users work with our products in practice, which helps us to optimize our products' features, and optimize the testing that we perform on them. This is an essential component of the feedback loop that continuous testing enables.

Testing everywhere

We just saw how testing is performed continuously throughout the pipeline. Although it sounds complex, it follows a pattern of:

- Running short tests to give a very quick response to the development team who commit changes to the source code repository about the quality of the change.
- Then a more extensive set of automated tests is run.
- Then a much more extensive nightly or weekly set of tests is run.
- And finally, testing the deployment to production and monitoring it.

Here is a visual representation of how continuous integration testing starts small and grows over time:


Quick
Full
Nightly
Estimated time = <15 minutes
Success=Green=Workable for Dev
Fail (Not Green): Everyone stops working until problem is solved
If tests pass, move to full build
Estimated time = 1.5 hours
Success=Green='Workable for QA' (Sanity Level)
Fail: Developer comments out the failure, and fixes it
- Developer must verify that their commits and fixes pass all tests
Estimated time = N/A
Success=Green='Releasable' (Regression Level)
Fail: Analyze and resolve

Drill down into testing

In the previous chapter, we looked at the various stages of the pipeline and where testing takes place. In this chapter, we'll take a look at the different types of tests and testing considerations.

Behavior-Driven Development

The conventional software development process involves defining the requirements for a feature, writing the code, and then testing it. Test-driven development (TDD) on the other hand, swaps the order of coding and testing, by writing the requirements for a feature in the form of tests, before writing the code for the feature. When the tests are run, they will of course fail, because there's no feature code to run it against. But as the code is written, tests start to pass, and as the requirements get implemented, more tests pass, until they all pass. Effectively, test cases serve as the requirements in addition to being a test. In HPE Software, we use an extension of TDD called Behavior-Driven Development (BDD), which lets the team use a structured natural language to specify the acceptance criteria for user stories. We use Gherkin for our BDD, where the criteria follow the form "Given [a context], When [an event occurs], Then [ensure this outcome]". These criteria are easy to understand, and can be followed manually, or executed automatically. BDD gives us a better understanding of the feature, and keeps us focused on the bigger picture.

User stories are elaborated collaboratively, so that everyone involved understands the motivation for the feature, and everyone agrees on the **definition of done**. The definition of done will include all of the tests, functional or non-functional, that the code will have to pass in order for the user story to be deemed complete. As the developers work on their code, the testers work on the test cases and test code/scripts, working together to ensure continuous integration and agreement around interfaces and the expected functionality. We've found that BDD has reduced ambiguity in acceptance criteria, by ensuring that product owners, testers and developers all use the same language.

Developers test too

Everyone in the delivery team is responsible for quality, and developers are no exception. There are a number of steps that they take before code enters the pipeline to ensure that tests will not fail because of an obvious and wholly avoidable defect. The idea that a team member would write code and fail to validate their own work is simply unprofessional and unacceptable. Developers own their code and must own their testing. Here are typical developer QA / validation activities:

Basic manual functionality testing

When a developer writes code that implements a button-click, they will test that the code works by clicking the button. When they develop and API, they will make sure that at least the 'happy path' of the API works. If they know that something doesn't work, they will either fix it, or comment it out before attempting to check it in.

Code Review

Before checking code in to the repository, each developer will ask another developer to review their code. A fresh pair of eyes can find problems that the original developer didn't notice, and having another person become familiar with the code is invaluable.

Static code analysis

Static code analysis tools can detect weaknesses in the source code, and non-compliance with coding standards. Although static code analysis is run as part of our deployment pipelines, developers run static analysis on their own machines before they commit code, and fix any issues found. They only commit code once it's passed the local static code analysis. The developers work with the DevOps engineers to define the static code analysis tests that must be run.

Automated Unit testing

Developers spend a lot of effort writing unit tests. They ensure that the unit, whether it's a method, a class, or other component, works as expected. Part of the pipeline involves running these unit tests, but developers also run the unit tests before checking the code in, to catch any issues before they get into the pipeline. Teams have coverage goals for their unit tests and a build might fail if the unit test coverage isn't extensive enough.

To enable unit testing, the developers must be able to run the unit in isolation, without being dependent on other components or services. They usually employ mock objects and virtualized services to replace those missing components and services.

Single-user performance testing

Another simple test that developers perform is making sure the software is responsive when only they are using the system. If it's taking more than a few seconds to display a web page taken from a local or emulated web server, they'll investigate the issue and fix it before checking the code in. They know that slow performance with a single user is often indicative of performance issues with additional users.

Next Step: DevTesters

While developers are responsible for many aspects of testing their code, there is an important role for QA specialists, who are dedicated to evaluating and validating the quality of a given system. In close partnership with the developers, these QA professionals, often called "DevTesters", take a deeper look at the overall quality of the system.

Functional testing

As befits an organization that develops the market-leading testing tools, we use our own tools extensively to test the products that we develop. In fact, we even use our tools to test themselves. Once code is in the pipeline, it is subjected to a battery of functional and non-functional tests.

Functional tests are designed to test that a feature works correctly. We perform a lot of end-to-end functional tests on the GUI, or presentation, layer, but we usually run far more tests on the API layer. Testing the API layer is vital, because APIs are how the various components of the application communicate with each other, and just as importantly, that's how other applications integrate with it.

Non-functional testing

Whereas functional testing looks at how a specific feature works, non-functional testing looks at broader aspects of the product. We test non-functional aspects extensively, and it is an

essential component of our continuous testing philosophy. We introduce short performance tests as part of our check-in testing, increasing the amount and scope of testing we do the further code flows down the pipeline.

Some of the non-functional aspects we test are:

- Security
- Performance, Load and Stress
- Compliance testing (where applicable)
- Localization (L10N) and internationalization (I18N) testing
- Documentation testing

Testing Environments

Testing does not happen in a vacuum. Every test must be executed in a specific environment, and it must be possible to reliably recreate that environment every time we test to ensure consistent and repeatable results. A testing environment refers to:

- **Servers** – The various application and database servers on the back-end. This includes the operating systems and installed software, such as the specific version of Tomcat, or Java, or database that is installed on the system.
- **Clients** – The different systems used to access the back-end servers, including operating systems, browser versions, and any specific configurations. This includes desktop and laptop computers, and also mobile clients, such as smartphones and tablets.
- **Services** – The back-end often depends on services, whether internal or external. Some testing environments allow the back-end to access these services, and other environments use service virtualization to ensure repeatability and consistency, and to avoid incurring costs or load on third party services. See later for a more detailed explanation.
- **Data** – Data is one of the most critical characteristics of the test environment. The data must reflect the planned test cases, scenarios, and transactions to ensure that tests will effectively evaluate the quality of the system under test.

We use different testing environments for the different types of testing that we perform in the pipeline. This section explains which environments we provision for each stage of testing.

Check-in

After a successful build triggered by a check-in, we provision an environment for unit and component testing. Some product groups will also run some short-running functional and non-functional tests at this point, on the same environment. To get the results within 15 minutes, we sometimes need to provision multiple environments and run the tests in parallel.

Full Build

A full build requires a repeatable and consistent environment, on which we install and configure the components of the application under test. For example, if there are multiple possible environment configurations like an on-premise product supported on both Linux and Windows, we will create configurations for each environment. When that’s not possible, we choose the most popular and critical configurations for testing.

Nightly Build

The nightly build (or weekly build, for some products) is where we perform the most extensive testing of the products. After the full build, integration tests are run on each supported environment for both functional and non-functional aspects. We often provision multiple environments in parallel, so that we don't have to wait for one environment to complete before starting to test a different environment.

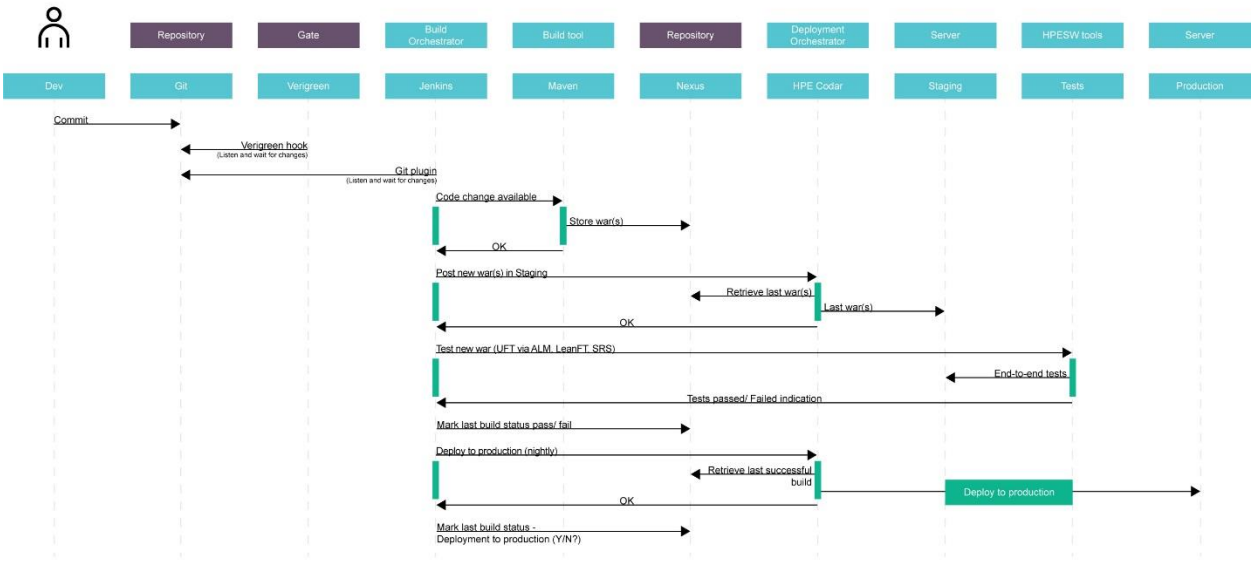
If the tests pass on the nightly build, the build is considered a *release candidate*. That doesn't mean that it will necessarily be released, but it does mean that the quality of the product at that point in time is at the level we require.

Deployment

Once we've made the decision to release the build, we deploy it. Some products keep the existing server, and deploy the updates to a new system, then gradually migrate users to the new system. This is known as a 'canary rollout'. Other products do 'blue-green' deployment, which involves maintaining two identical environments. The load balancer routes all requests to one of the environments (the 'green' one), and the other environment (the 'blue' one) is upgraded. The load balancer is configured to route all requests to the blue environment, which is monitored to ensure everything works well. If so, the green environment can be decommissioned. If not, the load balancer can quickly point users back to the green environment, which is still running the previous environment. These methods ensure that we can validate that the new deployment is working correctly before rolling it out to everyone. The deployment is done via automation, and we immediately run a suite of tests after the deployment is complete. We run validations on the configuration to ensure that it was deployed correctly, and then run some critical end-to-end tests. We will also run some short performance tests to ensure that the user experience is good. By rolling out the deployment gradually, these performance tests won't affect users that are still on the previous environment—even if they cause the system's response to slow down.

Sometimes we will roll out a new feature or a change to an existing feature, to some users, while the rest of our users continue to use the regular functionality. This is known as A/B Testing, and helps us understand the usability of the feature. If the new functionality isn't successful, we can simply revert back to the earlier functionality by using 'feature toggles' which turn the new functionality on or off.

Here's an example of a sequence diagram of a process from one of our products. Each step in the sequence involves testing, and the tests must pass in order to move to the next stage:


Repository
Gate
Build Orchestrator
Build tool
Repository
Deployment Orchestrator
Server
HPESW tools
Server
Dev
Git
Verigreen
Jenkins
Maven
Nexus
HPE Codar
Staging
Tests
Production
Commit
Verigreen hook
(Listen and wait for changes)
Git plugin
(Listen and wait for changes)
Code change available
Store war(s)
OK
Post new war(s) in Staging
Retrieve last war(s)
Last war(s)
OK
Test new war (UFT via ALM, LeanFT, SRS)
End-to-end tests
Tests passed/ Failed indication
Mark last build status pass/ fail
Deploy to production (nightly)
Retrieve last successful build
Deploy to production
OK
Mark last build status -
Deployment to production (Y/N?)

Roles and responsibilities

So far, we've described how continuous testing is carried out. But who actually does that testing in HPE? There are three main aspects:

- Developing tests
- Running tests
- Presenting and analyzing results

Sometimes, particularly in small projects, the person who develops the tests is responsible for running them—and perhaps also reporting the results. In other cases, each role is filled by many different people. This section explores the different people involved.

Developing Tests

Many different people are involved in developing the different tests that are part of our software lifecycle. This table shows the roles and the tests that they write, with the most technical roles at the top.

Role	Tests	Comments
Developer	• Unit tests • Component tests • Static Analysis	Developers write the code for the feature, but also write unit tests and component tests, typically in the same IDE and language as the code for the feature.
DevTester	• Component tests • End-to-end automated tests • Testing frameworks • Performance and load tests • Security tests	DevTesters are highly technical, and often use the same tools and environment as the developers. Their focus is on writing more extensive tests than the developers' tests. They are also responsible for building frameworks around the testing tools. Their work includes functional testing, performance testing and security testing.
Business Analyst, or Manual Tester	• End-to-end manual tests • Exploratory tests	Focuses on defining the test plan, test flows and required coverage. Regardless of the emphasis on automation, manual testing is still necessary to get a full picture of a product's quality.

Executing tests

In a continuous testing environment, the tests will be configured to run automatically as part of a build. The developers and the testers work with the DevOps engineers to decide which tests run as part of a gated check-in, a quick build, a full build, and a nightly build, and configure the Jenkins jobs accordingly. Nevertheless, it is still necessary to explicitly execute tests outside of the scope of an automated build. Here are some examples:

- Developers will perform some manual testing before checking their code into the source code repository. They might ask one of the testers to act as another pair of eyes to try

some scenarios before they check it in. They'll also run unit tests and components tests on their machine.

- Test Automation Engineers, specializing in functional testing, performance engineering and security, take a nightly build and run a full battery of regression tests on it, as well as end-to-end functional and non-functional tests.
- Manual Testers also work on nightly builds. When they come into work in the morning, they can start to perform manual tests on last night's build.

Presenting and analyzing results

The development progress and quality status of the product is automatically reported to the dashboard. Dashboards are configured to show various metrics from the tracking system, so the current development progress and last nightly build's status is usually displayed in parallel to the status of the last full build.

Significant issues are prominently displayed on the dashboard. Because everyone can see the dashboard all the time, and because the testers and the developers are on the same team, they work together to identify and resolve the problem. This problem could be as simple as a test being configured incorrectly to a major regression from existing functionality.

Essentials for continuous testing

Continuous testing doesn't happen spontaneously or without sustained effort. Rather, it requires planning, collaboration and the right technology to enable and manage robust automated testing. To achieve success with continuous testing, a combination of process and technical enablers are essential. Let's explore a few of these essential enablers:

Testing baked in to user stories (requirements)

User stories (requirements) must include how the story will be evaluated to ensure it meets the needs of the business. Effectively, the story must describe how the developer knows the work is done. User stories are typically created collaboratively at the planning stage, where developers and testers work together to elaborate the story. Developers and testers also define the set of tests that if passed, will indicate that the user story is complete. These criteria, known as the 'Definition of Done', are critical to the success of the implementation. They ensure that both the tester and the developer understand the same definitions and requirements for the feature. Because they work together during its development, they iron out any misunderstandings as they make progress. Gone are the days when testers don't get to see the software until the developers finish and hand it over for QA. Testers are involved in every step of the lifecycle.

Manual testing (exploratory testing)

While we're increasing our coverage of automated tests, manual testing is still part of our delivery processes at HPE Software. When a nightly build has completed, and all of its automated tests have passed, the products of the build are tested manually. We perform end-to-end tests and exploratory testing on specific areas of the product, especially new features. Exploratory tests that find defects that are frequently repeated are identified as automation candidates.

Testing Center of Excellence

The Testing Center of Excellence is still very much a part of our continuous testing. Years ago, the TCoE was part of our Waterfall development methodology, and each product with a queue that built up for the TCoE would have a window, defined a few weeks or months in advance, where they could test our software for performance, or security. Today, while we still have the concept of the TCoE, we use it in a different way. As before, the TCoE consists of engineers that specialize in an aspect such as performance or security, but today, these engineers are part of the pipeline, and work together with the rest of the product team to test the nightly builds. As part of the Center of Excellence group, they share knowledge and experience, define the tools that will be used, how to use them, and how to report the results in a standard way across the organization.

Automating the infrastructure (Continuous Delivery)

Configuring infrastructure (servers, middleware, databases and software) can be onerous and error prone, where small errors can result in significant rework and delays for the delivery team. The concept of Continuous Delivery as described by Jez Humble in his 2010 book of the same name described in detail the benefits of extensive automation of configuring and deploying software to all environments (test and production.) By automating configuration and deployment, these tasks can be completed both rapidly and accurately. The instructions (scripts) to drive the automated configuration is managed in the source code repository just like any other piece of code—which is why the concept is called 'Infrastructure as Code'. By automating the infrastructure, we can quickly and reliably provision numerous complete test environments. We never make manual changes to the configuration. Instead, we make the change to our scripts, and check them in to source control. Then we run the scripts to create the environment with the new configuration. In many cases, we also automate the provisioning of our production environments.

Container Technology

Over the last few years, we've been working on migrating some of our applications' services to a microservice architecture. Microservices are small, independent processes that communicate with each other through APIs. They offer many advantages, such as reusability across applications, and the capability to scale up to a large number of service instances and users. We use Docker containers to host our microservices. Containers isolate the microservice from the underlying infrastructure, and offer benefits such as portability, security, and better distributed computing capabilities. In production, containers offer major advantages over virtual machines which can take quite a few minutes to come online in order to respond to demand. Containers are much lighter, and start in seconds. We make extensive use of container technology in our continuous testing pipeline. Their lightweight nature allows us to deploy and provision services and pre-configured environments automatically and almost immediately, without having to perform any operating system installation or configuration. This allows us to run tests on multiple containerized environments, and increase our test coverage all without incurring an expensive time penalty setting up these environments every time we test.

Service virtualization

Often, an application under test depends on services such as API calls to other systems, whether external or internal. Because the dev and testing team doesn't control these outside

services, they become dependent on their availability to complete testing. There are many testing scenarios where a tester expects repeatable and reliable responses from a given service, but without control, this can be a real challenge. Simply said, external services can quickly become major bottlenecks for development and testing teams. To overcome this constraint, development teams can simulate, or virtualize, services so they can eliminate dependencies and regain control. This allows them to test as much as they want—particularly load testing—without fear of affecting the performance of the real services.

Service virtualization enables them to test their application's interaction with a variety of unique conditions, such as how the system behaves if a service is offline, or if it takes a long time to respond, or if its response is not what was expected (eg. an exception). Service virtualization gives teams flexibility and complete control over test environments.

Test automation framework

Rather than building all of our tests from scratch, teams use test automation frameworks to allow them to re-use automation code and make development faster and maintenance easier. They typically use a component-based approach for functional testing, whereby they create small components that can do something specific on a screen, such as click a button, and then chain these components together to create an automated test that drives the application.

With this approach, if the button ever changes, we simply need to update the component to reflect the differences, and all tests that use that component to click a button will automatically use the updated version. This also allows less technical testers to easily create tests based on components that were written and provided by the automation engineers.

ChatOps and Bots

We've already discussed the importance of collaboration. Having testers and developers—and in fact the whole team—talking to each other is a necessary condition for continuous testing. To make it even easier for testers and developers to collaborate and communicate, we use chat rooms, which brings everyone together. By "everyone" I am referring to the product managers, operations staff, DevOps engineers, and others all collaborating within a virtual space. Whether the team is physically sitting in the same room, or divided across different countries, continents and time zones, chat rooms keep the conversation in the same place and encourages people to share problems and solutions.

It works so well that we've started implementing automated chatbots in our chat rooms. If a tester finds an issue, they can report it to the developer immediately, and get resolution. But if the defect must be logged, the tester or developer can invite a bot into the conversation, and ask the chatbot to log the defect on the tester's or developer's behalf. Rather than having to leave the chat room, fire up the defect management system, collate all of the information, and then enter the defect. The chatbot can do all of this on behalf of the tester or developer— all from within the chat room. Similarly, anyone in the chat room can ask the bot to display the quality status of the pipeline, test coverage, and other metrics that the bot can provide on-demand. Bots can even enable team members to start a build and execute tests right from inside the chat room. Although this is in its infancy, we see a lot of potential for chatbots to help us streamline our continuous testing even further.

Tools

It's not a secret that our own testing tools and application lifecycle management play a central role in continuous testing at HPE Software. In addition to our own tools, we use a lot of open source software – and contribute some of our testing tools and utilities to the open source community. Here is a quick summary of the tools and technology we typically use to enable continuous testing.

Pipeline

Our products use Jenkins as the Continuous Integration server, and our source code is managed in GitHub Enterprise. GitHub Enterprise allows us to share and reuse our source code and projects across our own organization. Verigreen is used to ensure that only code that compiles and passes basic tests is allowed to reach the main trunk in GitHub.

We use many plugins on top of Jenkins, but the most useful for continuous testing is the HPE Application Automation Plugin, which can run various functional and performance tests, and report their results to Jenkins.

Functional Testing

We use HPE Unified Functional Testing and LeanFT for functional GUI and API testing. HPE Mobile Center, which manages the mobile device lab, allows tests to be developed and executed on both real and emulated mobile devices. For manual testing, HPE Sprinter is our tool of choice.

Performance Testing

We use all of our load and performance testing solutions: HPE LoadRunner, Performance Center, and StormRunner Load. Any application that relies on a network must be able to react and provide feedback to the user even when the network is slow, erratic, or even unavailable. To ensure that our applications perform well even in adverse network conditions, we use HPE Network Virtualization to simulate the application's behavior under different network characteristics.

Security Testing

HPE Fortify and HPE WebInspect are at the heart of our security testing throughout the pipeline.

Lifecycle Management

We use the HPE ALM products and HPE Agile Manager to manage the overall lifecycle of each product. We do our planning using the Agile backlog, and Kanban views help us to manage each sprint and release. ALM Octane also provides insight into the pipeline, and displays the status of the pipeline at any given moment. Defects are tracked in ALM Octane, and quality and other metrics are graphed and displayed on our dashboards from ALM Octane.

Deployment

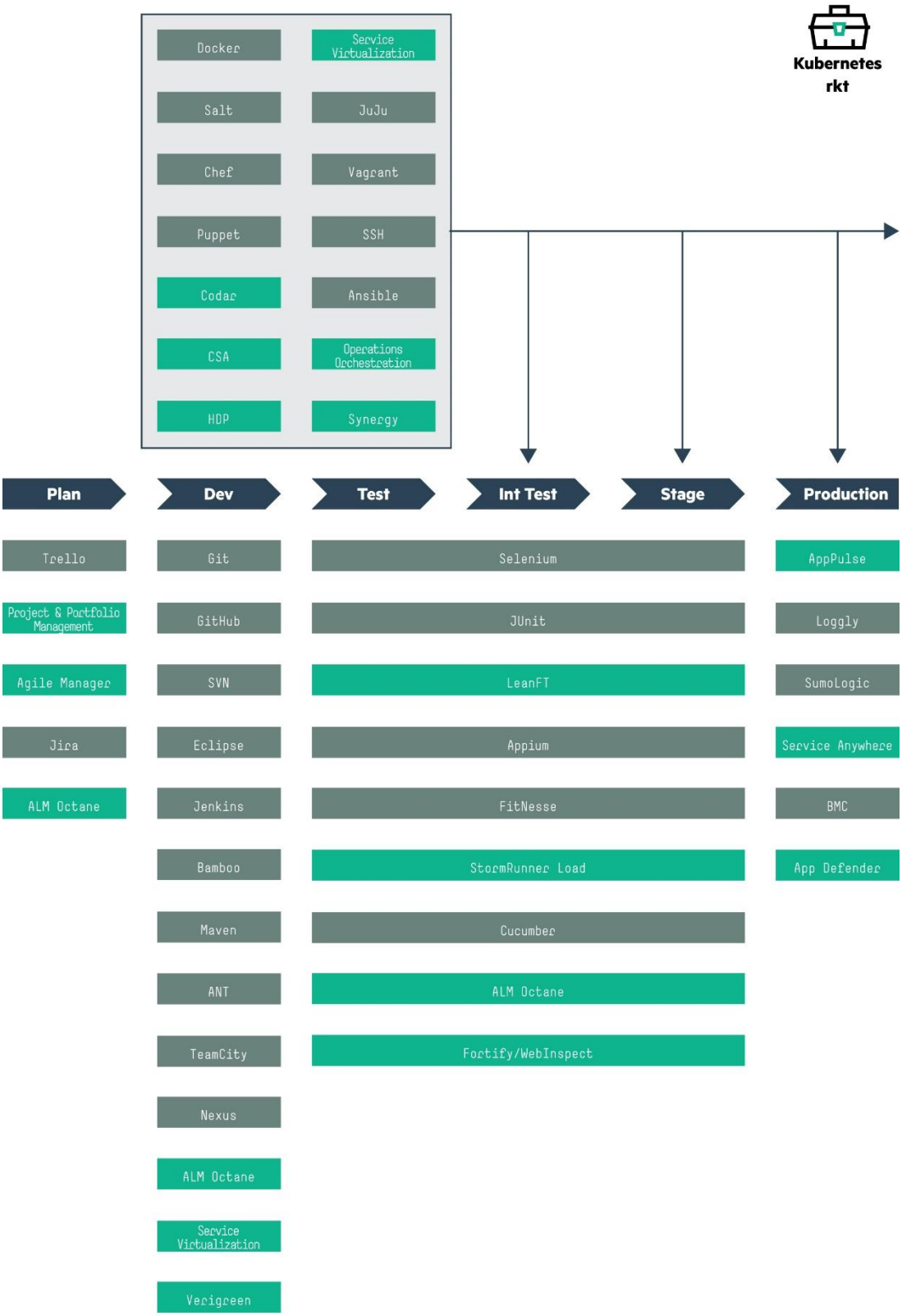
Deployment is done with a combination of tools. HPE Codar manages the process with Ansible to perform the actual deployment and configuration.

Production

Once the application is in production, we monitor it using a number of different tools. User experience and performance is monitored using the HPE AppPulse suite, and HPE AppDefender employs Runtime Application Self-Protection (RASP) to shield the application from attack and protect its data.

If issues arise, HPE Service Anywhere is always on call to handle incidents and route them to the support and development staff to resolve.

Here's a diagram of some of the tools that continuous testing teams use in the pipeline (HPE products are in green):


Docker
Service Virtualization
Salt
JuJu
Chef
Vagrant
Puppet
SSH
Codar
Ansible
CSA
Operations Orchestration
HDP
Synergy
Kubernetes
rkt
Plan
Dev
Test
Int Test
Stage
Production
Trello
Git
Selenium
AppPulse
Project & Portfolio Management
GitHub
JUnit
Loggly
Agile Manager
SVN
LeanFT
SumoLogic
Jira
Eclipse
Appium
Service Anywhere
ALM Octane
Jenkins
FitNesse
BMC
Bamboo
StormRunner Load
App Defender
Maven
Cucumber
ANT
ALM Octane
TeamCity
Fortify/WebInspect
Nexus
ALM Octane
Service Virtualization
Vericreen

Beginning the journey

The journey to Continuous Testing is long, but the key is to get started, and build on top of your successes. This is what worked for us.

Collaboration

We started our journey a few years ago by adopting agile methodologies and training our DevOps team members in effective agile techniques. We restructured our teams so that developers and testers would be part of the same agile teams and take collective as well as individual responsibility over tasks. This led to better communication and responsibility over features and user stories. By removing barriers to communication, we found that we were able to move more quickly and efficiently. Our testers became more technically aware, and our developers understood that unless a user story has been tested and achieves the criterial that were set together with the testers, it can't be marked as done. Good communication is an essential pre-requisite for effective continuous testing.

Continuous Integration

We adopted GitHub as our enterprise source code management system— standardized on Jenkins for build management. This allowed us to create a pipeline to shepherd code through the route from development to production, with everyone involved understanding the requirements and definition of done for each feature and user story. The pipeline is central to every activity involving the product, whether it's code, tests, configuration, or deployment. Once the pipeline was in place, we started looking at the processes we had around the pipeline, and started looking at opportunities to automate as much as we could.

Automation

Continuous testing requires automation. The purpose of continuous testing is to give the team and team members an instant, or close to instant, indication of whether a change has broken the product. A change could be an update to source code, a different environment configuration, a modification to a test, a different database, or any of a myriad possibilities. If, as soon as a change enters the pipeline, the team gets feedback about the quality of the change, a harmful change can be fixed before it gets further down the pipeline. Automation is the primary way of achieving this speed of feedback. Many components of the pipeline can be automated – tests, deployment, configuration, monitoring, etc. So what should be automated first?

To prioritize your automation, you need to understand what you need to achieve:

- It could be releasing software more quickly
- Getting quick feedback
- Reducing the cost of downtime
- Getting metrics into a single dashboard for the team
- Or any of a number of possible goals

Pick the goal that you want to tackle first, and analyze why you haven't achieved it yet: Do the developers tend to check bad code in to the source code repository? Are the tests taking a long time to run? Do you need to manually prepare data for testing?

Now consider why these bottlenecks exist. If bad code is getting into the pipeline, implement a gated check-in system. If testing takes too long because deploying test environments takes a long time, move to an automated process for provisioning environments. If preparing data is intensive and error-prone, start automating the process of populating the database that drives the tests.

By this point, you might have a number of tasks that you need to automate, so you need to prioritize them. Work out which task will give you the biggest return on investment in the shortest amount of time. Maybe a large task can be broken down into shorter ones, each of which delivers value.

Once you've implemented a task, monitor the pipeline to measure any improvement. Make sure you re-evaluate your priorities before tackling the next bottleneck. In today's dynamic environment, circumstances may have changed since you last looked at the bottlenecks.

We applied these techniques to prioritize our automation at the start, and we still use them today. There's always more automation that can be done, as new features are developed and new requirements come into play.

Testing

One of the first things we did was to look for manual tests that we could automate. We looked for places where we could introduce more testing, such as starting performance testing and security testing earlier in the pipeline. We made sure that we reported our results to a common dashboard to which everyone had visibility.

Our testers and developers worked together to expand test coverage at the unit test, system test, and integration test levels. We also worked with our DevOps engineers to ensure that these tests run as part of the gated check-in, build, full build and nightly build processes. Once the initial testing framework is in place, it's very easy to add more. We used the process in the previous section to prioritize the testing that we needed to do.

Afterword

Continuous testing is an essential part of continuous delivery. Without quick feedback about changes that have been introduced into the system, whether knowingly or unknowingly, any adverse effects will only be detected further down the pipeline. The further a change gets from its moment of inception, the more difficult it is to detect and fix.

Our journey to continuous testing in HPE goes hand-in-hand with our journey to DevOps. DevOps isn't something that can ever be declared as 'done'. It's a process that focuses on optimizing and improving the end-to-end flow of new features to meet business demand. It is business focused, not IT focused.

By continuously investing on continuous testing, you'll detect poor quality changes as close as possible to their point of origin, and be able to push them out of the pipeline before they go any further and do any more damage.

We've adopted continuous testing in Hewlett Packard Enterprise, and it's working well for us. We hope that this book will help you in your own journey towards continuous testing, and high-speed delivery of high-quality software.

Learn more at TechBeacon.com

Hewlett Packard
Enterprise

Enjoyed this eBook and want to read more?

Check out our extensive eBook library on Huddle

Join us online at the links below

EuroSTAR Software Testing