Exploring the Integration of Large Language Models in Industrial Test Maintenance Processes

LUDVIG LEMNER* and LINNEA WAHLGREN*, Chalmers University of Technology and Ericsson AB, Sweden
GREGORY GAY, Chalmers and the University of Gothenburg, Sweden
NASSER MOHAMMADIHA, Ericsson AB and Chalmers University of Technology, Sweden
JINGXIONG LIU, Ericsson AB, Sweden
JOAKIM WENNERBERG, Ericsson AB, Sweden

Much of the cost and effort required during the software testing process is invested in performing test maintenance—the addition, removal, or modification of test cases to keep the test suite in sync with the system-under-test or to otherwise improve its quality. Tool support could reduce the cost—and improve the quality—of test maintenance by automating aspects of the process or by providing guidance and support to developers.

In this study, we explore the capabilities and applications of large language models (LLMs)—complex machine learning models adapted to textual analysis—to support test maintenance. We conducted a case study at Ericsson AB where we explored the *triggers* that indicate the need for test maintenance, the *actions* that LLMs can take, and the *considerations* that must be made when deploying LLMs in an industrial setting. We also proposed and demonstrated implementations of two multi-agent architectures that can predict which test cases require maintenance following a change to the source code. Collectively, these contributions advance our theoretical and practical understanding of how LLMs can be deployed to benefit industrial test maintenance processes.

CCS Concepts: • **Software and its engineering → Maintaining software**; **Software evolution**; **Software verification and validation**; • **Computing methodologies → Machine learning**.

Additional Key Words and Phrases: Software Testing, Test Maintenance, Machine Learning, Large Language Models

ACM Reference Format:
Ludvig Lemner, Linnea Wahlgren, Gregory Gay, Nasser Mohammadiha, Jingxiong Liu, and Joakim Wennerberg. 2024. Exploring the Integration of Large Language Models in Industrial Test Maintenance Processes. In *Proceedings of Transactions on Software Engineering and Methodology (TOSEM)*. ACM, New York, NY, USA, 50 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software testing—the application of selected input to a system-under-test and inspection of the resulting behavior—is a crucial component of the development process. However, it is also a notoriously expensive component, said to represent up to half of the total development cost of the system [8].

*Both authors contributed equally to this research.

Authors' Contact Information: Ludvig Lemner; Linnea Wahlgren, Chalmers University of Technology and Ericsson AB, Gothenburg, Sweden; Gregory Gay, Chalmers and the University of Gothenburg, Gothenburg, Sweden, greg@greggay.com; Nasser Mohammadiha, Ericsson AB and Chalmers University of Technology, Gothenburg, Sweden; Jingxiong Liu, Ericsson AB, Gothenburg, Sweden; Joakim Wennerberg, Ericsson AB, Gothenburg, Sweden.

Manuscript submitted to ACM

Although there is an initial cost associated with creating test cases [17], much more cost is imposed by the need for *test maintenance* as the system-under-test evolves [103]. During test maintenance, new tests are created and obsolete tests are modified or removed to adapt to changes in the evolving source code [8]. Test maintenance can also be conducted to improve the quality or efficiency of the test suite—for example, to improve test coverage [103]. As a result, test maintenance requires long-term commitment of significant effort from software developers [103, 116].

Software tools have a role to play in reducing the cost and improving the quality or reliability of the test maintenance process [110, 116]. In particular, we focus on two forms of tool support. First, we examine cases where tools could completely or partially automate certain tasks conducted during the maintenance process, such as the creation or modification of test code. Second, we consider scenarios where tools can offer support, such as examples or targeted suggestions, to human developers. Despite its importance in practice [60], test maintenance is an under-explored and poorly understood aspect of the overall testing process [38, 49, 102, 109]. In relation to topics such as test case generation, there has been comparatively little research on automation of test maintenance [46].

Large language models (LLMs), machine learning models trained on massive corpora of text, are an emerging technology with great potential for language analysis and transformation tasks such as translation, summarizing, and decision support due to their ability to infer semantic meaning from text [129]. These capabilities extend to both natural language and software code [131]. As a result, LLMs have shown promise in automating or assisting with software engineering [127] and software testing [114] tasks, including test generation [97], documentation [42], refactoring [118], and automated program repair [104]. LLMs have shown potential in both forms of tools support mentioned above, including automation [107, 118] and serving as a conversational assistant to developers [93].

Although there has been significant interest in the general use of LLMs in software testing [114], the applications of LLMs in the test maintenance process have not been explored. Therefore, our purpose in this study is to examine the integration of LLMs and LLM agents—autonomous systems coupling an LLM with tool access and memory mechanisms [32, 111]—into this process.

We explore this topic through an industrial case study at Ericsson AB, a Swedish telecommunications company. We have conducted literature reviews, interviews, and a survey to explore (a) the *triggers* that indicate the potential need for test maintenance, (b) the potential *actions* that LLMs or LLM agents can take based on those triggers, and (c), the *considerations* that must be made when deploying LLMs in industry. To demonstrate the applicability of LLM agents in test maintenance, we also propose two architectures for multi-LLM frameworks that predict which test cases require maintenance following a change to the source code. We implemented four prototypes and evaluated their performance, limitations, and usefulness on an industrial codebase. Our study offers the following contributions:

- We have identified 37 low-level changes to the source code, as well as seven additional high-level development decisions, that can trigger the need for test maintenance.
- We offer guidance on how those triggers can be used to automate or assist with test maintenance tasks, as well as general guidance on the use cases and applications that LLMs are suited to in the test maintenance process.
- We explore considerations that must be made when deploying LLMs in industrial development environments, including both technological and organizational concerns.
- We propose two multi-LLM architectures to illustrate how LLM agents could be used as part of a test maintenance process. These architectures can serve as a starting point for future research.
- Our assessment of the performance and usefulness of the prototypes based on these architectures offers advice for the future implementation of LLM agents in test maintenance.

These contributions offer benefits to both researchers and practitioners, advancing our understanding of how new artificial intelligence techniques could be applied within test maintenance and offering practical guidance to software developers in industry.

The structure of this paper is as follows. In Section 2, we explain core background concepts. In Section 3, we explore related work. In Section 4, we explain our research questions and the methods used to address these questions. In Section 5, we provide the results of our research activities. In Section 6, we answer the research questions and discuss ethical aspects and threats to validity. Finally, in Section 7, we offer our conclusions.

2 Background

2.1 Software Testing

Software testing is an activity performed to ensure that software meets its stated requirements, including functional correctness [50] and non-functional requirements such as performance or usability [20]. During software testing, a *test suite*—consisting of one or more *test cases*—is executed against the system-under-test [50]. Each test case performs a series of input actions, records observations of how the software behaved following the application of input, then compares those observations to a set of expectations (known as a *test oracle* [10]).

Software testing is an important activity. However, it is also notoriously expensive [80], especially due to the significant manual effort involved in test management activities such as test case creation or manual execution of test cases [116]. Automation has a role to play in reducing the cost of testing by, e.g., transforming manual test cases into a form that can be automatically executed or suggesting what test input to apply [57, 110]. In particular, significant research efforts have been invested in automating the generation of test input [9], including recent investigations of the capabilities of LLMs [114, 125]. However, there are still significant concerns about the readability and maintainability of generated tests [40, 85]. In addition, any form of test automation requires significant up-front effort to implement and additional effort to maintain [8, 33].

Test maintenance is the process of updating the test suite as the source code evolves—including, e.g., adding new test cases to test new functionality, removing test cases that are no longer relevant, and adapting test cases to ensure their continued relevance to changed code [8]. The test suite may also evolve to improve its efficiency, e.g., by removing redundant tests, or to improve its thoroughness, e.g., by covering more diverse input or more paths through the codebase [103]. Test maintenance is understood to be an important part of quality assurance [102], and can account for a significant proportion of testing effort over the lifespan of a project [8, 103]. However, the area has received comparatively little research attention [102]. For example, "test smells" are known to negatively impact test maintainability, but are significantly less well-understood than "code smells" in the source code [109].

2.2 Generative Artificial Intelligence and Large-Language Models

Generative AI refers to machine learning models that produce content in the form of test, images, videos, or music in response to instructions delivered in the form of a *prompt* [21]. Prompts can include text and other forms of media, e.g., images or source code. Models are trained to infer the semantic meaning of a prompt, and use that meaning to produce an appropriate response.

Current Generative AI models generally are implemented using a transformer-based architecture, which is based on attention mechanisms designed to imitate the cognitive attention (i.e., the ability to focus on select stimuli) seen in humans [112]. Powerful models, known as *foundation models*, are trained on large and diverse data sets with the intention that they perform up to a minimal level on new tasks [43]. These models can then be adapted to new domains through fine-tuning using additional domain-specific training data [15, 134].

We focus in this research on *large language models (LLMs)*, a form of Generative AI trained on massive corpora of text. LLMs are suited for language analysis and transformation tasks such as translation, summarization, and decision support due to their ability to infer semantic meaning from textual input [84]. LLMs generate output by iteratively predicting the next token or word in the generated sequence [129]. Because of their ability to understand and output both natural language and code, LLMs are well-suited for software development tasks including test generation, automated program repair, and code review [39, 45, 93, 114, 128].

An LLM and a human may not draw the same meaning from a prompt, which has led to the development of different prompting strategies [132]. The process of developing a prompt or a series of prompts, known as *prompt engineering*, can be likened to a form of natural language programming intended to steer the generated output of the LLM [92]. Two common prompt engineering strategies include *few-shot prompting*—where input-output examples are supplied along with the prompt [18]—and *chain-of-thought prompting*—where examples are augmented with reasoning explaining how the correct output was reached [117].

Although LLMs have shown significant potential to automate software engineering tasks, many challenges and limitations have also emerged. LLMs are known to *hallucinate*, issuing output that appears initially plausible, but is incorrect and not justified by the underlying dataset [52]. Additionally, there is the risk of *data leakage*, where the LLMs may have seen a benchmark during training, leading to artificially high performance when it is asked to make predictions based on data from that benchmark [114].

An *LLM agent* is an advanced AI-based system coupling an LLM with additional tool access and memory capabilities [111]. These mechanisms allow LLM agents to reason, plan, and perceive or interact with an environment [32, 111]. For example, an LLM agent may have access to an organization's version control system or issue tracker. This would allow it to reason about the code, make changes to it, and push those changes to a remote repository. Multiple LLM agents can work together to address complex tasks. For example, agents can be given different roles or sub-tasks, e.g., simulating how human development teams would cooperate [119].

A common tool built into LLM agents is *Retrieval-Augmented Generation* (RAG) [63]. RAG allows an LLM to access information from external sources, and use that information to better respond to prompts. For example, even if a particular source code file was not part of the training data for an LLM, RAG could be used to retrieve that file and make inferences from its contents.

3 Related Work

3.1 Test Maintenance

Kochar et al. examined testers' perspectives on the characteristics of good test cases, finding that maintainability was one of the primary aspects mentioned [60]. However, in practice, Gonzalez et al. found that only a quarter of examined open-source projects implemented tests following patterns related to maintainability [38]. Imtiaz et al. also found that there is a need for studies on test maintenance in an industrial context [49].

Multiple authors have identified factors that influence the probability that test maintenance will be required, frequency that maintenance must be performed, or the difficulty of performing maintenance. For example, Pinto et al. found that many of the tests "added" to test suites are actually older tests that have been updated and renamed [87]. They also found that tests are generally deleted because they are obsolete, not because they are difficult to update. When tests are added, it is to cover new functionality, validate bug fixes, and validate refactored code. Alégroth et al. identified thirteen factors that affect the difficulty of test maintenance for GUI-based automated test suites [8], with test case length having the greatest impact. Berglund et al. also identified five factors that affect test maintenance for machine learning systems—e.g., non-determinism and explainability of the system-under-test—and nine further factors that affect both traditional and machine learning systems—e.g., the required precision of the test oracle and consistency of testing practices between teams [13].

Researchers have also examined the co-evolution of test and source code, under the assumption that changes to the source code and its corresponding test case within a short time-frame indicate that the changes are linked [106]. Kita et al. proposed a metric, Tconf, to assess the extent to which the source and test code have co-evolved [59]. Ens et al. have presented a visualization tool that shows how source and test code have changed over time [30]. Sohn and Papadakis developed a tool that infers traceability links between source and test code based on co-evolution [105].

Co-evolution has been used to identify source code changes that may trigger a need for test maintenance. We draw on this literature as part of answering our first research question (see Section 5.1). For example, Shimmi and Rahimi extracted patterns of co-evolution, classified as triggering addition, deletion, or modification of test cases [100]. Reich and Maalej extracted similar patterns, dividing code changes into low-level (changes to a single file, e.g., a change to a variable's data type) and high-level changes to the project (e.g., merging two packages) [90]. Levin and Yehudai also identified source code changes that tend to lead to test changes, e.g., removing a class or changing a method signature [62]. Marsavina et al. used association rule mining to identify patterns of co-evolution [74]. Their findings were later confirmed by Vidács and Pinzger [113].

Wang et al. also identified fine-grain source code change patterns—e.g., changes to `if-statements`—that may trigger test maintenance, and used these patterns to train a model to predict whether a test case is outdated [116]. Liu et al. improved the accuracy of this approach with more complex method of establishing traceability between source and test code and by automatically assigning code change patterns through static analysis, rather than manually labeling change patterns [69]. Huang et al. also proposed a machine learning approach that predicts whether test code needs to be updated based on semantic inference from the code and code complexity metrics [47].

Mirzaaghaei et al. have developed an approach to automatically repair test cases based on changes to source code [75, 76, 78]. Their approach is based on changes to method parameters and return values. Hu et al. have also proposed an approach to adapt test cases when the source code changes by inferring edit patterns from datasets of source and test code changes [46].

Our study is influenced by and expands on prior work. We expand the range of triggers considered, we broadly explore the theoretical applications of automation in test maintenance, and we demonstrate one proof-of-concept for automation—predicting which test cases need to be updated. Our proof-of-concept is similar to the approaches of Wang et al. [116] and Liu et al. [69]. However, we (a) demonstrate how LLMs could be used for this task, and (b), employ different forms of data in making predictions. Our study is also one of the few conducted in an industrial setting, and incorporates the experience and opinions of practitioners as an important source of data.

3.2 Large Language Models

Due to their ability to infer semantic meaning from both natural language and source code [86], large language models hold immense potential for automating software engineering tasks and activities [39, 45, 127]. This includes software testing, where Wang et al. have presented a survey on the use of LLMs in the field [114]. They found that there has been a heavy focus on test case generation—in particular, on unit test generation (e.g., [97, 101, 125]). LLMs overcome some issues with traditional test generation techniques, e.g., generating non-trivial assertions [97] and more readable test cases [34, 125]. They can also generate tests from natural language information, such as bug reports [55]. However, there are still challenges to address, including non-compilable test cases [125], hallucinated calls to non-existent code [125], and issues with conforming to strong type systems [101]. Gay also found that LLMs can improve the readability of existing test cases, which could improve their maintainability as well [34].

LLM agents are a very recent research topic, and there is limited work on their use in software engineering. Feldt et al. developed a taxonomy of LLM agents in the area of testing, and outlined an example conversational testing agent [32]. Yoon et al. have also demonstrated how LLM agents can perform Android GUI testing [123]. Hong et al. also proposed a framework based on standard operating procedures to facilitate cooperation between agents and humans [44]. As an example of how humans and LLM agents can interact, they model the roles in a software development team. Rasheed et al. have shown that cooperating LLM agents—each working on a particular subtask (e.g., code design, review, testing)—can generate higher quality code than a single LLM or LLM agent working alone [89].

To date, we are aware of no research examining the use of LLMs or LLM agents within test maintenance. We address this gap in our study by, first, broadly exploring how LLMs could be integrated into the maintenance process, then presenting a proof-of-concept—predicting which test cases may need to be updated—in an industrial setting.

4 Methods

In this study, we are interested in exploring the potential applications that LLMs or LLM agents could have—and the potential benefits or limitations of such applications—within the test maintenance process. Specifically, we address the following research questions:

RQ1 When changes occur in a project, what factors (e.g., changes to specific code elements) trigger the need for maintenance of the corresponding test code?

RQ2 What applications could current-generation LLMs or LLM agents have in the test maintenance process?

- **RQ2.1** Which of the triggers from RQ1 could be acted upon by an LLM or LLM agent?
- **RQ2.2** What viable test maintenance actions could an LLM or LLM agent perform, based on these triggers?
- **RQ2.3** What considerations must be taken when deploying an LLM within an industrial environment?

RQ3 What is the performance of our prototype multi-LLM frameworks when predicting the need for test maintenance?

The purpose of **RQ1** is to identify and categorize the specific factors (such as source code changes) that could trigger the need for test maintenance. The goal of **RQ2** is to explore how these triggers could be acted upon by LLMs or LLM agents, reducing the need for manual test maintenance. We explored three aspects of this application. First, we identified which triggers could be acted on. Second, we explored what actions an LLM or LLM agent could take, based on these triggers. We then investigated the considerations to be made when deploying LLMs in an industrial setting.

Finally, we implemented four prototypes, following two multi-LLM architectures, to demonstrate how LLM agents could be applied to predict which test cases need to be updated following a change to the source code. The purpose of **RQ3** is to evaluate the current performance of these prototypes, including strengths, limitations, and ideas for future research and development work.

We address these research questions through a case study at Ericsson AB, a Swedish telecommunications company. Our case study follows the guidelines from Runeson and Höst [95]. To address the research questions, we performed the following steps (shown visually in Figure 1):

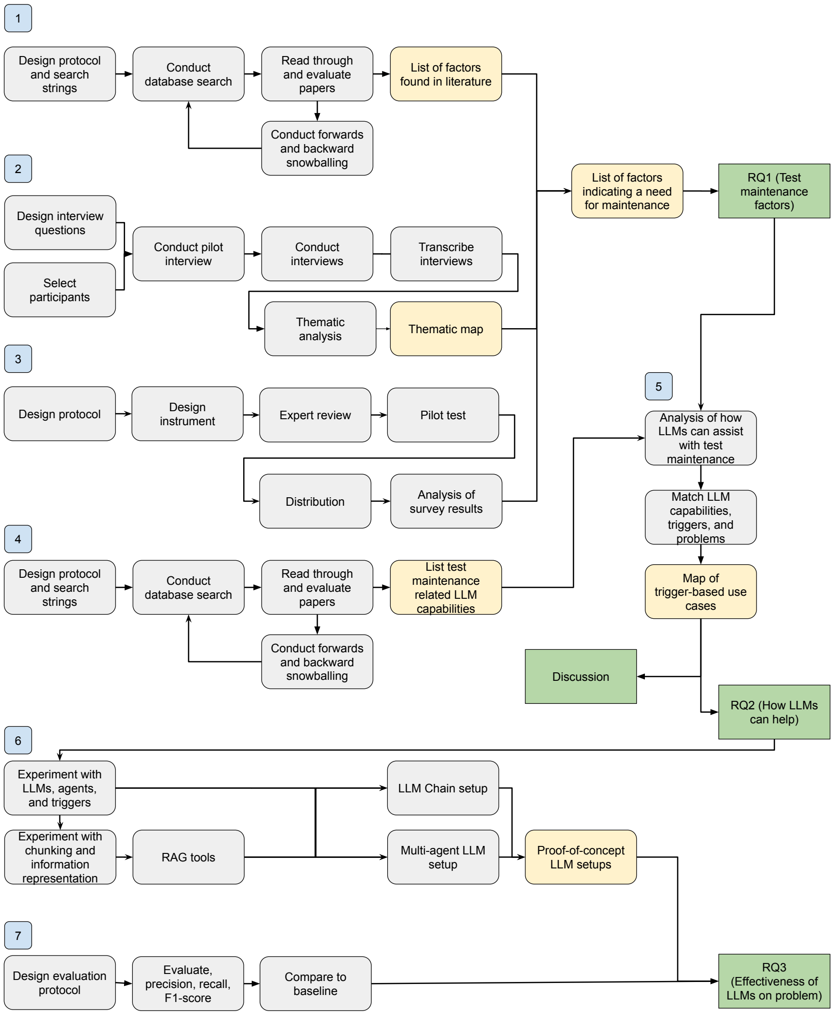

1
Design protocol and search strings
Conduct database search
Read through and evaluate papers
List of factors found in literature
Conduct forwards and backward snowballing
2
Design interview questions
Select participants
Conduct pilot interview
Conduct interviews
Transcribe interviews
Thematic analysis
Thematic map
List of factors indicating a need for maintenance
RQ1 (Test maintenance factors)
3
Design protocol
Design instrument
Expert review
Pilot test
Distribution
Analysis of survey results
5
Analysis of how LLMs can assist with test maintenance
Match LLM capabilities, triggers, and problems
Map of trigger-based use cases
4
Design protocol and search strings
Conduct database search
Read through and evaluate papers
List test maintenance related LLM capabilities
Conduct forwards and backward snowballing
Discussion
RQ2 (How LLMs can help)
6
Experiment with LLMs, agents, and triggers
LLM Chain setup
Experiment with chunking and information representation
RAG tools
Multi-agent LLM setup
Proof-of-concept LLM setups
7
Design evaluation protocol
Evaluate, precision, recall, F1-score
Compare to baseline
RQ3 (Effectiveness of LLMs on problem)

Fig. 1. Overview of the case study. The numbers correspond to the steps outlined in Section 4. Grey = activity, yellow = artifact, green = research question.

(1) We performed a **literature review to identify test maintenance triggers** (Section 4.1). Data extracted from the identified publications partially address **RQ1**.

(2) We conducted **interviews with testers** to understand the test maintenance process, triggers, and challenges at Ericsson (Section 4.2). We also discuss potential tool support. Thematic analysis of the transcripts contributed to addressing **RQ1 and RQ2**.

(3) To gather additional data from a broader set of participants, we conducted a **survey** at Ericsson regarding the test maintenance process, triggers, and challenges, as well as opinions regarding LLMs and tool support (Section 4.3). Analysis of survey results contributed to addressing **RQ1** and **RQ2**.

(4) We performed a **literature review on LLM and LLM agent capabilities** to understand how LLMs have been used for software testing tasks, as well as to understand their suitability and limitations in test maintenance (Section 4.4). The results of this review contributed to addressing **RQ2**.

(5) We **mapped the capabilities of LLMs and LLM agents with test maintenance triggers and actions** identified in previous steps (Section 4.5). This process contributed to addressing **RQ2**.

(6) We propose two **multi-LLM architectures**—and implemented four prototypes based on these architectures—to explore the viability of using LLMs to predict the need for test maintenance (Section 4.6). These prototypes are used to address **RQ3**.

(7) We **evaluated the performance of the prototypes**, based on their precision, recall, and F1-score (Section 4.7). This evaluation addresses **RQ3**.

4.1 Literature Review to Identify Test Maintenance Triggers (RQ1)

We conducted a literature review to identify aspects of the source code or development process of a project that—when updated—may trigger a need for test maintenance. In particular, we focused on identifying individual project changes (e.g., concrete changes to code, requirements or other artifacts) that previous research literature noted as potential triggers. Our review was inspired by Keele's guidelines for systematic literature reviews [58].

We applied the following search strings to ACM, IEEE, Science Direct, SCOPUS, and Google Scholar:

- ("test case" OR "test suite") AND (update OR create OR refactor OR generate OR maintenance OR evolution OR management OR repair OR co-evolution) AND ("source code" OR codebase) AND (factors OR criteria)
- ("test case" OR "test suite") AND (update OR create OR refactor OR generate OR maintenance OR evolution OR management OR repair OR co-evolution) AND ("foundational model" OR "machine learning" OR "LLM" OR "large language model")

The first string was developed iteratively through informal experimentation with synonyms, with the intent of capturing relevant research on test maintenance while filtering publications that do not discuss potential triggers. The second string adds an additional filter for publications related to LLMs, and was applied to ensure that no related work was missed. The strings were adapted for each database. In Scopus, the subject area was limited to computer science.

Database results were sorted by relevancy, then we selected publications by inspecting the title, followed by the abstract and conclusion. Inspection of results from each database ended after no new relevant papers were identified from three pages of results. We applied the following inclusion criteria:

- Publications must have been published within the past 15 years—i.e., from 2009-2024—to ensure relevancy to modern software development and testing paradigms.
- Publications must have been written in English.
- Peer-reviewed and pre-print articles were considered. However, preference was given to peer-reviewed articles.
- Publications must relate to test maintenance.
- Publications must discuss specific factors that could trigger a need for test maintenance.

A total of 48 publications were found through inspection to potentially meet these criteria. The publications were then read in full. If they discussed test maintenance triggers, these were recorded for later analysis. Eight publications discussed such triggers. We performed backward and forward snowballing on these publications, yielding 64 additional potentially-relevant publications. From these, four more discussed test maintenance triggers.

Table 1. Demographics of the interviewees. Experience refers to years of experience with testing, testing level refers to the level of granularity at which they regularly perform testing. Grouped participants were interviewed at the same time.

ID	Experience (Years)	Role	Testing Level
P1	15.00	Developer	Unit
P2	3.50	Data Scientist	Unit
P3	6.00	Developer	Unit
P4	5.00	Developer	Unit
P5	3.00	Developer	Unit
P6	2.00	Test Manager	Integration, System
P7	2.00	Test Manager	Integration, System
P8	25.00	Principal Developer	Overseeing Process

In total, 12 publications were found that discussed test maintenance triggers. We extracted these triggers. Then, we used a Miro board [4] to visually cluster them based on commonalities, such as belonging to particular levels of granularity within the source code. We also merged redundant triggers.

4.2 Interviews (RQ1–2)

Interviews were held with Ericsson employees to better understand their test maintenance process, triggers, and challenges. The interviews also included questions about the potential use of LLMs and tool support.

Participant Selection: Our target population was Ericsson employees with experience in testing at the company. We utilized convenience sampling, with elements of purposive sampling, to identify participants. Based on our knowledge of the organization, relevant teams were identified and invited to participate. Ultimately, we conducted five interview sessions, with a total of eight participants. The interviews were conducted in the same manner regardless of the number of participants present at the same time.

Table 1 presents the demographics of the participants. The most common role was "developer", followed by "test manager". The developers tended to work at the unit testing level, while test managers worked at the integration and system testing level. The participants ranged in testing experience from two to 25 years, with a median of 4.25 (average of 7.69) years of experience.

Interview Instrument: We conducted semi-structured interviews, where we followed a prepared interview instrument (Table 2), but asked follow-up questions to gain further insight. Interview sessions were conducted using Microsoft Teams and were recorded for transcription. All participants signed a consent form before beginning the interview. On average, each interview lasted for approximately 40 minutes.

We performed a pilot interview to test the instrument. This pilot session led to removal of an irrelevant question and minor clarifications of other questions. Because changes were minor, the pilot interview was used in the final analysis.

Interview Analysis: Interviews were first transcribed automatically using functionality provided by Microsoft Teams, then the transcripts were manually corrected. After the interviews were transcribed, we performed thematic analysis following the guidelines described by Braun and Clarke [16].

During the analysis process, we highlighted relevant parts of the transcripts and assigned "code labels"—short identifiers—to each. We then developed "codes" describing each highlighted segment. Then, in an iterative process, the codes and code labels were grouped into themes and sub-themes. This process was completed by the first two

Table 2. Interview instrument.

	Demographic Questions
1	What is your role?
2	How many years of experience do you have with testing?
3	At what level do you perform testing most often?
4	Which programming language do you primarily work with?
5	How much time do you spend working on source code versus test code?
6	What is the proportion between test code and source code in your project?
	Test Maintenance Questions
7	What reasons have you encountered for making changes to the test suite or a test case?
	Can you give examples of the smallest code change that led to the largest test changes, the most common types of changes, the most tedious changes, or other unique cases? What information do you use to make appropriate changes to the tests? What kind of modifications do you usually apply to the tests? What are factors do you consider when making changes? How do changes in source code reflect changes in test code?
8	What specific changes in the source code lead to updates in the test suite?
	Why does that source code change lead to a test suite change? Can you provide specific examples you have encountered? (E.g., adding new class or method or modifying an existing method, class, return statements, etc.)
9	How often do you make changes to the test suite?
	How often do you think the test suite needs to be changed? Is this different from the actual update frequency?
10	Can you describe a normal sequence of events from a factor instigating a change in the test suite to when the change is complete?
	What actions do you take as part of this process? How much effort do the steps to update the test suite take? Which step takes the most effort?
11	What consequences can there be from a change to the test suite?
	Do these consequences affect how you approach your work? How much additional maintenance effort can these consequences require?
12	What are the differences in the testing process between updating existing test cases versus creating new test cases?
	Are there different reasons or factors that cause each to occur? Are there generally differing amounts of effort required? Which of these is the most common and why?
	Automation and LLM Questions
13	What tool assistance would you like with updating the test suite?
	Which of these potential assistance use cases would be most valuable?
14	Do you currently use any tools to help update or create test cases?
	Are any of these tools automated? Do you think these tools, automated or not, work well? If you have used LLMs, which, and how did you find the experience? Would you like to use LLMs regularly as part of testing or test maintenance? LLMs can potentially be used for either creative purposes (e.g., providing advice) and for boilerplate purposes (e.g., generating code). Do you have a preference for how you would use them?

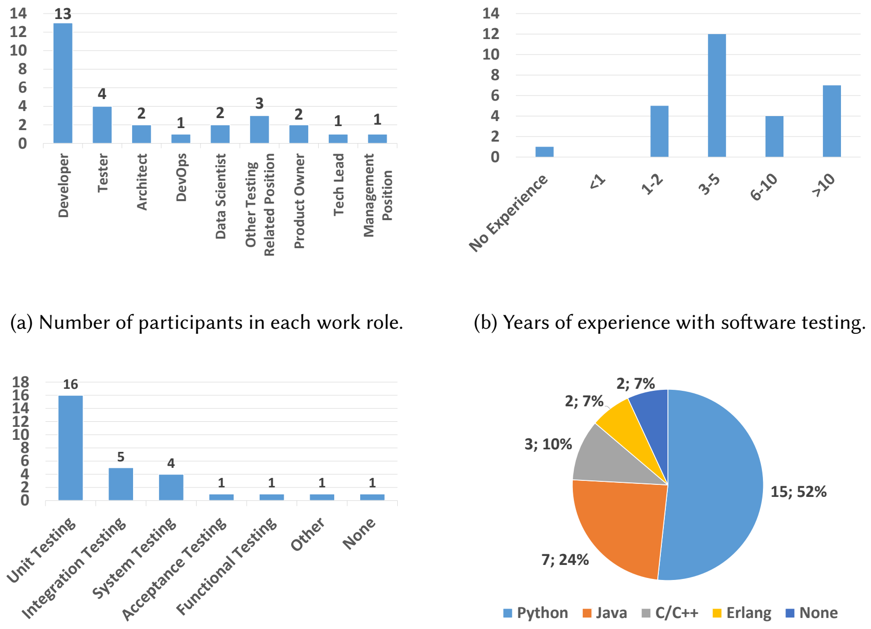
authors, with feedback from the other authors. We judged that we had reached saturation in codes after analyzing the transcripts from these eight participants. Therefore, we decided against conducting further interviews.

4.3 Survey (RQ1–2)

As interviews require significant effort to perform, we decided to conduct a survey with a broader set of Ericsson employees to gain further insight into the test maintenance process, triggers, and challenges, as well opinions regarding LLMs and tool support.

Participant Selection: Our target audience was, like the interviews, Ericsson employees with testing experience. We again employed convenience sampling with elements of purposive sampling. We initially distributed the survey to a "developer community" within Ericsson, consisting of over 100 developers across different countries and sections within the company that work within the same product area. Due to a low initial response rate, we then distributed the survey to additional developer communities. The survey was open for a total of 54 days.

Ultimately, there were 29 participants who completed the survey. The demographics of the respondents are shown in Figure 2. The most common role among survey participants was developer (45%), followed by testers and those in other testing-related positions (24%). Participants had an median of 3-5 years of testing experience, and worked most



(a) Number of participants in each work role.

(b) Years of experience with software testing.

(c) Most common testing level.

(d) Most common programming language.

Fig. 2. Demographics of survey respondents.

often at the unit testing level. The most common programming language in use was Python. These demographic details are, broadly, similar to those of the interview participants. Therefore, we hypothesize that the data from interviews and the survey responses are complementary.

Survey Instrument: The instrument for the survey was designed based on guidelines from Ghazi et al. [37] and Kasunic [56], and is presented in Table 3.

The survey was designed to take 5–10 minutes to ensure a reasonable completion rate. The intention of the survey was to provide quantitative data to augment the qualitative data gathered in interviews. Thus, questions were designed to multiple-choice, rather than open-ended. Respondents could provide multiple answers to some questions, but were asked to limit the number of selected options to force them to prioritize their selections. It should also be noted that the survey was designed prior to the analysis of interview data.

We evaluated the wording of all questions using understandability criteria established by Kasunic [56]. These criteria offer rules for the phrasing and structure of questions to minimize misunderstandings. The survey instrument was evaluated by a data analytics expert at Ericsson. It was also pilot-tested by three Ericsson developers to ensure there were no ambiguities in the questions, as well as to ensure it could be completed in less than ten minutes.

Survey Analysis: The results of the survey were analyzed using descriptive statistics, and were contextualized by the thematic analysis of the qualitative data obtained during the interviews.

Table 3. Survey instrument.

	Question	Response Options
	Demographic Questions	
1	What is your work role?	Developer, Tester, Architect, DevOps, Other
2	How many years of experience do you have with testing?	None, < 1, 1–2, 3–5, 6–10, > 10
3	At what level do you perform testing most often?	Unit, Integration, System, Acceptance, Other
4	Which programming language do you primarily work with?	C/C++, Erlang, Go, Java, Julia, Python, Other
	Test Maintenance Questions	
5	When making changes in the code, do you most often update source code or test code first?	Source, Test, Both Simultaneously, I Do Not Make Changes, Do Not Know
6	What reasons have you encountered for making changes in the test suite or a test case? [max 4]	Bug in Test Code, Bug in Source Code, Addition of New Feature, Changes in Requirements, Improve Test Coverage, Improve Test Performance, Improve Test Functionality, Changes in Tech Stack, Does Not Apply, Other
7	Which specific types of source code changes most commonly necessitate adjustments in the associated test code? [max 5]	New Method, Remove Method, Change Method Body, Change Method Return Type or Value, Change Method Signature, New Class, Remove Class, Change Class Declaration, Change Object or Variable Type, Add Parameter, Remove Parameter, Change Documentation, Change Loop Handling, Change Conditional Statements, Change Single Line of Code (e.g., assignment), Change Error or Exception Handling, Change Interface or Class Hierarchy, Change Interface, Change Parallelism/Concurrency, Other
8	During the test maintenance process which step is most effort intensive?	Identifying Solution, Designing Solution, Implementing Solution, Verifying Correctness of Solution, Other
	Automation and LLM Questions	
9	Have you used LLMs?	Yes–Often, Yes–Sometimes, Yes–Once or Twice, No–But Curious, No–Do Not Care, No–Do Not Want To, Do Not Know
10	How would you use LLMs during test maintenance? [max 3]	Would Not Use, Code Generation, Improving My Code, Providing Suggestions, Suggesting Best Practices, Tracking Task Progress, Documenting Code, Understanding Code, Do Not Know, Other

4.4 Literature Review on LLM Capabilities for Test Maintenance (RQ2)

To gain an understanding of how LLMs could potentially assist with test maintenance as well as considerations for industrial deployment, we conducted a literature review focusing on how LLMs have been applied within the field of software testing. This review followed a similar procedure to the review discussed in Section 4.1.

We applied the following search strings to ACM, IEEE, Science Direct, SCOPUS, and Google Scholar:

- (llm OR “large language model” OR “generative ai”) AND (“test management” OR “test maintenance” OR “software testing”)
- (llm OR “large language model” OR “generative AI”) AND (“test case” OR “test suite”) AND (“software”)
- (llm OR “large language model” OR “generative AI”) AND (“software engineering” OR code) AND (suitability OR appropriateness OR abilities OR methods)
- (llm OR “large language model” OR “generative AI”) AND (trigger OR “trigger point” OR action OR apply OR automate OR improve OR simplify OR update OR updating OR create OR creating OR develop OR enable OR interact OR understand OR analyze OR generate OR generation) AND (“test management” OR “test maintenance” OR “test suite” OR “test case” OR test OR “test code” OR “quality assurance” OR “software testing” OR “software documentation” OR “test scenario” OR “test design”) AND (suitability OR appropriateness OR abilities OR methods)

These strings were iteratively developed. The first two capture publications where LLMs are used in the area of software testing. The third expands more broadly to the capabilities of LLMs in performing tasks involving source code. Finally, the fourth string expands the set of applied synonyms to try to capture additional publications potentially missed by the previous strings.

The search strings were adapted for each database and, in Scopus, the subject area was limited to computer science. We applied the same inspection and stopping criteria previously discussed, with the following inclusion criteria:

- Publications must have been published between 2017—when the first significant developments related to LLMs emerged [112]—and 2024.
- Publications must have been written in English.
- Peer-reviewed and pre-print articles were considered. However, preference was given to peer-reviewed articles.
- Publications must discuss the use of LLMs to perform tasks based on source or test code. The tasks performed must have some relevance to test maintenance. Publications must discuss the suitability (e.g., strengths and/or limitations) of LLMs to perform such a task.
- Alternatively, a publication must discuss considerations for industrial deployment of an LLM.

After inspection, we identified 67 publications for further examination. From this set, 10 publications were particularly relevant for addressing RQ2. We performed forward and backward snowballing on these 10 publications, which added an additional 92 publications for examination. From those, two were particularly relevant. In total, we identified 12 publications with particular relevance for illustrating how LLMs could be used to automate actions related to test maintenance and considerations for industrial deployment.

4.5 Mapping Test Maintenance Triggers and Tasks with LLM Capabilities (RQ2)

To identify how LLMs or LLM agents could assist with test maintenance tasks, we compared the data gathered on triggers, practices, and challenges from the literature review on test maintenance triggers (Section 5.1), the interviews (Section 5.2), and the survey (Section 5.3). We then mapped this data to the capabilities of LLMs demonstrated in previous literature (Section 5.4).

We developed a list of potential matches between test maintenance tasks or challenges and ways that LLMs have been applied through an open-ended brainstorming session. To decide which specific test maintenance tasks a LLM or LLM agent could be implemented for, we utilized a mind-mapping process. This was done by brainstorming which LLM actions and applications might fit specific triggers, and how these could be combined to solve the test maintenance tasks and challenges identified through the interviews and survey. We also considered how the use of LLMs should be adapted to match developers' preferences and the teams' way of working.

As part of addressing RQ1, we split test maintenance triggers into high- and low-level triggers, where high-level triggers relate to process-level decisions (e.g., requirement changes or a decision to improve the efficiency of the test suite) and low-level triggers referring to individual changes to project artifacts. We created one map for high-level and one for low-level triggers.

During this mapping process, we also decided that LLM agents are more appropriate for addressing test maintenance tasks than using LLMs on their own. This is because an LLM agent can utilize external tools, like retrieval-augmented generation [63], to obtain relevant additional information without retraining the core model. Additionally, an LLM agent generally has memory capabilities and can, therefore, perform complex multi-step processes as well as potentially correct previous mistakes. LLM agents typically demonstrate improved performance over non-agent LLMs, especially in zero-shot prompting interactions [115, 120].

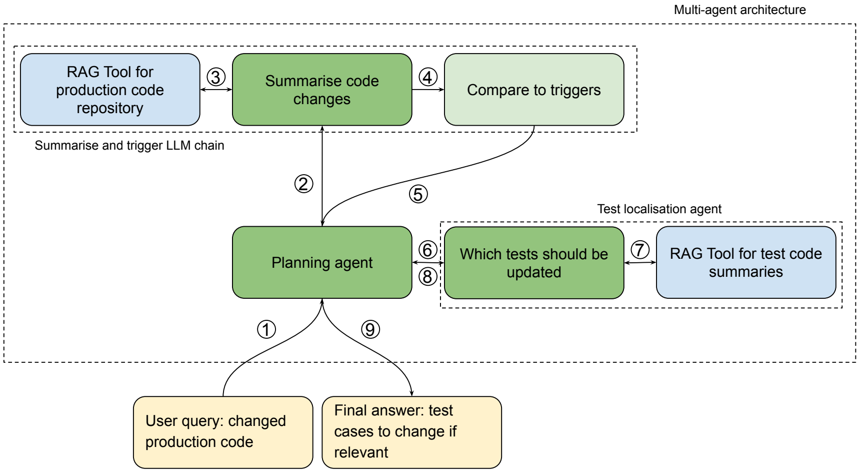

Multi-agent architecture
RAG Tool for production code repository
③
Summarise code changes
④
Compare to triggers
Summarise and trigger LLM chain
②
⑤
Test localisation agent
Planning agent
⑥
⑧
Which tests should be updated
⑦
RAG Tool for test code summaries
①
⑨
User query: changed production code
Final answer: test cases to change if relevant

Fig. 3. Architecture incorporating a planning agent and test case summaries. Yellow represents input and output, light green for LLM instances, dark green for LLM agents, and blue for tools used by the agents.

An alternative to an LLM agent would be to pre-train an LLM for a specific task. However, as the costs of pre-training can be very expensive, we decided that an LLM agent would be more appropriate. In addition, pre-training is not as adaptive as an agent architecture as the model would still need to be retrained to gain access to updated information. However, these two concepts are complementary—a pre-trained model could be used within an agent architecture—and their combination could be explored in future work.

4.6 Proof-of-Concept Design (RQ3)

The results of RQ1–2 illustrate the theoretical applicability of LLM agents within the test maintenance process. To explore the *practical* applicability of current-generation LLM agents, we decided to implement a proof-of-concept that could act on the source and test code developed at Ericsson. To establish a scope for this proof-of-concept, we decided to focus on a use case where a multi-LLM framework would observe a change to the source code, then—based on the specific changes made—would predict which test cases require maintenance.

We iteratively explored multiple potential architectures for a framework that could predict the need for test maintenance. Ultimately, we implemented four prototypes in Python, based on two proposed architectures. This section will present the architectures and implementation details for the four prototypes.

Overall Architectures: We developed two core architectures for performing the prediction tasks. Both employ a multi-LLM architecture to improve performance [41, 44, 66]. This allows each LLM agent or LLM instance to focus on a particular subtask, and means that each agent can be developed to employ specialized tools. The first architecture (Figure 3) employs a planning agent that coordinates the effort of other LLM instances and agents. The second (Figure 4) is architected as a chain of calls between LLMs instances and an agent.

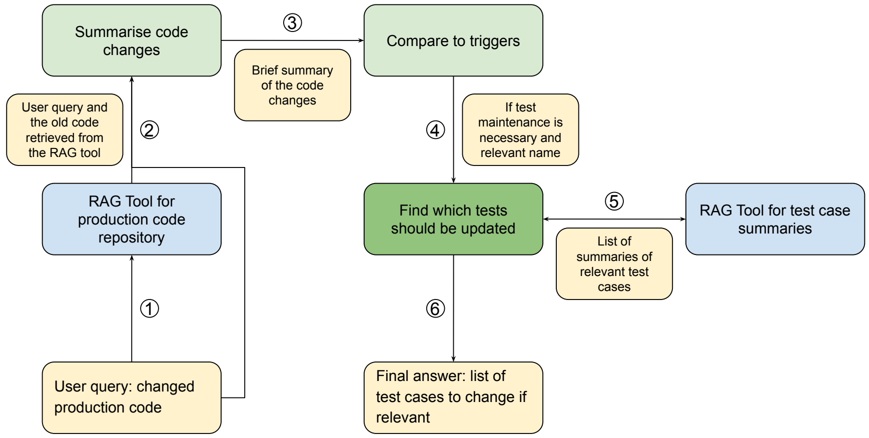

Summarise code changes
③
Compare to triggers
Brief summary of the code changes
User query and the old code retrieved from the RAG tool
②
④
If test maintenance is necessary and relevant name
RAG Tool for production code repository
Find which tests should be updated
⑤
RAG Tool for test case summaries
List of summaries of relevant test cases
①
⑥
User query: changed production code
Final answer: list of test cases to change if relevant

Fig. 4. Pipeline-based architecture employing test case summaries. Yellow represents input and output, light green for LLM instances, dark green for LLM agents, and blue for tools used by the agents.

We have also implemented two versions of each architecture, resulting in four concrete implementations. The first examines the test code directly to determine whether a test case requires maintenance. The second employs an LLM to generate a natural language summary of the test case, then acts on that summary instead of the test code.

The architecture employing a planning agent (Figure 3) makes predictions following this sequence of steps:

(1) The user queries the framework with the changed source code.
(2) The planning agent contacts the summarize and trigger LLM chain to identify whether test maintenance is necessary based on the code change.
(3) The summarizer retrieves the current source code and compares it to the changed code submitted by the user.
(4) The summarizer sends a summary of the code change to compare the change to the test maintenance triggers.
(5) The summarize and trigger LLM chain responds to the planning agent about whether test maintenance is necessary by comparing the summary of the code changes to the triggers found when addressing RQ1.
(6) If test maintenance is necessary, the planning agent contacts the test localization agent to identify which tests need to be changed.
(7) The test localization agent retrieves the test code summaries, to identify which test cases need to be changed.
(8) The test localization agent responds to the planning agent on which test cases need to be changed.
(9) The planning agent responds to the user and either says that test maintenance is unnecessary or provides information on which test cases need to be changed based on the original code change.

The pipeline-based architecture (Figure 4) makes predictions through the following steps:

(1) The user queries the chain with the changed source code.
(2) The RAG tool retrieves the current source code and sends it to the code summarizer. The code summarizer also receives the user query—the new code—for comparison.
(3) The summarizer sends a summary of the code change to the test maintenance trigger comparator.

(4) The trigger agent determines whether test maintenance is necessary and, if so, also includes the name of the relevant code method.
(5) If test maintenance is necessary, the test localizer retrieves test case summaries.
(6) The test localizer outputs the list of relevant test cases to the user or states that test maintenance is unnecessary.

LLM Agent and Instance Implementation: The agents are implemented using the LangChain framework [22]. This framework was chosen because it offered implementations of functionality including agent wrapper methods, vector stores, and embedding models. This allowed us to focus on implementing our particular use case rather than reinventing core aspects common to other LLM agents.

The LLM agents and instances employ the Mistral 7B - Instruct model as their core LLM [53]. Mistral 7B - Instruct is a version of the Mistral 7B model that has been fine-tuned to be deployed as a conversational agent [53]. Ericsson must approve LLMs for internal use. At the time of implementation, we determined that Mistral 7B - Instruct was the best model for our use case, considering availability, capabilities, and the Ericsson approval process and guidelines.

The structures of these individual modules in the architecture are implemented through the LangChain framework, following the ReAct agent structure [122]. Each agent is implemented with its own individual memory, planning mechanisms, and tools. Agents record observations into an "agent scratchpad", which serves as a form of memory, and is also printed to `stdout` to provide additional context to explain an agent's decisions. Each agent has an individual prompt that gives them instructions on their role, as well as how to act. The prompts provided to each LLM agent or instance are provided in our supplementary data package[1].

Each non-agent LLM instance is provided with prompts through a chain of calls. As the instances are not full agents, they do not have access to memory structures that let them follow a train of thought over several queries. However, they offer a faster response as they do not have to interface through the agent framework. Tools are utilized through separate calls that then send results as input to the LLM instances.

The implementation of the planning agent is such that the other agents and instances that it communicates with are available to it as tools that it can utilize. The agent prompt specifies that it needs to use the tools for solving its tasks. The prompt further expands on its tasks, states that it can work with only receiving changed code as input, and offers additional instructions. After the agent has received the query, it can choose how to approach the problem, which tools to use, and when it considers the problem solved. These decisions are made by the model. The planning agent does not make all decisions, but does control the overall process. If it determines that the output from the other agents is insufficient, it can invoke them again.

The implementation of the "summarize and trigger" LLM chain consists of a cooperating LLM agent and instance, as can be seen in Figure 3. The summarizer is implemented as an agent with tool access, while a simple LLM instance was used for comparing the summary of the code changes to the triggers from RQ1. The LLM instance is given a list of triggers in its prompt and compares the triggers to the summary of the code change it was given by the summarizer agent. This functionality was implemented as an instance rather than an agent because its task is relatively simple and requires no additional tools. The two modules are connected in a chain instead of having one be a tool of the other.

For the architecture without the planning agent, the code summarizer agent is instead implemented as an LLM instance. The RAG tool for the source code is called separately first from the user query and then the results from the RAG tool, along with the user query, are sent to the LLM to be inserted in the prompt instead of utilizing a tool

[1] Available at https://doi.org/10.5281/zenodo.13353593

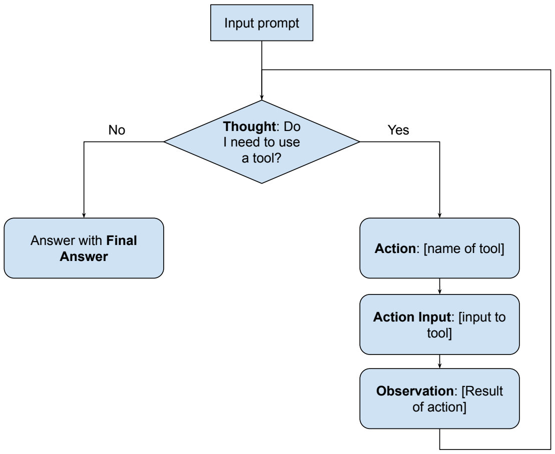

Input prompt
No
Yes
Thought: Do I need to use a tool?
Answer with Final Answer
Action: [name of tool]
Action Input: [input to tool]
Observation: [Result of action]

Fig. 5. Flow diagram showing how a LangChain ReAct agent invokes tools to perform tasks.

structure. The agents and LLM instances are also connected in a chain where the outputs of the LLMs and agents serve as inputs for the following LLM or agent.

The test localization agent has its prompt and access to a RAG tool that consists of an embedded vector store of the latest version of the relevant code repository, this includes both source code and test code. In the chain architecture, the test localisation agent works directly with output from the trigger comparator. It chooses on its own whether to search for test cases or not depending on the output of the trigger comparator.

ReAct Agent Structure: The LLM agents are implemented following the ReAct agent structure [122]. After receiving a prompt, the model goes through a cycle of Thought-Action-Observation, where an action is taken based on a thought, and an observation is made from the results of the action. If the observation provides a satisfactory answer, the agent stops there, and if not, it continues to go through this cycle until an answer is reached.

LangChain offers multiple agent structures, including ReAct. The framework provides functions for customizing an agent with tools and specialized prompts, and also provides a scratchpad where the agent can record its thoughts and decisions for use in subsequent decisions. Figure 5 shows how this structure is used to perform tool invocation.

In our implementation, we impose a limit of five iterations of the cycle in Figure 5. If the agent fails to answer using the correct format, it will be reminded what the correct format looks like and asked to answer again. This, too, counts as an iteration. We also impose a limit of 300 seconds per prompt to an LLM.

Tool Use: Our prototypes utilize two RAG tools to access the code repository. The first RAG tool is utilized by the code summarizer to examine the current and previous versions of the source code. This gives the summarizer the ability to accurately describe the code change. The second RAG tool is utilized by the test localization agent to determine which test cases are relevant to the changed code.

Both RAG tools use the Nomic embedding model to encode code for processing by an LLM [83]. This model was used because it was the only embedding model in current deployment at Ericsson that we had access to. We also tested other non-embedding models, but these offered worse performance than Nomic. We employed the Facebook AI Similarity Search (FAISS) library [29] as a vector store used for storing the embedded vectors.

4.7 Proof-of-Concept Evaluation (RQ3)

We performed an evaluation of the four prototypes to understand their current capabilities and limitations and to provide inspiration for future research and development work.

It is worth clarifying our expectations regarding the performance of the prototypes prior to the evaluation. Mistral 7B - Instruct is not a state-of-the-art model[2]. The model has also neither been trained, nor fine-tuned, on the Ericsson codebase. Therefore, we did not expect the prototypes to attain sufficient performance to be directly usable in a real-world application. Rather, our intention was to show the feasibility of employing LLMs for such a use case, to provide a model for future implementations, and to explore the current limitations that must be overcome.

Evaluation Dataset: To evaluate the accuracy of the four prototypes, we developed a dataset from the commit history of a Java-based `git` repository at Ericsson. The project contained in the repository is a microservice that performs data processing tasks. The test code contains both unit and integration tests.

We compare the predictions of each prototype to a ground truth obtained from the commit history. Our ground truth is based on the assumption that source code and test cases that are changed in the same commit are examples of co-evolution. The dataset spans 57 commits, which together contain 374 distinct code changes.

The prototypes are designed to return test case recommendations for a code change. Therefore, each commit has been manually split into individual code changes. We define an individual code change as a chunk of source code (containing the changed code) of 20 lines of code or the entire modified method, whichever is smaller. The reason for using individual code changes instead of entire commits at once was due to limitations in Mistral 7B - Instruct's ability to understand and work with large contexts. The following types of code changes were excluded from the dataset:

- Commits with no changes to the source code.
- Changes to source code that only modified imports or comments.
- Additions or removal from source code. This evaluation focused solely on modifications.
- All changes to test code that were not within a test case, such as changes to test support code.
- The addition and deletion of files and new test cases in the test code. Deletion of existing test cases was included as it can be seen as a modification of an existing test case.

Evaluation Procedure: For each commit, we performed the following procedure:

(1) The information stored in the RAG tools was updated to reflect the current commit.
(2) Each individual code change was separately fed into the agent setup, which would output test cases that needed to be updated or modified based on that change. These test cases suggested by the agent were saved as results.
(3) In the dataset, ground truth is known for a commit, not for each code change. To assess accuracy, the suggested test cases from all code changes in that commit were combined into a set, which was compared to the test cases that had actually been changed in the commit (the ground truth).

We define a true positive (TP) as a correctly identified test case, a false positive (FP) as a test case that was identified for modification that should not have been identified, and a false negative (FN) as a test case that was not identified for modification that should have been identified. These measurements are calculated individually for each commit[3]. Then, to assess performance across the full dataset, we sum together the TP, FP, and FN from each commit.

[2] For example, it was recently ranked in 86th out of 135 models on the HumanEval code generation benchmark [3].
[3] The per-commit results are available in our supplementary data package: https://doi.org/10.5281/zenodo.13353593

From the aggregated TP, FP, and FN, we calculate the precision as $\frac{TP}{(TP+FP)}$, i.e., how many of the identified test cases were correct across the dataset. Recall can be defined as $\frac{TP}{(TP+FN)}$, that is, how many of the correct test cases were identified across the dataset. We use the recall and precision to calculate the F1 score, the harmonic mean between the precision and the recall, as a measure of accuracy. The F1 score is measured as $2 * \frac{(precision*recall)}{(precision+recall)}$.

We omit commits from the performance calculation where any of the prototypes reached the iteration limit (five iterations) and failed to issue output. The planning agent with summaries reached this limit on 12 commits, the planning agent without summaries on 11 commits, the pipeline with summaries on one commit, and the pipeline without summaries on two commits. We calculate performance on the remaining subset of 42 commits.

Due to limitations in access to the industrial data and development environment, we only performed one evaluation per prototype for each commit. We employed a low temperature setting, 0.2, to constrain randomness. Therefore, results should not vary considerably if the experiment was repeated. However, these results should be taken as a demonstration of the feasibility of performing this task rather than a fully generalizable indication of performance.

5 Results

5.1 Literature Review to Identify Test Maintenance Triggers (RQ1)

We identified 12 publications that explicitly define concrete project changes that can trigger a need for test maintenance. Broadly, these triggers can be divided into changes to the *documentation*—information about the software, such as comments, requirements, or annotations in the code—and changes to the functionality of the *source code*.

In Figure 6, we list the triggers that we discovered. We also group changes to the source code based on the level of granularity—functionality, class-level, method-level, and line-level. We define these triggers in Tables 4–7. There is some overlap in the triggers—for example, "addition of functionality" could include "addition of a class" or "addition of a method". However, we retain all three to preserve as much information as possible about changes to the code and to capture cases where it was not clear how the new functionality was added.

Table 4. Test maintenance triggers that affect the general functionality of the system.

Type of Change	Description	Sources
Changes to Functionality	**Changes made in source code that affect system behavior.**	
Add Functionality	Implementing new functionality that did not exist before. This is a general trigger for cases where it is not clear how the functionality was implemented.	[62, 75, 100]
Add Interface	Implementing a new interface that did not exist before.	[62, 75, 77, 78]
Error and Exception Handling	Changes made to code that handles errors and exceptions, such as `throw`, `try`, and `catch` keywords.	[69]
Parallelism and Concurrency	Changes to how the system handles parallel threads and concurrency, such as the `synchronised` keyword.	[69]
Remove Functionality	Code that implements functionality is removed. This is a general trigger for cases where it is not clear how the functionality was removed.	[62, 100]
Update API	Changes to how an external API is invoked.	[46]

Table 5. Test maintenance triggers that describe changes made to a class.

Type of Change	Description	Sources
Changes to a Class	**This grouping covers changes made at the class-level.**	
Add Class	Implement a new class that previously did not exist.	[62, 69, 74, 77, 90, 113]
Class Declaration	Changes made to a class declaration, including extending class hierarchy.	[62, 74, 75, 78, 113, 116]
Constructor	Changes made to the class constructor, for example, creating a constructor or adding a constructor parameter.	[90]
Fields	Changes made to the fields of a class.	[74, 113, 116]
Remove Class	Remove a class that previously existed.	[62, 74, 113]

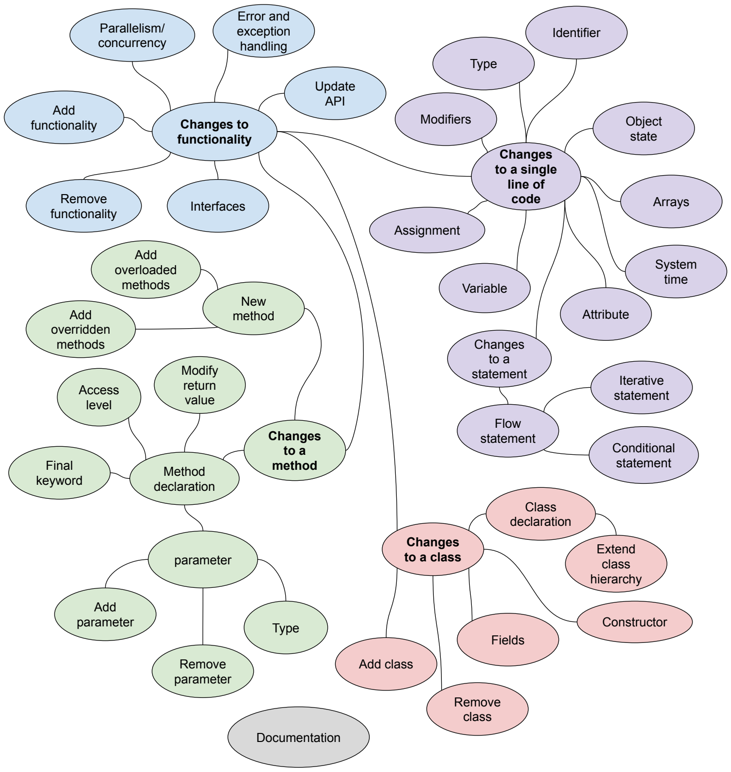


Fig. 6. Test maintenance triggers identified in the literature review. The triggers are grouped into clusters, indicated by color.

Table 6. Test maintenance triggers that describe changes made to a method.

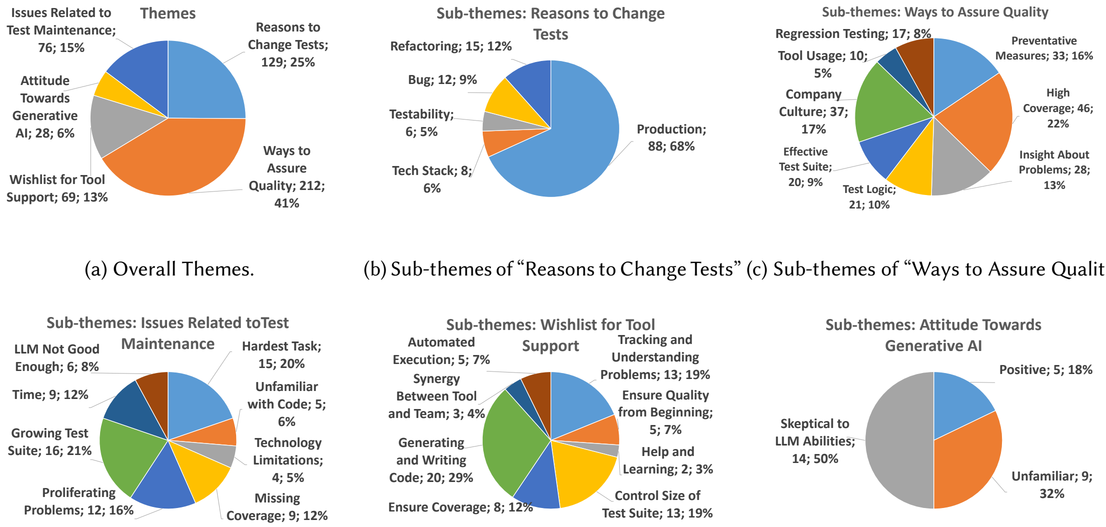
Type of Change	Description	Sources
Changes to a Method	**This grouping covers changes made at the method-level.**	
Add Method	The implementation of a new method. This includes sub-triggers adding an overloaded method and adding an overridden method.	[75, 77, 78, 90, 113]
Method Declaration	Changes made to a method declaration or signature. This includes the sub-triggers changing access level (both widen and decrease), adding or removing `final` keyword, changing value or type of the return value, and adding, removing and changing the type of a parameter.	[62, 62, 69, 69, 74–76, 76–78, 78, 90, 113, 116]

Table 7. Test maintenance triggers that describe changes made to a single line of code.

Type of Change	Description	Sources
Changes to a Single LOC	**Changes made to the source code that affect a single line.**	
Arrays	Changes to arrays, e.g., adding an array and changing the access level.	[69]
Assignments	Changes to the value assigned to a variable, as well as compound assignments.	[69]
Attribute	Declaration of attributes as well as extraction for assertion.	[74, 90, 113]
Modifiers	Changing and updating modifiers, e.g., the `public` keyword.	[46, 69]
Identifier	Changes to variable names as well as other identifiers such as package names.	[46, 69, 116]
Object State	Removing or adding object state.	[62]
Override System Time	Overriding the system time.	[90]
Statement	Changes to control-flow, conditional statements, or loop statements	[62, 69, 74, 116]
Type	Modify the type of an object, either the keyword or by casting.	[69]
Variables	Changes to the declaration of a variable.	[69]

Table 8. Overview of interview themes.

Theme	Description
Reasons to Change Tests	Reasons for performing test maintenance.
Ways to Assure Quality	Ways to ensure that the test cases and test suite hold high quality.
Issues Related to Test Maintenance	Current issues experienced with test maintenance.
Wishlist for Tool Support	Type of automated help wanted from tools by interviewees to help with test maintenance.
Attitudes Towards Generative AI	Attitudes expressed about using LLMs for test maintenance.



(a) Overall Themes.

(b) Sub-themes of “Reasons to Change Tests”

(c) Sub-themes of “Ways to Assure Quality”

(d) Sub-themes of “Issues Related to Test Maintenance”

(e) Sub-themes of “Wishlist for Tool Support”

(f) Sub-themes of “Attitudes Towards Generative AI”

Fig. 7. Number and percentage of codes associated with each theme and sub-theme inferred from interview transcripts.

5.2 Thematic Analysis of Interviews (RQ1–2)

In total, five major themes were identified during the analysis of the interview transcripts—*Ways to Assure Quality*, *Reasons to Change Tests*, *Issues Related to Test Maintenance*, *Wishlist for Tool Support*, and *Attitudes Towards Generative AI*. Each has multiple sub-themes.

An overview of the themes is presented in Table 8, and we discuss each theme and sub-theme below. In addition, in Figure 7, we indicate the number of codes corresponding to each theme and sub-theme. The number of occurrences of a theme or sub-theme does not necessarily indicate its relative importance, but it does indicate that the interview participants had more to say about some subjects.

5.2.1 Reasons to Change Tests. This theme concerns test maintenance triggers. The primary triggers are summarized as sub-themes in Table 9.

Production: The most prominent reason for test maintenance—discussed in 68% of codes in this theme—is a change in the source code. Naturally, if functionality is changed or new functionality is introduced, the test suite must generally

Table 9. Overview of sub-themes of “Reasons to Change Tests”.

Sub-theme	Description
Production	Changes on the source side cause changes in tests.
Refactoring	Refactoring should not change functionality of source code, but can still induce changes in the test suite.
Bug	The discovery of a bug leads to changes in tests.
Tech Stack	Changes in the tech stack necessitate changes to tests.
Testability	Test and source code are changed to be easier to test.

evolve as well. Participants, in particular, described test maintenance occurring following new feature implementations, changes in requirements, modifications to existing code methods, and changes to conditional statements—all of which were also derived as triggers in Section 5.1.

> “Production is driving us to change testing. I mean both finding bugs and adding features. Those two really drive up the testing.” - P2

Refactoring: The source code often evolves as a result of refactoring:

> “You know your first try at something is very rarely your best attempt, so generally the same piece of code might be rewritten three or four times, perhaps with different technology or with a different architectural viewpoint, or just in terms of code complexity.” - P1

Participants noted that it is difficult to completely separate logic and code. Though the “functionality” remains unchanged, how that functionality is tested may still require modifications. Refactoring at a small scale—e.g., changing the implementation of a function to improve its performance—should ideally not require test maintenance. However, large-scale refactoring, such as a change to the fundamental architecture of the project, may require such maintenance. For example, methods and classes invoked by tests may no longer exist, or interface changes may have been implemented.

Bug: Naturally, if a bug is discovered, the source code will evolve. This evolution may induce evolution in the test suite as well. For example, developers may discover that an area of the code is under-tested:

> “It could be that we noticed something in production not working the way it should. So we locate, OK, where is this code? And then we might connect to, oh, isn’t this tested? This logic should work like this, but it works like this. And then we notice that, OK, we don’t cover this in tests.” - P2

Participants also discussed cases where tests were added to the suite to reproduce a bug, as well as cases where bugs were discovered in the test themselves.

Tech Stack: Changes to the tech stack—e.g., programming languages, third-party libraries, testing frameworks, or build systems—can also trigger the need for test maintenance. An unstable tech stack can intensify test maintenance effort, and may constrain the forms and quantity of testing that can take place.

> “And then you have a limited tech stack that you can actually use. ... and these things also change over time. Components that you might have been able to use yesterday, you might not be able to use anymore tomorrow because the licensing requirements change, for example. So a lot of that also impacts the changes that we need to make in our code base in order to keep things running.” - P1

Testability: At times, the source code is modified to improve its testability, with the aim of improving the efficiency and effectiveness of testing. These changes also can trigger the need for test maintenance, as the test suite must be adapted to the changing system and new forms of testing may now be possible. For example, a participant noted efforts to make components more modular or to make them mockable in tests:

> “... there are some cases where code has been written that’s not really testable. Let’s say we have a structure where we can’t mock stuff ... And then I would say sometimes the testing is driving changes in production. So we try to make [a component] more modular.” - P2

Table 10. Overview of sub-themes of “Ways to Assure Quality”.

Sub-theme	Description
Preventative Measures	Finding faults and stopping them from occurring.
High Coverage	Execute all code to ensure thorough testing.
Company Culture	Ensure testing activities are valued and prioritized.
Insight About Change	Ensure a proper understanding of a change before trying to implement it.
Test Logic	Tests should not only test the code itself, but the logic it represents.
Effective Test Suite	Tests should be efficient, up-to-date, and relevant, to keep the test suite from growing needlessly.
Regression Testing	Ensure changes do not break unchanged code.
Tool Usage	Tools provide help in discovering faults and increasing code quality.

5.2.2 Ways to Assure Quality. This theme encapsulates the different ways the interviewees discussed to ensure the quality of test cases and test suites, both during their initial creation and during maintenance. An overview of the sub-themes is shown in Table 10. Several of these sub-themes relate to how the testing process is conducted to ensure faults are detected before deployment. The sub-themes also describe the importance of having a high quality test suite to ensure that code and test maintenance and evolution are as painless as possible.

Preventative Measures: Participants strongly stressed the importance of ensuring both code and test quality as a means of discovering and fixing faults before the project is deployed to customers. Participants states that it is better to have a more rigorous and time-consuming testing process than to fix faults after the code has been deployed.

> “It’s less work to prevent an error than it is to go and fix the error.” - P1

One method of ensuring quality is test-driven development, where tests are written before the source code:

> “I start with the test and then build the code based on that and, therefore, it does definitely change the approach that I take to testing in general. Because the test code isn’t written after the feature is written. The test code is part of the feature, every time I put down two lines of code, two lines of test code goes with it. You build it piece by piece by piece by piece by testing and testing and testing continuously.” - P1

Other examples of preventative measures pointed out by participants include targeting high test coverage, regression testing, code reviews, and collaborating with other teams that make use of the same source code.

Participants also stressed the importance of making careful test maintenance decisions. One discussed a case where a test was removed from the suite, which led to a fault being missed:

> “After that, we have become more cautious and careful when we make the decision to remove test cases ... because we are only part of the CI flow and there are other CI flows. Also, each area is testing different scenarios and, so, when we are thinking of removing something, we should always check with the other parts of the CI flow if they have coverage. Then it’s probably a lower risk for us to remove it. ” - P7

High Coverage: One of the major preventative measure taken by the participants was to ensure that testing is thorough. Two discussed ways of measuring “thoroughness” were high structural coverage of the code and high input and output diversity in the test cases.

> “I want to know that every single path through the code is covered. Otherwise, I get nervous.” - P1

One participant noted that, if coverage measurements are used, it becomes more obvious when the test suite has not co-evolved with the source code:

> “Every line of code should be tested. So when we have just changed a small part of the code then we need to update the test code because otherwise, it complains that not all the lines are covered in test.” - P3

Company Culture: It is important that rigorous testing is valued within the company culture. A team that takes pride in delivering high-quality code will need to edit that code less often. Further, such teams will acknowledge that there is always room to improve the testing process and will be open to change and feedback. One interviewee noted:

> "I think we are pretty good at testing now. We have had problems with that before ... but I think we're at the point that we actually prioritise it." - P2

Insight About Change: Participants discussed the importance of properly understanding why a change must occur within the source or test code before starting to plan or implement a change. This same mindset affects any change—whether fixing a bug or performing a planned change. Taking time to gain an understanding reduces the total time to perform maintenance or evolution. As one interviewee put it when describing how to update a test case:

> "So first you have to identify: what am I changing and why? Then maybe doing it is not that hard." - P2

Taking the time to understand the fault and explicitly adding a report to the issue tracking system also allows the team to match the issue with the right developer, as different developers have different strengths and competencies.

> "The next step is always to have some kind of ticket, right? Because you, the person that picks it up, is not necessarily the best-suited person to actually complete the task. Make the fix so everything goes through a ticket in the backlog ... And then the next step would be for somebody in the team to pick it up. They will then do a little bit of more of a deep dive. Try and go and look at the logs. Try and see why the behaviour is what it is." - P1

Test Logic: When performing testing, it is important to not design tests around the implemented code, but to focus on the logic that is supposed to be represented in that code. Participants emphasized that code coverage is not enough, but that tests should also be designed to show that all possible outcomes of a piece of functionality are demonstrated and that the code actually implements the requirements. Participants also stressed the importance of isolating individual logical elements of the functionality when testing:

> "So, we say that first step is to be able to run the method, and this might mean that we need to mock away some API calls, things like that, because we do not want it to actually affect anything outside." - P2

Effective Test Suite: Participants discussed their views on an ideal test suite, and how that view influences test maintenance. Each test case inside the suite should be meaningful, and test cases should regularly be evaluated to ensure they are up-to-date, relevant, and of high enough quality.

> "In our team, we evaluate the statistics of our test cases. How many product faults are they catching? And in that evaluation, we try to find out [which test cases] are not really performing well and are not very efficient. And then we try to remove them." - P6

It can be easy for a test suite to become bloated over time. Therefore, a significant element of test maintenance is ensuring that irrelevant or redundant test cases are removed. Ensuring that the test suite is lean and relevant reduces the time spent waiting for feedback and clarifies the debugging process.

Regression Testing: Participants also placed importance on ensuring that integration with new code does not affect existing code that has passed testing. Code is naturally interconnected, and it is critical to understand how a change in one part of the source code affects another to prevent unintended side effects. Regression testing is one means of ensuring that unintended changes are detected quickly.

> "We then run automated tests, make sure that the fixes haven't broken anything else in the delicate ecosystem." - P1

Tool Usage: Tools can help ensure quality. Two different types of tools were described. First, linters and other static analysis tools help developers quickly discover problems, such as code smells, before code is committed to the repository.

Table 11. Overview of sub-themes of “Issues Related to Test Maintenance”.

Sub-theme	Description
Hardest Task	Opinions on the most effort-intensive task in test maintenance.
Growing Test Suite	As a test suite grows, it gets harder to interact with and efficiently use.
Proliferating Problems	Interlinked problems, such as dependencies between test suites, require more effort to fix.
Time	Manual testing is slow, and test maintenance is de-prioritized when time is short.
Missing Coverage	Coverage may be reduced during maintenance, or lack of coverage may trigger maintenance.
Unfamiliar with Code	It is harder to find and change unfamiliar test cases.
Technology Limitations	Tech stack limits possible solutions.
LLM Not Good Enough	Help from LLMs feared to be not good enough to justify the effort required to implement suggestions.

Second, the importance of continuous integration was emphasized. All commits must pass automated testing. In addition, code quality and coverage checks are performed as part of the continuous integration pipeline.

> “If I’m breaking anything in the legacy code, I will notice it because the CI tells me something is wrong.” - P8

5.2.3 Issues Related to Test Maintenance. This theme collects challenges affecting test maintenance, summarized in the sub-themes described in Table 11.

Hardest Task: This sub-theme focuses on the most challenging aspects of test maintenance. In particular, participants’ were evenly split on which task was hardest between identifying how the suite must evolve, modifying existing test cases, adding new test cases, and implementing the source change that triggered the need for maintenance. For example, participant P2 thought that identifying the change is the most difficult aspect:

> “Something is wrong, either with the functionality itself or with the test. So first you have to identify what am I changing and why, and then maybe doing it is not that hard. Yeah, because then you have these first steps, right? ... the manual work there is already done, right?” - P2

However, participant P1 thought writing new test cases was harder:

> “Updating the test cases... The big thing is the logic is generally there and it’s usually just one exception that you found, so it’s one specific niche thing. Whereas if you’re writing new test cases, then you really need to think about all of the different ways in which things can go wrong, and you need to look meticulously at every single path through the code and make sure that you’ve covered every path and have two or three different outcomes that could come out of that. Whereas if you’re updating the test code, that’s generally quite small. You know exactly what needs to change.” - P1

Growing Test Suite: As the test suite increases in size, it gets harder to manage, understand, and navigate. In addition, the execution time of the suite increases. A test suite grows naturally as the product is developed, but can continue growing afterwards during test maintenance if care is not taken to remove outdated test cases and to modify test cases instead of simply adding new tests.

> “By adding test cases it will take a longer time to execute, which is a drawback. Even though we test more, it takes a long time and that is one problem that it takes time when we deliver code ... to go through all the test cases” - P3

> “We don’t want to always put in new test cases, because then that would mean we exponentially grow like crazy. So what’s happening is that the maintenance of the test code is as huge as it is to actually develop the product” - P8

Proliferating Problems: Elements of the source and test code are often interlinked, which can affect the test maintenance process. Naturally, a change to the source code affects the test cases that call that code (or that call dependencies of the changed code). Less obvious is that test cases are often—at least indirectly—linked through their collective contribution to the overall test suite. For example, changing a test case may leave the suite more likely to miss certain types of faults, or may reduce coverage of certain outcome of a function that were previously covered by the changed test case.

> "If you change the tests you could actually be introducing or reintroducing bugs that were there. And you could very easily be changing functionality that you didn't intend." - P1

Dependencies between tests make it harder to modify or update them, as the effect of a change on the test suite must, ideally, be understood before making the change. In turn, this can make them hesitant to perform test maintenance, or—at least—increases the time and effort required to successfully perform maintenance.

Time: One challenge noted by participants is that, when a deadline is approaching, testing may be neglected. Test maintenance activities may be delayed or de-prioritized, and risk being forgotten. A second time-related challenge raised by participants is that testing and quality assurance processes are often slow and require more manual effort than would be preferred, reducing the total amount of testing or test maintenance that can take place.

> "I mean what we have is we, we have quite a robust, arduous process in place when making a commit, making a change, ... So the impact that I think this has is the fact that it's a very manual process for us at least and it slows down the time that it takes and it puts in a lot of effort." - P1

Missing Coverage: Participants emphasized that care be taken during test maintenance, as changing or removing a test case may result in a loss of test coverage. When maintenance is performed, it is important to ensure that the overall level of test coverage remains the same, or is improved.

> "A consequence could be that when we remove one test, make changes in the test suite, and then add something else, then the coverage is gone." - P6

Unfamiliar with Code: To perform proper testing or test maintenance, it is important to understand the source code that is being tested. As the project evolves, this task becomes more difficult.

> "I wouldn't say dark parts of our code, but like you know, places where we don't really work. So, I don't really know how they work." - P2

In terms of test maintenance, participants pointed out that it becomes more difficult to change test cases. Instead, new test cases are added, unnecessarily enlarging the test suite. Larger test suites tend to have high test coverage. However, this can mask problems with the test suite. Larger suites are also more difficult to understand, especially for developers that have newly joined the project.

Technology Limitations: The tech stack can also induce limitations on the forms of testing and test maintenance that can take place. For example, there may be limitations on the specific versions of a programming language, testing framework, or other project dependency that can be used. In other cases, the developers may not be able to use certain tools or frameworks because of privacy or security concerns.

> "We're limited in terms of what tech we can use, there are certain products that we can't use for technical or for security reasons and so on." - P1

LLMs Not Good Enough: Some participants expressed concern that LLMs would not be able to offer sufficiently good or trustworthy performance for use in testing or test maintenance. For example, they would not generate sufficiently accurate test code to be worth the effort required to modify their suggestions. They believed that testing work could not be fully automated, and would still require significant human oversight.

> "You know, you still have to tell it: OK, but this isn't exactly what I meant. I need this, OK, but I need that and you've missed this, so you still need that dialogue back and forth." - P1

It should be noted that none of the interviewees had tried to use LLMs for test maintenance and that this experience came from other interactions with LLMs. Still, given the importance of software testing, skepticism towards automated

Table 12. Overview of sub-themes of “Wishlist for Tool Support”.

Sub-theme	**Description**
Generating Code	Generating test code and improving already-written code.
Tracking and Understanding	Discovering how and where problems might occur in source code.
Control Size of Test Suite	Prevent test suite from growing too large and keeping tests relevant and efficient.
Ensure Coverage	Prevent and find faults by ensuring high test coverage.
Ensure Quality from Beginning	Alerting developers to inadequate testing or development efforts.
Automated Execution	Automating execution of tests.
Help and Learning	Making it easier to find and understand new information.
Synergy Between Tool and Team	Tool should fit into the team’s workflow.

approaches is warranted and should be taken seriously. We further discuss participants’ general attitudes towards LLMs in Section 5.2.5.

5.2.4 Wishlist for Tool Support. This theme is focused on the requests and opinions of interviewees regarding automated tool support for test maintenance, whether performed using LLMs or other means, expressed as sub-themes in Table 12.

Generating Code: The most common request for assistance—29% of all codes for this theme—was for code generation. As writing code is one of the main activities in both testing and development, the prominence of this request was not unexpected. Participants noted that this help could come in many forms, from generating entire test cases, to augmenting existing test cases, to simply producing an outline or template for a test.

> “A tool that can generate boilerplate tests. That would be the ideal for me.” - P1

> “So let’s say I have a method and it takes a list and a string. That’s simple. Maybe it should create the empty list on this method and an empty string or something like that. Just have something here so I don’t have to think about it. ” - P2

> “We don’t know what tools exist, but if it could detect the production code changes and then just see what test cases needs to be applied. Just press a button and it will generate all the test cases. That would be great.” - P3

Tracking and Understanding Problems: This sub-theme reflects a desire for tools that detect potential issues in the source or test code—whether a fault, a potentially bad practice, or some other anomaly—explain what type of issues are occurring, and track whether those potential issues have been corrected.

For example, if a failure occurs, the tester may want to know the source of the failure. This could be because some failures are more serious than others, or because particular sources of failure may external to the code-under-test:

> “Today we have a tool that will say [...], is it a product failure or is it an environment failure because that’s something happening here in the lab. We don’t want to spend time on the environment problems, just fix them. We want to spend time on product failures; for this purpose, it is important that this tool has a very good success rate.” - P8

After attempting to fix the cause of a failure, a developer may want to know whether the same failure re-occurs. Tools support could monitor for known failures or anomalies.

> “You know, if something’s gone wrong before then you don’t want to repeat it and you want to at least be alerted if it goes wrong again. So we create those dashboards manually and then create alerts based on that. But I can easily see how a generative AI, for example, could go and look at the data set and see, OK, this is normal behaviour, this is what I expect and then flag what goes wrong.” - P1

Participants pointed out that such tools have immense potential value as significant effort is spent on finding the root causes of, and correcting, failures.

> “What is hindering us from becoming even better is the time it takes for us to actually correct a failure ... How do we correct the failure? Quick problem identification is key to improve this...” - P8

Control Size of Test Suite: As previously noted, issues emerge over time because the test suite is unnecessarily large, containing redundant or irrelevant tests. Multiple participants sought tool support for detecting situations where the test suite could be reduced without losing quality.

> “There is also a limitation in how long testing can be done. ... So in that case, we also think that because a test case is quite old and not really catching a lot of product fault, then we, ourselves, decide to remove this one. ” - P6

Controlling the size of the test suite could be done by a tool providing information on how test cases overlap in code coverage or coverage of equivalent functional outcomes. Alternatively, tools could detect test cases that are potentially outdated—e.g., where new test cases have been added instead of modifying existing test cases when the source code has changed—for modification or removal.

> “Maybe we can use this AI or machine learning to automatically select test cases for changes.” - P7

Ensure Coverage: Since having high coverage is recognised as an important way to assure the quality of the final product, getting help achieving that coverage is important as well. As attaining high coverage is time-consuming, multiple interviewees expressed a desire for tool assistance.

> “Just having something that makes sure that our coverage is better there, then we know that we can handle different cases in it. That would increase my confidence in the code itself, I think.” - P2

The most obvious form of assistance would be to simply generate test cases. However, participants also suggested that tools could augment the input applied in existing tests.

Ensure Quality From Beginning: Participants expressed a desire for tool support that detects problematic development or testing practices from the start, thereby lessening the need for test maintenance. For example, one participant stated that test coverage may be higher from the start if a tool would alert than to deficiencies that should be corrected:

> “To increase coverage, we could have something telling us coverage is bad and maybe forcing it a little bit more on us. You run Pytest and it passes and then you’re happy usually, but you don’t look at the coverage report.” - P2

Another participant requested a tool that could pre-check for potential integration issues when adding functionality:

> “I want to have something here that can help the developer to actually pre-check the feature interaction.” - P8

Automated Execution: Although aspects of test execution are already automated, there are still steps of the testing process that require manual execution, such as running forms of static analysis on the code or locally executing unit tests before committing the code to the repository. Particularly taxing are cases where test execution itself must be performed manually. Participants sought further automation of the testing process, including assistance with transforming manual test cases into automated test cases.

Help and Learning: Participants also expressed the desire to mitigate a lack of knowledge or experience with the help of tools. For example, this could be help in learning how to write tests in a new language or testing framework, or tips on how to make a method more testable.

> “It would definitely cut down a lot on development time if you have something that can give you a few general starting points, and especially when it comes to using new technologies.” - P1

Table 13. Overview of sub-themes of “Attitudes Towards Generative AI”.

Sub-theme	Description
Skeptical	Do not think LLMs can efficiently help with software development.
Unfamiliar	Have not used LLMs, either due to lack of opportunity or interest.
Positive	Like the idea of getting help from LLMs.

Synergy Between Tool and Team: Most industrial software development is done in teams, and their way of working has a large effect on how effective any activity—not just test maintenance activities—can be. Incorporating a new tool requires some consideration to make sure that current ways of working are not disrupted, and that code or test quality are not made worse because of the disruption.

> “We have a structure to creating tests right? We start by trying to run the method. We check the output and maybe for the first step at least you’re covered. The assertions, maybe, I want to do myself because I’m not expecting the AI to understand what my method is ... I don’t think it can understand that, but maybe understanding kind of the basics of how our pipeline works. We create the method, we have a test on it, we always instantiate the method.” - P2

5.2.5 *Attitudes Towards Generative AI.*

As part of the interview, the interviewees were asked to give their general opinions on the use of generative AI—especially LLMs—as part of software development. These attitudes are an important way to gauge the feasibility and desirability of implementing LLM-based tool support for test maintenance. Opinions could largely be categorized, as explained in Table 13, as skepticism, unfamiliarity, or positivity.

Before explaining each, it should be noted that the number of codes (Figure 7) does not indicate the number of participants who feel a particular way, and individuals offered multiple opinions. For example, the same individual may be both positive towards trying LLMs while still skeptical about certain aspects of their use.

Unfamiliar: A few of the participants were unfamiliar with LLMs and generative AI in general. This mainly stemmed from the fact that, for the majority of the time, they are not yet allowed to use such tools for their work. This is caused by security and privacy risks in providing data to third-party software. Although Ericsson has approved some LLMs, this is still very recent, and not a lot of employees have used it. Beyond that, there was a general lack of interest from some participants that prevented them from trying LLMs in their free time.

Skeptical: Some participants expressed skepticism about the abilities of LLMs. This could stem from either a bad experience, things they have heard, a fear that LLM capabilities are over-hyped, or concern that LLMs are being used prematurely to automate certain aspects of development.

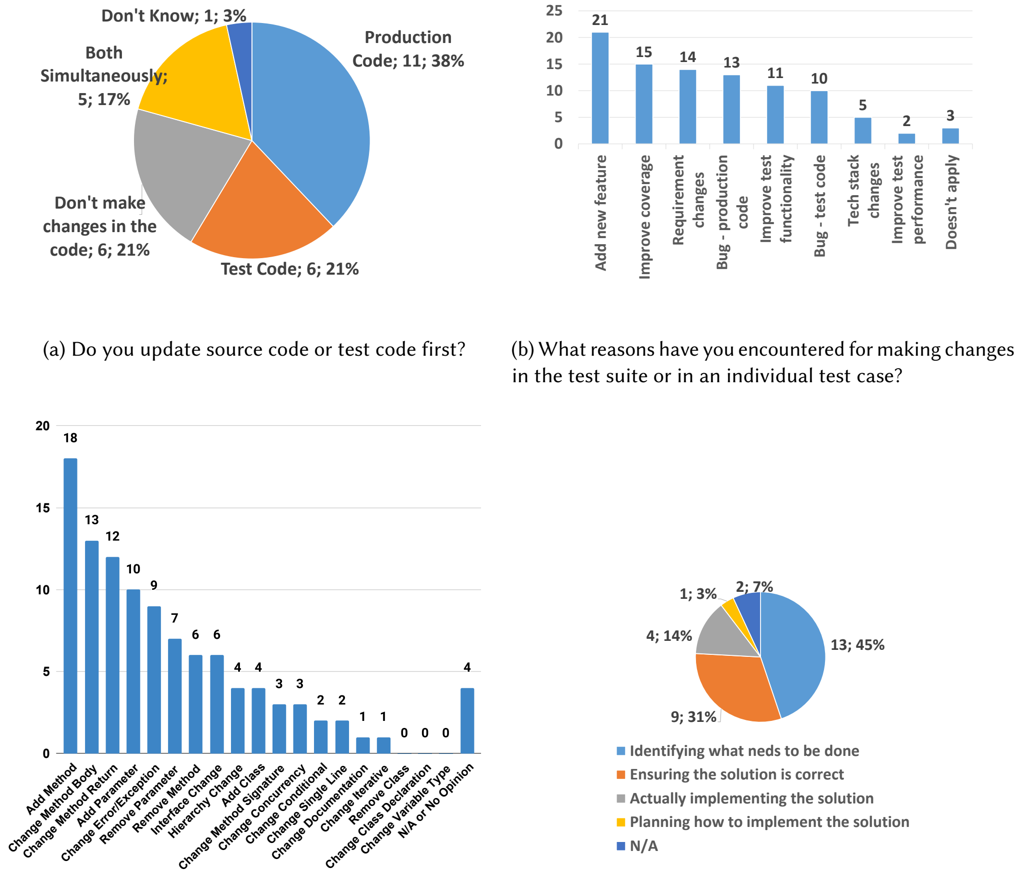
> “At the same time, though, am I ready to trust it? To cover all of those bases? No, I’d still like to have human eyes over it. Somebody with some background and the code itself to look at things.” - P1

As noted previously, Ericsson must approve LLMs for internal use, and is still in an early experimental phase of LLM adoption. Therefore, some teams have recommended waiting and not using the currently-approved LLMs.

> “[...] in Scrum master meeting they recommended not to use that [LLM] because it can’t track a lot of information [...]” - P5

Positive: Despite some skepticism, five out of the eight interviewees did state that these tools could help out in various ways in their work, either at the moment or in the future.

> “I’m quite keen on the idea of generative AI taking over those types of tasks.” - P1



(a) Do you update source code or test code first?

(b) What reasons have you encountered for making changes in the test suite or in an individual test case?

(c) Which types of source code changes most commonly necessitate adjustments in associated test code?

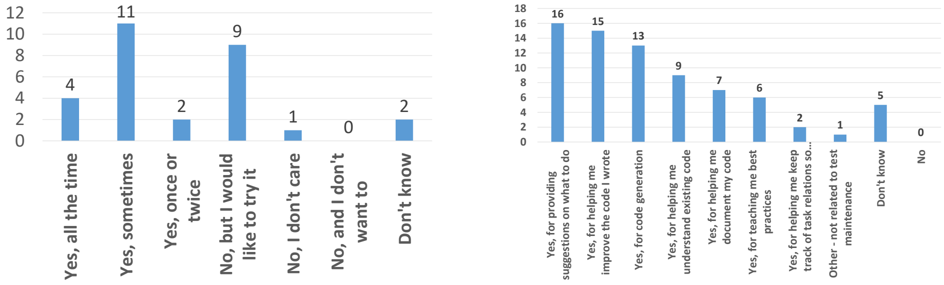
(d) During the test maintenance process, which step is most effort-intensive?

Fig. 8. Survey responses to questions about the test maintenance process.

5.3 Analysis of Survey Responses (RQ1–2)

This section presents quantitative results from the test maintenance survey. Responses to questions related to test maintenance are shown in Figure 8, while responses related to automation and LLM use can be seen in Figure 9.

With regard to test maintenance, while changes to the source code are often made first, there are many cases where test code is changed first or both are changed simultaneously. This suggests that test maintenance is not always caused by an initial change to the source code—i.e., the list of triggers in Section 5.1 is not exhaustive. This is further evidenced in Figure 8(b), where several common reasons for test maintenance—i.e., improving coverage, improving test functionality, and fixing a bug in test code—do not require a change to the source code. Still, the source code is often updated first, and the most common reason for maintenance is the addition of a new feature.



(a) Have you used generative AI or LLMs?

(b) Would you like the help of generative AI/LLMs for test maintenance activities?

Fig. 9. Survey responses to questions about automation and LLMs.

The split in responses in Figure 8(a) could also indicate that changes to the source and test code are, at times, planned together. This would result in simultaneous change, and—in test-driven development—even a change to the test code in anticipation of the change to the source code.

When test maintenance is triggered by a change in source code, it is most often associated with the addition of a method. This is followed by a change to a method body or method return type or value. Three triggers identified in literature (Section 5.1 were not reported by respondents as being among the five most common reasons for test maintenance—removing a class, changing a class declaration, or changing a variable or object's type. While these should not be ignored as potential triggers, it should be understood that not all triggers occur with the same frequency, and that a particular type of source code change may not always require test maintenance.

Finally, as shown in Figure 8(d), the most effort-intensive task was identifying what change needed to be made, followed by ensuring that the change was implemented correctly. Tool support—e.g., the use of LLMs to identify required changes and to identify shortcomings in solutions—could potentially help with both tasks.

With regard to LLM use, Figure 9(a) shows that a majority of respondents had—at least—tried to use an LLM. Approximately a third of respondents still have not tried an LLM, but almost all expressed a desire to try one eventually.

As shown in Figure 9(b), all respondent expressed a desire to use an LLM in some form as part of test maintenance. The most common requests for assistance included providing suggestions on how to perform test maintenance, for improving the code that they had written, and for generating test or source code. Code generation was only the third most-common desire, suggesting some skepticism and desire to remain in control of test suite evolution—with the LLM supporting human effort.

5.4 Literature Review on LLM Capabilities for Test Maintenance (RQ2)

We performed a literature review on the applications of LLMs in software testing and development to, first, suggest test maintenance actions that LLMs may be able to perform and, second, the considerations that must be made when deploying LLMs in an industrial environment. As we did not find any research specifically on test maintenance, we examined software testing and development more broadly and considered publications that discussed applications with relevance to test maintenance.

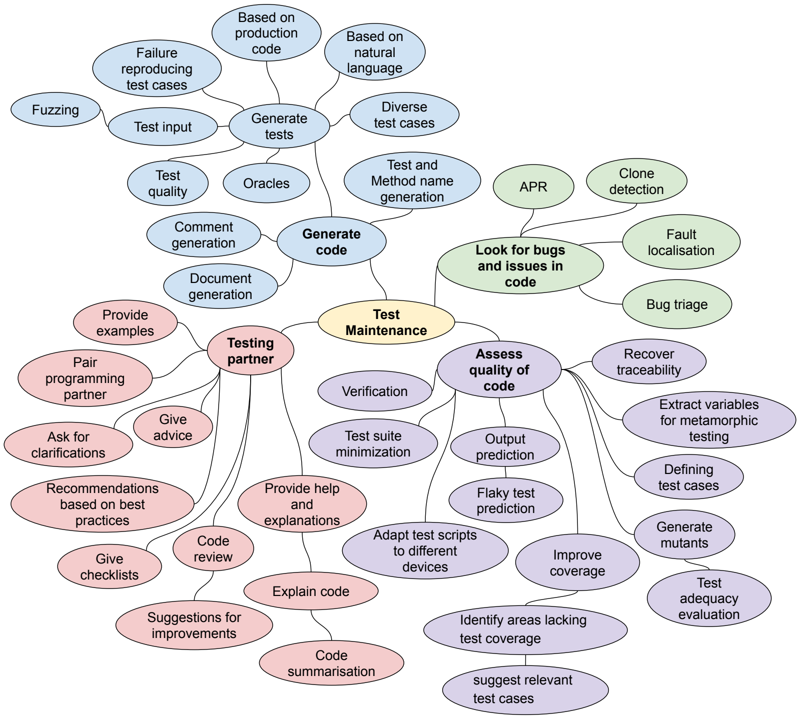


Fig. 10. Overview of actions an LLM could take that could assist with test maintenance. Colors indicate clusters of related actions.

Test Maintenance Actions: Figure 10 offers an overview of the actions—suggested by past literature—that an LLM could take that could assist with the test maintenance process. We divide these actions into four clusters, indicated by color in Figure 10, including code generation (blue), testing partner (red), assessing code quality (purple), and looking for bugs or other potential issues in the code (green).

One of the most common uses of LLMs within software engineering is code generation (e.g., [12, 24, 31, 42, 45, 54, 82, 91, 93, 96, 121, 128]). Naturally, one of the most researched applications of LLMs in software testing is test generation, including both inputs and test oracles (e.g., [11, 27, 28, 31, 45, 61, 97, 101, 114, 125, 128, 128]). Part of the test maintenance process is the creation of tests for new code, and test generation could automate or assist with this task.

In particular, due to their inference capabilities, LLMs have shown adeptness at generating both syntactically valid and semantically appropriate test input, e.g., for filling out forms in GUI testing [70, 135] or identifying test input for deep learning libraries [27, 28]. LLM-generated test cases can suffer from hallucinations, resulting in non-compiling code. However, an agent setup where the LLM is automatically provided with compilation errors or where test creation is divided into sub-tasks such as planning and understanding can overcome this challenge [125].

An important part of test maintenance is establishing traceability between source and test code, as well as other project artifacts. LLMs have been used to identify traceability links between natural language and code artifacts [67, 133].

Of course, it is important to not just understand what code has changed, but to understand how it has changed. LLMs can explain and summarize code [45, 81, 82, 128]. These same capabilities could extend beyond explaining the code in a general sense, to also explaining how it has evolved.

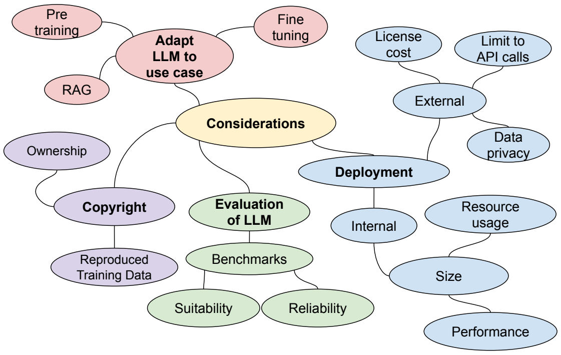


Fig. 11. Considerations for LLM deployment in an industrial environment. Colors indicate clusters of related considerations.

The conversational nature of LLMs could also bring benefits to the testing process. An LLM can act as a testing partner, offering advice, examples, explanations, and checklists. The user can ask follow-up questions for clarification [32]. Researchers have explore the use of LLMs as pair programming partners [14, 93]. Test maintenance is a process where a developer requires a deep understanding of both the source and test code, as well as knowledge of good testing practices. LLMs could offer conversational assistance.

LLMs can perform code review, providing suggestions for improvement and optimization [12, 45, 71, 128]. Such capabilities could be applied to the test suite. In addition, LLMs could be used identify weaknesses in the test suite through mutant generation [31, 45, 48, 128], defining additional test scenarios to explore [96], identifying areas lacking code coverage [31, 45], and identifying flaky tests [31, 45]. LLMs have also been shown to be capable of adapting test cases to different devices [124], extract properties for metamorphic testing [108], improve test readability [34], and minimize test suites [11, 31]. LLMs have also been used to generate documentation [31, 42, 45, 82], code comments [34, 35, 82], as well as method and test names [34, 45]. All of these capabilities could be used to improve or otherwise modify the existing test suite, replacing part of the human effort in test maintenance.

A common maintenance task is to add test cases that reproduce bugs, and LLMs have been applied to generate failure-reproducing test cases [55]. LLMs can also be used localize faults and as part of bug triaging [11, 45, 82, 114]. Once localized, LLMs have also been used to perform automated program repair [45, 121, 126]. Generally, LLMs have shown some capacity for identifying both faults and other issues (e.g., code smells) in the source code [12, 24, 31, 42, 45, 65, 82, 96, 128]. These same capabilities may have applications as part of the test maintenance process as well.

Considerations for LLM Deployment in Industrial Development: An overview of considerations discussed in past literature that affect the deployment of LLMs in industrial environments is shown in Figure 11. There are many different LLMs, with more emerging regularly. These considerations are not exhaustive, but can help guide LLM selection.

Different LLMs have varying performance across different tasks, depending on how they have been trained and tuned. For example, Roziere et al. noted that—in their study—the Code Llama model performed better than the GPT-4 and GPT-3.5 turbo models for code generation tasks [94]. This was because Code Llama was specifically trained and tuned for source code-based tasks, while the other models were trained on a mixture of code and natural language and were intended for reasonable general performance. Naturally, then, the first consideration when selecting a model is the types of tasks that the model has been designed for. A model tuned for a particular task will almost always outperform models of the same size—measured in the number of parameters [131].

However, there are still likely to be many models appropriate for a given task. Evaluations on public benchmarks can be used to perform a comparison. There are well-known benchmarks for code generation tasks, for example, such as HumanEval [23]. The LMSYS Chatbot Arena also ranks LLM performance in tasks where they serve as conversational assistants [25]. However, many software engineering tasks do not yet have “standard” benchmarks [114].

Benchmark performance may also be affected by data leakage. This is especially a risk for widely-used benchmarks or those not designed specifically for LLMs [114]. In addition, current benchmarks, such as HumanEval, have been criticized for having insufficient test suites to judge the correctness of generated code and for using vague problem descriptions [68]. Both factors could result in misleading performance judgements of LLMs, with weak test suites offering inflated assessments and vague problem descriptions causing otherwise capable models to be judged as incapable.

Benchmark performance is still a valid starting point, but as a result of these factors, stated performance scores may not reflect real-world performance on industrial codebases. Unless a company trains their own model, industrial code should not appear in the training data for the LLM, preventing data leakage. Industrial test suites are also likely to be stronger than ones built for a benchmark.

In general, the larger the model, the better it performs [131]. Larger models also demonstrate emergent abilities, such as in-context learning [18] and step-by-step reasoning [98], that are not always found in smaller models [130]. That said, larger models impose significant computational and energy costs [114]. Models can be fine-tuned on an industrial codebase to improve their performance, and smaller fine-tuned models can sometimes outperform larger models [131].

Beyond technical factors related to model performance, there are also a number of organization factors to consider when deciding whether to deploy a particular LLM [73]. If considering a model deployed by an external company, the cost of a license and limitations on the number of API calls must be considered [73]. In addition, there may be significant data privacy concerns, related to sending proprietary or customer-related data to a third party [114].

If an organization can afford the computational costs of an internal deployment, then that is generally a better option. Internal deployment avoids privacy concerns [114], and offer greater control over the deployment. For example, the organization can tune such models, can change their parameter settings, and can choose when and how to deploy updated models—these aspects are not always controllable when the models are controlled by an external firm [114].

An additional concern is copyright. LLM output, including both code and natural language, is derived from training data. As a result, LLMs may include copyrighted text in the output, which could lead to issues for an organization—e.g., if copyrighted code is incorporated into the organization’s own codebase [64, 79]. In addition, it is unclear whether an organization can own the copyright over code generated by an LLM [36]. Copyright issues can be mitigated by choosing an open-source LLM where the training data can be inspected. However, open-source LLMs are less common than “open-weight” models, where weights are available but not underlying training data [88].

5.5 Proof-of-Concept Evaluation (RQ3)

We present the results for each prototype in Table 14. The planning agent with summaries attained the highest precision (0.2596), recall (0.3367), and F1 score (0.2932).

We observed three general trends in the results. First, there appears to be benefit from using summaries over using the test code directly to make predictions. In many fields, analyses are simplified by filtering unnecessary or distracting details. LLMs have also been shown to benefit from relying on summaries rather than raw data. We see the same effect in our evaluation. For both architectures, there is an increase in both precision and—especially—recall between the version with summaries and the version without summaries. Recall reflects the percentage of test cases that needed

Table 14. Comparison of results of the prototypes. The highest performance in each measurement is **bolded**.

Prototype	Recall	Precision	F1 Score
Planning (With Summary)	**0.3367**	**0.2596**	**0.2932**
Planning (Without Summary)	0.1867	0.2343	0.2078
Chain (With Summary)	0.3000	0.1863	0.2299
Chain (Without Summary)	0.2233	0.1138	0.1507

maintenance that were correctly identified. The use of summaries may have made it easier to determine which aspects of the code were invoked by each test case.

In addition, there appears to be a benefit to precision from employing a planning agent over a pipeline architecture. Employing a planning agent introduces the ability to repeat aspects of the analysis if an individual LLM agent or instance fails to deliver a satisfactory solution. Precision reflects the proportion of the tests flagged by the framework that actually required maintenance. An improvement to precision could emerge from repeating a failed analysis, e.g., more test cases could emerge that were missed before or re-analysis may remove test cases that have a high uncertainty associated with their inclusion to the set predicted to need maintenance. However, the use of a planning agent also introduces an additional risk of not delivering a solution within the iteration limits. The planning agent prototypes hit the iteration limit significantly more often than the pipeline prototypes.

Third, there were many commits in the data set where no test maintenance was performed. When examining performance on individual commits, all prototypes performed significantly better on commits where test maintenance was actually performed, compared to commits where the correct prediction is “no test cases need maintenance.” This suggests that the identified triggers are too general—such code changes *can* indicate a need for test maintenance, but do not guarantee such a need. Additional safeguards and consideration of the list of triggers may be needed to evaluate whether test maintenance is really necessary for a given change to the source code.

6 Discussion

In this section, we provide answers to the research questions. We also discuss directions for future research, AI ethics, and threats to validity.

6.1 Test Maintenance Triggers (RQ1)

Observations from the literature review, interviews, and survey suggest that the test maintenance triggers can be divided into *low-level triggers* related to specific changes to documentation or source code and *high-level triggers* that represent decisions made during the development process.

Low-level Triggers: Past research on test maintenance has focused predominately on changes to the source code or documentation as being triggers of subsequent test maintenance. As indicated in Figure 6, these changes can occur at different levels of granularity—broad changes to functionality, to documentation, to a single class, to a method, or to a single line of code. These triggers were discussed in Section 5.1, and summarized in Tables 4–7.

That low-level changes can trigger test maintenance is intuitive. If a change occurs in the project, then a thorough test suite that exercises the changed aspects of the code must evolve as well. Likewise, if new functionality is added or removed, then tests should be added or removed as well. As may be expected, many of these triggers were also raised in the interviews and survey.

Table 15. Description of high-level triggers that may indicate a need for test maintenance.

Trigger	Description
Adding New Feature	Expanding on an existing product by adding new functionality, leading to the creation of new tests and adjustment of existing tests.
Change in Requirements	Existing feature requirements have been modified, leading to a need to change both source and test code.
Discovery of a Fault	The discovery of a fault in the code base, either by a developer or documented in an issue report, cause a need to add or modify tests to ensure the fault has been repaired and that it will be caught if it appears again.
Increase Coverage	The test suite is modified and new tests are added to increase code coverage.
Refactor Code	The code is refactored or enhanced with the intention to preserve the current functionality while improving quality, e.g., its performance. Both test and source code can be refactored, both can require test maintenance.
Tech Stack Changes	Changes in technologies used in the development process may require test maintenance.
Evaluation of Test Suite	Systematic evaluation of a test suite may identify weaknesses and lead to maintenance.

However, it is important to emphasize that these triggers *can* indicate a need for test maintenance, but *do not guarantee* that maintenance is required. In practice, a test will only need maintenance if the test executes the changed lines of code and if the test is somehow no longer in alignment with the the changed source code—e.g., there has been some change in how the code is invoked or a change in its behavior that is detected by the test oracle.

A test suite may not execute changed code, or if it does, it may not detect the change (e.g., due to a weak oracle, low input diversity, or low coverage of alternative paths containing changed code). In such cases, the test suite may not require maintenance—although a lack of need for maintenance, in itself, may be an indication that the test suite is weak. It may also be the case that test maintenance was not performed until a later commit, as we only considered co-evolution that occurred in the same commit.

Some of these triggers are also broad enough that they may not be useful for prediction purposes. For example, "removing functionality" can describe many different code changes. Because it is so broad, it could be considered a trigger, even for changes that do not necessitate test maintenance.

Alternatively, it may be that these triggers should be combined with additional data when used for prediction and decision-making. In our prototypes, we did not provide the LLM agents with code coverage information. Instead, the agents made their own determination of traceability between source and test code. Coverage information could ensure, at least, that the changed code is covered by a test. Combinations of triggers and other analyses of source and test code or other artifacts may be needed to determine whether test maintenance is actually required. Future research should examine the causal relationship between source and test code evolution in more detail.

Test Maintenance Triggers (RQ1): We identified 37 low-level changes to functionality, documentation, classes, methods, and single lines of code that can trigger a need for test maintenance. However, such changes only trigger a need for maintenance in situations where test cases invoke the changed code and detect changes to its behavior.

High-Level Triggers: From interview and survey responses, we also identified high-level triggers of potential test maintenance. By "high-level", we refer to decisions made as part of the development process rather than individual changes to the source code. For example, there may be a decision to increase the code coverage of the current test suite. There may not have been a change to the source code—hence, no low-level trigger—but new test cases may be added or existing test cases may be augmented. Like with the low-level triggers, such development decisions do not always guarantee a need for test maintenance, but are frequently associated with it.

We present these high-level triggers in Table 15. These triggers were not integrated into the prototypes developed to address RQ3. However, we discuss how they could be used by LLM agents in Section 6.2.

Test Maintenance Triggers (RQ1): We have identified seven high-level development decisions that frequently trigger a need for test maintenance, including adding new features, requirements changes, fault discovery, increasing the coverage of a test suite, refactoring, tech stack changes, and test suite evaluation.

6.2 Applications of LLMs in Test Maintenance (RQ2)

The purpose of RQ2 is to examine the theoretical applications of LLMs or LLM agents within the test maintenance process. We consider three aspects—which triggers could be acted upon, what actions could be taken based on these triggers, and what considerations should be made when deploying LLMs for these tasks within an industrial environment.

Use of Low-Level Triggers by LLMs (RQ2.1): The low-level triggers reflect specific changes to the source code. Because low-level triggers can be detected in the code, they can be directly used by LLM agents as the impetus to perform various actions on the test code. In other words, if alerted to the presence of a subset of the triggers, an LLM or LLM agent can determine which corresponding actions to take (or not take) on the test code.

We demonstrated such a scenario in our prototypes, which used the triggers to determine which test cases may require maintenance. The prototypes showed that this theoretical scenario is possible to perform using current-generation LLMs. As previously noted, however, the low-level triggers only indicate a potential need for test maintenance. Additional context is still required to determine whether test maintenance is required.

Figure 12 illustrates one extension of the scenario modeled in our prototypes. Once the need for test maintenance is predicted, additional LLM agents could automatically suggest or directly perform modifications on those test cases. The low-level triggers could be further used as part of performing these actions, as some triggers may suggest certain corresponding actions. In addition, LLM agents could perform code analyses—e.g., inspecting code quality or code coverage—to identify specific modifications to make to the test suite. These analyses could be performed by an LLM, or an LLM agent could call existing static and dynamic analysis tools and incorporate their results into determinations.

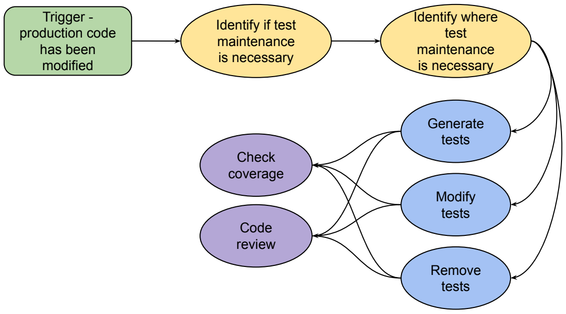

Trigger - production code has been modified
Identify if test maintenance is necessary
Identify where test maintenance is necessary
Generate tests
Modify tests
Remove tests
Check coverage
Code review

Fig. 12. LLM agents could generate tests, or modify or remove existing tests, based on a combination of triggers, code coverage, and code quality review. Green = trigger, yellow = explored aspects, blue = new test maintenance actions, purple = new code analyses.

Use of Triggers by LLMs (RQ2.1): Our prototypes illustrate that low-level triggers can be used by LLM agents to perform test maintenance actions. Future work should augment the triggers with additional contextual information (e.g., coverage or code quality analyses) to increase the accuracy and complexity of the actions taken.

Use of High-Level Triggers by LLMs (RQ2.1): As the high-level triggers reflect software development decisions, these triggers could be used by LLM agents as an impetus to perform or assist in high-level development and testing activities. For example, an LLM agent could be given a change in the requirements as input, reason about how this change would affect the relevant test cases, then perform the modification of those test cases. This theoretical agent may divide the task into sub-tasks, which could be performed by other LLM agents.

Figure 13 suggests the types of actions that LLM agents could take based on high-level triggers, including general actions, code modifications, and actions intended to improve test or source code quality that use these code modifications. Figure 14 suggests specific actions for each trigger. These suggestions have been extracted from interview and survey responses, as well as from past literature.

For example, if there is a change in the project requirements, an LLM agent could identify the artifacts that need to be adjusted. These could include both source and test code, as well as other artifacts such as user stories. A second

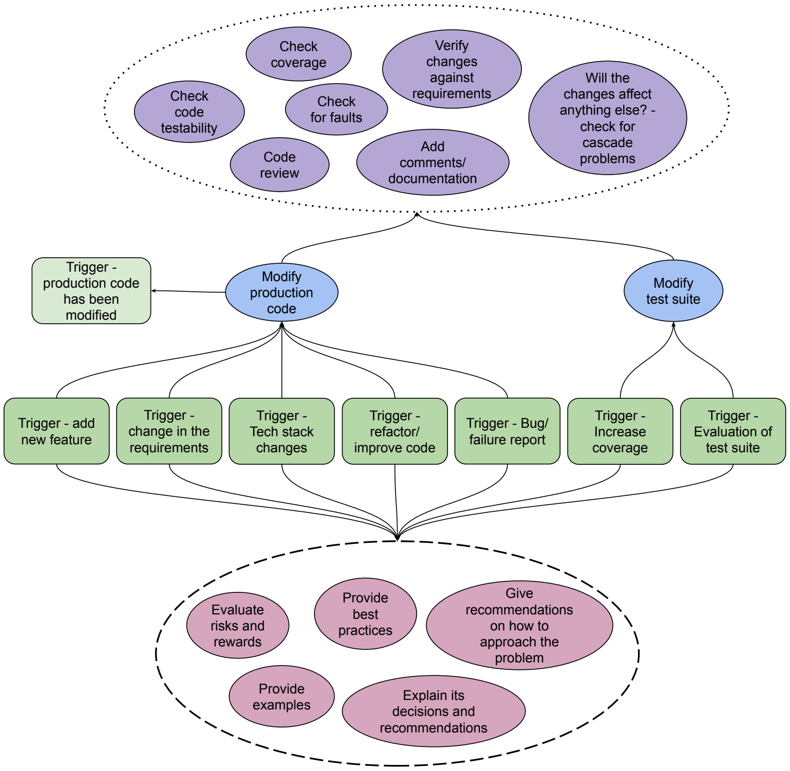

Check coverage
Verify changes against requirements
Check code testability
Check for faults
Will the changes affect anything else? - check for cascade problems
Code review
Add comments/ documentation
Trigger - production code has been modified
Modify production code
Modify test suite
Trigger - add new feature
Trigger - change in the requirements
Trigger - Tech stack changes
Trigger - refactor/ improve code
Trigger - Bug/ failure report
Trigger - Increase coverage
Trigger - Evaluation of test suite
Evaluate risks and rewards
Provide best practices
Give recommendations on how to approach the problem
Provide examples
Explain its decisions and recommendations

Fig. 13. General actions an LLM agent could take based on a high-level trigger. Green = high-level trigger, light green = low-level trigger, purple = potential code or test quality actions, pink = general LLM actions, blue = code modification actions.

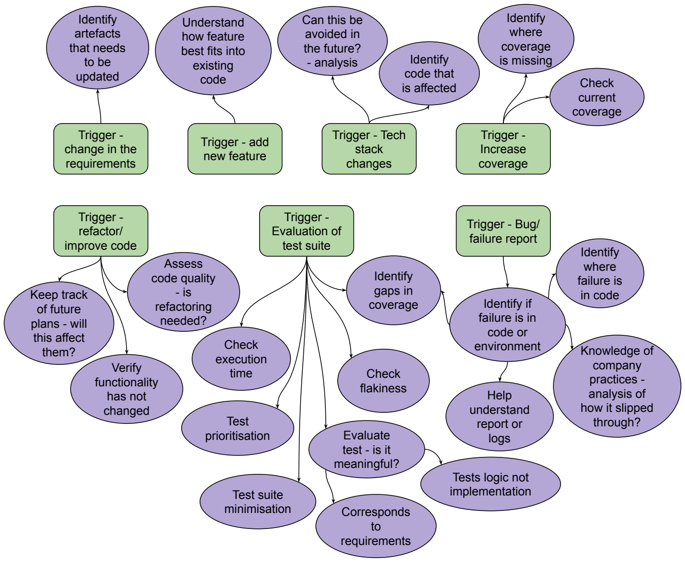

Identify artefacts that needs to be updated
Understand how feature best fits into existing code
Can this be avoided in the future? - analysis
Identify code that is affected
Identify where coverage is missing
Check current coverage
Trigger - change in the requirements
Trigger - add new feature
Trigger - Tech stack changes
Trigger - Increase coverage
Trigger - refactor/ improve code
Trigger - Evaluation of test suite
Trigger - Bug/ failure report
Assess code quality - is refactoring needed?
Keep track of future plans - will this affect them?
Verify functionality has not changed
Identify gaps in coverage
Check execution time
Check flakiness
Test prioritisation
Evaluate test - is it meaningful?
Tests logic not implementation
Test suite minimisation
Corresponds to requirements
Identify where failure is in code
Identify if failure is in code or environment
Knowledge of company practices - analysis of how it slipped through?
Help understand report or logs

Fig. 14. Suggestions on specific actions an LLM agent could take based on high-level triggers to improve the test suite quality. Green = high-level triggers, purple = potential LLM actions.

agent could then recommend best practices for updating the code and test suite, while another could assess risks and rewards of different solutions. A fourth agent could perform the modifications to the source code, while the fifth could generate tests that verify that the code meets the new requirements. A final agent could then evaluate code quality and suggest refactoring that would improve performance.

Use of Triggers by LLMs (RQ2.1): LLM agents could act on high-level triggers by automating or assisting in development or testing activities such as providing examples or guidance, evaluating and offering recommendations on how to improve potential solutions, or modifying the source code or the test suite.

Test Maintenance Actions (RQ2.2): Interview and survey responses—as well as the literature review—suggest that LLMs have the potential to perform or assist with the following types of test maintenance tasks:

- Establishing traceability between development artifacts.
- Helping developers understand where maintenance is needed.
- Generating or adapting existing test cases.
- Explaining or summarizing code.
- Acting as a testing or pair programming partner. LLMs can provide examples and advice, and because of their conversational nature, the developer can ask for explanations, clarifications, and follow-up questions.
- Assisting with general software engineering activities that might appear during the test maintenance process, such as code review, analyzing and improving source and test code quality, quality assurance, evaluating existing test cases and code coverage, and generating documentation.

Our prototypes, developed to address RQ3, demonstrate that LLMs can summarize code and explain the differences between new and old versions of the same code. The prototypes can also establish traceability, identify the presence of low-level triggers in source code changes, and use those triggers to predict the need for test maintenance. We have not yet developed a proof-of-concept for the other use cases, e.g., acting as a conversational partner. However, Figures 12–14 illustrate how the triggers could be used by LLM agents to perform certain actions.

Significant research efforts are being dedicated to the use of LLMs for code and test generation, and such efforts are valuable. However, we also would like to emphasize the value of LLM agents for tasks beyond code generation, such as code review or acting as a testing partner. Test maintenance is more than the simple process of adding new tests for new code and updating tests when the code changes—it is a long and complex process intended to improve the overall test suite quality for the project. This process includes many tasks, such as assessing the strengths and limitations of the existing test suite, improving the efficiency of the test suite, and trying to improve the diversity of the scenarios tested. In particular, LLMs show great potential for tasks that blend source code and natural language, and researchers should explore a variety of these use cases within the context of the test maintenance process.

Test Maintenance Actions (RQ2.2): LLM agents can perform or assist with a variety of test maintenance activities, including—broadly—code generation or modification, assisting developer understanding, acting as a conversational assistant, and assisting with general development activities such as code review or documentation.

Industrial Considerations (RQ2.3): Our literature review on LLM capabilities suggests the following considerations for LLM deployment in an industrial environment:

- Except for code generation, there are few established benchmarks for software development tasks. Further, existing benchmarks may offer inaccurate estimations of LLM performance due to data leakage, insufficient test suites, poorly-defined prompts, or because the benchmark was not designed for LLMs. As a result, benchmarks should be taken as a basic indication of LLM capabilities, but not as an absolute estimation of their real-world performance. In general, the larger the size of the LLM (in number of parameters), the better it will perform without tuning. However, larger models also carry significant computation and energy requirements.
- If the computational demands can be met, deploying open-source or open-weight models in-house is generally a better option for an organization than using a model controlled by an external organization. In-house deployment offers greater control over the model, opportunities for tuning on organization data that may not otherwise exist, and avoids security and privacy risks from sending proprietary or customer data to another organization.
- (Re-)training or fine-tuning a model on organizational data can be an expensive and time-consuming process. Developing an LLM agent may be a more efficient option, as the agent could pair an LLM with a RAG tool to access up-to-date information on the codebase. Since chunking data is a much faster task than training or tuning a model, RAG tools can be easily updated. Over time, the model could also be tuned or re-trained to attain even better performance.
- Copyright infringement and ownership of AI-generated output are also significant factors that must to be carefully analyzed when LLMs are used to generate or modify code. Using open-source models, where the training data is public, can mitigate these concerns.

The interview and survey responses suggest additional considerations:

- Skepticism should be taken seriously—trust in LLMs must be earned through positive experiences. Expectations on LLM performance should be set carefully, i.e., the organization should not promise results that will not be attained. In addition, LLM deployment should be done slowly and carefully to ensure that the models are sufficiently effective before all developers are encouraged to use them as part of their daily work.

- One of the main wishes that Ericsson employees had for support from LLMs was, naturally, generating and improving existing code. However, their test maintenance challenges were much more varied, as are LLM capabilities. Organizationally, LLM usage could benefit many different development and testing tasks, and these use cases should be widely explored.
- Many test maintenance challenges are rooted in a lack of knowledge, both project-specific and general (e.g., on testing practices). For example, to perform test maintenance, one may need to familiarize themselves with the source code, investigate the cause of a fault, or evaluate how code coverage is affected by code and test suite changes. Some of the most valuable capabilities an LLM agent can provide when analyzing project artifacts are summarization, insight, and recommendations.
- Our observations suggest that LLMs should be deployed in a human-in-the-loop configuration. Even if an LLM can automate a task, performance limitations and skepticism suggest that LLM output should be treated as a "recommendation" that is reviewed by a human. LLMs also have great value as conversational testing partners. An organization should not view LLMs as a way to replace human effort, but as a way to augment the effectiveness and efficiency of the human effort spent on test maintenance.

Industrial Considerations (RQ2.3): When deploying LLMs, organizations should consider model performance and size, whether to deploy in-house or to rely on an externally-controlled model, potential copyright issues, how to tune the model on company data (e.g., retraining, fine tuning, or using a RAG tool), how to establish and manage company culture and community expectations, which tasks the LLM should support, and whether tasks should be fully automated or whether LLM output should instead be offered as a recommendation to a human developer.

6.3 Proof-of-Concept Evaluation (RQ3)

Prototype Performance: Of the four prototypes that we implemented, the best performing was the planning agent that used test summaries, with an F1 score of 0.2932. The results suggest that employing a central planning agent yields improved precision over a pipeline-based architecture, and that employing summaries yields improved recall over working with test code directly.

All four prototypes demonstrate that it is feasible to employ LLM agents within the test maintenance process, and offer a starting point for future research and development. However, the performance of all four prototypes is not yet sufficient to be deployed in practice.

The most comparable approach from previous research is the DRIFT technique proposed by Liu et al. [69]. DRIFT achieved an average F1 score of 0.6950 for cross-project test maintenance predictions. We cannot directly compare performance, as they trained their model on different projects and we were not able to re-implement DRIFT for the industrial codebase used in our study. They also employed different data processing techniques. However, we can consider the performance of DRIFT as a "performance target." In addition, general advice regarding the F1 score [19] suggests that an F1 score between 0.5–0.8 is considered "acceptable", 0.8–0.9 is considered "good", and 0.9–1.0 is considered "very good." By both points of comparison, there is clearly room for improvement.

As discussed in Section 4.7, the prototypes used the Mistral 7B - Instruct model, which is a small and relatively weak model [3]. In future work, newer and larger models could be deployed for better performance [18], e.g., the Mistral Large or Mistral 8x22B models [5]. For code-based tasks, it may also be better to use a model specifically tuned for code, such as Codestral [5]. In addition, better performance could be attained by fine-tuning the model on the Ericsson codebase. A RAG tool can partially overcome training limitations by ensuring access to current information, but the model will still make more accurate predictions if adapted [99].

The prototypes also used a single embedding model. A future implementation could identify task-specific embedding models. Beyond the model itself, the similarity measurement used by the embedding model can have an impact on performance. Furthermore, query expansion could be used to reformulate prompts and tool input—e.g., adding synonyms—to attain more relevant results [51].

The dataset contained situations where no test maintenance was required following a change to the source code. In general, the prototypes were worse at predicting that no change was needed than they were at correctly identifying cases where maintenance was needed. As discussed before, it may not be sufficient to use the triggers alone to make a prediction. Instead, the triggers could be augmented with code coverage or additional analyses of the changed code.

Manual inspection of false positives revealed multiple cases where test cases were marked as relevant by the prototype because they invoked the changed method in the source code but were not updated in the ground truth. These recommendations may not be "correct", but there may still be value in making such recommendations to users. It may be better to offer all "potentially relevant" tests than to omit test cases. This means that the prototypes may be more valuable to users than would be suggested by the F1 score alone.

Prototype Performance (RQ3): Our prototypes demonstrate that LLM agents can be used to predict the need for test maintenance. Prototypes that employed planning agents showed improved precision over pipeline architectures. Employing test summaries improved recall. However, the current performance of all prototypes is still too low to employ in practice. Performance could be improved through larger or specialized models, fine-tuning, alternative embedding models, and augmenting the triggers with additional contextual information.

Prototype Usefulness: In addition, we held an informal evaluation of the prototypes with two developers from Ericsson. Both were familiar with LLMs and the specific context of this study.

Both participants offered generally positive impressions. Although the prototypes did not fully "solve" test maintenance challenges, they were described as something that would be nice to have. They noted two main benefits. First, the prototypes run much faster than executing the tests—generally returning a result in under two minutes—which means they could be made aware of problems faster. The prototype may also be more reliable for some cases, as a test may still pass despite requiring maintenance. Second, they saw value in a "second opinion" beyond the intuition of the human reviewing the test cases. Although a human still made decisions, the prototypes offered guidance that could be factored into their decision, increasing their confidence.

Currently, to increase explainability, the prototypes issue output on the result of each step performed during the process of arriving at the final recommendation. The participants expressed that the output was too verbose, and that they would prefer a shorter trace coupled with a better explanation during the final output. They also requested suggestions on how to change the test cases. In addition, both participants stressed that the prototypes must be integrated with existing development tools, e.g., as an IDE plug-in, or at least easy to trigger from these tools.

Prototype Usefulness (RQ3): The prototypes were considered potentially useful, as they are faster to execute than the full test suite and offer additional input to human-driven test maintenance decisions. However, their output should not be too verbose, and they should be integrated into or easy to invoke from a developer's workflow.

6.4 AI Ethical Considerations

Pillars of Trustworthy AI: IBM has presented five “pillars”, or properties, that should be present for AI-based systems to be considered trustworthy [1]. We discuss below how the prototypes conforms to these pillars.

- *Explainability:* The prototypes provide the output of each step, as well as which fragments they pull using the RAG tool. The LLM instances issue their output, and the LLM agents record thoughts and decisions in an agent scratchpad. This provides a degree of explainability of the final output.
- *Fairness:* The prototypes do not process data related to humans. Therefore, fairness is not a concern at this time.
- *Robustness:* With the exception of utilizing a model internally deployed at Ericsson, no effort has been taken to minimize security risks. If the prototypes are further developed, care should be taken to ensure they are protected from adversarial attacks, data poisoning, and other interference.
- *Transparency:* The implementations of the prototypes cannot be made available as they contain proprietary elements. However, detailed descriptions of their architecture and all prompts are detailed in this publication. The Mistral 7B - Instruct model is open-weight, but the training data is not publicly available [6]. No additional data was used to fine-tune the model. The Nomic embedding model is open-source, including both training data and training code [83].
- *Privacy:* The prototypes require access to a source code repository, but do not use or collect any user data or data about humans. By using an internally deployed model, care has been taken to safeguard the information stored in the code repository.

Environmental Impact: LLMs training incurs a significant energy cost, especially as the size of models increases. For example, the training of GPT-3 is estimated to have consumed 1287 MWh [72]—approximately the same energy as 64 Swedish homes [7]. Energy continues to be consumed when models are prompted. For example, ChatGPT has an estimated power consumption of 564 MWh per day [26]. The models used in the prototype are much smaller (7B parameters versus 176B parameters, as is the case for GPT-3). However, employing a larger model would incur additional energy costs, as computational demands and time needed for training and prompting would increase. Care should be taken to balance LLM capabilities with potential environmental impact.

Legal Compliance: New regulations concerning AI-based software are under development. In particular, the Artificial Intelligence Act has been instated by the European Parliament [2]. Our research prototypes are largely not affected by such regulations, as they do not handle data on humans. The prototypes were deployed within the company, on local computational resources. This means that there is minimal risk of private data being exposed to malicious actors.

The prototypes are not directly implemented in high-risk applications, meaning that they are not directly impacted by the software quality requirements of such laws. However, Ericsson does develop safety-critical telecommunications systems. The use of LLM agents in test maintenance carries the risk of lowering the quality of such systems. If such agents are deployed in practice, care should be taken to evaluate how they affect the quality of the final product.

6.5 Threats to Validity

External Validity: We conducted a case study at a single company. Further, the evaluation of the prototype was conducted on a single project. However, we hypothesize that our conclusions would hold outside of Ericsson. The low-level triggers were derived from a literature review—where many of the sources also considered Java projects—and were validated by Ericsson employees. The high-level triggers are not closely tied to specific products or ways of working at the company. Similarly, the results of RQ2 largely emerge from a literature review on LLM capabilities and were validated within the company, suggesting that the findings of RQ2 are likely to generalize beyond the case subject. The architectures of the prototypes are also not closely coupled to Ericsson data or products, although some aspects of the implementation are. Therefore, the core ideas guiding the design of the prototypes and suggestions regarding improved performance should hold outside of our evaluation.

Internal Validity: Interviews and a survey were used to gather data for RQ1–2. Both data collection methods carry threats to validity. For surveys, limitations include a lack of control over the response rate as well as difficulties in understanding the reasoning or motivations behind answers to survey questions [37]. For both, we are drawing conclusions based on a limited sample size that may not be fully representative of the domain. Interview responses may also be affected by the biases of participants, and interpretation is affected by our biases. To mitigate these threats, we performed data triangulation—validating findings from the interview, survey, and literature review against the other data sources. Therefore, our findings are rooted in observations from multiple data sources [95].

For the evaluation of the prototypes, an assumption was made that co-evolution between test cases and source code held for each commit, i.e., that updates to test cases share a causal link to updates to source code in the same commit. This may not always be the case—changes to test cases may not be directly related to the changes in the source code in the same commit. Similarly, there may be cases where no test cases were changed in a commit, instead being changed in a later commit. In the dataset, these would be marked as "no test cases need maintenance", when—in reality—the changes came later. As a result, the performance of the prototypes in the evaluation may not accurately reflect real-world performance. However, we speculate that the dataset is sufficiently accurate to offer an approximate indication of the capabilities of the prototypes.

The dataset for the evaluation was split into different commits, which were divided into individual code changes. The prototypes are designed to return test case recommendations for a code change. In the dataset, ground truth is known for the commit, and not for each code change. To assess accuracy, we form the results for each code change into a full set of test cases for the commit, then compare that set to the ground truth. This could lead to some noise in the assessment of results, as we cannot judge the accuracy of a single code change, and instead operate at the commit-level.

Construct Validity: During the interviews and the survey, there was a risk of participants misunderstanding or misinterpreting terminology or questions. This risk was mitigated by running pilots of both interview and survey. The wording of some questions was clarified after this process in both instruments.

The prototypes have not undergone rigorous verification, and may contain faults. This threat was partially mitigated by using open-source functionality for LLM agents and RAG tools from the LangChain framework [22].

During the evaluation, some source and test code was excluded from the dataset (Section 4.7) due to limitations in the prototypes and the focus of our use case. These exclusions could introduce noise in evaluating the performance of the prototypes, e.g., test cases may be included in the ground truth that were connected to excluded source code, or vice-versa. However, we manually reviewed all exclusions to try to ensure that such cases do not exist.

7 Conclusions

In this study, we explore the capabilities and applications of Large Language Models (LLMs) to support the test maintenance process. We hypothesize that, broadly, LLM agents can perform or assist with a variety of test maintenance activities, including code generation or modification, assisting developer understanding, acting as a conversational assistant, and assisting with general development activities such as code review or documentation.

We identified 37 low-level changes to code and documentation that can trigger the need for test maintenance. However, such a need only occurs in situations where test cases exist that invoke the changed code, then cause and detect changes to the behavior of that code. Future work should augment triggers with additional contextual information (e.g., coverage or code quality analyses) to increase the accuracy and complexity of the actions taken by LLMs.

We also identified seven high-level development decisions that frequently trigger a need for test maintenance, including adding new features, requirements changes, fault discovery, increasing the coverage of a test suite, refactoring, tech stack changes, and test suite evaluation. LLM agents could act on such triggers by, e.g., providing examples or best practices, evaluating potential solutions, offering recommendations on how to improve potential solutions, modifying source code, or modifying the test suite.

When deploying LLMs, organizations should consider model performance and size, whether to deploy in-house or to rely on an externally-controlled model, potential copyright issues, how to tune the model on company data (e.g., retraining, fine tuning, or using a RAG tool), how to establish and manage company culture and community expectations, which tasks the LLM should support, and whether tasks should be fully automated or whether LLM output should instead be offered as a recommendation to a human developer.

Our prototypes demonstrate that LLM agents can be used to predict the need for test maintenance. Prototypes that employed planning agents showed improved precision over pipeline architectures. Employing test summaries improved recall. However, the current performance of all prototypes is still too low to employ in practice. Performance could be improved through larger or specialized models, fine-tuning, alternative embedding models, and augmenting the triggers with additional contextual information. The prototypes were still considered potentially useful, as they are faster to execute than the test suite and offer additional input to human-driven decisions. However, their output should not be too verbose, and they should be integrated into or easy to invoke from a developer's workflow.

Acknowledgments

Support for this research was provided by Software Center project 63 "AI-Enabled Test Automation, Generation, and Optimization".

References

[1] 2023. *IBM Artificial Intelligence Pillars.* https://www.ibm.com/policy/ibm-artificial-intelligence-pillars/

[2] 2024. *Artificial Intelligence Act: MEPs adopt landmark law.* https://www.europarl.europa.eu/news/en/press-room/20240308IPR19015/artificial-intelligence-act-meps-adopt-landmark-law

[3] 2024. *Code Generation on HumanEval.* https://paperswithcode.com/sota/code-generation-on-humaneval

[4] 2024. *Miro.* https://miro.com/

[5] 2024. *Mistral: Models.* https://docs.mistral.ai/getting-started/models/

[6] 2024. *Mistral: Open-weight models.* https://docs.mistral.ai/getting-started/open_weight_models/

[7] 2024. *Normal elförbrukning för villa & lägenhet - vattenfall.* https://www.vattenfall.se/fokus/tips-rad/vad-ar-normal-elforbrukning/

[8] Emil Alégroth, Robert Feldt, and Pirjo Kolström. 2016. Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. *Information and Software Technology* 73 (2016), 66–80.

[9] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of systems and software* 86, 8 (2013), 1978–2001.

[10] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525.

[11] Vahit Bayrı and Ece Demirel. 2023. AI-Powered Software Testing: The Impact of Large Language Models on Testing Methodologies. In *2023 4th International Informatics and Software Engineering Conference (IISEC)*. IEEE, 1–4.

[12] Lenz Belzner, Thomas Gabor, and Martin Wirsing. 2023. Large language model assisted software engineering: prospects, challenges, and a case study. In *International Conference on Bridging the Gap between AI and Reality*. Springer, 355–374.

[13] Lukas Berglund, Tim Grube, Gregory Gay, Francisco Gomes de Oliveira Neto, and Dimitrios Platis. 2023. Test maintenance for machine learning systems: A case study in the automotive industry. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 410–421.

[14] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2022. Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue* 20, 6 (2022), 35–57.

[15] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258* (2021).

[16] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[17] Lionel C Briand. 2007. A critical analysis of empirical research in software testing. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 1–8.

[18] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[19] Nikolaj Buhl. 2024. *F1 Score in Machine Learning*. https://encord.com/blog/f1-score-in-machine-learning/

[20] Cristina Rosa Camacho, Sabrina Marczak, and Daniela S Cruzes. 2016. Agile team members perceptions on non-functional testing: influencing factors from an empirical study. In *2016 11th international conference on availability, reliability and security (ARES)*. IEEE, 582–589.

[21] Yihan Cao, Siyu Li, Yixin Liu, Zhiling Yan, Yutong Dai, Philip S Yu, and Lichao Sun. 2023. A comprehensive survey of ai-generated content (aigc): A history of generative ai from gan to chatgpt. *arXiv preprint arXiv:2303.04226* (2023).

[22] Harrison Chase. 2022. LangChain. https://www.langchain.com/

[23] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[24] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128* (2023).

[25] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E Gonzalez, et al. 2024. Chatbot arena: An open platform for evaluating LLMs by human preference. *arXiv preprint arXiv:2403.04132* (2024).

[26] Alex de Vries. 2023. The growing energy footprint of artificial intelligence. *Joule* 7, 10 (2023), 2191–2194.

[27] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[28] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[29] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The faiss library. *arXiv preprint arXiv:2401.08281* (2024).

[30] Barrett Ens, Daniel Rea, Roiy Shpaner, Hadi Hemmati, James E Young, and Pourang Irani. 2014. Chronotwigger: A visual analytics tool for understanding source and test co-evolution. In *2014 Second IEEE Working Conference on Software Visualization*. IEEE, 117–126.

[31] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533* (2023).

[32] Robert Feldt, Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Towards autonomous testing agents via conversational large language models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1688–1693.

[33] Vahid Garousi and Michael Felderer. 2016. Developing, verifying, and maintaining high-quality automated test scripts. *IEEE Software* 33, 3 (2016), 68–75.

[34] Gregory Gay. 2023. Improving the Readability of Generated Tests Using GPT-4 and ChatGPT Code Interpreter. In *Search-Based Software Engineering: 15th International Symposium, SSBSE 2023, San Francisco, USA, December 8, 2023*. Springer.

[35] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. (2024).

[36] David Gewirtz. 2023. *Who owns the code? If ChatGPT's AI helps write your app, does it still belong to you?* https://www.zdnet.com/article/who-owns-the-code-if-chatgpts-ai-helps-write-your-app-does-it-still-belong-to-you/

[37] Ahmad Nauman Ghazi, Kai Petersen, Sri Sai Vijay Raj Reddy, and Harini Nekkanti. 2018. Survey research in software engineering: Problems and mitigation strategies. *IEEE Access* 7 (2018), 24703–24718.

[38] Danielle Gonzalez, Joanna CS Santos, Andrew Popovich, Mehdi Mirakhorli, and Mei Nagappan. 2017. 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). IEEE, 391–401.

[39] Roberto Gozalo-Brizuela and Eduardo C Garrido-Merchan. 2023. ChatGPT is not all you need. A State of the Art Review of large Generative AI models. *arXiv preprint arXiv:2301.04655* (2023).

[40] Giovanni Grano, Simone Scalabrino, Harald C Gall, and Rocco Oliveto. 2018. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of the 26th Conference on Program Comprehension*. 348–351.

[41] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680* (2024).

[42] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. 2023. Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects. *Authorea Preprints* (2023).

[43] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, et al. 2021. Pre-trained models: Past, present and future. *AI Open* 2 (2021), 225–250.

[44] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352* (2023).

[45] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620* (2023).

[46] Xing Hu, Zhuang Liu, Xin Xia, Zhongxin Liu, Tongtong Xu, and Xiaohu Yang. 2023. Identify and Update Test Cases When Production Code Changes: A Transformer-Based Approach. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1111–1122.

[47] Yuan Huang, Zhicao Tang, Xiangping Chen, and Xiaocong Zhou. 2024. Towards automatically identifying the co-change of production and test code. *Software Testing, Verification and Reliability* (2024), e1870.

[48] Ali Reza Ibrahimzada, Yang Chen, Ryan Rong, and Reyhaneh Jabbarvand. 2023. Automated Bug Generation in the era of Large Language Models. *arXiv preprint arXiv:2310.02407* (2023).

[49] Javaria Imtiaz, Salman Sherin, Muhammad Uzair Khan, and Muhammad Zohaib Iqbal. 2019. A systematic literature review of test breakage prevention and repair techniques. *Information and Software Technology* 113 (2019), 1–19.

[50] ISO/IEC/IEEE 24765:2017 2017. *ISO/IEC/IEEE International Standard - Systems and software engineering*. Technical Report ISO/IEC/IEEE 24765:2017. International Organization for Standardization.

[51] Rolf Jagerman, Honglei Zhuang, Zhen Qin, Xuanhui Wang, and Michael Bendersky. 2023. Query Expansion by Prompting Large Language Models. arXiv:2305.03653 [cs.IR] https://arxiv.org/abs/2305.03653

[52] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *Comput. Surveys* 55, 12 (2023), 1–38.

[53] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[54] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689* (2023).

[55] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[56] Mark Kasunic. 2005. Designing an effective survey.

[57] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. 2010. Software test automation in practice: empirical observations. *Advances in Software Engineering* 2010 (2010).

[58] Staffs Keele et al. 2007. Guidelines for performing systematic literature reviews in software engineering.

[59] Tenma Kitai, Hirohisa Aman, Sousuke Amasaki, Tomoyuki Yokogawa, and Minoru Kawahara. 2022. Have Java Production Methods Co-Evolved With Test Methods Properly?: A Fine-Grained Repository-Based Co-Evolution Analysis. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 120–124.

[60] Pavneet Singh Kochhar, Xin Xia, and David Lo. 2019. Practitioners' views on good software testing practices. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 61–70.

[61] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)*.

[62] Stanislav Levin and Amiram Yehudai. 2017. 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)*. IEEE, 35–46.

[63] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[64] Haodong Li, Gelei Deng, Yi Liu, Kailong Wang, Yuekang Li, Tianwei Zhang, Yang Liu, Guoai Xu, Guosheng Xu, and Haoyu Wang. 2024. Digger: Detecting Copyright Content Mis-usage in Large Language Model Training. arXiv:2401.00676 [cs.CR] https://arxiv.org/abs/2401.00676

[65] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models. *arXiv preprint arXiv:2308.00245* (2023).

[66] Junyou Li, Qin Zhang, Yangbin Yu, Qiang Fu, and Deheng Ye. 2024. More agents is all you need. *arXiv preprint arXiv:2402.05120* (2024).

[67] Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang. 2021. Traceability transformed: Generating more accurate links with pre-trained bert models. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 324–335.

[68] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[69] Lei Liu, Sinan Wang, Yepang Liu, Jinliang Deng, and Sicen Liu. 2023. Drift: Fine-Grained Prediction of the Co-Evolution of Production and Test Code via Machine Learning. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware*. 227–237.

[70] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1355–1367.

[71] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 647–658.

[72] Alexandra Sasha Luccioni, Sylvain Viguier, and Anne-Laure Ligozat. 2023. Estimating the carbon footprint of bloom, a 176b parameter language model. *Journal of Machine Learning Research* 24, 253 (2023), 1–15.

[73] Shreekant Mandvikar. 2023. Factors to Consider When Selecting a Large Language Model: A Comparative Analysis. *International Journal of Intelligent Automation and Computing* 6, 3 (2023), 37–40.

[74] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. 2014. Studying fine-grained co-evolution patterns of production and test code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 195–204.

[75] Mehdi Mirzaaghaei. 2011. Automatic test suite evolution. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 396–399.

[76] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezze. 2010. Automatically repairing test cases for evolving method declarations. In *2010 ieee international conference on software maintenance*. IEEE, 1–5.

[77] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. 2012. Supporting test suite evolution through test case adaptation. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 231–240.

[78] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. 2014. Automatic test case evolution. *Software Testing, Verification and Reliability* 24, 5 (2014), 386–411.

[79] Felix B Mueller, Rebekka Görge, Anna K Bernzen, Janna C Pirk, and Maximilian Poretschkin. 2024. LLMs and Memorization: On Quality and Specificity of Copyright Compliance. arXiv:2405.18492 [cs.CL] https://arxiv.org/abs/2405.18492

[80] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. 2004. *The art of software testing*. Vol. 2. Wiley Online Library.

[81] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an llm to help with code understanding. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 881–881.

[82] Anam Nazir and Ze Wang. 2023. A comprehensive survey of ChatGPT: Advancements, applications, prospects, and challenges. *Meta-radiology* (2023), 100022.

[83] Zach Nussbaum, John X Morris, Brandon Duderstadt, and Andriy Mulyar. 2024. Nomic Embed: Training a Reproducible Long Context Text Embedder. *arXiv preprint arXiv:2402.01613* (2024).

[84] OpenAI (2023). 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[85] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2016. On the diffusion of test smells in automatically generated test code: An empirical study. In *Proceedings of the 9th international workshop on search-based software testing*. 5–14.

[86] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants? (2023).

[87] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*. 1–11.

[88] Sunil Ramlochan. 2024. *Openness in Language Models: Open Source vs Open Weights vs Restricted Weights*. https://promptengineering.org/llm-open-source-vs-open-weights-vs-restricted-weights/

[89] Zeeshan Rasheed, Muhammad Waseem, Mika Saari, Kari Systä, and Pekka Abrahamsson. 2024. Codepori: Large scale model for autonomous software development by using multi-agents. *arXiv preprint arXiv:2402.01411* (2024).

[90] Pavel Reich and Walid Maalei. 2023. Testability Refactoring in Pull Requests: Patterns and Trends. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1508–1519.

[91] Xiaoxue Ren, Xinyuan Ye, Dehai Zhao, Zhenchang Xing, and Xiaohu Yang. 2023. From Misuse to Mastery: Enhancing Code Generation with Knowledge-Driven AI Chaining. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 976–987.

[92] Laria Reynolds and Kyle McDonell. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–7.

[93] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. 2023. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of the 28th International Conference on Intelligent User Interfaces*. 491–514.

[94] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[95] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14 (2009), 131–164.

[96] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213* (2022).

[97] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* (2023).

[98] Murray Shanahan. 2024. Talking about large language models. *Commun. ACM* 67, 2 (2024), 68–79.

[99] Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. 2024. Small llms are weak tool learners: A multi-llm agent. *arXiv preprint arXiv:2401.07324* (2024).

[100] Samiha Shimmi and Mona Rahimi. 2022. Patterns of Code-to-Test Co-evolution for Automated Test Suite Maintenance. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 116–127.

[101] Mohammed Latif Siddiq, Joanna CS Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using Large Language Models to Generate JUnit Tests: An Empirical Study. (2024).

[102] Mats Skoglund and Per Runeson. 2004. A case study on regression test suite maintenance in system evolution. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 438–442.

[103] Harry M Sneed. 2004. A cost model for software maintenance & evolution. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 264–273.

[104] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 23–30.

[105] Jeongju Sohn and Mike Papadakis. 2022. CEMENT: On the Use of Evolutionary Coupling Between Tests and Code Units. A Case Study on Fault Localization. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 133–144.

[106] Weifeng Sun, Meng Yan, Zhongxin Liu, Xin Xia, Yan Lei, and David Lo. 2023. Revisiting the Identification of the Co-evolution of Production and Test Code. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–37.

[107] Nigar M Shafiq Surameery and Mohammed Y Shakor. 2023. Use chat gpt to solve programming bugs. *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290* 3, 01 (2023), 17–22.

[108] Christos Tsigkanos, Pooja Rani, Sebastian Müller, and Timo Kehrer. 2023. Large language models: The next frontier for variable discovery within metamorphic testing?. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 678–682.

[109] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 4–15.

[110] Mubarak Albarka Umar and Chen Zhanfang. 2019. A study of automated software testing: Automation tools and frameworks. *International Journal of Computer Science Engineering (IJCSE)* 6 (2019), 217–225.

[111] Tanay Varshney. 2023. *Introduction to LLM Agents*. https://developer.nvidia.com/blog/introduction-to-llm-agents/

[112] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[113] László Vidács and Martin Pinzger. 2018. Co-evolution analysis of production and test code by learning association rules of changes. In *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 31–36.

[114] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).

[115] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science* 18, 6 (2024), 1–26.

[116] Sinan Wang, Ming Wen, Yepang Liu, Ying Wang, and Rongxin Wu. 2021. Understanding and facilitating the co-evolution of production and test code. In *2021 IEEE International conference on software analysis, evolution and reengineering (SANER)*. IEEE, 272–283.

[117] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[118] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839* (2023).

[119] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155* (2023).

[120] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864* (2023).

[121] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*.

[122] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629* (2022).

[123] Juyeon Yoon, Robert Feldt, and Shin Yoo. 2023. Autonomous Large Language Model Agents Enabling Intent-Driven Mobile GUI Testing. *arXiv preprint arXiv:2311.08649* (2023).

[124] Shengcheng Yu, Chunrong Fang, Yuchen Ling, Chentian Wu, and Zhenyu Chen. 2023. LLM for test script generation and migration: Challenges, capabilities, and opportunities. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, 206–217.

[125] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. *arXiv preprint arXiv:2305.04207* (2023).

[126] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. 2023. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–69.

[127] Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A Survey on Large Language Models for Software Engineering. *arXiv preprint arXiv:2312.15223* (2023).

[128] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2023. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989* (2023).

[129] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223* (2023).

[130] Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. 2023. Towards an understanding of large language models in software engineering tasks. *arXiv preprint arXiv:2308.11396* (2023).

[131] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: Evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372* (2023).

[132] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910* (2022).

[133] Jianfei Zhu, Guanping Xiao, Zheng Zheng, and Yulei Sui. 2022. Enhancing traceability link recovery with unlabeled data. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 446–457.

[134] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2020. A comprehensive survey on transfer learning. *Proc. IEEE* 109, 1 (2020), 43–76.

[135] Daniel Zimmermann and Anne Koziolek. 2023. Automating gui-based software testing with gpt-3. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 62–65.

Received 31 August 2024; revised N/A; accepted N/A