When Not to Test

There are reasons for dropping tests .
Be aware of the reasons and the associated risks.

Testing is an integral part of any software development process. After an initial test run during development there is usually a repeated test procedure in place. Ideally, it is fully automated and runs on a regular schedule. However, the scope of these tests rarely covers 100% of all functionality. There are always some parts that are not tested. What are those parts? When should you stop testing? How do you identify the components, where testing might not be worth the cost?

No Change

The main origin of software defects are unintended side effects of routine code changes. However, what about code that is not changing? The claim that all software and all its components are always evolving is just false. In our testup.io stack we have a number of services that run almost unchanged for years. One such service collects usage statistics using SQL and another translates security tokens from one system to another. No test plan has ever been created for them. They just continue to work.

Of course it is sometimes hard to predict whether a component will evolve or not. Adding tests later can be much more difficult than doing it straight away, but it might waste resources if that test is never needed. Pretending that no change is expected can be an easy excuse for avoiding the immediate extra work. That is why this argument might be misused by many. Just because others make such a mistake doesn't mean that you can't do it right. If no change is expected don't plan regular tests.

External Dependencies

Some of our services are actually thin wrappers around external systems. Examples are providers that check kubernetes for the state of simulated devices, check cloud APIs for the state of virtual machines and additional cloud resources that are needed to run a specific device configuration. The major task of such components is to translate an external state into a unified internal one. Their actual logic is fairly thin.

Because this external system is not easily available when running tests there are three common strategies to test in such a scenario.

- Mock the external system
- Create a test environment of the external system
- Test in production

None of these strategies leads to 100% coverage. The mock is a look-alike for the external system. It is supposed to behave like the real system within the tested scope. However, they do not discover problems originating from 3rd party regressions or deviations from their original specification. Test environments can be expensive to set up. Their configuration is difficult to match the production version within all relevant parameters. Testing in production is certainly an option, but always comes with the risk of breaking real assets and therefore can only have limited scope.

The complexity of tests in these discussed scenarios might quickly generate costs that outgrow their benefits. Of course, this is not good news. The lack of testability creates bigger problems that need to be mitigated elsewhere. In our case we had to build complex monitoring strategies, that permanently check the system's integrity. Since the affected components are just known to produce more production issues than others, we can only compensate by speeding up our response times. To summarize, if testing is too expensive then invest in monitoring instead. If errors can't be avoided speed up your recovery times.

Heavy change

Many components are under heavy development. Their interfaces, their actual use case and their architecture may change constantly. Writing tests for each version of an evolving software can just slow down the process without benefit. A test would be outdated even before it becomes useful. Usually, this is a temporary state. It doesn't mean that tests are not important. Testing just happens elsewhere. A feature may be tested in practice by real users. Sometimes AB tests roll out a feature just to get feedback on user acceptance. The chance of the added feature being dropped can be substantial.

In pursuit of rapid user feedback and real world results we might want to delay the installation of planned tests for a brief amount of time. The danger, of course, is that tests are forgotten eventually. This strategy creates what is known as technical debt in software development. If tests are missing you may live under the assumption that the feature is complete, when in fact later problems just become more costly to fix. Just like real debt interest is accruing. Hold business accountable. If tests are dropped for speed the resulting costs need to be transparently communicated.

Low value

let's face it, sometimes we just don't care. Ask yourself, is it you who doesn't care, or is it your company. If the answer is the former you may want to change your attitude. In case of the latter you may want to kill the feature and start appreciating the value of simplicity. Some of your users may have already started depending on a low value feature, not because it's of great use, but because it is there. Its failure may cause issues in a larger process where the original feature might have only played an subordinate role. Hence, if testing creates more costs than a feature adds value consider dropping that feature early rather than late.

Summary

Regression testing means risk mitigation. It never adds immediate value. Every new feature most likely passed some sort of initial test run or an acceptance test. These features would just be rolled out at the normal pace. However, in the absence of testing this would lead to frequent und unexpected failure of old features. This delayed reward makes testing prone to procrastination. There are many excuses to postpone or completely skip the test development. As with every excuse they each have a core of truth in them. Very often it is just the better option not to test something periodically. Listing the valid reasons may help to distinguish the excuse from the real argument.