TestOps in a DevOps World

Simon Howlett

simon.howlett@workiva.com

Abstract

A DevOps team's remit is that of 'frictionless development'. This is a desire for a development ideal driven by a 'need for speed' and the conscious & intentional removal of hold ups in taking an idea from concept to customer. Frictionless development allows product teams to concentrate on innovation instead of being beholden to often-arbitrary processes. This movement poses an interesting question for Quality Assurance Engineers as the 'need for speed' mindset places challenges them to do more with less (time, mostly), and the challenge to sync the two sides of the conversation is often fractious.

Focusing on increasing the speed of delivery often brings issues of quality into focus. Careful relationship management and compatible tooling coupled with a quality driven mindset across an organization is required to attempt to ensure that 'need for speed' is taken as a metaphor for efficiency not just 'fast'.

Aligning both DevOps and TestOps teams is a technique that can speed up the product release and support a robust and reliable quality focused product, two areas that often comprise each other.

Biography

Simon Howlett is a Senior Product Development Manager in Test Engineering at Workiva. Focusing on providing Test Frameworks and Solutions with an emphasis on reliability combined with ease of use/support. Simon currently manages a team of engineers based mostly in Ames Iowa, though he himself resides in Portland, Oregon.

Copyright Simon Howlett, 2015

1 Introduction

Workiva provides a cloud-based SaaS platform for collaborative reporting across multiple industries. They have been successful in part through creating a customer first culture throughout the whole organization. To Quality Assurance (QA) team members, that culture means striving to ensure everything released to customers reaches a level of quality and reliability the customer can count on. Workiva's Wdesk platform handles many large complex documents for customers working in often highly regulated industries and this demands a focus on reliability, security, accuracy all of which provide our engineering teams with some quite challenging opportunities. Workiva also like to release every day, go fast and in most cases rewrite conventional wisdom, particularly in terms of people's roles in producing a 'quality' product.

Any Agile software organization looks to release product updates often (in some cases more than daily) and see benefit from putting their teams in charge of their own releases. DevOps teams are often the focal point of this process, creating tooling for build and deployment environments allowing teams to go as quickly as they desire but in the testing world there is often a disconnect in this ideal due to checks and gates created as part of a QA process. This process creates an impression that QA and testing speed is a bottleneck to the organizations goals. An equivalent discipline, 'TestOps' exists, that when working alongside DevOps can be vital in ensuring this is not the case, and can more importantly assist in team empowerment by creating a seamlessly deployed testing environment alongside DevOps build and deployment infrastructure.

A 'TestOps' team in essence concentrates on availability of infrastructure and platforms required for testing at all levels, from functional testing through integration testing to lower level unit and API tests. This is a change from the traditional Test Engineer role that can often be the person responsible for writing test fixtures and maintaining test frameworks. Through making all test tooling available to all of the product team as part of the same workflow they already use allows a change in mindset which helps to move the responsibility for quality (through test coverage and process) onto the whole team (instead of the traditional 'tester'). When combined correctly with a DevOps 'frictionless development' mindset a very robust clean development, testing and release process can occur.

Where a DevOps team would focus on build, configure and deploy, a TestOps team can interface with the build and configure stages to communicate with this system provide testing feedback mechanisms, that are defined in testing specifications owned by product teams themselves. This gives a seamless commit-tested workflow that allows for fewer bottlenecks in the development process. This flexibility extended with continuous Integration removes hours of manual intervention to setup testing environments, reducing the time it takes to get quality, tested code in front of the customer. This can only be successful if DevOps and TestOps platforms are in sync, and the two teams work closely together both on products and process.

2 Case Study

In early 2010 Workiva had a team of 30 development engineers and a QA team of 5 people, all working on Wdesk, a cloud based application aimed at enabling companies to create and file accounts, pre-releases and other documents with the Securities and Exchange Commission (SEC).

Pre-release testing was completed on a manual exploratory basis and very quickly the ability to cover the application with just manual testing became an issue and graduating from manual test scenarios to an automated testing process became a project for a small group of QA engineers.

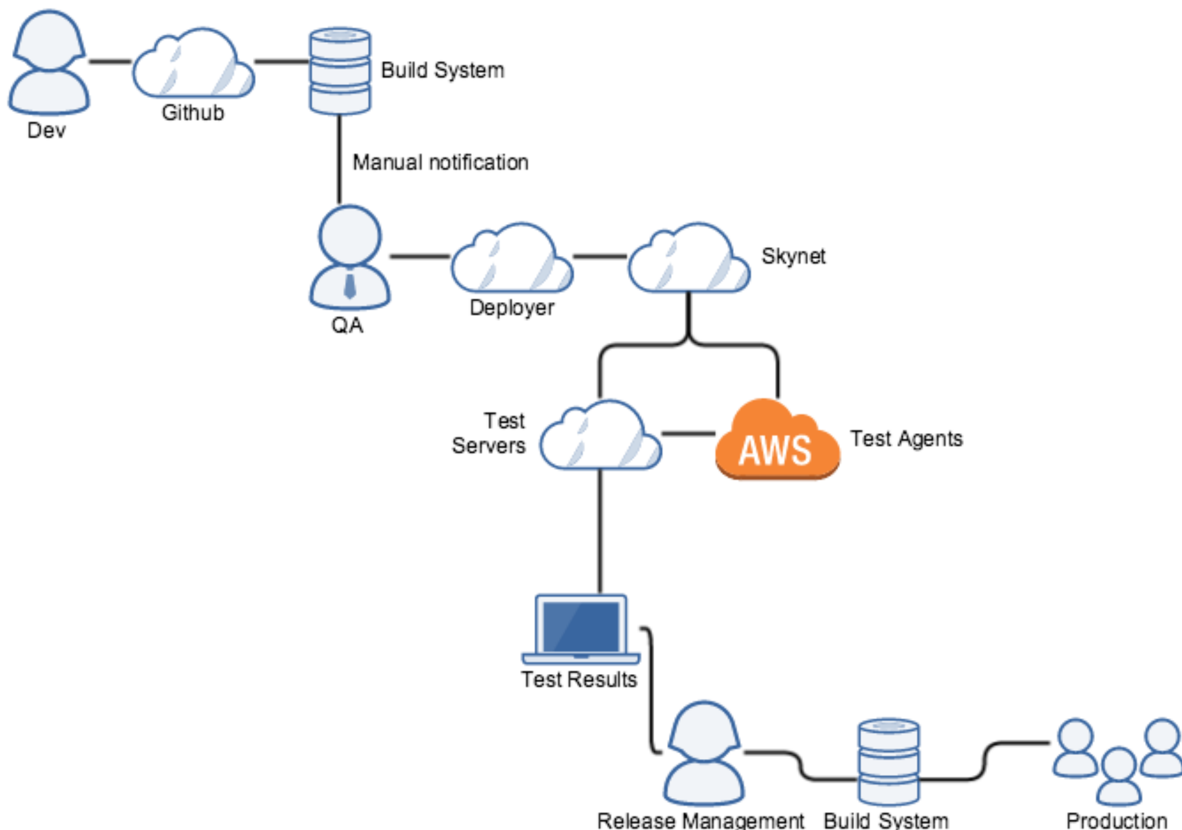
The initial goal for this new team was to automate test coverage for the 800 manual test scenarios the QA team worked through prior to each release (at this point Workiva were on a release schedule of every two weeks, with up to 2000 updates in each release). These test scenarios included mostly end to end tests ranging from document creation and editing to format translation to confirm that documents content and formats were correct when translated to formats defined by the SEC for corporate accounting.

An initial challenge to this plan was the platform on which the testing needed to occur was built using Adobe's Flex language. This only provided a small number of off the shelf testing options, really only one that was deemed suitable for the task, SmartBears 'TestComplete', and its test runner, 'TestExecute'. To make things simple for testers to create coverage we created an easy to use drag and drop interface to create tests built using business language driven test fixtures. This separation of test runner and tests meant that rapid progress could occur on gaining test coverage for our product without the need for testers to write code or incur large license costs for test complete.

Over the next two years test coverage grew from the 800 tests per release that covered the initial manual test cases (the initial goal for the project), to 7000 full stack tests. These tests were now running 24 hours a day, full runs on release candidates, customized components for product team builds and small hourly smoke test builds.

Over time the supporting infrastructure also increased from running tests on a small farm of physical machines to a virtualized platform of 40 Google App Engine Servers, 400 Amazon ec2 machines replicating the users, and running through around 1,000,000 test scenarios run per month. Somewhere along the way the framework adopted the name 'Skynet'.

Figure 1 - Initial Test and Release Process



2.1 Initial Testing Success, future scaling issues

The initial goal of providing test coverage had been met and had found hundreds of bugs prior to release (and continues to), though it was quickly becoming apparent that these tools are very 'tester' focused. Although functional test coverage could be created by developers at the same time as putting products and features together, the tooling to do so lived outside of the environments and repositories developers worked in each day. This was not going to be scalable as the organization moved quickly to a model of multiple product teams and services releasing independently through a streamlined release infrastructure championed by DevOps. Other time consuming issues arose around required manual intervention (as seen in figure 1) a manual handoff between development and QA post build, and the same to Release Management, post QA results review, and these needed to be mitigated going forward.

In some ways this testing approach had been too successful. Functional tests were used for all testing, in some places where much faster lower level tests could have been used. Particularly in the area of service testing (such as document translation and comparison), where over 1000 functional tests existed. These tests took around 4 minutes each performing the same user actions that were not actually the point of the test – the translation itself took seconds, so this was a costly problem for everyone

This was not the only challenge to testing in the existing workflow;

- **Flaky tests.**
  Release Candidate runs average around 99.1% passing rate on over 7000 tests. This means that on any test run QA has to research 1-30 failed tests. There are some tests that are unreliable given the frequency of failure due to latency or odd infrastructure conditions. As a result the aim to have 100% passing tests runs on our main customer product is made unrealistic by the platform it is testing – there has still to this day never been a full functional test run on this part of our application with 0 failed tests!
- **Lots of maintenance**
  Rather than a product team owning and updating tests a secondary team has to update fixtures and mappings for the flex based tests due to prohibitive costs of TestComplete. This also introduces risk by having a team not familiar with the product under test, investigating the reasons for failing tests – often coding around bugs to resolve issues without realizing.
- **Time**
  To complete a full test run takes somewhere in the region of 3 hours. This is mostly restricted by the number of TestExecute licenses, and our server platforms ability to scale to handling a given number of test tasks at any one instance. Going forward working in a much more distributed model, test runs could not possibly take this length of time
- **Cost**
  Each test run stands up the full application stack. This involves storing, copying, amending data on 40 servers and around 300 user machines. This proves expensive (much more than service level/integration tests) though when compared against the cost of those bugs caught getting passed to the user the effect is still positive – even though seen as 'revenue protection' it is clearly not the most cost effective way to do this. Costs for testing approached around $80,000 per month at their most expensive point.
- **Tests could not be dependent on each other, no collaborative testing**
  Each test is run in isolation and state is not preserved between tests. Also collaboration scenarios could not be tested which was a large problem given a large focal point for our customer was robust collaboration on documents.

As the organization moved to break up into discrete teams each contributing its own micro-service outside of the monolithic codebase it became important to re-evaluate this model with the entire product team in mind, and the new DevOps delivery model.

3 Scaling with new DevOps and TestOps teams

With DevOps teams taking responsibility for build and deployment architecture allowing engineering organizations to scale as quickly as their business demands, its vitally important that TestOps teams are able to interface accordingly and grow alongside DevOps. Having a test infrastructure that is not as easily maintainable and usable as a nimble build and release mechanism will lead to delays in software releases as QA gets backed up in checking for release confidence in its prescribed test activities. This cohesion is made more important as Agile teams move to releasing smaller release more frequently to customers. In turn, with many cloud-based vendors available to customers, quality must be upheld even with the ability to update applications at a much greater frequency.

In early 2014 Workiva created a DevOps team to rethink release infrastructure to support more products, with an intention to move to a model of interoperating product teams and micro-services, allowing each team to release independently and much more frequently – more than once per day.

What had previously been an application platform based on a monolithic codebase was being broken into individual products, modules and micro-services. This changes the release model from one that can be managed in one place by a Release Management team, to one where teams could benefit from being able to release independently as and when they choose.

The DevOps team was to build and support a new architecture and workflow enabling product teams to release software at a time and frequency their own choosing. Building in support for container based application deployment (using Docker) and creating an eco-system that supported all technologies that product teams saw the best fit for their products/services.

Alongside DevOps, the Test Engineering team (prior to being known as TestOps) was already owners of a test framework and this team had already began to realize that the model for testing that was currently in use could not scale, certainly in an organization that was moving quicker and wanting to deliver software much faster than before.

Ensuring that the DevOps and TestOps teams and tooling are in sync is a challenge, competing priorities, differing customers and outlooks mean there needs to be a very simple interface between the two, and an understanding that this interface must always be available. Common REST API's proved to be an efficient choice, allowing the TestOps team insight into what was building/deploying in DevOps world, and in return, Test Results and Reporting from TestOps to support releases were provided back to the build system (which in turn needed to be provided to Release Management to support questions from external auditors).

All of this must be close to seamless as any manual intervention or delays to development reduces buy in from teams and becomes a perceived bottleneck or flakiness in the process. So communication on dependencies between teams was established at the outset, with an architect from each team in place to ensure the vision of both teams, whilst focused in different areas leads to something that appears to be the same outcome.

Another interesting component in making all these changes work together is a Release Management team. Responsible previously for managing every part of every release of updates to customers, in the new model teams are empowered to be able to release on their own at any time they feel ready to do so.

The Release Management focus moves to building tools to assist in quicker release of products through the removal of checks that we previously done by humans, providing a suite of tools such as test coverage reports and automated release process checks (checking code reviews and testing sign offs are as expected on pull requests, for example) to check for auditable reviews and test artifacts and streamlining teams release process to allow them to move faster.

This new found team autonomy poses an interesting challenge for a Quality Assurance Analyst/Engineer as the model they are used to working in of testing one 'ball' of code where all assumptions can be clearly

seen, changes to one where they must know the status of each of these component parts at time testing occurs – and where any interoperability occurs.

The QA Analysts role itself is also adapting to become more of Quality Coach, or in some cases, a Quality Engineer (more accurately, the 'Software Development Engineer in Test', or SDET) and the role of a traditional tester was seen as possibly becoming obsolete. The idea of a separate entity solely responsible for the quality of an application does not fit the agile paradigm of the 'whole team' being responsible for quality. Whilst the QA person on a team was responsible for ensuring quality, they should not be seen as the only person responsible for building it into a product and process.

Critically at this point, the established Test Engineering team focus needs to move to a more 'TestOps' like outlook to ensure we could provide the tools and support to ensure the high standards upheld in terms of quality were maintained at scale, and where teams are now accountable for their whole release process, but kept in sync with the 'frictionless development' path being championed by DevOps and engineering as a whole.

3.1 DevOps & 'Frictionless Development'

DevOps teams are the providers of 'frictionless development'. Allowing teams to innovate and not be hampered by environmental or workflow issues, putting new code into production and respond to feedback quicker. In an Agile environment teams themselves are empowered and accountable for making what they need to happen to release quality software that customers love, without any noticeable drop in quality. This means the DevOps team undertaking the task to build and support an infrastructure to support a wide ranging build, test and deployment system on whatever platform teams are needed by team to build and deploy their new products.

3.1.1 DevOps Tooling

Continuous Integration (CI) systems are the backbone of DevOps success. A good CI system enables quick feedback on build stability, quicker artifacts into QA, and ultimately quicker to production code.

Typically DevOps will own the build system, deployment tools (assuming cloud/server based products), production monitoring systems (occasionally referred to as CloudOps) and often release consistency measuring tools such as code coverage reporting. All of these tools can also be leveraged by TestOps, to deploy test builds, report on performance and consistency etc. providing feedback to product teams on exactly the same infrastructure used for production deployment reducing time and cost to setup and maintain separate test resources for deployment and also give a much more real world feedback loop for release confidence.

3.2 Build and Test Infrastructure Cohesion

DevOps and TestOps infrastructure need be in sync. It should appear as one smooth system/workflow to any engineer, committing code (and test coverage) should be all that they need to be concerned with, transparency and reliability is key for success anything that adds to the time taken to gain feedback on features loses the momentum 'frictionless development' strives to provide...

A quick, reliable build system is the cornerstone of any DevOps infrastructure, Travis Ci, Jenkins, Bamboo etc. are well trusted build systems, though Workiva chose to build a new in-house build system was created with an emphasis on speed and reliability.

A new build system *'Smithy'*

Smithy is instrumented to run unit, integration or functional tests, defined in a yaml configuration file at build time. Failed unit or integration tests prevent a build artifact for release being created (again

increasing the focus on quality and accountability on the product team as a whole). Test coverage and result metrics are recorded within Smithy to support a teams' release, and numerous API's are available for use in other applications for release management, auditing, documentation etc.

These build system API's are the key to working with other systems as they provide the view into the state of any application and its teams requirements at any time. Using triggers like file changes, GitHub pull requests, release branch merges/tags, it can be possible to send notification to TestOps that testing resources are required due to the type of build in progress and at completion tests can be commenced. This makes the experience smooth for the engineer committing code as if they are aware of a given trigger then can start tests without any intervention from others. No requirement for a QA engineer to deploy a build manually to a test server for example.

A new deployment architecture;

A dependable deployment architecture is the secondary DevOps platform piece, again making getting applications out to the customer for feedback / into production simple. It should be configurable to define set standards where deployment is allowed (no drop in unit test coverage, no functional test failures etc.), but these should be safeguards not bottlenecks to a team.

Based on the concept of isolating applications in and environments in Docker containers Workiva two further applications for deployment;

- ***'Shipyard'***: Workiva's Docker container build tool
- ***'Harbor'***: Workiva's package and container release service

Both of these are also needed by TestOps for deployment of builds for testing, so API's etc. where needed should be available to consumers/users.

4 TestOps

TestOps as a concept revolves around ensuring product teams have access to any required test infrastructure, platforms and frameworks they require without needing large amounts of time consuming configuration before commencing tests. Any benefit from use of a CI system will be lost if the QA process takes days to complete due to environmental setup and tear down on the part of the QA person on a team.

A DevOps/TestOps model is intended to focus the whole team on quality and as such anything TestOps does needs to be seamless and reliable. A simple example is any test frameworks must be reliable enough to very rarely have downtime or connectivity issues. Any time lost evaluating flakiness, or delaying feedback on release build will create a bad perception of the usefulness of the system. This makes the mindset of the TestOps team critical.

4.1 Test Coverage, and a Zero Bug Policy

Any TestOps team has to be highly functioning, highly accountable team to get this right. It must be understood what impact is incurred if any TestOps items are out of service or not functioning correctly. Two techniques that can be employed to help with this are high unit test coverage for in house built testing tools, and a 'Zero bug policy'.

The unit test policy is essentially any drop from 100% line and statement coverage fails a build. This means we can rely on our test coverage at a lower level and engrained in the team is a mindset of writing better code. Understanding the impact of loss of confidence in a testing system is great incentive here.

The zero bug policy essentially means any bug raised on production systems at any point during a team sprint must be evaluated and either worked on at the expense of a lesser item, or if deemed not customer impacting, closed and explained as to why.

The result of introducing these policies at Workiva has ensured a highly accountable TestOps team that has a high uptime rate and great respect in the engineering organization. This assists with the communication with DevOps and other teams as we talk as equals, as well as internal customers.

4.2 TestOps Tooling

For a TestOps team, the most important activity is providing exactly what a product team needs to test and receive feedback. For agile product teams and the advent of micro-services a need for new types of test infrastructure arise, instead of a traditional model where an environment for testing depends on the whole application stack being available some more considered requirements emerge:

- What tests do is entirely within the teams control, test governed and managed by an external team promotes risk to product quality
- Testing should need as little setup as possible – any dependent services should automatically be available (or mocked)
- Test results reported automatically along with any generated artifacts, preferably using a defined format (such as xUnit, jUnit, Rich Test Format etc.)
- Tests are executed within an environment that can be mutated in any fashion desired without impacting other tests (such a virtual environments, containers etc.).
- Interoperability, backwards-compatibility, and well-defined APIs have never been more important. Careful attention to API and service level testing is a must.

Most importantly, newly formed TestOps team needed to plan a workflow where teams can own their test frameworks and leverage DevOps infrastructure to enable simple to setup test solutions.

4.2.1 Integration with DevOps infrastructure

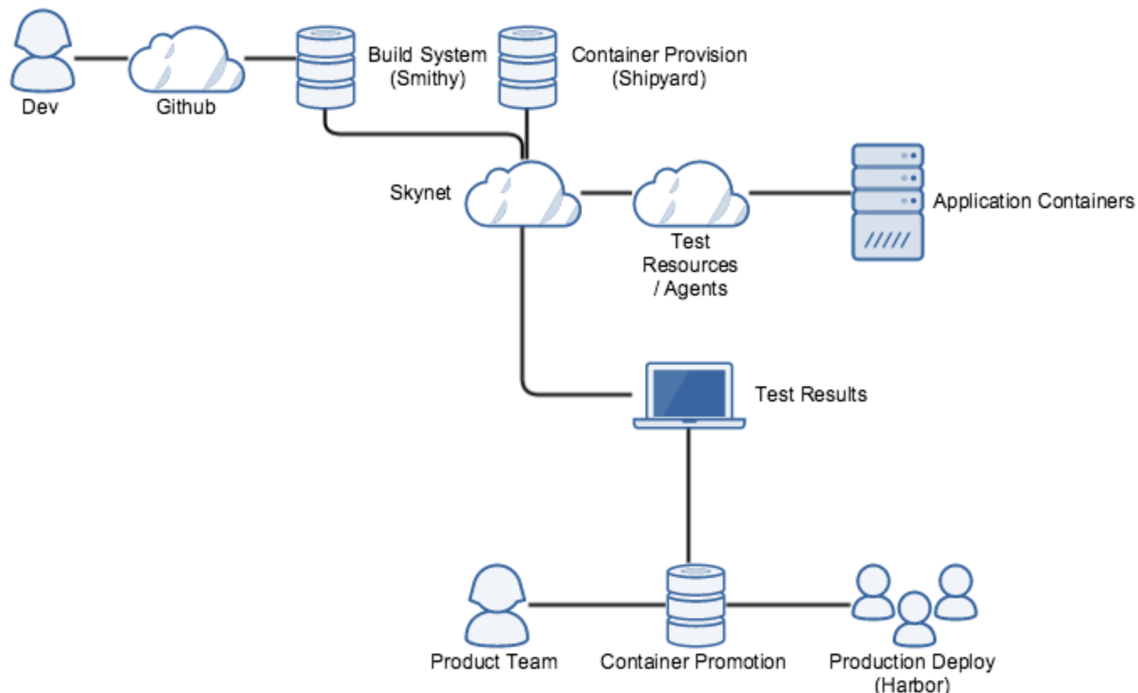
For DevOps the build system is the central application for all teams to create release artifacts and perform lower level tests. In terms of functional testing workflow, given the long running and often brittle nature of these tests preventing build completion on failing tests is a problem, with this in mind functional testing is kicked off as a post-process of the build system, and as such requires TestOps to build an infrastructure that not only takes it cue from DevOps systems, but integrates with them as much as possible to allow for transparent workflow for teams

The connection from a build system to a test system should be as simple as checking an API for notice of build completion, readily for testing. No human interaction to start up functional tests should be required. In Workiva's workflow completed builds are marked as 'success' in the API. Each build in is noted via a listener application in TestOps on starting and when it is completed the API shows a value for the build of "status": "success", 'Skynet' can look to the build itself to see if testing is needed.

To make the call for testing to start Skynet checks for the presence of a skynet.yaml configuration file in the repository from which the code came and if one is found, Skynet reads in its content to pick up requirements & configurations for testing.

This API watching creates the important seamless transition fro DevOps infrastructure to TestOps. The build artifact is available, its low level tests have passed and it is off into any selected functional Testing.

Figure 2 - New Test and Deploy workflow, syncing DevOps and TestOps tooling



4.2.2 Enable teams to easily specify what tests to run and when

To fully realize the value of this seamless workflow, teams must specify what they want testing and when. A specification must exist that TestOps can programmatically understand removing the delay caused by human intervention to configure testing, containing details such as;

- Definition of when the tests are run – every commit, or whenever a specified identifier (in this case Github tag/release is present) and any subset of these
- Where the tests for the given build/module/feature are located
- What test runner commands are needed
- What upstream and downstream tests need to be located and run, again based on the same tag identifier as above
- What service template is to be used for these test – essentially which resources and environment such as servers, Docker images etc. are needed for these test to be executed

TestOps can then translate these requirements to begin testing as soon as a build is complete, testing can commence with no human intervention and at this point QA's task is to review test results.

4.2.3 'You don't have to stand up the whole system to test'

As teams are creating and consuming other teams products, modules and micro-services it is inefficient to need to recreate the whole platform in testing for all testing. Standing up dependent services should be enough as long as it is implied there is continuity between services (consistent APIs etc.).

To test a service simply would involve sending a message from the originating service, and then asserting that the message is received by the message broker. As long as the service that is receiving the message maintains their API, then the secondary and downstream services do not always need to be tested at this point.

For example, when a teams service depends on Vessel, a message broker, to send messages to another service, which depends on a dozen other things all that is needed to be tested is the linkage between their service, and vessel itself, not any other services further downstream

Proper versioning is a necessity here as If an API changes the type of message it receives failure will occur unexpectedly. To ensure consumers can continue to function until the update is available a given standard for API versioning must (and does) exist

4.2.4 Enable teams to also test their changes with upstream and downstream consumers

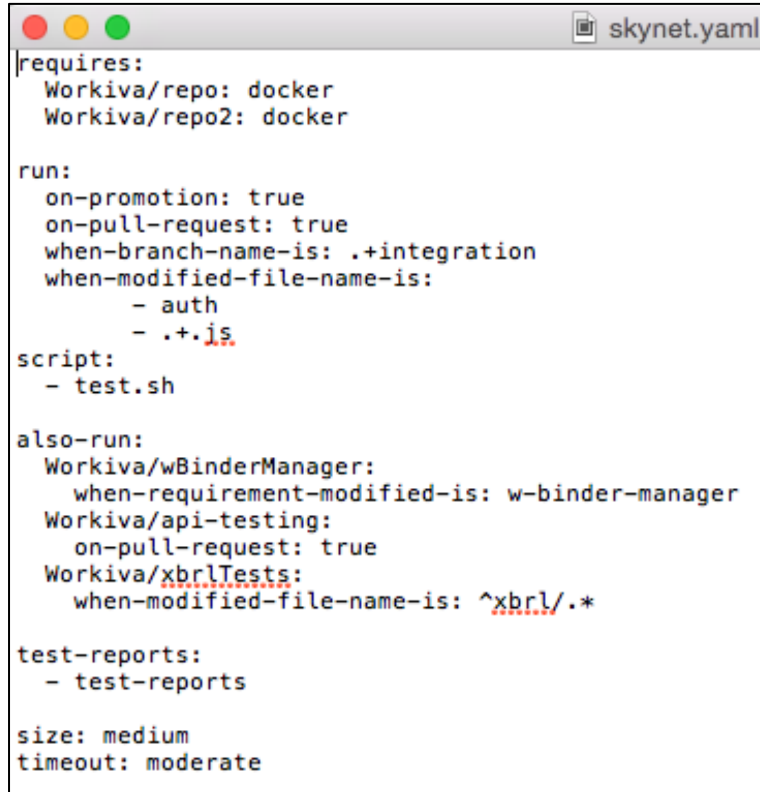
Most team's features are not lucky enough to work in isolation and many depend on external services, or contribute to other products/features. Being able to ensure release confidence on both the changes under test and any consumers is a challenge especially when those consumers are often not in the same location or codebase. TestOps should provide a mechanism for configuring what other modules tests can be utilized post build to ensure consistency and root out any breaking changes to downstream consumers without manual intervention. This configuration is best placed in the same location as the other configurations for testing, the testing specification as noted in the previous sections. This specification is another item that removes risk, as the product team should in most cases know best what parts of an application/platform their changes will interact with.

4.2.5 Testing Specification Example

Keeping testing specifications as simple as possible is key to success, it should be human readable and of a simple format to ensure consistency. For Workiva the YAML format works perfectly and it is also widely used already across the organization in other products. The skynet.yaml file provides a working test specification which defines when and what testing should occur and if another product or features tests need to be run. This allows teams to ensure that their release does not break another team's work and/or functions in the way that is expected once consumed by others.

TestOps track builds from the DevOps API list of in-flight items to identify that we may need to test. Examining the repository for a skynet.yaml on these builds triggers an event if test criteria is satisfied, queuing the build in skynet ready for tests to commence as soon as the package is available

Copies may not be made or distributed for commercial use



skynet.yaml

```
requires:
  Workiva/repo: docker
  Workiva/repo2: docker

run:
  on-promotion: true
  on-pull-request: true
  when-branch-name-is: .+integration
  when-modified-file-name-is:
        - auth
        - .+.js
script:
  - test.sh

also-run:
  Workiva/wBinderManager:
    when-requirement-modified-is: w-binder-manager
  Workiva/api-testing:
    on-pull-request: true
  Workiva/xbrlTests:
    when-modified-file-name-is: ^xbrl/.*

test-reports:
  - test-reports

size: medium
timeout: moderate
```

Figure 3 - Example skynet.yaml file indicating what testing requirements exist for an application

This above example skynet.yaml file specifies that;

- when a pull request is made from a branch with ‘integration’ in its title (or any of the other specified criteria)

- Using the specified Docker containers run the tests, using any commands noted within test.sh
- run the tests in the also-run section if other criteria are met
- output the reports to the test-reports directory for submission
- a timeout value, if exceeded will cause tests to be failed

Further configurations can include:

- Environment variables needed at run time, such as cloud service credentials
- An over-ride to allow container promotion to production even with test failures
- ‘App-id’ – a request for a GAE Server to specify to deploy a build to and run tests (similar to the original testing model)
- ‘Artifacts’ – a location to store artifacts such as test files created during testing
- ‘Env’ - A list of strings or maps that declare the environment in which the scripts run.
- ‘Image’ – specify a Docker image to use
- ‘Requires’ - Can be set to a map of source keys and artifact values. This creates an explicit dependency on a build artifact for these tests to execute and all build artifacts will be downloaded and made available in the test execution environment before the tests are started. If the build triggering the test run is listed as a dependency, then that version of the artifact will be obtained. Any other dependencies will obtain the latest released version of that artifact.
- ‘On-promotion’ - Execute tests when an application is being promoted to the production application repository.

This ability to set configuration options at the end of build time syncs TestOps work on test frameworks and platforms, with DevOps build and release tools. If all of the pre-build tests pass, the build artifact is passed to Skynet, the tests configured in the skynet.yaml are run, and if these return without error the build is passed for promotion to production, and placed in DevOps deployment queue.

5 Team Investment

With the two sides of the infrastructure in line between DevOps and TestOps, the remaining piece is the cultural change to ensure the benefits of this system are seen by all the product team and they all invest in the quest for quality. The traditional model of separation of duties in testing between development and QA team members can be seen in part as a reason for the slow down in a release process. Developers had the friction removed in their workflow, only to get hung up in labored QA who also had the task of providing test coverage. This slowdown removes any benefit from aligning DevOps and TestOps

The model intentionally puts the test tooling in everyone's hands to counteract this perception, developers could write tests whilst coding features at all levels of the stack and could run them all on commit, rather than waiting for feedback from a QA resource after a round of manual testing. Each build gives them the artifacts required for release or items to be resolved. This allowed the QA resource to assist in test writing, exploratory test with less of a release pressure from the rest of the team who viewed their part as 'done', and also time to concentrate on deeper investigations, such as performance.

This also removes any element of blame. The often heard 'well, did QA test it' is no longer a valid question because it is the whole team's responsibility to support their release, not just the QA or Release Engineers say so. This increases accountability on the team and encourages participation and pride in gaining solid releases. The ideal place to end up is to be able to release software without QA intervention, because the whole team is so invested in test coverage and quality, the DevOps/TestOps tooling allows for quick accurate feedback on the state of a release.

6 Evaluation

Whilst this model is relatively new, Workiva has already seen benefit of this model:

- Because of changes to testing and release tooling we have moved away from an irregular release schedule, where we released an average of twice per week. We now consistently release to production four times per week.
- Moving builds to be deployed automatically to test environments reduced the average time from build completion to testing starting down from an average delay of 1 day between builds completing and tests starting, to the delay being zero, and the tests start as soon as the build completes. This puts tested builds in a ready to release state 1 day earlier on average.
- With this and other efficiency gains from Development, Testing and Release Management tooling since fall 2013 we have decreased the time from when a pull request is created (after code review and testing) to production, from an average of one week to one business day
- Moving translation tests into a service level test instead of part of the functional test run enables them to test in around 1 hour, outside of the current Skynet queue. Previously running the 335 translation tests took 100 minutes (duration), which equates to 50 instance hours (100 minutes on 30 test servers) for each test run that used these tests. This would be every release candidate run (once per day) and any other test runs utilizing translations, reducing costs and increasing throughput in the test queue. This workflow will be replicated for at least 3 other features in the coming months.
- Teams moving off the old flex based test frameworks no longer have to wait in line for an available test server and 'tester' machine, Docker containers are immediately available to run tests and test feedback is available in an order of minutes vs. hours (or even days)
- We have a 20% reduction in server costs for automation since starting this work, this comes in the form of instance costs, data write/deletes etc. as less needless test scenario setup is being done as tests move to a more appropriate place in the testing stack

- Added to this 20% we have achieved a 50% cost reduction overall in 12 months (a saving of around $40,000 per month), some of this through moving tests into test frameworks that test just the service under test, reducing test times and the number of scenarios run for each release. For our flex application, the number of total tests per run has reduced by 15% over a six-month time frame, and with the introduction of new test frameworks and test tools we would expect this to continue at a similar rate over the next year.

One relatively simple example of progress is where previously a team may or may not opt to run our full battery of test against their development builds, and an issue downstream in a consumer of that changes feature does not get exposed. In the previous model, we would 'luckily' catch that error in our Release Candidate test run, and it would have to be reverted from the release and through another round of development then release testing. In the model where each team releases its own updated container for its own codebase, that Release Candidate test run simply not occur, left unchecked that bug gets out into the wild, and breaks the consumers feature, yet with test specifications in place, if a test fails in a consumers tests when the parent is updated, the release is flagged straight away as no good. It does not take another release run of all of our tests to catch that, nor is there an extra risk of it getting into the customers hands broken

The challenge of keeping up with DevOps frictionless development really is centered around working with them and the product teams to see where TestOps can fit in the tools needed for testing without hampering development workflow. So far at Workiva we have had some success with a dramatically changed model for Testing and Release, but have managed to avoid separating a development organization empowered by DevOps tooling and process, from a QA Team using 'TestOps' tooling by ensuring the two are in sync. Ensuring teams are empowered to release at will through coordinated work through DevOps, TestOps, Release Management (and no shortage of available API's) ensures we can go fast and intentionally remove gates in the release process, whilst keeping the level of quality in our products our customers expect, and keep our resources happy and engaged.

Glossary

CI – Continuous Integration
DevOps – Development Operations
Ec2 – Amazon's Elastic Computing virtualization platform
GitHub – Code Repository and Version Control
REST API - A standard for externalizing cloud services to consumers
ROI – Return on investment
SaaS – System as a Service
TestComplete – A Front End Test Automation Application, made by SmartBear
TestExecute – The Test Runner for TestComplete
TestOps – Testing Operations
Skynet – The name given to Workiva's Test Framework Controller
YAML - a human-readable data serialization format

References

Web Sites:

Zellermeyer, Gal, 2013, '0 Bugs Policy'
http://galzellermayer.blogspot.com/2013/05/0-bugs-policy.html