Exploring the integration of large language models in industrial test maintenance processes

The Journal of Systems and Software 242 (2026) 113049

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Exploring the integration of large language models in industrial test maintenance processes☆

Jingxiong Liu [a,b], Ludvig Lemner [a,b], Linnea Wahlgren [a,b], Gregory Gay [a,*], Nasser Mohammadiha [b,a], Joakim Wennerberg [b]

[a] Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden
[b] Ericsson AB, Gothenburg, Sweden

ARTICLE INFO

Keywords:
Software testing
Test maintenance
Machine learning
Large language models
LLM agent

ABSTRACT

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 explore 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 propose and demonstrate a multi-agent architecture that can predict which tests require maintenance following a change to the source code. Collectively, these contributions advance our understanding of how LLMs could be deployed to benefit industrial test maintenance processes.

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 (Alégroth et al., 2016).

Although there is an initial cost associated with creating test cases, much more cost is imposed by the need for *test maintenance* as the system-under-test evolves (Sneed, 2004). During test maintenance, new tests are created and obsolete tests are modified or removed to adapt to changes in the evolving source code (Alégroth et al., 2016). Test maintenance can also be conducted to improve the quality or efficiency of the test suite—for example, to improve test coverage (Sneed, 2004).

As an example, consider a tester who is responsible for a test suite containing 300 code-based unit and integration tests for a standalone web application. As illustrated in Fig. 1, certain events may occur that induce the need for test maintenance. We refer to such events as *triggers*. For example, if the source code of the project is updated, than certain test cases may need to be updated to reflect the changes to the underlying functionality-under-test. Alternatively, the tester may notice that the time needed to execute the test suite is too long, suggesting the need for a reduction in the size of the suite. Following an analysis of the trigger, the source code, and the test suite — as well as other artifacts, if needed — the tester then takes *actions* to maintain the test suite. For the first trigger — a change to the source code — the tester might update or remove obsolete test cases. To address the second trigger, the tester may identify tests that are redundant in terms of the code or behaviors tested and remove them to improve the execution time.

Test maintenance requires long-term commitment of significant effort from developers (Sneed, 2004; Wang et al., 2021). Software tools have a role to play in reducing the cost and improving the quality or reliability of the test maintenance process (Wang et al., 2021; Umar and Zhanfang, 2019). In particular, we focus on two forms of tool support. First, we examine cases where tools could automate actions 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 (Kochhar et al., 2019), test maintenance is an under-explored and poorly understood aspect of the overall testing process (Gonzalez et al., 2017; Imtiaz et al., 2019). There has been

☆ Editor: Wesley Klewerton Guez Assunção.

* Corresponding author.
E-mail address: greg@greggay.com (G. Gay).

https://doi.org/10.1016/j.jss.2026.113049
Received 3 July 2025; Received in revised form 10 June 2026; Accepted 20 July 2026
Available online 25 July 2026

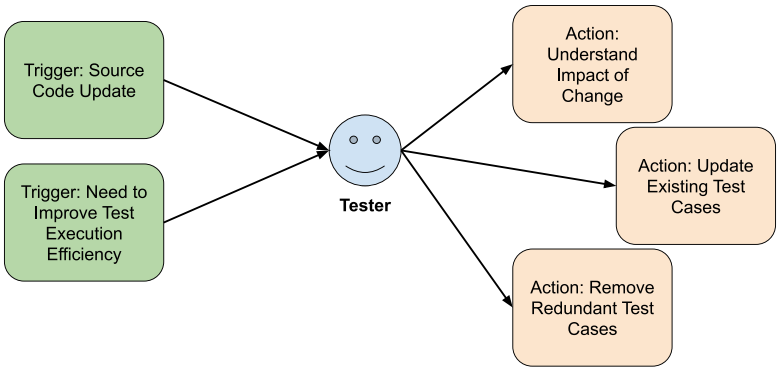


Fig. 1. Motivational example showing potential triggers and actions.

relatively little research on test maintenance automation (Hu et al., 2023).

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 (Zhao et al., 2023). These capabilities extend to both natural language and software code (Zheng et al., 2023b). As a result, LLMs have shown promise in automating or assisting with software engineering (Q. Zhang et al., 2023) and software testing (Wang et al., 2024) tasks, including test generation (Schäfer et al., 2023), documentation (Hadi et al., 2023), refactoring (White et al., 2023), and automated program repair (Sobania et al., 2023). LLMs have shown potential in both forms of tools support mentioned above, including automation (White et al., 2023; Surameery and Shakor, 2023) and serving as a conversational assistant to developers (Ross et al., 2023).

Although there has been significant interest in the general use of LLMs in software testing (Wang et al., 2024), the applications of LLMs in the test maintenance process have only begun to be explored (Hu et al., 2023; Chi et al., 2025; Liu et al., 2024b). 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 (Feldt et al., 2023) — into this process. We explore this topic through an industrial case study at Ericsson AB, a Swedish telecommunications company, with the aim of offering both exploratory and practical contributions to the field. Although we take a broad view of the problem of test maintenance, in terms of scoping, we primarily focus on maintenance of code-based test cases at the unit, integration, and system-level.

The exploratory contributions of this work consist of an investigation of (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 potential practical applicability of LLM agents in test maintenance, we also propose a multi-agent architecture that predicts which test cases require maintenance following a change to the source code. We evaluate the performance of two implementations of this architecture on an industrial dataset, with additional qualitative analysis of false positives. This prototype offers a starting point for future implementation of LLM agents in test maintenance. These contributions have potential benefit to both researchers and practitioners, advancing our understanding of how LLMs and LLM agents could be applied within test maintenance and offering practical guidance to industrial developers.

The structure of this paper is as follows. In Section 2, we explain core background concepts. In Section 3, we explore related work. Section 4 presents our exploratory study on triggers, actions, and industrial considerations. Then, based on this study, in Section 5, we present and evaluate our proof-of-concept implementation. In Section 6, we discuss threats to validity. Finally, in Section 8, we offer conclusions.

2. Background

2.1. Software testing

Software testing is an activity performed to ensure that software delivers correct functionality and meets quality goals (e.g., performance) (ISO/IEC/IEEE 24765:2017, 2017). During testing, a *test suite* — one or more *test cases* — is executed against the system-under-test (ISO/IEC/IEEE 24765:2017, 2017). Each test case performs input actions, then compares the resulting program behavior against expectations (*test oracles* (Barr et al., 2014)).

Software testing is an important, but notoriously expensive, activity (Myers et al., 2004), especially due to the significant manual effort involved in test case creation or manual test execution (Wang et al., 2021). Automation has a role to play in reducing the cost by, e.g., enabling automated execution (Umar and Zhanfang, 2019). In particular, significant research efforts have been invested in automating the generation of test input (Anand et al., 2013), including recent investigations of the capabilities of LLMs (Wang et al., 2024; Yuan et al., 2023).

Test maintenance is the process of updating the test suite as the source code evolves—including, e.g., adding new tests for new functionality, removing tests that are no longer relevant, and adapting tests to ensure their continued relevance (Alégroth et al., 2016). 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 (Sneed, 2004). Test maintenance is understood to be important, and can account for a significant proportion of testing effort over the lifespan of a project (Alégroth et al., 2016; Sneed, 2004). However, the area has received comparatively little research attention (Skoglund and Runeson, 2004). For example, “test smells” negatively impact test maintainability, but are significantly less well-understood than code smells (Tufano et al., 2016).

2.2. Generative artificial intelligence and large-language models

Generative AI refers to systems, typically leveraging machine learning, that create content, e.g. text or images, in response to instructions delivered as a *prompt* (Cao et al., 2023). Prompts can include text and other media. Models are trained to infer the semantic meaning of a prompt, and use that meaning to produce an appropriate response.

Current Generative AI models are implemented using a transformer architecture, which is based on mechanisms designed to imitate the cognitive attention (i.e., the ability to focus on select stimuli) seen in humans (Vaswani et al., 2017). *Foundation models* are trained on large and diverse datasets with the intention that they perform at a minimal level on new tasks (Han et al., 2021). These models can then be *fine-tuned* using additional domain-specific training data (Bommasani et al., 2021).

Large language models (LLMs) are a form of Generative AI suited for language analysis and transformation tasks such as translation, summarization, and decision support (OpenAI (2023), 2023). LLMs iteratively predict the next element to add to the generated output (Zhao et al., 2023). 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 (Wang et al., 2024; Ross et al., 2023; Hou et al., 2023; Z. Zhang et al., 2023).

An LLM and a human may not draw the same meaning from a prompt, which has led to the development of different prompting strategies, including *few-shot prompting* — where input–output examples are supplied along with the prompt (Brown et al., 2020) — and *chain-of-thought prompting*—where examples are augmented with reasoning explaining how the correct output was reached (Wei et al., 2022).

LLMs are known to *hallucinate* at times, issuing output that appears initially plausible, but is incorrect (Ji et al., 2023). Additionally, there is the risk of *data leakage*, where the LLM may have seen a benchmark during training, leading to artificially high performance on that benchmark (Wang et al., 2024).

An *LLM agent* is a system coupling an LLM with tool access and memory capabilities (Feldt et al., 2023). These mechanisms allow LLM agents to reason, plan, and perceive or interact with an environment (Feldt et al., 2023). For example, an LLM agent may have access to a version control system or issue tracker through *Retrieval-Augmented Generation* (RAG) (Lewis et al., 2020). This would allow it to reason about the code, make changes to it, and push those changes. Multiple agents can work together, each given different roles or sub-tasks (Wu et al., 2023).

3. Related work

Maintainability is one of the characteristics of good test cases most prized by testers (Kochhar et al., 2019). However, in practice, few open-source projects implement tests following patterns that promote maintainability (Gonzalez et al., 2017). Imtiaz et al. also noted a need for studies on test maintenance in industry (Imtiaz et al., 2019).

Multiple authors have examined factors that influence the probability, frequency, and difficulty of test maintenance. For example, Pinto et al. found that many of the tests "added" to suites are older tests that have been updated and renamed, that tests are generally deleted because they are obsolete and not because they are difficult to update, and that when tests are added, it is to cover new functionality or to validate bug fixes and refactored code (Pinto et al., 2012). Alégroth et al. identified thirteen factors that affect the difficulty of test maintenance for GUI-based test suites (Alégroth et al., 2016), with test length having the greatest impact. Berglund et al. also identified factors that affect test maintenance for machine learning systems — e.g., non-determinism and explainability of the system-under-test — as well as factors that affect both traditional and machine learning systems—e.g., the required precision of the test oracle and consistency between teams (Berglund et al., 2023).

Researchers have examined co-evolution of test and source code, finding that changes to both source and test code within a short timeframe (Sun et al., 2023) or shared naming conventions (Hu et al., 2023; Chi et al., 2025) often indicate co-evolution. Metrics such as Tconf can also assess the extent source and test code have successfully co-evolved (Kitai et al., 2022). Visualizations can also show how source and test code have evolved (Ens et al., 2014). Traceability links can also be inferred between source and test code based on co-evolution (Sohn and Papadakis, 2022).

Many researchers have worked to identify changes to source code that trigger a need for maintenance. We draw on this literature as part of answering our first research question, and discuss the studies in Section 4.2.1. Such triggers have been used in past research to identify tests that need maintenance, based on supervised learning models (Wang et al., 2021) or by combining supervised learning with additional data sources, such as static analysis (L. Liu et al., 2023) or code complexity metrics (Huang et al., 2024).

Researchers have also developed approaches to automatically repair tests following source code changes. Mirzaaghaei et al.'s approach is based on changes to method parameters and return values (Mirzaaghaei et al., 2010; Mirzaaghaei, 2011; Mirzaaghaei et al., 2014), while Hu et al. infer edit patterns from datasets of source and test code changes (Hu et al., 2023).

LLM agents have recently gained interest more broadly in software testing. Feldt et al. developed a taxonomy of agents and outlined an example conversational testing agent (Feldt et al., 2023). Yoon et al. have also demonstrated how LLM agents can perform Android GUI testing (Yoon et al., 2023). 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 (Rasheed et al., 2024).

LLMs have recently been applied to either identification or repair of obsolete tests. Hu et al. applied a transformer model to both identify and update tests (Hu et al., 2023). They do not separate these tasks into individual predictions, but perform a single repair task. If the final test is identical to the input, they assume that the test has been determined to still be relevant. Their approach is limited to production changes contained within a single method. Recently, Chi et al. proposed a multi-agent framework for test identification (Chi et al., 2025). Their approach is conceptually similar to our own, employing a RAG-based memory mechanism containing natural language summaries of reasons why previous test updates took place. This is implemented in a similar manner to our analysis, but focusing on different contextual information. A limitation of their approach is that it assumes that the set of all tests possibly affected by a production code change is already known, while our approach infers this relationship, and their evaluation dataset has a one-to-one mapping between production and test code changes. Neither approach considers commits where no test maintenance took place at all following a production change, only situations where particular individual test cases were not updated. In addition, Liu et al. use LLMs to update tests, but do not consider the identification task (Liu et al., 2024b).

Our study is influenced by and expands on prior work. In terms of our exploratory contributions, we expand the range of triggers considered beyond past research, and we broadly explore the theoretical applications of automation in test maintenance—extending beyond prediction and update of individual obsolete test cases. In terms of our practical contributions, we offer a multi-agent architecture for prediction of obsolete test cases. Our proof-of-concept offers novel extensions over past approaches to test identification. First, we examine the use of natural language summaries of test semantic meaning as a way to identify relevant test cases in addition to raw test code. Second, our approach is designed to be deployed in a realistic development environment, which required overcoming specific limitations of past work. Unlike, for example, Hu et al. (2023), our implementation takes a `git diff` as input — a standard method of summarizing code changes — enabling direct invocation following a commit. While Hu et al. (2023) was limited to analyzing changes to a single production method, our approach can process repository-level changes. Previous approaches, e.g., Chi et al. (2025), also assumed a precise, pre-existing mapping of production and test artifacts, while our prototype infers that relationship. These offer novel extensions over the state-of-the-art that are necessary to ensure the direct applicability of our tool for its intended use case. We have also performed, to our knowledge, the first exploration and evaluation of LLM agents for test maintenance in an industrial setting and incorporate the experience and opinions of practitioners as an important source of data in our analysis.

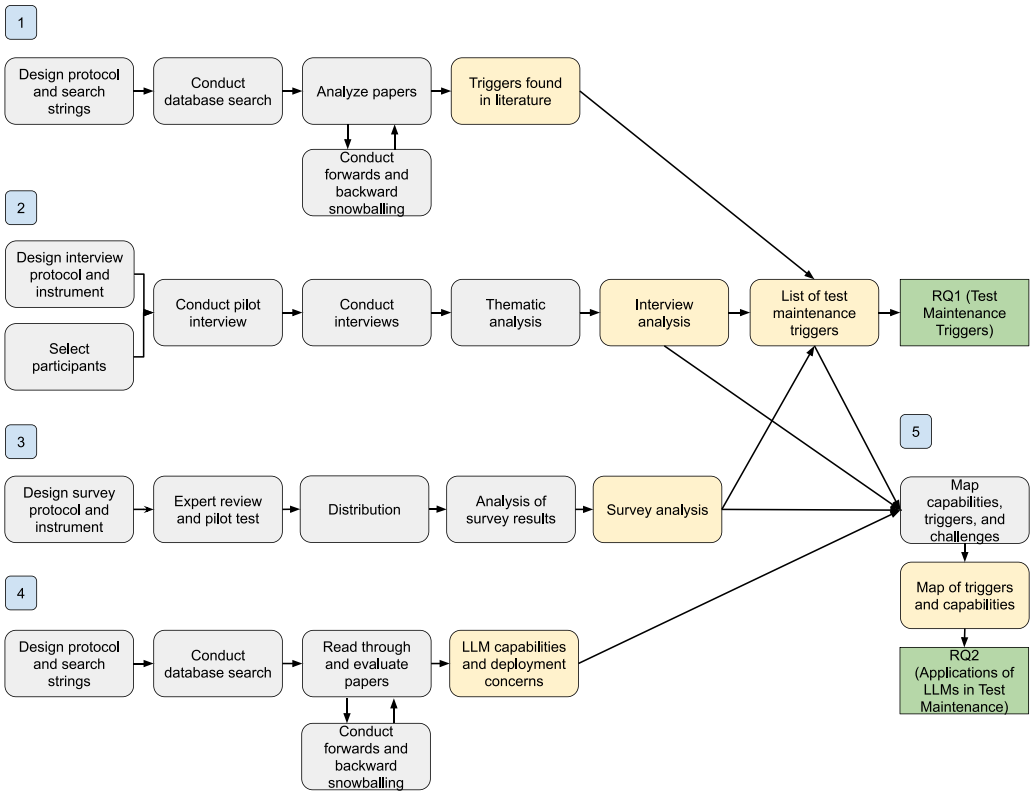


Fig. 2. Overview of the case study. The numbers correspond to the steps outlined in Section 4.1. Gray = activity, yellow = artifact, green = research question. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4. Exploratory study

This section presents our exploratory study, where we used literature review, interviews, and a survey to explore the potential applications of LLM-based support for industrial test maintenance processes. In particular, we explore the *triggers* of maintenance, potential *actions* that can be taken in response to the triggers, and *considerations* for industrial deployment of LLMs in the maintenance process. Section 4.1 outlines our research questions and the methods used to answer them. Section 4.2 offers the results of our research activities. Finally, in Section 4.3, we answer the research questions.

4.1. Methods

The **goal** of this study is to *understand how LLMs or LLM agents can support an industrial test maintenance process*. We operationalize this goal into the following **research questions**:

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

RQ2 What applications could 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?

We address these questions through a case study at Ericsson AB, a major Swedish telecommunications company, following the guidelines from Runeson and Höst (Runeson and Höst, 2009). Our analysis is based on the following **data**:

- **Qualitative Data:** Thematic analysis of literature, interview responses, survey responses.
- **Quantitative Data:** Descriptive statistics based on survey responses.

This data was gathered through the following **activities** (shown in Fig. 2):

1. We performed a **literature review to identify test maintenance triggers** (Section 4.1.1). Data extracted from the identified publications partially address **RQ1**.
2. We conducted **interviews** to understand the test maintenance process, triggers, and challenges at Ericsson, as well as potential tool support (Section 4.1.2). Thematic analysis contributed to addressing **RQ1–2**.
3. To gather additional data, we conducted a **survey** at Ericsson regarding the test maintenance process, triggers, and challenges, as well as opinions regarding LLMs and tool support (Section 4.1.3). Analysis of responses contributed to addressing **RQ1–2**.
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 and industrial deployment (Section 4.1.4). 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 4.1.5). This process contributed to addressing **RQ2**.

4.1.1. Literature review to identify test maintenance triggers (RQ1)

We conducted a literature review to identify changes to project artifacts (e.g., to code, requirements, or other artifacts) or to the development process that may trigger a need for test maintenance. Our review was inspired by Keele's guidelines for systematic literature reviews (Keele et al., 2007). We did not attempt to capture all research in this field. Rather, our aim was to obtain a diverse representative sample of studies that could be used to characterize past research. To identify studies, we applied the following search strings to ACM, IEEE, Science Direct, and Scopus:

- ("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 was developed iteratively, with the intent of capturing relevant research on test maintenance while filtering publications that do not discuss potential triggers. The second adds a filter for 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 — 2009–2024 — to ensure relevance to modern software development.
- 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 factors that could trigger test maintenance.

48 publications were found to potentially meet all criteria. They were then read in full. This process was conducted by two of the authors. Each paper was assessed by one author, where the assessment was based on whether the publication explicitly discussed triggers of test maintenance. Eight publications discussed test maintenance triggers and were retained for analysis. We performed backward and forward snowballing on these, yielding 64 additional potentially-relevant publications. From these, four additional publications were retained because they explicitly discussed maintenance triggers.

We extracted triggers from these 12 publications. This process was performed by the two authors, with both reading each of these publications and extracting triggers. The authors then collaboratively clustered the identified triggers based on commonalities, such as belonging to particular levels of granularity within the code. They also merged redundant triggers. The authors discussed and resolved any disagreements during this stage.

4.1.2. 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: We targeted Ericsson employees with experience in testing. 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 interviews, with eight participants. The interviews were conducted in the same manner regardless of the number of participants present.

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

Table 1 presents the participant demographics. The most common role was "developer", followed by "test manager". The developers tested at the unit level, while test managers focused on integration and system level. The participants had a median of 4.25 (average of 7.69) years of experience.

Interview Instrument: We conducted semi-structured interviews (instrument in Table 2), so we could ask follow-up questions to gain further insight. Sessions were conducted online and were recorded for transcription. On average, each interview lasted 40 min.

We performed a pilot interview, which led to removal of an irrelevant question and minor clarifications. Because changes were minor, the pilot interview was used in the final analysis.

Interview Analysis: Interviews were first transcribed automatically, then manually corrected. We then performed thematic analysis following the guidelines described by Braun and Clarke (2006). This process was completed by the second and third authors, with validation by the other authors.

We highlighted relevant parts of the transcripts and assigned "code labels" — short identifiers — to each. We then developed "codes" describing each highlighted segment. Coding was inductive; we did not begin with any pre-decided codes. Rather, we allowed the raw data to guide our analysis. The initial coding was done individually, with both researchers coding each interview separately. We then had multiple discussions and iterations of the codes and code labels, where we merged the individual work and resolved disagreements. All disagreements were minor, e.g., small differences in the segments highlighted. These disagreements were resolved through discussion, where the researchers collaboratively decided on the final version to use. Because the differences were minor, we did not formally assess inter-rater agreement.

Then, we iteratively grouped the merged set of codes and labels into themes and sub-themes. We conducted the process visually during multiple meetings where the researchers collaboratively performed the clustering using the Miro software. Clustering was conducted after each interview, with the new codes and labels used to refine the existing map. We judged that we had reached saturation after analyzing the transcripts and finding that later interviews yielded few new findings.

4.1.3. Survey (RQ1–2)

As interviews require significant effort, we also conducted a survey to gain further insight into the test maintenance process, triggers, and

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 creative purposes (e.g., providing advice) and boilerplate purposes (e.g., generating code). Do you have a preference for how you would use them?

challenges, as well as opinions regarding LLMs and tool support. The survey instrument was designed based on the thematic analysis of the interview results as well as the literature review on test maintenance triggers.

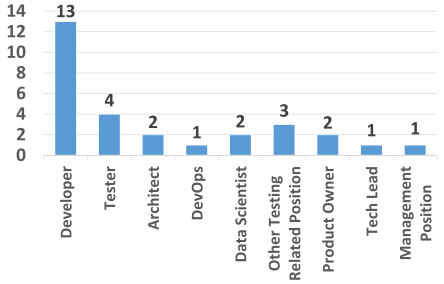
Participant Selection: Our audience was again Ericsson employees with testing experience, targeted using convenience sampling with purposive elements. The interview participants were asked not to also participate in the survey. We initially distributed the survey to a developer community, consisting 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 communities. The survey was open for 54 days.

29 participants completed the survey. Their demographics of are shown in Fig. 3. The most common role was developer (45%), followed by testing-related positions (24%). Participants had a median of 3–5 years of testing experience, and tested most often at the unit 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 interview and survey responses are complementary—i.e., because the two populations are highly similar, we hypothesize that we can use the survey answers to validate or identify discrepancies in the interview responses and vice-versa.

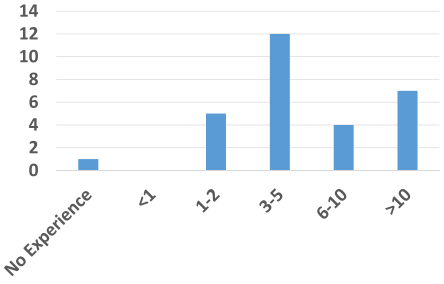
Survey Instrument: The survey instrument (Table 3) was designed based on guidelines from Ghazi et al. (2018) and Kasunic (2005). It was designed to take 5–10 min to ensure a reasonable completion rate. The intention was to provide quantitative data to augment the qualitative interview. Thus, questions were multiple-choice, rather than open-ended. Respondents could provide multiple answers to some questions, but the number of options they could select was limited to force prioritization. The survey was designed prior to analysis of interview data.

We evaluated question wording using criteria established by Kasunic (2005) that offer rules for 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.

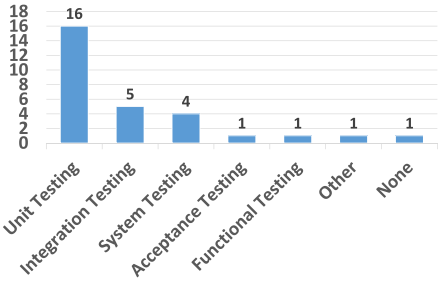
Survey Analysis: The results of the survey were analyzed using descriptive statistics, and were contextualized by the thematic analysis of the qualitative data obtained during the interviews.



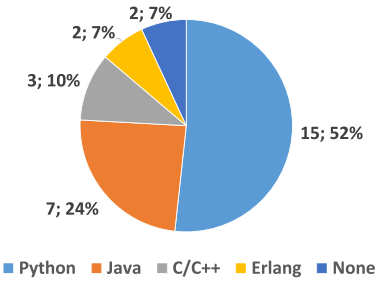
(a) Number of participants in each work role.



(b) Years of experience with software testing.



(c) Most common testing level.



(d) Most common programming language.

Fig. 3. Demographics of survey respondents.

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

4.1.4. Literature review on LLM capabilities for test maintenance (RQ2)

To understand how LLMs could potentially assist with test maintenance and considerations for industrial deployment, we conducted a literature review focusing on how LLMs have been applied within software testing. This review followed a similar procedure to the review in Section 4.1.1. We applied the following search strings to ACM, IEEE, Science Direct, and Scopus:

- (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 software testing. The third to LLMs performing tasks involving source code. The fourth expands the applied synonyms to capture additional publications.

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 LLM-related developments emerged — 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.

We identified 67 publications for further examination. From this set, 10 were relevant for addressing RQ2. We performed forward and backward snowballing on these, which added an additional 92 publications. From those, two were deemed relevant, yielding 12 publications in total. This process was again conducted by two of the researchers, with each independently assessing the relevance of publications.

Both researchers read the remaining 12 publications, extracting qualitative data that explained how LLMs were applied, what their strengths and limitations were, what results were obtained, and what considerations may affect their deployment in an industrial setting. This data includes quotes and observations of the researchers. This data was then collaboratively clustered based on commonalities into themes and sub-themes, with any disagreements resolved through discussion.

4.1.5. Mapping maintenance triggers and tasks with LLM capabilities (RQ2)

To identify how LLMs or LLM agents could assist with test maintenance, we compared data gathered from the literature review on test maintenance triggers, the interviews, and the survey. We mapped this data to the capabilities of LLMs demonstrated in previous literature.

To decide which specific maintenance tasks LLMs or agents could perform, 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 test maintenance tasks and challenges identified in the interviews and survey. We also considered how LLMs should be implemented to match developers' preferences and ways of working.

4.2. Results

In this section, we present result and observations, which will be synthesized into answers to the research questions in Section 4.3.

4.2.1. Literature review to identify test maintenance triggers (RQ1)

We define the test maintenance triggers that we identified from literature in Tables 4–7. Broadly, the triggers can be divided *documentation changes* — e.g., comments, user manuals, or requirements — and *source code changes*. Code changes can also be grouped based on the level of granularity — functionality, class-level, method-level, and line-level.[1]

4.2.2. Thematic analysis of interviews (RQ1–2)

An overview of the high-level themes identified from the interviews is presented in Table 8, and we discuss each below. We also indicate the number of codes corresponding to each theme or sub-theme, as well as the percentage of total codes (for all themes, or for all sub-themes of a particular theme).[2] All themes and sub-themes are depicted visually in Fig. 4, where the font size also indicates the prevalence of that theme or sub-theme in the codes. The number of codes associated with each theme are listed in Table 8 and for each sub-theme in the tables for each respective theme.

Theme: Triggers. The primary test maintenance triggers discussed in interviews are summarized as sub-themes in Table 9.

Production Code: Naturally, the most prominent reason for test maintenance is a change in the source code. Participants, in particular, described test maintenance occurring after new feature implementation, changes in requirements, modifications to code methods, and changes to conditional statements—all of which were also derived in Section 4.2.1.

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

> "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. Refactoring at a small scale — e.g., to improve performance of a function — should ideally not require test maintenance. However, large-scale refactoring, such as a change to the core architecture, may require maintenance, as methods and classes may no longer exist or there may be interface changes.

Bug: Naturally, if a bug is discovered, the source code will evolve. This 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, libraries, testing frameworks, or build systems — can also trigger test maintenance. An unstable tech stack can intensify test maintenance, and may constrain the forms and quantity of testing performed.

Testability: At times, the source code is modified to improve its testability. The test suite must be adapted to these changes—e.g., new forms of testing may now be possible. For example, one participant discussed making components more modular or mockable:

[1] There is some overlap in triggers — e.g., "addition of functionality" could include "addition of a class" or "addition of a method". However, we retain "functionality" to capture cases where it was not clear what form the functionality took.

[2] A high number does not necessarily indicate higher importance, but does indicate that participants had more to say about the subject.

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

Type of change	Description	Sources
Changes to functionality	**Code changes that affect system behavior.**	
Add functionality	Implementing new functionality. This is a general trigger for cases where it is not clear how the functionality was implemented.	Mirzaaghaei (2011), Shimmi and Rahimi (2022) and Levin and Yehudai (2017)
Add interface	Implementing a new interface.	Mirzaaghaei (2011), Mirzaaghaei et al. (2014), Levin and Yehudai (2017) and Mirzaaghaei et al. (2012)
Error and exception handling	Changes to code that handles errors/exceptions, such as `throw`, `try`, and `catch` keywords.	L. Liu et al. (2023)
Parallelism and concurrency	Changes to how the system handles parallel threads and concurrency, such as the `synchronised` keyword.	L. Liu et al. (2023)
Remove functionality	Code is removed. This is a general trigger for cases where it is not clear how the functionality was removed.	Shimmi and Rahimi (2022) and Levin and Yehudai (2017)
Update API	Changes to how an external API is invoked.	Hu et al. (2023)

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	Implementing a new class.	L. Liu et al. (2023), Levin and Yehudai (2017), Mirzaaghaei et al. (2012), Marsavina et al. (2014), Reich and Maalei (2023) and Vidács and Pinzger (2018)
Class declaration	Changes made to a class declaration, including extending class hierarchy.	Wang et al. (2021), Mirzaaghaei (2011), Mirzaaghaei et al. (2014), Levin and Yehudai (2017), Marsavina et al. (2014) and Vidács and Pinzger (2018)
Constructor	Changes made to the class constructor, e.g., creating a constructor or adding a parameter.	Reich and Maalei (2023)
Fields	Changes made to the fields of a class.	Wang et al. (2021), Marsavina et al. (2014) and Vidács and Pinzger (2018)
Remove class	Removing a class that previously existed.	Levin and Yehudai (2017), Marsavina et al. (2014) and Vidács and Pinzger (2018)

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	Implementation of a new method. This includes adding an overloaded or overridden method.	Mirzaaghaei (2011), Mirzaaghaei et al. (2014, 2012), Reich and Maalei (2023) and Vidács and Pinzger (2018)
Method declaration	Changes to a method declaration or signature. This includes changing access level, adding or removing `final/static` keyword, changing value or type of the return, and adding, removing and changing a parameter type.	Wang et al. (2021), L. Liu et al. (2023), Mirzaaghaei et al. (2010), Mirzaaghaei (2011), Mirzaaghaei et al. (2014), Levin and Yehudai (2017), Mirzaaghaei et al. (2012), Marsavina et al. (2014), Reich and Maalei (2023) and Vidács and Pinzger (2018)

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

Type of change	Description	Sources
Changes to LOC	**Changes made to the source code that affect a single line.**	
Arrays	Changes to arrays, e.g., new array or changing access level.	L. Liu et al. (2023)
Assignments	Changes to value assigned or compound assignments.	L. Liu et al. (2023)
Attribute	Declaration of attributes and extraction for assertion.	Marsavina et al. (2014), Reich and Maalei (2023) and Vidács and Pinzger (2018)
Modifiers	Changing and updating modifiers, e.g., `public` keyword.	Hu et al. (2023) and L. Liu et al. (2023)
Identifier	Changes to identifiers, e.g., variable or package names.	Wang et al. (2021), Hu et al. (2023) and L. Liu et al. (2023)
Object state	Removing or adding object state.	Levin and Yehudai (2017)
Override system time	Overriding the system time.	Reich and Maalei (2023)
Statement	Changes to control-flow, conditional, or loop statements	Wang et al. (2021), L. Liu et al. (2023), Levin and Yehudai (2017) and Marsavina et al. (2014)
Type	Modify data type, either keyword or by casting.	L. Liu et al. (2023)
Variables	Changes to the declaration of a variable.	L. Liu et al. (2023)

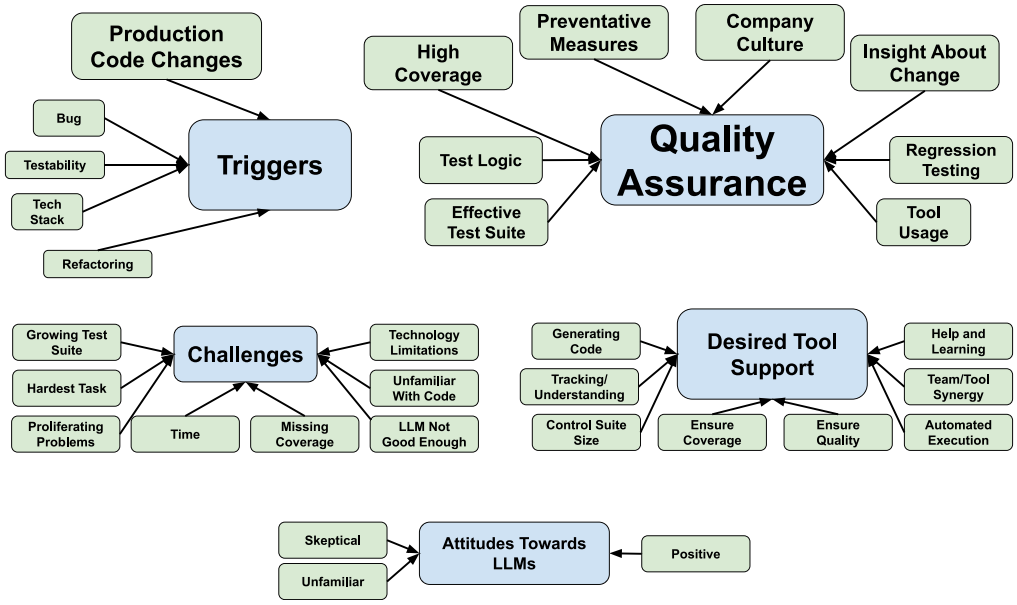


Fig. 4. A visual overview of all interview themes (blue) and sub-themes (green), where the font size reflects the prevalence of codes that correspond to that theme or sub-theme. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 8
Overview of interview themes, with description and number of codes (and percentage of total) that correspond to that theme.

Theme	Description	Num (%)
Triggers	Reasons for performing test maintenance.	129 (25%)
Quality Assurance	Ways to ensure that test cases and suites hold high quality.	212 (41%)
Challenges	Current issues experienced with test maintenance.	76 (15%)
Desired Tool Support	Type of automated help wanted with test maintenance.	69 (13%)
Attitudes Towards LLMs	Attitudes about using LLMs for test maintenance.	28 (6%)

Table 9
Overview of sub-themes of "Triggers", with description and number of codes (and percentage of total for that theme) that correspond to that sub-theme.

Sub-theme	Description	Num (%)
Production	Changes on the source side cause changes in tests.	88 (68%)
Refactoring	Should not change functionality, but can still induce test changes.	15 (12%)
Bug	The discovery of a bug leads to changes in tests.	12 (9%)
Tech stack	Changes in the tech stack necessitate changes to tests.	8 (6%)
Testability	Test and source code are changed to be easier to test.	6 (5%)

Table 10
Overview of sub-themes of "Quality Assurance", with description and number of codes (and percentage of total for that theme) that correspond to that sub-theme.

Sub-theme	Description	Num (%)
High coverage	Execute all code to ensure thorough testing.	46 (22%)
Preventative measures	Finding faults and stopping them from occurring.	33 (16%)
Company culture	Ensure testing activities are valued and prioritized.	37 (17%)
Insight about change	Ensure understanding of a change before implementing it.	28 (13%)
Test logic	Tests should not only test the code, but the logic it represents.	21 (10%)
Effective test suite	Tests should be efficient, up-to-date, and relevant, to keep the test suite from growing needlessly.	20 (9%)
Regression testing	Ensure changes do not break unchanged code.	17 (8%)
Tool usage	Tools help in discovering faults and increasing code quality.	10 (5%)

> "... there are cases where code is not really testable. Let's say we have a structure where we can't mock stuff ... And then I would say sometimes testing is driving changes in production. So we try to make [a component] more modular." - P2

Theme: Quality assurance. This theme encapsulates how interviewees ensured test quality during test creation and maintenance. The sub-themes are explained in Table 10.

Preventative Measures: Participants strongly stressed the importance of ensuring code and test quality as a means of ensuring faults are avoided or removed. Participants state that it is better to have a rigorous and time-consuming testing process than to fix faults after the code has been deployed to customers. 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... 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 preventative measures pointed out by participants include targeting high test coverage, regression testing, code reviews, and collaborating with other teams that use the same code.

Participants also stressed the importance of 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 ... we are only part of the CI flow and there are other CI flows. Also, each area is testing different scenarios and 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: Two of the major preventative measure taken by the participants were to ensure high structural coverage and high input and output diversity in the tests. A 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 testing is valued. A team that takes pride in code quality will need to edit that code less often. Such teams will also be open to change and feedback.

Insight About Change: Participants discussed the importance of understanding why a change must occur within source or test code before implementing it—whether to fix a bug or perform a planned change. Gaining an understanding reduces the time to perform maintenance or evolution.

Explicitly adding a report to the issue tracking system allows the team to match the issue with the right developer, as each have different competencies:

> "The next step is always to have some kind of ticket. 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 more of a deep dive. Try and look at the logs. Try and see why the behaviour is what it is." - P1

Test Logic: It is important to design tests based not on the code, but on the logic that is the code is supposed to represent. Participants emphasized that code coverage is not enough, but that tests should also be designed to show that all possible outcomes of a function are demonstrated and that the code fulfills the requirements. They also stressed the importance of isolating individual logical elements 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 should be meaningful, and tests should regularly be evaluated to ensure they are up-to-date, relevant, and of high quality.

> "In our team, we evaluate the statistics of our tests. 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 suite to become bloated over time. Therefore, a significant element of test maintenance is ensuring that irrelevant or redundant test cases are removed.

Regression Testing: Code is naturally interconnected, and participants placed importance on ensuring that integration with new code does not affect code that has passed testing. Regression testing is one means of ensuring that unintended changes are detected quickly.

Tool Usage: Two different types of tools were described that help ensure quality—static analysis tools (e.g., linters) and automated testing. Code quality and coverage checks are performed as part of continuous integration.

Theme: Challenges. This theme collects test maintenance challenges affecting test maintenance, summarized in Table 11.

Hardest Task: Participants were evenly split on which test maintenance task was hardest, between identifying how the suite must evolve, modifying tests, adding new tests, and implementing the source change itself. Regarding identifying the change being the hardest task:

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

Participant P1 thought writing new test cases was harder:

> "Updating 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. If you're writing new test cases, you really need to think about all of the ways things can go wrong, and you need to look meticulously at every path through the code and make sure that you've covered it and have two or three different outcomes that could come out. If you're updating the test code, that's generally quite small. You know what needs to change." - P1

Growing Test Suite: Large test suites are harder to manage and understand, and have a long execution time. Test suites grow naturally over time, and continue growing during test maintenance, if care is not taken to remove outdated tests or to modify tests instead of simply adding new tests.

Table 11
Overview of sub-themes of "Challenges", with description and number of codes (and percentage of total for that theme) that correspond to that sub-theme.

Sub-theme	Description	Num (%)
Growing test suite	As test suite grows, harder to efficiently interact with.	16 (21%)
Hardest task	Opinions on hardest tasks in test maintenance.	15 (20%)
Proliferating problems	Interlinked problems, such as dependencies between test suites, require more effort to fix.	12 (16%)
Time	Test maintenance is de-prioritized when time is short.	9 (12%)
Missing coverage	Coverage may be reduced during maintenance, or lack of coverage may trigger maintenance.	9 (12%)
LLM not good enough	Help from LLMs feared to be not good enough to justify the effort required to implement suggestions.	6 (8%)
Unfamiliar with code	It is harder to find and change unfamiliar test cases.	5 (6%)
Technology limitations	Tech stack limits possible solutions.	4 (5%)

> "We don't want to always put in new test cases, because then that would mean we exponentially grow. What's happening is that the maintenance of the test code is as huge as actually developing the product." - P8

Proliferating Problems: Multiple elements of both source and test code can be interdependent, which can affect test maintenance. Individual tests are also often linked through their collective contribution to the overall test suite. Changing a test may increase the probability that the suite misses certain types of faults, or may reduce coverage of certain outcomes.

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

The effect of a change on the test suite should be understood in advance. This need introduces hesitance to perform updates and increases the time and effort of test maintenance.

Time: Participants noted that, when a deadline is approaching, test maintenance activities may be delayed or de-prioritized. A second time-related challenge raised by participants is that testing and maintenance require more manual effort than would be preferred, reducing the total amount of testing or maintenance that can take place.

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

Missing Coverage: Changing or removing a test may result in a loss of coverage. When maintenance is performed, it is important to ensure that the overall level of coverage remains the same or is improved.

Unfamiliar with Code: As the project evolves, it becomes more difficult to understand the source code, increasing the difficulty of test maintenance. Instead of changing tests, new tests are added, unnecessarily enlarging the suite. Large suites tend to have high test coverage. However, this can mask problems within the suite. Larger suites are also more difficult to understand.

Technology Limitations: The tech stack can limit the forms of testing and maintenance that can take place. For example, there may be requirements on the specific version of a programming language, testing framework, or code library used. Developers may also not be able to use certain tools or frameworks due to privacy or security concerns.

LLMs Not Good Enough: Some participants expressed concern that LLMs would not be able to offer sufficiently good or trustworthy performance during testing or test maintenance, and that it would require significant effort to correct their mistakes. They believed that testing could not be fully automated, and would still require significant human oversight.

> "You still have to tell it 'This isn't exactly what I meant. I need this' so you still need that dialogue back and forth." - P1

Given the importance of testing, skepticism towards automated approaches is reasonable and should be taken seriously.

Theme: Desired tool support. This theme focuses on requests and opinions regarding automated support for test maintenance, using LLMs or other means, summarized in Table 12.

Generating Code: As writing code is one of the main activities in testing and development, it is unsurprising that the most common request was for code generation. This help could come in many forms, from generating tests, to augmenting existing tests, to simply producing an outline or template.

> "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

> "It could detect production code changes and then just [generate] all tests that need to be applied. That would be great." - P3

Tracking/Understanding Problems: There is a desire for tools that detect potential issues in source or test code — a fault, bad practice, or another anomaly — explain the issues, and track whether they have been corrected.

If a failure occurs, the tester may want to know the source to help in debugging and prioritization:

> "We have a tool that will say [...] it a product failure or an environment failure ... 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

A developer may also want to know if a failure reappears. Tools could monitor for known failures or anomalies:

> "If something's gone wrong before then you don't want to repeat it and you want to be alerted if it goes wrong again. We create those dashboards manually and then create alerts. I can easily see how a generative AI, for example, could go and look at the data set and see, OK, this is normal behavior ... and then flag what goes wrong." - P1

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

Control Size of Test Suite: Participants sought tool support for detecting tests that could be removed without losing quality, whether redundant or because the tested code has not been modified for a long time. This could be done by a tool providing information on how tests overlap in code or outcome coverage or by identifying tests that are potentially out of sync with the code.

Ensure Coverage: Since having high coverage is recognized as important, getting help achieving coverage is important as well. A tool could simply generate tests. However, participants also suggested that tools could augment the input in existing tests.

Ensure Quality: Participants desired tools that detect problematic development or testing practices and make targeted suggestions, reducing the need for test maintenance. For example:

Table 12
Overview of sub-themes of "Desired Tool Support", with description and number of codes (and percentage of total for that theme) that correspond to that sub-theme.

Sub-theme	Description	Num (%)
Generating Code	Generating test code and improving already-written code.	20 (29%)
Tracking/Understanding	Discovering how/where problems might occur in code.	13 (19%)
Control test suite size	Ensure test suite is relevant/efficient.	13 (19%)
Ensure coverage	Prevent and find faults by ensuring high test coverage.	8 (12%)
Ensure quality	Alerting developers to inadequate testing or code quality.	5 (7%)
Automated execution	Automating execution of tests.	5 (7%)
Tool/Team synergy	Tool should fit into the team's workflow.	3 (4%)
Help and learning	Making it easier to find and understand new information.	2 (3%)

Table 13
Overview of sub-themes of "Attitudes Towards LLMs", with description and number of codes (and percentage of total for that theme) that correspond to sub-theme.

Sub-theme	Description	Num (%)
Skeptical	Do not think LLMs can efficiently help with software development.	14 (50%)
Unfamiliar	Have not used LLMs, either due to lack of opportunity or interest.	9 (32%)
Positive	Like the idea of getting help from LLMs.	5 (18%)

> "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 requested a pre-check for potential integration issues:

> "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 that require manual execution, such as running static analyses or locally executing unit tests before committing. Particularly taxing are cases test execution must be performed manually. Participants sought further automation of the testing process, including assistance with transforming manual tests into automated tests.

Help and Learning: Participants expressed a desire to mitigate knowledge or experience gaps with tool support—e.g., help learning how to write tests in a new language or framework or tips on how to make code more testable.

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

Tool/Team Synergy: The way of working in a team has a large effect on how effective any activity — not just test maintenance — can be. Incorporating a new tool requires consideration that current ways of working are not disrupted, and quality is 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 my method... I don't think it can understand that, but maybe understanding the basics of how our pipeline works." - P2

Theme: Attitudes towards LLMs. Participants' opinions on the use of LLMs in development could broadly be categorized (Table 13) as skepticism, unfamiliarity, or positivity. Many participants held a mixture of opinions, e.g., both positive towards trying LLMs while skeptical about certain aspects of their use. These attitudes are an important way to gauge the feasibility and desirability of implementing LLM-based tool support for test maintenance.

Unfamiliar: A few participants were unfamiliar with LLMs at the time the interviews were conducted. At the time, this mainly stemmed from restrictions on their use as part of their work at Ericsson due to security, privacy, and copyright concerns. Although Ericsson has approved LLM use since then, at the time, many employees had not used them at work. Some participants also expressed a lack of interest.

Skeptical: Some expressed skepticism about the abilities of LLMs. This could stem from a bad experience, things they have heard, a fear of over-hype, 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

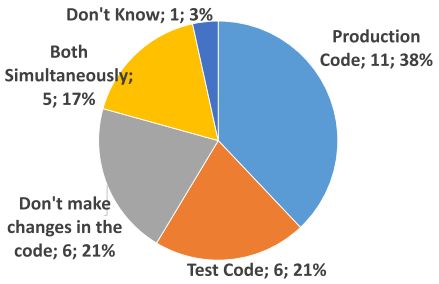
Positive: Despite some skepticism, five out of eight interviewees did state that these tools could help out in their work, either currently or in the future.

4.2.3. Analysis of survey responses (RQ1–2)

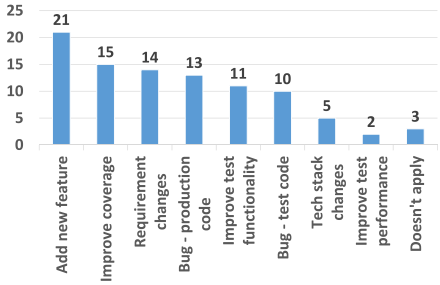
Survey responses related to test maintenance are shown in Fig. 5. Fig. 5(a) shows that, while changes to the source code are often made first, there are many cases where tests are changed first or simultaneously. This suggests that test maintenance is not always caused by a code change—i.e., the earlier list of triggers is not exhaustive. This is further evidenced in Fig. 5(b), where several common reasons for test maintenance—improving coverage, improving test functionality, fixing a bug in test code—do not require a source code change. In test-driven development, there could also be a change to the tests in anticipation of the change to the source code.

When test maintenance is triggered by a source code change (Fig. 5(c)), it is most often associated with a method addition, followed by a change to a method body or method return type/value. Three triggers identified in Section 4.2.1 were not reported by respondents as being among their 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, not all triggers occur with the same frequency, and some source code change may not always lead to test maintenance. The most effort-intensive task (Fig. 5(d)) was identifying what change needed to be made, followed by ensuring that the change was implemented correctly.

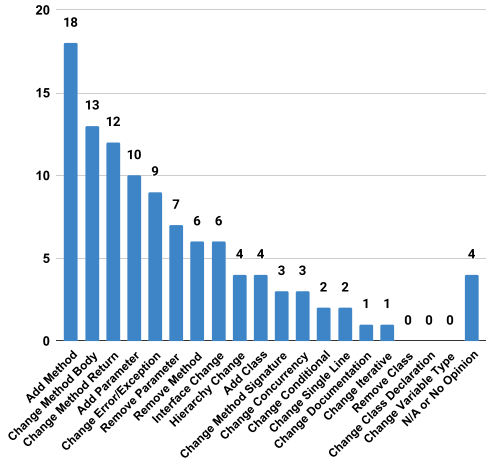
Responses related to automation and LLM use are shown in Fig. 6. Fig. 6(a) shows that a majority of respondents had tried to use an



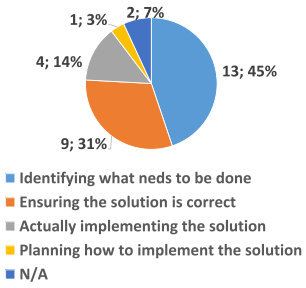
(a) Do you update source code or test code first?



(b) What reasons have led you to make changes in the test suite or in an individual test?

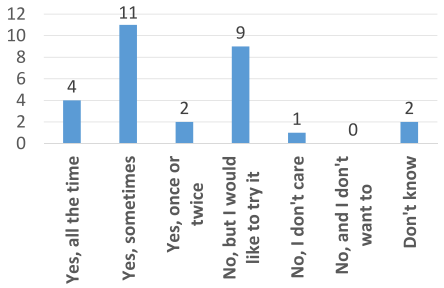


(c) Which source code changes most commonly necessitate adjustments in associated test code?

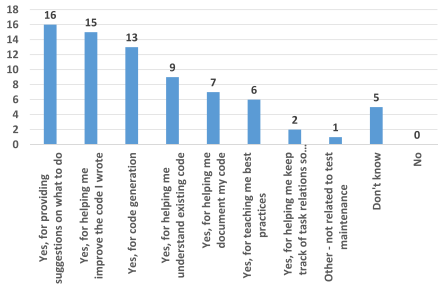


(d) During the test maintenance process, which step is most effort-intensive?

Fig. 5. Survey responses to questions about the test maintenance process.



(a) Have you used generative AI or LLMs?



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

Fig. 6. Survey responses to questions about automation and LLMs.

LLM. Approximately a third still have not tried LLMs, but as shown in Fig. 6(b), all respondent expressed a desire to use LLMs as part of test maintenance. The most common requests for assistance included providing suggestions on how to perform test maintenance, improving code that they had written, and generating test or source code. Code generation was only the third most-common desire, suggesting some skepticism and desire to retain control over test suite evolution.

4.2.4. 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.

Test Maintenance Actions: Fig. 7 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, including code generation (blue), testing partner (red), assessing code quality (purple), and discovering issues in the code (green).

One of the most common uses of LLMs is code generation (Hadi et al., 2023; Z. Zhang et al., 2023). Naturally, then, LLMs have been used to generate tests (e.g., Wang et al. (2024), Schäfer et al. (2023), Yuan et al. (2023), Lemieux et al. (2023) and Deng et al. (2024)). Creation of tests for new code is part of the maintenance process. LLMs have shown particular adeptness at generating complex input, e.g., for

Fig. 7. Overview of actions an LLM could take that could assist with test maintenance. Colors indicate clusters of related actions. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

forms (Z. Liu et al., 2023) or deep learning libraries (Deng et al., 2024). Multi-agent frameworks that iteratively integrate compilation errors have shown promise for reducing hallucinations (Yuan et al., 2023). LLMs can also generate failure-reproducing tests (Kang et al., 2023).

LLMs have been used to identify traceability links between natural language and code artifacts (Zhu et al., 2022; Lin et al., 2021), and could be applied to link source and test code. LLMs can also explain and summarize code (Hou et al., 2023; Z. Zhang et al., 2023; Nam et al., 2024), which can be applied to understand how source or test code has changed.

Researchers have explored the use of LLMs as pair programming partners (Ross et al., 2023). The user can ask follow-up questions for clarification (Feldt et al., 2023). The conversational nature of LLMs could allow them to act as a testing partner, offering advice, examples, explanations, and checklists.

LLMs can provide suggestions for code improvement (Hou et al., 2023; Z. Zhang et al., 2023; Lu et al., 2023). Such capabilities could be applied to the test suite. In addition, LLMs can identify additional scenarios to explore, areas lacking code coverage, or flaky tests (Hou et al., 2023; Fan et al., 2023; Sarkar et al., 2022). LLMs can also adapt tests to different devices (Yu et al., 2023), extract properties for metamorphic testing (Tsigkanos et al., 2023), improve test readability (Gay, 2023), and minimize test suites (Fan et al., 2023). LLMs can generate documentation (Hadi et al., 2023; Hou et al., 2023; Fan et al., 2023), code comments (Gay, 2023; Geng et al., 2024), as well as method and test names (Hou et al., 2023; Gay, 2023). All of these capabilities could be used to improve a test suite.

LLMs can also be used as part of debugging (Wang et al., 2024; Hou et al., 2023) and automated program repair (Hou et al., 2023; Xia et al., 2023). LLMs can also identify issues (e.g., faults, code smells) in source code (Hadi et al., 2023; Hou et al., 2023; Z. Zhang et al., 2023; Fan et al., 2023).

Considerations for LLM Deployment in Industrial Development: Considerations discussed in past literature are shown in Fig. 8.

LLMs have varying performance across tasks depending on how they have been trained and tuned, e.g., some models are tuned purely for source code tasks, while others are tuned for a blend of source code and natural language (Roziere et al., 2023). The first consideration when selecting a model is the type of task that needs to be performed. A model tuned for that task will almost always outperform models of the same size (measured in the number of parameters) (Zheng et al., 2023b). If many models are appropriate for a given task, evaluations on public benchmarks can be used to perform a comparison—e.g., HumanEval (Chen et al., 2021) for code generation or the LMSYS Chatbot Arena for conversational tasks (Chiang et al., 2024). However, many software engineering tasks do not have "standard" benchmarks yet (Wang et al., 2024).

However, benchmark performance may be affected by data leakage (Wang et al., 2024). Current benchmarks, such as HumanEval, have also been criticized for having insufficient test suites and for using vague problem descriptions (Liu et al., 2024a). Both could result in misleading performance estimates. In general, the larger the model, the better it performs (Zheng et al., 2023b). Larger models also demonstrate emergent abilities, such as in-context learning (Brown et al., 2020) and step-by-step reasoning (Shanahan, 2024), that are not always found in smaller models (Zheng et al., 2023a). However, larger models impose significant computational and energy costs (Wang et al., 2024). Models can be fine-tuned on an industrial codebase to improve their performance, and smaller fine-tuned models can sometimes outperform larger models (Zheng et al., 2023b).

There are also organization factors to consider. If a model is deployed by an external company, the cost of a license and limitations on the number of API calls must be considered (Mandvikar, 2023). There may also be privacy concerns, related to sending proprietary data to a third party (Wang et al., 2024). If an organization can afford the computational costs of internal deployment, then that is generally a better option. Internal deployment avoids privacy concerns (Wang et al., 2024). The organization can also tune models, change parameter settings, and choose when and how to deploy updated models (Wang et al., 2024).

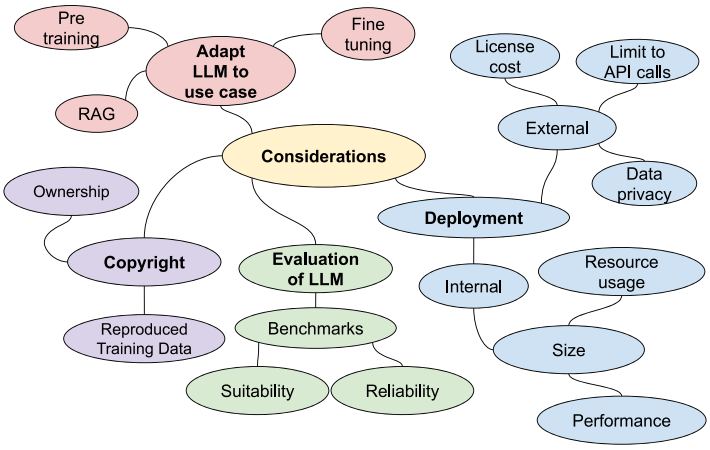


Fig. 8. Considerations for LLM deployment in an industrial environment. Colors indicate clusters of related considerations. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

An additional concern is copyright. LLM output, code and natural language, is derived from training data. LLMs may output copyrighted text, which could lead to issues if, e.g., that code is incorporated into the organization's codebase (Li et al., 2024). It is unclear whether an organization can own copyright over code generated by an LLM (Gewirtz, 2023).

4.2.5. Triangulation across literature, interviews, and surveys

We base our conclusions on three information sources: literature review, interviews, and surveys. We use each source for different purposes, but some observations were directly yielded or supported by multiple sources. To gain further insight, in this section, we summarize observations made across multiple information sources.

Triggers: The literature review on triggers identified 37 specific triggers that lead to test maintenance, generally corresponding to changes to the source code or documentation. The interviews generally confirmed the validity of these triggers, with participants specifically mentioning feature implementation, requirements changes, modifications to methods, and modifications to conditional statements as being causes of maintenance. The interviews also offered additional triggers corresponding to process-level activities such as refactoring, test suite assessment, or technology changes leading to test maintenance, supplementing the initial set extracted through literature review. We based the specific triggers that we asked about in the survey on the literature review. Therefore, the survey did not offer any additional triggers directly. However, the survey results further confirmed that not all test maintenance stems from a code change. Further, participants ranked the frequency of triggers, offering additional insight missing from the literature.

LLM Support for Test Maintenance: The interview participants suggested multiple ways that LLMs could support test maintenance, including test generation and augmentation, template creation, debugging support, anomaly detection, optimization of test suite size, education and training, transformation of manual tests into automated tests, and targeted suggestions based on detection of bad test smells. In the survey, we asked participants to rate these suggestions, and they identified targeted suggestions, test augmentation, and test generation as the most beneficial. All of these forms of LLM-based support have also been suggested by the research literature — as shown in Section 4.2.4 — suggesting that LLMs have great potential to fulfill the needs of practitioners.

LLM Attitudes: During the interviews, the participants were split between being unfamiliar with LLMs and having either positive or skeptical attitudes, with most participants having a positive outlook. The survey respondents showed a similar split, with the majority both familiar and positive about the potential of LLMs in software development. One interview participant raised questions about copyright over generated code, which was also an issue raised in the literature review on LLM deployment considerations. However, we also observe one gap in the literature—consideration of how LLM-based tools can fit into existing ways of working. In the interviews, participants raised the concern that tool support must fit into a team's culture and workflow. The literature review surfaced a number of technical concerns — adaptation, deployment, benchmarking — but little consideration of human-AI interaction methods, workflow integration, or limitation of the disruption of introducing new tools and ways of working. To ensure successful deployment of LLM-based tools, it is important to consider team culture and workflows. Other challenges identified in the interviews may also affect LLM deployment, such as technical limitations, time and budget limitations, and concerns about the quality and accuracy of LLM output.

4.3. Discussion

In this section, we provide answers to the research questions, based on the results detailed previously.

4.3.1. Test maintenance triggers (RQ1)

Our results suggest the existence of both *low-level* and *high-level* triggers that create a need for test maintenance.

Low-level Triggers: Past research has focused predominately on test maintenance triggers related to individual changes to the source code or documentation. These triggers are summarized in Tables 4–7, and were confirmed during the interviews and survey.

That low-level changes can trigger test maintenance is intuitive. If functionality evolves, its test must also evolve. If functionality is added or removed, then tests should be added or removed as well. However, the existence of such triggers *does not guarantee* that maintenance is required. All of the commits in the evaluation dataset used to address RQ3 contain source code changes — i.e., low-level triggers — but not all of those changes led to test code changes. 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 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. Future research should examine the causal relationship between source and test code evolution in more detail to better predict when a trigger will lead to requires maintenance.

Table 14
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.

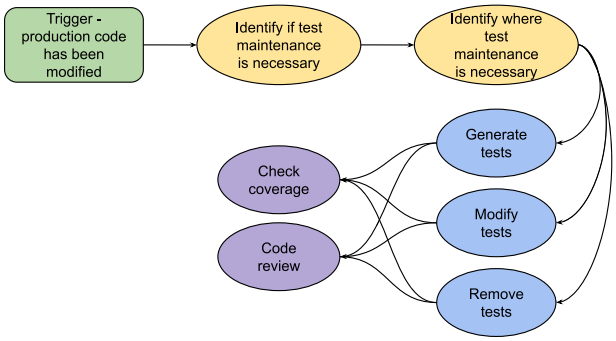


Fig. 9. LLM agents could generate tests, or modify or remove existing tests, based on a combination of triggers, code coverage, and code quality review. Green = trigger, yellow = explored aspects, blue = new test maintenance actions, purple = new code analyses. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Test Maintenance Triggers (RQ1): We identified 37 low-level changes to functionality, documentation, classes, methods, and single lines of code that can trigger test maintenance in situations where test cases invoke and detect those changes.

High-Level Triggers: From interview and survey responses, we identified additional "high-level" test maintenance triggers. These triggers represent 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 — hence, no low-level trigger — but tests may be added or existing tests may be augmented. Like with the low-level triggers, such decisions do not always guarantee test maintenance, but are frequently associated with it. We present these high-level triggers in Table 14.

Test Maintenance Triggers (RQ1): We identified seven high-level development decisions that often trigger test maintenance, including adding features, requirement changes, fault discovery, increasing coverage, refactoring, tech stack changes, and test suite evaluation.

4.3.2. Applications of LLMs in test maintenance (RQ2)

We examined the theoretical applications of LLMs or agents in test maintenance, considering how the triggers could be acted upon, what actions could be taken based on these triggers and other data signals, and what considerations should be made when deploying LLMs in an industrial environment.

Use of Low-Level Triggers by LLMs (RQ2.1): Because low-level triggers can be detected in the code or documentation, they could be directly used by LLM agents — potentially with other data signals — as the impetus to perform actions on the test suite.

Fig. 9 illustrates one potential scenario where an LLM agent performs an initial assessment of a low-level code change. The triggers could be be blended with other analyses by the LLM to increase the confidence in the action determined to be appropriate. Once the need for test maintenance is predicted, additional agents could automatically modify the test suite. The low-level triggers may suggest certain corresponding actions. For example, a change to a method signature necessitates changes to all test cases that invoke that method.

Data extracted from other tools — e.g., code coverage information — or other analyses performed by LLMs — e.g., inspecting code quality — could be used along with the triggers to suggest specific modifications to the test suite. For example, if new test cases are needed because new functionality has been added to the project, then these analyses and data could be used to explore how to test the new function, generate tests, then refine the tests to reduce redundancy or improve efficiency.

Use of Triggers by LLMs (RQ2.1): Low-level triggers, as well as other contextual information (e.g., coverage or quality analyses), could — among other tasks — identify and modify tests or suggest new tests.

Use of High-Level Triggers by LLMs (RQ2.1): As high-level triggers reflect development decisions, these triggers may not be detectable automatically. However, an alert that such a decision has been made could be used by LLM agents to perform or assist in planning and testing activities. Fig. 10 suggests the types of actions that LLM agents could take based on high-level triggers, including general actions, code modifications, and other actions intended to improve test or source code quality.

Fig. 11 offers examples of specific actions for each trigger, extracted from interview and survey responses and past literature. For example, given changed requirements, agents could identify the artifacts that need to be adjusted, including code and documentation (e.g., user stories). They could then recommend best practices for updating the code and test suite, assess risks and rewards of different solutions, modify the artifacts, generate tests that verify that the new requirements are met, evaluate code quality, and suggest performance-improving refactoring to the code and test suite.

Use of Triggers by LLMs (RQ2.1): LLM agents could act on high-level triggers by performing or assisting in activities such as providing guidance, evaluating and offering recommendations on 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 test maintenance tasks such as:

- Establishing traceability between artifacts.

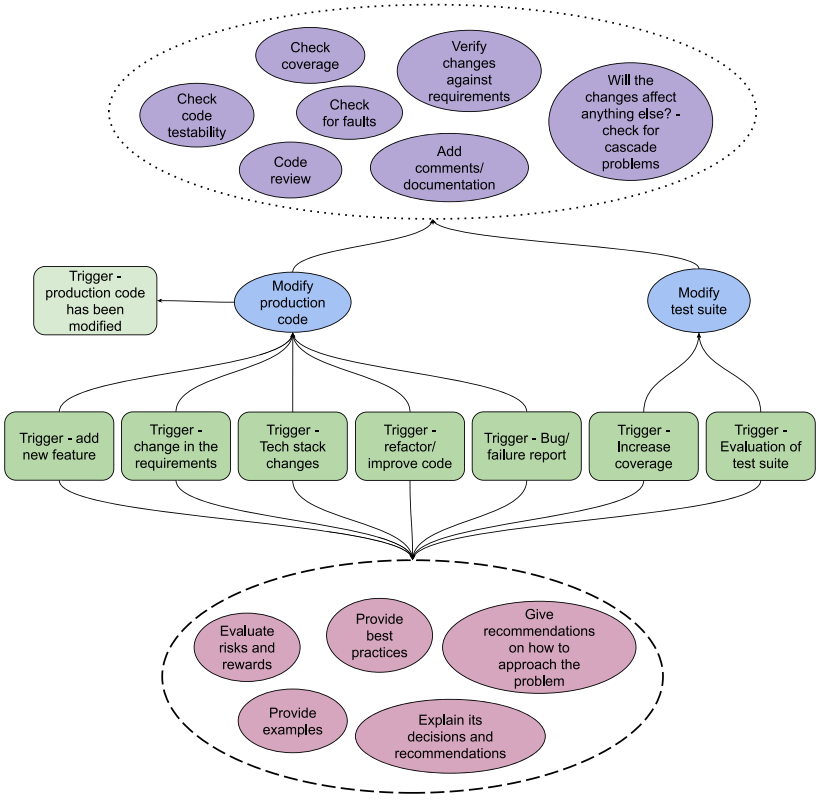


Fig. 10. General actions an LLM agent could take based on a high-level trigger. Green = high-level trigger, light green = low-level trigger, purple = potential code or test quality actions, pink = general LLM actions, blue = code modification actions. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

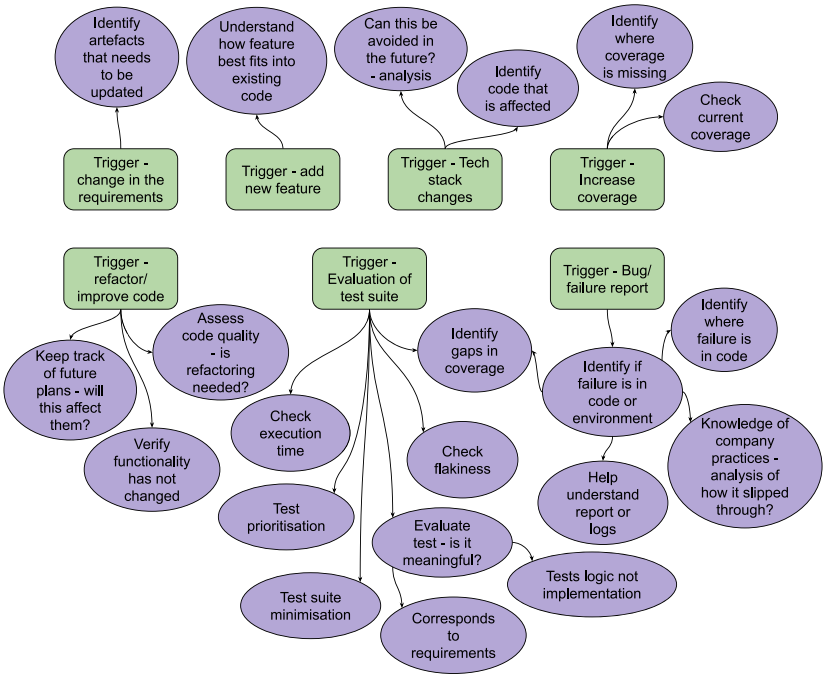


Fig. 11. Suggestions on specific actions an LLM agent could take based on high-level triggers to improve the test suite quality. Green = high-level triggers, purple = potential LLM actions. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

- Helping developers understand where and why maintenance is needed.
- Generating, adapting, or optimizing test cases and suites.
- Explaining or summarizing code and tests.
- Acting as an interactive testing partner, providing examples and advice.
- Assisting with general development activities that might appear during test maintenance, such as code review, analyzing and improving source and test code, and generating documentation.

Our prototypes demonstrate that LLM agents can summarize source code changes and test code, establish traceability between code changes and test code, and use those analyses to predict the need for test maintenance. Figs. 9–11 further illustrate how triggers and other data could be used by LLM agents to perform additional actions.

Significant research effort has been dedicated to code and test generation. However, it is important to also consider how LLMs can be applied to a broad range of test activities, as well as how LLMs and humans can interact as part of the test maintenance process. Test maintenance is a long and complex process intended to improve the quality of the project, consisting of more than simple adaptation of tests or creation of tests for new features.

Test Maintenance Actions (RQ2.2): LLM agents can perform or assist with test maintenance activities, including — broadly — code generation or modification, acting as a conversational partner, and code analysis or documentation.

Industrial Considerations (RQ2.3): Our literature review, as well as the interview and survey responses, suggest that the following considerations are the most important when deploying LLMs in an industrial environment:

- Except for code generation, there are few established benchmarks. Performance estimates may also be affected by data leakage, insufficient test suites, or poorly-defined prompts. Benchmarks should be taken as a basic, but not absolute, performance indicator.
- In general, the larger the number of parameters, the better an LLM will perform without tuning. However, larger models carry significant computation requirements.
- Deploying in-house offers control over the model, opportunities for tuning on organization data, and mitigates security and privacy risks.
- Fine-tuning a model on organizational data can be expensive and time-consuming. LLM agents may be a more efficient option, as RAG can offer access to current organizational data.
- Copyright and ownership of AI-generated output is a concern. Using models where the training data is public is a possible solution.
- Skepticism should be taken seriously. Expectations on LLM performance should be set carefully and LLM deployment should be done in a controlled manner to ensure that the models are sufficiently effective before all developers are encouraged to regularly use them.
- Many test maintenance challenges are rooted in a lack of knowledge, both project-specific and general (e.g., on testing practices). To perform test maintenance, one may — for example — 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 LLMs can provide are summarization, insight, and recommendations regarding project artifacts.
- LLMs should be deployed with some human oversight. Performance limitations and skepticism suggest that LLM output should be treated as "recommendations" reviewed by a human. LLMs also have value as conversational partners. LLMs should not fully replace human effort, but can augment the effectiveness and efficiency of human effort.

Industrial Considerations (RQ2.3): When deploying LLMs, organizations should consider model performance, whether to deploy in-house or externally, copyright and ownership, how to incorporate organizational data (e.g., fine tuning or RAG), how to establish and manage expectations, which tasks LLMs should support, and how humans and LLMs should interact when performing these tasks.

5. Tool study

In the exploratory study, we identified low-level and high-level triggers that signal a potential need for test maintenance (Tables 4–7, 14). Figs. 9–11 suggest potential actions that LLMs or LLM agents could take in response to those triggers to support test maintenance.

These results suggest that LLMs have many *potential* applications in the test maintenance process. As a initial contribution to the field, we have designed a multi-agent framework that performs one *concrete* task—**prediction of which test cases need to be updated following a change to the source code**.

This task represents a partial implementation of the scenario outlined in Fig. 9. If a low-level trigger occurs — i.e., a change has been made to functionality, a class, a method, or a line of code — then changes may be required in the test suite that invokes that code. The first step that must occur is that the affected test cases must be identified. This is necessary before subsequent actions — like modification to match the source code changes, addition of new tests, or removal of obsolete tests — can take place.

Specifically, our implementation takes as input a source code change, then issues output either confirming that no test maintenance is required or — alternatively — suggestions of which test cases have been affected by the change. We target code-based test cases at the unit and integration level. We implemented two variants of this tool, with one operating on the test code and the other operating on natural language summaries of test cases. This tool can serve as a starting point for future extensions that perform additional actions with the identified test cases.

Section 5.1 explains the study and the research methods employed. Section 5.2 describes the results of this study. Finally, Section 5.3 discusses the implications of the results.

5.1. Methods

The **goal** of this study is to *propose and evaluate the performance of two variants of a proof-of-concept tool that predicts the unit and integration tests that must be updated following a low-level change to the source code.* We operationalize this goal into the following **research question**:

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

This study was also conducted as a case study at Ericsson AB, following the guidelines from Runeson and Höst (2009). We answer RQ3 using the following data:

- **Quantitative Data:** We evaluate performance on a dataset of commits. In cases where test maintenance was required in the ground truth, we evaluate accuracy, precision, recall, F1 and F2 score. In cases where maintenance is not required, we evaluate accuracy and average number of false positives.

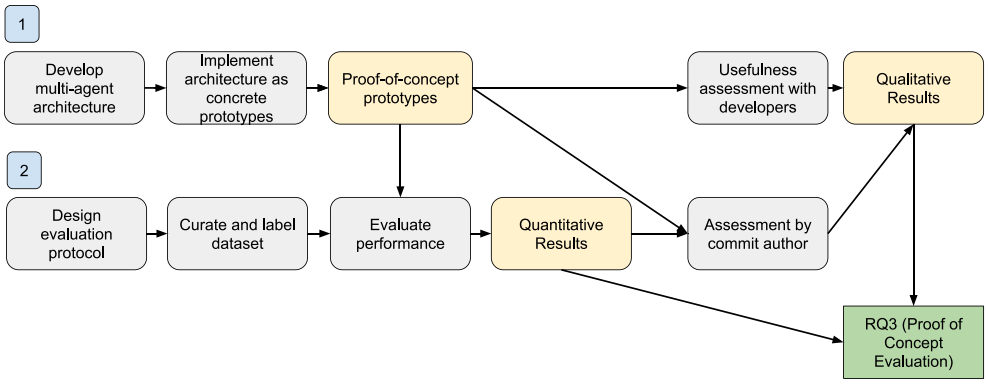


Fig. 12. Overview of the tool study. The numbers correspond to the steps outlined in Section 5.1. Gray = activity, yellow = artifact, green = research question. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

- **Qualitative Data:** We provide additional qualitative context from a usefulness validation conducted with two practitioners and an assessment of false positives by a developer responsible for the repository. This data is not intended as a formal assessment, but as additional context for interpreting the quantitative data.

The data was gathered through the following **activities** (Fig. 12):

1. We propose a **multi-agent architecture** — and implemented two prototypes based on this architecture — to explore the viability of using LLMs to predict the need for test maintenance (Section 5.1.1). These prototypes are used to address **RQ3**.
2. We **evaluated the performance of the prototypes** (Section 5.1.2). This evaluation, along with two additional qualitative characterizations of the tool and its results, addresses **RQ3**.

5.1.1. Proof-of-concept design (RQ3)

To explore the practical applicability of current-generation LLM agents, we have developed a proof-of-concept multi-agent framework that observes changes to the source code, then predicts which test cases require maintenance. This section presents the architecture for this framework, as well as details on two concrete implementations of this architecture.

Overall Architecture: We focus on an agent-based architecture, where retrieval-augmented generation is used to obtain information from the source and test code, and where memory capabilities are incorporated to enable complex multi-step processes and correction of previous mistakes. Each LLM agent and individual LLM instance in the architecture focuses on a particular subtask, and each agent employs specialized tools for that task.

This architecture (shown in Fig. 13) employs a planning agent that coordinates the effort of other LLM agents, following this sequence of steps:

1. A code change (`git diff`) is provided to the framework.
2. The planning agent provides the code change to the test maintenance agent to determine whether test maintenance is necessary.
   - The test maintenance agent requests a summary of the change from the code summarizer agent.
   - The test maintenance agent will provide the summary to an LLM instance to determine whether test maintenance is necessary.
3. The test maintenance agent sends the result to the planning agent. If maintenance is necessary, the agent will send a confirmation and the change summary. Otherwise, the agent will send an explanation, which the planning agent will use to provide a final response.
4. When test maintenance is necessary, the planning agent will contact the test localization agent, sending the code change and the summary from the test maintenance agent.
   (a) The test localization agent invokes the test retriever tool, which uses RAG to retrieve relevant test cases based on the input.
   (b) The retriever tool will either fetch test code or summaries of test code, depending on which prototype is being used.
5. The test localization agent responds to the planning agent with a list of test cases that need to be changed.
6. The planning agent responds to the user, either indicating that test maintenance is unnecessary or providing information on which test cases need to be changed.

The framework could either be directly invoked by a user, or could be automatically be invoked by a build system once code changes are committed to a repository. An example of the output of the tool is shown in Fig. 14. Additional examples of the output are shown in Section 5.2.1.

Application Setting and Differentiation from Related Work: Our proof-of-concept was designed to be deployed in a real-world setting, i.e., in the development environment at Ericsson. This aim informed the workflow of the tool and our design decisions. Unlike, for example, Hu et al. (2023), our implementation takes a `git diff` as input — a standard method of summarizing code changes — enabling direct invocation following a commit by either a developer or as part of an automated process. While Hu et al. (2023) was limited to analyzing changes to a single production method, our approach can process repository-level changes. Previous approaches, e.g., Chi et al. (2025), also assumed a precise, pre-existing mapping of production and test artifacts, while our prototype infers that relationship. These design decisions offer novel extensions over the state-of-the-art and were necessary to ensure the direct applicability of our tool for its intended use case.

LLM Agent and Instance Implementation: We have implemented two versions of this architecture. The first examines the test code directly to determine whether a test requires maintenance. The second employs an LLM to generate a natural language summary of all test cases beforehand, then acts on that summary instead of the code. Both prototypes implement the architecture depicted in Fig. 13, with the difference being the content accessed by the Test Retriever Tool using RAG.

The agents are implemented using the LangChain framework (Chase, 2022). The LLM agents and instances employ the Qwen2.5-32B-Instruct model (Yang et al., 2024). Qwen2.5-32B-Instruct is a version of the Qwen2.5-32B model that has been fine-tuned to follow

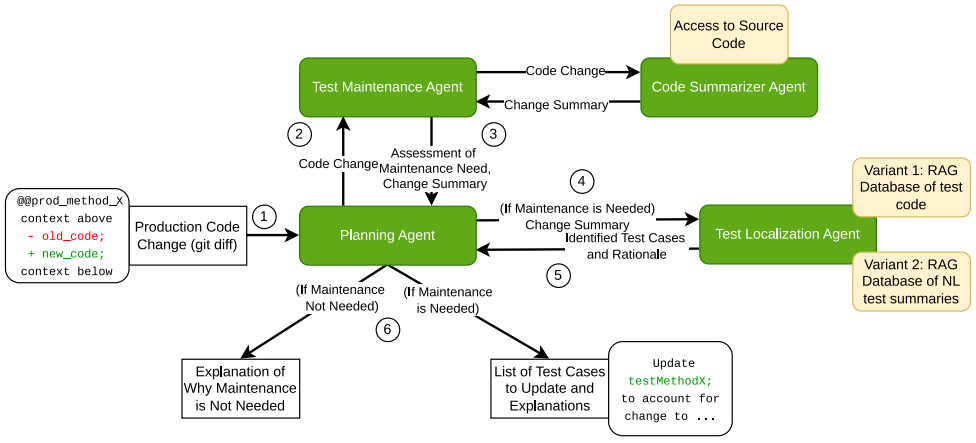


Fig. 13. Architecture incorporating multiple LLM agents (dark green). Yellow represents tool use (ability to retrieve external context). Step numbers (in circles) refer to the steps in the explanation in the text. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

```
The following test case requires maintenance due to the addition of the `postfixFields`
parameter to the `rejectInvalidFileNotification` method:

1. **testProcessingRejectedFile()**

This test need to be updated to accommodate the new parameter to ensure it continues to
validate the correct behavior of the method with the updated signature.
```

Fig. 14. An example of the output to the user from the tool, predicting the need for maintenance of a test case and explaining why maintenance is needed.

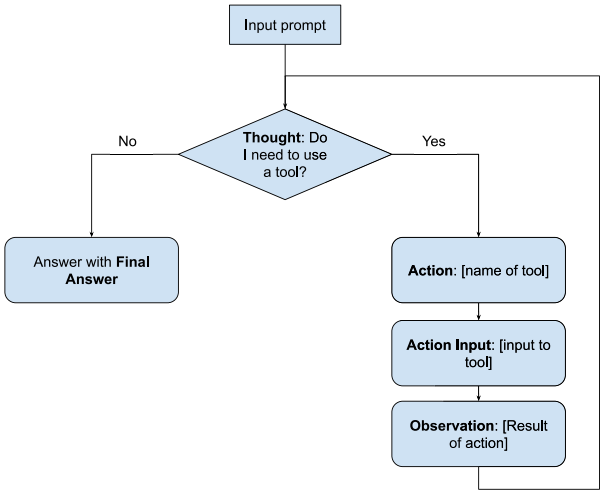


Fig. 15. Flow diagram showing how a ReAct agent invokes tools to perform tasks.

instructions. Ericsson must approve LLMs for internal use. At the time of implementation, we determined that Qwen2.5-32B-Instruct was the best model for our use case, considering availability, capabilities, and the Ericsson approval process and guidelines.

The LLM agents are implemented following the ReAct agent structure (Yao et al., 2022), visualized in Fig. 15. 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 is satisfactory, the agent stops, and if not, it continues to iterate until an answer or a timeout is reached (maximum 3 iterations for each LLM agent in our implementation). 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 (counting as an iteration). We also impose a limit of 300 s per prompt.

Each agent is implemented with its own individual memory, planning mechanisms, and tools. The memory 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 and how to act.

LLM instances are invoked through a chain of prompts. As they are not full agents, they do not have access to memory or tools. However, they offer a faster response. Tools are utilized through separate calls that then send results as input to the LLM instances. The prompts provided to each LLM agent or instance are provided in our supplementary data package.[3]

The planning agent controls the overall process. It invokes other agents as tools that it can utilize. If it determines that their output is insufficient, it can invoke them again. Its prompt specifies that it needs to use these "tools", expands on its tasks, specifies the input format (`git diff`), and offers additional instructions.

The test maintenance agent invokes the code summarizer agent as a tool to construct a natural language summary of the `git diff`. It then provides that summary to a simple LLM instance to determine whether test maintenance is likely to be necessary.

The test localization agent uses a RAG tool to access either the raw test code, or natural language summaries of test cases, for the specific commit that the `git diff` references. This tool is used to determine which test cases are relevant to the code change. The RAG tool uses the bge-m3 embedding model (Chen et al., 2024) to encode code for processing by an LLM. This was the best embedding model in current deployment at Ericsson. We employed the Facebook AI Similarity Search (FAISS) library (Douze et al., 2024) as a vector store for the embedded vectors.

[3] Available at https://doi.org/10.5281/zenodo.13341060.

5.1.2. Proof-of-concept evaluation (RQ3)

We evaluated the performance of the two prototypes using an Ericsson repository. To help contextualize the results, we then performed two additional qualitative validations with Ericsson developers—the first a general examination of the usefulness of the prototypes, and the second an assessment of the prototypes' predictions by a developer responsible for a subset of the commits used to benchmark the prototypes.

The evaluation and additional exploration of the results can help characterize the current capabilities and limitations of LLM agents to predict the need for test maintenance. Note that we do not compare to a baseline result, as we are not aware of other tools that can be assessed on the same basis or in the same environment (the limitations of past work were discussed in Sections 3 and 5.1.1). Therefore, we do not present our tool as the best possible methods for performing test maintenance prediction, but as a proof-of-concept to illustrate feasibility.

Evaluation Dataset: We developed a dataset from the commit history of a Java-based repository at Ericsson. The repository contains code for a microservice that performs data processing tasks. The test code contains unit and integration tests.

Our ground truth is based on the assumption that source and test cases in the same commit represent co-evolution. The individual code changes are captured in the form of the `git diff` files for that commit. The following code changes were excluded:

- Commits with no changes to the source code.
- All changes to test code that were not encapsulated within a test case, such as changes to test support code.

The dataset spans 54 commits, which collectively contain 634 distinct code changes. The default `git diff` only includes three context lines above and below each change. We extended this to 9 nine lines to incorporate more contextual information around the changed code. This number was selected based on informal experimentation.

Evaluation Procedure: For each commit, we performed the following[4]:

1. The information stored in the RAG tool was updated to reflect the test suite used to assess the current commit.
2. Each individual code change was input into the agent setup. These output was saved. If maintenance was necessary, we extracted the names of suggested test cases.
3. Ground truth is known for a commit, not for each code change. To assess accuracy, the suggested test cases for all code changes in that commit were compared to the test cases changed in the actual commit.

In this experiment, we used a temperature of 0 to limit stochasticity. However, we performed the evaluation twice, as there may still be some variance. As the results were very similar across both trials, we did not perform further repetitions. For each commit, we define:

- A **true positive (TP)** is a test predicted as *requiring* maintenance that *actually* required maintenance.
- A **true negative (TN)** is a test predicted as *not requiring* maintenance that *actually* did not require maintenance.
- A **false positive (FP)** is a test predicted as *requiring* maintenance that *did not* require maintenance.
- A **false negative (FN)** is a test predicted as *not requiring* maintenance that *actually* required maintenance.

[4] The per-commit results are available in our supplementary data package: https://doi.org/10.5281/zenodo.13341060.

The commits in the dataset reflect two distinct outcomes. In 32 commits, test maintenance was *required*. In the remaining 22 commits, *no test cases were updated*. We split these cases when evaluating performance for two reasons. First, in cases where no maintenance was required, there can be no TP or FN, only TN and FP. Second, the commits for each case are imbalanced in the number of code changes per commit. In cases where tests were updated, there is an average of 16.7 changes per commit. In contrast, the cases where maintenance was not required average only 4.5 changes per commit.

We calculate the following metrics:

- Accuracy $= \frac{TP+TN}{(TP+FP+TN+FN)}$
- Recall $= \frac{TP}{(TP+FN)}$
- Precision $= \frac{TP}{(TP+FP)}$
- F1 score $= 2 * \frac{(precision*recall)}{(precision+recall)}$
- F2 score $= (1+2^2) * \frac{(precision*recall)}{(2^2*precision+recall)}$

We use these measurements to provide multiple viewpoints on performance. In particular, as the datasets contain a large number of TN, the F1 and F2 scores are useful because they focus on TP, FP, and FN. While the F1 score is commonly used to assess machine learning tasks, we also include the F2 score, which places a heavier weight on recall than precision. We do this because we hypothesize that recall is more important than precision in this task—omitting tests that should have been included in a prediction is potentially more detrimental than including additional false positives.

For the subset of commits where test maintenance was required, we report all five metrics. For the subset of commits where test maintenance was not required, we only report accuracy, as TP and FN outcomes are not possible.

Usefulness Validation: To help understand the applicability of the tool, we held an informal validation of the prototypes with two developers from Ericsson. Both were familiar with LLMs and the specific context of this study. During the validation, the developers were shown the prototypes, the results of the experiment, and examples of their output. They were asked to provide reflections on what adaptations or requirements would need to be met to deploy the prototypes in practice. This validation is not meant to provide rigorous evidence, but to help contextualize the tool.

Assessment by Commit Author: To further contextualize the results, we conducted an interview with the primary developer responsible for most of the commits in the dataset. The interview questions are listed in Table 15.

The interview was conducted in three sessions. In the first, we asked general questions about test maintenance. In the second, we presented three commits selected from the dataset, performance results for those commits, and a list of 38 test cases identified as FPs. These results were from the prototype that utilizes test summaries.

We asked the developer to classify each suggestion as "useful" or "not useful", and to explain why. We then presented the prototype's output for each FP — the explanation of why the prototype thinks the test potentially needs to be updated — and asked the developer to re-examine their previous comments on each FP. The developer determined whether the output was useful or not. A suggestion was deemed potentially useful if:

- The test could be affected by the changed code.
- The explanation is substantiated.
- The suggestion is either correct, or if not correct, could benefit the test suite or maintenance process (e.g., suggests new tests to create).

In the final session, we asked questions about their performance preferences when using the prototype as part of their workflow.

Table 15
Interview with commit author.

Session 1: Questions about test maintenance	
1	When you modify the code, how do you identify which test cases need to be updated?
	How difficult do you think it is to locate the correct test cases to update? How confident are you that you have identified all relevant test cases? If all tests pass for your change, do you still update any test cases?
Session 2: Questions regarding FPs	
2	What is your impression of these FPs (**before** checking the prototype output)?
	Do the FPs offer any useful hints about which test cases might need to be changed? Do you think any of them might actually be TPs?
3	What is your impression of these FPs (**after** checking the prototype output)?
	When you think that output for a FP is **useful**, what do you mean? When you think that output for a FP is **not useful**, what do you mean?
Session 3: Questions after checking FPs	
4	Which metric matters more to you, recall or precision?
5	If you are using this tool, how fast do you want it to respond to, e.g., one code change?
6	For the FP that you feel should actually be TP, why do you think you missed those cases?
7	What other suggestions do you have regarding the prototype?

Table 16
Evaluation of prototypes when >0 tests are updated in the ground truth.

Prototype	Recall	Precision	Accuracy	F1 score	F2 score
Planning (With summary)	**0.676**	**0.275**	**0.911**	**0.391**	**0.523**
Planning (Without summary)	0.412	0.213	0.908	0.281	0.347

Table 17
Evaluation of prototypes when no tests are updated. Only FP and TN are possible in this scenario.

Prototype	Avg. FPs/commit	Accuracy
Planning (With summary)	5.5	0.974
Planning (Without summary)	**3.7**	**0.982**

5.2. Results

This section presents the results of the evaluation and observations from the two qualitative explorations.

5.2.1. Proof-of-concept evaluation

We evaluated the performance of two prototypes — one that makes predictions based on the raw test code and one that generates and operates on natural language summaries of the test code — over the dataset. This evaluation was performed twice to assess variance. As the results of the two executions were very similar, we present average results. To perform this evaluation, we split the dataset based on two scenarios. The first subset contains cases where one or more tests were updated in the ground truth, while the second contains cases where no tests were updated.

Table 16 presents the average performance for cases where tests were updated in the ground truth. The best performance in each measurement is **bolded**. The planning agent that makes use of summaries attained the highest performance in all measurements, suggesting a potential benefit from translating the test code to natural language for comparison with the natural language summary of the corresponding source code change.

Table 17 presents performance measurements for cases where no tests were updated in the ground truth. For this subset, only FP and TN results are possible, meaning that recall, precision, F1, and F2 score are undefined for this subset. For this subset, the best prototype was the one based directly on the test code.

The accuracy for both subsets is very high. However, this measurement is skewed by the large number of TN cases. The test suite for the repository is large. In most situations, a very small number of tests will be updated when the source code is updated. Therefore, it is useful to focus on the other measurements to better understand the performance of the prototypes.

In the first subset (Table 16), the recall is relatively high for the prototype based on summaries, but precision is low. In Table 17, we also indicate the average number of FP per commit. Together, these results suggest that the primary factor affecting performance is FP—this prototype is more likely to suggest *extra* tests than to miss an important test.

Because it works with natural language rather than code, the prototype based on summaries makes a larger number of positive predictions (TP and FP) than the prototype based on test code. This improves its performance in situations where tests have been updated in the ground truth, but leads to worse performance in cases where no tests were updated.

The predictions made by the prototype are not strict "yes" or "no" predictions, but are more nuanced. At times, the prototype will confidently suggest that certain tests should be changed (Fig. 16) or that no maintenance is required (Fig. 17). At other times, when less confident, it may suggest that a particular test should be reviewed or make general recommendations about reviewing tests related to particular code elements (Fig. 18). Sometimes, it even suggests creating new tests rather than performing maintenance (Fig. 19). The languages used in the output also varies, e.g., suggesting that a test "needs to be updated", "should be updated", or "might need to be updated", based on the prototype's level of confidence.

These general suggestions or even a FP — even if not technically *correct* — may still benefit a tester as they update the test suite. These suggestions may have a role in the brainstorming process, as long as the number of FP remains relatively low. We further investigate the utility of these suggestions in Section 5.2.2. FN results are more concerning, and in future work, we will explore means of improving recall further.

5.2.2. Qualitative explorations with practitioners

Usefulness Validation: 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 prototype runs much faster than executing the tests — generally returning a result in under 30 s — 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 the existing test suite despite requiring maintenance. However, there is still risk. If there is a hallucination or a false-positive, then significant time may be wasted. Future research should evaluate the actual time savings of using such a tool.

```
The following test cases require maintenance due to the code changes:

1. `testParameters()`: Needs to be updated to reflect the new nested structure of the
   `parameters` field.
2. `testConfigLoads()`: May need to be updated to ensure that the new configuration
    elements, including the `DEFAULT_KEY`, are being loaded properly."
```

Fig. 16. An example of a confident prediction that maintenance is required.

```
No test maintenance is needed based on the provided code changes. The changes involve
configuration updates that do not directly impact the existing test cases identified.
```

Fig. 17. An example of a confident prediction that no maintenance is required.

```
While no specific test cases were identified as needing updates based on the provided
information, it is recommended to review and potentially update any test cases that
interact with the `FlowConfig` class or depend on the behavior of the methods
`isRenamingEnabledFor`, `getEmptyFileSize`, `getSubdirectoryDatePattern`,
`getParameterValue`, and `getParameters`. This is due to the introduction of the `project`
parameter and the change in the return type of `getParameters`, which may affect the
expected behavior of these methods.
```

Fig. 18. An example of a general suggestion that tests be reviewed.

```
No existing test cases are directly impacted by this change, but new test cases should be
created to cover the updated behavior of `getNextTransformation`, specifically focusing
on scenarios involving different combinations of `eventData.getProject()` and
`eventData.getFileClassification()`.
```

Fig. 19. An example of a suggestion that new tests be created.

Table 18
Categorization of FP suggestions by commit author.

Category	Subcategory	Count	Total
Useful	Test case needs to be changed (TP)	7	16
	Suggestion carries benefits to suite or maintenance	9	
Not useful	Test not relevant to changed code	10	14
	Unreasonable explanation	4	
Other	Problem with ground truth (TP)	7	8
	Unable to understand test case	1	

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 prototype offered guidance that could be factored into their decision, increasing their confidence. Due to the limited number of participants, both observations should be validated in a larger future study.

Assessment by Commit Author: In the first session, we asked questions about how the developer performs test maintenance. Similar to suggestions made by the interview and survey participants, this developer primarily relied on two sources of information. The first is their intuition about the codebase, and the second is the results of running the test suite on the updated codebase. However, they noted that these signals are not enough to feel confident that all tests that need maintenance have been correctly identified.

In the second session, we presented 38 FPs from three commits. Initially, we presented the name of each test and asked the developer to classify the suggestion as useful or not, and to explain why. We then presented the prototype's full output for each FP and asked them to re-examine their previous classification. The results of this exercise are shown in Table 18.

Of the 38 FPs, the developer identified 16 suggestions as useful. In seven cases, the developer determined that these cases were actually TP—they had missed these tests during the initial maintenance process. An additional nine suggestions were not correct, but the suggestion was still seen to have some potential benefit, such as suggesting new tests to create.

A further 14 cases were confirmed as FP. In 10 cases, the suggested tests had no relation to the changed code. In the remaining four cases, the explanations from the tool were deemed unreasonable.

Among the remaining eight FPs, seven were actually updated (i.e., were TP), and were missed when creating the dataset. Creating the dataset required determining which test updates were relevant to the code changes. These cases reflect either labeling errors or cases where a test was updated in a later commit. Finally, there was one case where the developer could not understand the test well enough to make a determination.

Based on this assessment, we observe that the results in Section 5.2.1 are potentially conservative. Of the 38 FP cases, 14 were actually correct suggestions—either missed by the developer or during the creation of the dataset. In a further 10 cases, the suggestion was not correct, but still offered some perceived benefit, suggesting tests that should be closely reviewed or newly created.

The final session revolved around the developer's perceptions of the tool and preferences on how it should be tuned. The developer placed some emphasis on recall over precision:

> " It's good to have a balance between recall and precision, but if there's a slight decrease in precision with improved recall, this is more acceptable. It's probably better to optimize recall than precision in our use case. " - The Developer

The developer positively noted the feedback from the prototype, the suggestions of new tests to create, that explanations were provided even when it indicated that no tests needed to be updated, and that the language employed varied with the prototype's level of confidence. However the developer would like to see more explanation of each

suggested test. During the development process of the prototype, we incorporated instructions into the prompt to limit the length of the explanation, based on earlier feedback that indicated that the explanations were too long. This suggests that multiple prompts could be employed based on a users' preference for explanation quantity.

5.3. Discussion

In this section, we answer RQ3 based on the previously-discussed results.

Prototype Performance: The prototype based on test summaries attains the best performance when test cases were updated in the ground truth, and worse performance when maintenance was not performed. The reason, in both cases, is that this prototype makes more positive predictions than the other prototype. Given the large improvement in performance in the former case, we recommend summarizing tests rather than working with raw code.

The initial performance of this prototype is promising, with relatively high recall. A comparable approach is DRIFT, proposed by L. Liu et al. (2023). DRIFT achieved an average accuracy of 0.689, 0.712 for precision, 0.679 for recall, and 0.695 for F1 score. We cannot directly compare performance, as their approach is intended for use in a different context and there are differences in the scope and data used in evaluation. However, we can make a rough comparison—our approach has a higher accuracy and equivalent recall, but lower precision.

The primary factor negatively affecting performance of our prototype is low precision—the prototype yields is more likely to falsely predict that maintenance is required than to miss a test that actually requires maintenance. However, recall is arguably more important than precision for this task, as long as the user is not inundated with too many FP results.

Future work should consider how to improve recall even further, as well as how to improve precision without a loss in recall. Potential paths to better performance include newer and larger models, consideration of models tuned for code tasks, and fine-tuning the model on the Ericsson codebase. We also used a single embedding model, rather than selecting different embedding models for different tasks. The similarity measurement used by the embedding model can impact performance. We can also consider prompt engineering techniques such as query expansion (Jagerman et al., 2023).

Prototype Performance (RQ3): Our prototype based on test summaries has a high recall, showing its potential for this task. However, its precision is relatively low, and could be improved in future work through larger or specialized models, fine-tuning, prompt engineering, and alternative embedding models.

Developer Assessment: In both qualitative explorations, the participants showed interest in the prototype as a "second opinion" on their own judgment, even if the performance was not perfect. The prototype could be executed more quickly than the test suite, and offered an additional data signal that could be incorporated into the existing maintenance workflow.

In the second validation, the developer of the industrial codebase provided additional insight into the quantitative performance results. The developer found that several of the "false positives" were actually correct—either tests that the developer missed initially when performing maintenance, or cases where the ground truth was incorrectly captured. Further, in several cases, even an inaccurate suggestion can offer some benefit. The prototype does not just state the names of test cases, but explains its rationale, indicates its confidence, and may make additional suggestions. This output can offer insight or suggest additional tests to create. Naturally, due to the limited number of participants, these results should be further explored and validated in a larger future study. However, these results offer insights that can inform research in this area.

Developer Assessment (RQ3): The prototype was considered useful, as it is faster to execute than the test suite and offers additional input for test maintenance decisions. Some of the suggested tests were missed by the developer during the original maintenance process, and the suggestions made by the prototype may be useful, even if not fully correct.

6. Threats to validity

External Validity: We conducted a case study at one company, and the prototype was evaluated on a single project at that company. Therefore, aspects of this study may not generalize beyond this specific context:

- Regarding the exploratory study (RQ1–2): The findings of this study come from triangulating three sources—literature review, interview responses, and survey responses. While the interview and survey respondents all come from one company, the literature review reflects a more general view of the topic, helping to partially mitigate the threat. Further, the respondents come from a range of backgrounds and experiences, and many of the testing tools, processes, and technologies employed at Ericsson are also used across the software industry. Therefore, we hypothesize that the findings of the exploratory study can be applied by practitioners at other companies. In future work, however, the results should be validated in a broader context.
- Regarding the tool study (RQ3): The prototype architecture is not closely coupled to any particular technology or project at Ericsson. However, the specific performance results of the tool (i.e., precision, recall) are likely to be project-dependent. Specific characteristics of the project used in the dataset may have influenced the results. For example, a similar distribution of test updates-per-commit may not be found in all repositories. Rather than being interpreted as an absolute indication of performance, the results of RQ3 should be interpreted as a proof-of-concept showing the feasibility of the context. In future work, the tool should be evaluated on a wide variety of projects to better demonstrate its general performance.

Internal Validity: We draw conclusions based on a limited sample that may not be fully representative of the domain. When conducting surveys, we lack control over the response rate and there can be subjectivity in interpreting quantitative answers (Ghazi et al., 2018). Interpretation of qualitative data may also be biased. To mitigate these threats, we performed data triangulation, validating findings across multiple data sources (Runeson and Höst, 2009).

When creating the evaluation dataset, we assumed that tests were updated because of the source changes in the same commit. There may also be cases where tests were changed in an earlier or later commit, leading to false positives that were actually valid suggestions. As a result, prototype performance may be lower than the potential real-world performance.

Construct Validity: There is a risk of participants misunderstanding or misinterpreting questions. This was mitigated by running pilot trials. The wording of some survey and interview questions was clarified after the pilot.

Some source and test code was excluded from the dataset. These exclusions could introduce noise, e.g., tests may be included in the ground truth that were connected to excluded source code, or vice-versa. However, we manually reviewed all exclusions.

As previously discussed, when evaluating RQ3, we did not compare our tool to an existing baseline, as no baseline was found that could be executed under equivalent conditions. Therefore, we emphasize that our tool does not represent the best possible solution for this task. Rather, it should be taken as a demonstration of the feasibility of the concept and as a point of comparison for future work.

7. Implications and recommendations for practitioners

In this section, we summarize implications and recommendations for practitioners based on our results.

Triggers: We identified 37 low-level project changes, as well as seven additional high-level process decisions, that can induce a need for test maintenance. Awareness of these triggers can be very informative for ensuring that test maintenance occurs when needed and for guiding the maintenance process. However, it is also important to understand that a low-level trigger will not always necessitate changes to all tests that execute the affected project code. Rather, the specific nature of the change will determine the subsequence need for test maintenance. Triggers should be taken as a signal to be aware of a potential need for maintenance, but do not offer clear guidance on their own for how to act. LLM-based tools, as well as traditional program analysis methods, can help establish a clearer understanding of how a code change will affect a specific test case.

LLM Deployment in Test Maintenance: It is clear that LLM-based tools can have many applications in the test maintenance process, and their use should be widely explored. Many of these uses are obvious, like test identification, generation, and repair. However, LLMs should not just be deployed as standalone, fully autonomous tools. Multiple participants, when asked about tool support, envisioned a scenario where LLMs acted as a ''pair tester''—a partner that offered suggestions, templates, and a tutor to explain the project or testing concepts. This type of use could accelerate and improve the quality of the work of human testers. In general, LLM-based tools should be deployed with human oversight to ensure long-term test suite quality.

Our Proof-of-Concept: We envision two primary use-cases for the prototype we presented in this study. First, the tool could be manually invoked after the production code is updated, but before it is committed to the repository, by the developer. The developer could use its recommendations, along with their own intuition or other tools, to perform any required test maintenance. Second, the tool could be invoked in the continuous integration pipeline after the production changes are committed (along with any planned test maintenance) as a way to identify potentially missed maintenance cases. Its recommendations could then be offered as advice to the developer.

A natural question is whether the results of our proof-of-concept are good enough to use in practice. To contextualize the results, an average of eight test cases are updated in each positive commit in the ground truth. When the tool is executed for a commit, it will generally yield five correct recommendations (TP) and miss three tests that should have been identified (FN). These results are a promising start for research on LLM support in test maintenance. However, we do not believe that the tool is sufficiently trustworthy to rely on in practice at this stage.

The main barrier to relying on this tool is the false negatives—the missed cases. Testers may be able to tolerate false positives, but the false negatives must be eliminated to the extent possible. That said, the false positive rate should also be improved as well; otherwise, a user may waste time inspecting misleading suggestions. Both aspects must be improved in future iterations of this or other tools. Therefore, our recommendation is to use tools like our prototype as a way to accelerate the test maintenance process, but not as the sole way to determine what tests to update and not as a replacement for human judgment. The predictions made by the tool should be taken as recommendations that are validated by the developer.

8. Conclusions

In this study, we explore the capabilities of LLMs and LLM agents to support test maintenance. We have identified 44 low and high-level triggers that can indicate a need for test maintenance. Based on these triggers and other data signals, we speculate that LLMs and LLM agents could perform or assist with test maintenance activities, including code generation or modification, acting as a conversational partner, and code analysis or documentation.

When deploying LLMs, organizations should consider model performance, whether to deploy in-house or externally, copyright and ownership, how to incorporate organizational data (e.g., fine tuning or RAG), how to establish and manage expectations, which tasks LLMs should support, and how humans and LLMs should interact when performing these tasks.

Our prototype based on test summaries has a high recall, demonstrating that LLM agents could play a practical role in test maintenance. When inspecting a subset of false positives from the prototype, we discovered that some of these tests were missed by the developer during the original maintenance process, and that the suggestions made by the prototype may be useful, even if not fully correct. Still, future research should consider how the precision of the prototype could be improved as well as a larger pool of repositories and programming languages.

Collectively, these contributions advance our theoretical and practical understanding of how LLMs can be deployed to benefit industrial test maintenance processes. The discussion of triggers, actions, and the tool results can inform early industrial application of LLMs in test maintenance. In future work, we are also interested in developing detailed guidelines, checklists, and tool support for practitioners in this area. However, this requires further empirical and theoretical exploration, as well as the development of tools that implement additional test maintenance actions.

CRediT authorship contribution statement

Jingxiong Liu: Writing – review & editing, Writing – original draft, Validation, Supervision, Software, Methodology, Investigation, Data curation. **Ludvig Lemner:** Writing – original draft, Visualization, Methodology, Investigation, Data curation, Conceptualization. **Linnea Wahlgren:** Writing – original draft, Visualization, Methodology, Investigation, Data curation, Conceptualization. **Gregory Gay:** Writing – review & editing, Writing – original draft, Supervision, Project administration, Methodology, Funding acquisition, Conceptualization. **Nasser Mohammadiha:** Writing – review & editing, Supervision, Resources, Methodology, Conceptualization. **Joakim Wennerberg:** Writing – review & editing, Supervision, Resources.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Gregory Gay reports financial support was provided by Knut and Alice Wallenberg Foundation. Gregory Gay reports financial support was provided by Software Center. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

Support for this research was provided by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, as well as Software Center project 68 ''Trustworthy and Responsible AI-Centric Test Engineering (TRACE)''.

Data availability

The data that has been used is confidential.

References

Alégroth, E., Feldt, R., Kolström, P., 2016. Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. Inf. Softw. Technol. 73, 66–80.

Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P., Bertolino, A., et al., 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. 86 (8), 1978–2001.

Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S., 2014. The oracle problem in software testing: A survey. IEEE Trans. Softw. Eng. 41 (5), 507–525.

Berglund, L., Grube, T., Gay, G., de Oliveira Neto, F.G., Platis, D., 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, pp. 410–421.

Bommasani, R., Hudson, D.A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M.S., Bohg, J., Bosselut, A., Brunskill, E., et al., 2021. On the opportunities and risks of foundation models. arXiv preprint arXiv:2108.07258.

Braun, V., Clarke, V., 2006. Using thematic analysis in psychology. Qual. Res. Psychol. 3 (2), 77–101.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al., 2020. Language models are few-shot learners. Adv. Neural Inf. Process. Syst. 33, 1877–1901.

Cao, Y., Li, S., Liu, Y., Yan, Z., Dai, Y., Yu, P.S., Sun, L., 2023. A comprehensive survey of ai-generated content (aigc): A history of generative ai from gan to chatgpt. arXiv preprint arXiv:2303.04226.

Chase, H., 2022. LangChain. URL: https://www.langchain.com/.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W., 2021. Evaluating large language models trained on code. arXiv:2107.03374.

Chen, J., Xiao, S., Zhang, P., Luo, K., Lian, D., Liu, Z., 2024. BGE M3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. arXiv:2402.03216.

Chi, J., Wang, X., Huang, Y., Yu, L., Cui, D., Sun, J., Sun, J., 2025. REACCEPT: Automated co-evolution of production and test code based on dynamic validation and large language models. Proc. ACM Softw. Eng. 2 (ISSTA), http://dx.doi.org/10.1145/3728930.

Chiang, W.-L., Zheng, L., Sheng, Y., Angelopoulos, A.N., Li, T., Li, D., Zhang, H., Zhu, B., Jordan, M., Gonzalez, J.E., et al., 2024. Chatbot arena: An open platform for evaluating LLMs by human preference. arXiv preprint arXiv:2403.04132.

Deng, Y., Xia, C.S., Yang, C., Zhang, S.D., Yang, S., Zhang, L., 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. pp. 1–13.

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., Jégou, H., 2024. The faiss library. arXiv preprint arXiv:2401.08281.

Ens, B., Rea, D., Shpaner, R., Hemmati, H., Young, J.E., Irani, P., 2014. Chronotwigger: A visual analytics tool for understanding source and test co-evolution. In: 2014 Second IEEE Working Conference on Software Visualization. IEEE, pp. 117–126.

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., Zhang, J.M., 2023. Large language models for software engineering: Survey and open problems. arXiv preprint arXiv:2310.03533.

Feldt, R., Kang, S., Yoon, J., Yoo, S., 2023. Towards autonomous testing agents via conversational large language models. In: 2023 38th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1688–1693.

Gay, G., 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.

Geng, M., Wang, S., Dong, D., Wang, H., Li, G., Jin, Z., Mao, X., Liao, X., 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning.

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

Ghazi, A.N., Petersen, K., Reddy, S.S.V.R., Nekkanti, H., 2018. Survey research in software engineering: Problems and mitigation strategies. IEEE Access 7, 24703–24718.

Gonzalez, D., Santos, J.C., Popovich, A., Mirakhorli, M., Nagappan, M., 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, pp. 391–401.

Hadi, M.U., Qureshi, R., Shah, A., Irfan, M., Zafar, A., Shaikh, M.B., Akhtar, N., Wu, J., Mirjalili, S., et al., 2023. Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects. Authorea Preprints.

Han, X., Zhang, Z., Ding, N., Gu, Y., Liu, X., Huo, Y., Qiu, J., Yao, Y., Zhang, A., Zhang, L., et al., 2021. Pre-trained models: Past, present and future. AI Open 2, 225–250.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., Wang, H., 2023. Large language models for software engineering: A systematic literature review. arXiv preprint arXiv:2308.10620.

Hu, X., Liu, Z., Xia, X., Liu, Z., Xu, T., Yang, X., 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, pp. 1111–1122.

Huang, Y., Tang, Z., Chen, X., Zhou, X., 2024. Towards automatically identifying the co-change of production and test code. Softw. Test. Verif. Reliab. e1870.

Imtiaz, J., Sherin, S., Khan, M.U., Iqbal, M.Z., 2019. A systematic literature review of test breakage prevention and repair techniques. Inf. Softw. Technol. 113, 1–19.

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.

Jagerman, R., Zhuang, H., Qin, Z., Wang, X., Bendersky, M., 2023. Query expansion by prompting large language models. URL: https://arxiv.org/abs/2305.03653. arXiv:2305.03653.

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y.J., Madotto, A., Fung, P., 2023. Survey of hallucination in natural language generation. ACM Comput. Surv. 55 (12), 1–38.

Kang, S., Yoon, J., Yoo, S., 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, pp. 2312–2323.

Kasunic, M., 2005. Designing an effective survey.

Keele, S., et al., 2007. Guidelines for performing systematic literature reviews in software engineering.

Kitai, T., Aman, H., Amasaki, S., Yokogawa, T., Kawahara, M., 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, pp. 120–124.

Kochhar, P.S., Xia, X., Lo, D., 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, pp. 61–70.

Lemieux, C., Inala, J.P., Lahiri, S.K., Sen, S., 2023. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In: International Conference on Software Engineering. ICSE.

Levin, S., Yehudai, A., 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, pp. 35–46.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al., 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. Adv. Neural Inf. Process. Syst. 33, 9459–9474.

Li, H., Deng, G., Liu, Y., Wang, K., Li, Y., Zhang, T., Liu, Y., Xu, G., Xu, G., Wang, H., 2024. Digger: Detecting copyright content mis-usage in large language model training. URL: https://arxiv.org/abs/2401.00676. arXiv:2401.00676.

Lin, J., Liu, Y., Zeng, Q., Jiang, M., Cleland-Huang, J., 2021. Traceability transformed: Generating more accurate links with pre-trained bert models. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, IEEE, pp. 324–335.

Liu, Z., Chen, C., Wang, J., Che, X., Huang, Y., Hu, J., Wang, Q., 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, pp. 1355–1367.

Liu, L., Wang, S., Liu, Y., Deng, J., Liu, S., 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. pp. 227–237.

Liu, J., Xia, C.S., Wang, Y., Zhang, L., 2024a. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. Adv. Neural Inf. Process. Syst. 36.

Liu, J., Yan, J., Xie, Y., Yan, J., Zhang, J., 2024b. Fix the tests: Augmenting LLMs to repair test cases with static collector and neural reranker. In: 2024 IEEE 35th International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 367–378.

Lu, J., Yu, L., Li, X., Yang, L., Zuo, C., 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, pp. 647–658.

Mandvikar, S., 2023. Factors to consider when selecting a large language model: A comparative analysis. Int. J. Intell. Autom. Comput. 6 (3), 37–40.

Marsavina, C., Romano, D., Zaidman, A., 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, pp. 195–204.

Mirzaaghaei, M., 2011. Automatic test suite evolution. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 396–399.

Mirzaaghaei, M., Pastore, F., Pezze, M., 2010. Automatically repairing test cases for evolving method declarations. In: 2010 Ieee International Conference on Software Maintenance. IEEE, pp. 1–5.

Mirzaaghaei, M., Pastore, F., Pezzè, M., 2012. Supporting test suite evolution through test case adaptation. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE, pp. 231–240.

Mirzaaghaei, M., Pastore, F., Pezzè, M., 2014. Automatic test case evolution. Softw. Test. Verif. Reliab. 24 (5), 386–411.

Myers, G.J., Badgett, T., Thomas, T.M., Sandler, C., 2004. The Art of Software Testing, vol. 2, Wiley Online Library.

Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., Myers, B., 2024. Using an llm to help with code understanding. In: 2024 IEEE/ACM 46th International Conference on Software Engineering. ICSE, IEEE Computer Society, p. 881.

OpenAI (2023), 2023. Gpt-4 technical report. arXiv preprint arXiv:2303.08774.

Pinto, L.S., Sinha, S., Orso, A., 2012. Understanding myths and realities of test-suite evolution. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. pp. 1–11.

Rasheed, Z., Waseem, M., Saari, M., Systä, K., Abrahamsson, P., 2024. Codepori: Large scale model for autonomous software development by using multi-agents. arXiv preprint arXiv:2402.01411.

Reich, P., Maalei, W., 2023. Testability refactoring in pull requests: Patterns and trends. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 1508–1519.

Ross, S.I., Martinez, F., Houde, S., Muller, M., Weisz, J.D., 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. pp. 491–514.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al., 2023. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950.

Runeson, P., Höst, M., 2009. Guidelines for conducting and reporting case study research in software engineering. Empir. Softw. Eng. 14, 131–164.

Sarkar, A., Gordon, A.D., Negreanu, C., Poelitz, C., Ragavan, S.S., Zorn, B., 2022. What is it like to program with artificial intelligence? arXiv preprint arXiv:2208.06213.

Schäfer, M., Nadi, S., Eghbali, A., Tip, F., 2023. An empirical evaluation of using large language models for automated unit test generation. IEEE Trans. Softw. Eng..

Shanahan, M., 2024. Talking about large language models. Commun. ACM 67 (2), 68–79.

Shimmi, S., Rahimi, M., 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, pp. 116–127.

Skoglund, M., Runeson, P., 2004. A case study on regression test suite maintenance in system evolution. In: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings. IEEE, pp. 438–442.

Sneed, H.M., 2004. A cost model for software maintenance & evolution. In: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings. IEEE, pp. 264–273.

Sobania, D., Briesch, M., Hanna, C., Petke, J., 2023. An analysis of the automatic bug fixing performance of chatgpt. In: 2023 IEEE/ACM International Workshop on Automated Program Repair. APR, IEEE, pp. 23–30.

Sohn, J., Papadakis, M., 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, pp. 133–144.

Sun, W., Yan, M., Liu, Z., Xia, X., Lei, Y., Lo, D., 2023. Revisiting the identification of the co-evolution of production and test code. ACM Trans. Softw. Eng. Methodol. 32 (6), 1–37.

Surameery, N.M.S., Shakor, M.Y., 2023. Use chat gpt to solve programming bugs. Int. J. Inf. Technol. Comput. Eng. (IJITC) (ISSN: 2455-5290) 3 (01), 17–22.

Tsigkanos, C., Rani, P., Müller, S., Kehrer, T., 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, pp. 678–682.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2016. An empirical investigation into the nature of test smells. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 4–15.

Umar, M.A., Zhanfang, C., 2019. A study of automated software testing: Automation tools and frameworks. Int. J. Comput. Sci. Eng. (IJCSE) 6, 217–225.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. Adv. Neural Inf. Process. Syst. 30.

Vidács, L., Pinzger, M., 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, pp. 31–36.

Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., Wang, Q., 2024. Software testing with large language models: Survey, landscape, and vision. IEEE Trans. Softw. Eng..

Wang, S., Wen, M., Liu, Y., Wang, Y., Wu, R., 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, pp. 272–283.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q.V., Zhou, D., et al., 2022. Chain-of-thought prompting elicits reasoning in large language models. Adv. Neural Inf. Process. Syst. 35, 24824–24837.

White, J., Hays, S., Fu, Q., Spencer-Smith, J., Schmidt, D.C., 2023. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. arXiv preprint arXiv:2303.07839.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., Wang, C., 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. arXiv preprint arXiv:2308.08155.

Xia, C.S., Wei, Y., Zhang, L., 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.

Yang, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Li, C., Liu, D., Huang, F., Wei, H., Lin, H., Yang, J., Tu, J., Zhang, J., Yang, J., Yang, J., Zhou, J., Lin, J., Dang, K., Lu, K., Bao, K., Yang, K., Yu, L., Li, M., Xue, M., Zhang, P., Zhu, Q., Men, R., Lin, R., Li, T., Xia, T., Ren, X., Ren, X., Fan, Y., Su, Y., Zhang, Y., Wan, Y., Liu, Y., Cui, Z., Zhang, Z., Qiu, Z., 2024. Qwen2.5 technical report. arXiv preprint arXiv:2412.15115.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y., 2022. React: Synergizing reasoning and acting in language models. arXiv preprint arXiv:2210.03629.

Yoon, J., Feldt, R., Yoo, S., 2023. Autonomous large language model agents enabling intent-driven mobile GUI testing. arXiv preprint arXiv:2311.08649.

Yu, S., Fang, C., Ling, Y., Wu, C., Chen, Z., 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, pp. 206–217.

Yuan, Z., Lou, Y., Liu, M., Ding, S., Wang, K., Chen, Y., Peng, X., 2023. No more manual tests? Evaluating and improving ChatGPT for unit test generation. arXiv preprint arXiv:2305.04207.

Zhang, Z., Chen, C., Liu, B., Liao, C., Gong, Z., Yu, H., Li, J., Wang, R., 2023. Unifying the perspectives of nlp and software engineering: A survey on language models for code. arXiv preprint arXiv:2311.07989.

Zhang, Q., Fang, C., Xie, Y., Zhang, Y., Yang, Y., Sun, W., Yu, S., Chen, Z., 2023. A survey on large language models for software engineering. arXiv preprint arXiv:2312.15223.

Zhao, W.X., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., Min, Y., Zhang, B., Zhang, J., Dong, Z., et al., 2023. A survey of large language models. arXiv preprint arXiv:2303.18223.

Zheng, Z., Ning, K., Chen, J., Wang, Y., Chen, W., Guo, L., Wang, W., 2023a. Towards an understanding of large language models in software engineering tasks. arXiv preprint arXiv:2308.11396.

Zheng, Z., Ning, K., Wang, Y., Zhang, J., Zheng, D., Ye, M., Chen, J., 2023b. A survey of large language models for code: Evolution, benchmarking, and future trends. arXiv preprint arXiv:2311.10372.

Zhu, J., Xiao, G., Zheng, Z., Sui, Y., 2022. Enhancing traceability link recovery with unlabeled data. In: 2022 IEEE 33rd International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 446–457.