Continuous Everything Handout

Continuous Testing Strategy

Author: Stefan Friese
#
v1.0.0 – Oct 10 2019
#
stefan.friese@code.berlin
stefan.friese@pm.me
#

Introduction

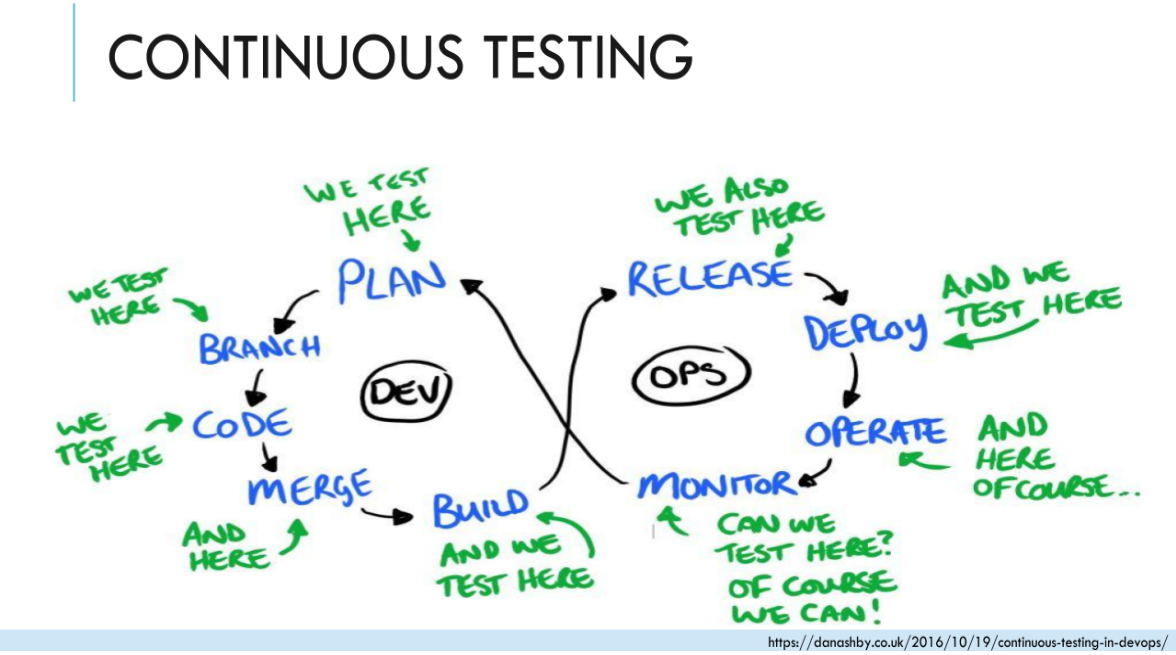
Test automation is a key factor in succeeding with continuous delivery, getting these small frequent changes deployed to production.

As organizations shift to a culture of intense collaboration and rapid delivery, the role of testing is changing. What does testing look like in an environment with automated build and deployment pipelines? Is testing in production feasible? How can testers work together with others effectively to deliver quality software?

To be able to release with both speed and confidence requires having immediate feedback from Continuous Testing.

DEFINITION: CONTINUOUS TESTING

Testing is a cross functional activity that involves the whole team, and should be done continuously throughout the development and delivery process to provide fast, meaningful, and reliable feedback on quality and business risks.



The main goals of Continuous Testing are:

1. Tight feedback loops:
    a. Fast and meaningful feedback.

    b. Get insight into application quality at all times.
2. Support collaboration.

This can be achieved with:

- Close collaboration in test implementation
- Automate tests within CI & CD
- Informative reporting, targeted towards all relevant audiences
- Effective test environment management (as part of CI & CD)
- Continuous improvement
- Testing during and after releases on production

As Continuous Testing is about executing the right set of tests throughout the development and delivery process, the type of tests vary and include:

1. automated checks for known knowns:
    a. unit tests to check that implementation is correct,
    b. acceptance tests to test features,
    c. performance tests,
    d. security scans, and
    e. contract tests
2. automated tests for known unknowns:
    a. shadow release tests,
    b. fuzz (security) tests,
    c. A/B tests, and
    d. fault tolerance experiments
3. manual tests for unknown unknowns:
    a. exploratory tests

Besides testing, monitoring is also important:

1. production monitoring for known unknowns
2. observability for unknown unknowns (e.g. testability aspects)

More information

Jez Humble (2017): Continuous Testing. https://continuousdelivery.com/foundations/test-automation/

Katrina Clokie (2017): A Practical Guide to Testing in DevOps. Leanpub. http://leanpub.com/testingindevops

The Whole Team Approach to Continuous Testing by Lisi Hocke - Test Automation University Course: https://testautomationu.applitools.com/the-whole-team-approach-to-continuous-testing/

Test Automation in DevOps by Lisa Crispin - Test Automation University Course: https://testautomationu.applitools.com/test-automation-in-devops

System under Test

Microservices and webservices

Microservices are small autonomous services that work together. Microservice applications are composed of those services that communicate with each other over standard protocols with well-defined interfaces.

Key aspects of microservices:

1. Functional decomposition: They are responsible for one functionality
2. Independence: They can be deployed and released independently.
3. Loosely coupled
    a. Loosely coupled implies also that micro services shall communicate via APIs and shall not share data (see also: http://www.freshblurbs.com/blog/2015/10/03/dependencies-of-microservices-database.html).
4. Autonomy principle: They are autonomous and truly loosely coupled since each micro service is physically separated from others (i.e. running in its own container).
    a. They shall embed dependencies. If not, the number of integration points or touch points increases massively (see also the service autonomy principle and "autonomy over coordination").
5. Observability
6. Resilience: They isolate failures by acting autonomously (resilience in terms of the reactive manifesto).

Microservices Test Focus

MONOLITH TRAPS

- Application monolith
- Joined at the database
- Monolithic build (rebuild everything; "how complex is the integration test environment?")
- Monolithic ("drop") releases ("indicates high coupling")
- Testing monolith ("avoid heavy E2E tests")

Webservices are software components that communicate with other components using standards-based Web technologies including HTTP and XML-based messaging. They offer an interface describing the collection of operations. RESTful web services are communicating with stateless operations. Web services allow applications to be integrated more easily. However the distributed and loosely coupling make it also challenging to test web services.
The Server APIs are pieces of software you write that others have a runtime dependency on. They are always producers, and might also be a consumer of other services and libraries. These services are often stateless but they may also transform data.

See also:

- https://www.infoq.com/articles/twelve-testing-techniques-microservices-intro/

Webservice Test Focus

Testing of webservices focuses on tests against the APIs of the service as well as integration testing with other services (i.e. server-side integration).
The HTTP status codes have to be tested and also the version compatibility is an important test aspect (lifecycle tests of clients/apps consuming web services, Consumer Driven Contracts / CDC).
Besides the functional tests, non-functional tests are important to determine the scaling behavior (needed for capacity planning). If webservices are already live, tests with real-time API calls can be done. Also, fault tolerance tests should be considered.

Also, it has to be considered if the web service is stateless (RESTful) or stateful. For example for stateful services which are processing (map) data, dedicated data tests may be needed.

Serverless Architecture

The Serverless Architecture is an integration of separate, distributed services. It introduces a lot of simplicity but when it comes to serving business logic, testing has to address some challenges

1. Integration of separate, distributed services: services must be tested both independently, and together.
2. Dependence on internet/cloud services, which are hard to emulate locally.
3. Event-driven, asynchronous workflows are hard to emulate.

Therefore, you shall write your business logic so that it is separate from your FaaS provider (e.g., AWS Lambda), to keep it provider-independent, reusable and more easily testable. When your business logic is written separately from the FaaS provider, write unit tests. Every single FaaS function needs to be treated as a microservice for testing. Additionally, integration tests can verify that the integrations with other services are working correctly.
Based on: https://serverless.com/framework/docs/providers/aws/guide/testing/

Data Streams

Applications that handle data streams have to deal with big amounts of live data. Often, multiple components are chained together to process the data stream.
For data streams, tests shall not only focus on validating software changes. Instead, the data processing has to be validates which includes software and data.

Data Stream Test Focus

One possible test strategy is to generate an artificial data stream: data schema changes must be tested, various patterns and loads of data, and tests shall include also unexpected input data.
Performance testing data streams shall be done for the individual components in isolation. Additionally, the whole process chain may be performance tested with the focus on the throughput and end-to-end latency (and thus, how long it does take until a message is delivered to the consumer).
However, testing with artificial data streams is not sufficient as it is often not possible (or at least not feasible) to build data streams that are truly representative in regards of data volume and content. Therefore, testing in production (TiP) is a reasonable option - in particular by following a dark-release and shadow deployment approach.

Machine Learning

Machine Learning (ML) is software that learns from a given set of data (train data set) and then makes predictions on the new data set based on its learning. In other words, the Machine Learning models are trained with an existing data set in order to make the prediction on a new data set.
Therefore, for ML it is important that the test data set is large enough and representative. Another challenge is that for ML is about predictions which makes it difficult to verify that the model is correct.

Test approaches (based on https://dzone.com/articles/qa-blackbox-testing-for-machine-learning-models):

1. Model performance
2. Metamorphic testing
3. Coverage guided fuzzing
4. Comparison with simplified, linear models
5. Testing with different data slices

SDK & Libraries

Your SDKs and libraries are pieces of software (components) that you publish that may be included by yourself or others as a compile time dependency. Obvious examples of this type include Java or C libraries or property files to be packaged with an application. Libraries are pieces of a larger application that you produce for yourself or others. SDKs are a special class of Library that you provide to interact with your services or programming paradigms. Eventually, libraries may be included in SDKs.

SDK & Library Test Focus

Testing libraries and SDKs require a test driver, i.e. a reference application. It is preferred to execute not only library and SDK level tests as part of CI & CD but rather to trigger also tests of applications integrating the library / SDK updates. Also client/server integration tests should be done to ensure the integration with Server APIs.
This integration and testing strategy has the advantage that the feedback cycle is significantly reduced and that the tests are taken the application specific API usage into account. It is also important that the CI & CD client-side SDK and library tests should be done on the target devices.
In general, SDKs must go through a more rigorous testing strategy than a basic library.

E2E Applications

Your end-applications are pieces of run-time software that other systems or components do not have a run-time dependency on. In our case this is most often desktop applications that consume outside services and present a user-interface, this is the only classification where the component is not a producer.

End-Application Test Focus

If the end applications have an UI, end-to-end tests can be done on frontend level.

Testing applications should not only be done with end-to-end tests. Instead, the integration strategy should define tests for the different software layers.

See also:

- http://googletesting.blogspot.de/2015/04/just-say-no-to-more-end-to-end-tests.html
- http://googletesting.blogspot.de/2012/10/hermetic-servers.html

Application tests have to take the usage scenarios into account, web-based application tests have to be done for the (mostly) used browsers and operating systems and client-side application tests on the target device.

Also client/server integration tests should be done to ensure the integration with Server APIs.

For applications such as webpages and mobile applications, as best practice A/B testing should be done: In A/B testing, two variants, A and B, are presented to the end users and the effectiveness of each variation is measured (as conversion rate).

For mobile applications, besides strong automated testing (in CI&CD), testing shall be also done with actual users:

- Have Alpha (internal) and Beta (external) test users groups to get early feedback on upcoming features.
- A/B testing as controlled experiments on production provide feedback about new features from a subset of users.

Test Types

We distinguish different test types to ensure that we speak the same language and make it possible to group and analyze the test results.

Why shall we align test type terms?

1. Share a common language among testers, developers, POs, and stakeholders.
2. Foundation for aligned test automation, reporting and analytics.

Activity	Definition
Unit testing	Technology facing whitebox test of a self-contained unit of code, i.e. testing a function or method call. Unit testing is the easiest, fastest, and most consistent way for developers to verify their assumptions about the code they produce. • The size of the unit under test is not strictly defined. They shall verify the smallest possible unit of functionality and are typically written at the method - class level. • Unit testing should be done following the FIRST principles: "Fast", "Isolated", "Repeatable", "Self-Verifying", "Timely". • You may need some of your unit tests only when you design software components whereas others can also help to prevent future regressions and allow for refactoring and shall be executed in your CI&CD pipeline. • Unit tests are focusing on individual units and therefore normally executed as solitary tests. This means that test doubles replace dependencies to other components. Sometimes, also sociable unit tests (that are involving collaborators) can be considered. • Unit tests validate intent against implementation and focus on the inner quality of the software. https://blogs.itemis.com/en/unit-tests-are-tests-of-modularity

Acceptance testing	Functional testing with respect to user needs and requirements to verify if the acceptance criteria are met and to determine whether or not to accept the system under test. An acceptance test is a formal description of the behavior of a software product, generally expressed as an example or a usage scenario. Acceptance tests shall test real scenarios and always interact with the system under test the same way a user would. The acceptance tests typically follow the "Given, "When", "Then" pattern. The goals of acceptance testing can be summarized as: • Ensuring that your application delivers the functionality you expect it to. • Providing a human-readable description of the functionality under test. • Incorporating acceptance testing into the agile development process to ensure you have delivered working code and a working application to your specification. • Providing a regression-testing suite, allowing you to add more functionality and ensure you have not broken or changed the previously delivered behavior. • Highlighting issues or bugs in your application, as a user would, before it has gone live to your customers. • Engaging all members of the team in the testing process by allowing even nontechnical personnel to contribute to the description and wording of a feature.
API & Contract Tests	APIs are contracts that enable communication 1. Tests generated from Swagger (OpenAPI) specifications can check for other (i.e. external) consumers, if the HTTP response codes and data types are handled as defined in the specification. 2. Contract tests ensure that the known consumer requirements are satisfied as they verify the integration scenario from the point of view of the consumer. A contract tests an interface such as REST.

Performance testing	With performance tests it is possible to determine if the response time behavior and the system utilization is as expected for a foreseen load scenario. They determine if the performance requirements (defined as SLA or OLA) are met. Performance testing is a type of testing intended to determine the responsiveness, throughput, reliability, and/or scalability of a system under a given workload. It is critical and essential to the success of any software product launch and its maintenance. Performance testing is typically done to help identify bottlenecks in a system, establish a baseline for future testing, support a performance tuning effort, determine compliance with performance goals and requirements, and/or collect other performance-related data to help stakeholders make informed decisions related to the overall quality of the application being tested. What would be the response times for 2000 concurrent users? >> Performance / load testing What happens under excessive load? >> Stress testing Are there any memory leaks? >> Endurance/soak testing (load test with prolonged test duration) Where are performance bottlenecks? >> Profiling, performance analysis What is the best configuration for the target platform? >> Performance Tuning How many VMs do we need for 1000 new routing requests / sec? >> Scalability Tests ("Max QPS test") Performance Test Types: 1. Load test is a "classic" performance test, where the application is loaded up to the target concurrency but usually not further. Load/performance testing is therefore the closest approximation of real application use. 2. Stress tests increase the load further and are focusing on critical load scenarios. A stress test causes the application or the infrastructure to fail. The purpose is to determine the upper limits or sizing of the software/system under test. 3. Endurance tests (also named soak tests) are executed to identify problems that may appear only after an extended period of time. As one example, memory leaks can be detected with endurance tests. 4. Baseline performance tests are done to establish a point of comparison for further test runs.

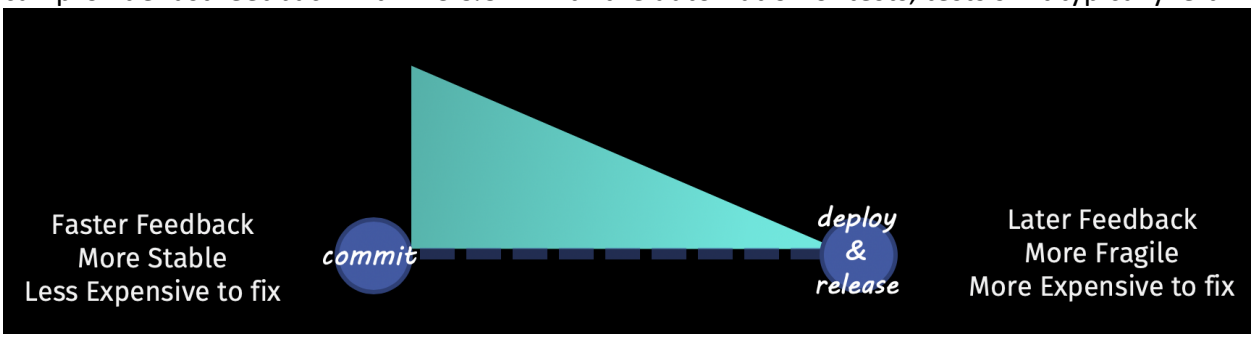
	5. Performance Profiling: Task of analyzing, e.g., identifying performance bottlenecks based on generated emtrics, and tuning the performance of a software component or system using tools.
Fault tolerance testing	Fault tolerance testing (aka resilience testing) is typically done by intentionally creating controlled failures to determine if the system under test is still able to operate under those failure conditions.
Security testing	Non-functional test to determine if the application is secured by revealing flaws in the security mechanisms and finding vulnerabilities. A vulnerability is a weakness which can be exploit by a Threat Actor, such as an attacker, to perform unauthorized actions. Security testing is typically looking at authentication, authorization, confidentiality and data protection, integrity, resilience. Common security test types are penetration, robustness and fuzzing testing.

Continuous Testing - Engagement

Continuous Testing Strategies

Shift Left

Continuous Testing is not only about test automation but automate tests is crucial so that tests can provide fast feedback within CI&CD. With the automation of tests, tests shift typically left:



Katrina Clokie (2017, p. 41):
"Unit tests, integration tests, and end-to-end tests are all examples of the usual automated tests that a development team might create. They focus on confirming behaviour of the software at different levels of abstraction - from individual components through to the entire solution"

Ensure fast and meaningful feedback by executing tests in earliest test stage and at the lowest integration level possible: Development teams must have automated unit tests, complemented by acceptance and performance tests and eventually a small number of automated E2E tests:

1. The test environment must fulfill test execution requirements: Test environment shall be comparable to production, i.e. it should be provisioned automatically and comparable to the upgrade scenario on production.
2. It may be not necessary to have 100% unit test code coverage. A general rule of thumb is 80-90%. But more important is that the unit tests that exist must be rapid, reliable and relevant. This means a unit test is responsible for a single function result, it returns a result fast in a consistent and reliable way.
3. A comprehensive set of acceptance tests must be executed within CI & CD. Preferably, Acceptance Test Driven Development (ATDD) is practiced.
4. Automated performance tests have to be integrated in CI & CD pipelines.
5. Tests results shall be gathered in a framework such as TAO, providing a single entry point and access to the data for detailed analysis.

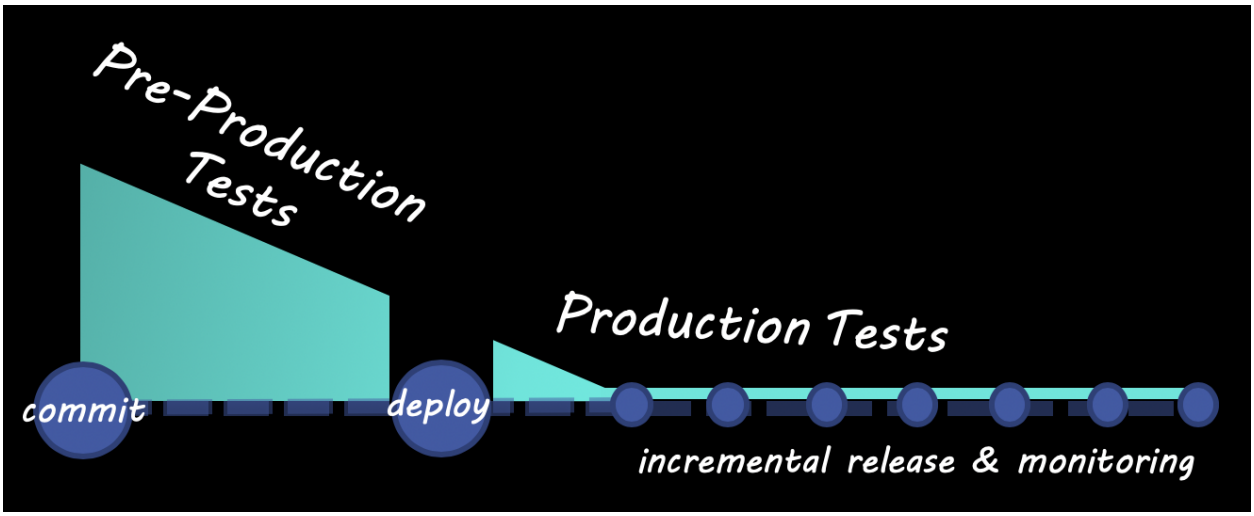
Shift Right

If a development team has already progressed to a high degree of Continuous Delivery, teams may consider production tests.
There are different types of production tests (aka TiP / Test in Production):

1. Shadow deployment tests with real-time production traffic, prior to release. Note that the use of live data for testing may only be permitted where adequate controls for the security of the data are in place.
2. Functional tests, i.e. as semantic monitoring.
3. Introducing controlled experiments on production (e.g. A/B testing as data-driven experiments for new features, Chaos Monkey fault tolerance and resiliency tests).

Test in production is only possible if the way of working supports experimentation and exploration:

1. Everyone must react system, everyone must react when a build is no longer green and teams must ensure that the pipeline is maintained in a stable state.
2. A comprehensive test suite must be actively maintained for any release to customer or production.
3. Continuous Improvement to review test cases and improve efficiency and effectiveness.

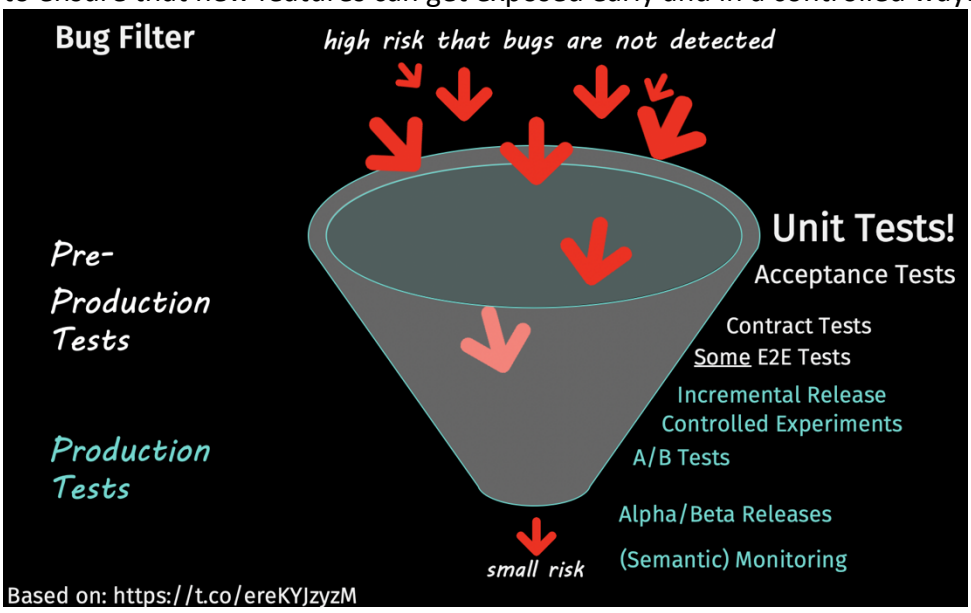


In contrast to automated tests, the test execution and analysis of the tests that are executed as experiments is not fully automated. For example, A/B testing is done with a high degree of automation but the final analysis of the results is a manual step.

Experiments are crucial for Hypothesis driven Development, by ensuring that you measure the impact of the changes and adapt accordingly.
With controlled experiments, you get feedback from customers, using your application in real world conditions and thus you are able to learn and adapt fast .

Shift Left and Right

Continuous Testing means executing the right set of tests continuously throughout the development and delivery process to provide fast, meaningful, and reliable feedback on quality and business risks. Therefore, besides a shift left, also a shift right shall be considered in order to ensure that new features can get exposed early and in a controlled way:



Continuous Testing Way of Working

1. Whole Team Approach: The whole team is responsible for the quality of its product. Therefore, testing is a collaborative practice that occurs continuously and focuses on building quality into the product and uses fast feedback loops to validate our understanding. See also: https://agiletestingfellow.com/
2. Teams must ensure that there is a stable mainline and that it is maintained in a stable state. Everyone must react when a build is no longer green.
3. Test Driven Development (TDD) shall be practiced.
4. All tests, required for the release, are automated and owned by the product development team: Testing is not an isolated activity done in a separate silo but owned by a cross-functional team.

Don't execute manual pre-scripted tests, instead focus on test automation, supplemented by exploring: "Testers can be extremely valuable to explore our applications in unexpected ways, ways that just a human could do. But testers should not waste their time executing test plans that could be automated by the development team." (Sandro Mancuso 2014).

1. The manual tests that are still needed must not slow down the continuous delivery pipeline and must not be mandatory for the release decision.
2. Manual tests should focus on exploring the system under test instead of following a manual.
3. Explore early since fast feedback is important and to support that the team can adapt and learn fast.

Exploratory tests and experiments should be done by a cross-functional product development team and therefore during development (i.e. exploratory tests) and as part of the release (e.g. canary release).

For more information about organizational best practices for testing, see also: Riley, Chris (2017)

Experiments and exploratory tests must be executed in small batches: A manual test is a queue - queues delay feedback from the downstream processes, and delayed feedback leads to higher costs and reduced quality. Testing related queues are caused by limited capacity and variable input frequency. Testing also has queues, because tests are often done with large batch sizes. If you are executing manual tests, execute them as soon as possible (i.e. during development) and in small batches.

Further information

- Chris Riley (July 25, 2017): Organizational Best Practices for Testing Success. https://saucelabs.com/blog/organizational-best-practices-for-testing-success
- **Dave Farley (2019): Acceptance Testing for Continuous Delivery.** https://www.infoq.com/presentations/acceptance-testing-continuous-delivery
- **Katrina Clokie (2017): A Practical Guide to Testing in DevOps**. Leanpub. http://leanpub.com/testingindevops
- Sandro Mancuso (2014): The Software Craftsman: Professionalism, Pragmatism, Pride.
- Stefan Friese (2018): Architecturally Aligned Testing. https://www.infoq.com/articles/architecturally-aligned-testing