# Test Maintenance for Machine Learning Systems: A Case Study in the Automotive Industry

Lukas Berglund, Tim Grube, Gregory Gay, Francisco Gomes de Oliveira Neto
*Chalmers | University of Gothenburg*
Gothenburg, Sweden
(gusberglluu, gusgruti)@student.gu.se, greg@greggay.com, francisco.gomes@cse.gu.se

Dimitrios Platis
*Zenseact*
Gothenburg, Sweden
dimitris@plat.is

***Abstract*—Machine Learning (ML) systems have seen widespread use for automated decision making. Testing is essential to ensure the quality of these systems, especially safety-critical autonomous systems in the automotive domain. ML systems introduce new challenges with the potential to affect test maintenance, the process of updating test cases to match the evolving system. We conducted an exploratory case study in the automotive domain to identify factors that affect test maintenance for ML systems, as well as to make recommendations to improve the maintenance process. Based on interview and artifact analysis, we identified 14 factors affecting maintenance, including five especially relevant for ML systems—with the most important relating to non-determinism and large input spaces. We also proposed ten recommendations for improving test maintenance, including four targeting ML systems—in particular, emphasizing the use of test oracles tolerant to acceptable non-determinism. The study's findings expand our knowledge of test maintenance for an emerging class of systems, benefiting the practitioners testing these systems.**

***Index Terms*—Software Testing, Test Maintenance, Test Evolution, Machine Learning, Automotive Software**

## I. Introduction

Software testing aims to find faults in the behavior of a system-under-test (SUT) through the execution of test cases. Each test case applies input to the SUT and judges the subsequent output using a test oracle—embedded expectations on that output, often in the form of assertions. Testing is especially important in safety-critical domains, such as the automotive industry, since undetected faults can lead to life-threatening failures. The risks of failure intensify as systems grow more complex, as is the case with the emergence of autonomous vehicles. Machine Learning (ML) is at the core of the functionality of autonomous vehicles, supporting tasks such as collision avoidance and lane-keeping. A common form of ML, supervised learning, trains a model to make predictions using labeled training data [1]. Autonomous vehicles are an example of the emerging category of *ML systems*—software systems that contain components that depend on ML, e.g., that use a model as the basis of functional logic.

ML systems present new testing challenges. For example, ML algorithms and models are usually non-deterministic, introducing challenges in specifying expected output [1]. In addition, many ML problems have a large input space—e.g., potential traffic situations—making it challenging to identify failure-revealing test inputs [1]. Finally, common ways of measuring how well a system is tested (e.g., code coverage) cannot be applied to ML systems as behavioral logic is often embedded in models instead of code [1].

Tests created for a SUT typically require *maintenance* as the project evolves [2]. Tests must be updated when, e.g., requirements are refined, new functionality is implemented, or a model is retrained. While research has been conducted on test maintenance in the past (e.g., [2]–[4]), there is a lack of knowledge on factors that affect test maintenance for ML systems [5]. Past research for traditional systems may not be directly applicable due to differences in system design and behavior between ML and traditional systems.

The purpose of this study is to gain understanding of the factors that affect test maintenance for ML systems, as well as to make recommendations on how to minimize or improve the maintenance process. To that end, we have conducted an exploratory case study at Zenseact, a company specializing in software for autonomous driving. This study was motivated by three challenges encountered at Zenseact regarding test design and maintenance for ML systems. First, developers have encountered flaky test cases that yield inconsistent results. Second, it was noticed that test cases often had to be updated after retraining an ML model. Changed predictions caused test failures, causing testers to expend effort to communicate with other teams and understand whether the failure was fault-revealing or indicative of the need for test maintenance (i.e., updating the test oracle). Finally, ML systems are challenging to test at the unit level, as components cannot be tested in isolation without—at least—integration of the model.

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 our results with test maintenance factors reported in past research for traditional systems. Ultimately, we observed:

- Nine factors were identified that affect test maintenance for all systems. *Continuity* and *scenario setup* are especially relevant in the automotive context. Additional factors that affect maintenance for ML systems include

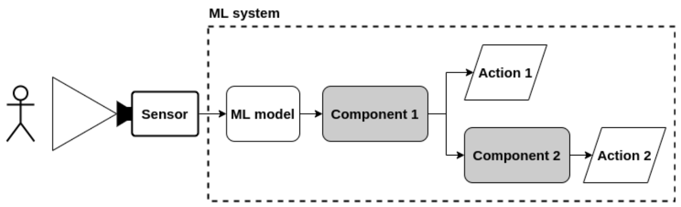

Figure 1: Example ML system, including the flow between model and components. A sensor provides input.

*amount and quality of training data*, *non-determinism*, *explainability*, *input space size*, and *testing granularity*.

- The main differences between ML and traditional systems are the *degree of determinism*, *explainability of the code*, and *scope and testing granularity*.
- It is not clear that flaky tests influence test maintenance for ML systems differently from traditional systems. ML systems result in more tests with flaky behavior—because the SUT is non-deterministic—but the tests themselves are not inherently more flaky.
- Discussion threads and test cases 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 challenges with regard to explainability and communication.
- *Knowledge sharing, maintainer tags, slack bots, test-driven development, detailed failure messages*, and a *unified scenario setup* can improve or reduce test maintenance for all systems. For ML systems, *tolerant oracles, forced determinism and isolation, consistent hardware*, and *property-based testing* can have a further impact.

Our findings offer insights into test maintenance of ML systems from an industrial context. We offer our observations and recommendations to researchers and practitioners to inspire future advances in this rapidly evolving area.

## II. Background

**ML Systems:** ML algorithms infer patterns from observations to make predictions in previously-unseen situations [1]. ML has become especially popular in the last decade, as new techniques can make complex predictions based on vast amounts of data. ML is regularly used to enable autonomy, support decision making, process images, and perform natural language processing, among other applications [1]. We discuss ML in terms of "models" and "ML systems". In supervised learning, a model is responsible for making predictions after being trained on pre-labeled data. In the context of autonomous vehicles, models often process images or sensor data to identify objects (e.g., pedestrians, lane markers, or traffic signs).

An ML system is a system whose components depend on ML—e.g., a model or a Reinforcement Learning agent [6]—to determine how the system behaves. Figure 1 shows an ML system where one component uses the output of the model and the second component uses the output of the first. As an example, an emergency braking system stops a vehicle if an object is detected. Object detection is performed by an ML model, and the predictions are acted upon by the emergency braking system components.

**Test Maintenance:** When changes are made to a SUT, the corresponding test cases must be updated. Test maintenance (or test evolution) is the act of maintaining or repairing test cases to keep them up to date with the requirements, design, and source code of the SUT [4], [7].

Testing is one of several factors that can affect the development cost of a system, consuming up to 50% of effort and cost [8], [9]. Test maintenance adds to this cost. For instance, Alégroth et al. found that maintaining test cases constitutes up to 60% of the time spent on testing [2]. Consequently, it is possible to significantly reduce development costs by designing test cases that require less maintenance and streamlining the maintenance process when it must occur.

**Flaky Tests:** A flaky test is a test that produces inconsistent results, and can fail even when the SUT has not been changed. Flaky tests disturb the flow in the CI pipeline and add significant noise to the process of determining whether a system is acting correctly. Parry et al. found that 59% of developers have to deal with flaky tests frequently [10].

Flakiness can be caused by test or source code, or other external factors such as a network connection—e.g., if a database cannot be reached, tests can fail. Even if an external factor caused flakiness, this could still indicate the need for the system or tests to be more robust. Another common cause for flakiness is dependency on a specific execution order for events. If the order of execution changes, resources might not be in the expected state, leading to a failure.

If a system is non-deterministic, this must be accounted for in test design. If flaky tests occur, a root cause analysis should be done. If the test itself leads to flakiness, it should be refactored. This requires maintenance effort, but is better than ignoring flaky tests, which leads to a costly long-term maintenance effort the longer the necessary refactoring is postponed [11].

## III. Related Work

There is limited research explicitly on test maintenance for ML systems [5]. However, past research has been conducted on testing ML systems and on factors affecting test maintenance for traditional systems. This research informs our efforts.

**Challenges of testing ML systems:** In Table I, we collect challenges identified in past studies on testing ML systems. In contrast to traditional systems, characteristics like non-determinism and large input spaces are prevalent for ML systems, increasing the difficulty of testing.

**Factors affecting test maintenance:** Factors from studies exploring test maintenance for traditional systems are presented in Table II. We compare our findings for ML systems against these factors in Section V-D.

Table I: Challenges of testing ML systems, from literature.

| Challenge | Description | Ref. |
|---|---|---|
| Non-determinism and Test Oracles | When retraining on the same training data, there is no guarantee that the model delivers the same output. This makes it harder to define a test oracle. | [1], [12], [13] |
| Large Input Space | It is difficult to find valid, representative, realistic, or fault-revealing test input. Identifying enough valuable input is costly. | [1], [12], [13] |
| Model Mispredictions | Tests need to be tolerant against mispredictions from a model, as long as the system still fulfills the requirements. | [1] |
| Code Coverage Not Relevant | Since at least parts of the logic is within the model and not code, traditional coverage criteria are not effective in evaluating test suites. | [1], [12] |
| Difficulties in Debugging and Understanding | Since parts of logic are within the model, stack traces may not enable analysis of the SUT. It is challenging to understand behavior and failures. | [1] |

Table II: Test maintenance factors, from literature.

| Factor | Description | Ref. |
|---|---|---|
| Knowledge and Experience | Tester experience and knowledge (both general and of the SUT) may affect optimality of test design and ease of maintenance. | [2] |
| Variable Names, Script Logic | Names and logic are connected to understandability and readability of tests and can affect the ability to perform maintenance. | [2] |
| Test Length and Complexity | Long tests are often complex and test several scenarios simultaneously. Branching paths in tests can increase complexity. This makes it difficult to gain a clear overview, affecting readability, understandability, and maintainability. | [2], [14] |
| Missing Functionality | Functionality that is not yet implemented can hinder the testing process. It can be hard to maintain existing test cases until functionality is added. | [2] |
| Faults in 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 fixed. | [2] |
| Suite Size | A large test suite can increase the amount of test maintenance effort. Tests can also overlap, which increases the maintenance needed for updates to the same functionality. | [4] |
| Interaction Quantity | The number of input interactions affects the maintenance effort. If behavior or requirements of a component change, then many test inputs must also change. If a test with many interactions fails, more effort will be needed for analysis than for a test with few interactions. | [4] |
| Test Debuggability | Tests should be informative and help explain the cause of a failure. If not, maintenance cost can increase due to additional time spent on understanding the test and failure. | [4] |
| Test Interdependency | Tests may rely on each other. If one fails, others may fail as well. Interdependencies are fine if the tests are created with this in mind and the result is examined correctly. However, this complicates understandability and maintainability. | [4], [14] |
| Naming Conventions | Naming conventions can increase consistency and mitigate the risk of confusion due to low understandability. As such, naming conventions can make test maintenance easier. | [4] |
| Test Documentation | Documentation should give a general overview of the test case and its purpose, which helps improve understandability while performing maintenance. | [4] |

**Flaky tests and test maintenance:** In traditional software systems, flaky tests are a well-known problem, introducing noise in fault analysis and hindering reproducibility of test results [15]. In an ML context, Dutta et al. found that algorithmic non-determinism commonly led to test flakiness, and that it could be fixed by using a more permissive test oracle [16].

Flaky tests impose a maintenance cost if time is taken to fix them [17]. Tests can be disabled if they cannot be immediately fixed, but this also imposes a long-term technical debt and maintenance cost [11].

## IV. 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 systems (traditional and ML)?
  - **RQ1.2:** What factors affect test maintenance specifically for ML systems?
  - **RQ1.3:** What are the main differences in test maintenance between ML and traditional systems?
  - **RQ1.4:** How do flaky tests affect test maintenance for ML systems?
- **RQ2:** How are the identified factors reflected in the test design and maintenance process?
- **RQ3:** How can the test maintenance process for ML systems be improved?

We answer these questions through an exploratory case study conducted at Zenseact, a Swedish company in the automotive industry specializing in autonomous vehicle software. We conducted our study based on the guidelines by Runeson and Höst [18]. Figure 2 presents an overview of the case study process and the artifacts used for different research questions. Each step is further explained below.

- **Conduct literature study:** We gathered test maintenance factors for traditional systems and challenges for ML system testing from past research (Section IV-B).
- **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 IV-C).
- **Collect and analyze artifacts:** Test cases and discussion artifacts were collected and analyzed to evaluate the effect of test maintenance factors in the testing process (Section IV-D).
- **Form recommendations:** We synthesize our findings into guidelines on how to potentially lower the need for, and cost of, test maintenance.

### A. Case Study Context

Zenseact's system includes the whole stack, from low-level sensors to decision-making based on sensor data. ML is, for example, used in combination with data input from cameras

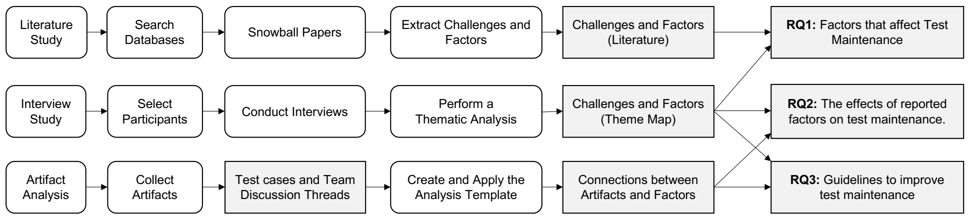
Literature Study
Search Databases
Snowball Papers
Extract Challenges and Factors
Challenges and Factors (Literature)
RQ1: Factors that affect Test Maintenance
Interview Study
Select Participants
Conduct Interviews
Perform a Thematic Analysis
Challenges and Factors (Theme Map)
RQ2: The effects of reported factors on test maintenance.
Artifact Analysis
Collect Artifacts
Test cases and Team Discussion Threads
Create and Apply the Analysis Template
Connections between Artifacts and Factors
RQ3: Guidelines to improve test maintenance

Figure 2: Overview of the case study, including the activities (rounded rectangles) and corresponding deliverable (rectangles).

and sensors to make precise inferences about the surroundings and detect objects.

The usage of ML in the company is divided into three stages. First, ML teams are responsible for creating and training models based on input from low-level sensors and cameras. In a second step, the produced result is filtered and processed by the ML teams to prepare the data for the consuming systems. Finally, the refined output is used to perform functionality such as emergency braking and steering. These are the system components responsible for deciding when an intervention—e.g., braking the car—is required. This functionality is developed by separate "feature teams". In this case study, we focus on feature teams and their test cases.

The models are continuously refined to improve efficiency and accuracy. While doing this, the behavior of the models can change, which can lead to test failures. An example is timing problems that emerge when tests expect an intervention at a specific timestamp. Hence, they might fail if the updated model affects when the intervention takes place. In such cases, the SUT behavior may still meet the requirements, but the specific expectation of the test case is violated.

## B. Literature Study

A literature study was conducted to collect test maintenance factors and challenges of testing ML systems encountered in the past, for comparison with our findings. Past research also indicated the need for further exploration and validation of test maintenance factors, which is also provided by our case study [2]. The literature study was intended to be lightweight, and to provide inspiration and a basis for comparison. It was not intended to fully capture all possible studies on this topic, as would be done for a full systematic literature review.

**Database Search:** We gathered publications from Google Scholar using search strings related to test maintenance and testing ML systems (including synonyms such as "test evolution"). Publications with the following keywords in the title, introduction, or conclusion were selected to form a starting set [19]: *challenges, difficulties, factors, maintainability, maintenance, or repair*. After evaluation, several of the papers in the set were found to be irrelevant and were removed.

**Snowballing and Literature Analysis:** We used backward and forward snowballing on the filtered starting set of papers [19]. We selected additional papers when snowballing following the same criteria used for the database search. The process was conducted iteratively. After filtering for topic relevance, we retained 6 relevant publications, where three cover the challenges of testing ML systems and three discuss test maintenance factors for traditional systems. Most of the publications defined lists of factors and challenges. However, some factors or challenges were domain or project-dependent, and were excluded from our analysis (e.g., maintenance factors that exclusively apply to GUIs [2]). Lastly, we merged similar factors and challenges resulting in the lists from Tables I-II.

Table III: Demographics of interviewees. MLS=ML System.

| ID | Roles | Experience in Years | | |
|---|---|---|---|---|
| | | Zenseact | 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 |

## C. Interview Study

**Selection of Interviewees:** The selection of the interviewees was based on convenience sampling [20]. However, we took some measures to select interviewees with knowledge of the research topic. We contacted teams based on their descriptions in internal documentation and recommendations of Zenseact employees. We were interested in teams that work with ML systems and have tests that depend on ML models directly or indirectly. Nine employees from five teams agreed to participate in interviews. A summary of the participants and their background is given in Table III[1].

**Interview Instrument:** We conducted semi-structured interviews with a length of approximately 60 minutes. Interviews were conducted remotely, and participants agreed to be recorded. The interview consisted of three parts[2]. The first section covers general questions about the interviewee's experience and how testing and test maintenance is handled in their team. The second section deals with challenges of testing ML systems and traditional systems, how these challenges affect test maintenance, how challenges can be mitigated,

[1]Zenseact, in its current form, was founded in 2020. Experience at the company includes both experience at Zenseact, as well as employment at predecessors Zenuity (2017-2020) and Volvo Cars (until 2017).

[2]The interview guide can be found in [21] (Appendix A).

Table IV: Questions used to guide artifact analysis.

**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?

**Questions only for 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?

**Questions analyzing test history, using Git history and commit messages:**

- 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?

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 third section, the reasons why flaky tests occur were discussed, as well as 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 transcribed and anonymized to remove sensitive material. After that, important parts of the interviews were identified. These sections ("codes") are assigned a label that allows clustering of codes into themes and sub-themes. We applied open coding [22], where codes are summarized with a label in our own words. We followed an inductive coding approach [22]. Two authors coded and labelled all interviews individually, then compared results to mitigate the effects of bias.

After coding and labeling, we clustered the results to form a theme map. The themes represent our findings, such as factors affecting maintenance. Even though we familiarized ourselves with the topics, we did not predefine themes.

### D. Artifact Evaluation

**Artifact Selection:** We selected test cases based on information from the feature teams who participated in interviews. We identified six relevant test cases and a further two already-replaced test cases that help support analysis of test evolution. Discussion threads were also collected from Zenseact's primary communication tool. Discussions of interest were connected to at least one factor in RQ1 and, at times, included relevant test cases. We received recommendations on relevant discussions, and also searched in the communication tool for the keywords *maintenance, update, tolerance, failure, factor, flaky*, and the names of relevant test cases.

**Artifact Analysis:** The artifact analysis was qualitative, and based on three sets of questions similar to those used in past research [23]. The questions are listed in Table IV.

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 test evolution. Answering these questions enabled linking artifacts to the factors from RQ1. Finally, we analyzed the factors to report how they affect the testing process at Zenseact, and support these observations with concrete examples from the artifacts.

## V. Results and Discussion

### A. Test Maintenance Factors (RQ1)

**Factors Affecting Traditional and ML Systems:** The identified factors affecting test maintenance for all systems are presented and explained in Table V.

**RQ1.1 (Factors, All Systems):** Nine factors affect test maintenance for all systems. *Continuity* and *scenario setup* are especially relevant in the automotive context.

**Factors Affecting ML Systems:** Table VI presents and explains identified test maintenance factors relating specifically to ML systems for autonomous driving.

**RQ1.2 (Factors, ML Systems):** Factors that affect test maintenance for ML systems include *amount and quality of training data*, *non-determinism*, *explainability*, *input space size*, and *testing granularity*.

**Comparing Factors Between Traditional and ML Systems:** Test maintenance factors for traditional systems also affect ML systems. In addition, specific characteristics of ML systems introduce unique factors described below. An ML system behaves differently than a traditional system, mostly because of non-determinism and lack of transparency of models. This creates unpredictability that makes it more challenging to design and update tests. It can also be challenging to get an overview of the system and break it into smaller pieces due to dependencies between components and development teams.

*Degree of Determinism:* Non-determinism makes it challenging to design test cases. It also leads to more maintenance, often requiring adjustment of the test oracle. It is more important to design test oracles that are tolerant of some non-determinism, and that do not expect a precise value. Several tests at Zenseact expect interventions at certain timestamps. Rather than expecting an intervention at exactly 20 seconds, it may be better to establish an acceptable interval (e.g., +/- 200 ms) around this timestamp where the intervention can occur.

Over-precision of an oracle can affect any system where variance can occur, but ML systems are more non-deterministic by nature. Defining the correct tolerance can be challenging as well, but consideration of it can significantly reduce maintenance in the longer term.

> "*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

Table V: Test maintenance factors affecting traditional and ML systems.

| Factor | Definition | Impact on Maintenance |
|---|---|---|
| Communication | How easy it is to communicate between teams regarding testing and test maintenance activities. | Teams responsible for different features must collaborate. Poor decisions on how and when to communicate when a test case fails or needs to be updated increases maintenance effort. |
| Consistency Between Teams | How consistent testing practices and tools are applied across teams. | Low consistency in testing practices and tool use can lead to higher maintenance efforts due to overlapping work, inconsistent testing thoroughness, unclear responsibilities related to development and testing activities, and ineffective collaboration between teams. |
| Continuity | How much test cases, practices, and tools change over time. | Changes in test cases, practices, or tools over time are unavoidable and even desirable when they bring testing in line with state-to-the-art. Nevertheless, these changes usually require adaptation of existing test cases. Testers also must learn to use new tools, requiring effort. Decisions of when to change the testing process must be considered carefully. E.g., changes in simulation tools introduced maintenance effort. |
| Debugging Support | Ability of test cases to support a failure analysis with insights into the underlying system. | If a test does not support failure analyses sufficiently, it may need to be maintained to improve its observability into the SUT. E.g., after changes to a dependency caused tests to fail, affected tests were updated with checks on dependency output. |
| Dependency on Implementation | Degree of dependency of test cases on implementation details (e.g., tests written based on the code instead of intended behavior). | Tests can be written based on existing source code or behavioral expectations (e.g., requirements). When there is a high dependency on implementation details, tests need to be updated more often as they are highly sensitive to changes in the code. |
| Oracle Precision | The precision of the output expected by the test oracle. | Sensitive test oracles (e.g., an oracle that requires specific timing when variation is possible) require frequent updates to account for variance in SUT output. Precise expectations are also hard to update, as detailed feature knowledge is required to know what precise values to expect after an update. |
| Scenario Setup | The source of information for scenarios covered in test cases (simulations or real-world data collection from a road vehicle). | Scenario creation techniques differ in their realism and the time required to create test cases and perform maintenance. Scenarios from real-world data are more realistic, but data is time-consuming to collect as a human must collect it. It can also be hard for humans to replicate the intended scenario. In contrast, scenarios created from simulations may lack realism and require human effort to evaluate and improve realism. Simulations have complex configuration parameters that may be difficult to understand or use correctly during test creation or maintenance. |
| Understandability | How easy it is to understand the purpose of a test, 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 Maintenance is Performed | The reason for, and point in time when, test maintenance is performed. | Depending on the way of working, this factor can lead to technical debt (if maintenance is not done regularly) or more work than necessary (if tests are always refactored following every code change). Consideration must be given to the conditions that trigger test maintenance. |

Table VI: Test maintenance factors affecting ML systems.

| Factor | Definition | Impact on Maintenance |
|---|---|---|
| Amount and Quality of Training Data | The quality of ML models strongly depends on the amount and quality of training data. | Early versions of a model often have low accuracy and improve over time with retraining on additional data. Tests need to be designed to deal with this uncertainty and maintained accordingly. To avoid early test failures, tests should be designed with a more tolerant oracle. Later, stricter oracles should be employed. There are also challenges associated with maintaining data quality and with required processing time and power as datasets grow. |
| Non-determinism | ML algorithms and models can yield non-deterministic output even with the same setup and input. Output can differ even after re-training with the same training data. | Varying output due to non-determinism must be factored into test design to minimize later maintenance effort. Both software and hardware (e.g., GPUs running at different clock speeds) can induce non-determinism. Oracles should be as flexible as possible, while ensuring requirements are met. Early in the project, failing tests can be disabled rather than maintained. However, these tests must eventually be refactored or removed, or they will lead to technical debt. |
| Explainability | The ability to explain the intended system behavior from looking at the code and the model output. | ML systems are considered black-box systems due to their reliance on complex, non-transparent models. This makes it hard to design test cases by looking solely at code or requirements, leading to maintenance effort redesigning and adjusting tests. There is a lack of traceability between tests and distinct code elements. |
| Input Space | Describes the size of the input space. The input space of ML systems is often huge (e.g., environments surrounding a vehicle). | Many test cases are required to cover a reasonable variety of input from a large and complex input space, and it is challenging to find all important edge cases. Consequently, many tests may need to be maintained, and the test suite needs to be updated when new edge cases are identified. |
| Testing Granularity | The level of code granularity tests are written at (i.e., unit, integration, system tests). | Test design at unit level for ML systems is challenging, as modeling complex scenarios requires model integration and multiple interacting components. Tests at system level are easier to design, but introduce complex interdependencies where a change to one component causes tests for other components to fail. Performing maintenance at system level requires detailed knowledge of many components. High code modularity increases understandability and decreases maintenance effort. |

*Explainability:* While traditional systems can be understood by inspecting the code, much of the logic of ML systems is hidden within opaque models. This makes it challenging to understand processing between input and output. Even though ML systems have similar development challenges to traditional systems, the lack of explainability of the code makes those challenges harder to handle.

*Scope and Testing Granularity:* Traditional systems are tested at multiple levels of granularity, enabling separation of concerns and testing of components in isolation. ML systems are difficult to test at lower—e.g., unit—levels, as integration of ML elements such as models typically requires integration of multiple system components. This necessitates that more testing take place at higher—e.g., system—levels.

It is generally harder to create test cases of isolated portions of the SUT at higher levels, as many dependencies exist. If there

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

| Cause | Description |
|---|---|
| Test Teardown and Execution Order | If tests manipulate shared resources (e.g., a database), execution of other tests may be affected. Every test should reset data and shared variables after execution (teardown). Otherwise, a test can fail because it is executed after a test without proper teardown. |
| Infrastructure | Sudden infrastructure issues can lead to temporary test failures—e.g., network issues, overloaded servers, and timing issues when building components. |
| Parallel Test Execution | Parallel test execution can lead to data races and synchronization issues—e.g., if thread A changes a value in a database and thread B expects another value, thread B's test fails. If the timing is different, it would pass. |

Table VIII: Approaches to handle flaky test cases.

| Approach | Description |
|---|---|
| Analyze Flakiness | 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 Tests | Rerun a test until it passes or fails a set number of times. However, this requires additional time, reduces trust in the test, and may hide faults. |
| Remove Test | A viable solution if flakiness is due to the test being deprecated. |
| Configure CI | Make the CI system tolerant of failures of the test. However, that lowers meaningfulness of the test and adds noise to analysis. |
| Increase Awareness | Bots can report test results and provide information on whether a test has shown flaky behavior before. |

is a failure, it can be hard to trace it to a single component. If a dependency changes, tests for an unchanged component can fail. All of this introduces challenges when testing ML systems that are less of a concern when testing traditional systems, where testing can more easily take place at unit level.

**RQ1.3 (Factors, Comparison):** ML and traditional systems differ in their *degree of determinism*, *explainability*, and *scope and testing granularity*.

**Effect of Flaky Tests on Test Maintenance:** Our interviews did not suggest that flaky tests affect ML systems differently from traditional systems. However, flaky tests are still a factor that influences the cost of test maintenance for all systems. Identified causes for flaky tests are described in Table VII, while ways to handle them are described in Table VIII.

The connection to test maintenance can be seen in two aspects. First, flaky tests cause repetitive failure analyses, as developers must assess whether the test itself or the SUT is responsible for flakiness. Second, if the test causes flakiness, the test must be refactored. Redesigning and updating the test cases adds to the test maintenance effort. Interviewees saw flakiness as more of a test design concern than a maintenance concern—stressing the importance of designing tests to be robust and deterministic in the first place.

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

The interviewees noted that ML systems have more *tests with flaky behavior* than traditional systems. After retraining a model, or when executing tests on different GPUs, tests may yield different results even though neither the tests nor SUT code were altered. The core problem is the non-determinism of the SUT. ML systems do not tend to have more flaky tests than traditional systems, but they may have more tests with flaky behavior. Test design must account for such non-determinism to avoid flaky results.

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

| Thread | Summary | Tests | Factors |
|---|---|---|---|
| D1 | A test had a too strict oracle, which made it fail when updating other features. Maintenance effort was required to evaluate the new feature behavior before the oracle could be updated. | T3, T3.1 | Non-determinism |
| D2 | Thread about how to design end-to-end tests and collaborate between teams to neither disturb ML developers' workflow nor reduce feature team test quality. | T3 | Non-determinism, Communication, Testing Granularity, Explainability |
| D3 | 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, who increased the oracle's tolerance based on their feature knowledge. | T1 | Non-determinism, Communication |
| D4 | An ML team contacted a feature team to get help with a test failure. There was difficulty explaining why this specific code change exceeded the oracle tolerance—the main reason could be an already-integrated code change. | T3 | Non-determinism, Explainability |
| D5 | An ML system test case was affected by a dependent feature change, leading to a test failure and the need for test maintenance. | T1 | Testing Granularity |
| D6 | Discussion on whether the oracle tolerance should be increased after a test failure. 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 |

**RQ1.4 (Factors, Flaky Tests):** Flaky tests seem to not influence test maintenance differently for ML systems. ML systems result in more tests with flaky behavior—because the SUT is more non-deterministic—but the tests are not more inherently flaky.

## B. *Effect of Test Maintenance Factors (RQ2)*

Analyzed discussion threads are listed in Table IX, where each discussion thread is connected to one or more test cases. Data about analyzed test cases is shown in Table X, including the age of the test, whether it is currently in use, how many times it was updated, how many people updated it, and how many times the oracle tolerance was changed[3].

*Non-Determinism:* Many maintenance issues are caused by retraining the ML model. Table X shows that test maintenance frequently involves an update to the oracle tolerance. The

[3]Test T6 has a strict oracle with no tolerance for varying output. The other tests all have some tolerance built into the oracle.

Table X: Overview of analyzed test cases. Tests T3.1 and T3.2 were merged to form T3.

| 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 Tolerance |

frequency of oracle updates indicates how challenging non-determinism can be in test design and maintenance.

In artifact D2, it was discussed how a ML system, including the model, should be tested. One 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 worsen. Therefore, in addition to statistical measures, critical concrete scenarios should still be checked (e.g., checking that emergency braking intervenes).

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

Since the model's overall accuracy can differ from correctness in a specific scenario, tests need to be designed robustly. An interval must be defined for acceptable timing and number of interventions made by a feature. Four approaches for assessing correctness were seen in the analyzed test cases. T1, T2, and T5 checked the number of sent intervention requests, while T3 checked when the first intervention request was triggered. T4 used reference GPS data to set a distance range for braking. Finally, T6 used an exact, but simple, oracle that checked whether there was an intervention request at all.

T4 and T6 were rarely updated, while T1–T3, and T5 were updated many times to adjust the oracle. Much analysis time was spent on T1–T3 to determine whether the tolerance was too strict or if there was a fault. The approach used for T4—using recorded data and precisely calculated tolerances—avoided later maintenance effort. If a test fails, it should indicate a fault and not an inadequate oracle tolerance. However, this can be difficult to define in the early phases of a project. In D2, participants raise the need for experimentation during development of the model. It might be necessary to exceed the tolerance until the model fulfills all requirements. This additional "experimental" tolerance must be distinguished from the tolerance that describes the actual acceptable behavior range. D2 suggests that when a model change breaks a test, a specific data set should be collected for use together with a tool to visualize the influence of the model change on the feature. Then, a decision can be made on whether a tolerance adjustment is acceptable.

*Testing Granularity:* Interdependencies between components affects T1 and T3, and were discussed in D5. A change made by Team A to one of their features caused T1, which was owned by Team B, to fail. Because T1 was a system-level test, it was sensitive to changes in any of its connected components. This is not necessarily negative, but can complicate test analysis and add noise to test execution results.

*Explainability:* D2 and D4 discussed the inability to understand the code alone. Model opacity made it challenging to understand the code's behavior, which affected the outcome of test cases and increased test maintenance. Developers on ML teams understood the model better than developers on other teams, and could explain the behavior. This is an example of why it is important to have good communication and knowledge sharing, e.g., documentation of tests and code.

*Communication:* Discussions D2–D3 were particularly affected by team communication. D2 discussed a failure in a feature test caused by non-determinism from retraining of an ML model. Consequently, the ML team had to contact the responsible feature team to inform them about the updates that made the test fail. There was no clear process to communicate this kind of test failure. Consequently, there was confusion in the ML team on whom to contact and when to do it, which affected the test maintenance time and hindered development. Agreements on communicating test failures were later visible in D3, where the process discussed in D2 was applied. This is also reported by participants in our interview data.

> "*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

**RQ2 (Effect of Factors):** Discussion threads and test cases show the influence of *non-determinism*, *testing granularity*, *explainability*, and *communication* on test maintenance. In particular, *non-determinism* creates the need for maintenance and introduces challenges with regard to explainability and communication.

### C. *Recommendations for Test Maintenance Process (RQ3)*

This section presents recommendations for reducing the need for test maintenance or for improving the maintenance process, based on observations made while answering RQ1 and RQ2.

**Recommendations for ML Systems:**

*Use Tolerance in the Test Oracle:* Defining and using a suitable tolerance rather than a precise oracle is especially important in an ML system to handle non-determinism. In almost all cases in Table X, the tolerance was updated in $>70\%$ of test updates. Therefore, selecting the correct tolerance is tightly connected with the total amount of test maintenance required.

Too large of a tolerance can make a test case insensitive to actual faults. It is crucially important—given the safety-critical nature of automotive systems—that faults be detected. Care *must* be taken in defining a tolerance to ensure that the risk of missing a fault is minimized. It is better to be strict, even if some test maintenance effort remains. However, careful definition of oracles can reduce the need for test maintenance while ensuring faults are detected.

One way to better define a tolerance is by collecting statistical data from a real-world situation. For instance, test T4 uses reference data from a car with a high-precision GPS, which was used to determine a range for the braking distance in a particular situation. In Table X, it can be seen that the tolerance in T4 has not been updated. The other test cases did not use reference data for the tolerance, which can be one reason for higher maintenance efforts. However, collecting such data is not always possible or may be too costly, so additional research is needed on defining tolerance.

Early in development, the model can be less precise and produce more significant output variations compared to later in development. A test that is too strict at the beginning can hinder the development process. Therefore, it needs to be examined whether (a) test failures are accepted and useful for providing feedback, (b) the tolerance should be allowed to be exceeded temporarily (while being closely observed), or (c), the oracle should check properties of the output rather than specific values. Ideally, tolerances should represent absolute boundaries in behavior that should not be crossed, indicating that any violation represents a situation that must be addressed.

*Force Determinism / Isolation:* A component can be tested in isolation from the model to force determinism. This can be done by mocking the output from the model when testing. We identified two mocking strategies at Zenseact. First, the output of models can be replaced with manually-selected mocked values that represent a scenario that the feature needs to react to. These mocked values are fixed and only change when it is decided that they need to be updated. Another strategy is to record model output and save it in log files, which can later be used to mock the model output. Compared to the first approach, the data is less prone to human bias and may be less time consuming to collect. The disadvantage is that data is limited to the scenarios represented in the log file.

This approach allows for lower-level testing or components in isolation, which can reduce maintenance effort. Nevertheless, end-to-end tests cannot and should not be avoided since only those tests can ensure that everything works together appropriately. In addition, consideration must be given to when mocked input should be updated—it should reflect the current model, especially if major changes have occurred.

*Use Consistent Hardware:* The performance difference between hardware platforms (e.g., GPU architectures) can affect SUT behavior in tests related to performance or timing. This is especially important in automotive since the hardware in a vehicle may not be same as that used during development and testing. In cases where hardware specifications are changing rapidly, simulation should be used heavily, as it can be more easily controlled. However, hardware-in-the-loop testing is also needed, and the hardware used should match the final hardware as closely as possible.

Table XI: Recommended improvements for all systems.

| **Rec.** | **Maintenance Improvement** |
|---|---|
| Knowledge Sharing | General knowledge sharing activities like a book circle and mob reviews can improve shared understanding of the SUT and teach developers about new testing tools or practices. |
| Maintainer Tags | A maintainer tag (information about the responsible person or team) in the test case can ease communication by increasing understanding of whom to contact. |
| Slack Bots | Slack bots can share information about test failures, flaky test cases, and other testing issues. This allows for quicker information sharing among developers, reducing maintenance time. |
| Test-driven Development | TDD can improve the quality of tests and ensure they reflect the requirements, reducing dependency on implementation details. |
| Failure Messages | Using detailed failure messages in tests increases understandability and provides insights into the system at the time of the failure. |
| Unified Scenario Setup | Complex scenarios require configuration, which can be difficult to maintain. Unifying the scenario setup, e.g., by providing a framework or template for setup, encapsulates the required logic. Complex parts of the scenario setup only need to be maintained in one place, which is also advantageous when refactoring tests. |

*Property-Based Testing (PBT):* PBT frameworks generate a large volume of random inputs, which are applied to the SUT. The output is evaluated by oracles, expressed in the form of properties that can be applied to any input situation [24]. PBT enables coverage of large input spaces without the need to manually maintain all tests. Zenseact uses PBT already in some circumstances. However, PBT should not be seen as a replacement, but as an addition, to unit testing.

**Recommendations for All Systems:** Additional recommendations for all systems are listed and explained in Table XI.

**RQ3 (Recommendations):** Knowledge sharing, maintainer tags, slack bots, test-driven development, detailed failure messages, and a unified scenario setup can improve or reduce test maintenance for all systems. For ML systems, tolerant oracles, forced determinism and isolation, consistent hardware, and property-based testing can have a further impact.

## D. Comparison With Existing Literature

**ML Testing Challenges:** In Table I, we identified five challenges of testing ML systems with relevance to test maintenance in past literature. The most important challenge is non-determinism and, consequently, the difficulty of defining a test oracle for varying model outputs [1], [12], [13]. This has been confirmed by the observations in our study. Another challenge from the literature are model mispredictions, which a ML system must be tolerant against while still ensuring requirements are fulfilled [1]. Similarly, tests must also be tolerant to non-determinism.

The challenge of a large input space is also reflected in our observations. A large input space leads to challenges, such as finding effective test input. The two remaining challenges are the limitations of coverage-based testing and difficulties in debugging and understanding. Both are caused by lack of model transparency. These five challenges from literature also affected test maintenance in our study. In general, the factors and challenges identified tend to be domain-specific rather than specific to the partner company.

**Maintenance Factors:** We list 11 factors (Table II) that affect test maintenance observed in past studies which, in turn, highlight the need to confirm the impact of these factors [2]. Our case study contributes to this need. The factors *variable names and script logic*, *interdependences between tests*, *naming conventions*, and *test documentation* connect to the *understandability* factor from our case study. Documentation and naming conventions are tightly connected with the degree of understandability of a test [4]. The ability of a test case to support debugging also was observed in both past literature and in our case study. Both literature (*knowledge/experience*) and our observations suggest a clear need for knowledge sharing and communication for improved test maintenance.

Other literature factors—*test length and complexity*, *missing functionality*, *faults in SUT*, *suite size*, and *interaction quantity*—were briefly mentioned by individual interviewees, but not encountered to a broader extent in our study. However, they are still important to consider. We also identified further factors for validation and deeper exploration in future research.

**Flaky Tests:** Dutta et al. [16] found that most flaky tests in ML systems occur due to SUT non-determinism rather than the tests themselves being flaky. We came to similar conclusions, where non-determinism—especially from model retraining—led to flaky behavior, rather than factors of test design. In the literature, there are indications that flaky tests influence test maintenance, especially when postponing analysis [11]. Our study confirmed this, although some interviewees thought of flaky tests as primarily a test design issue.

## VI. Threats to Validity

**Construct Validity:** Multiple terms are used for test maintenance (e.g., test evolution) and ML system (e.g., ML-enabled system). Hence, the interpretation of terms could have differed between us and the interviewees. We mitigated this threat by giving an introduction to terms during interviews.

**Internal Validity:** Discussions before interviews, and interview questions themselves, could have planted suggestions on factors that influence test maintenance. We mitigated this risk by not stating possible factors ourselves. Another risk was not finding valid or sufficient artifacts in the relatively large codebase. We addressed this issue by contacting employees in a discussion channel and meeting with employees with expertise in testing and ML. Since a wide range of topics needed to be covered, there is a risk that the interview guide was not focused enough or did not include all relevant questions. In addition, even though we conducted semi-structured interviews, we might have missed opportunities to expand on topics. We mitigated these risks through iterative refinement of the interview guide. We asked additional questions during the interviews if it made sense.

**External Validity:** The study was only conducted at one company, and the results may not be applicable to other contexts. We formulated the factors and suggestions so that they should apply, at least, to companies from the same domain. The results of RQ2 are more company-specific than RQ1 and RQ3. However, the observations should also be of interest to others. The literature comparison shows that many of the factors identified in this study confirm past literature.

**Reliability:** Thematic analysis introduces risk of biased interpretations. The authors mitigated this risk by coding separately before comparing results. If there were doubts about a statement, the interviewee was contacted for clarification. This study is based on a relatively small number of interviews and artifacts. However, multiple Zenseact employees were able to provide feedback and clarification to address ambiguities. The analyzed artifacts were identified by several employees and seemed to be the most relevant. Interviewees were also selected from several different teams to be representative.

## VII. Conclusion

In this study, we explored test maintenance for ML systems. We identified 14 factors that affect test maintenance, including five especially relevant for ML systems. The most frequently-mentioned factor was non-determinism, which led to frequent test case updates—particularly adjustments to test oracles. Finally, we reported ten recommendations that can help improve test maintenance. Four of them (test oracle tolerances, 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 a large input space—and the corresponding challenges of ML testing.

We hope to inspire future research on test maintenance for ML systems. Many of our recommendations are not simple to implement, and further exploration will reveal how to best perform these actions. Therefore, future work aims to explore test maintenance at more companies and domains outside the automotive industry to validate the factors identified. In addition, a longer-term study on how test maintenance differs over time between traditional and ML systems would benefit knowledge in this area, as would quantitative studies on the impact of oracle tolerances on test maintenance. Automated methods of adjusting oracle tolerance could also be explored.

## Acknowledgements

We thank Zenseact for their support and collaboration throughout our study. We are also grateful to all participants in our interview study for their availability, time and invaluable input to our research.

## References

[1] 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.

[2] 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584916300118

[3] 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)*, 2017, pp. 391–401.

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

[5] H. B. Braiek and F. Khomh, "On testing machine learning programs," *Journal of Systems and Software*, vol. 164, p. 110542, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220300248

[6] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning, Second Edition An Introduction*. Bradford Books, 2018.

[7] M. Skoglund and P. Runeson, "A case study on regression test suite maintenance in system evolution," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 438–442.

[8] 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.

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

[10] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, oct 2021. [Online]. Available: https://doi.org/10.1145/3476105

[11] D. J. Kim, B. Yang, J. Yang, and T.-H. P. Chen, *How Disabled Tests Manifest in Test Maintainability Challenges?* New York, NY, USA: Association for Computing Machinery, 2021, p. 1045–1055. [Online]. Available: https://doi.org/10.1145/3468264.3468609

[12] 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)*, 2019, pp. 101–102.

[13] 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.

[14] 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)*, 2017, pp. 391–401.

[15] 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*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 643–653. [Online]. Available: https://doi.org/10.1145/2635868.2635920

[16] 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*. Association for Computing Machinery, Inc, 7 2020, pp. 211–224.

[17] 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*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 821–830. [Online]. Available: https://doi.org/10.1145/3106237.3106288

[18] 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.

[19] *Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering*, ser. EASE '14. Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2601248.2601268

[20] P. Sedgwick, "Convenience sampling," *BMJ*, vol. 347, 2013. [Online]. Available: https://www.bmj.com/content/347/bmj.f6304

[21] L. Berglund and T. Grube, "Test maintenance for machine learning systems: A case study in the automotive industry," Master's thesis, University of Gothen-

burg, 2022, available from https://greg4cr.github.io/pdf/22thesis_mltest.pdf.

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

[23] 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*, 2011, pp. 91–101.

[24] J. Hughes, “Software testing with quickcheck,” in *Central European Functional Programming School - Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, 2009, pp. 183–223. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17685-2_6