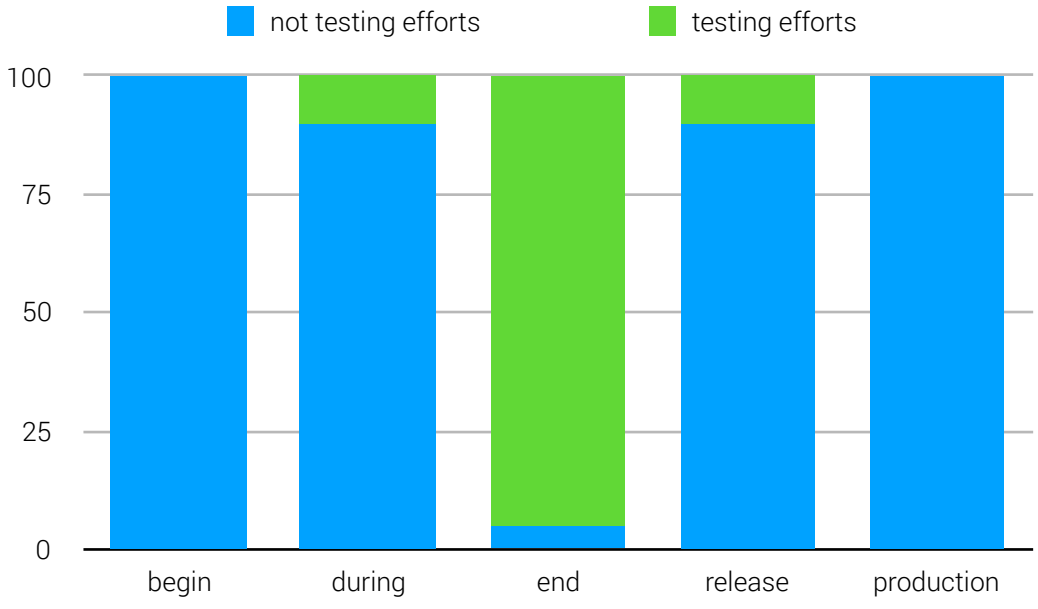
Testing Approach Comparison in the development cycle

DISCLAIMER: THE GRAPHS AND IT'S NUMBERS ARE A REPRESENTATION OF MY PERSONAL OPINION BASED ON MY EXPERIENCE.

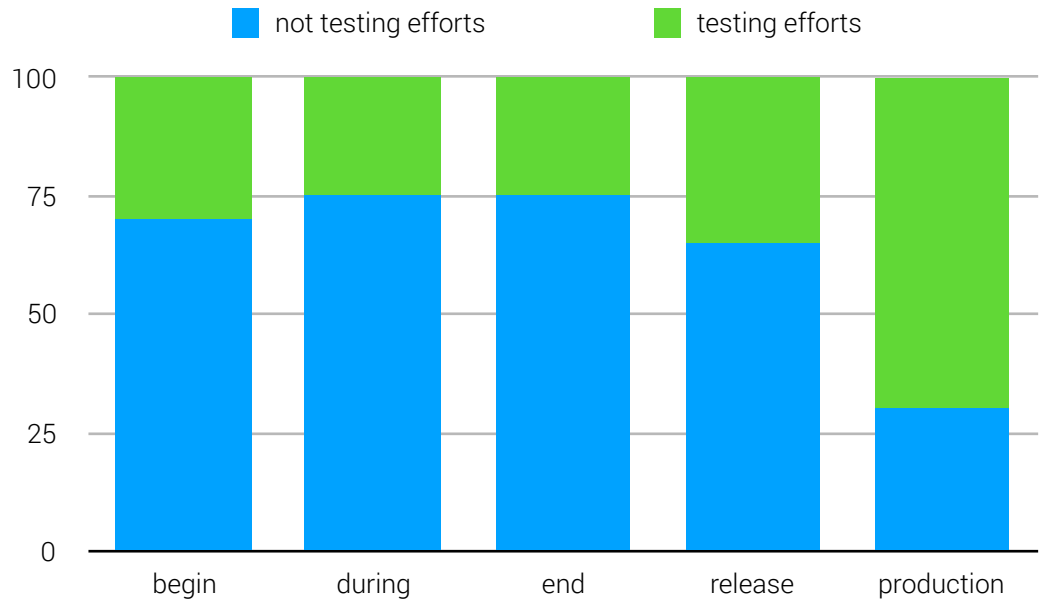
Classical Testing

Classically our testing efforts focus on the time relatively short before the release. "Relatively short" refers to the whole development cycle. In a scrum sprint of two weeks this could be the last couple of days. Often consisting of a large scale system test and sometimes relying on manual regression tests.



Continuous Testing

Here our testing approach is following the idea of shift left as well as shift right. Testing is an inherent part of every step in our development cycle. During requirements engineering we test by explicitly and vocally discuss the technical and business impacts of upcoming features. During the implementation phase we integrate testing by adopting test driven development, pair programming, code reviews as well as continuous integration measures (trunk based development, “fail fast/fail often”).
It is key to notice that there is no significant difference in testing efforts nearing the release. This results from being able to have a functioning release at any point in time during implementation. Keeping in mind this is strictly a visualisation in relative efforts, our testing work should increase during the release. This is achieved by applying appropriate deployment strategies such as canary releases or staged rollout. In addition to a more secure release, this allows for zero downtime deployments and fast recovery in case of failure.
Classically testing in production is a no go. When faced with a highly complex software landscape testing in a productive environment might be a definite requirement to assure quality and reliability. Extremely significant is having exhaustive observability in place. Combined with the continuous state of deployability, the quick and painless deployment and the immediate feedback from production we can focus on reducing our mean time to repair (MTTR). This is a big factor which differs continuous testing from the classical testing in which we focused on increasing our mean time to failure (MTTF). Especially now that with cloud computing and complex systems failure is inevitable: the question is not “will it fail at all?” but rather “when will it fail?” and “can we recover quickly?”.



Preventive vs Reactive Measures

We can differentiate our testing efforts in two groups: preventive and reactive.
In the classical testing world we applied almost exclusively preventive measures. Typical actions include elaborating our business cases, performing a risk analysis, trying to predict our end users actions as well as their expectations on how our software is supposed to work. From that we derive happy, negative, exception cases which we then execute before any change is impacting our end users. We try to cover the best grounds by predicting what is supposedly going to happen on production. This strategy boils down to increasing the mean time to failure (MTTF).
With smaller increments and faster rollout cycles we have a hard time following the same principle. It is simply not economically feasible to execute all of our preventive testing efforts on every deliverable. So we apply a twofold solution. On the one hand we reduce our preventive testing efforts, on the other hand we introduce reactive testing efforts. The difference being that preventive testing efforts are applied on the change before it is affecting end users while reactive are executed when a change is already affecting end users.
Thus reactive measures can only happen in your productive environment (by definition, as the production environment is the one your end users are using). They can uncover issues you would have never anticipated, especially corner and exception cases. Representative methods are Beta Testing, A/B Testing and Monitoring as Testing. As they are in production we should aim to recover from those issues as fast as possible, so our general strategy shifts towards reducing mean time to recovery (MTTR). This is one reason why we need to reduce our preventive testing efforts. Here we should primarily test our core business cases; those which are essential to our value chain. That means corner and exception business cases are not explicitly tested and only uncovered reactively. On a fast and cheap unit test level we obviously don't skip them.
Manual testing is almost exclusively exploratory and loosely coupled to the actual feature release cycle. It is executed on test environments as well as on production.