CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Test Maintenance for Machine Learning Systems

A Case Study in the Automotive Industry

Master's Thesis in Computer Science and Engineering

LUKAS BERGLUND & TIM GRUBE

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Test Maintenance for Machine Learning Systems

A Case Study in the Automotive Industry

LUKAS BERGLUND & TIM GRUBE

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Test Maintenance for Machine Learning Systems
A Case Study in the Automotive Industry
LUKAS BERGLUND & TIM GRUBE

Supervisors: Francisco Gomes de Oliveira Neto & Gregory Gay, Department of Computer Science and Engineering
Advisor: Dimitrios Platis, Zenseact
Examiner: Christian Berger, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Test Maintenance for Machine Learning Systems
A Case Study in the Automotive Industry
Lukas Berglund & Tim Grube
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Machine Learning (ML) is widely used nowadays, including safety-critical systems in the automotive industry. Consequently, testing is essential to ensure the quality of these systems. Compared to traditional software systems, ML systems introduce new testing challenges with the potential to affect test maintenance—the action of updating test cases. Test maintenance is important to keep the tests consistent with the system under test and reduce the number of false-positive tests.
This exploratory case study was conducted with Zenseact, a company in the automotive industry with a focus on software for autonomous driving. The study included a thematic interview analysis and an artifact evaluation to identify factors that affect test maintenance for ML systems, examine the influence of flaky tests on test maintenance, and provide recommendations on how to improve the test maintenance process. We identified 14 factors, including five especially relevant for ML systems—the most frequently mentioned factor was non-determinism. There were no clear indications that flaky tests have a different effect on test maintenance for ML systems compared to traditional systems. We also proposed ten recommendations on improving test maintenance, including four specifically suitable for ML systems. The most important recommendation is the use of oracle tolerances to deal with variations caused by non-determinism, rather than using a strictly-defined oracle. The study's findings can help fill an existing gap in the literature about test maintenance factors. Practitioners can profit from industry insights that increase awareness of potential test maintenance issues and give inspiration on how to approach challenges in the context of test maintenance for ML systems.

Keywords: Testing, Test maintenance, Test maintenance factors, ML systems, Software testing, Software engineering, Case study, Thesis

Acknowledgements

First of all, we would like to especially thank our academic supervisors Francisco Gomes de Oliveira Neto and Gregory Gay for their great support during the thesis. Their continuous feedback and support in shaping the direction of the study were greatly appreciated. Next, we would like to thank Zenseact for the collaboration on this study and providing us with the necessary resources, including insights into their testing processes. We would especially like to express our gratitude to all employees who participated in interviews and meetings. Without their input, this study would not have been possible. We would also like to thank our industrial supervisor Dimitrios Platis for providing insights into the company and ensuring that we came into contact with the teams and employees most suitable for our study. Furthermore, we would like to thank our examiner Christian Berger and our opponents for evaluating the thesis and giving valuable feedback. Last but not least, we would like to give a special thanks to our families and friends for all the support during this thesis.

Lukas Berglund & Tim Grube, Gothenburg, August 2022

Contents

List of Figures

List of Tables

1

Introduction

Software testing aims to evaluate and verify the behavior of a whole or a smaller part of a system, using test artifacts, e.g., test cases. The different artifacts aim to find faults that should be resolved to eliminate or lower the chance of failures. Testing is especially important in safety-critical domains since undetected faults can lead to life-threatening failures. The case study was conducted with a company in the automotive industry with a focus on autonomous driving. Developing software in an automotive context requires high-quality standards, especially in terms of safety, as failures can lead to damage and death.

When developing software for autonomous cars, Machine Learning (ML) has an important role. As described by Riccio et al. [2], ML can mimic human behavior in terms of gaining experience from training data, abstracting from concrete scenarios, and eventually applying the knowledge to unseen situations. These predictions are made by an ML model, which was trained with an ML algorithm. The algorithms are usually non-deterministic, leading to variance in the predictions even with the same input. ML is already used in many areas, such as Natural Language Processing and Image Processing. One of the most important application areas is the automotive industry, where ML is a major component of realizing autonomous driving by supporting tasks such as collision avoidance and lane-keeping.

An ML system is a software system that contains at least one component that depends on ML. Testing these systems entails new challenges in comparison to testing traditional systems. For example, according to Riccio et al. [2], one major challenge of testing ML systems is the potentially huge input space, making it challenging to find representative input test data that are both diverse and realistic enough. For example, when processing an image from traffic, an almost infinite number of possible situations needs to be considered and reacted to appropriately. Furthermore, it is challenging to find a correct way of describing the expected output due to a possibly complex domain and the non-determinism in ML systems. Finally, popular ways of measuring how well a software system is tested cannot be applied to ML applications. For example, code coverage is not well applicable, since a huge part of the logic is hidden in the ML model [2].

Test cases created for a software system will most likely require maintenance as the project evolves to be up to date with the desired functionality of the system

under test (SUT). In this study, test maintenance means updating test cases, for example, due to changes in the SUT, such as newly implemented functionality, refined requirements, and data set changes. Even a small change in an SUT can make test cases fail, meaning that maintaining test cases should be considered when introducing a change to the code. Furthermore, research has been conducted on test maintenance in different areas, such as GUI-based tests [3], testing patterns affecting test maintenance [4], and factors that indicate the need for maintenance in traditional systems [5, 3]. However, there is a lack of research on test maintenance for ML systems [6]. The conducted research for traditional systems may not directly be applied to ML systems due to the differences in system design and behavior—the main reason is the non-deterministic behavior of an ML system.

The partner company Zenseact, a company in the automotive industry, mentioned challenges with flaky test cases in ML systems, leading to test maintenance efforts. A flaky test is a test with inconsistent results, even though neither the SUT nor the test design changed. An unclear communication process amplified these challenges regarding how to handle the test failures. Therefore, this Master's thesis aimed to analyze test maintenance for ML systems via a case study on Zenseact. We conducted a semi-structured interview study and an artifact analysis to identify test maintenance factors, examine the influence of flaky tests on test maintenance, and provide recommendations on reducing the test maintenance effort. In addition, we compared the results with previous work from the literature. The study aimed to give both researchers and practitioners valuable insights into test maintenance of ML systems based on data from the partner company.

1.1 Problem Description

This study is based on two challenges from the partner company Zenseact about test maintenance for ML systems. First, it was noticed that test cases often had to be updated in relation to a change or retraining of an ML model. It was also found that these issues sometimes hindered the development process since test failures in the CI pipeline had to be solved before further development was possible. Additionally, there is a lack of knowledge in research on factors that affect test maintenance for ML systems.

Test Failures After Updating ML Models

In the automotive industry, ML can be utilized to detect objects by using input data from sensors. When retraining an ML model, the potentially changed predictions can cause test failures in other system components that depend on the ML model. Consequently, maintenance would be required to adjust the test oracle (a mechanism that determines the test result) to the new prediction. However, it has to be determined whether it was a true-positive or false-positive test result before being updated. Whenever a new change in the ML model is made, this process must be done again. Hence, it can be very costly in time and money if test cases for ML system are unstable and require a lot of maintenance. Also, it can be challenging to

test the ML dependent parts together with an ML model on a Unit level. Because of this, a test failure may have to be discussed across teams, especially the ML teams, to define the new test oracle. It can also be challenging to identify the cause of the failure—is the failure due to an internal or external issue? All of this can create confusion, which is costly in terms of test maintenance.

Limited Research on Test Maintenance for ML Systems

Research has previously been conducted on test maintenance for traditional software systems. However, there is limited research on test maintenance for ML systems. Imtiaz et al. [7] claim that most research for test maintenance has been conducted for GUI-based systems, similar to the one investigated by Alégroth et al. [3]. The same assumption can be confirmed by G. Giray [8], who has identified five state-of-the-art papers about testing for ML systems. None of them mentions test maintenance (and how it can be applied) for ML systems. However, a test suite is only meaningful if the tests are trustworthy and up to date with the SUT and the requirements. More research on test maintenance can ease the process for practitioners to improve their testing process, which connects to the quality of the development process and product.

1.2 Purpose of the Study

The purpose of this study was to contribute to both research and industry in terms of increasing the knowledge in the area of test maintenance for ML systems. The study was conducted as a case study with Zenseact since they encountered challenges with the maintenance of ML system tests that depend on regularly retrained ML models (see Section 1.1).

We first aimed to identify factors that affect test maintenance, particularly factors that refer to the specific characteristics of ML systems. However, we also identified factors that are generally applicable to software systems, including ML systems. In addition, we examined the influence of flaky test cases on test maintenance for ML systems. With the help of an artifact analysis, we determined which factors play a role in the company's test cases and discussion threads. Finally, the study provides recommendations on how to reduce test maintenance efforts.

1.3 Significance of the Study

As a scientific contribution, we elicited test maintenance factors for ML systems and shared insights from the industry about how to improve test maintenance to reduce the research gap described in Section 1.1. The test maintenance factors presented in this study are only from one company. However, they are generally formulated (see external threats to validity in Section 6.3.3) and many of them connect to previous work from literature. Researchers can take inspiration from the study and artifacts like the interview guide for future work.

As a practical contribution, we provide recommendations based on the identified test maintenance factors to improve test maintenance. The factors themselves can help to be aware of what can influence the test maintenance to avoid issues in the first place. In addition, since there is limited research on this topic, practitioners can profit from insights from an industrial context.

1.4 Thesis Outline

Chapter 2 (Background) presents essential research topics and terminology relevant to this study and the following chapters. It also includes a general comparison between traditional software systems and ML systems.

Chapter 3 (Related Work) presents previous work closely related to the researched topic and contains an overview of the current research for testing ML systems, test maintenance, test maintenance factors, and flaky test cases. It also presents findings from the literature study in form of challenges of ML system testing and test maintenance factors for traditional software systems.

Chapter 4 (Methodology) presents the research questions and describes the activities and methods used to answer them. It also includes a description of the case study design and how the methods connect to the different working stages. The central part of the case study is the interview study at the partner company. In addition to this, we also conducted a literature study and an artifact analysis.

Chapter 5 (Results) presents the results and answers to the research questions from the activities described in Chapter 4.

Chapter 6 (Discussion) summarizes and discusses the main findings from Chapter 5, compares the results with the related work, and points out the threats to validity.

Chapter 7 (Conclusion) summarizes the contributions of this thesis and outlines potential future work.

2

Background

This chapter describes the background knowledge required to understand the topic of Machine Learning systems in combination with test maintenance.

2.1 ML Systems

Machine Learning (ML) is part of the area of Artificial Intelligence and builds upon learning from data by identifying patterns and allowing predictions on unseen data. ML has been known for a long time but has become especially popular in the last decade due to the ability to now use it efficiently for a vast amount of data. Areas of application are, amongst others, autonomous cars and Natural Language Processing, as well as other domains like the health sector [2].

This thesis differentiates the two terms ML model and ML system. An ML model, also sometimes referred to as an ML network, is responsible for making predictions after identifying patterns in the data. In the context of autonomous cars, this can be processing images from situations on the road to detect obstacles. Learning from data enables the ML model to identify objects like another car, pedestrians, traffic signs, and the street course. When evaluating the performance of an ML model, precision and recall are essential values to determine. According to K. Ting [9], these terms are defined as:

"Precision = Total number of documents retrieved that are relevant / Total number of documents that are retrieved."

"Recall = Total number of documents retrieved that are relevant / Total number of relevant documents in the database."

Therefore, the precision describes how many positive predictions were actually correct (e.g., how many of the detected cars are actually a car). In contrast, the recall describes the sensitivity of the ML model (e.g., how many of the actual cars on the road were correctly detected as cars).

In contrast, an ML system can be described as a pipeline where the ML model connects to different components. Figure 2.1 shows an ML system where one feature uses the output of the ML model and the second feature uses the output of the first

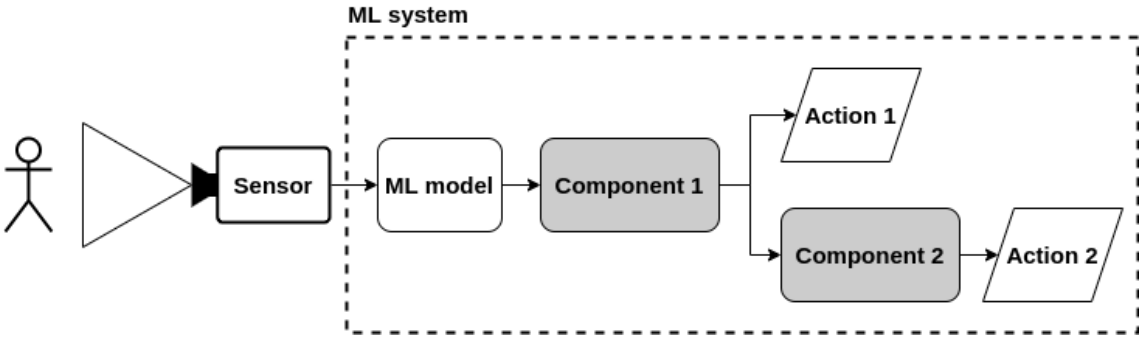

Figure 2.1: Example of an ML system including the flow between an ML model and components. A sensor provides the input to the ML system.

feature. As an example, an emergency braking system in a car is an ML system responsible for braking immediately if an object is detected that requires the vehicle to stop. The object detection is done by an ML model, a trained instance of an ML algorithm for detecting objects.

We refer to traditional systems as software systems without a connection to ML. Their algorithms are programmed and fixed, which means that the behavior is deterministic—as long as the same input is provided, the output will be the same. In contrast, an ML algorithm is usually non-deterministic, which affects other components within the ML system.

Testing an ML system is different from testing a traditional system. The characteristics of an ML system—such as depending on learning from data instead of working according to a programmed algorithm—make it difficult to apply traditional testing approaches [2]. For example, it is challenging to find out whether all relevant cases in an almost infinite input space are covered by tests.

2.2 Test Maintenance

When changes are made in the functionality of a system under test (SUT), the corresponding test cases may need to be updated, which is known as the process of test maintenance (or test evolution). Test maintenance can be described as the act of maintaining and repairing test cases to keep them up to date with the desired functionality of the SUT [10]. Extending or changing already existing parts of a system will most likely require the existing test cases to be updated to be up to date with the SUT, including its source code, design, and requirements [5].

Testing is one of several factors that can affect the development cost of a software system. It is stated by Kumar et al. [11] that software testing stands for approximately 30% of the total development effort and 50-60% of the cost. The same observation was found by Ellims et al. [12], who suggest that software testing stands for 30-50% of the effort in a software project. In addition to this, there is also the cost of maintaining the already existing test cases. Alégroth et al. [3] found that

maintaining test cases at Siemens constitutes up to 60% of the time spent on testing each week. Consequently, the total cost of testing and maintaining test cases is noticeably in relation to the development cost itself. Hence, it is theoretically possible to minimize the development cost by designing optimized and well-thought-out test cases that require less maintenance. On the other hand, time spent on designing test cases is also a factor contributing to the development costs. Designing a test that requires minimal maintenance will lower the future maintenance cost but increase the design phase cost. For that reason, it is essential to find a balance between the cost of designing and the potential cost saving with lower maintenance [13].

The definition of maintaining test cases can have several names, such as test repairing, test case evolution, test augmentation, etc. They all have the same meaning, to keep test cases updated with the SUT. In this study, we refer to this with the name test maintenance.

2.3 Flaky Tests

A flaky test is a test case that produces inconsistent results and can fail without a change in the SUT or the test case itself. Flaky tests can disturb the flow in the CI pipeline and should be taken seriously since the failing tests cannot check the required behavior of the systems and, in the worst case, cannot indicate a fault. In a recent study, Parry et al. [14] state that 59% of software developers have to deal with flaky tests frequently.

Flaky tests can be caused by the test code, the source code, or other external factors like the network connection. For example, if a resource like a database cannot be reached or the network connection is not working, tests can fail without an actual fault in the system. Even if an external factor caused the flaky test, this could still indicate that the system or the tests need to be designed more robust. Another common cause for flaky tests is the dependency on a specific execution order. If the order of execution changes, resources might not be in the expected state for a test, leading to a test failure. The test design can be problematic when the system is non-deterministic by design, but this is not considered in the test design.

If flaky tests occur, a root cause analysis should be done. If the test itself causes flakiness, it needs to be refactored, causing maintenance effort. If flaky tests are ignored, the necessary refactorings are only postponed to the future (long-term maintenance). In the meantime, more tests may be added even though underlying problems or general test issues still exist, which causes avoidable maintenance efforts.

3

Related Work

From our understanding, there is limited research explicitly on test maintenance for ML systems. On the other hand, there is research on ML systems and factors affecting test maintenance for traditional software systems. All of which are combined to form the fundamental part of this literature review. This chapter presents relevant literature and previous work in the topic area.

3.1 Testing ML Systems

In a systematic literature review, Riccio et al. [2] summarize the key challenges of testing ML systems and techniques to deal with them. Among the challenges, the oracle problem is prevalent and a recurrent challenge in many research studies. The oracle problem is the challenge of defining when the tested functionality's behavior is considered correct or incorrect. Due to the uncertainty and non-determinism of ML models, it is difficult to define a test oracle.

In their study about challenges of testing ML systems, Marijan et al. [15] name non-determinism of ML models as the most significant challenge of testing ML-based systems due to providing different output while using the same input and preconditions. Instead of depending on a pre-programmed algorithm like traditional systems, they learn from data; retraining on the same data can lead to different results. Consequently, the behavior is less predictable and it is difficult to define a test oracle. Another challenge is large input spaces that make it more difficult to find all relevant test cases since the number of different inputs can be almost infinite (e.g., all possible scenarios when driving a car). When traditional systems are tested, metrics like code coverage can be used to ensure that the relevant paths are covered. However, as Marijan et al. [15] also state, the application of traditional testing approaches usually fails when testing ML systems, so an adaption to the characteristics of an ML system is necessary. Finally, white-box testing (code is visible to the testers) in an ML system is complicated since much of the logic is in the ML model, which is not based on program code.

There is also literature on how to overcome the challenges of testing ML systems. Metamorphic testing to diminish the absence of oracles is mentioned by Xie et al. [16] in their work on ML classifications algorithms and by Marijan et al. [15]. The latter

paper also discusses differential testing, which means running different software or different software versions with the same specifications and then comparing the behavior on the same input data. However, they also state that this approach is usually inefficient because multiple software runs are required, and the same error can occur in both versions of the system. Huang et al. [17] also contributed with approaches, for example, to measure test coverage, which is more complicated due to the ML model being a black-box. An ML model is considered a black-box because the behavior of a trained ML model is basically not understandable to humans. However, all these approaches focus on the ML model, but they may be valuable in overcoming the challenges of ML systems.

Furthermore, Lwakatare et al. [18] conducted a study close to the industry on challenges and solutions for developing and maintaining large-scale ML-based software systems in industrial settings. They also include systems from the automotive domain and cover a wide range of topics, including testing. However, in contrast to our study, they refer to the maintenance of the SUT but not to the maintenance of tests.

Moreover, there are general differences in testing an ML system due to the use of Machine Learning. Research studies discuss that ML models with entirely correct predictions in real-life scenarios are unrealistic [2]. Therefore, incorrect predictions have to be seen in the whole application context—they are acceptable as long as the ML system can handle them properly. Consequently, writing tests for an ML system that depend on a 100% correct prediction of the ML model is not very valuable or practical. Instead, tests have to be designed to test whether the whole system covers all requirements. For example, an ML model may detect an object one frame later than usual. As long as the depending systems are still able to react according to requirements (especially the safety requirements), this would be an acceptable situation.

In summary, papers exist about testing ML systems and occurring challenges. However, we could not find papers that mention a connection of those challenges to test maintenance or generally papers that connect the characteristics of ML to test maintenance. This includes both the already described papers and other papers found in the literature study—for example, a survey about ML system testing techniques and ML testing research [19], or the influence of ML on software engineering practices like testing [20]. The same is valid for G. Giray [8] where five state-of-the-art papers about testing ML systems were identified, all without a connection to test maintenance. Therefore, this study aims to look into which factors affect test maintenance and considers both general factors and factors especially relevant for ML systems. The special characteristics of ML systems usually also lead to testing challenges—therefore, they are listed in Section 3.4.1 and are later compared with the study results.

3.2 Test Maintenance and Maintenance Factors

Imtiaz et al. [7] state in their literature review that a test suite must be updated together with the SUT to prevent test cases from breaking. A broken test case will make a test suite less effective since it provides a false view of the test coverage. This will lead to false-positive behavior since it is impossible to distinguish between tests that fail due to a fault and those that fail due to not being updated. However, it is possible to repair (maintain) broken test cases by identifying the underlying issues.

Several studies report on factors affecting test maintenance for different types of systems [5, 3, 21]. Fewster et al. [5] present several attributes that affect test maintenance and propose possible solutions that can increase the maintainability of the affected test cases. An example of such an attribute is the *time to run test cases*. Test cases with a long execution time will most likely test too many functionalities simultaneously. This affects the ability to locate the possible faults in the SUT since such tests often execute a large number of functions [5]. However, the findings are general and applicable to most software systems. Similarly, Alégroth et al. [3] identify 13 factors affecting test maintenance costs related to GUIs. For example, they found that the length and flow of a test case will affect maintainability since a long or complex test case will be more difficult to understand by developers and testers. The factors identified by Alégroth et al. [3] are common in the field of automated testing and can therefore be applied to other types of systems. However, it is also stated that further research is needed to confirm this.

We have conducted a thematic analysis to identify and evaluate test maintenance factors. Gonzalez et al. [22] had a similar approach in their large scale empirical study on xUnit testing patterns that can affect the maintainability of test cases. They also present a list of testing patterns connected to a context, problem, and possible solution. For example, one of the patterns is about multiple assertions in the same test cases. Something like this can make it challenging to find the cause of a failure because several things are tested simultaneously. The patterns defined in this study are for general software systems but feel relevant for ML systems. For that reason, they will be considered in the selection of literature maintenance factors for our study.

In this study, we have evaluated the effect of test maintenance factors in the testing process at Zenseact. Similar to this, Levin et al. [23] have conducted a large-scale study on the connection between test maintenance, code maintenance, and semantic changes from several open-source projects to evaluate if they are performed simultaneously or not. The data for the statistical evaluation was collected from the code changes made in the same commit. It was found that test maintenance activities were performed in 15.2–47.7% of the commits within a project. Also, developers have different patterns for when and how much test maintenance they perform. For instance, some developers maintain test cases in all their adaptive commits (adding new features), some in all their corrective commits (fixing faults), and others in all their perfective commits (improving the system). However, it was most frequently

done in the adaptive commits. In our study, we used similar techniques to collect statistical data from the testing processes, especially from the commit history. Additionally, the different patterns for when developers perform test maintenance can potentially influence test maintenance.

3.3 Flaky Tests

In traditional software systems, flaky tests are a well-known problem. Luo et al. [24] point out that three main problems occur during regression testing because of flaky tests: (i) bad reproducibility because of the varying test results, (ii) the huge amount of time spent on (sometimes even non-successful) fault analysis, and (iii) the hiding of actual faults. They also state that the most frequent causes of flaky tests are async wait, concurrency, and test order dependency. When it comes to handling flaky tests, the most common approach is to repeat the execution of these tests and consider them as passing if they pass in one of the executions. Another approach is to disable tests until the underlying issue is found. However, as Kim et al. [25] state, "although disabling tests may help alleviate maintenance difficulties, they may also introduce technical debt. [...] In the end, disabled tests may become outdated and a source of technical debt, harming long-term maintenance". Therefore, flaky tests have at least an indirect influence on maintenance. If the test itself causes flakiness, the time needed to fix it can also be seen as a test maintenance effort. According to Labuschagne et al. [26], "failures caused by broken or obsolete tests represent a test suite maintenance cost".

Moreover, several tools have been suggested to reduce the effects of flaky tests. For instance, DeFlaker identifies whether a test failed due to a code change by monitoring whether the changed code is actually executed by the failing tests [27]. In a similar study, Pinto et al. [28] discover which vocabulary in tests indicates that a test might be flaky. Building upon this study, Verdecchia et al. [29] developed the FLAST tool for the prediction of flaky tests based on static analysis of the source code. A recent overview of the research on flaky tests is given in a survey by Parry et al. [14].

Flaky tests have been given less attention in an ML context. Therefore, Dutta et al. [30] have conducted a study to find "common causes and fixes for flaky tests in projects that build upon and use probabilistic programming systems and machine learning frameworks". The study is similar to Luo et al. [24], but only includes projects that make use of probabilistic programming systems or ML frameworks. The results of the study by Dutta et al. [30] include that most flaky tests (45 of 75 classified tests) in an ML context occur due to *Algorithmic Non-determinism*, which can be fixed in many cases by adjusting the assertion thresholds. They also propose the first tool for detecting flaky tests in an ML context (FLASH).

Since there are indications that flaky tests influence test maintenance, we will further analyze this in the study. It will also be considered whether there is a difference between ML and traditional system testing in how flaky tests affect test maintenance.

3.4 Factors and Challenges From Related Work

3.4.1 ML System Testing Challenges

This section summarizes the findings of challenges of testing ML systems from literature. In contrast to traditional systems, characteristics like non-determinism and large input spaces are prevalent for ML systems, making it more difficult to test these systems. Our study will analyze whether these challenges are factors that affect the test maintenance of ML systems. The challenges are listed and explained in Table 3.1.

The challenges are collected from three papers that explore testing of ML systems. One of these papers has a focus on ML testing challenges and how testing needs to be adapted in comparison to traditional system testing [15], one paper is a systematic literature review on testing ML systems [2], and one paper is on challenges of developing ML in an industrial setting while also covering testing [18]. It was essential to distinguish between papers that focus entirely on ML model challenges and papers that mention challenges relevant to the whole ML system. For example, the large input space has much influence on the ML model itself, but it also means that the tests of the whole ML system have to consider many scenarios and find representative ones to test.

Table 3.1: Challenges of testing ML systems.

Nr	Challenge	Description	Ref.
1	Non-determinism and test oracle	When repeating the training phase on the same training data, there is no guarantee that the model delivers the same output as before (stochastic learning process). This makes it harder to define a test oracle.	[2] [15] [18]
2	Large input space	It is difficult to find representative test data that is diverse, realistic, and able to reveal faults. In addition, it is not always clear which input is valid and which reflects unrealistic scenarios. Furthermore, covering all relevant tests is costly.	[2] [15] [18]
3	ML model mispredictions	Tests need to be tolerant against single mispredictions of an ML model, but only as long as the ML system still fulfills the requirements.	[2]

4	Coverage-based testing is difficult and ineffective	Since at least parts of the logic of an ML system are dependent on the ML model and not program code, traditional measurement of code coverage is not effective in evaluating whether the logic is tested appropriately.	[2] [15]
5	Difficulties in debugging and understanding	Many important parts of an ML system are not coded algorithms, so there are also no stack traces to follow for analyzing the running code. This makes it challenging to understand the behavior and failures.	[2]

3.4.2 Test Maintenance Factors

Factors from three papers exploring test maintenance for traditional systems are presented in Table 3.2. One of the papers has a focus on test maintenance for GUIs, and how that affects the cost of test maintenance [3], one explores testing patterns that can satisfy test maintenance [22], and one book explores software test automation and includes a chapter (7) on how to build maintainable test cases [5]. Only general factors and not topic-specific ones were considered for the selection. Also, factors with a similar meaning or purpose from the same or different sources were merged to reduce redundancy. More information about the selection can be found in Section 4.2.1.

Table 3.2: Factors from literature that affect test maintenance.

Nr	Factor	Description	Ref.
1	Knowledge / Experience	It is generally good to have knowledge and experience when designing and maintaining test cases. Otherwise, the design might neither be optimal nor easy to maintain, which increases development and maintenance cost.	[3]
2	Variable names and script logic	Names and logic are connected to the understandability and readability of test cases and can affect the ability to perform test maintenance.	[3]

3	Test case length and complexity	Long test cases are often complex and might test several things simultaneously. Additionally, branching paths can increase complexity. All of this makes it difficult to gain a clear overview, affecting readability, understandability, and maintainability.	[3] [22]
4	System functionality	System functionality that is missing or not yet implemented can hinder the testing process. As such, it can be impossible to maintain the already existing test cases until this functionality is added.	[3]
5	Fault in the SUT	Faults in the SUT can affect how and when test maintenance can be performed. Test cases depending on a faulty SUT should not be maintained before the SUT is maintained itself.	[3]
6	Number of test cases	A larger number of test cases can increase the total amount of test maintenance effort. Test cases can also overlap, which increases the amount of maintenance needed for the same tested functionality.	[5]
7	Interaction quantity	The number of input interactions in the system and the test is directly connected to the test maintenance effort. If the behavior or requirements of a system component changes, then a large number of test inputs must also be changed. Also, if a test case like this fails and data analysis is needed, that will require more time and effort than for a test with fewer interactions.	[5]
8	Debug-ability of test cases	Test cases should be informative and help explain the cause of a failure. If not, maintenance cost can increase due to the additional time spent on understanding the test failure.	[5]

9	Interdependences between tests	Test cases may rely on each other. However, if one fails, then others may fail as well. Interdependences do not have to be bad if the tests are created with this in mind and the result is examined correctly. Nevertheless, this complicates maintainability since it can be hard to distinguish between an actual failure and a failing dependency.	[5] [22]
10	Naming conventions	Naming conventions can increase consistency and mitigate the risk of confusion due to low understandability. As such, the use of naming conventions can make test maintenance easier.	[5]
11	Test documentation	Documentation is essential for a shared understanding, both now and for future development. Documentation should give a general overview of the test case and its purpose, which helps improve understandability while performing test maintenance.	[5]

4

Methodology

In this study, we investigated the following research questions:

RQ1: What are the factors that affect test maintenance for ML systems and the resulting challenges that emerge?

> **RQ1.1:** What factors affect test maintenance for all software systems?
>
> **RQ1.2:** What factors affect test maintenance for ML systems?
>
> **RQ1.3:** What are the main differences between test maintenance for ML and traditional systems?
>
> **RQ1.4:** How do flaky tests affect the test maintenance for ML systems?

This research question, including its sub-research questions, aims to identify and compare factors affecting test maintenance for ML and traditional systems. It also looks into how flaky test cases affect test maintenance for ML systems. We conducted a thematic analysis of interviews with practitioners to answer RQ1 (described in Section 4.2.2).

RQ2: How are the identified factors reflected in the testing process?

The purpose of this research question is to evaluate the empirical influence of test maintenance factors in RQ1 on the testing process. We answered RQ2 by analyzing artifacts such as test cases and discussion threads at the partner company (Section 4.2.3).

RQ3: How can the test maintenance process for ML systems be improved?

Lastly, RQ3 aims to recommend improvements on test maintenance based on the factors from RQ1.1 - RQ1.2 and the observations in RQ2. The recommendations can potentially help improving test maintenance, which can also lower the cost of testing.

4.1 Case Study

We conducted the case study on Zenseact, a company in the automotive industry. The knowledge and data were collected from a thematic interview study and a qualitative test cases and discussion threads analysis. The company's system includes the whole stack, from low-level sensors to the decision-making based on sensor data. ML is, for example, used in combination with data input from cameras and sensors to make more precise assumptions about the surroundings and detect objects. For that reason, this is considered an ML system since it includes ML parts in the system stack and not only traditional parts.

Furthermore, the usage of ML in the company is divided into three stages, including three different types of teams. First, there are ML teams responsible for creating and training the ML models based on input from low-level sensors and cameras. In a second step, the produced result is filtered and processed to prepare the data for the consuming systems. Lastly, the refined output is used for functionality such as emergency braking and steering. These are the system parts responsible for deciding when an intervention like braking the car is required. This case study focuses on the teams working on the last step (Feature teams), including their test cases.

The ML models are continuously refined to improve efficiency and accuracy. While doing this, the behavior of the ML models can change, which can lead to test failures. An example of this is the timing problem of test cases that expect an intervention at a specific timestamp. Hence, they might fail if the updated ML model affects when the intervention takes place.

4.2 Case Study Design

The case study is divided into three data collection stages. The first stage was a literature study to identify challenges and test maintenance factors from previous work, both for traditional and ML systems. After this, we conducted an interview study collaborating with Zenseact to ask developers, testers, and scrum masters about their experience and knowledge about test maintenance. From this, we have extracted factors and challenges by transcribing and coding the interviews into a clustered theme map. Lastly, test cases and discussion threads were collected from the partner company and examined to evaluate the impact of test maintenance in the testing process. Table 4.1 presents an overview of the case study based on the guidelines by Runeson and Höst [1]. This includes the objective of the study, the concrete case, how data was collected, and how the interviewees and studied artifacts were selected.

Table 4.1: Case study design according to Runeson and Höst [1].

Objective:	Explore test maintenance for ML systems
Case:	ML systems from Zenseact, Employees experienced with ML systems
Theory:	Test maintenance for traditional and ML systems, Flaky tests
Research Questions:	RQ1, RQ2, RQ3
Methods:	First-degree Data Collection: Interviews and artifact analysis (test cases and discussion threads)
Selection Strategy:	Interviewees working in ML teams/feature teams or have experience with ML systems. Test cases that are connected with an ML model (both regularly and not regularly maintained). Discussion threads referring to maintenance factors and dealing with test maintenance of tests connecting to an ML model.

Figure 4.1 presents an overview of the case study process and the artifacts used for different research questions. Each step is further explained below.

(1) **Conduct literature study:** We gathered test maintenance factors for traditional systems and challenges for ML system testing (Section 4.2.1). These factors and challenges were later used in a comparison with the factors identified in RQ1.

(2) **Conduct interview study:** We conducted semi-structured interviews with employees at Zenseact to gather knowledge about test maintenance and to identify potential test maintenance factors (Section 4.2.2).

(3) **Collect and evaluate artifacts:** Test cases and discussion artifacts were collected and analyzed to evaluate the effect of test maintenance factors in the testing process (Section 4.2.3). Both artifacts were collected from several meetings with development teams and a search in the company's primary communication tool.

(4) **Form recommendations based on data:** Recommendations on how to potentially lower the need for test maintenance were defined based on results from the theme map and also considers results from RQ1 and RQ2.

4.2.1 Literature Study

The purpose of the literature study was to collect test maintenance factors and challenges to use those as a comparison to our findings. Traditional literature factors that connect to our findings can support their validity. In addition, Alégroth et al. [3] stated that more research is necessary to validate the listed factors, which is

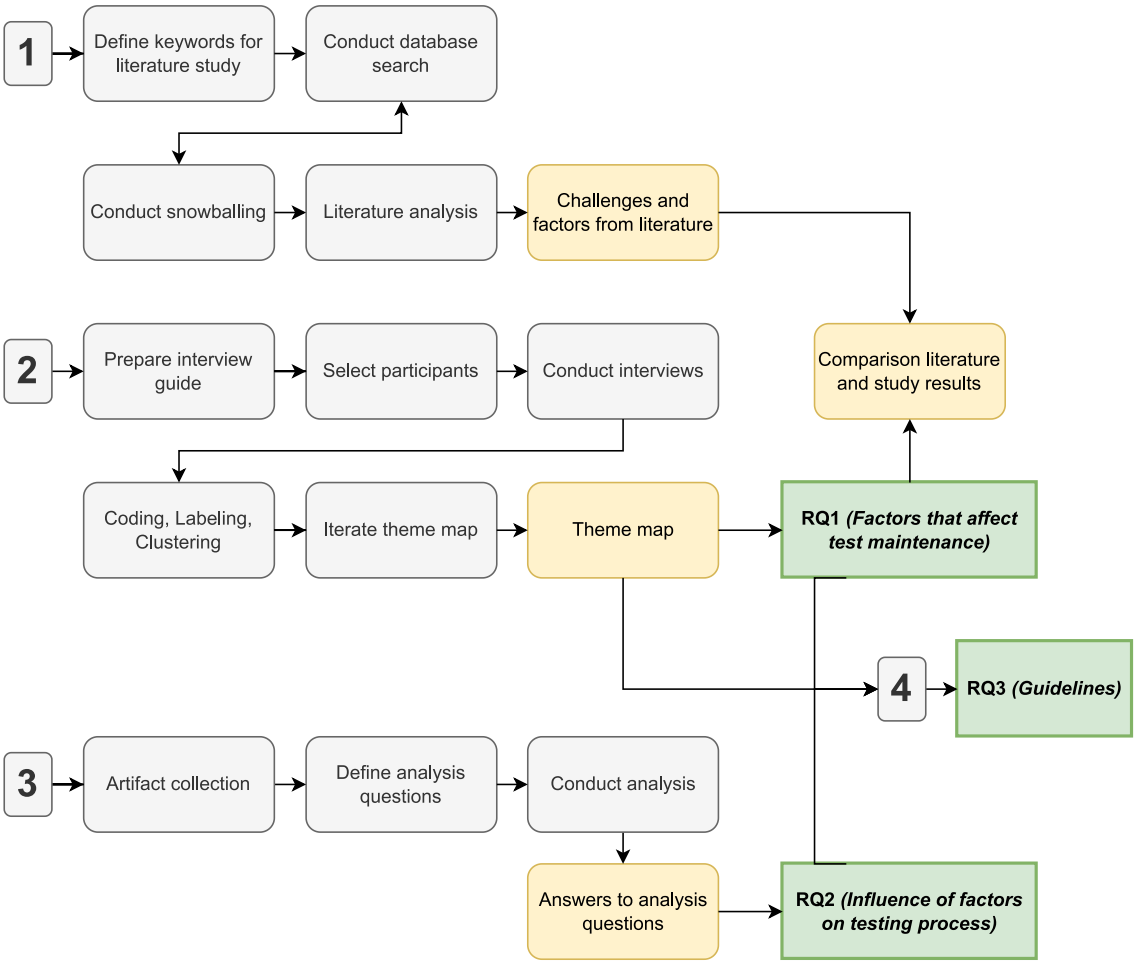

Figure 4.1: Overview of the case study process. All activities are grey, deliverables are yellow, and research questions are green.

also provided by this study. Moreover, if an identified ML system test maintenance factor corresponds to an ML testing challenge, it indicates that the factor is actually referring to an ML system-specific characteristic. These factors from literature are not the basis of the interview and artifact analysis to allow for new themes to emerge—some overlap is expected but not exclusively.

The literature study was performed in three steps over several iterations. Papers were first collected from a database search on a predefined list of keywords. After that, the snowballing technique was applied to the collected papers to further look for potential papers in the reference list. Lastly, an analysis was made on all the collected papers in an attempt to identify data for the case study, such as test maintenance factors and challenges.

Database Search

The search engine used for the database searches was Google Scholar using the patterns in Table 4.2. Papers that included the keywords in Table 4.3 in the heading, introduction, or conclusion were selected to be further analyzed and put into a start set [31]. After a more careful evaluation, several of the papers in the start set were found to be irrelevant and removed from the selection. Some words in the search patterns were combined in a different order, not shown in the table. For example, we always used “test”, “tests”, and “test cases” for the same search string.

Table 4.2: List of search patterns used for the database search.

Search Patterns
Broken test cases
Cost of test maintenance
Factors for test maintenance
Machine learning challenges
Maintainability of test cases
Test augmentation
Test evolution
Test maintenance
Test maintenance factors
Test repair
Testing machine learning systems
Well maintained test cases

Snowballing

In addition to the traditional database search, we used the snowballing technique (backward and forward) on the start set from the first phase. Backward Snowballing is the process of analyzing papers from the reference list in the selected paper, while Forward Snowballing is identifying and examining new papers based on the current one [31]. We selected the papers from both snowballing techniques with the same

Table 4.3: List of keywords used for the selection of literature.

Keywords
Challenges
Difficulties
Factors
Maintainability
Maintenance
Repair

criteria used for the database search. The process was iterative and included further iterations for papers found in the earlier iteration of snowballing.

Literature Analysis

The primary purpose of the literature analysis was to identify test maintenance factors and ML system challenges from the papers selected using the database and snowballing technique. The final result of the selection can be found in Section 3.4. We only selected papers with a clear connection to our topic. We did this to eliminate the risk of making incorrect assumptions that would affect our result.

Papers were selected in three stages. The first selection was applied during the database and snowballing search with the help of predefined keywords. There were several potential papers found. However, only 11 included the keywords in Table 4.3 and showed potential from a quick read-through. We did a more thorough examination of these 11 papers in the next step. From there, we decided to keep three papers on challenges for ML systems and three on test maintenance factors for traditional systems.

Most of the selected papers had predefined lists of factors and challenges. However, some of them were system-dependent and therefore excluded from the selection. For example, some of the maintenance factors from the GUI-based maintenance paper were aimed toward GUI-based systems and cannot be applied to all systems. Furthermore, some of the identified factors and challenges were similar and merged as one in the Table 3.1 and Table 3.2. We argue that the more frequently a factor or challenge is mentioned, the more trustworthy it is.

4.2.2 Interview Study

Selection of Interviewees

The selection of the interviewees was based on convenience sampling [32]. However, we took some measures to select interviewees with as much knowledge of the researched topic as possible. We contacted teams based on their descriptions in the company-internal documentation and recommendations of Zenseact employees. After contacting seven teams, six teams agreed on a meeting. We introduced the teams

Table 4.4: Demographics of interviewees. "Company" includes employment at Zenuity (2017-2020) and Volvo Cars (until 2017), since Zenseact was founded out of these companies. SW stands for Software and MLS for ML System.

ID	Roles	Experience in Years		
		Company	SW Testing	MLS Testing
P1	Developer	3	3	1.5
P2	Developer, Scrum Master	3	3.5	3
P3	Developer	8	10	1
P4	Developer	4	4	4
P5	Developer	4.5	4.5	4.5
P6	Deep Learning Engineer	2.5	3	2.5
P7	Developer, Scrum Master	2	6	0.5
P8	Developer, Test Engineer	9	11	8
P9	Developer, Researcher	5	10	5

to our study and asked about their relation to ML models, test cases that depend on ML model output, and recommendations for other potential interviewees. We were interested in teams that work with ML systems and have tests that depend on ML models directly or indirectly.

Finally, five teams showed interest to participate in the interviews. In addition, we aimed to cover the ML model domain and also consider employees that are generally experienced in testing but are not in these teams. A summary of the participants and their background is given in Table 4.4.

Interview Instrument

We conducted semi-structured interviews with a length of approximately 60 minutes on average. All eight interviews were conducted remotely via the company's digital meeting software, and all nine participants agreed on recording the meeting. The interview guide is shown in Appendix A and consists of three parts. First, the introduction covers general questions about the interviewee's experience and how

testing and test maintenance is handled in their team. The following main part deals with challenges of testing ML systems and traditional systems, how these challenges affect test maintenance, how challenges can be mitigated, and whether there are any other factors to be considered. If applicable, we also asked more detailed questions on specific company tests and how they have been maintained. In the last part of the interview, flaky tests were discussed to examine why flaky tests occur, how to deal with them, whether they influence test maintenance and whether there is a difference between flaky tests in ML systems and traditional systems.

Interview Analysis

All interviews were first transcribed based on the recordings and the transcriptions were anonymized to remove company-internal material. After that, all important parts of the interviews had to be identified. These so-called codes are assigned a label that allows clustering them to convey similar ideas into themes and sub-themes. We applied open coding, which is finding "labels for chunks of data that capture something of the literal essence of the data" [33], so we summarized codes with a label in our own words. It is important to not over-interpret the results early in the process while summarizing codes with code labels. Moreover, we followed a mainly inductive coding approach [33]. Even though we familiarized ourselves with the topics via existing literature and short meetings with employees from the company, we did not predefine themes for the coding. We coded and labelled the interviews individually and afterward compared the results with each other to ensure a unified analysis and to mitigate the effects of bias and missing experience in this kind of analysis.

When the first interviews were coded and labeled, we started to cluster the results. This way, a theme map has been developed and evolved with every added interview. All identified themes are reported in Section 5.1 and Section 5.3.

4.2.3 Artifact Evaluation

Artifact Selection

We selected test cases based on information from the feature teams, which were selected as described in Section 4.2.2. Overall, we found six relevant test cases and also identified two already replaced test cases that help support the analysis of the test case evolution. In turn, the discussion threads were collected from the company's primary communication tool. Discussions of interest can be connected to at least one factor in RQ1 and may include relevant test cases. We received recommendations on potentially relevant discussions during some of the team meetings. We also searched in the communication tool ourselves—we considered feature team channels and two other general channels, where we searched for the keywords maintenance, update, tolerance, failure, factor, flaky, and the names of relevant test cases.

Artifact Analysis

The artifact analysis was a qualitative analysis based on three sets of questions (see below). Similar to Treude and Storey [34], we inspected all artifacts aiming to answer the applicable questions. We complemented our answers with knowledge from employees who participated in the discussions or were involved in the test case evolution. The answers to these questions made the artifacts comparable, and we were also able to assign them to the test maintenance factors from RQ1. Finally, we analyzed the identified factors in more detail to report how they affect the testing process of Zenseact and support these statements with concrete examples from the artifacts.

The first set includes questions for both test cases and discussion threads:

- How is the artifact connected to test maintenance for ML systems?
  - What is the issue or reason for the maintenance need?
- Which factors from RQ1 is the artifact connected to?

The second set of questions referred only to the discussion threads:

- To which test cases does the discussion thread connect?
  - How are the tests connected to ML?
  - How have the tests evolved in relation to the discussion?

Finally, we also aimed to analyze the test cases from a statistical point of view. Therefore, we analyzed the Git history of the test cases and considered commit messages as well.

- How old is the test?
- Did the test replace another test when created?
- Is the test still in use?
- How often has the test been updated?
- How often has the test oracle tolerance been updated?
- How many different people have updated the test?

5

Results

5.1 RQ1—Test Maintenance Factors

This section reports the identified test maintenance factors from the thematic analysis of the interviews. Section 5.1.1 covers factors that are applicable to all kinds of systems, so both traditional and ML systems. The following Section 5.1.2 focuses on the ML-specific factors, while Section 5.1.3 compares test maintenance for traditional and ML systems. Finally, Section 5.1.4 deals with flaky tests and whether they can influence test maintenance for ML systems.

5.1.1 RQ1.1—Factors for All Systems

This section reports the test maintenance factors that were identified for all systems. An overview of all factors and their connection to test maintenance is given in Table 5.1. After that, we describe all factors and their connection to test maintenance, supported by quotes from the conducted interviews.

Table 5.1: Test maintenance factors for all software systems.

Factor	Definition	Impact on Maintenance
Communication	Describes how easy it is to communicate between teams regarding testing and test maintenance activities.	Poor decisions on how to communicate when a test case fails or requires to be updated increases test maintenance time.
Consistency between teams	Degree of how consistent testing and tools are applied across teams.	A low consistency in testing approaches and tools can lead to higher maintenance efforts due to overlapping work, unclear responsibilities, and ineffective collaboration between teams.

Continuity	Describes how much the testing approaches change over time.	Changes in testing approaches over time are unavoidable and even desirable to stick to state-to-the-art practices. Nevertheless, these changes usually require adaptions of test cases.
Debugging support	Ability to support a failure analysis with insights into the underlying system.	If a test does not support test failure analyses sufficiently, the test may need to be maintained to improve the test's observability (the ability to give insights into the tested system).
Dependency on implementation	Degree of dependency of test cases on implementation details (e.g., tests that fit the code instead of the intended behavior).	With a high dependency on implementation details, tests need to be updated more often since most of the changes on these implementation details affect the test cases as well.
Oracle precision	The precision of the output expected by the test oracle.	Sensitive test oracles require frequent updates to account for variances in the tested outcome.
Scenario setup	Describes which scenario setup is used (simulations or real-world data collection).	Scenarios are different in achieving realism and the time required to do so. Scenarios created from computer simulations can lack realism and can have a complex configuration that is challenging to understand. Scenarios from real-world data (driving a car) are realistic, but the data is time-consuming to collect.
Understandability	How easy it is to understand the purpose of a test case, how it assesses SUT behavior, and how it supports attainment of acceptance criteria.	An understandable test case requires less maintenance because it has a clear purpose, supports the assessment of an SUT, and supports debugging if it fails (because one can understand how it interacts with the code).

When test maintenance is performed	The reason and point in time for when test maintenance is performed during the development process.	Depending on the way of working, this factor can lead to technical debt or more work than necessary.

Communication

The factor *communication* connects to how well teams can communicate with each other. Depending on the size and type of system, one team might be responsible for input to another team. For example, team A is responsible for a feature that produces the input for team B. With this setup, a change made by Team A will affect the performance of the feature developed by Team B, hence risking tests from Team B to fail. Consequently, decisions should be made on how to communicate efficiently to lower the maintenance time. A good communication strategy can also limit the risk of a test failure in the first place by directly informing the dependent teams about the change, leading to less time spent analyzing the failure. Also, a lack of communication can lead to frustrating maintenance cycles, as mentioned by P6.

> "For a period of time we had quite frustrating attempts at just merging a simple change or updating one model. And only after setting up a good communication channel and discussing it across teams, we were able to come up with a good workflow." - P6

> "The communication is really a big factor when it comes to easing these [maintenance] problems and making it easier for individual developers and teams to work with test cases." - P6

Consistency Between Teams

There are two main strategies for sharing responsibilities between teams. First, it is possible to have one team that is primarily responsible for the testing framework, and the development teams follow that team's approaches. Besides clear responsibilities, the advantage is high consistency among the teams since everybody works with the same test process and standards. This makes it, for instance, easier for team members to switch between teams and communicate with other teams. On the other hand, the team specialized in testing might become less knowledgeable about the actual system code. One participant mentioned concerns of the specialized team in the past:

> "I know though that there was concerns with respect to the distance to the code and the understanding of the code." - P9

The opposite approach is sharing responsibilities across the teams, to involve all developers in testing, even though that can impede consistency. Consequently, additional effort can be necessary to synchronize teams regarding testing, which requires a well-established knowledge-sharing process and good communication. In addition, one participant mentioned the challenge that teams have different work focuses (e.g., high-level driver-oriented features and low-level calculations), which can lead to a lack of clarity about who is responsible for which tests on the different test levels.

> "The collapsing of [the responsibility of the different levels of testing] on the team level creates either overload of work or unclarity of responsibilities." - P9

Independently of the approach used, consistency influences test maintenance by enabling the avoidance of technical debt and allowing smooth collaboration between teams. Good consistency between teams lowers the risk of inventing the wheel repeatedly, which would cause extra maintenance effort for the different versions.

> "I think that consistence and maintenance of the tools and the setup of the tests: It is as much important as maintaining the functionality of the tests that you are trying to test." - P9

Continuity

The factor continuity refers to how test cases, testing processes, and tools evolve over time. Testing tools evolve quickly, especially in a domain that uses Machine Learning. It is important to stay up to date with modern testing tools to be able to write effective tests. However, switching a tool means that people have to learn to work with the new tool. In addition, it can affect the test cases, which must be adapted according to the new tool, causing maintenance effort. Therefore, it needs to be weighed up whether switching tools makes sense or is avoidable. An example from Zenseact is the use of simulation tools for testing, which were replaced several times.

> "The issue is that we have changed simulation tools a lot of times in the past." - P8

Similar to the tools, the way of writing test cases also evolves. For example, this can be the case due to the availability of new frameworks, new knowledge about how to test, or changes in the company organization.

> "One of the biggest challenges is that the standards and the way people test is always changing." - P9

Debugging Support

Test failure analyses are easier to perform when supported by a test that gives information about the SUT and helps find the failure source. If this is not sufficiently

implemented in the test, the test may need to be updated to save time on failure analyses and debugging in the future. For example, one participant mentioned that they were surprised several times by input changes due to a dependency on another system. Therefore, they decided to add checks to the test to be aware of system changes and save time on test failure analyses.

> "We were a bit confused [since] we haven't changed anything on our side and then we have added some checks to see if the input has changed." - P2

In general, tests can support debugging by facilitating failure analyses as much as possible. For instance, by setting up the test scenario with real-world scenarios (see also factor *Scenario setup*), it is less likely that the test itself is the cause of a test failure.

> "The better and more realistic scenario you create [...] the more you can trust your simulation environment." - P8

Dependency on Implementation

When developing a system, either a feature can be implemented first and then tests are written accordingly, or the other way around. However, there is a risk of testing a specific code implementation instead of the desired behavior when implementing code before writing tests. Tests that rely too much on implementation details are prone to test maintenance. For example, interviewee P2 mentioned a test case strictly depending on a specific initialization of a variable in the implementation. When the implementation was changed, the test failed and had to be updated.

> "There is risk that you fit the tests to work with the code and not really to test, maybe, intended behavior as good." - P8

> "Previously, we have had quite some issues that we were actually testing our implementation. So as soon as we were changing something in the implementation, the tests were failing." - P2

> "They [the too specific test cases] were more blocking us than being useful." - P2

Oracle Precision

With a more precise test oracle, a test case will be less flexible to variations in the tested data. An example of a too precise test oracle could be a test case that checks for a precise number in a situation where variations can exist in the tested output. Consequently, test maintenance would be needed to update the test oracle to fit the variation. Test oracles should be designed for the precision required by the specific use case. If greater precision is not needed, it introduces an unnecessary risk that

can lead to more test maintenance.

> "We should design for not being so dependent on exactly the values and more of like looking at the scenario." - P3

Detailed feature knowledge is required to define a precise oracle. However, this increases the complexity of maintaining the tests since, most likely, not all developers have the same feature knowledge and the ability to define the new values.

> "You need a very deep understanding of the feature and all the steps in the feature to understand how to change these values." - P3

Scenario Setup

This section focuses on test cases that depend on scenarios (for example, a car driving on the street) and how they are created. We have considered two ways of creating the scenarios—simulations and real-world recordings collected from driving a car.

The disadvantage of collecting data from a real-world situation is the human factor needed to collect the data using cameras and sensors and the large amount of data from those recordings. Triggering specific situations at a particular moment in the real-world scenario can be challenging since it can be difficult to guarantee that it represents the expected tested scenario. Also, collecting new data when a scenario is updated is time and resource-consuming because of the human factor, meaning that the time to maintain a test case increases when new data is required. However, a real-world data collection will always represent a realistic behavior of the world around the car. For instance, how objects look and move in relation to the car will automatically be realistic. P3 mentioned that this (using the log files for scenario testing) makes achieving realism in test cases easier.

> "By using a log file which is coming from a car, we don't need to think about how does the car move relative to the world around it, because that is already logged." - P3

> "[It is challenging] to find the right scenario to use, to cut out one very short scenario that should still be somewhat representative" - P3

With simulated scenarios, there is a possibility that the scenario is not realistic compared to what a real-world situation would look like. Consequently, manual work may be required to evaluate the different scenarios during the testing process to evaluate their realism.

On the other hand, simulated scenarios depend on a manual implementation that, if long and complex, can be hard to understand. One of the interviewees (P3) mentioned that these scenarios sometimes include a lot of different parameters that

define how the world will look like, where objects are placed, and how things move in relation to each other. These parameters can complicate and hinder test maintenance since they increase complexity.

> "[Simulated scenarios] can also be very hard to read afterwards and maintain because it is a lot of magic numbers [...] it can be very difficult to figure out what the actual problem is." - P3

Understandability

A test case should be designed so that it is easy to understand, both from the code itself and documentation [35]. We refer to understandability as, for example, the testing standards used, how clear the test flow is, the usage of comments, and how well documented the test is.

> "If something is more readable, it's more maintainable almost by definition." - P7

> "[Test cases] need to be meaningful. They need to be readable. They need to be easy to understand for other developers." - P7

The complexity of test cases also connects to understandability during maintenance. A more complex system will most likely have more complex test cases, which require a better understanding by the developer of both the code and the system's intended functionality.

> "The more advanced the scenario, the harder it gets to actually look at the data and understand if it is correct or not." - P3

When Test Maintenance is Performed

The interviews conclude that there might not be a right or wrong time for when test cases should be maintained. If test maintenance is not done regularly but only when requested, the technical debt can increase. For instance, if useful or necessary refactorings are done later, the new tests cannot profit from them (following the old strategy), and the list of pending refactorings gets longer and longer (technical debt).

Trigger-based test maintenance is about solving the issues when a test case fails or when told to, meaning that more people might be affected by the failure. For instance, if Team A changes the source code of a feature that makes a test case fail, Team B might detect the failure. Then, Team B needs to understand the cause of the failure (factor: *Understandability*) and communicate the failure to Team A (factor: *Communication*). The opposite approach would be to refactor tests along with changes to the code. However, not all code changes require a test case to be updated, meaning that this approach can lead to wasted time because of regular

analyses if the test is still up to date with the SUT. This might not be needed if the change was small and not something that could affect the test outcome.

> "Test maintenance is done trigger-based" - P9

> "Continuously, I would say. [...] We never merge stuff without adding tests or lapping existing tests or maintaining the tests that are related." - P2

RQ1.1: What factors affect test maintenance for all software systems?

Nine factors were identified during the interview analysis that affects test maintenance for all systems, including traditional and ML systems. The factors *Continuity* and *Scenario Setup* are especially relevant in the automotive context. All test maintenance factors are summarized in Table 5.1.

5.1.2 RQ1.2—Factors for ML Systems

This section describes all identified test maintenance factors for ML systems. In addition, the context of the automotive industry and autonomous driving is considered. The results are summarized in Table 5.2 and afterward, each factor is described in detail and supported by quotes from the interview study.

Table 5.2: Test maintenance factors for ML systems.

Factor	Definition	Maintenance Connection
Amount and quality of data	The quality of ML models strongly depends on the amount and quality of data.	Early versions of an ML model often have a lower accuracy and will improve over time with more data. Tests need to be designed to deal with this uncertainty and adapted accordingly.
Non-determinism	ML algorithms introduce non-determinism. Even with the exact same setup, the output can be different after retraining. Both software and hardware can cause non-determinism.	Varying output due to non-determinism influences the way of designing test cases to avoid maintaining test cases often. Being based on statistical outputs, ML models can cause test maintenance efforts, especially in the early phases of a project.

Explainability	The ability to explain the intended system behavior from looking at the code and the ML model.	ML systems are considered black-box systems due to their reliance on complex, non-transparent models. This makes it hard to design optimal test cases by looking at the code or expected behavior, which leads to maintenance for redesigning and adjusting the tests.
Input space	Describes the size of the input space. The input space of ML systems is often huge.	It requires many test cases for a large and complex input space to achieve good coverage, and it is challenging to find all edge cases. Consequently, many test cases may need to be maintained, and the test suite also needs to be updated when new edge cases are identified.
Testing granularity	Describes the level of detail considered when testing and how that affects the test outcome and amount of maintenance.	Testing on a high level exposes the test to other components, meaning that a change in another component can lead to a failure of the test. Consequently, the amount of test maintenance needed is closely related to the test level used.

Amount and Quality of Data

The quality of trained ML models greatly depends on the amount and quality of the data used to train the models. Especially at an early stage of the project, the available data can have biases and inconsistencies that influence the outcome of the system. However, the data improves over time and, consequently, the ML model as well. Until this point, the corresponding test cases have to be designed with this uncertainty in mind: For example, to avoid tests that fail all the time, they need to be more tolerant of deviations in the result or have a different test focus. However, stricter test cases will still be needed later in the process when the ML model is more stable, so tests will need to be added and updated later.

> "Any potential biases or inconsistencies or skewness and so on in that data may affect the outcome." - P7

Even though a large amount of data usually has a positive effect on the ML model quality, it also introduces challenges, as mentioned by two interviewees.

> "There we both have the challenge with the amount of data and processing power that is needed." - P3

Non-Determinism

Machine learning algorithms are usually non-deterministic, which means that the ML model output can differ after retraining, even if retrained on the same data as before. This is by design and acceptable behavior, as long as the ML model accuracy meets the requirements.

Due to non-determinism, an ML model can yield generally better accuracy after retraining, but it can perform worse in one specific scenario. Early in the project, an ML model may not be able to already fulfill all requirements. Therefore, this lower performance in one scenario can sometimes be accepted since the model is more accurate in other scenarios. Retraining an ML model is a very time-consuming activity, so the failing test has to be maintained in some way to not block the CI pipeline. It also depends on the focus of the test, for example, if it is only a general check whether all components are connected or whether safety-critical features are tested. It is essential to emphasize again that these statements refer to an early stage of the project. Later, especially when the product is on the market, inaccurate results in specific scenarios are unacceptable.

> "Since you don't have these deterministic requirements [...] with each training and with each deployment, then things change." - P6

The oracle of the test has a huge influence on test maintenance for ML systems. Depending on what is validated on and how it is validated, the test can be more or less robust against varying outputs caused by non-determinism.

> "Trying to figure out what to validate on in the design scenario is very tightly connected to the maintenance of those values or criteria." - P3

Finally, the robustness of the tests is also influenced by the hardware. In some cases, the non-deterministic behavior is caused by different execution speeds on different GPUs, leading to different results.

> "We were running on two different GPU hardwares that gave different outputs." - P3

Explainability

The black-box characteristic in an ML system comes from the reliance on ML models, which are usually not transparent (i.e., not easy to understand). With the low transparency, it can be challenging to understand the behavior of an ML model, why specific decisions were made, and affects a person's ability to explain their understanding of the model.

> "These uncertainties when it comes to predicting model behavior and model understanding is one big challenge." - P6

As a result of the black-box environment, test cases can be more fragile since it is hard to predict the expected behavior of the SUT and, therefore, also challenging to define a good test oracle.

> "With the ML systems you really don't have the chance to approach the problem with white-box tests because you don't know what is inside." - P6

Also mentioned by one of the interviewees is the lack of connection between test cases and the test code. In a traditional system, test cases can be linked to specific functionality in the code. The link between test cases and functionality in an ML system is not clear due to the low transparency. Tests are still linked to code, but that code depends on an ML model. Therefore, there is no direct link from tests to functionality but an indirect link. As a result, it can be harder to do failure analyses beyond the test case itself.

> "With ML I would say [...] lacking that mapping [between the tests and the code] kind of leaves you in shadow when it comes to maintaining the tests." - P6

> "How can you trace [failures] perhaps to the annotations or the original data that caused this problem, or how do you interpret a failure in that particular test case?" - P7

Input Space

When testing, it is necessary to cover all important cases from the input space to ensure the quality of the product. The larger the input space, the more scenarios there are to test. Zenseact works in a very complex domain with a near-infinite input space (all possible real-life situations around a car). Therefore, many inputs need to be tested, and this is even necessary with an excellent strategy to find representative tests with good diversity. The higher the number of tests, the more need to be maintained.

> "If you try to cover all the input space, then your test case definition and test input is basically infinite. And the number of combinations that you can supply." - P6

Since many ML systems are used in complex domains, it can be hard to select representative input from a huge input space and still cover all edge cases. The more difficult it is to identify edge cases, the higher the risk of not finding all relevant cases in the first iteration. Therefore, the existing test suite might need to be extended and updated in the following iterations. It could also be that new edge cases require a different kind of testing (causing a test restructuring), which can influence the existing tests.

> "The challenging part is to cover and think about all the edge cases in advance." - P6

Testing Granularity

Testing granularity is about the detail level considered in the test, which, for example, can be Unit tests, Integration tests, and System tests. When testing ML system parts that directly use the output of an ML model, it is challenging to test on a Unit level due to the higher level needed to include the ML model component. Unit tests are on the lowest level where only one component (unit) is tested in isolation from other components. The opposite is system-level testing, where all system components are tested simultaneously.

A disadvantage with a lower test granularity (testing on a higher level) is that a test is not isolated and instead exposed to other components in the ML system. Consequently, a change in one of these other components can affect the outcome of this test.

> "For ML systems [...] there the maintenance might be larger and harder because it will be more changes in the behavior there on that level than [...] on a unit test level." - P8

> "In the system level and the HIL [Hardware-in-the-loop simulation] [...] that is where you get more exposed to things that are outside of your own control." - P9

Also mentioned during the interviews is that the modularity of the code affects the testing process. Modularity describes the independence between different system parts, which allows for isolation during testing, and is important to consider in relation to the level of granularity used in testing. Hence, low modularity can mean that test cases depend on more system parts than what is expected to be tested. Consequently, the test case can be more sensitive to change, increasing the amount of maintenance required.

> "We decouple things from each other, and we can in a very controlled manner test different slices of the system." - P7

It is generally easier to design a test case on a lower level than on a higher level since there are more factors and uncertainties, such as input space and isolation, to be aware of. However, a higher-level test can provide a more realistic representation

of the actual behavior of the complete system.

> "The higher you go on the system, all the challenges that I mentioned get much harder." - P9

Since a complete ML system cannot be tested on a unit level when the output of the actual ML model is considered, there is currently no solution to this issue, making this factor something to be aware of instead of something to prevent. On the other hand, the more traditional parts of an ML system, with no dependencies on ML output, can be tested on a lower level to decrease the maintenance need.

RQ1.2: What factors affect test maintenance for ML systems?

Five factors that affect test maintenance for ML systems were identified during the thematic interview analysis: *Amount and quality of data*, *Non-determinism*, *Explainability*, *Input space*, and *Testing granularity*. All test maintenance factors are summarized in Table 5.2.

5.1.3 RQ1.3—Comparison of Test Maintenance in Traditional and ML Systems

The specific characteristics of an ML system introduce some unique factors that affect test maintenance. Additionally, it was found from the interview study that the traditional factors in Section 5.1.1 are generally applicable to software systems, including ML systems. When comparing the two types of systems, it was found that it is easier to maintain test cases for traditional systems.

> "I would say it is easier [to maintain test cases] for traditional systems because [...] [for ML systems] there are these unpredictable shifts and jumps." - P6

An ML system's behavior is different from the behavior of a traditional system, mostly because of being non-deterministic and the lack of transparency in the ML model. One of the interviewees mentioned that this creates an unpredictability that makes it more challenging to perform testing and test maintenance. It can also be challenging to get an overview of the system and break it down into smaller pieces due to the dependencies with other teams and parts of the system. In the subsections below, the main differences in test maintenance for ML and traditional systems are highlighted based on the findings from the thematic analyses in Section 5.1.1 & 5.1.2.

Degree of Determinism

As described in the factor *Non-determinism* (Section 5.1.2), an ML system can produce different results when, for example, an ML model is retrained. The non-determinism causes varying outputs, making it challenging to design test cases and perform testing in general. It also leads to more maintenance for adopting the test oracle to the new variations, to not cause a false-positive test result. Introducing a tolerance in the test oracle's precision range solves the issue of having too strict test cases for ML systems. Tolerance is an additional margin to the expected result to allow for some variance in the tested value. As a general example, an expected intervention for a time-based test can be at 20 sec with a tolerance of +/- 200 milliseconds. The test case now has an acceptance interval of *19.8 – 20.2 sec* (more about the tolerance in Section 5.3.1). This connects to the factor *Oracle precision* since ML systems require a more flexible precision range, which can be achieved with the tolerance. However, defining the correct tolerance value can be challenging as well. On the other hand, traditional software systems may not include variations that can affect the performance or outputs of components in the same way. Hence, a traditional system may not require test maintenance to the same degree as ML systems regarding the degree of determinism.

> "When writing tests cases and interpreting the results [for ML systems], you need to be a little bit tolerant. And that is not necessarily the case for the traditional systems." - P7

Black- vs White-Box Systems and Explainability

The next significant difference between ML systems and traditional systems is the degree of explainability of the system itself (factor *Explainability*). As mentioned before, ML systems are considered a black-box due to the difficulty of understanding the ML model—making it challenging to understand the process between the input and output. A black-box can be described as a room with no light, meaning that it is completely dark and impossible to see what is inside. In comparison, traditional systems do not include a complex model and are therefore easier to understand. Hence, this is why a traditional system is called a white-box system.

> "Not having the opportunity [for ML systems] of looking inside, in code. That would be the largest difference [to a traditional system]." - P6

One participant mentioned that an ML system has similar challenges to a traditional system but that the black-box environment makes them all more challenging to handle.

> "It's problems of the traditional systems but now [for ML systems] on steroids because it is a black-box." - P7

Scope and Testing Granularity

As described in the factor *Testing granularity* for ML systems in Section 5.1.2, there is a difference between the testing level for an ML and a traditional system. The key takeaway is that ML systems are difficult to test at a low level when the ML-dependent parts are included, meaning that they are more often tested on a higher level with a larger scope.

> "To decrease the focus, the scope of it [for ML test case] [...] I think that's harder." - P8

It is generally harder to create isolated test cases on a higher level since more system parts are included and can affect the result. In contrast, traditional systems can have fewer dependencies between components, allowing for isolated tests.

> "The fact that you don't have isolated and very small tests, I would assume you don't in a ML systems, makes maintenance more trick perhaps and more cumbersome." - P7

RQ1.3: What are the main differences between test maintenance for ML and traditional systems?

The main differences in test maintenance between ML and traditional systems are the *degree of determinism*, *explainability of the code* (black-box or white-box), and the *Scope and testing granularity*.

5.1.4 RQ1.4—Effect of Flaky Tests on Test Maintenance

The goal of RQ1.4 is to investigate the connection between flaky test cases and ML systems and how that affects test maintenance. The idea is that the ML systems' characteristics (like the dependency on an ML model which introduces non-determinism) might influence the effect of flaky tests on test maintenance. In addition, the aim is to find out to which degree flaky tests affect test maintenance.

Causes for the occurrence of flaky test cases and how to handle them have been discussed a lot in literature [24, 14]. The results from the interview study have not shown other causes and improvements for ML systems than already reported for systems in general. However, this data may still indicate that these are the most prevalent causes and improvements for ML systems. The identified causes for flaky tests are shown in Table 5.3, while the ways to handle them are described in Table 5.4.

Table 5.3: Causes for the occurrence of flaky test cases.

Cause	Description
Test teardown and execution order	If tests manipulate shared resources, e.g., data from a database, the execution of other tests may be affected. Therefore, every test should reset data after execution (teardown). The same is necessary for shared variables. Otherwise, a test can fail just because it is executed after a test without a teardown.
Infrastructure	Sudden infrastructure issues can lead to temporary test failures. For example, network issues, overloaded servers, and timing issues when building components can cause flaky tests.
Parallel execution of test cases	Parallel execution of test cases can lead to data races and synchronization issues. For example, if thread A changes a value in the database and thread B expects a specific value at this place, the test with thread B fails. If the thread timing is different, it would pass.

Table 5.4: Approaches to handle flaky test cases.

Approach	Description
Detailed analysis of flakiness	The best solution is to analyze the test failure, find the issue, and fix software or tests. For example, one participant had to mirror a repository to avoid server issues.
Rerun test	Rerun the test until it passes. This requires additional test runs, makes the test's meaningfulness questionable, and may hide a potential fault in the SUT.
Remove test	Possible if the flakiness is due to the test being deprecated and no longer needed.
Configure CI	Make the CI system tolerant of the failures of the test. However, that removes the meaningfulness of the test and could irritate other developers. Hence, this is not a recommended approach.
Increase awareness	Bots can help report test results and provide information on whether a test has shown flaky behavior before.

The connection to test maintenance can be seen in two aspects. First, flaky tests cause repetitive failure analyses. It has to be investigated whether the test itself or the system is responsible for the flakiness. A flaky test should not just be accepted; there could be a serious underlying issue in the system. Second, if the test case itself or the test suite causes flakiness, the test (or the test suite) must be refactored. Redesigning and updating the test cases adds to the test maintenance effort, as also mentioned by interviewee P7.

> "When you have some tests that are unstable, usually you do need to do some maintenance or refactoring." - P7

However, when asked about the connection between flaky tests and test maintenance, three interviewees stated that they do not see a clear connection; instead, they turn to the test design and robustness as the most important topic in terms of flaky tests.

> "I wouldn't say that it's more on the maintenance, but more on the design and on the robustness of the setup that you have." - P9

The interview study shows no clear indication that the test maintenance for ML systems is affected explicitly by flaky tests. Instead, it seems necessary to distinguish between the terms *flaky test* and *test with flaky behavior*. Test failures caused by changes in the ML model (for example, by retraining it) are tests with flaky behavior. Even though these tests sometimes pass and sometimes fail, they are not flaky tests by definition. An actual flaky test has different test results, even though neither test nor SUT has changed.

However, if the tests are executed on different GPUs, flaky tests are possible depending on the hardware it is run on. The reason is the difference in execution speed for different architectures. Changing the hardware architecture (executing the test on another computer) can influence the ML model's predictions, potentially leading to a test failure because the tested feature is activated at a different timestamp than expected. Consequently, no actual change was made to the SUT or the test, meaning that this would count as a flaky test. A flaky test due to hardware is not uniquely valid for ML systems, but the unique requirements of the hardware of ML systems make flaky tests more likely.

> "We were running on two different GPU hardwares that gave different outputs." - P3

RQ1.4: How do flaky tests affect the test maintenance for ML systems?

The interview study showed different views on to which degree a flaky test affects test maintenance. We did not see a clear indication that the influence of flaky tests on test maintenance is different for ML systems compared to traditional systems. In contrast to the assumption at the beginning of the study, the tests that fail after retraining the ML model are not actual flaky tests but should instead be referred to as tests with flaky behavior.

5.2 RQ2—Test Maintenance Factors in the Testing Process

RQ2 aims to analyze the connection between the test maintenance factors identified in RQ1 with the testing process at the partner company Zenseact. For this purpose, discussion threads and test cases were analyzed to show how the test maintenance factors affect practitioners at Zenseact.

All analyzed discussion threads are listed in Table 5.5 and each discussion thread is connected to one or more test cases. These test cases, as well as the remaining analyzed test cases, can be found in Table 5.6.

Table 5.5: Overview of analyzed discussion threads, including their connection to test cases and test maintenance factors.

Artifact	Summary	Test Cases	Maintenance Factors
D1	A redesigned test case had a too strict test oracle, which made it fail when other features were updated. A lot of test maintenance effort was required to evaluate the new feature behavior before the tolerance in the test oracle could be updated.	T3, T3.1	Non-determinism
D2	Long discussion thread about how to design end-to-end tests and collaborate between teams to neither disturb the ML developers' workflow nor reduce the quality of the feature team tests.	T3	Non-determinism, Communication, Testing granularity, Explainability

D3	An example of how ML and feature teams agreed to work together (after D2). An ML team had issues with a test failure, so they contacted the feature team that decided to increase the oracle tolerance based on their feature knowledge.	T1	Non-determinism, Communication
D4	Like in D3, an ML team contacted a feature team to get help with a test failure. The teams discussed the difficulty of explaining why this specific code change exceeded the tolerance; the main reason could also be an already integrated code change.	T3	Non-determinism, Explainability
D5	An ML system test case is affected by a change in a connecting feature, leading to a test failure and the need for test maintenance.	T1	Testing granularity
D6	Like in D3 and D4, the discussion was whether the oracle tolerance should be increased after a test failure. The discussion's focus was that the test was still failing for some team members even though the tolerance was increased. The reason was execution on different hardware.	T3	Non-determinism

The statistical data from the analyzed test cases can be seen in Table 5.6. During the analysis, we collected data about the age of the test case, if it was still in use, how many times it was updated, how many people were involved in the updates, and how often the tolerance of the test oracle was updated. Some of the test cases are no longer in use. However, the change history and the connection to discussion threads play an essential role in this study to evaluate how changes in the test case designs over time have affected test maintenance.

In Table 5.6, it can be seen that there is a difference in the amount of performed maintenance between the test cases. Additionally, the oracle tolerance was frequently updated in most test cases (T1, T2, T3, and T5), which could indicate a potential issue with the test design or process. It can be seen that test case T4 has

been most stable according to the performed amount of maintenance. Test cases T3.1 and T3.2 are two predecessors of T3. These two test cases were merged into T3 and later removed.

Table 5.6: Overview of analyzed test cases and statistics about performed maintenance.

Test Case	Age (Months)	In Use	Times Updated	People Updating	Tolerance Updated
T1	9	Yes	14	11	13 (*92.9%*)
T2	9	Yes	6	5	5 (*83.3%*)
T3	6	Yes	10	7	7 (*70.0%*)
T3.1	15	No	14	11	10 (*71.4%*)
T3.2	8	No	7	7	3 (*42.9%*)
T4	5	Yes	1	1	0 (*0.0%*)
T5	15	No	4	4	3 (*75.0%*)
T6	11	Yes	10	8	No toler.

The artifact analysis showed that four test maintenance factors from RQ1 are especially connected to the testing process. In the following sections, we present the impact of these four factors on the testing process of Zenseact using the data presented in Table 5.6 and Table 5.5. We will also explain how each factor connects to the analyzed artifacts.

Non-Determinism

The artifact analysis showed that many test maintenance issues are caused by changes in other components, specifically the ML model. The non-deterministic character of ML makes these changes more complex to handle. For example, even an ML model retrained with the same input data can change the behavior and lead to test failures.

> "We could even have retrained the old network with the exact same configuration and maybe not pass the test." - D2

In artifact D2, it was discussed how a complete ML system chain, including the ML model, should be tested in the best way. One discussion participant stated that the tests should be based on statistical measures instead of deterministic thresholds.

However, even if an ML model becomes statistically better, the performance for particular scenarios can still become worse. Therefore, only statistical tests are not sufficient—instead, tests of concrete scenarios are also needed.

> "We should all keep in mind that even if the network statistically improves, it can degrade in one particular sequence/example." - D2

Some of these concrete scenario tests are the end-to-end tests analyzed in this study, aiming to ensure the functionality of some basic scenarios. They check for different aspects like the number of activations of the tested feature or the feature's trigger timing. The feature activation is called an intervention since an activated feature, like emergency braking, intervenes in the driving. Since the ML model's overall accuracy can differ from the accuracy of a specific scenario, the tests need to be designed robustly. Hence, an interval must be defined for which timing and number of interventions are acceptable. The ways of testing these interventions differ between the analyzed test cases and can be summarized as four different approaches:

- T1, T2, and T5 are testing for the number of intervention requests sent to the ML system.
- T3 is only testing when the first intervention request is triggered.
- T4 uses reference data recorded on a test track where it is possible to use a high precision GPS in the car. This allows setting oracle tolerances according to statements like "Braking needs to be initiated between 15m and 10m away from the object".
- T6 uses no tolerances since its focus is on whether there is an intervention request at all.

According to Table 5.6, T4 and T6 were rarely updated, while T1, T2, T3, and T5 were updated a lot due to adjustments of the test oracle tolerance. Much analysis time was spent in T1-T3 to determine whether the tolerance was too strict or if it was an actual system issue (D1, D4).

The approach of T4 (using scenarios recorded with high-precision GPS data and precisely calculated tolerances) allows for defining a reasonable tolerance right from the beginning, which reduces or even avoids further tolerance adjustments. If a test fails, the tolerance should not be the issue—instead, the acceptable range of results is actually exceeded, meaning that the issue is in the system. However, the situation is slightly different in the early phases of a project. As discussed in D2, there needs to be some room for experimentation in the development of the ML model. Therefore, it might be necessary to exceed the tolerance to avoid failing tests until the ML model is trained well enough to fulfill all requirements. This additional tolerance must be distinguished from the tolerance that describes the acceptable range of intervention requests.

> "At the moment, we are however still developing the network and need room for some experimentation." - D2

Discussion thread D2 also mentions an idea on how to unify the decision on whether a tolerance should be changed or whether an actual issue exists when a test failure occurs. Whenever an ML model change breaks a test, a specific data set has to be collected, which can then be used together with a tool to visualize the influence of the code change on the feature. Based on this, a decision can be made on whether a tolerance adjustment is acceptable.

Finally, the GPU architecture influenced the ML model output, which led to some confusion in discussion thread D6. For instance, a test case had a large oracle tolerance, but the test was still failing for one team member. In contrast to the others, this person used different hardware, which was found to cause the deviating behavior.

Testing Granularity

The factor *testing granularity* was identified in the discussion thread D5 and test cases T1 and T3. In D5, a code change that caused a failure in T1 was discussed between teams. The failure was not because of a change in the ML-related parts of the system. Instead, the change was made in a connected team (Team A), which affected the test outcome of Team B. This is an example of how testing on a higher level can make a test sensible for changes in other parts of the system and not only the non-determinism of the ML model.

Explainability

Not having the ability to understand the code from looking at it was discussed in artifacts D2 and D4. From the discussions, the black-box environment of the ML model made it challenging to understand the code's behavior, which affected the outcome of test cases and increased the amount of test maintenance. There was also a difference in understandability between developers from the ML teams and those from other teams. Naturally, the developers working with the ML model have more knowledge about the model itself, which increases their understandability and explainability. This is an example of why it is important to have good communication and knowledge sharing, like documentation of test cases and code.

> "For us on the feature [team] side, the entire [ML] part of the system feels like a big black-box. It's very possible that these [failures] had nothing to do with the [ML] network itself and we just don't know the difference." - P3

Communication

We have identified the effect of the factor *communication* in several discussion artifacts, especially in D2 and D3. The reason behind discussion D2 was a failure in a feature test caused by the non-determinism from the retraining of an ML model. Consequently, the ML team had to contact the responsible feature team about the updates in the ML model that made the test case fail. From our understanding, there was no clear information about how to communicate a failure for this test case. Consequently, there was confusion in the ML team on whom to contact and when to do it, which badly affected the total test maintenance time and hindered the development process. Nonetheless, the agreements on communicating test failures were later visible in D3, where the process discussed in D2 was applied.

RQ2: How are the identified factors reflected in the testing process?

The examined discussion threads (Table 5.5) and test cases (Table 5.6) show the influence of *non-determinism*, *testing granularity*, *explainability*, and *communication* on test maintenance. In particular, *non-determinism* creates the need for test maintenance and introduces further challenges with regard to explainability and communication.

5.3 RQ3—Improvements of the Test Maintenance Process for ML Systems

The goal of RQ3 was to identify ways to improve test maintenance, especially for ML systems. These are presented as recommendations in the following subsections, based on observations made while answering RQ1 and RQ2 and results from the interviews.

5.3.1 Recommendations for ML Systems

This section presents recommendations especially relevant for ML systems. The results are summarized in Table 5.7 and afterward described in more detail.

Table 5.7: List of recommended practices and corresponding improvements on the test maintenance for ML systems.

Recommendation	Maintenance Improvement
Use tolerance in the test oracle	Test oracles for ML systems should be tolerant to the varying—but still correct—behavior of the ML model. A tolerance for the expected test behavior allows for more flexibility in the test oracle, which leads to less maintenance for adjusting test cases that otherwise would be too strict.
Force determinism / Isolation	If a component is tested in isolation, effects like non-determinism caused by other components (like the ML model) are prevented. The isolation can be achieved by not actually using the ML model but instead mocking the ML model output. Consequently, the tests do not need to be updated regularly—unless the developers actively decide to do so.
Use consistent hardware	Using the same hardware architecture is essential when testing ML systems since it affects the ML model's execution speed, which can lead to different test results and more maintenance.
Property-based testing (PBT)	PBT is a testing technique where a PBT framework generates tests to find inputs that violate a set of properties. Since ML systems usually have a large input space, PBT can help to cover an extensive range of inputs while reducing the number of manually created tests that need to be maintained.

Use Tolerance in the Test Oracle

Using a tolerance rather than a precise oracle is especially important in an ML system to handle *non-determinism*. For example, a test case expects an intervention to happen at 11 sec, but the actual intervention happened at 11.5 sec in the first test execution and 10.9 sec in the second. In both cases, this would lead to a test failure since the actual value was different to the expected. However, a tolerance of +/- 1 sec would have solved this issue and removed the need for maintenance.

> "For the non-deterministic output [...] we should design our test cases to be robust [...] So that we can accept small variations" - P3

> "You should not have like 'if you give this input you have to get exactly this output' [...] it's better to see if you give this input, then it should go towards that direction" - P2

> "Adds an interval to the test so that the test is more robust to small varieties in the number of requests." - T5 *(Commit message)*

It is essential that the tolerance is selected and defined correctly. A too large tolerance can make a test case less meaningful regarding validation since the acceptance criteria in the oracle will be too large. Hence, faults in the SUT might no longer be detected. In contrast, a too small tolerance removes the advantages of having a tolerance in the first place since there might not be enough room for variations in the test oracle.

One way to better define a tolerance for the expected value is by collecting statistical data from a real-world situation. For instance, test case T4 uses reference data collected from driving a car with a high-precision GPS. The data was later used to determine a min and max value expected for the braking distance in a particular situation. In Table 5.6, it can be seen that test case T4 is stable—the tolerance has never been updated so far—meaning that this approach for selecting a tolerance could potentially lower the maintenance need for adjusting those values. The other test cases did not use reference data for the tolerance, which can be one of the reasons for higher maintenance efforts.

Force Determinism / Isolation

A component can be tested in isolation to force determinism, meaning that there is no direct connection to the ML model. Since the tested component still needs data from there, this data can be provided by mocking the ML model so that the test works with data that only represents the ML model output.

> "What is run now automatically is not really dependent on ML because we always take the same input." - P2

> "Try to isolate them [the tests] and mock the relevant parts." - P6

We identified two mocking strategies at Zenseact. First, the output of the ML models can be replaced with mocked values that are manually selected and represent a scenario that the feature needs to react on according to the requirements. These mocked values are fixed and only change when it is decided that they need to be updated. This type of testing focuses on testing only the behavior of the feature itself and does not depend directly on an ML model, which can be seen as testing a unit in isolation, similar to unit tests for traditional systems.

Another strategy is to record actual ML output and save it in log files, which can later be used to mock the ML model output. Compared to the first approach, the data is more realistic, not prone to faults in the scenario due to the manual work, and the approach is less time-consuming. The disadvantage is that the feature developers depend on the scenarios processed by the ML model and, consequently, are more limited in which scenarios they can test.

Nevertheless, since the input data is fixed, there is a risk that the tests provide a false sense of security because the tests are not based on the latest ML model. Hence, it has to be observed when the mock data have to be updated. One interviewee stated that sometimes issues occur that can be traced back to the input—if that is the case, the feature developers need to update the mock values manually or use a log file with up-to-date recorded data.

Isolating components for testing allows for more lower-level testing, which usually causes less test maintenance effort than high-level testing. Nevertheless, end-to-end tests cannot and should not be avoided since only those tests can ensure that everything works together appropriately.

Use Consistent Hardware

The performance difference between hardware platforms, like GPU architectures, can affect the execution speed of ML models or other system components. Consequently, test cases that depend on the performance of a component (time-based test cases) can be affected and potentially fail depending on the acceptance criteria.

> "Once we started running on only one hardware [...] we got rid of most of these like large variations" - P3

This is especially important in the automotive industry since it can happen that the hardware in a car is not the same as the one used during development and testing. Therefore, it is also essential to use hardware that is as close to that running in the automobile as possible. One way to ensure this is with hardware-in-the-loop (HIL) testing. HIL testing is a technique where the hardware from the real end product is tested with the developed software in a simulated scenario. Doing this makes it possible to better understand how the system will behave on the hardware to be used.

Property-Based Testing

Property-based testing (PBT) frameworks can generate a wide range of inputs, which are all automatically tested on the SUT. Therefore, the test oracles must apply to any input—the test oracle is not defined explicitly for every input. If this is possible, PBT allows covering large input spaces easier and without manually maintaining tests for every tested input. Zenseact already uses this testing technique for various tests and, according to interviewee P6, has had positive experiences with it. However, property-based testing should not be seen as a replacement but as an

addition to unit testing.

> "You can decide to write a 1000 of tests manually by hand that coverage the separate spaces of the input of that possible input space. Or you can just use property-based testing and use the frameworks that are written towards it." - P6

5.3.2 Recommendations for All Systems

The interviews and the artifact analysis also led to recommendations for both traditional and ML systems. These recommendations are listed in Table 5.8 together with how they improve the test maintenance process.

Table 5.8: List of recommended practices and corresponding improvements on the test maintenance for all software systems.

Recommendation	**Maintenance improvement**
Knowledge sharing	General knowledge sharing activities like a book circle and mob reviews can improve the shared understanding of the SUT and teach developers about new useful testing tools or procedures.
Maintainer tags	A maintainer tag (information about the responsible person or team) in the test case can ease the communication between teams by increasing the understanding of whom to contact during a test failure, which can also speed up a failure analysis.
Slack bots	Slack bots can be used to share information about test failures, flaky test cases, and other testing issues. This allows for quicker information sharing among developers, reducing the test maintenance time for the failure analysis since the failure is noticed earlier.
Test-driven development (TDD)	TDD can improve the quality of the test cases and ensure they reflect the requirements, reducing the risk of tests depending on the implementation.
Test failure messages	Two interviewees recommended to use very detailed failure messages in the tests to increase the understandability of a test failure. They can also be used to provide insights into the system at the time of the failure.

Unified scenario setup	Complex scenarios require many configurations, which can become difficult to maintain. Unifying the scenario setup, for example, by providing a framework or template for the setup, encapsulates the required logic. Consequently, the complex parts of the scenario setup only need to be maintained in one place, which is also advantageous when refactoring several tests.

RQ3: How can the test maintenance process for ML systems be improved?

Ten recommendations on how to improve the maintenance activities for ML systems were proposed. Four of these are especially relevant for ML systems since they address significant ML testing challenges (Table 5.7). The other recommendations are presented in Table 5.8.

6

Discussion

6.1 Oracle Tolerances for ML System Tests

When examining the thematic analysis from the interview study and artifact analysis, we identified that a tolerance in the test oracle plays an important role in handling the non-deterministic behavior caused by the ML model. For instance, instead of defining a precise value as an expected output, our data suggest that expecting values based on a confidence interval alleviates maintenance effort. Because of the varying ML model output, strict test oracles for test cases that depend on the ML model can lead to test failures and, consequently, more test maintenance.

Two things to consider when deciding on the size of a tolerance are: (i) A small tolerance would not make a large difference in the flexibility, meaning that it can still be too strict. (ii) A too large tolerance can make a test case less meaningful since it now can allow for too much flexibility and could miss potential faults or other issues in the system. Defining the correct tolerance size is almost more important than the reason for using a tolerance in the first place. This becomes clear when looking at how often the tolerance was updated for the test cases in Table 5.6. In almost all test cases, the tolerance was updated in more than *70%* of the total changes—in one case, even *92.9%*. Updating the tolerance is a test maintenance activity, so selecting the correct tolerance size is tightly connected with the total amount of test maintenance required.

Our collected data indicates that the tolerance for a test oracle can be better defined using high precision recordings of a real-world situation. In the automotive context, these can be measurements done using a high-precision GPS in a car to define an expected min and max value for a specific use case. For instance, the actual braking distance of a car can be measured to define an average value to be tested, and the min and max values can represent the tolerance to be used. One of the testing artifacts (T4) in Table 5.6 uses this approach to define the tolerance and has been stable so far.

One of the key takeaways identified during the study is that tolerance should probably be adapted according to the development process. Earlier in the development of a system, the ML model can be less precise and produce more significant variations in the output compared to what is expected later in the development. A test case

that is too strict at the beginning of the project can hinder the development process with unnecessary test failures. Therefore, it needs to be examined whether (i) the test failures are accepted and used for feedback to the ML model, (ii) the tolerance is allowed to be exceeded temporarily but is closely observed, or (iii) the test oracle aims to check the general functionality instead of depending on a specific accuracy of the ML model (which can be changed later in the project). This also connects to other factors such as *communication* and *understandability*—especially as (1) and (2) require good communication and that everybody involved is able to interpret test failures and tolerances in the same way.

One idea that was briefly explored was the ability to use the accuracy of an ML model to predict the tolerance to be used. A solution like this could potentially allow for a tolerance that fits better with the current version of the ML model, both for decreased and increased accuracy. When an ML model gets statistically better, it would also be reasonable to decrease the tolerance size to not have a too loose test case. This idea was briefly discussed and received some positive feedback during the interviews. However, no further research or experiments were conducted on this, and it was decided to leave it for future work.

Lastly, tolerances can also lead to a lack of clarity on which code change is responsible for a test failure due to crossing the tolerance boundaries (D4). Several code changes in combination can slightly move the expected intervention time and eventually cross the tolerance boundary. Even though this triggers a test failure, an actual fault may not have been introduced. However, an earlier change may not have been optimal in turns of execution speed, leading to this happening. An example from the interview study: A test case expects an intervention at 10 sec with a +/- 1 sec tolerance. Code change A moves the intervention time from 10 sec to 10.9 sec. This moves it quite close to the tolerance boundary, but is still acceptable. After this, a new code change B is added, which further moves the intervention time to 11.1 sec, triggering a test failure. However, even though change B caused the failure, change A had a larger impact on the intervention time, meaning that change A might be the actual reason for the failure. Hence, this leads to a lack of clarity about the cause of the failure.

6.2 Comparison With Existing Literature

6.2.1 ML Testing Challenges

Factors and challenges are two concepts that need to be distinguished—while a factor is measurable, a challenge is something to overcome. For example, the factor *input space* describes how large an input space is. Challenges occur depending on the size of the input space—a large input space leads to problems, such as finding effective test input.

In addition to large input spaces, we identified four more ML testing challenges from literature, as presented in Section 3.4.1. The most important challenge is non-

determinism and, consequently, the difficulty of defining a test oracle with varying ML model outputs [2, 15, 18]. This has been confirmed by the observations in the study and refers to the factor *non-determinism*.

Another challenge from the literature are ML model mispredictions, which an ML system must be tolerant against while still fulfilling the system requirements [2]. The same is valid for ML system tests—they also need to be tolerant, similar to dealing with *non-determinism*.

The two remaining challenges from literature are the difficulties of coverage-based testing and the difficulties of debugging and understanding. This is caused by the complex behavior of an ML model, which makes it basically a black-box. Hence, both challenges refer to the test maintenance factor *explainability*.

6.2.2 Maintenance Factors

From the literature study in Section 3, we were able to identify eleven factors that can affect the maintainability of test cases in general. These findings can be seen in Table 3.2, together with an explanation and references. Several of the factors were identified in the paper from Alégroth et al. [3] on test maintenance factors for GUI systems. However, it was also mentioned that more research is required to confirm the impact of these factors. Based on this, we have decided to compare the identified test maintenance factors with those from the literature to validate our findings and extend on already existing knowledge in the literature.

From the comparison, we found that the factors *Variable names and script logic*, *Interdependences between tests*, *Naming conventions*, and *Test documentation* from literature connect to our *Understandability* factor. As mentioned by Fewster et al. [5], documentation of test cases and naming conventions are tightly connected with the degree of understandability that is provided. Two factors identified in our study were also identified in literature with a similar definition and purpose—*understandability* (e.g., the benefit of comments in test cases as documentation) and *debugging support* (e.g., via information failure messages). Additionally, the literature factor *Knowledge / Experience* connects to our recommendation on *Knowledge sharing* for improved test maintenance.

The factors *Test case length and complexity*, *System functionality*, *Defects in the SUT*, *Number of test cases*, and *Interaction quantity* from literature were briefly mentioned by individual interviewees, but not to a broader extent. However, we also identified six more factors in our thematic analysis that did not have a direct or indirect connection to identified factors for literature.

6.2.3 Flaky Tests

Dutta et al. [30] stated in their study that most flaky tests in ML systems occur due to algorithmic non-determinism. Since ML systems heavily use these kinds of algorithms, many flaky tests could occur, causing maintenance efforts. However, in

our study, most changes in test results were caused by a modification rather than flakiness. Therefore, we distinguish between flaky tests and tests with flaky behavior since the former assumes that no modifications were made that caused the failure.

In the literature, there are indications that flaky tests influence test maintenance, especially when postponing an appropriate analysis of the underlying causes [25]. Our study could also find indications for this, even though some interviewees stated that flaky tests primarily connect to test design and robustness of the test setup. We could not confirm the theory that the influence of flaky tests on test maintenance is different between traditional and ML systems. We also found that the GPU architecture can influence the test result and can actually lead to flaky tests. However, we could not identify clear ML system-dependent factors that lead to more flaky tests and more test maintenance.

6.3 Threats to Validity

This section points out the threats to validity to enable the reader to evaluate the trustworthiness of the presented results. In addition, the mitigation strategies for those threats are described. According to Runeson and Höst [1], the threats to validity are presented in the four subsections Construct Validity, Internal Validity, External Validity, and Reliability.

6.3.1 Construct Validity

Construct validity is about how the conducted analysis actually represents what was stated as the study's goal. This is mainly the absence of clear definitions of the terms Test Maintenance and ML System; the literature also uses similar terms for the same purpose, like Test Evolution and ML-based System. Hence, the interpretation of terms could have differed between the interviewees and the study's authors. We mitigated this ambiguity threat by (i) having initial meetings with all the teams of the interviewees to present the study and (ii) giving an introduction to the study and the used terms during the interviews.

6.3.2 Internal Validity

Internal validity describes risks that threaten the validity of the data collection and, consequently, the derived results. One risk is bias in participants through the interview questions or the meetings with the teams we had before. We mitigated this risk by asking the interview questions without mentioning factors ourselves. We concentrated on discussing the problem statement during the team meetings and asked the interview questions independently of the discussions in the meetings.

Another risk was not finding the right artifacts (especially test cases) in a relatively large codebase (sample selection). We addressed this issue by contacting employees in several ways: We asked for information in a discussion channel with over 500 employees, had several meetings with employees and whole teams in both written

and oral form, and asked our industrial supervisor for contact persons with expertise in the research area.

Since the interview quality depends a lot on the interviewees, maturation can also influence the study's results. People can change and gain additional knowledge during the time of the study. This is not a huge risk in our study since the main data comes from one interview per person. In case interviewees would not have been able to give answers of their usual quality, we could compare statements with knowledge from the team meetings, further follow-ups, and the artifact analysis.

Finally, since a wide range of topics needed to be covered during the interview, there is a risk that the interview guide was not focused enough on the study's goals or did not include all relevant questions. In addition, even though we conducted semi-structured interviews, we might have missed going into more detail about some topics. For example, we could have tried to gain more information on aspects like firmware and operating system drivers when discussing hardware aspects. We mitigated these risks through refinements in several iterations of the interview guide, including feedback from the academic and industrial supervisors. Both researchers also asked additional questions during the interviews whenever it made sense to the best of our understanding.

6.3.3 External Validity

External validity deals with the generalizability of the results. The main risk is that the study was only conducted on one company, and therefore, the results may not be applicable to other contexts. The results of RQ1 and RQ3 are derived from the specific company context, but we formulated the factors and suggestions so that they should apply at least to companies from the same domain. In comparison, the results of RQ2 are more company-specific. However, the main observations should also help other companies to introduce a smooth working process right from the beginning of a project.

Furthermore, the literature comparison shows that many of the factors identified in this study connect to challenges and factors from the literature. The studied literature does not focus on a specific domain, indicating that most of the identified factors are valid outside the studied company.

6.3.4 Reliability

Reliability describes to which extent other researchers can reproduce the study. The qualitative analysis of an interview study with a theme analysis leads to the risk that other researchers do not apply the same way of coding the interviews. Besides describing the process in Section 4, the authors of this study mitigated this risk by coding the interviews separately. In a second step, they compared the results to mitigate the potential bias of a single author. In addition, if there were any doubts about a statement, the interviewee was contacted in written form and asked for clarification and more examples.

Furthermore, the number of interviewees and identified artifacts can differ when applying the methodology to another company. This study is based on a relatively small number of interviews and artifacts, which is a risk regarding completeness. However, as mentioned in the internal validity, several sources with many different people were addressed to make the picture as complete as possible. The analyzed artifacts were mentioned by several employees and seemed to be the most relevant ones. The interviewees were selected from seven different teams (including five feature teams) to gain an understanding as representative as possible. We also had contact with seven potential feature teams at the beginning of the study to find the right teams to interview.

7

Conclusion

In this study, we explored the topic of test maintenance for ML and traditional systems, with a particular focus on the automotive industry. Maintaining test cases is important to keep consistency with the system under test and reduce the number of false-positive tests. Our partner company Zenseact has encountered challenges of maintaining ML system test cases, including indications of the potential influence of flaky test cases, especially when an ML model is retrained. Therefore, we conducted an exploratory case study, including eight interviews and an analysis of test case and discussion artifacts, to identify factors affecting test maintenance, evaluate their impact on the testing process, and report recommendations to improve test maintenance.

We identified 14 factors that affect test maintenance, including five especially relevant for ML systems. In general, the factors and challenges identified tend to be domain-specific rather than specific to the partner company. The most frequently mentioned factor was non-determinism, which was supported by the artifact analysis, where non-determinism was frequently discussed as a factor leading to test case updates. The study did not clearly show a different effect of flaky tests on test maintenance for ML systems compared to traditional systems. However, one key finding was that the term "flaky test" should be distinguished from tests with flaky behavior. Tests with a different result because of a change in another component (like retraining an ML model) do not correspond to the definition of a flaky test. Finally, we reported ten recommendations that can help improve test maintenance. Four of them (test oracle tolerance, force determinism, consistent hardware, and property-based testing) are especially suitable for ML systems. These recommendations address the test maintenance factors particularly prominent in this study (non-determinism and input space) and the corresponding challenges of ML testing.

In addition to the case study, we also collected eleven test maintenance factors and five ML testing challenges from research. The identified ML testing challenges connect to the ML system factors identified in our study. While some of the traditional factors from literature relate to the study results, the results also contain additional factors, some of them especially relevant in the automotive industry context. The data collection in the study was conducted independently of the identification of factors and challenges from the literature.

One limitation of the study is the focus on only one company. Therefore, specific company processes can play a role in the results, but we hypothesize that these factors and recommendations also apply in other contexts. Another limitation is the relatively small sample size of interviewees and artifacts. This was mitigated by the interview and artifact selection strategy, but a broader range of information is needed to validate the findings. Examining ML systems (including those from other domains) would be helpful to complete the picture.

There is limited research on test maintenance for ML systems. We hope to inspire future work with our study since test maintenance is a relevant topic, especially in safety-critical systems that require an extensive number of tests. We conducted the study on only one company and a relatively small sample of interviewees and artifacts. Therefore, future work could explore test maintenance at more companies and domains outside the automotive industry to validate the factors identified and add to the catalog of factors and recommendations. It would also be interesting to further analyze how ML systems differ in their need for test maintenance compared to traditional systems. In addition, the approach of using the accuracy of an ML model to define better tolerances for test oracles, as mentioned in Section 6.1, could be interesting to explore further. Finally, it would be valuable to complement qualitative studies (like this study) with quantitative studies, for example, measuring the effect of concepts like oracle tolerances on test maintenance.

Bibliography

[1] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, pp. 131–164, 2008.

[2] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, and P. Tonella, "Testing machine learning based systems: a systematic mapping," *Empirical Software Engineering*, vol. 25, pp. 5193–5254, 11 2020.

[3] E. Alégroth, R. Feldt, and P. Kolström, "Maintenance of automated test suites in industry: An empirical study on visual gui testing," *Information and Software Technology*, vol. 73, pp. 66–80, 2016.

[4] D. Gonzalez, J. C. Santos, A. Popovich, M. Mirakhforli, and M. Nagappan, "A large-scale study on the usage of testing patterns that address maintainability attributes: Patterns for ease of modification, diagnoses, and comprehension," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 391–401, 2017.

[5] M. Fewster and D. Graham, *Software test automation*. Addison-Wesley Reading, 1999.

[6] H. B. Braiek and F. Khomh, "On testing machine learning programs," *Journal of Systems and Software*, vol. 164, p. 110542, 2020.

[7] J. Imtiaz, S. Sherin, M. U. Khan, and M. Z. Iqbal, "A systematic literature review of test breakage prevention and repair techniques," *Information and Software Technology*, vol. 113, pp. 1–19, 2019.

[8] G. Giray, "A software engineering perspective on engineering machine learning systems: State of the art and challenges," *Journal of Systems and Software*, vol. 180, p. 111031, 2021.

[9] K. M. Ting, "Precision and recall," in *Encyclopedia of Machine Learning* (C. Sammut and G. I. Webb, eds.), pp. 781–781, Boston, MA: Springer US, 2010.

[10] M. Skoglund and P. Runeson, "A case study on regression test suite mainte-

nance in system evolution," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pp. 438–442, 2004.

[11] D. Kumar and K. Mishra, "The impacts of test automation on software's cost, quality and time to market," *Procedia Computer Science*, vol. 79, pp. 8–15, 2016. Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016.

[12] M. Ellims, J. Bridges, and D. C. Ince, "The economics of unit testing," *Empirical Software Engineering*, vol. 11, no. 1, pp. 5–31, 2006.

[13] K. Karhu, T. Repo, O. Taipale, and K. Smolander, "Empirical observations on software testing automation," in *2009 International Conference on Software Testing Verification and Validation*, pp. 201–209, 2009.

[14] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, oct 2021.

[15] D. Marijan, A. Gotlieb, and M. Kumar Ahuja, "Challenges of testing machine learning based systems," in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pp. 101–102, 2019.

[16] X. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *J. Syst. Softw.*, vol. 84, p. 544–558, apr 2011.

[17] S. Huang, E.-H. Liu, Z.-W. Hui, S.-Q. Tang, and S.-J. Zhang, "Challenges of testing machine learning applications," *International Journal of Performability Engineering*, vol. 14, pp. 1275–1282, 06 2018.

[18] L. E. Lwakatare, A. Raj, I. Crnkovic, J. Bosch, and H. H. Olsson, "Large-scale machine learning systems in real-world industrial settings: A review of challenges and solutions," *Inf. Softw. Technol.*, vol. 127, p. 106368, 2020.

[19] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *CoRR*, vol. abs/1906.10742, 2019.

[20] Z. Wan, X. Xia, D. Lo, and G. C. Murphy, "How does machine learning change software development practices?," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1857–1871, 2021.

[21] S. Wang, M. Wen, Y. Liu, Y. Wang, and R. Wu, "Understanding and facilitating the co-evolution of production and test code," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 272–283, 2021.

[22] D. Gonzalez, J. C. Santos, A. Popovich, M. Mirakhorli, and M. Nagappan, "A large-scale study on the usage of testing patterns that address maintainability

attributes: Patterns for ease of modification, diagnoses, and comprehension," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 391–401, 2017.

[23] S. Levin and A. Yehudai, "The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 35–46, 2017.

[24] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, (New York, NY, USA), p. 643–653, Association for Computing Machinery, 2014.

[25] D. J. Kim, B. Yang, J. Yang, and T.-H. P. Chen, *How Disabled Tests Manifest in Test Maintainability Challenges?*, p. 1045–1055. New York, NY, USA: Association for Computing Machinery, 2021.

[26] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of java projects using continuous integration," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), p. 821–830, Association for Computing Machinery, 2017.

[27] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 433–444, 2018.

[28] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, *What is the Vocabulary of Flaky Tests?*, p. 492–502. New York, NY, USA: Association for Computing Machinery, 2020.

[29] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "Know you neighbor: Fast static prediction of test flakiness," *IEEE Access*, vol. 9, pp. 76119–76134, 2021.

[30] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," in *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 211–224, Association for Computing Machinery, Inc, 7 2020.

[31] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," 2014.

[32] P. Sedgwick, "Convenience sampling," *BMJ*, vol. 347, 2013.

[33] C. Rivas, *Coding qualitative data*, pp. 367–392. Sage, 01 2012.

[34] C. Treude and M.-A. Storey, "Effective communication of software development knowledge through community portals," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 91–101, 2011.

[35] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability," *IEEE Transactions on Software Engineering*, vol. 47, no. 3, pp. 595–613, 2021.

A

Interview Guide

Introduction

- In which team are you currently working and in which teams have you been working?
- What is your role within the team?
- What roles exist in the team?
- How much testing experience do you have (years/months)?
- How much ML-related testing experience do you have (years/ months)?
- What kind of testing is your team performing at the moment?
- How have you been involved in the testing process? For instance, writing tests, maintaining tests, etc.
- Who is responsible for the testing within the team?
- Test cases written earlier in the development of a project may need to be changed or maintained as the project evolves. What is your personal understanding and experience of the process of maintaining test cases?

Overview of Terms and Purpose of the Study

- Traditional systems vs. ML systems
- Definition of Test maintenance
- Purpose of the study
- Structure of the interview

ML Systems

- What are common difficulties or challenges when designing test cases for ML systems?
- What are difficulties or challenges specifically when updating or maintaining test cases for ML systems as the project evolves?
- Would you say that the difficulties and/or challenges of designing test cases have an influence on the maintenance issues you just described?
- Do you have an example of how to prevent these challenges/difficulties when designing or maintaining test cases?

- Do the characteristics of an ML system influence the maintenance of the test cases?
- Can you think of any other factors/challenges that can affect how test cases for ML systems are maintained?

Traditional Systems

- What are common difficulties or challenges when designing test cases for traditional systems?
- What are difficulties or challenges specifically when updating or maintaining test cases for traditional systems as the project evolves?
- Would you say that the difficulties and/or challenges of designing test cases have an influence on the maintenance issues you just described?
- Do you have an example of how to prevent these challenges/difficulties when designing or maintaining test cases?
- Can you think of any other factors/challenges that can affect how test cases for ML systems are maintained?

Comparison ML Systems and Traditional Systems

- When recapping the discussions about ML systems and traditional systems, what are the largest differences in testing ML systems and traditional systems?
- What are the largest differences in test maintenance of ML systems and traditional systems?
- Is it easier or harder to maintain test cases for ML systems? Why?
- Are you using any patterns/strategies when creating test cases in general to improve, for example, readability and maintainability?
- Are any of these specifically for maintenance purposes?

Detailed Questions

- Within your team, when is test maintenance performed and how is it conducted?
- Do tests of your team need to be maintained regularly?
- Do you know if test cases of other teams have to be maintained regularly?
- Depending on answers: Follow-up questions on concrete test cases

Idea to use ML Model Accuracy for Determining the Test Tolerance

- Presentation of the idea
- What do you think of this idea?
- Is the idea something that you think is valuable for your team or for the company in general?

Flaky Tests

- Have you/your team experienced issues with flaky tests (both traditional and ML systems)?
- If yes: Can you describe the flaky behavior of these test cases?
- If yes: Was the issue solved or do you still experience issues with the test cases?
- If not: Do you know of any specific characteristics of ML systems that can cause a flaky test?
- Do you see any connections between flaky test cases and test maintenance? I.e, do you think that the flaky behavior means that test cases must be updated often?