Continuous Testing - from Trend to Pattern

Why the future belongs to automated tests
Why the future belongs to automated tests

What’s all this again? Do I really need it? It’s just going to cost me time and money again?! Can’t everything just stay the way it is? Lots of people repeatedly ask themselves such questions whenever there is a new trend and companies consider whether to adopt this trend or not. In the following, we are going to explain what Continuous Testing (CT) is, how to exploit the underlying concept to best advantage and why the automation of tests with CT is far more than just a trend.

What exactly is Continuous Testing and what are its advantages?

Put very simply, Continuous Testing is the integration of automated tests in the continuous software delivery process. Even multiple integrations of automated tests within a process, like after particular steps of a software delivery process, are not uncommon.

In principle, all kinds of tests are suitable for use: both functional tests, such as unit tests, integration tests, API tests, end-to-end tests or also layout tests along with non-functional tests, such as load and performance tests, security tests or usability tests. However, it usually makes no sense to carry out every test available after each step in the delivery process. In other words, there is no recipe or clearly formulated step-for-step instruction on how Continuous Testing is to be used or implemented. Continuous Testing is more a kind of pattern for Continuous Testing with short feedback loops. The basis here is that one always has to consider how automated tests can be used most effectively, measured in relation to the cost/benefit and according to their statement content for each test kind for its special environment, software, processes, infrastructure, etc. It is enormously important in this respect that all parties concerned consider and define in advance where and when what tests make sense. This can only be decided by everyone together, since different people have different roles as well as different views and priorities.
Consistency guarantees security and quality

The advantage of automated tests is that they only have to be implemented once, usually run faster than tests conducted manually and always guarantee an exactly identical process. Continuous Testing in particular provides these advantages. Automated tests, once implemented, are carried out repeatedly after each change in the existing software delivery processes. At first glance, it might appear that processes would be unnecessarily delayed if tests run repeatedly and as a result delay the software from going live. In pure time terms, this may be true – carrying out tests naturally takes time. But it is worth accepting this time expenditure, since:

- CT facilitates fast feedback about the correct sequence of my software delivery processes
  - through feedback on every single step for which automated tests are integrated downstream.
  - by avoiding waiting times until the entire process has been completed.
- CT saves manual effort, especially in the area of testing and debugging.
- CT reduces the risk of going live with defective software and configurations.
- CT facilitates better research into the causes of problems (defect analysis), since the software delivery process is immediately interrupted if tests fail.
- CT is the basis for continuous delivery processes from dev to ops.

In short, Continuous Testing provides the entire team with more security and a good feeling for successful deliveries.

How did the Continuous Testing approach come about?

In terms of its basic idea, Continuous Testing is really a very simple approach. As is so often the case, the complexity lies in the detail and execution. To find out how this approach came about, it makes sense to make the development of software delivery processes clear.
"Previously" (see Figure 1), software used to be developed according to the waterfall model. This means software was first planned and the requirements documented in writing. After this, development took place and the testing came right at the end for everything. Usually, there were only a few releases a year. Deployments in the test environment (stage) and live applications were frequently still associated with a great deal of manual effort and the tests that were carried out were to a great extent of a manual and frequently exploratory nature.

Checklists for tests were also typical, which were worked through repeatedly from point 1 to point n after each release. At that time, the subject of unit tests was already current, but had not yet been perfected; integration tests or API tests were more of the exception. However, the subject of end-to-end tests (user acceptance tests) was already in the air at the time and received further impetus in the web sphere through the launch of Selenium. However, these tests also proved to be very prone to errors at that time and comparatively slow and unresponsive. There were in part also burden or performance tests, which were carried out downstream. The motto here often was: "The user is our best tester and will provide us with feedback."
Thanks to the growing popularity of agile approaches and methods, testing and the mindset concerning the software delivery process has also adapted and undergone further development.

Figure 2: Software delivery process (today) © 2019 ASERVO Software GmbH

... today

Concepts such as "time to market" and "potentially shippable" resulted in deployments being automated for the most part in the form of CI/CD pipelines. As a result, the time cycle of releases was greatly reduced. However, it is often still the case that large deployments of large software components being rolled out.

There has also been a lot of optimisation in the field of testing. The importance of unit tests has continued to increase. Integration tests and API tests have increased in value and are more frequently automated. There still are many end-to-end tests. All of these tests have now been integrated in automatic deployment processes. End-to-end tests in part still run as downstream processes, since they continue to be more prone to errors and slower than unit or integration tests. Even non-functional tests, such as burden or performance tests, have in part already been integrated. (Figure 2)
... and in future

Consequently, the first step towards Continuous Testing has already been taken. And how will the current model now become a Continuous Testing model? Actually, the answer is quite simple: Divide your software into services / microservices and guarantee that each service can be provided independently of all the others via a deployment process. If necessary and appropriate, each deployment step of each service is checked for correctness in this connection by suitable automated tests to be defined.

In theory, this sounds very simple indeed, but the details require a great deal of effort in order to be implemented. Each service should have its own CI/CD pipeline for this. To avoid redundancies in the production and maintenance of the pipelines and obtain a uniform pipeline structure, a pipeline template should be defined, which then uses each service and implements and if necessary shapes the services (for graphic clarification, see Figure 3).
In this way, automated tests, such as unit tests, integration and API tests, end-to-end tests, burden and performance tests or whatever kind of test can be carried out for each deployment of each defined service. It is always necessary in each case to define which tests are necessary or make sense and when.

Particularly with regard to the error-prone and very slow end-to-end tests, the following questions need to be clarified to create the correct dimension of tests for the deployment pipelines:

- Does a smoke test suite make sense for the end-to-end tests and do I want to carry these out instead of end-to-end tests?
- Is it necessary to carry out end-to-end tests in each service, in part perhaps several times during the pipeline process?
- What end-to-end tests are really important for integration in my service deployment pipeline?
- Is the end-to-end testing as a kind of integration test of several services enough?
- Is the separate performance of end-to-end testing via an own process perhaps sensible or helpful in order to get regular feedback?

A how do I now start with the implementation?

//

The subject matter at first appears to be very theoretical. But to explain the subject in more detail in practical terms, we will make use of an example to provide an introduction to Continuous Testing. Figure 4 provides a graphic illustration of the potential procedure in implementation.

Let us assume, a company has decided to deploy Continuous Testing. It cannot find the resources or skills needed for this within its own workforce and obtains the corresponding know-how from consultants, who support its own team.

The point of departure is characterised by web-based software, consisting of a front-end platform and back-end platform. Both back-end as well as front-end are based on a service, which is in turn based on REST. Other functional web services / microservices are provided, for control by the front-end or via an API (likewise via REST). On the one hand, there are manual tests that run via the web GUI of the application. On the other, the automated end-to-end tests are carried out locally at intervals on a developer machine. There are no interface tests and the conduct of existing automated tests in Jenkins is neither implemented nor planned.
The solution proposed includes the following features:

- Front-end, back-end and the services must be provided separately via their own deployment pipelines.
- In this case, it is defined from case to case which tests are integrated and as a result implemented in each run.
- In addition, there is a separate process for conducting end-to-end testing. These are carried out daily in different target system configurations consisting of operating systems and browsers with the help of a Continuous Testing cloud.

As a result, continuous, integrated and automated testing arises as added value in all pipelines with the quality of each pipeline and deployment object being continuously tested. Consequently, corresponding reporting and if necessary dependencies between deployment pipelines prevent known and discovered defects from being rolled out. In addition, intervention through manual testing activities and the associated effort is avoided and deployment in a theoretical 24/7 rhythm facilitated.

The specific distribution in independent (micro-) services including the creation of deployment pipelines and integration of tests in this process is certainly associated with time expenditure and cannot be realised from one day to the next. Continuous Testing is just a fraction of the overall continuous life cycle. But the result of successful implementation is worthwhile.

Figure 4: Example of practical CT implementation planning

© 2019 ASERVO Software GmbH
What else? ...conclusion
/

Continuous Testing is a very helpful concept to secure and even boost the quality of software and the delivery processes. In this respect, Continuous Testing is also undoubtedly more than just a trend, which could quickly establish itself as a constant or pattern in the continuous integration environment.

Continuous Testing as part of continuous deployment / delivery is and remains a fascinating subject. In future, companies will hardly be able to dispense with this subject in order to ensure high quality in the provision of software in a quantitative time cycle. In an era where everything has to be delivered as quickly as possible, quality should not and may not suffer as a result. And for this, Continuous Testing is exactly the right approach.

8 tips for implementing Continuous Testing

- Decide from service to service which tests make sense and are sufficient for them to be integrated in the deployment process.
- Carry out automatic tests after each possible step (in each deployment stage).
- Do not carry out all existing tests in each deployment process of each service.
- Try to reduce end-to-end testing or exclude it from ancillary processes.
- Motivate and teach others to develop and integrate automated tests to distribute the understanding and knowledge (think like a DevOps).
- Only expand your end-to-end testing where necessary and maintain these tests with the objective that they are stable, reliable and robust.
- Concentrate on the essential:
  - The supreme objective is the clean deployment of the services.
  - Integrated tests should provide support here, but not be the focus.
- Share and discuss your experiences, problems and solutions on continuous testing with others.