AI4T – Advanced Intelligence for Testing

From research to reality

nagarro

Funded by the Austrian Research Promotion Agency (FFG), a team of Nagarro automation and AI experts has been researching the vision of a highly automated test life cycle. Their goal is to make QA-related processes more comprehensive, scalable, faster, and above all, more intelligent.

In software engineering, the subject of software testing has become an integral part of development processes. The automation of tests has also proven itself and contributes to shorter and more efficient development cycles. Many teams already integrate automated tests in their development pipelines and execute test cases at several test levels when checking-in the new or changed source code. This makes it possible to detect errors more quickly and, in the best case, prevent them from making their way into the common code repository in the first place. From a quality assurance perspective, you might think that we have achieved a satisfactory level of quality, right? Well, think again.

Day-to-day work in real life

For most software developers not just grey theory

Unfortunately, the reality is, often, quite different. As the following scenario exemplifies, the painstakingly achieved progress brings additional challenges, including those that had not been considered previously:

Tom, our UI developer, has implemented the new designs of our log-in page for our web application. After his commit, the check-in tests failed. Since Tom wants to finish the task before sprinting off to his family, he accepts the fact and checks-in the code. Revising the test will be the first thing to do tomorrow morning, he thinks. The next morning, Bernd, our back-end developer, finds out that about 40 test cases have failed in the nightly test run.

A quick analysis of the log files

It takes two hours of painstaking analysis of the log files to discover the root cause of the problem. This is followed by a serious conversation between Bernd and Tom. Bernd believes he has finally updated all relevant test cases. To be on the safe side, he starts the regression and system tests.

Now he must wait for the test run

Since an extensive test set has already been implemented, the test run takes 10 hours. Therefore, Bernd cannot get to know whether everything works, until the next day. The following morning, he realizes that some of the tests failed overnight again. Once again, the analysis becomes complicated and this time, it takes the entire morning to get completed. The result is a previously undetected malfunction of the legacy back-end component.

Why is the problem in the legacy code of all things?

In this case, the problem concerns that one legacy component. It is hardly documented and tested, and its developers are, of course, not available anymore. The component should have been transformed or migrated long ago. This will now become a very lengthy task.

Only the tests themselves are automated

For most software developers, the above-mentioned situation is not just grey theory. Test automation undoubtedly helps to increase software quality. Without it, the error in the back-end component (in our previously described scenario) would have remained undetected. Nevertheless, many processes around test automation are still manual. To master these and similar challenges in the future, our vision is to push automation across most of the test process. Our goal is not to replace humans but to get rid of most of the boring and repetitive tasks and to focus on improving the overall quality of the application even further.

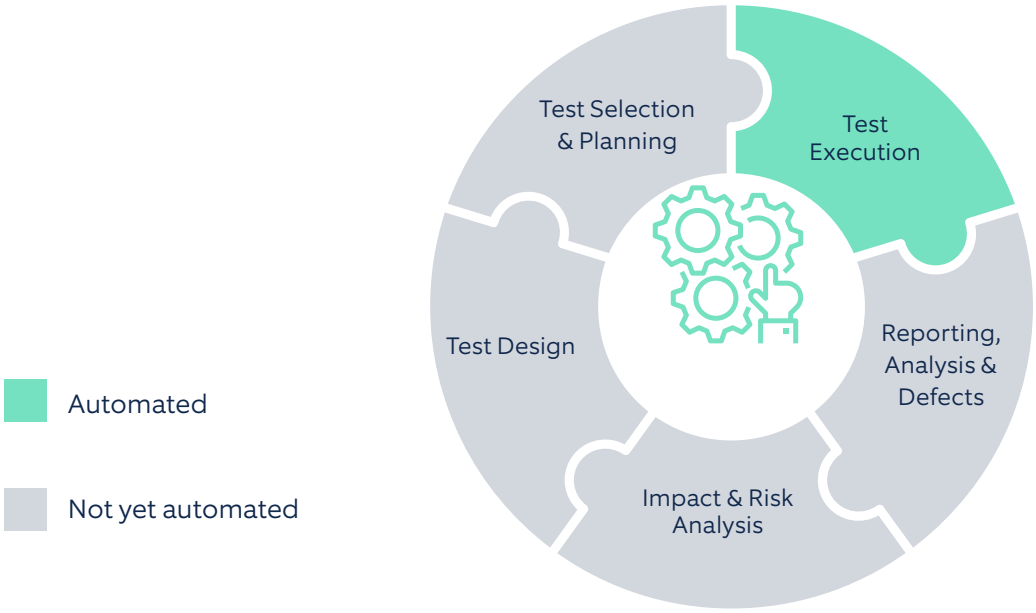

Fig. 1: Automated test execution is only a small part of the test life cycle. The remaining steps are mostly still manual and recurring tasks .

The vision: A highly automated test life cycle

In 2019, Nagarro collaborated with the Vienna University of Applied Sciences to launch a research project called “AI4T - Advanced Intelligence for Testing”, funded by the Austrian Research Promotion Agency (FFG). Managed by Nagarro’s Accelerated Quality and Test Engineering global business unit, the project aims to use machine learning and artificial intelligence to move towards the vision of a highly automated test life cycle.

In the very first year of the research project, four approaches have been developed that could support Tom and Bernd, the protagonists in our story (and of course, many testers & software developers) in their daily work. Let us discuss each use case in detail.

1. Smart Test Selection

Automated test cases are directly proportional to the complexity and scope of applications and systems. Higher the complexity and scope of applications and systems, higher the number of automated test cases. A complete test run of the full test set easily takes a few hours and is, therefore, too time-consuming for every small change or iteration.

A few years ago, this was still a luxury problem. But today, we are experiencing such situations much more often. A subset of these tests is usually performed as smoke tests before each commit to the master branch. The challenge is to select the relevant tests from the pool to find the right balance between the test coverage, significance of the results, and the test throughput time. However, if only a few tests are performed, there is a risk of overlooking errors. If the test takes too long, this leads to unwanted waiting time.

Limited runtime and still detect all errors

Smart Test Selection answers which subset of the test cases must be executed within a limited runtime. These selected test cases should detect all errors which would occur in a full run of the test suite. How do we achieve this?

AI methods are used to analyze the characteristics of the past test runs, and a subset of the test cases is selected from this data as per the following considerations:

- Test cases that have frequently failed in the past are selected more frequently.
- Test cases that have not been executed for a long time are more likely to be selected.
- Test cases that are relevant to the changes at hand are more likely to be selected.
- Test cases that have a long runtime are less likely to be selected.

The selected subset of test cases must have a full test coverage in the target area that is similar to the complete test suite. Other metrics such as the age of the test case, frequency of changes to the test case, the validity of the test case, and more can also be considered.

These factors are weighed by using intelligent models, and a subset of test cases is selected. With this framework, suitable selections of test cases can now be chosen for a short smoke test as well as for a detailed but time-limited nightly run.

In this procedure, sometimes it is important to also prioritize tests that have not been executed in the test runs in a while, even if they are not very relevant to the current situation. This will ensure that these test cases do not disappear from the radar over time and that there are no test gaps.

2. Self-Healing Test Automation

The importance of regular maintenance and service effort required in automated test cases is often underestimated. Modifications to the code and changes to the runtime environment, external interfaces, or the system configuration can influence previously automated test cases. For example, renaming a GUI button or a well-intentioned correction in the code quickly leads to an apparent error. It is only after thorough analysis, that it becomes evident later that this was a "false positive" due to it being a test case that no longer matches the code and not an error in the product itself.

Automatically adapt test cases in the source code in case of changes.

Self-Healing Test Automation deals with the question of how test cases can be (semi-)automatically adapted in case of changes in the source code. Thus, the test cases are kept in sync and maintained automatically, even as the manual effort of adapting the test cases is significantly reduced. Simple UI changes can often even be changed completely automatically. For more complex changes, show the programmer the changes and have him/her confirm them.

The process flow of a self-healing solution, which corrects for example: UI components, is as follows:

- **Monitor:**
  The test flow is monitored and data about the test cases and UI components is continuously collected.
- **Detect:**
  If a test case fails, it is detected immediately.
- **Analyse:**
  The analysis starts to see if the cause of the error can be corrected automatically. Intelligent algorithms are used to determine whether the error is caused by a moved and/or renamed UI element.
- **Heal:**
  If this is the case, the new element identifiers are automatically transferred to the test case and saved for future runs as well. Of course, the tester is informed of that change to be able to audit it or reject it post-hoc.

Technologically, this problem can be solved with AI-based methods such as Graph Matching. This involves comparing the original UI graph with the new UI graph. The most likely UI element in the new graph is selected based on the position, the name or the text descriptions. If several similarly probable UI components come into question, the user is notified to choose from the various options.

3. Automatic Fail Analysis

Large software applications produce multiple log files with many entries so that in case of an error, its exact cause can be traced. However, this process is quite time-consuming and often, it is difficult to find the actual root cause.

Many programmers use simple heuristics such as searching for terms like 'error' or 'exception'. The real reason for the error is often not directly visible from the error message but is hidden in earlier log entries. When test cases fail, it may well be that only a few log entries really point to an error and many others only happen because of subsequent errors. Another reason could be that there are already known errors that occur repeatedly due to further development. This is where Intelligent Fail Analysis can help.

Extract the actual root cause of errors from log entries

Intelligent Fail Analysis deals with methods to extract the actual root cause of the error from log entries. For this purpose, we can leverage machine learning, which provides many algorithms and procedures to do so.

In our AI4T research project, we investigated clustering failed test execution results. The idea behind this is that development teams classify failed test runs into meaningful categories, based on the log entries produced (either by the test automation or by the target systems or both). The choice of categories depends a lot on the application and use case.

Categorization can be done, for example, by areas of the application or by layers (e.g. database errors, UI errors) or by the importance of the error. After a period of manual annotation by developers, these categories are then automatically assigned by a system using machine learning. Targeted feedback from the developers on the suggestions can help AI learn more with each additional test run. Over time, this assignment is improved continuously.

Therefore, only previously unencountered failure types and representatives for the individual categories need to be manually investigated, thus reducing the testing effort considerably. The effort can be further reduced by a smart visualization and targeted aggregation of information based on the error type. For instance, in the case of a database-related failure, the tester can be immediately shown log files for related interfaces without having to first search for them manually.

4. Smart Refactoring

Improve code quality and increase code maintainability

From simply being an initial idea to becoming part of the application or system, it is common for prototypes and temporary solutions to unintentionally find their way into the finished product. Sometimes, as victims of their success, they remain there for eternity. The resulting 'impure' code is then carried through the years as "technical debt". The emergence of new technologies, methodologies, and paradigms makes legacy code more painful, as new paradigms need to be introduced without breaking existing functionality. Although workarounds, adapters, or the next middleware promise quick and easy support, these ultimately contribute to a fragmented ecosystem, increase the complexity of the overall system, and reduce its maintainability at the same time.

Smart refactoring is defined as a process that changes software code, that does not alter its external functionality and improves the internal structure. This improvement targets certain aspects of code quality directly to increase the maintainability and testability of the code.

Traditional Smart Refactoring analyzes the source code by using heuristic methods and applies established design patterns to restructure it. Machine learning methods can help in this process at many points. One approach, for example, is to use machine learning to learn from hundreds of open source projects when such design patterns are applied in practice. In this way, we can create a machine learning model that only suggests changes that have been implemented similarly in the same situations by experienced programmers in numerous open-source projects.

.

First successes and promising prospects

Focus on complex issues that are more valuable and also more fun!

Coming back to our original scenario, the four AI4T use cases described above can be applied to our example of Tom and Bernd: Bernd would not have had to wait 10 hours to get his test results – with the help of Smart Test Selection. It would also have helped Tom if his redesign of the UI had been automatically recognized and if the test cases could have adapted themselves to the new components. Thanks to Intelligent Fail Analysis, it would have been immediately clear that it was a UI error and one of them had to take care of it. And finally, Smart Refactoring could have avoided the scenario ending with a legacy component that would have made trouble-shooting extremely difficult. The benefits of AI4T are clear: lesser time and increased productivity for Tom, Bernd, and their team!

The results of the research project, after only its first year, show that many aspects around the test lifecycle can be automated in a meaningful way. We will never be able to replace humans in testing, and this is not even the goal of AI4T. However, it shows that by using intelligent technologies, many recurring tasks can be (partially) automated. This gives us that much extra time to focus on the other complex issues that are more valuable, and – let us face it – also more fun!

nagarro

About the authors

Jan Noessner is an Artificial Intelligence and Machine Learning expert at Nagarro. He has led numerous machine learning and data integration projects in various companies.

Matthias Wuerthele is a Software Test Consultant at Nagarro. He has over eight years of experience in IT project management and has successfully managed numerous test projects.

Kontakt:
aqt@nagarro.com
www.nagarro.com/AI4T

About Nagarro
In a changing and evolving world, challenges are ever more unique and complex. Nagarro helps to transform, adapt, and build new ways into the future through a forward thinking, agile and caring mindset. We excel at digital product engineering and deliver on our promise of thinking breakthroughs. Today, we are 8,400 experts across 25 countries, forming a Nation of Nagarrians, ready to help our customers succeed. www.nagarro.com