CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Predicting the Need for Test Maintenance Using LLM Agents

Applying Test Maintenance Factors to Changes in Production Code to Identify If and Where Test Cases Need to Be Updated

Master's Thesis in Computer Science and Engineering

LUDVIG LEMNER
LINNEA WAHLGREN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Predicting the Need for Test Maintenance Using LLM Agents

Applying Test Maintenance Factors to Changes in Production Code to Identify If and Where Test Cases Need to Be Updated

LUDVIG LEMNER
LINNEA WAHLGREN

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Predicting the Need for Test Maintenance Using LLM Agents
Applying Test Maintenance Factors to Changes in Production Code to Identify If and Where Test Cases Need to Be Updated
LUDIVG LEMNER
LINNEA WAHLGREN

Supervisor: Gregory Gay, Computer Science and Engineering
Advisors: Nasser Mohammadiha, Ericsson
Roy Liu, Ericsson
Joakim Wennerberg, Ericsson
Examiner: Robert Feldt, Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Predicting the Need for Test Maintenance Using LLM Agents
Applying Test Maintenance Factors to Changes in Production Code to Identify If and Where Test Cases Need to Be Updated
LUDVIG LEMNER
LINNEA WAHLGREN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Test maintenance, the act of modifying and updating test cases to ensure they keep up with the changes made in the production code, is a necessary but time-consuming and effort-intensive activity. One way to alleviate these efforts is by automating parts of the test maintenance process, however, setting up and maintaining automation tools can be time-consuming as well. Generative AI and Large Language Models (LLMs) offer new avenues for automation and lessening the test maintenance problem. One of these is through LLM agents, sophisticated AI systems that reason, plan, and use tools to help it achieve its goals.

This thesis was conducted as an exploratory case study at Ericsson and investigated how generative AI can help ease test maintenance, specifically how LLM agents can be used to predict test maintenance. The thesis had three phases: Identifying factors that trigger test maintenance; exploring the capabilities of generative AI and how it might be used to help with test maintenance; and, using the results from the two previous phases, building a prototype to help predict if and if so where test maintenance is needed based on changes to the production code. We identified 40 factors that when changed in production code cause a need for test maintenance, and successfully demonstrated how they can be used as triggers in a setup with LLM agents. Out of the four different setups that were evaluated, we found that using multiple LLM agents coordinated by a planning agent, and giving these access to both production code and natural language summaries of test cases, worked best. We also, through a thorough literature review, identify test maintenance actions LLMs can take and help with. These demonstrate both the possibilities and current limitations of LLMs when it comes to test maintenance, and the results highlight how—though a large focus of LLM studies within software engineering has focused on code generation—the capabilities of LLMs are much broader. This study provides examples of how LLM agents can be used more broadly and all-encompassingly.

Keywords: Software engineering (SE), test maintenance, large language model (LLM), LLM agent.

Acknowledgements

We would like to thank our academic supervisor Gregory Gay and our industrial supervisors at Ericsson: Nasser Mohammadiha, Roy Liu, and Joakim Wennerberg. A big thank you for your guidance and patience, and for always taking the time to support us. We would also like to thank all other Ericsson employees who kindly helped with and participated in the study, without which this thesis would not have been possible.

Ludvig Lemner & Linnea Wahlgren, Gothenburg, June 2024

Contents

List of Figures

List of Tables

1

Introduction

Software testing is a necessary but expensive activity, that can account for up to half of the total development cost of a system [1]. As the system lives on and evolves, test maintenance is needed to ensure the relevant tests are updated and new tests are created. This too requires substantial effort and resources. One solution to this is automation, which can be an important tool in reducing these costs, and could furthermore improve the resulting quality of the production code. Developers can save time through automation, which can be spent on other critical challenges instead, thereby reducing costs. However, to be able to automate test maintenance activities a proper understanding of these activities is required, and even then automation can be trickier than expected, and an effort-expensive process as well, e.g. test automation scripts may require a lot of upkeep. Because of this, many test maintenance activities are still performed manually [2].

These last few years have seen a rise in the use of generative AI, which is AI capable of generating content based on their input data, often text or images [3]. Large Language Models (LLMs) are a form of generative AI that has been trained on massive datasets, and outputs text and/or code (depending on the language), the most famous example at the time of writing perhaps being GPT-3.5 and GPT-4, which are used in ChatGPT [4]. LLMs have been applied to various aspects of software development (e.g. test case generation [5], documentation [6], refactoring [7], and automated program repair (APR) [8]) with varying degrees of success, often with the purpose to help developers by automating these activities or parts of the activities [7, 9]. LLMs have further been used as general assistants to developers, helping not only by automating activities but also by giving advice and helping developers work through problems [10] through continuous conversation.

This thesis seeks to understand how LLMs can be used to help with test maintenance, however, there are some known shortcomings when using LLMs. Due to data privacy concerns, organisations may not wish to use commercial LLMs, and may instead adopt open-source ones. Due to the high power consumption rate, they also tend to use medium-sized models, which generally do not perform as well as the larger models [11]. LLMs may also suffer from hallucinations, i.e. generating output that is untrue while presenting it as correct, when put in unfamiliar situations not covered by their training data, such as when asked about an organisation's internal data which will naturally be unknown to the LLM [12]. Another problem is that

LLMs may struggle with large contexts, or breaking down tasks into sub-tasks to more efficiently handle them [13]. One solution to these problems is LLM agents. An LLM agent is an advanced AI system with an LLM as its core, further equipped with tools or frameworks that allow it to reason, retain memory, plan, and perceive or interact with chosen parts of its environment [14, 15]. By, for example, giving an LLM access to an organisation's code repository it may more accurately answer how one change in the production code affects the remaining code base. In addition, different agents taking on different roles can be made to work together as multi-agents, similarly to how a software development team works in reality [16].

This thesis was conducted as a case study of test maintenance at Ericsson, a Swedish telecommunications company. Based on the found test maintenance problems, it was investigated how LLM agents could help alleviate them, specifically by predicting if and where test maintenance would be needed. Multiple steps were taken to analyse how LLMs can assist with test maintenance. First, a literature review was conducted to see which changes in production code lead to a need for test maintenance. This was accompanied by a survey and interviews with Ericsson employees to understand better the test maintenance problem, as well as how a solution might best fit with their practices. To complement this, a literature review of the current capabilities and uses of LLMs within software testing was conducted as well, to see what could apply to test maintenance. Based on the combination of these results, a setup with LLM agents was built to help predict whether test maintenance was needed because of changes to the production code, and if so, to which test cases.

1.1 Problem Description

Testing in software development is generally an expensive process. This is true for both the monetary cost as well as the effort required in the process itself [1]. This thesis project will be conducted in collaboration with Ericsson. They have many interests and products but what will be focused on in this study is their test suites and maintenance that is being performed on those test suites.

There is an initial cost associated with creating the tests and test suites [17], but a large part of the cost of testing comes from the maintenance aspect. Keeping test suites updated through changes in the production code or dependencies over time is challenging and expensive. It generally takes longer to modify existing methods and classes, whether to rectify faults or change their functionality, compared to adding new ones [18]. Additionally, maintenance includes further activities beyond maintaining the test suites. Maintenance is also affected by the process, environment, personnel, and the tools available.

Test maintenance encompasses a multitude of different potential activities that are both complex and time-consuming. These activities include, but are not limited to: Error correction, reverse engineering, program comprehension, re-engineering, impact analysis, repository construction, functional enhancements, renovation, migration, integration, optimization, and adaptation. During test maintenance, the

developers have to keep in mind the requirement specifications, the design documents, the test cases and the database schemas [18].

To reduce the expenses that testing can require, automation is an excellent tool that is being used more and more on a wider scale [19]. By automating large areas of the testing process, critical problems can be handled by developers with less focus laid on other tasks.

This study aims to tackle the high expense of test maintenance by utilising generative AI to reduce the costs involved in the maintenance process, specifically evolving test cases. Having LLM agents provide suggestions for if and where test maintenance is needed to the developers at Ericsson could help make the maintenance process more manageable and cost-effective.

LLMs and generative AIs are especially adept at analysing large amounts of data and based on that data give suggestions for best practices [20]. This particular ability we feel will be especially helpful in the areas of test maintenance that are hard to automate and that at the moment require human involvement. By partially automating tasks it could ease the burden on the developers and reduce the effort needed to complete their tasks. This especially goes for LLM agents, which are uniquely equipped to understand a changing code base, because of their access to tools that let them process new and changing information about the surrounding environment.

1.2 Purpose of the Study

The purpose of the study is to explore how LLM agents can be used to make test maintenance activities faster by allowing developers to more easily understand if and where changes are needed and being able to apply these changes in a way that ensures the test's relevance and readability. In particular, we want to investigate how LLM agents can help developers by predicting the need for test maintenance while ensuring the quality of the testing process is maintained.

First, we mapped out the existing research on test maintenance, particularly factors in the source code-under-test that indicate that a test needs maintenance. This literature review is then supplemented with corresponding data collected from developers and employees at Ericsson, that is examples of when tests have been updated and explanations of why. From the analysis of these results, the study maps out which problems in test maintenance can and are suitable to be partially addressed with LLMs. Factors concerning test maintenance have mostly been explored in general terms [1, 21], and less in which specific parts of the source code-under-test indicate a need for maintenance. We expect this part of the study to benefit both researchers and practitioners by exploring this area since a better understanding of the problem allows time and effort to be spent more effectively.

The second phase of the study built upon what had previously been found, and explored how the solution of using a setup with LLM agents may help with test

maintenance. It investigated how an agent with knowledge of both the production code and test code might help with keeping the test code up-to-date when the production code has been changed. Four different setups were investigated: Two variants where the output of one LLM or LLM agent was provided as input to the next, creating a chain of LLMs and LLM agents, and two multi-agent setups with a planning agent to coordinate the work. The purpose of the second phase is to evaluate the viability of the setup with LLM agents solution. This solution aims to help practitioners save time and effort during test maintenance activities, and by doing this increase the overall quality of software projects.

It should be noted that not all results from the first part of the study went towards building the test maintenance setup with LLM agents. Some are also used more as a basis for discussion by, based on the identified test maintenance problems, speculating about how agents could be incorporated into more parts of the entire development process to help more continuously and all-encompassingly with test maintenance. For example, it was considered what higher-level use cases, e.g. a change in requirements, might look like, to see how an agent might assist with and support effective test maintenance throughout the whole production chain.

1.3 Significance of the Study

This thesis makes both scientific and practical contributions. It firstly contributes both high and low-level factors that act as triggers for test maintenance. The low-level triggers, specific changes in production code, are collected and organised from existing literature, while the high-level triggers, reasons to change production code that also leads to a need to change test code, were gathered from interviews and a survey. Previous studies have focused more on general factors regarding test maintenance. They have mapped out factors that complicate maintenance, or indicate a need for maintenance, factors such as the size of the test suite and the understandability of the test. This study will have a larger focus on which specific changes in the production code indicate a need for maintenance, which makes it scientifically significant, as it is exploring and contributing to a less explored area.

The study secondly contributes to an understanding of how and where generative AI can be used to help simplify the test maintenance problem. This is significant to practitioners as test maintenance is a time-consuming and labour-intensive activity, and a further understanding of the activity can help alleviate this. Most research on LLMs within software testing focuses on test generation, however, LLMs have broad uses and this study helps both practitioners and researchers better understand how LLMs can be used for predicting test maintenance, which is a less studied area. It is also significant as LLMs may already be used informally by practitioners for this, but this study would help formalise that knowledge, and would also further it.

Lastly, this study sets up a proof of concept of how LLM agents can use the previously mentioned test maintenance triggers, and by using them help make test maintenance more efficient. This is significant to practitioners as a significant amount

of time can be saved by providing automation to help with test maintenance activities. It is significant to research since there, to our knowledge, exists limited or no research on how LLM agents can help with test maintenance.

1.4 Thesis Outline

This thesis is organised as follows:

Chapter 2: Background introduces relevant concepts for software testing and generative AI and introduces necessary terminology.

Chapter 3: Related Work explores existing research on software testing, generative AI, and LLM agents and positions the thesis' work to it.

Chapter 4: Methods lists the research questions, and describes the steps taken to answer them. In particular, it describes the design of the literature reviews, the interviews, the survey, and the architecture of the setup with LLM agents as well as the steps taken to evaluate the agent.

Chapter 5: Results describes the results of the sub-tasks.

Chapter 6: Discussion provides answers to the research questions, examines how the results fit with and relate to the research discussed in related work, proposes future research paths and discusses threats to validity.

Chapter 7: Conclusion summarises the study and its results.

2

Background

The background chapter will lay the foundation for the chapters to come by explaining the necessary knowledge to the reader and introducing essential concepts. It will first look at software testing, including test management and test maintenance. Secondly, it will look at generative AI and LLMs, before finishing with LLM agents.

2.1 Software Testing

Software testing is the act of verifying and validating a software system, i.e. ensuring both that it fulfils its stated requirements and its high-level purpose. It is a necessary activity to ensure the quality of the product and is generally considered one of the most important parts of software development [22].

In general, testing can be split into functional and non-functional testing. Non-functional testing tests how well the system performs, such as its responsiveness, usability, and stability [23]. In contrast, Functional testing tests the way the system operates by comparing the result of input and execution conditions to an expected output [24]. It can be split into three levels: Unit testing (sometimes also called module testing), which generally tests a single unit of code or piece of functionality, such as a class or a method [25]; Integration testing, which tests the interaction between two or more separate software components [26]; And system testing, where the entire software system working together is tested [25].

A test case is a specification of all the relevant information used to see that the program fulfils an intended objective [24], and collection of test cases in turn make up a test suite [27]. A test case is comprised of many different parts, some of the most important of which are:

Test oracle: A mechanism for determining if the output of the test is correct [28]. Some examples include oracles derived from external information, such as requirements documentation, and human knowledge on how the program should behave.

Test steps: Describes and outlines the steps that should be taken to run the test.

Initialization: The step that sets everything up in preparation for the test case, such as a separate environment to ensure the test can be run in isolation without affecting or being affected by the rest of the system. It also includes defining and declaring variables, and similar preparations.

Teardown: The step that, after the test has been run, removes all the temporary equipment used by the test, such as data structures.

Input: The data provided at the beginning of the test, i.e. what is fed into the test case. The input will determine which path is taken through the program, and it is therefore important to test with different inputs to ensure many different scenarios are tested.

Software testing is also an expensive process that can take up to 50% of both time and cost of the development of a system [25], which makes tools that can assist with the process or automate parts of it desirable. The need for these tools is widely recognised [29], and it is known that automation can help reduce time spent on testing [30]. Many types of automation tools (e.g. test generation tools and test code coverage tools) do exist [30], however, the tools themselves sometimes require significant time and effort to maintain [1].

2.1.1 Test Management

Test management refers to actively working to ensure the quality of the test suite by updating and evaluating it, to in turn be able to ensure the quality of the production code. Management includes several subareas, such as test maintenance, test automation, and test generation. As test maintenance is the area of this thesis it will be further explained in Section 2.1.2, while test automation and generation will be briefly explored here.

Test management is an expensive, time-consuming activity, and the need to alleviate the process through automation is widely accepted [29]. However, test management activities are still generally performed manually [2]. Best practices for test management are not widely established [1], but it is known that automation can play a vital role in reducing the time spent on testing [30]. Test automation refers to expressing tests as executable code, then using an automated system (e.g. a CI/CD pipeline) to execute the tests and process the results of the test execution [30]. Even though automation decreases time and effort, maintaining automation scripts still requires significant effort [1]. Automation also requires significant upfront investment and needs maintenance throughout the program's life-cycle [31].

In addition to automating test execution, automation can also be used to generate test cases, especially test input. Since manually creating test cases can be among the most labour-intensive parts of software testing, automatic test case generation is one of the more well-researched areas within software testing. Many different techniques to generate test cases exist: model-based testing, combinatorial testing, and search-based testing are a few of them [32]. These have recently been joined by using LLMs

to generate test cases [3]. Despite the many different approaches, difficulties still exist in ensuring the generated tests' maintainability and readability [33, 34]. Other than test cases and input, there have also been efforts made to generate test oracles, though they remain particularly difficult to generate automatically [28].

2.1.2 Test Maintenance

Test maintenance refers to the act of updating the test suite as the production code changes and evolves [1]. Systems evolve as new features are added, requirements are changed, and faults are discovered. As the production code is modified to accommodate this, the test code may need to be changed as well, to ensure that test results are accurate for the current system behaviour. This includes adding new test cases, repairing existing ones, and removing those that are no longer relevant.

Test maintenance is understood to be an important part of quality assurance, but has not received as much attention as the insurance of the quality of production code [35]. An example of this is the study of test smells, which despite their known effect on the maintainability of the test suite have received significantly less attention than their counterpart code smells [36]. This is despite reports of test maintenance accounting for up to 60% of the total time spent on testing in a week [1]. In other words, test maintenance is an important research area to further explore.

2.2 Generative AI

Generative AI is a form of AI that can understand the intent of a given instruction, and based on this intent generate output in the form of media such as text, images, or music, to name a few [20]. The form of the instruction and output differs between different types of generative AI. Some generative AIs respond to user prompts while others analyse a piece of received media, to name a few. Two examples of generative AI are large language models (LLMs), which input text in the form of a prompt and output text in the form of a response, and text-to-image AIs, which input a text prompt and output an image based on the prompt. A prompt is an input the user gives the LLM when interacting with it, and can for example be questions or instructions.

Most generative AI uses a transformer-based architecture, which is based entirely on attention mechanisms. These imitate the cognitive attention (i.e. the ability to focus on select stimuli) seen in humans [37]. The development of the transformer architecture was successfully combined with pre-trained systems, pre-training referring to training the model on a diverse data set to learn general patterns and features [38]. The model can then be fine-tuned to a specific domain or task, which is called transfer learning, as the model can transfer the general knowledge it learned during pre-training to another domain [39]. These powerful pre-trained models are called foundation models and can often be adapted to a wide variety of areas, such as software development, education, and healthcare [40].

2.2.1 Large Language Models

One form of generative AI is Large Language models (LLMs), which include famous examples GPT-3.5 and GPT-4 that are used within ChatGPT [4, 41]. LLMs are made for natural language processing (NLP) tasks, such as text generation. They take text as input and generate text as output by iteratively predicting the next token or word in a sequence to form a cohesive text [42].

One form of text-based language is code. Because of their ability to understand and output both natural language and code, LLMs are well-suited for software development. Within test maintenance, the most interesting applications are perhaps the generation of test cases and input, but LLMs have also been used for automated program repair, code review, and as a conversational programmer's assistant, to name a few [43, 44, 10]. Their strength lies in their versatile nature, their ability to generate code, then reason about said code and allow the user to ask questions about it. They can be used the other way around as well; help their user reason about a problem or set up a plan to tackle it, then based on that help the user generate or look over code. However, even with LLM's advanced reasoning capabilities there still exist problems.

LLMs are known to struggle with hallucinations. A hallucination, in the context of LLMs, is the LLM generating text that appears to be correct and fluent, but in reality, is nonsensical and unfaithful to the data the LLM was trained on. Even when overlooking how this affects the LLM's performance, hallucinations also cause trust issues for the user and can pose safety risks if the user acts on the hallucinated output [12].

Another problem is that at the moment there exist problems with how to rigorously and extensively evaluate and judge LLMs. Firstly, there is a lack of benchmark datasets to use, especially when testing more niche applications such as program repair. These datasets may not have been designed for testing LLMs either. Secondly is the problem of data leakage, where the LLMs may have seen the benchmarks during training, which means reports of LLMs performance may be misleading [11].

Applying LLMs in real-world applications is also not without its share of problems. Organisations may shy away from using commercial LLMs because of data privacy concerns and would prefer to use open-source models instead. These models can then be fine-tuned with the organisation's internal data. However, building these datasets for fine-tuning can take a lot of effort, both time and labour-wise. Organisations may also pay attention to computational power or energy consumption, and therefore choose a medium-sized model, with which it is harder to get state-of-the-art performance, even with fine-tuning [11].

An LLM model and a human may not understand a prompt the same way, which has given rise to the field of prompt engineering, i.e. how to best formulate the prompt sent to the LLM to get the user's desired output [45]. This can be likened to a kind of natural language programming, a way to steer the generated output of the LLM [46]. Prompt engineering is possible because of in-context learning, which

is defined by Brown et al. as *"a paradigm that allows language models to learn tasks given only a few examples in the form of demonstration"* [47]. Examples include chain-of-thought and few-shot prompting. Chain-of-thought prompting is a way to help the model with multi-step reasoning, something that is often challenging for LLMs. This is done by providing the model with intermediate reasoning steps. Few-shot prompting is done by including a few input-output examples into the model's input [48]. Following the same naming convention, one-shot prompts have exactly one example, and zero-shot have none. Chain-of-thought and few-shot prompting can be combined, as can several other techniques. These prompt engineering techniques are illustrated in figure 2.1.

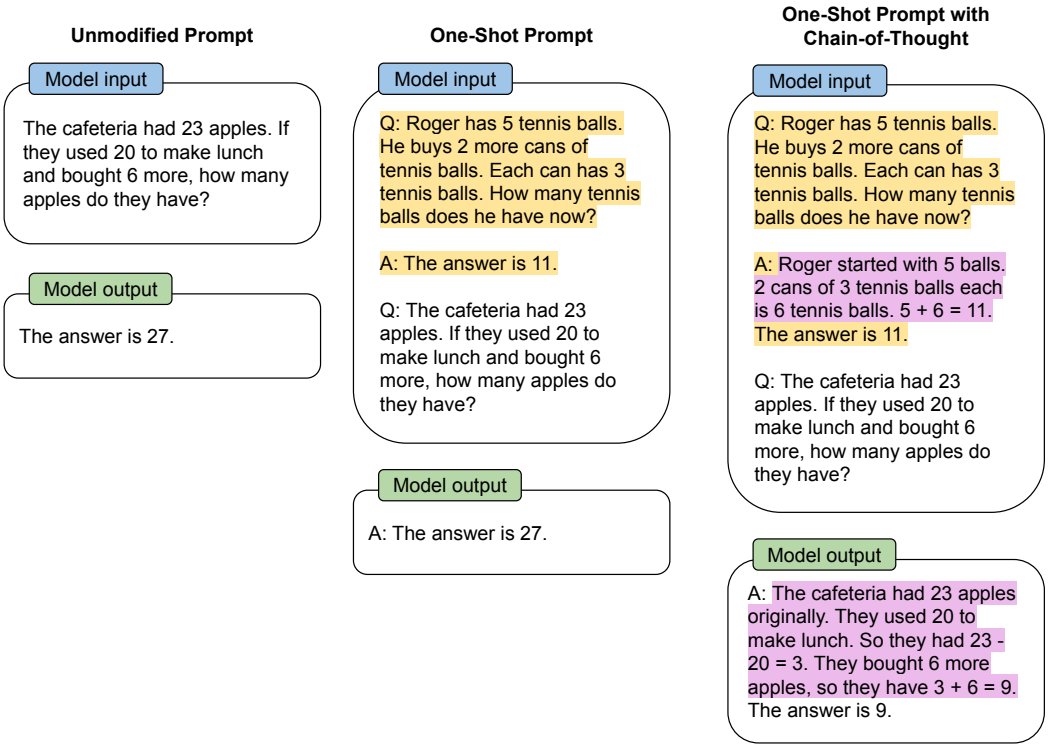


Figure 2.1: Example showing how one-shot prompting and chain-of-though prompting works. One-shot text is highlighted in yellow, and chain-of-thought reasoning is in pink. Note that in this example the model only arrives at the right answer in the final example. Example text and design are partially taken from Wei et al [48].

2.2.2 LLM Agents

An LLM agent is an advanced AI system with an LLM as its core which has access to tools to help it solve problems. These tools or frameworks may allow it to reason, retain memory, plan, and perceive or interact with chosen parts of its environment [14, 15]. For example, a hypothetical LLM agent may have access to an organisation's code base and a version control tool. This would allow it to reason about the code, make changes to it, and push those changes to a remote repository. To the best of our understanding, there exists no widely established definition of

what an LLM agent is, however, there is some consensus [14, 15]. We have drawn on this consensus when crafting the definition above, which is the definition that will be used throughout the thesis.

One tool to help an agent understand a code base is a RAG pipeline, RAG standing for Retrieval-Augmented Generation [49]. RAG allows the LLM to access information from external sources, and use that information to better answer related questions. For the code base example, a retrieval tool could be set up either for locally stored files or for files on a remote repository. LLMs may struggle to use the knowledge stored in their parameters for information-intensive tasks, and there is also the problem of having to fine-tune a model again to be able to update its knowledge about an organisational resource. RAG can help solve both of these problems [49], by giving the agent easier access to updated information.

Multiple LLM agents can be made to work together, as multi-agents, to further help with task complexity. There are many ways to set up how the agents work together. For example, the agents can be given different roles, opinions, and tasks, to help simulate the way an e.g. software development team would work in real life [50]. Other approaches can also be used to make the agents work together, such as combining task decomposition and task allocation [51].

3

Related Work

This chapter discusses related works relevant to the research topics. It first explores test management, especially test maintenance, before moving on to generative AI. There the sections examine related research on how LLMs can help with software engineering and test maintenance, as well as how LLMs can be used as agents.

3.1 Test Management

Many tools exist to help developers automate test suites, both by automatically executing tests [19] as well as generating tests [32]. Code and test generation with LLMs has been explored, but the use of LLMs to help with other test management has not, to our knowledge, been widely explored yet. Only a few studies have tackled this topic, see Section 3.3.3. There is reason to expand upon this topic, as White et al. note that LLMs hold immense potential for automating software engineering tasks and activities, of which test management is a part [7]. In other words, though automated solutions are in many cases used, there is room for these solutions to improve, and reason to further explore how LLMs may help.

Test generation is a way to automate the creation of new test cases, which is part of the various test management activities. Anand et al. describe test creation as having a strong effect on the efficiency of software testing but also note that test creation is among the most labour-intensive of software testing activities [32], and an intellectually demanding task. Because of this much work has been done on different fronts, and common methods for generating test cases include: symbolic execution, model-based, combinatorial, adaptive random and search-based testing. Despite the work done, the reliability of test generation has yet to be proven, with Gay et al. finding that automatically generated test suites in some cases would perform worse than randomly generated ones [52]. This is further supported by Palomba et al. who show that automatically generated test cases often suffer from poor test code quality [53]. They further establish that test cohesion and coupling are good metrics for test code quality. Xu et al. through an empirical study examine how factors influence the effectiveness and cost of test suite augmentation techniques [54]. They find that the primary factor is the test case generation algorithm, followed by how new and existing test cases are utilised.

3.2 Test Maintenance

Test maintenance is a topic that has been explored to a certain extent in previous literature. Pinto et al. examine why test suites evolve [27]. Their findings show that tests that appear to be added or deleted are often simply old tests that have been moved or renamed and that when tests are truly deleted it is most often because they are obsolete, not because they are hard to repair. They further find the main reasons for adding new tests are to cover new functionality, validate bug fixes, and validate refactored code. There are, however, further areas within the topic of test maintenance that are yet to be explored. Imtiaz et al. for instance, noted that there is much room for future studies on methods for repairing tests and that few of the studies done are done in an industrial context [55]. Even when research has been conducted on test maintenance, it is not always implemented. Gonzalez et al. looked at the usage of testing patterns in open-source projects and found that only a quarter of the projects that had tests used patterns related to maintainability [56]. Nevertheless, there is previous research on this topic. Kochar et al. provide an overview of users' perspectives on important aspects of software testing by performing a survey and a series of interviews on test cases [57]. The questions focus on characteristics of good test cases, which are divided into six dimensions, one of which is maintainability.

Factors that affect test maintenance is another area of research within the topic of test maintenance that is relevant to this thesis. Previous research has identified various such factors. Alégroth et al. investigated maintenance for visual GUI testing and found 13 factors that affect automated test suites, and also identified how long the maintenance for tests affected by these factors was estimated to take [1], and how much effort was required. An example is *test case length*, which had a high impact and increased maintenance by more than an hour. Berglund et al. similarly looked at test maintenance for machine learning systems and found 9 factors which affect test maintenance for all systems, machine learning and traditional systems included [21]. An example is *oracle precision*, where tests with sensitive oracles require more updates as the test suite is updated while simultaneously being harder to update. Sensitive, in this regard, refers to an oracle that is highly adapted to the precision of the program's output, where minor changes can cause the oracle to need to be updated. It should be noted that in both these studies the factors are related to how complicated the maintenance will be. They are not looking for specific factors in the code or program that indicate the test needs maintenance, nor are they looking at how changes in the production code might indicate a corresponding need for test maintenance. Factors looking directly at changes in the production code are presented in Section 5.1 and the relevant research investigated is summarised in Section 3.2.2.

3.2.1 Co-evolution of Production Code and Test Code

Co-evolution is the effect of test code and production code being modified in parallel. This concerns test maintenance in how test code is being modified accordingly

to changes in production code to make sure the test suite is relevant and useful. There have been a number of tools, metrics, and methods to help detect changes in production code that indicate test code will need to be changed, as well as to help with co-evolution. Huang et al. propose the tool Jtup, a machine learning approach using random forest [58], which analyses changes made to production code to see if the matching test code needs to be changed as well [59]. The decision on whether co-evolution is needed is based on code change features, semantic features, as well as complexity features of the code. This distinguishes it from other approaches, which mainly look at semantic changes as well as change features. Kita et al. propose the Tconf metric for evaluating how well a production method has co-evolved with its corresponding tests [60]. They make use of the evaluation of logical couplings between production and test code instead of code analysis. Ens et al. present a visualisation tool for co-evolution and co-change between production code and test code [61]. The tool is called ChronoTwigger and is interactive and shows co-change over time. Gall et al. present ChangeDistiller, a tool for examining fine-grained code changes, which makes use of how source code can be represented as abstract syntax trees [62]. ChangeDistiller has been used by several other studies on co-evolution [63, 64, 65]. Sohn and Papadakis present CEMENT, a tool making probable links between production and test code that has been updated in a short time frame, under the assumption that the fact they have been updated, or co-evolved, means they are related [66].

Beyond the tools, metrics, and methods themselves, there is further research on the usefulness and applicability of co-evolution. Sun et al. investigate the assumption that if a production class and its corresponding test class are updated within the same commit or within a short time frame they are an example of linked co-evolution between test and production code [67]. They found that the longer the time frame between the change in the production code and the change in the test code, the less likely they are to be a true example of co-evolution. Updates within the same commit contained 11.34% false positives, while pairs with more than 24 and 48 hours had 85.71% and 89.19% false positives respectively. Based on this they claim the co-evolution samples used by Wang et al. [2] include noise. Klammer and Kern show that visualisations can be used to understand and keep up with how a systems production and test code co-evolves [68]. This was tried when analysing co-evolution in industrial projects.

3.2.2 Co-evolution Factors

The following section will present the factors found in existing literature for which modifications in the production code lead to a need to modify the test code. These were used as part of answering the first research question RQ1, which is defined in Section 4.1 and the results of which can be found in Section 5.1. Previous studies have looked at the co-evolution between production code and test code, and extracted patterns of what triggers this co-evolution. This has been done at different scales, from certain types of changes (e.g. a change in the method body, or addition of a conditional statement), to looking at specific syntax and keywords.

Shimmi and Rahimi extracted and documented higher-level patterns on co-evolution between production code and test code [69]. The patterns were classified under *additon, deletion*, and *modification* and an example is *addition: added functionality* where the corresponding test cases are made when the functionality is added to the production code. Reich and Maalej similarly extracted patterns, but focused on refactorings and co-evolution to increase the testability of the production code [70]. They identified both high and low-level changes in the production code. They define a low-level change in the production code as a local change in a production file, such as *changing an attribute type*. A high-level change in the production code would have wider effects beyond the local area around the change, such as *merging a package*. These changes were used to find testability patterns, such as the *extract_method_for_invocation* pattern. Levin and Yehudai also use semantic changes and investigate the relationship between test code maintenance and production code maintenance [63]. They identify both high-level relationships (e.g. *REMOVED_CLASS*, where removing a production class will lead to the removal of a test class) and low-level relationships (e.g. *RETURN_TYPE_CHANGE* where changing a return type in the production code will lead to test maintenance). Marsavina et al. extracted patterns for fine-grained co-evolution between production and test code [65]. They extracted fine-grained changes in production code and linked them with the related test code, from which they identified six co-evolution patterns. Vidács and Pinzger later found support for five of the six patterns found by Marsavina et al [64].

Factors were also extracted from literature where tools to predict or help with test maintenance had been developed, and they reported which factors the tools acted on. DRIFT [71] is a further development of SITAR [2], which identifies outdated test cases based on changes in the production code at the method level. Within this work, they identified fine-grained changes in the production code that may be related to co-evolution. Some examples of these fine-grained changes include: *Try, Break,* and *If*. These fine-grained changes would require changes in the test code, hence their relation to co-evolution. TestCareAssistant, originally proposed by Mirzaaghaei et al. [72], was further developed by Mirzaaghaei et al. and can repair and generate new test cases as the production code changes [73]. TestCareAssistant looks at parameters and returns values to identify when a test case becomes outdated, which were identified as indicators that co-evolution was needed. As part of the work to develop TestCareAssistant, Mirzaaghaei worked to formalise test maintenance activities into test adaptation patterns [74]. CEPROT identifies outdated test cases and also updates them [22]. The main factors looked at in the production code to detect the need for co-evolution are API invocation and changes to identifiers and modifiers.

3.3 Generative Artificial Intelligence

The following section will present some of the relevant research on generative AI, its use cases and its performance. Generative AI has been extensively researched in the last few years with many different approaches. Gozalo-Brizuela and Garrido-

Merchán investigated various generative AIs and classified them into a total of 9 categories [75]. The most relevant topic areas for this thesis lie in research where generative AI and LLMs are used for coding and other software engineering purposes. Thus the most relevant categories are the text-to-text models as well as the text-to-code models. The most frequently used, and studied, LLMs at the time of writing is OpenAI's series of LLMs in the GPT series, most commonly GPT-3.5 and more recently GPT-4. These LLMs are often interacted with through ChatGPT, a chat robot that is implemented through the use of the GPT series. These are text-to-text LLMs that have gained immense popularity after ChatGPT's original introduction. Conversely, there are many different text-to-code LLMs [20, 75].

Beyond ChatGPT and its use cases, there are further factors to consider regarding how an LLM can perform. Mandvikar compares LLM models to each other and presents several factors that describe how LLMs can differ [76]. These factors include, but are not limited to, the kind of pre-trained data, the size of the model, the API capabilities, etc. These factors are then useful to consider when selecting an LLM for a specific task according to Mandvikar. Beyond these factors, Döderlein et al. investigate two LLMs and how they can be improved based on their input parameters [77]. Their findings indicate that the temperature, which is a parameter affecting how varied a response will be, and the initial prompt can have a significant effect on the performance of the LLM. To get a further understanding of the performance of LLMs Chang et al. perform a survey focusing on the evaluation of LLMs [78]. They focus on three aspects, namely *what* to evaluate, *how* to evaluate, as well as *where* to evaluate. Their findings include limitations in LLMs and their reasoning ability, as well as their robustness.

3.3.1 LLM Capabilities in Software Engineering

There is some previous literature aiming to collect and present various research papers that have investigated LLMs and specifically their use for software engineering. Zhang et al. investigate existing LLM-based software engineering (SE) studies, both studies focusing on LLMs as well as studies focusing on SE [79]. They discuss architectures, benchmarks, optimisation and application, as well as some challenges of LLM research. Their findings indicate how LLMs are being trained for more code-aware objectives compared to earlier natural language processing-derived objectives. Further findings include a consideration for variables, and structural features, as well as utilising cross-modal learning. This signifies advancements towards LLMs that consider the semantics and functional aspects of code beyond processing the code as a sequence of tokens. Hou et al. present a systematic literature review on how LLMs are utilised for software engineering [43]. Their findings provide a comprehensive list of different utilisation areas for LLMs including, but not limited to, code generation, code completion, code understanding, program repair, code review, bug prediction, vulnerability detection, and verification.

There are additional reviews and surveys of previous works for various software engineering activities. Zhang et al. present a review of the history of code processing and generating code from a natural language description, from natural language process-

ing models to few-shot prompting applications of LLMs [44]. Wang and Chen present a review of previous work on how LLMs can be utilised for code generation [80]. They focus on the application of LLMs for this topic as well as the evaluation of the generated code. They find several limitations with an LLM's application for code generation including, but not limited to, compatibility, maintainability, portability, correctness, and privacy. Despite the limitations presented Wang and Chen conclude that code generation with LLMs has progressed and can handle increasingly complex tasks. Their findings show how there is a lack of research on the evaluation of LLM-generated code. Zheng et al. provide a comprehensive review of the current stage of code LLMs through their survey [81]. Code LLMs are LLMs that have been trained mostly on code repositories instead of natural text, though some are trained on both. They list several code LLMs, their applications, as well as the relationships between them, both between themselves and compared to general LLMs. The performance of the code LLMs is investigated and compared to benchmarks for multiple software engineering tasks. They summarise their findings with code LLMs having a focus on code generation with some lesser emphasis on other tasks, e.g. vulnerability repair or evaluation.

Other research directions include more specific work on how LLMs can be utilised for various software engineering activities. Uusnäkki investigates the applications of generative AI on software development [82]. As part of this Uusnäkki performed an empirical study on the use of prompt engineering for enhancing software system maintenance. As part of the results, the PESD framework is presented, which is a framework for systematic prompt engineering. Fan et al. investigate how hybridising, i.e. using LLMs along with existing software engineering techniques, such as API search techniques or search-based test generation, can reduce hallucinations and improve performance [83]. Their findings indicate that this is a promising topic with several successful examples. Pei et al. investigate how LLMs can work with program invariants, including predicting them [84]. They present a method for predicting invariants through fine-tuning LLMs and find that LLMs are effective on this task, with 86% recall and 86% precision. The different invariants include object, class, function-entry, function-exit, and loop invariants. Liu et al. propose CodeExecutor, a model focused on enhancing code execution through LLMs [85]. They utilise pre-training and curriculum learning to improve the model on code execution tasks specifically. Liang et al. investigate the qualitative experience of LLMs as coding assistants [86]. Their findings show that LLMs are mostly used for code completion and faster keystrokes. On the other hand, most users in the study find that code generation does not reach quality requirements and creativity and ideas are underutilised.

As discussed in Section 3.3 ChatGPT is a popular LLM in a general sense at the time of writing. It has also been investigated on its applicability for software development. White et al. investigated the use of ChatGPT in software development and identified 14 prompt patterns that would make the answers from ChatGPT more helpful [7]. These patterns were focused on software development. Based on these patterns some benefits of using ChatGPT identified were rapid experimentation at different abstraction levels or identification of assumptions in the code of a project.

Rahmaniar discusses potential applications of ChatGPT in software development but also brings up several challenges that may arise when attempting to integrate ChatGPT or other generative AI into a development process [87]. Rahmaniar mentions topics that ChatGPT would be adept at handling or assisting with such as documentation, onboarding, reviewing, and of course code writing assistance. Worth noting is that each of these topics, among others not mentioned here, has its limitations and will according to Rahmaniar require some sort of human component for best results.

3.3.2 LLMs for Code Interaction

This section will focus on LLMs for software engineering activities specifically interacting with code. This includes various frameworks, techniques, and methods that can view or make modifications to the production code. Sghaier and Sahraoui present a framework for utilising LLMs for code review [88]. They believe, and their findings indicate, that fully automating code reviews does not lead to the best results and therefore their framework aims to lessen the workload of a code reviewer and provide assistance instead of automating the whole review. Zhang et al. present a survey of automated program repair (APR) solutions in current literature, many of them focusing on utilising LLMs for APR [89]. They describe the typical framework, and design strategies, as well as metrics and empirical studies. Similarly, Xia et al. investigate the use of LLMs for APR [90]. Ibrahimzada et al. present BUGFARM, a technique utilising LLMs to generate bugs [91]. They utilise attention analysis to attempt to find the weak spots of LLM models and then improve their performance through training on the generated bugs.

Further implementations using LLMs include Fried et al. who present InCoder, a model that can perform both program synthesis as well as editing through LLMs [92]. Additionally, they use causal modelling to improve the performance of their model, particularly the infilling capabilities of the model. Dou et al. investigate the capabilities of LLMs when it comes to code clone detection [93]. Their findings indicate that LLMs have the potential to outperform other automatic clone detection methods, especially regarding complex semantic clones. Geng et al. investigate how well LLMs can generate comments and summaries of code [94]. Their findings indicate that through few-shot learning an LLM can perform better than existing supervised learning approaches. Chen et al. present SELF-DEBUGGING, a framework where LLMs can iterate over their own generated code to find and rectify errors, both semantic and syntactic [95]. Their findings suggest that LLMs can improve their performance by going over the code it has previously generated. Ren et al. describe several limitations of exception handling by LLMs and present KPC to mitigate that, which is a code generation approach for using LLMs so that they handle exceptions better [96].

3.3.3 LLMs for Testing

This section will focus on how LLMs have been used for testing and testing-related activities. Wang et al. have investigated what testing activities have already been

performed using LLMs [97]. They have investigated 50 different utilisations of LLMs in software testing and have then reviewed and analysed the results from them. Wang et al. state that LLMs are more useful for automation in testing in comparison to automation in source code. There has mainly been unit testing being performed with LLMs, and while there is some system testing being done, no integration or acceptance testing being found by Wang et al. The majority of the testing was functional testing with a small amount of security testing. No performance or acceptance testing using LLMs was found by Wang et al. *"There is currently no clear consensus on the extent to which LLMs can solve software testing problems."* says Wang et al. in an overall view of the current state of the research on this topic. As found by Wang et al. there have been multiple studies on test code generation. For instance, Yuan et al. evaluated ChatGPT's ability to generate unit tests [3]. They found that about a quarter of the generated tests pass, but the rest suffer from issues with compilation, correctness, and execution. What is notable is that the tests that pass resemble manually written tests in quality. They describe ChatGPT's ability as promising if the correctness were to be improved. Another study on test code generation is by Schäfer et al. who presents TESTPILOT, an approach for unit test generation by LLMs [5]. Their findings indicate that LLMs provide higher coverage and a larger amount of non-trivial assertions compared to previous test generation techniques. They conclude that LLMs can lessen the work required for unit testing but not replace the need to write unit tests entirely, especially when it comes to more complex tests.

Further research on test generation is by Siddiq et al. who investigate the unit test generation capabilities of three code generation LLMs [98]. They compare strongly typed languages, e.g. Java, to weakly typed languages, e.g. Python, to see if the generation by LLMs differs. Their findings suggest that LLMs have more difficulties with more strongly typed languages from the fact that syntax has an increased importance compared to semantics. They also investigate the applicability of utilising LLMs for Test Driven Development (TDD). Their findings indicate that LLMs can work well with TDD. Kang et al. introduce LIBRO, a technique to use LLMs for generating tests based on bug reports [99]. The tests generated have the purpose of reproducing the bugs, which has a success rate of 33%. Lemieux et al. present CODAMOSA, an algorithm for using LLMs to enhance search-based software testing [100].

An important aspect of an LLM for this study is the understandability of the code and the suggestions made by an LLM. Gay investigated the readability of tests modified by LLMs [101]. Gay worked with GPT-4 through ChatGPT and a code interpreter plug-in and identified that over 90% of the investigated case examples had significant improvement in the test readability after GPT transformed the tests. There are some challenges present which include, but are not limited to: non-determinism, text and prompt limits, code interpreter limitations, and transformation order. Nevertheless Gay concludes that LLMs seem promising in the task of improving readability in tests.

3.3.4 LLM Agents

This section will present previous related work done on LLM agents. This is a very recent topic at the time of writing and therefore there is limited research done on the topic. Jiang et al. investigate the use of planning with LLMs, where the LLM first makes a plan for its actions before proceeding with those actions [102]. Their findings indicate that a planning phase can improve performance, despite planning being an emergent ability of LLMs. Although this is not specifically about LLM agents it serves as support for one of the bases of LLM agents. Zhao et al. present a method of choosing between chain-of-thought and program-aided language models [103]. Their findings indicate a benefit to performance for choosing the better-suited model for each problem. This is not specifically about LLM agents but also serves as a base idea for multi-agent frameworks, where multiple LLM agents cooperate and use their various specialised skills to collectively produce an improved result.

For work done specifically with LLM agents, Feldt et al. work towards SocraTest, an autonomous LLM agent that can invoke tools [15]. They present a taxonomy of agents as well as a concrete example. Hong et al. present MetaGPT, a multi-agent framework designed to solve various problems by simulating a software company structure [104]. Their findings indicate that taking inspiration from humans can improve the workings of LLM agents and how they work together. Rasheed et al. present CodiPori, a code generation model based on multiple LLM agents [105]. Their findings show that the performance of LLM agents working together can outperform existing single LLM usage. Shen et al. investigate the limitations of small LLMs when it comes to tool usage in LLM agents [106]. Their findings suggest that simplifying and dividing tasks into different instances can improve the performance of LLMs, especially LLMs of smaller sizes. Yoon et al. implement DROIDAGENT, an LLM agent that performs Android app GUI testing automatically [107]. Their findings indicate that LLM agents can contribute to autonomous GUI testing based on more meaningful exploration choices and depth of search.

3.4 Summary

This section will present a summary of the related works and position this master thesis against the open challenges of previous literature. The gaps in current research and the steps we take to fill them are discussed in this section with a finishing paragraph highlighting the research that has been utilised in this thesis.

Automating test management and maintenance are topics that have been somewhat explored in previous research. However, the majority of automation comes in the form of test case generation and focuses less on modifying existing test cases or identifying affected test cases when production code has been changed. The research on the co-evolution of test and production code also lacks research when it comes to automation beyond test case generation that concerns generating entirely new test cases. We address this gap by building a prototype that identifies test cases that might need to be modified based on a production code change.

Generative AI and LLMs are more recent topics that have seen extensive research over the last few years. Most research has been performed with larger models that are not available through open source and instead require access to APIs, and therefore lack control over the deployment of the model. In addition, most research includes pre-training or fine-tuning a model to fit a specific use case better. There is also a lack of research done in the area of multi-agent architectures that focus on topics beyond feature development. Previous research has explored the topic of code understanding but has not further applied this understanding to topics such as code traceability. Applying LLMs and LLM agents to test maintenance is also an unexplored area from what we have found. To our knowledge, we are the first to explore a multi-agent setup of open-source LLMs without pre-training or fine-tuning. In addition, our focus on using LLMs to help automate test maintenance is novel to our knowledge.

Some research that has been utilised to great effect in this thesis is previous research on triggers for test maintenance. For details on this see Sections 4.3 and 5.1. Additionally useful for this thesis is previous research done on actions that LLMs and LLM agents can take. For details see Sections 3.3.1 and 5.4. By building on previous research this thesis aimed to fill gaps in the research areas highlighted in this section. The aim is to expand the areas of application of LLMs while also providing more options for automating test maintenance activities. The inclusion of test maintenance triggers stems from the desire to have criteria that an LLM agent can utilise to know when to act. The inclusion of LLM actions and LLM agent actions stems from the desire to understand what applications of LLMs and LLM agents can be applied for this thesis' use case.

4

Methods

The methods chapter will present the different steps taken throughout the case study to answer the research questions. This chapter first presents the research questions (Section 4.1) before giving an overview of the case study (Section 4.2). The remaining sections provide more detailed explanations of each case study step.

4.1 Research Questions

This section will first present the research questions. It will then motivate them and explain their purpose, as well as connect them and the scope of the thesis.

RQ1 Which factors suggest that test maintenance needs to occur due to changes in the production code?

RQ2 What applications could current LLMs or LLM agents have within the area of test maintenance?

- **RQ2.1** Which factors from RQ1 can act as triggers for test maintenance in an LLM or LLM agent?
- **RQ2.2** What are potentially viable test maintenance actions that an LLM or LLM agent could take based on these triggers?
- **RQ2.3** Based on the present-day landscape, what are some considerations for building an LLM agent for test maintenance within a corporate setting?

RQ3 What is the precision, recall, and F1 score of our setup with LLM agents in predicting if and where test maintenance is necessary using the factors found in RQ1?

Identifying the need to evolve some part of a test suite is the first step of test maintenance. The purpose of RQ1 is therefore to identify and categorise the issues and changes that lead to the need to perform test maintenance. This was answered by a literature review to identify changes in the production code that lead to a need for test maintenance (described in Section 4.3), a thematic analysis of interviews with practitioners (described in Section 4.4), as well as conducting a survey with

practitioners (described in Section 4.5).

RQ2 builds upon RQ1 and aims to explore how LLM agents might fit within the problem space, as well as what should be taken into consideration when building them. This decision to use LLM agents stems from the literature review of LLMs and the results from that combined with the results of RQ1. RQ2.1 identifies which of the results from RQ1 are suitable to move forward with by reasoning about the triggers and the planned architecture of the agent. RQ2.2 is more exploratory. It uses existing literature on software testing and LLMs to both give ideas about areas of application for a test maintenance setup with LLM agents as well as suggestions regarding how particular triggers might suggest particular applications. The question is answered by presenting examples of how LLMs have been used within software testing. RQ2.3 draws upon current literature as well as the interviews and the survey to identify surrounding factors and limitations within Ericsson that must be considered when deploying an agent.

RQ3 assesses the performance of an initial prototype solution. It builds upon the results of RQ2, but does not seek to confirm all of RQ2s findings. This RQ only seeks to try out one of the possible use cases found in RQ1 and RQ2. As a proof of concept, a setup with LLM agents was designed and evaluated on its precision, recall, and F1-score. The design process is described in Section 4.8 and the evaluation process in Section 4.9.

4.2 Research Design

This study is a case study investigating the applicability of LLMs within the test maintenance domain at Ericsson, especially LLM agents. The case study follows the guidelines laid out by Runeson and Höst [108]. This section will present an overview of the case study methods, and how the different elements relate to each other and the research questions. The overall structure of the methods of the case study is also displayed in Figure 4.1. The case study had the following steps:

(1) Literature review of test maintenance factors: To find factors that when changed in production code lead to test maintenance, meaning a need to make changes to test cases, a literature review was conducted. The process is described in detail in Section 4.3, and the results contributed to RQ1.

(2) Interviews: The interviews were conducted with Ericsson employees to get a better understanding of test maintenance problems at Ericsson. A thematic analysis was done to analyse the results, which contributed to RQ1 and RQ2. The process is described in Section 4.4.

(3) Survey: Similarly to the interviews, a survey was sent out to Ericsson employees to better understand Ericsson employees' views of test maintenance, as well as opinions about generative AI. The results contributed to RQ1 and RQ2, and it is described in Section 4.5.

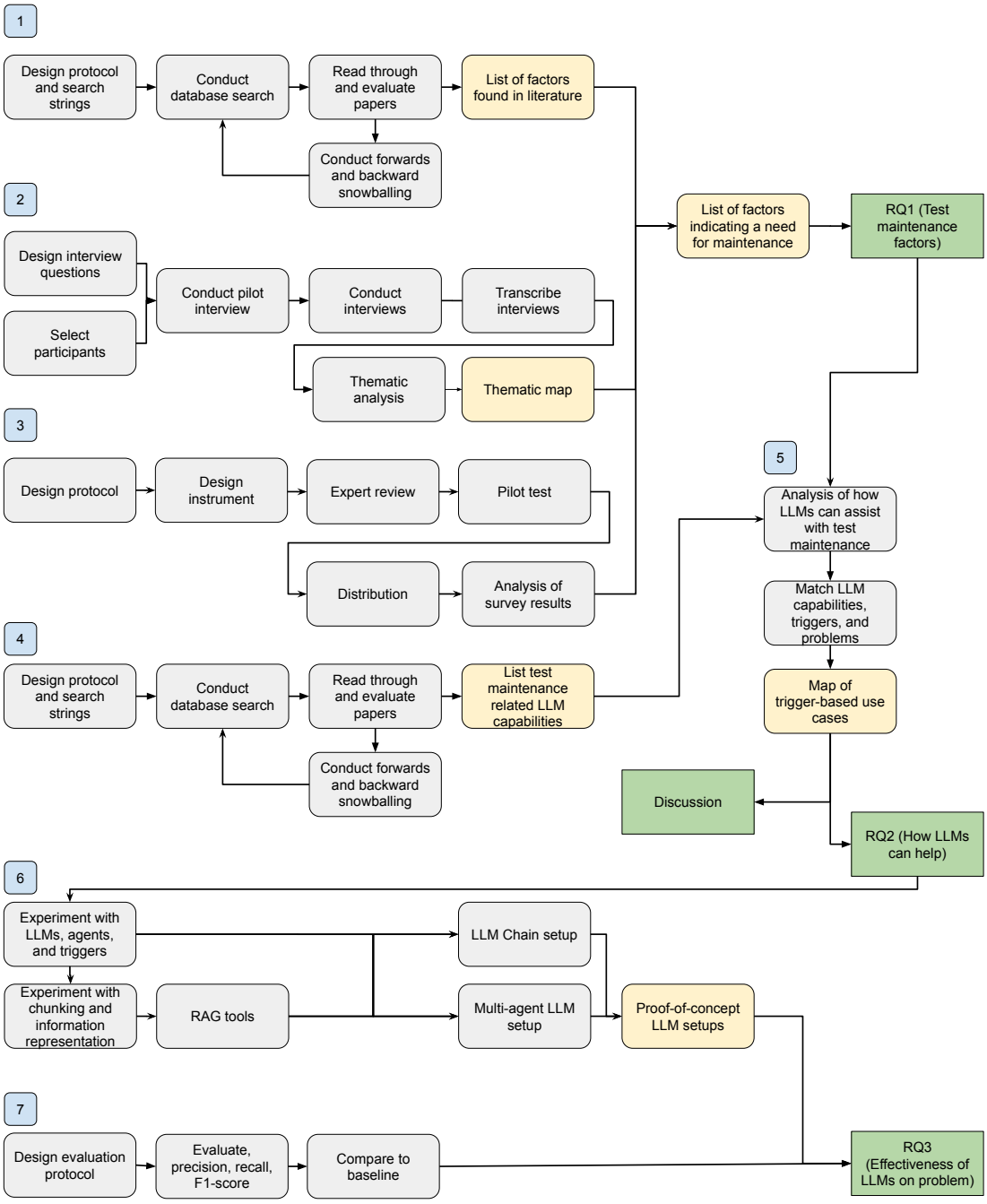


Figure 4.1: Overview of case study. The numbers correspond to the case study's different steps. 1 = literature review of test maintenance factors, 2 = interviews about test maintenance and LLMs, 3 = survey about test maintenance and LLMs, 4 = literature review of LLM capabilities, 5 = synthesise results, 6 = build prototype of setup with LLM agents, 7 = evaluate prototype of setup with LLM agents. Grey = activity, yellow = artefact, green = research question.

The results of these three steps were used to find the final list of factors indicating a need for test maintenance, which is the answer to RQ1. The decision to use three different methods was taken to get data and method triangulation, to help increase the precision and validity of the results.

(4) Literature review of LLM capabilities: To understand how LLMs are currently used for test maintenance, as well as understand their suitability and limitations within the current use case, a literature review was conducted. The results were used to answer RQ2, and the review protocol is described in Section 4.6.

(5) Analysis of which test maintenance problems and triggers to implement in agent: This step used the results from steps two, three, and four to decide which of the identified test maintenance problems would fit with which triggers. This was done ad-hoc, taking the time and resource limitations into account. The result was partly used to answer RQ2 and provide a basis for larger use cases in the discussion. For a further description, see Section 4.7.

(6) Design setup with LLM agents: Four proof-of-concept LLM setups were designed and implemented to explore the viability of using LLMs to predict test maintenance. This step contributed to the result of RQ3 and is described in Section 4.8.

(7) Evaluation of setup with LLM agents: The setup with LLM agents was evaluated on its precision, recall, and F1-score, the results of which were used to answer RQ3. For a further description, see Section 4.9.

4.3 Literature Review to Identify Test Maintenance Factors

A literature review of factors in the source code that indicate the need for changes in the test suite was performed in the early stages of the thesis to help answer RQ1. Though the literature review was not a Systematic Literature Review due to the time constraints, it did take inspiration from its strict protocol, as described by Keele [109]. Each step of the literature review was based on the guidelines provided by Keele. The majority of the steps described by Keele were followed, if less meticulously than in the original, with two exceptions: the data collection, and the dissemination. The data collection was not as rigorous due to the short time frame, and the dissemination, i.e. report writing, had less focus due to the aim of the literature review not being isolated but instead leading into the next step of the thesis.

The databases that were utilised to search for primary sources were: IEEE, Science Direct, ACM, SCOPUS, and Google Scholar. For the search strings used for each respective database, see Appendix A. The search was limited to papers released within the last 15 years, i.e. in the period 2009-2024. This range was chosen based

on the desire for relevancy to current-day software engineering and testing standards and practices.

Relevancy was judged first through the title, followed by the abstract and the conclusion of the papers. Once it was judged that no more relevant papers were being found the initial scan ended. 48 papers had been found through this step. These papers were then examined in more detail, and if they contained relevant factors, they were recorded in a separate document. This step yielded 8 different papers that named test maintenance factors in the source code.

These 8 papers were then used as a staging point for both backwards and forward snowball sampling. The method of examining the relevancy of the new research papers was identical to the method used for the original examination of research papers in the first step of the literature review where the relevancy of the papers was checked. This step yielded an additional 64 papers. From these papers, an additional four papers were found to describe relevant factors in production code that when changed would lead to a need for test maintenance. This led to a total of 12 found papers with relevant factors.

To sort and organise the factors, a Miro [110] board was used. See Section 5.1 for the result of the literature review.

4.4 Interviews

Interviews were held with Ericsson employees to get a better understanding of the current state of the test maintenance problem and where improvements can be made as part of RQ1. The interviews also included questions on LLMs and generative AI, both current use and opinions, as part of RQ2.

4.4.1 Selection of Interviewees

Table 4.1: Demographics of Interviewees. Experience refers to years of experience with testing, type refers to testing they are currently performing. IDs that were interviewed together have been grouped.

ID	Experience (Years)	Role	Type
P1	15	Software Developer	Unit
P2	3.5	Data Scientist	Unit
P3	6	Developer	Unit
P4	5	Developer	Unit
P5	3	Software Developer	Unit
P6	2	Test Manager	Integration, System
P7	2	Test Manager	Integration, System
P8	25	Principal Developer	Overseeing Process

Convenience sampling was utilised for the sampling of the interviewees. Based on the supervisors' knowledge of the organisation, emails were sent out to relevant teams and developers to explain the master thesis and inquire about participation in interviews. Some interviews were held in groups, for the convenience of both the interviewers and the interviewees. Table 4.1 presents the demographics of the interviewees who agreed to be interviewed.

4.4.2 Interview Instrument

The initial step for the interviews started with writing out relevant questions to the topics of RQ1 and RQ2. Questions were left open to avoid leading questions. After the questions were written, an expert review was performed by the academic and industrial supervisors of the thesis. A pilot interview was performed, and based on it small changes were made. One question was removed, as it was deemed irrelevant to the research questions. Some questions received minor clarifications. Because the changes remained relatively small, the data from the pilot interview was used in the final analysis. A consent form was presented to each interviewee before the interview could start in addition to receiving permission from the interviewee(s) to record the interview. The interview had a length of roughly 40 minutes on average. See Appendix C for the interview consent form. See Appendix D for the interview questions.

4.4.3 Interview Analysis

All the interviews were transcribed through Microsoft Teams [111] and were later manually corrected after a process of listening through the recordings of the interviews. After the interviews were transcribed a thematic analysis was performed to identify themes and common concepts and thoughts of the interviewees. The thematic analysis followed the steps and guidelines described by Braun and Clarke [112]. An inductive and semantic approach was mainly used. The results of the thematic analysis can be found in Section 5.2.

4.5 Survey

A survey was sent out to employees at Ericsson, whose work was related to software engineering, to gauge their way of performing test maintenance as part of RQ1 as well as their opinions on LLMs as part of RQ2. A protocol for the survey was designed based on Ghazi et al. [113] and Kasunic [114]. The nature of the survey was exploratory, and the motivation behind the survey was to find out how the test maintenance process is managed, what kind of help practitioners want from LLMs, and what in the source code triggers an update to a test case.

4.5.1 Selection of Participants

The desired population was Ericsson developers with testing experience, as well as other Ericsson employees who worked with testing. Convenience sampling was

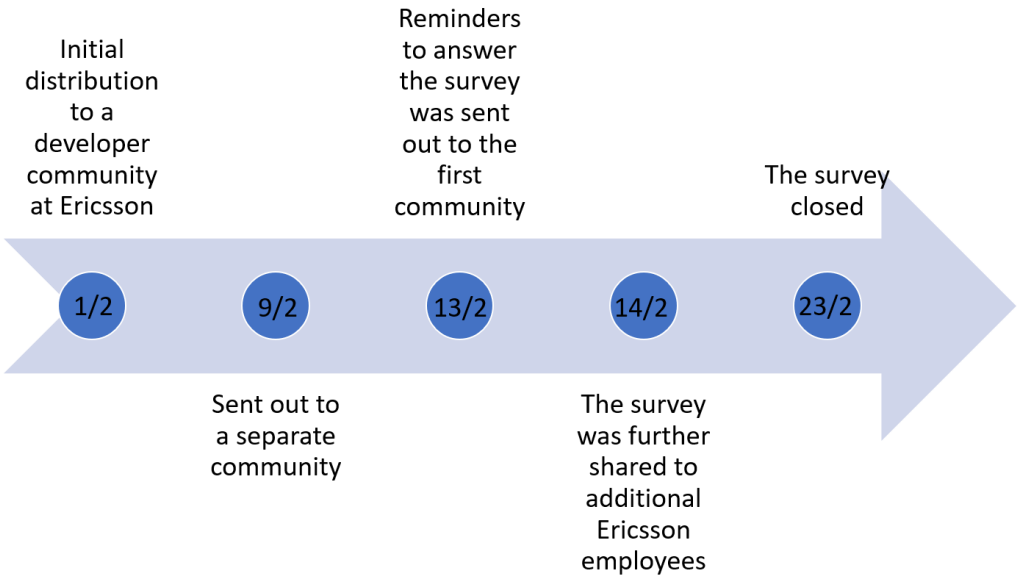


Figure 4.2: Timeline of the distribution of the test maintenance and LLM usage survey through February 2024.

used to distribute the survey. Based on the supervisors' knowledge of Ericsson and the various organisations within, emails were sent out to relevant communities and teams. For demographics of the respondents, see Section 4.5.3 and Figure 4.3.

The survey was initially planned to be distributed to a single developer community, consisting of over 100 developers across different countries and sections within Ericsson that work within the same area, and then be available for two weeks. However, based on a low response rate the survey was sent out to several different communities during the time the survey was available, necessitating an extension of the availability of the survey to make sure that respondents had time to answer. A timeline for the survey distribution can be seen in Figure 4.2.

4.5.2 Survey Instrument

Next, the survey instrument was designed. The survey was determined to be an unsupervised cross-sectional survey. The survey was designed to take 5-10 minutes to increase the chance of the respondents answering the survey. Care was taken to make sure that questions were not left open-ended nor that there were too many questions. The total number of questions was 10, with all of them being multiple-choice questions. Some questions allowed the respondent to choose multiple answers. These questions had a limit to the number of answers that could be chosen, to force the respondent to prioritise the most relevant choices. The wording of all questions was evaluated using a checklist based on understandability criteria set by Kasunic [114]. Understandability criteria are rules for the phrasing and structure of questions such that minimal confusion and misunderstandings occur.

The instrument started with an information page that included the identity of the surveyors as well as the purpose of the survey and how their answers would be treated. Following the information page were some initial attribute questions regarding the demographics of the respondents.

After the attribute questions about demographics, the next section focused on test maintenance activities. This section contained questions about the behaviour and belief types, as described by Kasunic [114]. Thereafter the final section consisted of two questions about the respondent's use of and attitude towards LLMs and generative AI. These two questions were of the behaviour and belief type respectively, as described by Kasunic. All questions in the survey can be found in Appendix E.

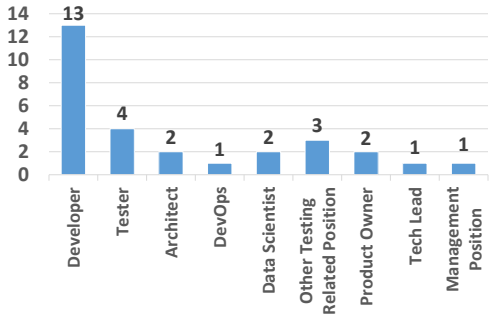
The survey instrument was evaluated by a Data Analytics Expert at Ericsson. It was also pilot-tested by three Ericsson developers to ensure there were no ambiguities in the questions, as well as to ensure it could be completed in less than ten minutes.

4.5.3 Survey Analysis

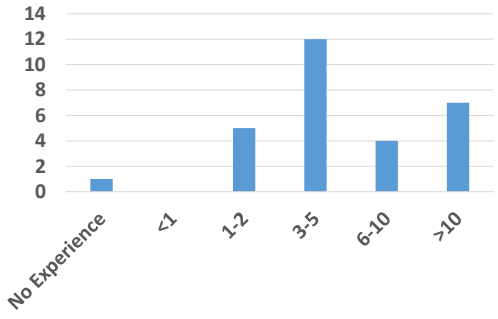
The total number of people who received the survey is unknown based on the fact that it can not be confirmed if respondents shared the survey with additional colleagues beyond the receivers of the emails that were originally sent out. What can be confirmed is that no respondent answered the survey more than once as each respondent had to log in with their Ericsson account and the survey was set to only accept one response per account. The total number of people who received the survey is at least 300 but it is otherwise unknown. The total number of responses to the survey was 29 and thus, while the exact number of recipients is unknown, the response rate is less than 10%.

The demographics of the respondents are presented in Figure 4.3. The median role of a respondent is developer, and the median experience and type of testing performed is 3-5 years and unit testing. The main programming language was Python. If respondents are separated into groups of up to five years of testing experience compared to more than five years of testing experience, the results differ. While respondents with up to five years of experience mostly work as developers with unit testing, the role of respondents with more than five years of experience was less uniform, and integration testing was more common than unit testing (see Figure 4.4). One explanation for these differences in results could be that experienced professionals are more suited to testing at levels where a broader and deeper understanding of the product and requirements is needed.

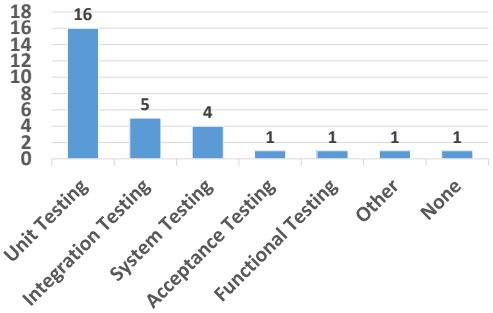
The results of the survey were analysed using descriptive statistics, to get an overview of the respondents' thoughts about test maintenance and to identify trends in the answers.



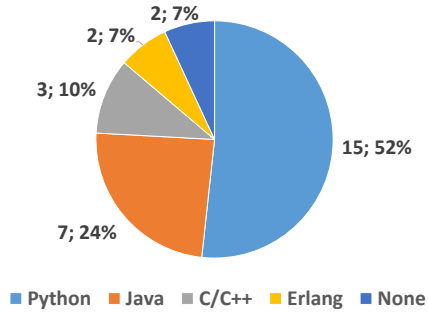
(a) Work role of respondents. Axes show frequency and responses.



(b) Years of experience with software testing. Axes show frequency and responses.



(c) Most common type of testing performed. Axes show frequency and responses.



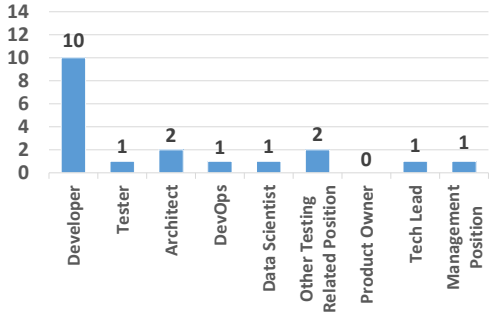
(d) Most commonly used programming language. Labels show the number of occurrences; percentage of total occurrences.

Figure 4.3: Demographics of survey respondents. All questions and answer alternatives can be seen in Appendix E.

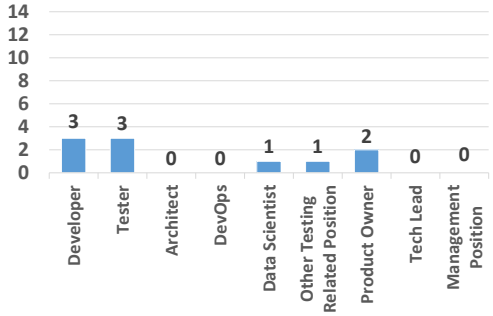
4.6 Literature Review of Large Language Models

A literature review of LLMs and their capabilities and applicability was performed to find answers to RQ2. The process of performing this literature review closely mimics the process for the previous literature review, see Section 4.3. A protocol was created with inspiration from Keele [109] in the same fashion as the previous literature review.

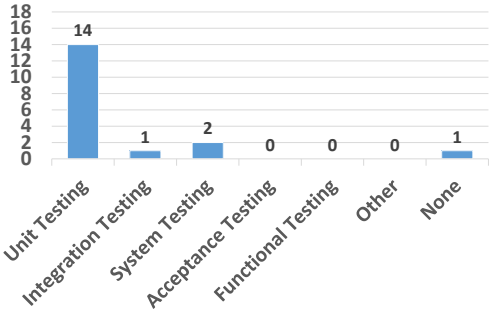
The databases that were utilised to search for primary sources were: IEEE, Science Direct, ACM, SCOPUS, and Google Scholar. For the search strings used for each respective database see Appendix B. The search was limited to papers released in the time frame of 2017-2024. This range was chosen based on the first significant developments of LLMs, as defined after consultation with the supervisors at Ericsson. The starting point was based on the release of papers such as Vaswani et al. [37]



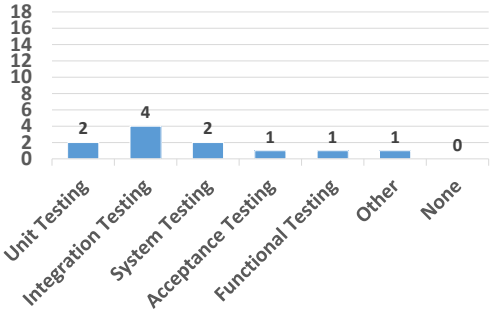
(a) Work role of respondents with less than 5 years of experience. Axes show frequency and responses.



(b) Work role of respondents with more than 5 years of experience. Axes show frequency and responses.



(c) Most common type of testing performed for employees with less than 5 years of experience. Axes show frequency and responses.



(d) Most common type of testing performed for employees with more than 5 years of experience. Axes show frequency and responses.

Figure 4.4: Difference in the work role and type of testing performed for employees with less and more than five years of experience. All questions and answer alternatives can be seen in Appendix E.

and the end date is up to the point where the literature review was conducted.

The papers were first checked for relevancy through their titles, abstracts, and conclusions. Relevant papers were added to a worksheet for later perusal. The inspection of papers for each search string continued until three subsequent pages of irrelevant papers were identified. In total, 67 papers were chosen for further examination.

After the initial scan ended the papers were examined in more detail. The papers were checked for information regarding how LLMs can interact with code and other artefacts, as well as how suitable LLMs are to help with different software engineering tasks, especially test maintenance tasks.

Papers that were deemed to provide relevant information were distinguished from papers that did not provide relevant information. These most useful papers were then used as staging points for both backwards and forward snowball sampling. There were ten papers used for the snowball sampling. The second selection of papers found through the snowball sampling was treated the same as the papers found through the initial scan. An additional 92 papers were selected for further examination through snowball sampling. Two papers of the second selection were deemed more useful than the rest, for a total of twelve papers that had the most relevant information for this literature review. Beyond these twelve papers there were additionally some papers that had lesser relevance but nonetheless provided some useful information.

4.7 Matching Test Maintenance Problems and LLM Capabilities

To analyse how LLMs can assist with test maintenance, data from the previously described steps was utilised to cross-compare test maintenance problems, practices, and possibilities with LLM actions and applications. The data include the results from the interviews (see Section 5.2), the survey (see Section 5.3), as well as from the literature review of test maintenance factors (see Section 5.1). The results from the literature review of large language models (see Section 5.4) provided data on potential applications of LLMs.

A list of potential matches between test maintenance problems and ways to apply LLMs was developed through a brainstorming session, consisting of an open-ended discussion based on the previously mentioned results. Through this brainstorming process, the idea of using LLM agents was introduced and eventually decided to be the most appropriate application of LLMs. The reasoning behind this decision stems from how an LLM agent can utilise external tools, like retrieval-augmented generation (RAG) [49] for example, to obtain relevant and current knowledge and information without retraining the LLM. Additionally, an LLM agent has memory and can therefore work in a multi-step process to accomplish more complex tasks and be able to review its work. In comparison to a non-agent LLM—in other words, a zero-shot prompting interaction with an LLM—the performance of the LLM would be drastically reduced for complex tasks [115, 116]. This stems from the LLM not being able to have access to current and relevant knowledge as well as introducing more risks for hallucinations and losing track of the tasks assigned to it. An alternative to a LLM agent would be to pre-train a LLM for the specific task. It is unclear which method would be superior. However, pre-training was deemed too expensive and too large of a time commitment to explore in this thesis. In addition, pre-training is not as adaptive as an agent architecture as the pre-training might need to be continuously performed to keep up to date with current changes. The ideal scenario would be to apply both pre-training of an LLM model and then implement it within an agent architecture as they are not mutually exclusive, however as previously stated pre-training would incur an additional cost that was deemed

unreasonable for the scope of this thesis. This cost comes in the form of having to build a dataset from scratch, which would be time-consuming.

To decide which specific test maintenance use case to implement the agent for, a mind-mapping process was utilised. The process sought to explore what different use cases might look like. This was done by brainstorming which LLM actions and applications (see Section 5.4.1) might fit different triggers (see Section 6.1.2), and how these could be combined to alleviate the different test maintenance problems found through the interviews and survey, as well as ensure they fit with developers' preferences and the teams' way of working. One map was made for each of the high-level test maintenance factors, and one map was made for the use case where the production code had been changed, encompassing all of the low-level test maintenance factors. Keeping multi-agent collaboration in mind, the decision was made to start by focusing the agent on the general use case where the production code had been changed. The agent could then check if any of the low-level triggers had been set off. If test maintenance turned out to be necessary, there were several potential actions the agent could take to help perform it. This agent could then have been incorporated as a part of a multi-agent collaboration handling a larger use case, like one of the high-level test maintenance factors.

In the interest of limiting the scope of this agent to fit within the frame of the thesis, it was decided to further narrow down its use case. The agent would accept the changed production code, and based on the changes decide if test maintenance was necessary and where. In other words, if test maintenance were deemed necessary the agent would point out the test cases that would need to be edited.

4.8 Setup with LLM Agents Design

This section will present the architecture of the prototypes that were implemented throughout the process of this thesis. Several architectures were developed organically through trial and error, and the best-performing ones are detailed here, and evaluated in Section 4.9. In this section, first, the overall architectures of the prototypes are presented. Described next is how several LLMs and agents are used throughout the prototypes, and how their general build and tools are similar. The tools utilised by the prototypes are described thereafter. Next, the models used for both LLM cores and embeddings are described. Finally, some details and differences between the different agents and LLM instances are discussed.

4.8.1 Overall Architecture

Four prototypes were developed in Python using two different architectures. This section will present the architecture of the prototypes and how each component works together. It will also present an overview of the implemented frameworks utilised throughout the architectures.

To improve the performance of the prototypes, a multi-agent set-up was used [117,

104, 16]. This allowed each agent and LLM instance to focus on a particular subtask and further allowed us to provide each LLM and agent with specialised prompts and the agents with specialised tools. All prompts are included in Appendix G.

The agents and the tools were implemented in the LangChain framework [118]. This framework was chosen based on the availability of the methods that we were looking for, such as agent wrapper methods and interactions with vector stores and embedding models. This allowed us to focus on implementing our particular use case rather than reinventing an agent framework.

The main difference between the two architectures stems from a planning agent that coordinates the effort of other LLM instances and agents. One architecture has the planning agent implemented while the other architecture has the LLMs and agents set up in a chain of calls to each other instead.

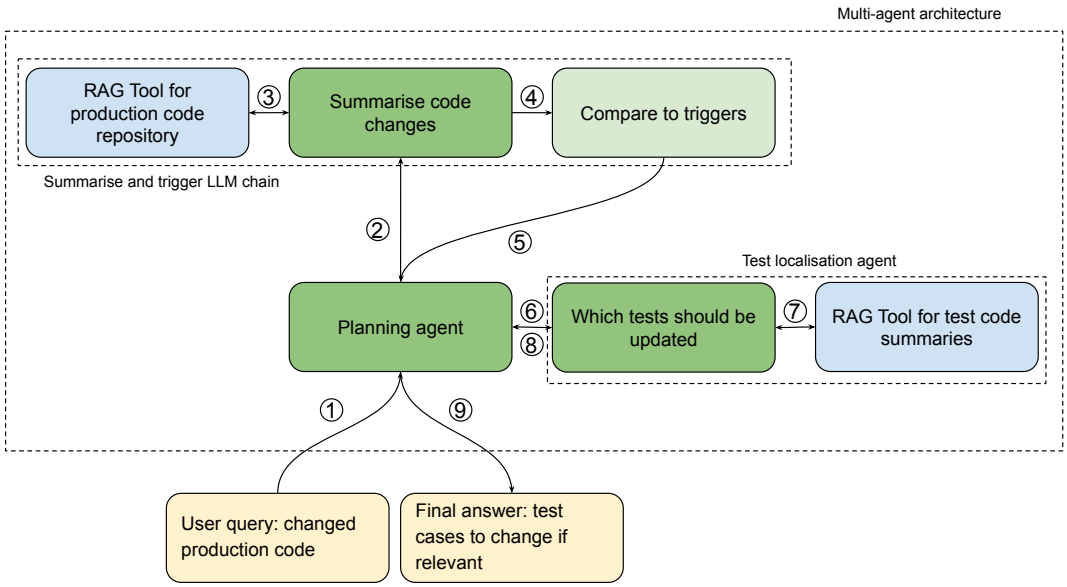


Figure 4.5: Architecture of the multi-agent prototype incorporating the planning agent and summaries of test cases. Yellow is used for input and output, light green for LLM instances, dark green for LLM agents, and blue for tools used by the agents.

The architecture of the prototypes with the planning agent is visualised in Figure 4.5 where the different steps are:

1. The user queries the multi-agent with the changed production code.

2. The planning agent contacts the summarise and trigger LLM chain to identify if test maintenance is necessary based on the code change.

3. The summariser retrieves the old code and compares it to the changed code.

4. The summariser sends a summary of the code change to compare the change

to the test maintenance triggers.

5. The summarise and trigger LLM chain responds to the planning agent about whether test maintenance is necessary. It does this by comparing the summary of the code changes to the triggers found in RQ1.

6. If test maintenance is necessary, the planning agent contacts the test localisation agent to identify which tests need to be changed.

7. The localiser retrieves the test code summaries, to identify what test cases need to be changed.

8. The test localisation agent responds to the planning agent on which test cases need to be changed.

9. The planning agent responds to the user and either says that test maintenance is unnecessary or provides information on which test cases need to be changed based on the original code change.

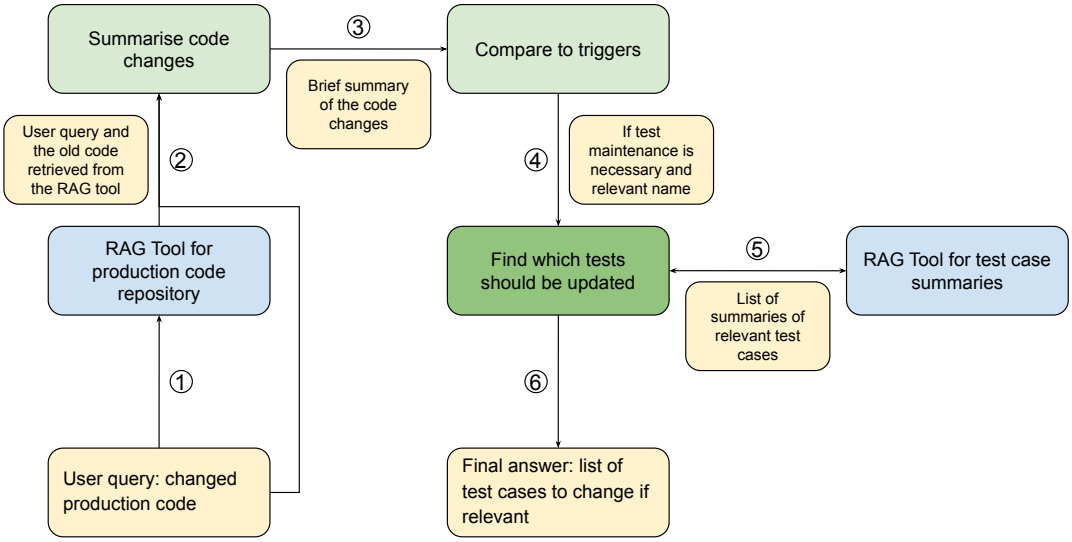


Figure 4.6: Architecture of the LLM chain architecture without a planner agent. Yellow is used for input and output, light green for LLM instances, dark green for LLM agents, and blue for tools.

The architecture of the prototypes without the planning agent is visualised in Figure 4.6 where the different steps are:

1. The user queries the chain with the changed production code.

2. The RAG tool retrieves the old code and sends it to the code summariser. The code summariser also receives the user query (the new code) to compare the two.

3. The summariser sends a summary of the code change to the test maintenance trigger comparator. This summary is one or two sentences long.

4. The trigger agent says if test maintenance is necessary and if so it also includes the name of the relevant method.

5. If test maintenance is necessary the test localiser retrieves test case summaries.

6. The test localiser either outputs the list of relevant test cases or mentions that test maintenance is unnecessary.

All LLMs and agents that are not used for embedding use the Mistral 7B - Instruct model as their core LLM [119]. Ericsson had a few different open-source models deployed at the time of this thesis and of the models available, Mistral 7B - Instruct was deemed the best available model through a series of unstructured tests for our use case. The available models were all tested and inserted into the architecture of the prototypes periodically throughout the development process and the output from several examples and implementations was compared to decide on the most suitable model for each agent.

Each agent has individual memory, planning, and lists of tools available to them. The structures of these individual modules are implemented through the LangChain framework, which provides ready-made methods and functions for implementing memory, planning, and tools to the agents. The agents are ReAct agents that utilise reasoning and display their thoughts and observations as well as what actions they take, such as calling for tools [120]. The recording of the thoughts and observations as well as remembering what has happened is done through an agent scratchpad, which functionally sends the information to `stdout` (the regular output of the environment). Each agent has an individual prompt that gives them instructions on both what their role is as well as how to act.

Each LLM instance that is not an agent is provided with specialised prompts through a chain of calls implemented in the LangChain framework. As the instances are not agents, they do not have access to memory structures that let them follow a train of thought over several queries. However, they give a faster response as they do not have to interface through the agent framework and also run no risk of giving output that conflicts with the formatting required of the agent frameworks. Tools are utilised as separate calls that then send in their results as input to the LLM instances.

4.8.2 Notes on ReAct Framework

As the previous section mentioned, the agents are based on the ReAct framework. ReAct is a paradigm built with the goal of combining reasoning and acting to achieve better results when prompting language models [120]. It uses a thought-action format that proves due to its general and flexible nature work even for diverse and unforeseen tasks. After receiving a prompt, the model goes through a cycle of

Thought-Action-Observation, where an action is taken based on a thought, and an observation is made from the results of the action. If the observation provides a satisfactory answer it stops there, and if not it continues to go through this cycle until an answer is reached.

LangChain uses and builds upon the ReAct paradigm as a way to structure an LLM agent. They provide functions for making an agent that can be customised with tools and specialised prompts, and also provide a so-called scratchpad where the agent can record its thoughts and decisions to help it stay on track. Figure 4.7 shows how the agent uses the ReAct thought process, and Figure 4.8 shows an example of what parts of the process can look like. After receiving a prompt, it decides if it needs to use a tool to fulfil the prompt. If so, it takes an action by calling on an available tool, and makes an observation about the results from said tool. Based on the observation, it once again decides if it needs to use a tool to acquire more information, or if it can answer with the information it already has. One such cycle is called an iteration. If the agent fails to answer using the correct format, it will be reminded what the correct format looks like and asked to answer again. This too counts as an iteration.

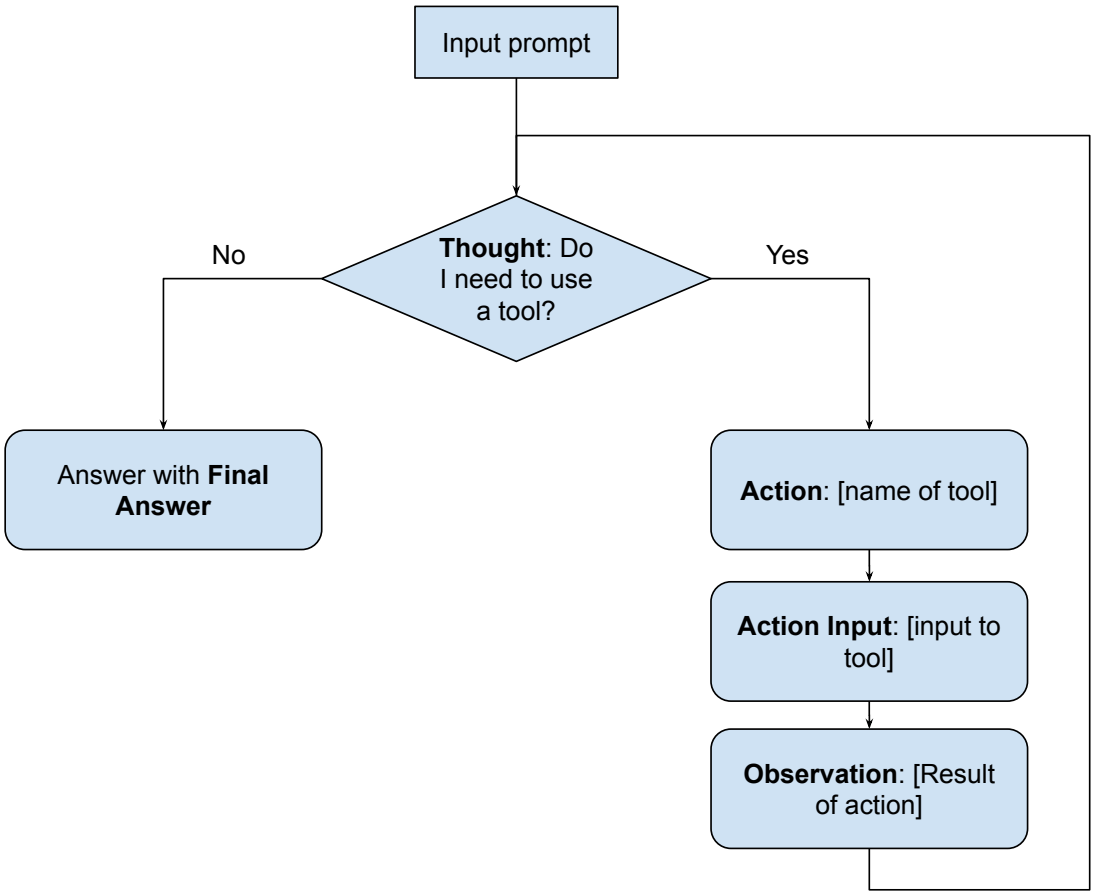


Figure 4.7: Flow diagram showing how a LangChain ReAct agent works.

```
[1m> Entering new AgentExecutor chain... [0m
[32;1m [1;3m Thought: Do I need to use a tool? Yes
Action: mockito_search
Action Input: "Find test cases affected by change in NullResultGuardian class handle method"
```

Figure 4.8: Screenshot showing the thought process of a ReAct agent after receiving a prompt. It firstly makes a thought about how to tackle the problem, then decides on an action. In this case, it decides to use a tool it has been informed it has access to, and as the tool is a RAG tool the agent provides an input about what information it wants from the tool. In this example, the agent has been asked how a change in the production code in the Mockito git repository affects test cases [121].

The prototypes in this thesis used LangChain's ReAct agent framework to implement all LLM agents. Strict limits of a maximum of 5 iterations or 300 seconds per prompt to an LLM were used. Details about the tools provided to the agents can be found in Scetion 4.8.3. The prompts used to set up the agents can be found in Appendix G.

4.8.3 Tool Use

The only type of tool utilised throughout the prototypes is a RAG tool for the code repository. This type of RAG tool provides the LLMs and agents with access to the production and test code and therefore allows the prototypes to give a relevant and significant result regardless of what code base these prototypes are utilised for.

There are two different RAG tools utilised throughout the different components of the architectures. One RAG tool is utilised by the code summariser to have access to the production code to have a context for the code change and be able to know the previous version of the code before the change. This gives the summariser the ability to accurately describe the change in comparison to the previous code and further be able to accurately inform the trigger comparison LLM about what has changed. The second RAG tool is utilised by the test localisation agent and has access to the test code of the relevant code repository. Access to the test code is necessary to be able to find relevant test cases.

Both RAG tools use the Nomic Embedding model [122], chosen for the pragmatic reason that it was the only deployed embedding model within Ericsson that we had access to. Other, non-embedding models were tested but gave worse results for the embedding portions of the prototypes. The vector store used for storing the embedded vectors was the Facebook AI Similarity Search (FAISS) library [123]. A few different vector stores were investigated informally and it was decided that the FAISS library worked best for our purposes.

4.8.4 Models

This section will further introduce the LLM models used in the prototypes. It will first talk about the Mistral 7B - Instruct model that was used as the core of the various LLM instances and agents in the multi-agent architecture. The notable features of the model will be described and explained in rough detail. Afterwards,

the Nomic Embed model that was used as the embedding model for the RAG tools will be described.

Mistral 7B - Instruct is a version of the Mistral 7B model that has been fine-tuned to follow instructions [119]. This was done to increase communication quality and to be able to chat with humans to provide more desirable answers. The Mistral 7B model has some unique features that distinguish it from other models. There are three main notable features of the model: Sliding Window Attention, Grouped Query Attention, and Rolling Buffer Cache.

The first feature of the Mistral 7B model is Sliding Window Attention (SWA) [124, 125, 126]. SWA provides a way to handle larger input for reduced cost. In most models, there is a strict limit to the number of tokens or words in a sequence that can be iteratively predicted (see Section 2.2.1). This limit is often referred to as a context window. An LLM consists of multiple layers of attention to be able to more effectively be able to predict tokens and words. Tokens and words are transformed into vectors to be more understandable to the LLM. SWA aims to increase the limit of the context window arbitrarily by moving the window for every layer of the model but still maintaining some semantic meaning of the previous tokens and words by using the transformed vector of the latest word in the window from the previous layer of the model. Effectively this means that the context window moves through the input as the model progresses through its internal layers and therefore increases the effective context that the model can understand at once, in other words effectively increasing the possible size of the input.

The next feature of the Mistral 7B model is Grouped Query Attention (GQA) [127]. This attention method increases the speed of inference. It does this by grouping queries and calculating the attention of the model for multiple queries simultaneously. GQA is a middle ground between Multi-Head Attention (MHA) and Multi-Query Attention (MQA). MHA has the highest quality since it considers each query individually but it is also quite slow because of this. MQA is very fast as it calculates the attention of every query using the same values but this leads to a decrease in quality. GQA instead groups queries for a faster inference speed like MQA while also maintaining some of the quality from MHA.

The third feature of the Mistral 7B model is the Rolling Buffer Cache [119]. This type of cache has a limited size which decreases memory usage. How it works is that the buffer utilises a first-in-first-out (FIFO) principle in a circular fashion to constantly keep the latest values in the buffer. By keeping the latest values the quality of the results from the model is not impacted severely while the limited size of the buffer decreases the cache memory usage.

The Nomic Embed model is an LLM model specifically trained for embedding [122]. What this means is that the job of the model is to transform data, like text or code, into vectors and then store those vectors separately. The special aspect of this is to give the vectors meaning and be able to relate different vectors, i.e. text and code, to each other and find similarities between them. These similarities are calculated

through cosine similarity in vector space [128]. The Nomic Embed model is based on the BERT model with the intention of increasing the sequence length, in other words, the amount of information able to be processed at once [129]. The model is pre-trained on 29 datasets with paired queries and documents to learn how to relate information with other information [130]. The training of the model is also focused on distinguishing questions and answers so that instead of relating a question with a question since they have similar semantic meaning, the model instead relates answers to the questions to get a more desirable result of an embedding retrieval.

4.8.5 Individual LLM and Agent Differences

This section will go into detail on the specific differences between the two architectures and their components. This includes the connections between the agents and LLM instances as well as the specific implementations of the components of the prototypes.

The implementation of the planning agent is such that the other agents and LLMs that it communicates with are available to it as tools that it can utilise. The prompt of the agent also specifies that it needs to use the tools for solving the tasks asked of it. The prompt further expands on its tasks so that it can work with only receiving changed code as input and have the rest of its instructions in the prompt. After the agent has received the query it itself is free to choose how to approach the problem, which tools to use, and when it considers the problem solved. These decisions are made by the model powering it. The planning agent is not the only agent making decisions, but it does make the overarching plan and has the final say in how to proceed with that plan, which is why it has been given its name. It is the only agent that gets to see the entire picture, as it is the agent receiving the users query, and can choose how much detail from it to share with the other agents. If it decides the output from the other agents did not answer the question it asked them, it can call upon them again, something the other agents are incapable of. This flexibility and holistic view is something that is missing in the chain architecture, as it cannot iterate and has no part responsible for overseeing the entire process.

The implementation of the "summarise and trigger" LLM chain consists of two different LLM instances, as can be seen in Figure 4.5. The summariser is implemented as an agent with tool access, while a simple LLM instance was used for comparing the summary of the code changes to the triggers. The LLM instance is given a list of the triggers (from RQ1) in its prompt and compares the triggers to the summary of the code change it was given by the summariser agent. The decision to use an LLM instance instead of an agent stems from the fact that this part of the multi-agent simply has to compare a summary of a code change to a list of triggers provided in the prompt. This use case does not need an agent structure and therefore it was decided that a simpler setup would lessen the risk of errors. The summariser and the trigger comparing LLM each have their prompt that specifies their tasks based on the input they receive. The two LLMs are connected in a chain instead of having one be a tool of the other, a chain refers to a setup where one model passes its output to the next model as their input and the last model's output will return

the final output of the entire chain. The test localisation agent has its prompt and access to a RAG tool that consists of an embedded vector store of the latest version of the relevant code repository, this includes both production code and test code.

For the architecture without the planning agent, the code summarising agent is instead implemented as an LLM instance. The RAG tool for the production code is called separately first from the user query and then the results from the RAG tool along with the user query are sent to the LLM to be inserted in the prompt instead of utilising a tool structure. The agents and LLM instances are also connected in a chain where the outputs of the LLMs and agents serve as inputs for the following LLM or agent. Another difference comes from the fact that the test localisation agent works directly with output from the trigger comparator. Therefore it chooses on its own whether to search for test cases or not depending on whether test maintenance is necessary according to the trigger comparator.

The four setups used for evaluation stem from the two architectures and differ in only one aspect beyond the differences already described. All prototypes look at test cases to see what needs to be updated. One setup of each architecture looks at the test cases as they are written in code while another setup of each architecture looks at plain text summaries of the test cases instead. The test cases were summarised with the the help of an LLM, with a prompt that can be found in Appendix G.5.4. The architectures of the models are otherwise identical between the architectures seen in Figures 4.5 and 4.6.

4.9 Evaluation of Setup with LLM Agents

To evaluate the four LLM and agent setups, their results were compared to the ground truth found in a repository's commit history. This is built on the assumption that production code and test cases that are changed in the same commit are examples of co-evolution, that the test cases have been changed because of the changes in the production code. This section will first describe the dataset built for the evaluation, before describing the evaluation procedure.

The dataset was built from the git history of one of Ericsson's Java repositories. The repository concerns a single service within the data processing domain. The test part of the repository contains unit and integration tests. The dataset covers 57 commits which together have 374 code changes. Each commit has been split into the individual code changes and all of the test cases that were modified in that commit. An individual code change is defined as a chunk of production code (containing the changed code) of around 20 lines of code or an entire method, whichever is smaller. The reason for using individual code changes instead of entire commits all at once was taken due to the the models' limited ability to understand and work with large contexts. The following types of code changes were excluded from the dataset:

- Commits with no changes to the production code.
- Changes to production code that modified imports or comments.

- The addition and removal of files and methods in the production code. This evaluation focused solely on modifications.
- All changes to test code that were not within a test case, such as changes to test support code.
- The addition and deletion of files and new test cases in the test code. Deletion of existing test cases was included as it can be seen as a modification of an existing test case.

For each commit, the evaluation worked the following way:

1. The information stored in the RAG tools about the code and test cases was updated to reflect the current state of the repository.
2. Each individual code change was separately fed into the agent setup, which would output test cases that needed to be updated or modified based on that change. These test cases suggested by the agent were saved as results.
3. The suggested test cases from all code changes in that commit were combined into a set, which was compared to the test cases that had actually been changed in the commit, the ground truth.

This procedure was repeated for each commit. A true positive (TP) was defined as a correctly identified test case, a false positive (FP) as a test case that was identified for modification that should not have been identified, and a false negative (FN) as a test case that was not identified for modification that should have been identified.

From this, the precision can be defined as $\frac{TP}{(TP+FP)}$, i.e. how many of the identified test cases were correct. The recall can be defined as $\frac{TP}{(TP+FN)}$, i.e. how many of the correct test cases were identified.

The calculations were complicated by the fact that sometime the correct number of test cases to identify was 0, which would lead to division by 0. In these cases, if the prototype correctly identified that no test cases needed to be modified, recall and precision were set to 1 as the correct conclusion was reached. If it instead incorrectly identified some test cases, the recall would be manually set to 1 (all correct test cases—in this case, none—were identified) and the precision to 0 (none of the test cases it identified were correct).

The recall and precision were used to calculate the F1 score, $2 * \frac{(precision*recall)}{(precision+recall)}$, the harmonic mean between the precision and the recall. Apart from calculating this result for every commit, the mean and aggregated precision, recall, and F1-score were calculated as well. The mean was calculated by adding the individual scores of each commit together and dividing by the total number of commits. The aggregated score were calculated by summing together all the true positives, false positives, and false negatives for the original calculations. The F1 score offers an overall view of the performance of the prototype. We use the F1 score instead of

accuracy, as true negatives (test cases that were not found and should not have been found) would inflate the calculations. This inflation stems from the fact that there are hundreds of test cases in the repository and for most commits less than 20 test cases were modified. This means that the true negatives would vastly outweigh the true positives, false positives, and false negatives.

5

Results

This chapter presents the results of the different activities in the case study. The chapter will first present the results of the two literature reviews, the survey, and the thematic analysis of the interviews, finishing with the evaluation of the prototype.

5.1 Literature Review of Maintenance Factors

The literature review on test maintenance produced several factors for test maintenance activities being required based on production code modifications. Two main groups of factors were found: documentation factors and factors related to the functionality of the production code. What is meant by documentation is text or non-executable code elements meant to provide information to a reader or to analysis software, e.g. comments, annotations, certain keywords, etc. Functionality on the other hand refers to code that can be executed and therefore provides some functionality, e.g. methods, classes, statements, etc. Functionality factors were the most relevant for this thesis. Within functionality, several sub-groupings were found, and these are all reported in Figure 5.1. A more detailed description of each factor within each grouping can be found in Tables 5.1, 5.2, 5.3, and 5.4. These results were used to answer RQ1.

There is some difference in granularity between the groupings, as well as some overlap. For example, the *New method* grouping could have been a part of the *Add functionality* grouping. The reason for this is that different papers reported on different granularity levels. The groupings were created with this differentiation in mind, to try and preserve as much granularity and information as possible without having to exclude papers.

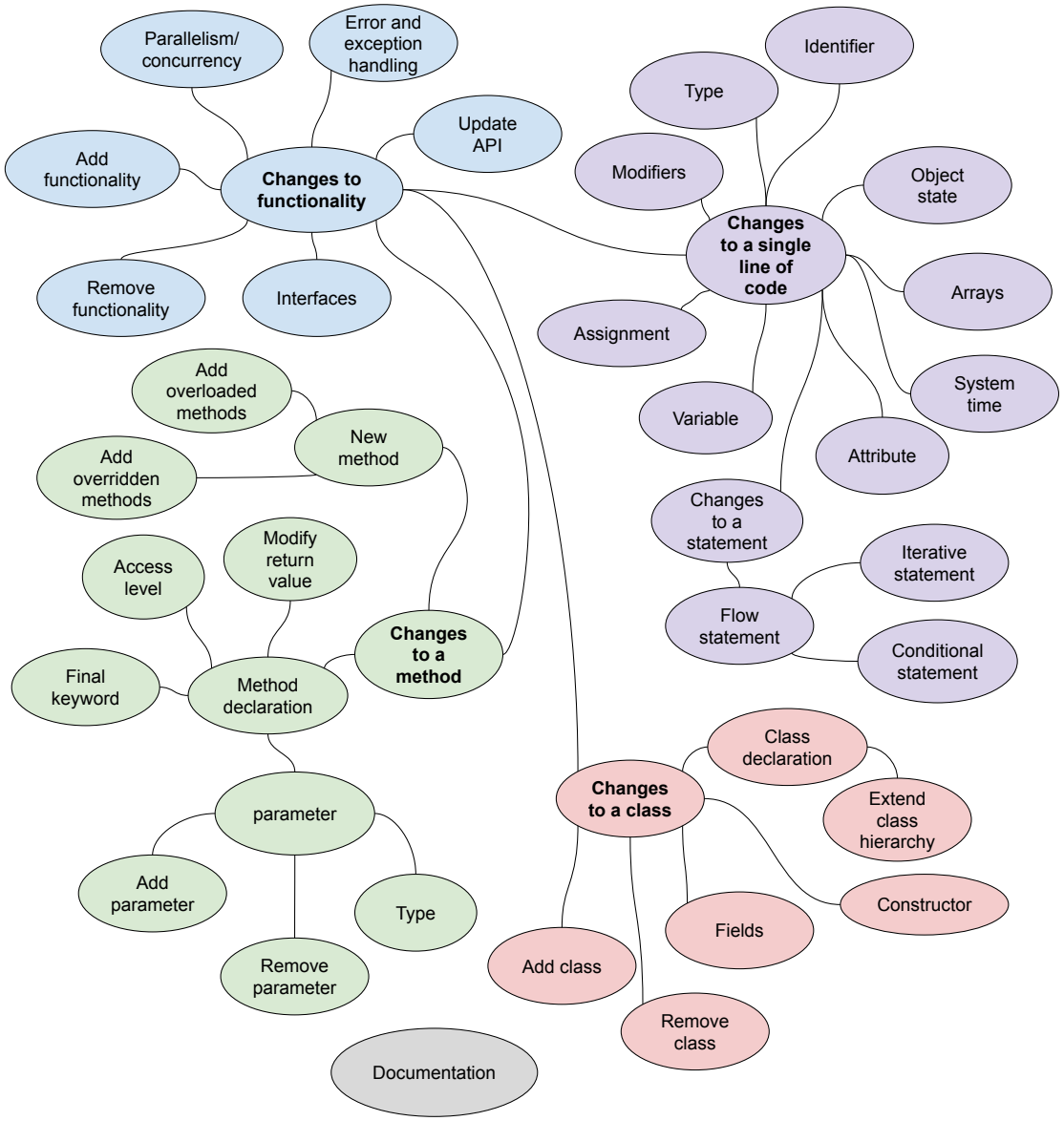


Figure 5.1: Overview of groupings of test maintenance factors found from the literature review. Colour is used to more easily see which factors are grouped. Blue = general functionality not connected strongly to another group, green= changes to methods, red = changes to classes, purple = changes to a single line of code.

Table 5.1: Test maintenance factors that affect the general functionality of the system. These factors are found in the production code.

Type of Change	Description	Sources
Changes to functionality	Changes made in production code that affect system behavior.	
Add functionality	Implementing new functionality or a new component that did not exist before. This is a general grouping for sources that did not specify the type of functionality that was added.	[69, 63, 74]
Add interface	Implementing a new interface that did not exist before.	[131, 63, 73, 74]
Class	Larger grouping of factors related to changes in classes. See Table 5.2 for details.	See Table 5.2.
Error and exception handling	Changes made to code that handles errors and exceptions, such as throw, try, and catch keywords.	[71]
Method	Larger grouping of factors related to changes in methods. See Table 5.3 for details.	See Table 5.3.
Parallelism and concurrency	Changes to how the system handles parallel threads and concurrency, such as the synchronised keyword.	[71]
Remove functionality	Code that implements certain functionality is removed. This is a general grouping for sources that did not specify the type of functionality that was removed.	[69, 63]
Single line of code	Larger grouping of factors related to changes in single lines of code. See Table 5.4 for details.	See Table 5.4.
Update API	Changes to how an external API is invoked.	[22]

Table 5.2: Test maintenance factors that are changes made to a class.

Type of Change	Description	Sources
Changes to a class	This grouping covers changes made to classes, mainly affecting how the class itself functions.	
Add class	Implement a new class that previously did not exist.	[65, 131, 71, 70, 63, 64]
Class declaration	Changes made to a class declaration. This includes the sub-group Extend class hierarchy.	[65, 63, 2, 64, 73, 74]
Constructor	Changes made to the class constructor, for example, creating a constructor or adding a constructor parameter.	[70]
Fields	Changes made to a class' fields, i.e. the internal variables of the class.	[65, 2, 64]
Remove class	Remove a new class that previously existed.	[65, 63, 64]

Table 5.3: Test maintenance factors that are changes made to a method.

Type of Change	Description	Sources
Changes to a method	This grouping covers changes made to methods that mainly affect how the method itself functions, including the method's body.	
Add method	The implementation of a new method. This includes the sub-groupings Add overloaded method and Add overridden method.	[131, 70, 64, 73, 74]
Method declaration	Changes made to a methods declaration and signature. This includes the sub-groupings Access level (both widen and decrease), Final keyword (adding or removing overridability), Return value (changing the value or type), and Parameter (adding, removing and changing the type of a parameter)	[131, 65, 71, 70, 63, 2, 64, 73, 74, 72, 71, 63, 73, 72]

Table 5.4: Test maintenance factors that are changes made only to a single line of the production code.

Type of Change	Description	Sources
Changes to a single LOC	Changes made to the production code that are the size of a single line and are not related to the other groupings.	
Arrays	Changes to arrays, specifically adding an array and changing the access level.	[71]
Assignments	Changes to the value assigned to a variable, as well as compound assignments.	[71]
Attribute	The declaration of attributes as well as extraction for assertion.	[65, 70, 64]
Modifiers	Changing and updating modifiers, e.g. the public keyword. This grouping groups sources that did not specify what the modifier belonged to.	[71, 22]
Identifier	Changes to variable names as well as other identifiers such as package names.	[71, 22, 2]
Object state	Remove or add object state.	[63]
Override system time	Override the system time.	[70]
Statement	Flow statements, conditional statements, Iterative statements	[65, 71, 63, 2]
Type	Modify the type of an object, either the keyword or by casting.	[71]
Variables	Changes to the declaration of a variable.	[71]

5.2 Thematic Analysis of Interviews

Based on the interviews, a thematic analysis was conducted to better understand the test maintenance process, and thereby better be able to understand how LLMs can facilitate the process. In total five major themes were found: *Ways to Assure Quality*, *Reasons to Change Tests*, *Issues Related to Test Maintenance*, *Wishlist for Tool Support*, and *Attitudes Towards Generative AI*. These each have multiple sub-themes. An overview and description of the themes found can be seen in Table 5.5. Descriptions of each sub-theme can be found in the respective theme's section. The percentages of the number of occurrences and the number of occurrences of each theme and sub-theme can be seen in Figure 5.2. The following sections will explore each theme in detail. Quotes are attributed to the ID of the interviewees, see Table 4.1. The themes were used to partially answer RQ1 and RQ2.

Table 5.5: Overview of themes.

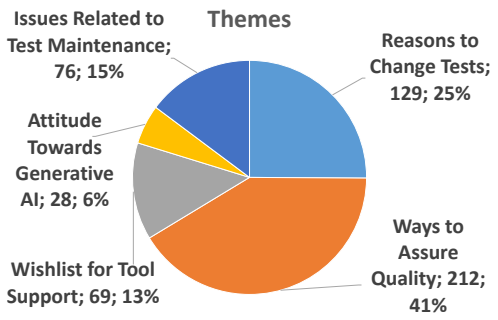
Theme	**Description**
Ways to Assure Quality	Ways to ensure that the tests and test suite hold a high quality, which in turn carries over to the final product.
Reasons to Change Tests	Reasons for performing test maintenance.
Issues Related to Test Maintenance	Current issues experienced with test maintenance.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

5.2.1 Reasons to Change Tests

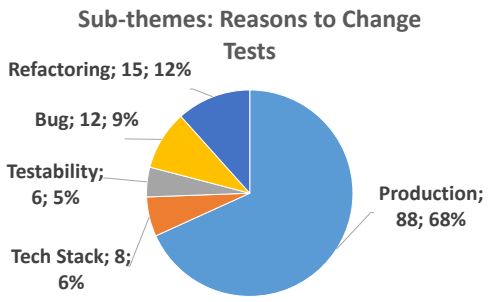
This theme concerns the reasons for modifying the test suite. Making changes in the test suite is, in and of itself, part of test maintenance and, therefore, the reasons to make the changes are of high interest to better understand the test maintenance process. There are a variety of sub-themes concerning the differing reasons to make changes. An overview of them can be seen in Table 5.6, a more detailed description will follow below.

Table 5.6: Overview of Reasons to Change Tests' sub-themes.

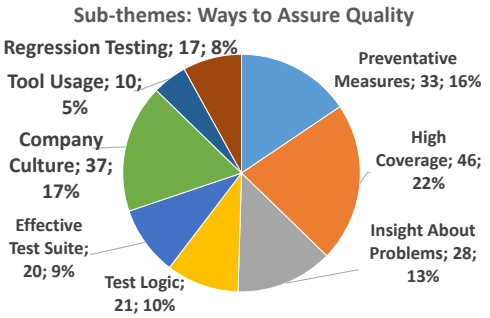
Sub-theme	**Description**
Production	Changes on the production side cause changes in tests.
Tech Stack	Changes in the tech stack necessitate changes to tests.
Testability	Test and production code changed to be easier to test.
Bug	The discovery of a bug leads to changes in tests.
Refactoring	Tests need changes when production code has been changed without its functionality changing.



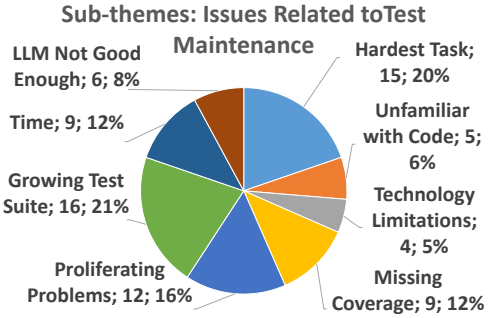
(a) Themes



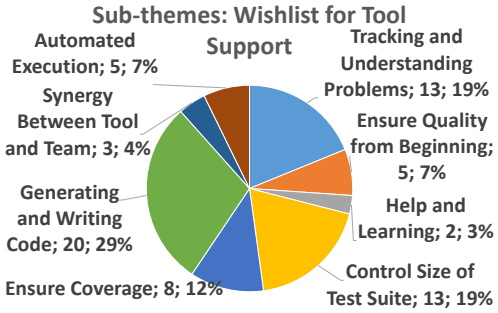
(b) Sub-themes: Reasons to Change Tests



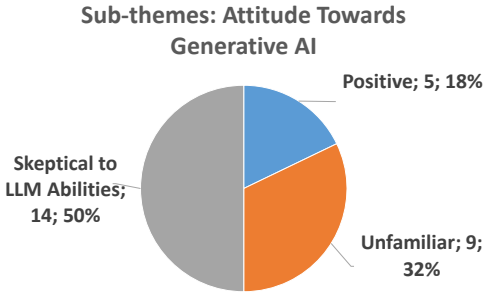
(c) Sub-themes: Ways to Assure Quality



(d) Sub-themes: Issues Related to Test Maintenance



(e) Sub-themes: Wishlist for Tool Support



(f) Sub-themes: Attitudes Towards Generative AI

Figure 5.2: Number and percentage of codes related to each theme and sub-theme. Labels show Name; Number of occurrences; and Percentage of total occurrences. An occurrence is a mention of a code by an interviewee. A theme's size does not necessarily indicate its importance.

Production

The most important sub-theme within this theme, with 68% of the occurrences being attributed to this sub-theme, is that changes in the test suite are caused by changes in Production. This sub-theme showcases how production code and test code are closely intertwined. From collecting responses from the interviewees and analysing their answers it seems that most, though not all, test maintenance activities (and non-maintenance testing) go hand-in-hand with production code changes and are often caused directly by them as well. This can be new feature implementations, changes in requirements, modifications to methods, or changes to conditional statements.

> "I would say that probably the production is driving us to change testing. I mean both finding bugs and adding features. Those two things really drive up the the testing." - P2

> "We have to deal with tests pretty much always when we change the product production code." - P3

Tech Stack

Another sub-theme is the Tech Stack. Changes in the tech stack can mean that code that worked before no longer function as intended. This means that either the production or test code might need to be modified to accommodate the new situation. This then leads to a further need for test maintenance activities. Additionally, the kind of testing and the ways to test can be limited by a changing tech stack which can then lead to a need to edit the test suite.

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

Testability

The sub-theme Testability concerns the changes made to tests to make sure that testing becomes more efficient and effective. To ensure the testability of the production code both test and production code may need to be changed. This includes various efforts to make components more modular or to make them able to be mocked in tests:

> "[...] there is some cases where we've had that code has been written that's not really testable. Let's say we have a structure where we can't mock away stuff or things like that. And then I would say sometimes the testing is driving changes in production. So we try to make it more modular [...]" - P2

Also included are certain development styles like test-driven development where the tests are written before the production code:

> "[...] my personal way of doing it, I start with the test and then build the code based on that and therefore it does definitely change the approach that I take to testing in general. Because the test code isn't written after the feature is written. The test code is part of the feature you know, every time I put down two lines of code two lines of test code goes with it. You know you build it piece by piece by piece by piece by testing and testing and testing continuously." - P1

This sub-theme in particular demonstrates that changes in the test suite should go beyond making sure that tests pass and have high coverage. Instead, the sub-theme shows that the complexity and understandability of the test suite should also be a driving factor in modifying it. It therefore is quite closely connected to the Effective Test Suite sub-theme from the *Ways to Assure Quality* theme (see Section 5.2.2) in how the values can serve as reasoning to make changes in the test suite itself.

Bug

Bug is another sub-theme when it comes to reasons to make changes in the test suite. Testing itself is about making sure that the code works as it should and, therefore, the discovery of a bug will generally necessitate changes to the test suite itself.

> "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

This can include the creation of a test that reproduces the bug or the addition of new test cases to cover under-tested code in areas where a bug was discovered. Alternatively, this may include rectifying bugs in the tests themselves. If tests themselves contain bugs then the entire process of testing is at risk and they need to be rectified whenever they are noticed.

Refactoring

The sub-theme of Refactoring concerns changes that do not necessarily impact the functionality of the production code. This could be from a change in the architecture, from having to iteratively work through tests or other code, or to simply making small improvements to whatever code artefact already exists because there is a way to improve it.

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

Though the requirements and logic remain unchanged, tests may still need to be modified to fit the changes. This sub-theme shows how it is hard to completely separate logic and code. Though the functionality remains unchanged, how that functionality is tested may still require modifications to fit the change.

5.2.2 Ways to Assure Quality

As the name suggests, this theme encapsulates the different ways the interviewees talked about ensuring the quality of tests and test suites, both their initial creation and further maintenance. This quality is in turn carried over to the final product. *Ways to Assure Quality* has eight sub-themes. An overview of them can be seen in Table 5.7. Several of these describe how to shape the testing process to ensure faults in the code are found before the code is deployed, but the sub-themes also describe the importance of having a high test quality to ensure future code maintenance and feature additions go as painlessly as possible. Lastly, the sub-themes additionally cover how to handle unexpected faults and how to modify the tests to ensure they do not reoccur.

Table 5.7: Overview of Ways to Assure Quality's sub-themes.

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

Preventative Measures

The Preventative Measures sub-theme expresses the value of stopping problems before they occur, by spending more time and effort on finding and fixing faults before the modified code is deployed. Though this analysis is mainly interested in test maintenance, interviewees stressed its importance both when working with production and test code. This sub-theme first encapsulates the general philosophy that it is better to have a more rigorous and time-consuming testing process than to fix faults after the code has been deployed, and second describes ways to implement this rigour. A higher initial quality of the code leads to fewer issues later, and a high initial code quality is created by having a rigorous process in place that minimises the chances of faults slipping through. Though this leads to more time spent earlier in the process, time is still saved compared to having to correct the faults later. As one interviewee said:

> "It's less work to prevent an error than it is to go and fix the error." - P1

If a fault slips through it will likely be discovered by a customer, which can damage their confidence in Ericsson's products. Examples of preventative measures include

doing code reviews and communicating and collaborating with affected teams about removing tests. One interviewee described how removing a test had led to a fault in the code slipping through, and the steps they had taken to prevent it from happening again:

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

Another example given was how putting higher-quality code into the CI pipeline makes it run more smoothly. The further into the pipeline a fault is discovered, the more tests have to be rerun to verify it has been fixed. By relying less on the CI pipeline to discover faults, changes in the code can be deployed faster. Some preventative measures, e.g. having a high code coverage and doing regression testing, were emphasised enough that they were given their own sub-themes.

High Coverage

The next sub-theme is called High Coverage. Having a high coverage of code and high input and expected output diversity in the test cases has two main benefits: confidence and validation. Covering a high proportion of potential inputs, outputs, branches and exceptions verifies that the changed code behaves as expected. This in turn lets the developers be confident about the code, which allows effort to be spent where it is needed, and not worry about unforeseen problems.

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

This was achieved by having a known predetermined strategy for writing and modifying tests, where steps were outlined to ensure that all paths would be covered.

Insight About Problems

Insight About Problems handles steps taken to properly understand a fault and why it has occurred before starting to plan or implement a fix. Taking these steps at the beginning of the process ensures that less time is wasted when solving it, as properly understanding the problem brings an understanding of what type of solution would be suitable. One interviewee said when describing how to update a test case:

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

Taking the time to understand the fault and creating a ticket also allows the team to match it with the right developer, as different developers have different strengths and competencies. P1 described how to solve a more complex bug after the team had been alerted about it:

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

Naturally, it is better to prevent the problem from occurring in the first place, however, if it has already occurred it needs to be solved as efficiently as possible.

Test Logic

Next is the sub-theme Test Logic. As suggested by the name, this sub-theme describes how it is not the code that should be tested, it is the logic the code represents. It emphasises testing the different possible outcomes that can occur when running a piece of functionality, as well as testing one part of the system's logic at a time, i.e. modularity, as illustrated by this quote:

> "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

There are some similarities between the High Coverage sub-theme and Test Logic, however High Coverage places importance on ensuring every part of the code is runnable and tested, while Test Logic emphasises ensuring that the test code accurately represents the requirements.

Effective Test Suite

The sub-theme Effective Test Suite contains current practices and opinions to reach an effective test suite. Each test case inside the suite should be meaningful, and test cases should regularly be evaluated to ensure they are up-to-date, relevant, and of high enough quality. This was illustrated by interviewee P6:

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

By removing less relevant test cases and keeping the test suite small, it can be run faster, which facilitates a faster development and deployment cycle. Not running irrelevant test cases also ensures the hardware is used efficiently.

Company Culture

Company Culture captures the importance of valuing testing activities. Prioritising testing, and having a rigorous testing routine, leads to fewer faults slipping through which in turn leads to fewer bugs disrupting production and deployed code. The essence of the sub-theme is how mindset and the company affect the final result. A team that takes pride in delivering high-quality code will need to edit that code

less often. Further, they will acknowledge that there is always room to improve the testing and will be open to change and feedback. One interviewee described when talking about updating test cases:

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

Tool Usage

The sub-theme Tool Usage encapsulates how tools can help ensure quality. Two different types of tools were described. Firstly, Linters and other static tools help developers quickly discover problems, such as code smells, in the code before committing it. Secondly, the importance of continuous integration (CI) was emphasised. An important aspect of CI with regards to test maintenance is how it helps with regression testing, but much like Linters, it can also help enforce certain quality standards, e.g. ensuring a minimum amount of test coverage.

> "[...] if I'm breaking anything of the legacy code, I will notice it because the CI tells me something is wrong [...]" - P8

Both types of tools have their limitations but are nonetheless important ways to discover problems earlier than they would otherwise have been discovered.

Regression Testing

The last sub-theme is called Regression Testing. Great importance is rightly placed on ensuring that integration with new code does not affect existing code that has passed testing. This sub-theme encapsulates that importance, as well as the different steps taken to ensure nothing has been broken. One example provided by an interviewee in the context of fixing a bug:

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

Code is naturally interconnected, and it is critical to understand how a change in one part affects another, to prevent unintended side effects.

5.2.3 Issues Related to Test Maintenance

Many different issues regarding test maintenance were raised during the interviews, and have been gathered under this theme. In general, these are all issues that hinder or slow test maintenance, as well as issues that cause a need for test maintenance. In addition, developers' opinions on what is the hardest or most effort-intensive task as well as their doubts about the use of LLMs to aid with development and testing have been captured here as well. The issues are described closer in the following sub-themes, of which an overview can be found in Table 5.8. Some are closely connected to test maintenance, while others can be generalised to testing or software engineering.

Table 5.8: Overview of Issues Related to Test Maintenance's sub-themes.

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

Hardest Task

The sub-theme Hardest Task captures what developers thought was the hardest part when changing and updating the test suite. Four different sub-sub-themes were discovered: **Find Problem** (identify what needs to be done), **Update Test** (modifying an existing test), **New Test** (having to add a new test to the test suite), and **Write Production Code** (it is harder to make the changes on the production side than the the test side). Opinions were quite evenly split between the four sub-sub-themes on which was the most difficult or effort-intensive. Out of the 15 occurrences of the sub-theme, the sub-sub-themes have 4, 3, 5, and 3 occurrences respectively. These differences of opinion appear to come both from differences in experience, as well as from which approach is taken to testing. The following quotes illustrate this. Participant P2 thought updating was harder:

> "Yeah, I mean updating is harder. [...] but now it's more like something is wrong, either with the functionality itself or with the test. So first you have to identify what am I changing and why, and then maybe doing it is not that hard. Yeah, because then you have these first steps, right? [...] the manual work there is already done, right?" - P2

Participant P1 thought writing new test cases was harder:

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

There is some overlap between the different sub-sub-themes, as doing one will generally involve doing another as well.

Unfamiliar with Code

The Unfamiliar with Code sub-theme highlights the problems that come with not knowing how parts of the code work.

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

This in turn makes the code harder to interact with. It is more difficult to both find and change existing test cases and would sometimes lead to new test cases being added instead, and as a consequence unnecessarily enlarging the test suite. Somewhat deceivingly, a large test suite can help ensure a high coverage, which can in turn help developers ensure the code at least works. That said a large test suite can lead to developers being unable to familiarise themselves with the test suite, and can make it hard to navigate and efficiently utilise. Old or duplicate tests worsen this.

Technology Limitations

The Technology Limitations sub-theme encapsulates how particularly the tech stack can provide limits. An example is how certain libraries or packages might require the use of a specific version of a programming language, or how they might be unable to use other technologies since they do not guarantee data privacy.

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

This issue was generally one that was accepted as an unavoidable part of development and maintenance.

Missing Coverage

The Missing Coverage sub-theme explains that when tests do not cover enough of production code faults can slip through. Missing coverage also leads to less confidence in the production code. In some cases, a loss in coverage can be caused by changing or removing test cases, as described by one interviewee:

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

In other cases, the coverage was never there to start with.

Proliferating Problems

The next sub-theme is Proliferating Problems, which describes how problems in code (e.g. faults, dependencies, missing coverage) are often interlinked, and how trying to fix one problem can in turn lead to more problems. For example, dependencies between tests make it hard to modify or update them without affecting other parts of the code, which in turn makes developers less likely to touch them, because of the

extra complexity and effort. Another is how making a change in one test can lead to unforeseen consequences, such as missing coverage or faults slipping through.

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

These problems demand time and effort and prevent other work from being done.

Growing Test Suite

Growing Test Suite is a sub-theme about how when the test suite increases in size it gets harder to manage and navigate, and the execution time increases. A test suite grows naturally as the product is developed, but can continue growing afterwards as requirements change if care is not taken to remove outdated test cases, as well as making sure to modify test cases instead of adding new tests. The problems were well illustrated by these interviewees:

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

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

A growing test suite also feeds the previously explained Unfamiliar with Code sub-theme, as developers have a harder time keeping track of the test suite.

Time

The Time sub-theme encapsulates two different time aspects. Firstly, when a deadline is approaching or stress is involved, testing may be neglected. Secondly, manual testing and other manual processes are time-consuming and slow, as illustrated by:

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

This is an issue that feeds itself; testing, when done manually, takes up a lot of time, leaving less time for the later stages of testing and quality assurance.

LLM Not Good Enough

The last sub-theme is called LLM Not Good Enough. It describes the issues experienced by the interviewees when using an LLM to aid with development and testing, e.g. for code generation. It deals not with the general attitude towards LLM tools, which is handled by the *Attitudes Towards Generative AI* theme (see Section 5.2.5). An experience of the code generated by LLMs not being good enough or that it was

not worth the effort it took to generate or modify it was expressed, as well as the LLM needing human oversight.

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

The issue lies both in the quality of the LLM's output, as well as the interviewees' expectations. It should be noted that none of the interviewees had tried to use LLMs for test maintenance and that this experience came from other interactions.

5.2.4 Wishlist for Tool Support

This theme is focused on collating the requests and thoughts of the interviewees regarding tool assistance for test maintenance. The type of desires have some variety but often correlate with topics brought up in the *Issues Related to Test Maintenance* theme (see Section 5.2.3) and the *Ways to Assure Quality* theme (see Section 5.2.2). An overview of *Wishlist's* sub-themes can be found in Table 5.9, and a more detailed description will follow below.

Table 5.9: Overview of Wishlist for Tool Support's sub-themes.

Sub-theme	Description
Tracking and Understanding Problems	Knowing how and where problems might occur.
Ensure Quality from Beginning	Preventing bad habits and problems from occurring.
Help and Learning	Easier to find and understand new information.
Control Size of Test Suite	Prevent test suite from growing too large and keeping the tests relevant and efficient.
Ensure Coverage	Prevent and find faults by having a high code coverage.
Generating and Writing Code	Generating code for tests and improving already written code.
Synergy Between Tool and Team	Tool should fit into the team's workflow.
Automated Execution	Automated execution of tests.

Tracking and Understanding Problems

The Tracking and Understanding Problems sub-theme expresses the desire for help in knowing the how and the where of any problems that might occur. This includes both faults in the code as well as issues that stem from test maintenance methods and practices.

> "Today we have a tool that will say [...], is it a product failure or is it an environment failure because that's something happening here in the lab. We don't want to spend time on the environment problems, just fix them. We want to spend time on product failures." - P8

The question of observability and explainability relates to the entire process environment, both within test suites as well as the surrounding tools that are utilised

for various tasks. Getting information about problems that might appear without having to scrounge through everything in the surrounding environment would simplify and ease the workload, both when it comes to test maintenance as well as other activities.

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

They also expand on how their current processes are very expensive in man-hours, serving as another example of how tools might help their processes and activities. Another interviewee said:

> "It takes a lot of time to actually take out a faulty something that's a failure. How do we correct the failure?" - P8

These quotes exemplify a desire to get assistance from tools in identifying anomalies, identifying the cause of an anomaly, and tracking whether a fault has been corrected. These examples relate to the desire to lessen some issues from the Time sub-theme in the *Issues Related to Test Maintenance* theme (see Section 5.2.3). One way a tool could help with tracking and understanding problems is to identify locations where tests are insufficient or may need to be updated.

Ensure Quality From Beginning

The Ensure Quality From Beginning sub-theme focuses on preventing bad habits or other problems from occurring, thereby lessening the need for test maintenance. An example of a current bad habit and a possible solution is illustrated by the following quote:

> "[...] I think that would be one thing, to increase coverage, we would have something telling us coverage is bad and maybe forcing it a little bit more on us than just because you run Pytest and it passes and then you're happy usually but you don't look at the coverage report." - P2

An LLM could further help ensure the quality of a test suite by checking if changes in tests fulfil quality requirements before tests are committed. There are similarities between this sub-theme and the Preventative Measures sub-theme under the *Ways to Assure Quality* theme (see Section 5.2.2) in what the interviewees are trying to accomplish. The difference is that this sub-theme pertains to the desire to accomplish these tasks with the help of tools as well as improving the practices that are already in place. The Preventative Measures sub-theme instead expounds on the current processes to ensure this behaviour.

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

Help and Learning

The Help and Learning sub-theme comes from the desire to mitigate a lack of knowledge with the help of tools, whether that lack of knowledge comes from unfamiliarity with a topic or a lack of experience. This could for example be learning how to write tests in a new language or testing framework or get tips on how to make a method more testable.

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

The sub-theme has strong correlations to the Unfamiliar with Code sub-theme from the *Issues Related to Test Maintenance* theme (see Section 5.2.3). An example of a way for a tool to help with learning is to have an LLM provide descriptions and instructions on how a new testing framework should be implemented and describe it in terms the user is familiar with.

Control Size of Test Suite

The Control Size of Test Suite sub-theme is related both to the Growing Test Suite sub-theme from the *Issues Related to Test Maintenance* theme (see Section 5.2.3) as well as the Effective Test Suite sub-theme from the *Ways to Assure Quality* theme (see Section 5.2.2). The distinction comes from this sub-theme concerning what solutions can be applied and what tools can be utilised to solve the issues that exist beyond the current methods when it comes to keeping the size of the test suite minimal and efficient.

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

Controlling the size of the test suite could be done by a tool providing information on what test cases overlap with each other or are not providing value so that the test cases can be modified or removed.

Ensure Coverage

The Ensure Coverage sub-theme is closely related to the High Coverage sub-theme from the *Ways to Assure Quality* theme (see Section 5.2.2). Since having a high coverage is recognised as an important way to assure the quality of the final product, getting help achieving a high coverage is important as well. Multiple interviewees expressed the need to receive tool-related assistance with this topic.

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

Another reason for desiring assistance with coverage is to ease the workload, as

achieving and maintaining it can be time-consuming. A tool could help increase the coverage by suggesting test input to apply in new tests as well as augment the input applied in existing tests.

Generating and Writing Code

The largest sub-theme for the *Wishlist for Tool Support* theme, 29% of all occurrences, is the Generating and Writing Code sub-theme. Since writing code is one of the main activities of testing and developing in general it is not unexpected that interviewees want help with this step. This help could come in many forms, from generating entire test cases to simply doing an outline of a test as a template or checking if some existing tests could be modified to suit the purpose of the task.

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

> "So let's say I have a method and it takes a list and a string. That's simple. Maybe it should create the empty list on this method and an empty string or something like that. Just have something here so I don't have to think about it. You know, tab between all the windows and yeah." - P2

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

> "[...] you can't because the files are big and there are a lot of different files. You cannot hand-write it and you can't generate it manually. You have to have some automation tools [...]" - P7

The sub-theme serves as an indicator of where the issues lie and how their work can become more efficient.

Synergy Between Tool and Team

The Synergy Between Tool and Team sub-theme displays an important point when it comes to any tool usage in someone's work. Most of the work that is being done is done in teams and their way of working has a large effect on how effective any activity, not just test maintenance activities, can be. The Company Culture sub-theme from the *Ways to Assure Quality* theme (see Section 5.2.2) showcases some importance of working together with good methods and processes. To insert any tool usage, especially if it is unfamiliar, requires some consideration to make sure that nothing of the current or future ways of working are interrupted, or that the change is too jarring, so that the work done is not worsened by the introduction of a tool. A tool that fits within current practices can instead enhance them, both in quality and speed.

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

> We create the method, we have a test on it, we always instantiate the method." - P2

Automated Execution

Finally, the Automated Execution sub-theme expresses the desire to tackle problems with executing tests and methods using tools. These can be problems such as difficulties in avoiding errors or additional time needing to be spent on the task. The desire for this kind of tool comes from a desire to ease the workload and be able to put more time and effort into other activities, for example, the different problems brought up in the *Issues Related to Test Maintenance* theme (see Section 5.2.3).

> "That would actually be the ideal system that would look at the interfaces of your different methods that you write and then give you a better idea of whether you are actually covering all of the different scenarios in your tests or even run automated tests on that." - P1

This quote exemplifies the desire to replace previously manual testing activities with automation and also serves as an example of how tools can help test maintenance activities.

5.2.5 Attitudes Towards Generative AI

As part of the interview, the interviewees were asked to give their opinion on LLMs and generative AI, mainly when it comes to its applicability for test maintenance but also their attitudes towards it in general. These opinions consisted of both their experience with LLMs and generative AI as well as their general thoughts based on what they have heard or encountered in regard to the topic. This theme thus encapsulates the attitudes towards the tools from the interviewees. These attitudes are an important way to gauge the feasibility and desirability of implementing a tool like this to help with test maintenance. An overview of the sub-themes can be found in Table 5.10, a more detailed description follows below.

Table 5.10: Overview of Attitudes Towards Generative AI's sub-themes.

Sub-theme	Description
Positive	Likes the idea of getting help from LLMs.
Unfamiliar	Have not used LLMs, either for lack of opportunity or interest.
Sceptical to LLM Abilities	Do not think LLMs can efficiently help with test maintenance.

Unfamiliar

A few of the interviewees were Unfamiliar with LLMs and generative AI in general. This constitutes the first sub-theme of the *Attitudes Towards Generative AI* theme.

> "I have never used it [...]" - P5

> "We haven't used it in our daily job [...]" - P7

This mainly stemmed from the fact that for the majority of the time they are not allowed to use such tools for their work. This is caused by security risks in providing data to third-party software. Although Ericsson has recently released their version of an LLM that is allowed, it is still very new and not a lot of employees have heard of it. Beyond that, there is a general lack of interest in a few of the interviewees that prevented them from trying in their free time.

Skeptical to LLM Abilities

A common trait among the interviewees who had heard about, or used, LLMs or generative AI were that they were Sceptical to LLM Abilities, which is the second sub-theme of this main theme. Either because they had heard rumours or because they had tried them themselves they were of the opinion that the tools could not help them with what they were looking for, or that the tools would not do a good enough job of helping them.

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

On top of that there have been recommendations to not use Ericsson's local version of an LLM:

> "[...] in Scrum master meeting they recommended not to use that because they can't track a lot of information [...]" - P5

Positive

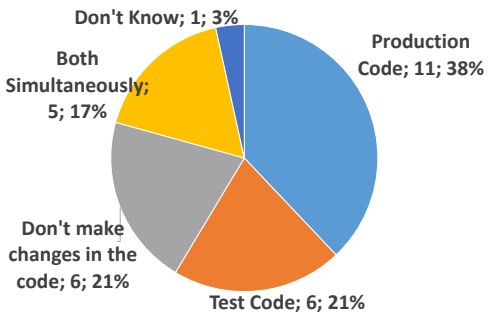
The last sub-theme focuses on the Positive attitude that interviewees had toward the potential of LLMs and generative AI. This attitude was clearly expressed by five out of eight of the interviewees, with at least one interviewee per session expressing this opinion. They believed that these tools could most probably help out in various ways in their work either at the moment or in the future.

> "I'm quite keen on the idea of generative AI taking over those types of tasks [...]" - P1

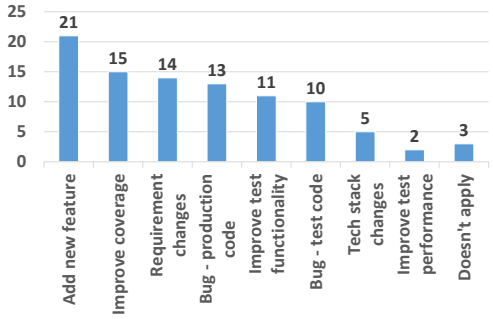
5.3 Analysis of Survey Responses

This section will present the results from the test maintenance survey. The results purely related to test maintenance can be seen in Figure 5.3, and the results related to generative AI can be seen in Figure 5.4. Because of the low response rate (see Section 4.5.3), these results should only be seen as an indication of the current practices and opinions from Ericsson. The demographics of the respondents can be seen in Figure 4.3. All questions and corresponding answer alternatives can be seen in Appendix E.

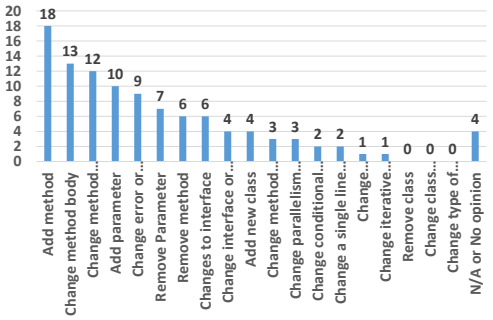
When looking at the results from the questions about test maintenance there are a few things worth noting. Firstly, though it is slightly more common to start making changes in production code compared to test code, there is no clear division (Figure 5.3a). This limits the use of the factors presented in Section 5.1 as triggers.



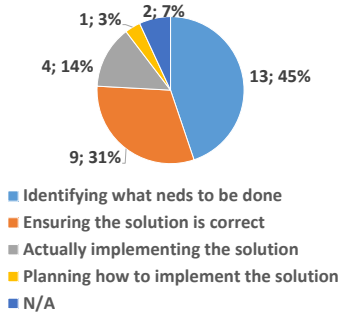
(a) When making changes in the code, do you most often update production code or test code first? Labels show response; frequency; and percentage.



(b) What reasons have you encountered for making changes in the test suite or in an individual test case? Axes show frequency and responses.



(c) Which specific types of production code changes most commonly necessitate adjustments in the associated test code? Axes show frequency and responses.

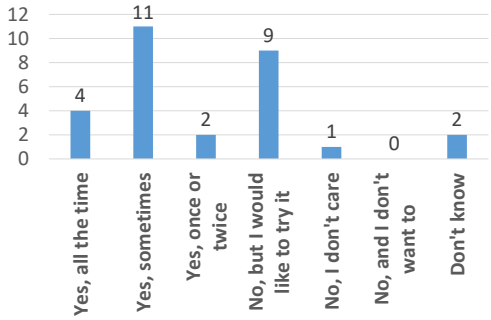


(d) During the test maintenance process which step is most effort-intensive? Labels show frequency; percentage

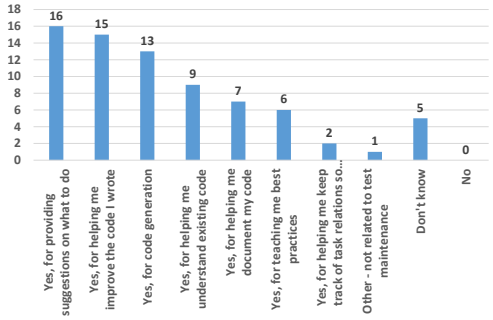
Figure 5.3: Results of survey questions about the test maintenance process. A-D shows the respondents' answers to the respective questions. All questions and corresponding answer alternatives can be seen in Appendix E.

Secondly, the need for test maintenance is triggered both by changes on the production side (such as the addition of a new feature and changes in the requirements), as well as purely test-related reasons (such as increasing the code coverage or improving a test's functionality) (Figure 5.3b). As for test maintenance triggered by changes in production, changes on the method level were more common than changes on the class level (Figure 5.3c). The most common reason was the addition of a new method, followed by changes to the method body. This shows the importance of existing research on test generation techniques, but also the need for test modification techniques. Lastly, when asked about the most effort-intensive task in test maintenance, implementing the solution was deemed much less effort-intensive compared to identifying what needed to be done and checking that the solution was correct (Figure 5.3d). This shows the relevance of using LLMs not only for code generation but also as testing assistants.

As for the results of the LLM questions, the results show that though their use might not yet be widespread, most respondents have tried them at least once, even though they might not use them in their daily work, and generally had a positive view of LLMs (Figure 5.4a). This means they may be more open to trying a test maintenance solution using LLMs. This is in line with Figure 5.4b, which shows that all respondents would like help with generative AI with test maintenance activities. It is worth noting that code generation is only the third largest and most popular category among respondents as to what type of assistance they would like. It is overtaken by suggestions on what to do as well as improving code already written by the respondent. This shows a disparity between the research done (see Section 3.3) and practitioners' wishes.

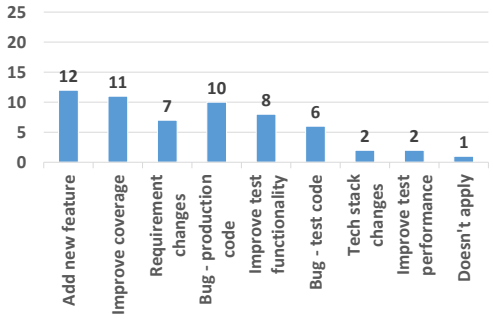


(a) Have you used generative AI or LLMs (Large Language Models)? Axes show frequency and responses.

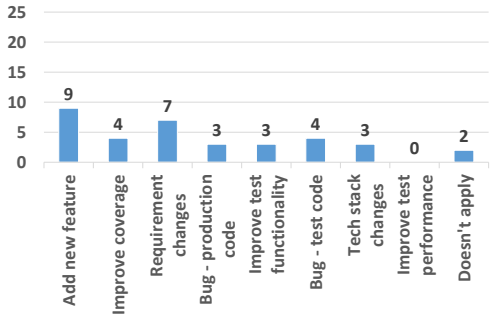


(b) Would you like the help of generative AI/LLMs for test maintenance activities? Axes show frequency and responses.

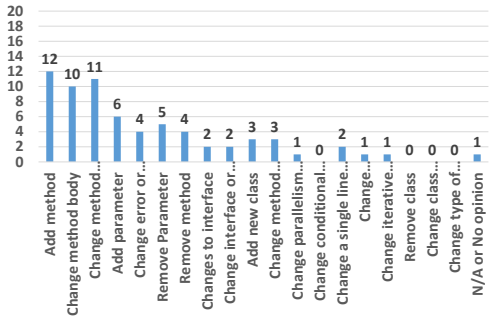
Figure 5.4: Results of survey questions about generative AI. Sub-figure A and B shows the respondents' answers to the respective questions. All questions and corresponding answer alternatives can be seen in Appendix E.



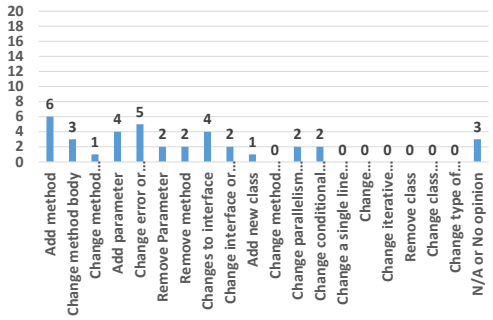
(a) Reasons for making changes in tests for respondents ≤ 5 years experience.



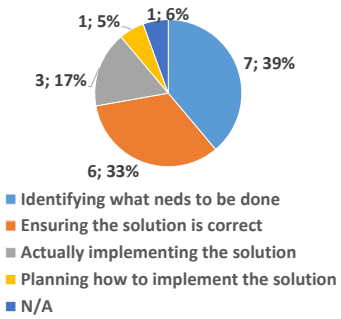
(b) Reasons for making changes in tests for respondents with > 5 years.



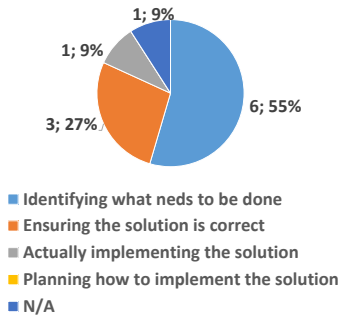
(c) Most common changes in production code that cause test maintenance according to respondents with up to five years of experience.



(d) Most common changes in production code that cause test maintenance according to respondents with more than five years of experience.



(e) Most effort-intensive test maintenance step according to respondents with up to five years of experience.



(f) Most effort-intensive test maintenance step according to respondents with more than five years of experience.

Figure 5.5: Difference in opinion about test maintenance for respondents based on experience. A-F shows the respondents' answers to the respective questions. Axes show frequency and responses, Labels show frequency; and percentage.

Some differences of opinion can be seen between respondents with up to and more than five years of experience. Statistics of questions with interesting differences of opinion are displayed in Figure 5.5. Notable differences include how respondents with more than five years of experience more rarely change tests to improve coverage (See Figure 5.5a and b), and more rarely see the need to change tests because of changes in the method return value for production code (See Figure 5.5c and d). Both groups found *Identifying what needs to be done* as the most effort-intensive test maintenance step, but it held a higher percentage for developers with more than five years of experience, with 55% compared to 39%(See Figure 5.5e and f). It should be noted that these differences of opinion do not necessarily stem from the difference in experience. As described in Section 4.5, respondents with up to and more than five years of testing experience often have different work roles and do different types of testing. The disparity could be because of the different work tasks simply bringing attention to different problems of test maintenance.

5.4 Literature Review of the Use of LLMs for Test Maintenance

This section will present the results of the literature review on how LLMs can be used for test maintenance. This literature review was performed to help answer RQ2, specifically which test maintenance actions LLMs can take and help with (RQ2.2) and what to consider when using LLMs within a corporate setting (RQ2.3). The literature review aimed to explore the research area to gain a better understanding of what to consider when the time came to build our agent, as well as gain inspiration of what use cases could be relevant to test it for. The literature review was not used to dive deep into one particular area, and this sentiment is reflected in the results. The results were also used as a basis for further exploration of possible uses for LLMs and LLM agents within larger use cases, which can be found in the discussion in Section 6.5.2 and 6.5.3. The literature review did not uncover any applications of LLMs specifically for test maintenance. Because of this, the remainder of this section will first present LLM uses and actions that are especially relevant to test maintenance, though they may belong to the broader field of software testing, before presenting some actions that are relevant to test maintenance but also have broader uses within software engineering. Lastly, it will present some considerations for choosing which LLM to use within an agent.

5.4.1 Test Maintenance Actions

Figure 5.6 shows an overview of the actions that an LLM can take that relate to test maintenance according to the findings from the literature review of LLMs. There are four main categories of actions that can be taken, with some overlap. The first category (blue) focuses on the generation of code using LLMs. The second category (green) focuses on an LLM's ability to identify and localise certain aspects of the code, e.g. faults. The third category (red) focuses on an LLM acting like a partner or coworker able to help and assist in various ways, e.g. providing explanations

or advice. The final category (purple) focuses on interacting with and modifying existing code, both production and test code, to ensure better quality of the code base.

Figure 5.6: Overview of actions an LLM can take that relate to test maintenance.

Within the topic of LLM-assisted software testing, one of the more researched areas is test case generation. Test case generation could help with test maintenance as changes to the production code may lead to new paths through the code that are not covered by current tests, nor have tests suitable to be modified to cover them. Though test case generation is not the focus of this thesis it is still a prominent enough area to consider. LLMs have been used to generate test cases and oracles from both code and natural language. These generation techniques generally perform well for simpler test cases [132, 5, 44, 100, 11, 3, 83, 98, 43]. Because of this, efforts have also been made to generate more diverse test cases as well as failure reproducing test cases [43, 83, 99]. It has further been established that the quality of the generated test cases can be improved by providing the LLM with the resulting compilation errors and by dividing the tasks into smaller subtasks to include planning and understanding [3]. Other than generating complete test cases and or-

acles, LLMs have further been used to generate test input to already existing test cases, e.g. to help with GUI testing, testing deep-learning libraries, and to help with fuzzing [44, 11, 43, 133, 134]. GUI testing requires text inputs with semantic relevance to the task at hand which is unfeasible for regular random text generators and thus LLMs are helpful in automatically generating semantically appropriate text as test input for GUI testing [135, 136]. Fuzzing is a testing technique focused on generating unexpected, random, or invalid data as input to see how the program behaves [137]. LLMs are particularly helpful with fuzzing when it comes to more complex domains such as deep learning libraries as the input required is more complex and other generators find it hard to generate input programs. LLMs on the other hand can fully automate this process [133, 134].

An important part of test maintenance is understanding which test cases test which parts of the production code, i.e. understanding the traceability between the artefacts, to know where maintenance is necessary. LLMs have been used to recover traceability links between natural language artefacts and code artefacts [138, 139]. This capacity may also apply between code artefacts, and would if so provide valuable help when performing test maintenance.

It has also been noted that the conversational nature of LLMs brings benefits to the testing process. Feldt et al. note how this allows LLMs to act as an intelligent testing partner by for example giving the user advice, examples, explanations, and checklists. The user can in turn ask follow-up questions and for clarifications [15]. It was similarly noted how the conversational nature of LLMs allows them to take the role of a pair programming partner [10, 140]. Performing test maintenance is very much a process where the developer needs a deep understanding of the relevant code as well as best practices. The kind of back-and-forth help LLMs provide is therefore relevant for test maintenance as well as other software engineering tasks.

As mentioned, an important part of effective test maintenance is understanding the code and how changes impact different parts of it. LLMs have been used to alleviate this problem both by explaining code and by summarising code, allowing developers to quicker familiarise themselves with it [141, 142, 43, 44].

LLMs have also been used to help with code review [143, 44, 144, 43]. The code review activity is by no means unique to test maintenance as it can be applied to all code changes. Nonetheless, as test maintenance includes modifying code, we have chosen to mention it here as well. The purpose of code review is manifold but LLMs have been found helpful specifically in providing suggestions for code improvements and optimisations, both by providing review comments as well as code snippets.

The following paragraphs will mention more general software engineering activities where the test maintenance applications are less clear. These have been included as inspiration of what LLMs are capable of doing, and how diverse their applications are. In many cases, they also talk about ways to improve code quality, and we therefore see a general applicability in them, where they may have some relevance whenever code is modified.

LLMs have many use cases in software testing beyond generating test cases. They have been used for generating mutants for mutation testing [44, 83, 43, 91], defining test cases [145], and predicting both test output and flakyness [83, 43]. Further, they have been used to evaluate test adequacy and improve code coverage by identifying areas lacking coverage [83, 43]. LLMs have also been noted for their ability to adapt test scripts to different devices [146], extract variables for metamorphic testing [147], and minimise test suites [83, 132].

Faults are a known problem within software engineering. LLMs have been shown to have the capacity to localise faults [132, 11, 43, 141]. They have further been shown to have the more general ability to identify problems and defects within code [43, 143, 141, 145, 44, 83, 148, 95, 6]. Beyond localisation, LLMs have been used to fix the discovered faults as they have been used for automated program repair [43, 89, 90]. Another way LLMs have been used to help alleviate faults is through bug triage, i.e. prioritizing faults and assigning them to the right developer [43].

Lastly, the most general use of LLMs within software engineering is of course code generation, which has seen plenty of applications [141, 145, 44, 90, 10, 102, 83, 43, 96, 143, 95, 6]. Outside of code, LLMs have also been used to generate documentation [141, 83, 43, 6], comments [141, 94], as well as method and test names [43].

5.4.2 Considerations for LLMs in Corporate Environments

This section will present the relevant findings about what technical aspects to consider when using LLMs in a corporate environment. It will cover the different aspects to consider when picking an LLM, including size, purpose, and performance. This is not meant to be a thorough guide for every possibility, but instead a way to show some of the facts that can guide the choice of LLM. An overview of some considerations can be found in Figure 5.7.

The first thing worth noting is that there are a lot of LLMs and that each of these LLMs has varying performance across different tasks. Because of this, the best model for software engineering may vary depending on what software engineering task the user requires the model for. For example, Zheng et al. note that the Code-LLaMA series performs the best for code generation tasks, while GPT-4 and GPT-3.5 turbo perform better for test case generation tasks [81]. They also note that, while there are well-known benchmarks for code generation tasks (e.g. HumanEval [149]), other software engineering tasks have fewer or less recognised benchmarks or none at all. This makes it harder to compare models. Wang et al. similarly note the lack of benchmark datasets and also draw attention to the potential data leakage issues, where the model may have seen the datasets during training, making the evaluations less trustworthy. This is especially a risk for the widely-used benchmarks, and for benchmarks not designed specifically for LLMs [11].

As for the HumanEval benchmark, Liu et al. describe current code benchmarks (including HumanEval) as often failing to cover all possible scenarios, as they are often manually crafted, which is a laborious process. They note two problems in

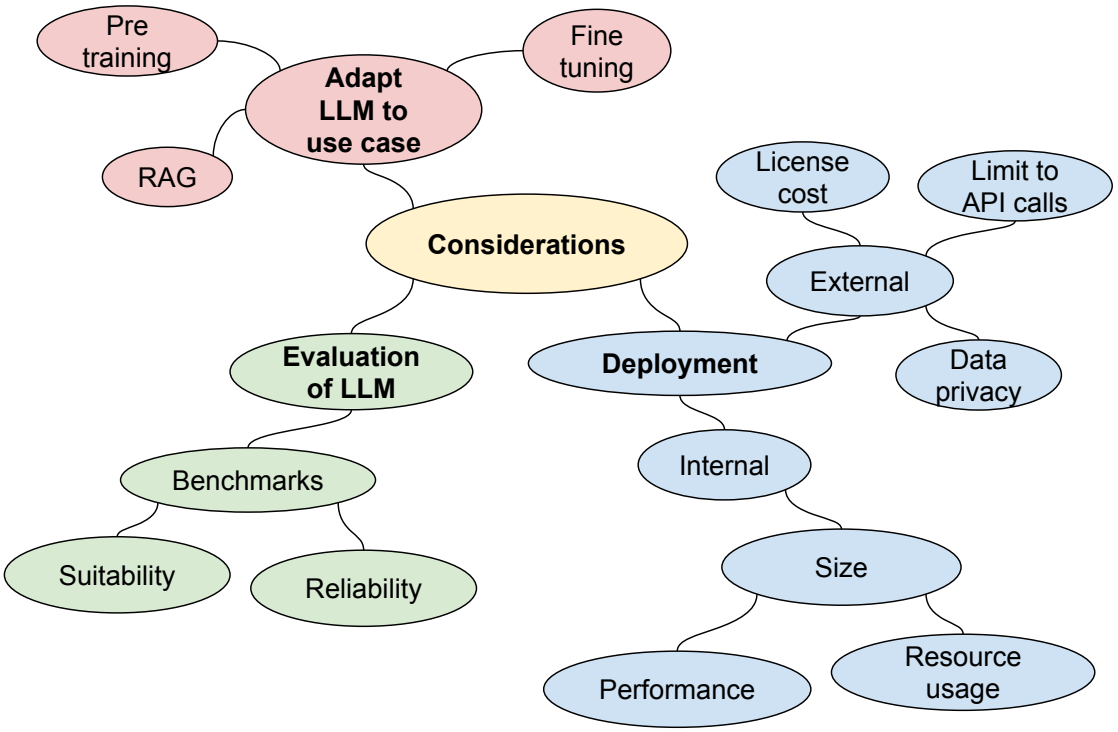


Figure 5.7: Considerations for LLMs in Corporate Environments.

particular—the solutions to the code problems are insufficiently tested with too few and too simple tests (they give an example of a faulty solution that still passes), and the problem descriptions for the code are too vague, leading capable LLMs to possibly be judged as incapable [150]. They present their HumanEval+ benchmark as a possible solution. There are other benchmarks and rankings, for example, Zheng and Chiang et al. present LMSYS Chatbot Arena where chat-LLMs are evaluated based on human preference [151]. However, it should be noted that since this is a very active field, these rankings can be volatile and move around as new LLMs are introduced.

This is to note that it is currently hard to accurately compare LLMs, even when you have a specific task in mind. The following paragraphs will discuss how to make more general assessments.

In general, the larger the model, the better it performs. Larger models also demonstrate emergent abilities, such as in-context learning [47] and step-by-step reasoning [152], that are not necessarily found in smaller models [153]. Zheng et al describe the number of parameters the model has as having a significant impact on the model's performance within the same model architecture [81]. That said, they also note that a fine-tuned model can outperform its base model on the task it was fine-tuned for. In a similar vein, they also describe code-LLMs as outperforming general LLMs for code generation tasks, when their number of parameters are of comparable size. In other words, there are some general patterns to draw upon for model performance.

Mandvikar presents an analysis of factors for organisations to consider when selecting an LLM [76]. These factors cover the performance of the model (such as model size, training time, and the quality of the pre-training data) but also less technical aspects, such as the license cost and if there is a limit to the number of API calls. These are important aspects to consider if the organisation is not planning on deploying the model themselves.

There are advantages to open source and smaller models as well. Wang et al. observe that in practice software organisations generally prefer open-source models over commercial LLMs due to data privacy concerns [11]. Deploying the model on their hardware gives them more control. Many open-source models are smaller than their commercial counterparts. This means they use less computation power and have a smaller energy consumption [11]. Wang et al. also observe that even fine-tuned medium-sized models have a hard time performing as well as the larger models and that collecting a high-quality dataset with organisational data is a great effort, both concerning time and labour.

5.5 Evaluation of Prototypes

A comparison between the prototypes and total scores is presented in Table 5.11. The results of the evaluation of each of the four prototypes for each commit can be found in Appendix F. The prototype with the best general performance was using a planning agent with natural language summaries of test cases. We count it as the prototype with the best performance as it has either the highest or second-highest score in all categories. Context and comparisons for the results are provided in the discussion, Chapter 6.

Table 5.11: Comparison of results of prototypes. Bold is used for the highest score in each category.

	Average Recall	Average Precision	Average F1	Aggregate Recall	Aggregate Precision	Aggregate F1
Planning with summary	0.5618	0.3461	0.4283	**0.3108**	**0.2494**	**0.2767**
Planning without summary	0.5558	**0.4799**	**0.5151**	0.1723	0.2295	0.1968
Chain with summary	**0.5871**	0.1992	0.2974	0.2831	0.1716	0.2137
Chain without summary	0.5323	0.1638	0.2505	0.2123	0.1068	0.1421

Some general trends could be seen in the results: Firstly, the prototypes using natural language summaries of test cases generally performed better than their counterpart that used the code version of the test cases (the exception being the average precision and F1 score for the planning agent). This could be because they could more easily extract the relevant information from this format, leading to better suggestions for test cases.

Secondly, all prototypes performed better when there were test cases to be found, compared to when it would be correct to not recommend any test maintenance. This could mean that the test maintenance triggers are either too general or that there should be additional safeguards in place when evaluating if test maintenance is necessary.

Thirdly, the prototypes using planning agents would often stop due to time or iteration limits. This result was counted as zero test cases needing maintenance. As there were many data points in the dataset where zero test cases had been changed, these prototypes might have benefited from this way of calculating the result. This can be seen in how the average precision scores are a lot higher for the prototypes with planning agents, compared to the LLM chains. These results are technically correct according to our evaluation procedure, as the correct number of test cases have been suggested, but are not useful to a user in practice.

The planning agent with test case summaries benefits from both the first- and third-mentioned trend, both of which contribute to making it the best-performing prototype. These results could be viewed as unfair and undeserved. On the other hand, it still has the highest aggregated recall score, which should not be affected by the agent stopping due to time or iteration limits. This means it still has performed quite well, compared to the others.

Table 5.12 compares the evaluation results of the prototypes with evaluation results that exclude the outputs that hit the iteration limit. By excluding the results that hit the iteration limit a few patterns can be seen. The average recall is decreased for the prototypes that include summaries while it is increased for the prototypes that do not include summaries. The average precision is decreased for all prototypes. The average precision also has a higher magnitude of change compared to the average recall and thus the average F1 score is decreased for all prototypes. The aggregate recall is increased or identical for all prototypes, while the aggregate precision is identical for all prototypes. Finally, the aggregate F1 score is increased or identical for all prototypes due to the change in the aggregate recall.

These changes in the results can stem from three reasons. Firstly, as previously mentioned the results that hit the iteration limit could fit with the results that should have zero test cases. That could explain some of the decreases in the average results, particularly the average precision as that would remove some of the perfect scores when no wrong test cases were found due to no test cases being found. Secondly, by excluding the outputs that hit the iteration limit there could be fewer production code changes per commit that lead to outputs than there otherwise would be for some

Table 5.12: Comparison of results of prototypes including and excluding results where the agents hit the iteration limit. Bold is used for the highest score in each category.

	Average Recall	Average Precision	Average F1	Aggregate Recall	Aggregate Precision	Aggregate F1
Filtered results - Planning with summary	0.5561	0.1717	0.2624	**0.3156**	**0.2494**	**0.2786**
All results - Planning with summary	**0.5618**	**0.3461**	**0.4283**	0.3108	**0.2494**	0.2767
Filtered results - Planning without summary	**0.6019**	0.3555	0.4470	**0.1860**	**0.2295**	**0.2295**
All results - Planning without summary	0.5558	**0.4799**	**0.5151**	0.1723	**0.2295**	0.1968
Filtered results - Chain with summary	0.5797	0.1849	0.2803	**0.2831**	**0.1716**	**0.2137**
All results - Chain with summary	**0.5871**	**0.1992**	**0.2974**	**0.2831**	**0.1716**	**0.2137**
Filtered results - Chain without summary	**0.5335**	0.1334	0.2135	**0.2156**	**0.1068**	**0.1429**
All results - Chain without summary	0.5323	**0.1638**	**0.2505**	0.2123	**0.1068**	0.1421

commits. This could then skew the results of that commit. Thirdly, some commits are excluded in total as all the production code changes of that commit resulted in the output hitting the iteration limit. This could decrease the total number of test cases to be found, e.g. an entire commit which had some test cases to be found was excluded completely, which meant that the aggregate recall would be increased due to the decrease in the denominator for the calculation. What is not shown in the table is how many commits were excluded from the dataset for each prototype based on the iteration limit. The original dataset had 57 commits, after removing the results that hit the iteration limit the planning with summaries prototype had a total of 45 commits, while the planning without summaries prototype had a total of 46 commits. In comparison, the chain with summaries prototype had a total of 56 commits while the chain without summaries prototype had a total of 55 commits. This shows how the iteration limit mostly affected the architectures that included the planning agent. If the overall new scores are compared, then the best-performing models for each metric stay mostly the same. The sole exception is that the planning without summaries prototype is now the highest-performing prototype in the average recall metric.

6

Discussion

This chapter presents the results of the research questions as well as a discussion about them. Additionally, it presents possible ways forward and threats to validity, as well as a short note on AI ethics with regard to the prototypes.

6.1 RQ1

This section will present and discuss the results of RQ1: **Which factors suggest that a test case needs to be updated or created due to changes in the source code-under-test?** This question is answered based on the results from the literature review of test maintenance factors (see Section 5.1) as well as the interviews and survey about test maintenance (see Sections 5.2 and 5.3). Two types of factors were found: high- and low-level factors. The following subsections will present these, and how they provided an answer to RQ1.

6.1.1 Low-level factors

Several low-level factors were found through the literature review of test maintenance factors (see Section 5.1). These factors were also included in the survey (See Section 5.3) and were confirmed based on the responses therein. A low-level factor is categorised as a change in the production code that will most likely cause a need for a corresponding change in the test code, thereby causing a need for test maintenance. Because of this, they provide relevant answers to RQ1. These factors are thoroughly documented in Section 5.1, as well as Figure 5.1 and Tables 5.1-5.4, and will therefore not be repeated here.

These factors were used in the prototype built for RQ3, which generally suffered from being a bit "trigger-happy", i.e., it would say test maintenance was needed more often than it actually was. This could indicate that these factors are a bit too wide and general, and capture more than they should. An example of this is the trigger *Add functionality*, which is very broad and covers a lot of potential code changes. Because it is so broad, it could be that it is triggered even for changes that do not necessitate test maintenance. Alternatively, it could mean that these factors should be combined with some other additional safeguards, that could help give a deeper understanding of the code change itself to help see if any test cases need to

be updated. At the moment, they are more suitable to be used as an indication that test maintenance *may* be necessary instead of strictly necessary.

6.1.2 High-level factors

From the interviews and the survey with Ericsson employees, several high-level factors were found. What is meant by high-level factors is factors that encompass an entire use-case, or an entire process from start to finish. The goal of this process is not necessarily test maintenance, but the process will likely cause a need for test maintenance since the process will likely make changes in the production code. For example, if adding a new feature to the product, new tests will need to be created and existing tests might need to be adjusted. They are therefore included as an answer to RQ1, even though they are not strictly production code changes. The factors are presented in Table 6.1.

Table 6.1: Description of high-level factors that may cause test maintenance.

Factor	**Description**
Adding new feature	Expanding on an existing product by adding new functionality, leading to the creation of new tests and the adjustment of existing tests.
Change in requirements	Existing feature requirements have been modified, leading to a need to change both production and test code.
Discovery of a fault/ failure report	The discovery of a fault in the code base, either by a developer or documented in a failure report, cause a need to add or modify tests to ensure the fault has been repaired and to ensure it will be caught if it appears again.
Increase coverage	The test suite is modified and added to from the intention to increase the code coverage, which is used as an important metric.
Refactor/improve code	The code is refactored or enhanced with the intention to preserve the current functionality. Can be done both to test code and production code, in which case it may still cause test maintenance.
Tech stack changes	Changes in any of the technologies used in the development process may cause test maintenance.
Evaluation of tests	The systematic evaluation of a test suite may lead to the decision to modify it.

These factors were partially used as a basis for RQ2, and were not integrated into the prototypes. They were however used to speculate about larger more all-encompassing use cases for LLM agents in Future Work (Section 6.5.2)

RQ1: Several changes to the production code were found to cause a need for test maintenance. These could be categorised under changes to the code's functionality, changes to a method, changes to a class, and changes to a single line of code, e.g. type changes. The complete list is presented in Figure 5.1. Additional high-level development or testing activities may lead to the need for maintenance as well (see Table 6.1).

6.2 RQ2

This section will present and discuss the results from RQ2: **What applications could current LLMs or LLM agents have within the area of test maintenance?** These results further build upon the results from RQ1, and are based upon the results from the survey (Section 5.3) and interviews (Section 5.2), as well as the literature reviews about LLM capabilities (Section 5.4 respectively). The answer to this research question will be presented by presenting the answers to RQ2.1-RQ2.3, which is done in the following subsections.

6.2.1 RQ2.1

This subsection will present the answer to RQ2.1: **Which factors from RQ1 can act as triggers for test maintenance in an LLM or LLM agent?** The factors presented in Section 6.1 belong to two groups: low- and high-level factors. The low-level factors are direct changes to the production code that are likely to lead to a need for test maintenance. It was reasoned that these factors could act as triggers for an LLM agent. This was tested through the thesis' setup with LLM agents, and it was confirmed that presented with the modified production code and equipped with a search tool for the code repository, the agent could use the factors as triggers to predict if and where test maintenance was necessary (see Section 6.3). As previously mentioned, though the factors can be used to predict the need for maintenance, they are a bit too easily triggered. This suggests that the triggers might need to be used together with some other information to get a deeper understanding of the code change and to be better able to judge if test maintenance is necessary.

As for the high-level factors, it was reasoned that these factors could act as triggers for agents designed to reason about higher-level project artefacts or development and testing activities. For example, an agent could be given a change in the requirements as input, and then reason and plan how this would affect the relevant artefacts, before modifying them. From RQ1, we know that a change in the requirements is likely to lead to test maintenance, and therefore the high-level factor *Change in requirements* would—in an agent like this one—be a trigger that alerted it that test maintenance will probably be necessary as a result of the rest of the changes it would have to make. This reasoning was not tested during this thesis but is further explored in Section 6.5.2.

RQ2.1: The low-level factors were confirmed to work as triggers for an LLM agent, while the high-level factors are hypothesized to work, but have not been tested.

6.2.2 RQ2.2

The answer to RQ2.2, **What are potentially viable test maintenance actions that an LLM or LLM agent could take based on these triggers?**, will be presented in this subsection. Based on the literature review provided in Section 5.4 the following actions were identified as viable and applicable to the test maintenance process:

- Generate test cases.
- Understand traceability between artefacts, and help developers understand where maintenance is needed.
- Understand code and explain and summarise code for developers' benefit.
- Act as a testing or pair programming partner. LLMs can provide advice on best practices and examples, and because of their conversational nature, the developer can ask for explanations, clarifications, and follow-up questions.
- Help with general software engineering activities that might appear during the test maintenance process, such as code review, improving code quality and quality assurance, evaluating test cases and code coverage, and generating documentation and comments.

The viability of the actions of summarising code and understanding traceability between tests and methods were validated through the thesis' setup with LLM agents. The agent further confirmed that LLMs can also use test maintenance triggers to evaluate if test maintenance is necessary as well as compare new and old code and summarise the changes. Due to time limitations and a desire to focus on the agent's area of use, this study did not confirm how well LLM agents could generate test cases, act as a testing or pair programming partner, or help with other software engineering activities.

A large amount of the current research is focused on code generation, however, throughout this study, it has become clear to us that there are many further applications of LLMs beyond code generation. LLMs are flexible and versatile in nature, and to put too large a focus on code generation would be doing them a disservice. Though we have listed the test maintenance actions previously identified by literature, there are many other creative potential applications. We believe inspiration should be taken from how things are done now, but that should not stop future research from exploring new directions and re-visualising how test maintenance is approached.

RQ2.2: Four types of activities were identified: generation of code and test cases, helping developers understand code and artefacts, acting as a pair-programming partner, and helping with related SE activities (e.g., code review or generating documentation).

6.2.3 RQ2.3

This section presents the results of RQ2.3: **Based on the present-day landscape, what are some considerations for building an LLM agent for test maintenance within a corporate setting?**

From the LLM literature review (see Section 5.4) the following conclusions could be drawn:

- Except for code generation, there are few available and established benchmarks for judging LLMs' performance for software tasks. The available benchmarks are also not without issues. There exist few made specifically for LLMs, and the ones that exist may suffer from incompleteness and data leakage issues. They can, therefore, be used as a general indication of the LLMs' capabilities, but should not be taken as absolute truth.

- In general, the larger the size of the LLM (generally measured in terms of the number of parameters), the better it will perform out-of-the-box.

- For control and data privacy, companies may prefer to use open-source LLMs that they can deploy themselves. There are few large open-source LLMs, and they require significant computation and electrical power. Smaller open-source LLM models evade these problems but may instead have a hard time achieving comparable results as the large LLM models even when they have been fine-tuned.

- Collecting company data for fine-tuning may require significant effort. Because of this, using a RAG tool may be an appealing option. Since chunking data is a much faster task, RAG tools can be easily updated to hold current information that can help an LLM make informed decisions, while both fine-tuning and updating a model require significantly more effort.

From the interviews and the survey (see Section 5.2 and 5.3), the following considerations were discovered:

- Though the level of experience with LLMs varies among Ericsson employees, the attitude towards them is generally positive. That said, there is some scepticism about if they are currently good enough to be worth using. Because of this, expectations should be set carefully.

- One of the main wishes that Ericsson employees had for support from generative AI was generating and improving code. However, their test maintenance

problems were much more varied, as are LLM capabilities. There are many directions worth exploring with an agent.

- Many of the test maintenance complications are rooted in a lack of knowledge, meaning the developer will first have to find this knowledge before being able to solve the problem. For example, they may need to familiarise themselves with code, track down the origin of an error message, or evaluate how code coverage is affected by changes. Because of this, one of the most valuable things an agent can provide is not only knowledge about Ericsson's artefacts but also insight about them.

RQ2.3: Companies should consider which size of LLM to use, look for areas of use beyond the obvious use of code generation, and consider how to deploy it, e.g. whether to use a closed or open-source LLM.

6.3 RQ3

RQ3 asked: **What is the precision, recall, and F1 score of our setup with LLM agents in predicting if and where test maintenance is necessary using the factors found in RQ1?** Four prototypes were tested, the results which can be seen in Table 5.11. The highest aggregated scores were produced by the planning agent with test case summaries (recall = 0.3108, precision = 0.2494, F1 = 0.2767). As for the average scores, the highest recall was produced by the LLM chain using test case summaries (0.5871), while the planning agent not using summaries had the highest precision (0.4799) and F1 score (0.5151).

Looking only at the final results, the prototypes perform worse than comparable state-of-the-art technology. DRIFT uses a machine learning-based approach (classification with gradient boosting) to identify outdated test cases and archives an average accuracy and F1 score of 0.816 and 0.806 for within-project scenarios, and 0.689 and 0.695 for cross-project scenarios [71]. In comparison, the highest average F1 score achieved by one of our prototypes was 0.5151, which is considerably worse. It should be noted that DRIFT was tested with a different dataset and using other data preprocessing, which means the scores provide context but are not directly comparable. There are a few possible reasons for the differences in the results. First, the evaluation dataset contained many data points where there were 0 correct test cases to be found. In general, the prototypes performed worse on these compared to the data points where there were correct test cases to be found. The high number of data points with 0 correct test cases to be found may have skewed the results to the worse.

Second, the prototypes used a Mistral 7B model, which is a comparatively small model, compared to the state-of-the-art LLMs. For complex tasks, such as RAG or building agents, the Mistral team recommends using their Mistral Large model [154]. To give a comparison of their capabilities, their scores on the Massive Multitask Language Understanding (MMLU) benchmark are 62.5 for Mistral 7B and 81.2 for

Mistral Large. To put it into perspective, ChatDev, which uses a multi-agent setup to generate software programs, made use of GPT-3.5 Turbo [13]. During this thesis, Ericsson had a number of models of similar size as Mistral 7B deployed. Mistral was used since the prototypes achieved the best performance using it. It is possible that the performance could have been improved by using a larger model, but that was not available during the thesis.

Something that might skew the results is that the prototypes that used a planning agent would often not arrive at an answer at all, and would instead stop due to the set time and iteration limit. We counted that result as zero test cases needing to be updated, as the output contained zero test cases. Because of the high number of data points where there were no correct test cases to find, the prototypes using a planning agent might have inadvertently benefitted from this, as there was a high chance this unhelpful answer would be interpreted as correct. To test this the evaluation calculations for the prototypes were redone while excluding results where the iteration or time limit was hit. This resulted in mostly decreasing average scores while increasing or retaining the aggregate scorings. This shows how the iteration limits inadvertently benefitted the average scorings of the prototypes but also acted detrimentally towards the aggregate scorings. Despite these changes in results the comparisons between the models still hold true with the results that hit the iteration limits excluded with the sole exception of the average recall, where the planning agent without summaries then outperformed the other models instead of the chain with summaries being the best performing model. Worth noting is how the prototypes with the planning agents had roughly 10 less commits for these results and therefore the improved average recall could be affected by the decrease in commits in comparison to the prototypes with the chain setups.

In general, though the prototypes find many test cases, their recall and precision are not great. Manual inspection of the prototypes' false positives reveals that some of them list test cases that are relevant (e.g. the method containing the code change is called in the test case) even though they do not corroborate with the ground truth. These might be of some worth to the user even if they are currently classified as incorrect answers. This means the prototypes may have a higher usefulness to a potential user than implied from the F1 scores. It also further suggests that their performance could be enhanced by incorporating some additional context beyond traceability, as it seems like the prototypes can establish traceability, but are missing some information needed to establish if a test case needs to be updated.

The results also presented both average and aggregated scores. These should both be taken into account when reviewing the prototypes' performance. The average F1 score is comparatively high as the data points are not weighted. This means the score from a data point (a code change) where there were 90 test cases to be found is valued the same as a data point with zero test cases to be found. This has driven up the score. The aggregated score, on the other hand, will be lower for two reasons. Firstly, a sort of weighing between data points occurs, as the data points with fewer test cases take up a smaller part of the calculation. Secondly, all the prototype's mistakes add up in a way they do not do for the average score. We think the true

capabilities of the model lie somewhere between the average and aggregated scores.

RQ3: The prototype with the best general performance was the setup using a planning agent with summaries of test cases. It achieved average and aggregated F1 scores of 0.4283 and 0.2767.

6.4 Evaluation of Usefulness of the Prototypes

An informal evaluation of the usefulness of the prototypes was held, where their functionality was demonstrated and the participants got to share their thoughts. Two developers participated. One had participated in the earlier interviews, and the other was one of the supervisors for the thesis. As such, they were both familiar with the premise of the thesis.

The general impression was positive, and though it was not described as solving the test maintenance problem, it was described as something that would be very nice to have. The developers saw two main benefits. Firstly, the prototype runs much faster than running the tests, which means you can become aware of problems faster. The prototype currently provides a similar service to running the tests, in that a failing test can be an indication it needs to be updated. That said, it should be mentioned that a test that needs to be updated may still pass, meaning the prototype is more reliable in that aspect. The tests are normally run as a part of a CI/CD pipeline, which means it can take a while to get the results. In comparison, using the different prototypes takes between 30 seconds and 2 minutes.

Secondly, it was expressed that knowing that other people used the prototype would give them more confidence in their code reviews. Knowing that the other person had during their code review consulted the prototype, and thereby been given something similar to a guide of which test scenarios to review, could improve other developers' confidence in the code review, as they would know the reviewer had had help in finding which code to consider when doing the review.

Some wishes for improvements were described as well. Currently, the prototypes output the results of each step on the way to the final result, before outputting which test cases need to be updated. This was implemented so that the user may understand how the prototypes arrived at their output, a form of transparency and explainability. The developers were of the opinion that this output was too long and verbose, and would prefer a shorter trace coupled with a better explanation during the final output of why a certain test case needs to be changed. It was also mentioned that suggestions on how to change the tests would also be appreciated.

When evaluating which test cases need to be updated, the prototypes currently look at which methods are called in the test. A deeper understanding (e.g. which methods do the methods called in the test use, would a change there affect the test?) was also thought to improve the benefit of the model, as finding and evaluating these relationships can be time-consuming for developers.

Lastly, it was mentioned that the fact that the prototypes currently exist separately from the other development tools and need to be manually triggered makes them inconvenient to use. An improvement would be to provide the service as an IDE integration, where it can easily be triggered by the user. This would also put the prototypes into a visual interface, which would further improve the experience.

Based on this evaluation and the previously presented results, we think the main things that prevent the prototype from being integrated as a useful tool into the current workflow are the low F1 score and the fact the prototype has not been made into a proper development tool. The low F1 score limits the benefit of the prototype's output, and the low fidelity of the prototype makes it less attractive to use.

6.5 Future Work

This section will present what directions future work should take if this thesis is to be further built upon. It will start by discussing ways to improve the prototypes, before moving on to the test maintenance triggers and how they might fit into a larger multi-agent setup.

6.5.1 Possible Improvements of Prototypes

The prototypes could be made more user-friendly by integrating them in the workflow of an end user. This could be done by integrating them in an IDE and having them available as a feature instead of running them separately.

An avenue for improving the performance of the prototypes is to change the LLM model used. By using a model that has a larger parameter size the prototype would be more capable of analysing and inferring information [47]. Further investigation would need to be done to know the extent to which the prototypes would improve by changing to a larger LLM model. It is difficult to know what problems stem from the model size compared to the architecture of the prototypes without testing using a larger model to power the prototype.

The prototypes used in this thesis have undergone no additional pre-training or fine-tuning beyond what was already present in the LLM models. By applying pre-training or fine-tuning to the model its performance and reliability might improve. Previous research has shown that the size of LLM models utilised in this thesis can improve its performance using fine-tuning towards following instructions [106]. The focus of this training should lie in making the model better at understanding instructions and being able to follow the structure of the agent setup. This would for example potentially alleviate the issues that the planning agent prototypes encountered when they hit the iteration limit, which was caused by the agent outputting its results in the wrong format. The Mistral 7B - Instruct model has already been pre-trained to follow instructions, and a further focus on the prompt formatting present in the prototypes might improve the performance even further.

There are a few potential improvements to be made to the RAG setup in the prototypes. For the prototypes, a single embedding model was used for all embeddings. Future work may consider testing multiple embedding models that might be more suitable for specific tasks. A separate evaluation of determining the most suitable embedding model for the job might be prudent as well. Also part of the retrieval tools is the similarity measure used by both the embedding model as well as the vector database where the embeddings are stored. In this work, the default similarity measure between vectors, i.e. embeddings, was used and it might be prudent to investigate the use of different similarity measures to improve the retrieval of relevant embeddings.

Another potential improvement to the RAG setup is to use query expansion [155, 156]. An LLM can look at the user input and reformat it to better suit any RAG tool parameters so that the retriever can find more relevant results. On top of query expansion, re-ranking the most relevant vectors could be a potentially useful method to improve the RAG setup [157, 158]. The re-ranking would work by taking the most similar vectors, transforming them into text chunks, and then comparing the chunks in the text domain either through the use of an LLM or with another similarity metric, e.g. comparing keywords.

Improvements can be made to the way that the prototypes determine if test maintenance is necessary, by looking at the triggers. The triggers can be made more specific to account for connections to changes in test cases from prior data. Other avenues of determining if test maintenance is necessary can be combined with the test maintenance triggers, e.g., comparing the semantic meaning of the changes in the production code and any effects this might have on connected method calls.

6.5.2 Future Uses for High-level Triggers and Agents

From the interviews and survey we identified several good practices and problems that were not strongly connected to test maintenance, but nonetheless deserve attention. The LLM literature review also identified potential and tried areas of use for LLMs and agents within software development but outside of test maintenance. We envision that in future work where the performance of LLM agents has been improved, this knowledge could be used together with the high-level triggers described in RQ1 (Section 6.1). This is presented in Figure 6.1 and 6.2, which shows possible uses of LLMs from when a high-level trigger is first set off until the change has been deployed.

For example: if the changed requirements trigger is set off, a future sophisticated LLM multi-agent setup might be able to identify which artefacts need to be changed. This could be both production and test code, but also other related artefacts such as agile user stories. The LLM multi-agent setup could provide recommendations on best practices and risks and rewards to help a human better approach the problem or could act upon that information itself and make the necessary code modification. It could then evaluate the quality of the modification to improve them further, and of course perform the necessary test maintenance.

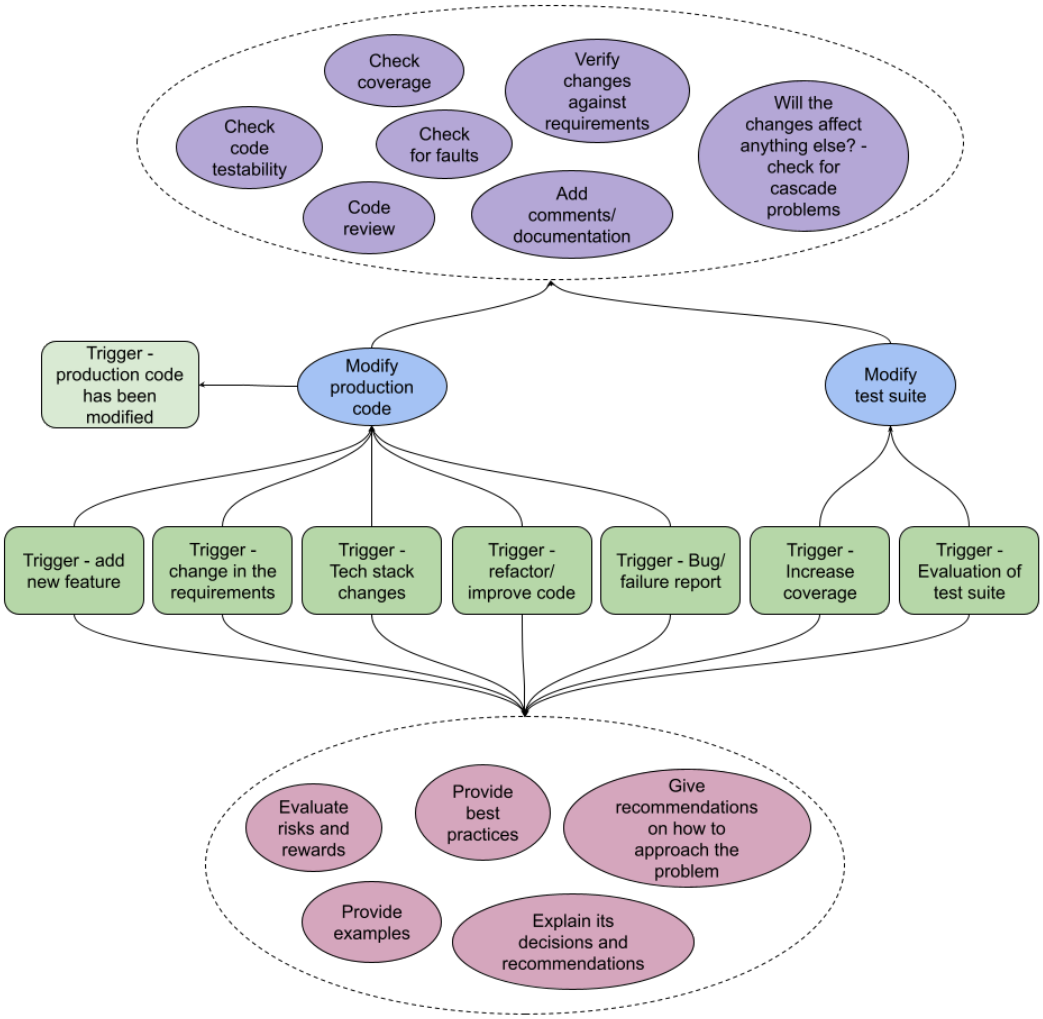


Figure 6.1: Overview of LLM uses for high-level factors. The figure shows potential actions an LLM could take once a high-level trigger has been set off to ensure quality through the modifications to the code. More details for each trigger are presented in 6.2. Green = high-level triggers, light green = low-level triggers, purple = potential LLM actions focused on code quality, pink = general potential LLM actions, blue = potential LLM action that modifies code. Solid lines are used to connect triggers and actions, while dotted lines are used to group actions.

6.5.3 Future Uses for Low-level Triggers and LLMs

This thesis explored how LLMs can be used to predict the need for test maintenance, and if test maintenance is necessary also identify which test cases need maintenance. It did this by using low-level test maintenance triggers, that were activated when production code had been modified. An avenue for future research is to expand upon this use case by making modifications to the test suite. There is ample research on generating new test cases, but less on how to modify existing ones, which can therefore be a promising research area. Another research area not explored in this thesis is how test suite modification during test maintenance affects its quality. There is some research on using LLMs for code review, checking the test suite's

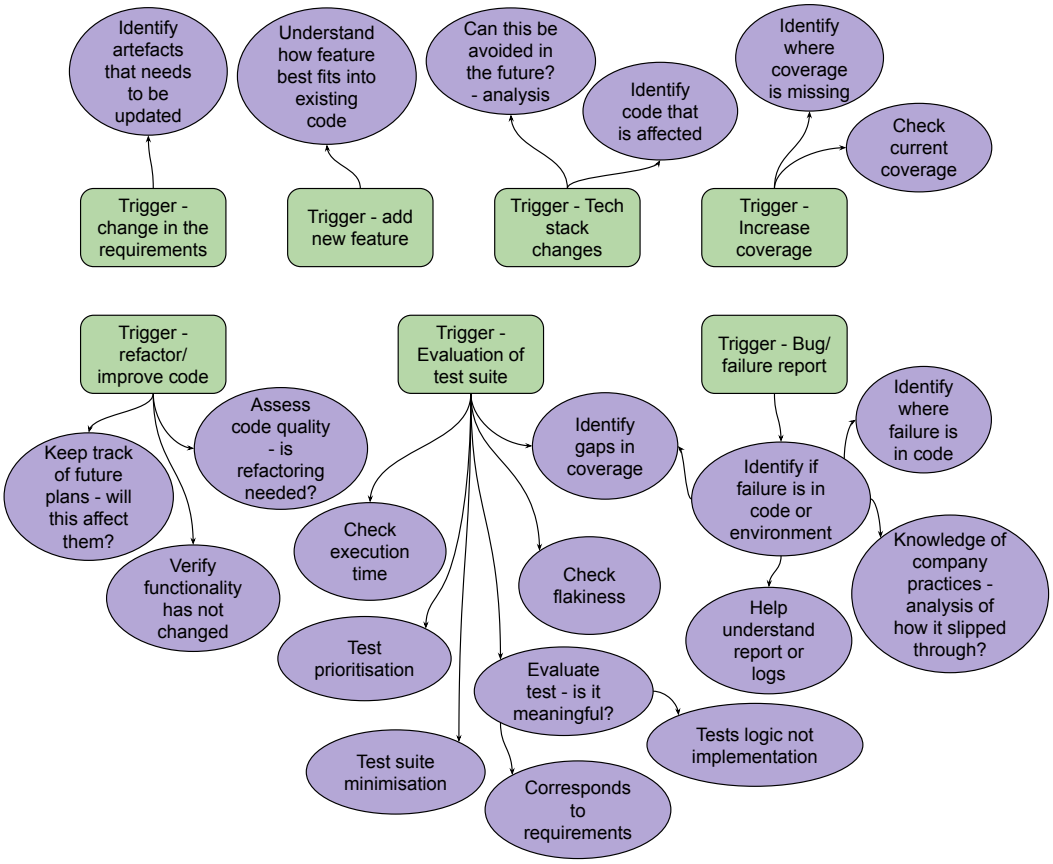


Figure 6.2: Detailed uses of LLMs for high-level factors. The figure shows potential actions for an LLM once a high-level trigger has been set off to ensure quality through the modifications to the code. This is a companion figure to Figure 6.1. Green = high-level triggers, purple = potential LLM actions

coverage, and other ways to improve the test suite's quality, but the research has not been strongly connected to test maintenance. This is another possible research direction. Figure 6.3 shows possible further research directions to expand upon the low-level trigger use case.

6.6 A Note on AI Ethics

Though AI shows great promise in several areas within software development, it is a technology associated with ethical and environmental concerns. This section will cover the relevant ethical considerations for the prototype, as well as the potential environmental impact and how it is affected by the EU's AI act.

6.6.1 Pillars for Ethical AI

IBM present five pillars with properties that should be present for AI or AI systems to be considered trustworthy [159]. We present to which degree the prototypes hold these properties.

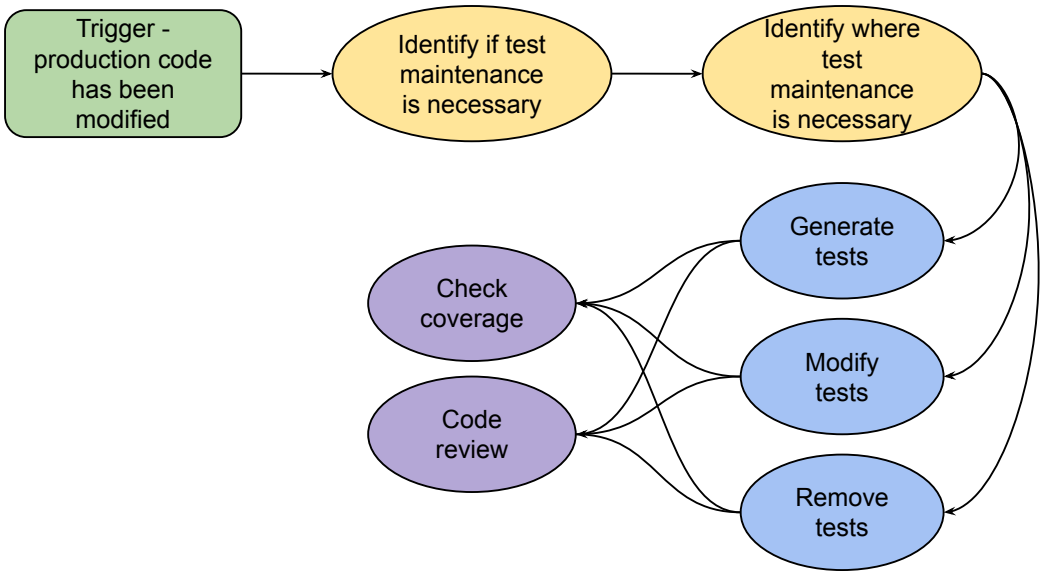


Figure 6.3: Figure shows research areas connected to the low-level triggers, and how the prototype might be expanded. Green = trigger, yellow = explored in this thesis, blue = modifications to test cases, purple = check code quality.

Explainability: The prototypes provide the output of each step and decision, as well as which fragments they pull through RAG. In the LLM chains, this is provided by outputting the results of each step in the chain. The agents on the other hand use a so-called "Agent Scratchpad" that records and outputs the thoughts and decisions that the agents make, as well as which tools they use. This provides a degree of explainability of the final output of the prototypes to the user.

Fairness: The prototypes do not process data related to humans, neither individuals nor groups. Therefore, fairness is not a concern.

Robustness: With the exception of powering the prototypes with a model internally deployed at Ericsson, no effort has been taken to minimize security risks. If the prototypes are further developed, care should be taken to see if they need to be protected from adversarial attacks, data poisoning, and other malicious interference.

Transparency: The details of the prototypes' architecture, as well as what they base their recommendations on, are provided in this thesis. No extra data was used to fine-tune or modify the models. The Mistral 7B model is open-source and has publicly available weights, but the training data is not publicly available [160]. The Nomic Embed model is open source as well. Besides the model itself, they provide access to the training code and data [122]. To summarise, the prototype fulfils some, but not all, of the transparency properties.

Privacy: The prototypes need access to a repository's production and test code,

but do not use or collect any user data or data about humans. By using an internally deployed model, care has been taken to safeguard the information stored in the code repository.

6.6.2 Environmental Impact

Artificial intelligence is a technology that consumes a lot of power, both during training and deployment. Luccioni et al. estimate that the training of BLOOM (a 176-billion parameter LLM) consumed 433 MWh, and the training of GPT-3 (which has 175 billion parameters) 1287 MWh [161]. To put this into perspective, a house in Sweden consumes about 20 MWh per year [162]. The training of GPT-3 could power 64 houses for an entire year.

The models continue to consume power during their deployment. ChatGPT, for example, has an estimated power consumption of 564 MWh per day [163]. In other words, the training phase represents a relatively small amount of a model's total power consumption.

The models used for the prototypes in this thesis are much smaller (around 7 billion parameters), but it should still be noted that AI as a technology is energy-intensive. If the prototypes were to be scaled up, the additional computational resources should be taken into account. In its current state, thought should be given to whether the worth it provides compensates the resources it consumes. In Sweden, more than 60% of electricity comes from renewable sources, but care should still be taken to minimise the footprint of the prototype and use resources efficiently [164].

6.6.3 AI Laws and Regulations

During the course of this thesis, new laws concerning AI and its use were instated, in particular, the Artificial Intelligence Act which was instated by the European Parliament [165]. This act brings forth several regulations on how AI can be utilised and deployed. The results of this thesis are largely not affected by the new laws as the laws focus on general-purpose AI as well as citizens' rights of privacy as well as high-risk applications. The prototypes developed in this thesis are quite specific in what they do and they do not use any biometric categorisation or emotion recognition or any other identifying method focused on individuals, the prototypes are fully focused on analysing code. The prototypes are also not involved in any high-risk system, e.g. critical infrastructure or essential public service. It should however be noted that Ericsson works with 5G and telecommunications systems, which are important public infrastructure. If the prototypes were ever properly added to their workflow, care should be taken to evaluate how they affect the quality of the final product. When it comes to the privacy of data, the prototypes are run locally on either personal machines or on a company-owned cluster to make sure that no data will be available externally from the use case. The process of running the prototypes and receiving the results are all a part of an internal process within the working environment and on its own will not lead to the availability of the data to anyone beyond those who have already accessed it.

6.7 Threats to Validity

This section will present the threats to the validity of this thesis. It will first present the external validity, which is to what extent the results of this thesis can be generalised beyond the scope that the thesis has worked within. Next, the internal validity is presented, which concerns the production of results and how external factors might have impacted it. Thereafter, the construct validity will be discussed, comparing the results to how they were produced and seeing if the intended goals were represented by the methods to reach them. Finally, some writing tools used for the thesis report are brought up.

6.7.1 External Validity

The data used for this thesis project is mainly provided by Ericsson. Although efforts were taken to utilise data from various sources within the domain, the domain is limited to one company and their associates and might therefore not be applicable on a broader scale. For the interviews and the survey, individuals from various sections of the organisation were asked to participate to get a more comprehensive and generalisable result. For the evaluation of the prototypes, however, a single repository was used which makes the results less generalisable than if multiple repositories were utilised. There were plans to utilise an additional repository, an open-source one, for the evaluation, but it had to be dropped due to time constraints.

The triggers that were found in this thesis have several sources. The high-level triggers were mainly obtained from interviews and a survey within Ericsson and therefore might not be as applicable within a broader context. The low-level triggers on the other hand are mainly based on previous research and should be more general. Both high- and low-level triggers are focused on test maintenance and their generalisability should be considered within that domain.

Most of the low-level triggers found in the literature were sourced from studies on Java projects. Similarly, the evaluation dataset from Ericsson also consisted of purely Java code. This means that there is very limited generalisability for the results when it comes to other programming languages as no other languages have been tested or explored in any major way.

6.7.2 Internal Validity

As part of the data-collecting stage of this thesis project surveys were utilised. The utilisation of surveys as a data-collecting tool has some inherent limitations that might affect the validity of our results. These limitations include a lack of control over the response rate of our target sampling group as well as difficulties in understanding the reasoning or motivations behind answers to survey questions [113].

Another issue comes from interviews, which were also utilised in the data collection stage. There is a limited sample size which might not be fully representative of the domain we were aiming to cover. There might also be biases involved in answering

questions during the interviews.

To attempt to combat the validity threats from the different data-gathering methods, a mix of the different methods was utilised. On top of surveys and interviews, a comprehensive literature review was performed and the results from all the data-gathering methods were then synthesised. The use of triangulation with multiple methods reduces potential threats to internal validity [108].

The literature reviews were conducted based on the protocols of a systematic literature review. However, the literature reviews were not as rigorous as the protocols demanded and there might be some research that has been missed due to that fact. The selection of research papers was done individually by the thesis authors and the relevancy was determined by the content of the title, abstract, and conclusion. Therefore, some papers might have been discarded even if they had useful information if they did not make it clear in the abstract or conclusion. This could have led to relevant research not being considered and either certain topics were not covered or already researched topics may have been regarded as unexplored. The threats of this were mitigated by utilising several different databases to get an exhaustive list of papers as well as cross-examining the research papers found by the other author.

Another limitation of the project is the limited time constraint on the project itself. This means that the data-gathering stage was constrained to a shorter time frame. The prototype development process was also constrained to a shorter time frame. These shorter periods can affect both the quantity of data being gathered as well as the quality of the final prototype.

As previously mentioned, the evaluation of the prototypes was done on one dataset constructed from a single code repository. Due to the source of the data, there might be limited variations in the code in the dataset. This might lead to the prototypes not being comprehensively tested regarding their performance on different kinds of code changes.

For the evaluation of the prototypes, an assumption was made that co-evolution between test cases and production code held for each commit, i.e. that updates to test cases share a causal link to updates to production code in the same commit. This might not always be the case and there might be other reasons to make changes in the test cases unrelated to the production code changes, e.g. fixing a bug or improving performance. In fact, in a discussion after the evaluation was already done with one of the main contributors to the chosen repository they mentioned how they often update unrelated test cases and production code. This means that the results of the evaluation might not properly reflect the capabilities of the prototypes as the evaluation dataset could be faulty.

The dataset for the evaluation was split into different commits which were themselves divided into individual code changes. The prototypes ran through the individual code changes and returned lists of test cases for each code change. However, the individual code changes did not have their respective list of correct test cases to

look for and instead, the correct test cases were available for each commit instead. This presents a risk to the validity of the test cases in relation to the code changes themselves. To mitigate this all test cases collected from each code change were made into a set of test cases for the corresponding commit. This still might have impacted the results.

6.7.3 Construct Validity

During the interviews and the survey, there was a risk of participants misunderstanding or misinterpreting terminology used in the questions. This risk was mitigated by running a pilot interview to review and go over the understandability of the interview instrument. The wording of some questions was clarified after this process. Additionally, the survey was shown to a few Ericsson developers in advance to make sure that the questions were as clear as possible and to minimise the risk of misinterpretation.

The development of the prototypes was done entirely by the thesis authors and has not undergone code review by any external party. This presents a risk as there might be differences in the implementation in comparison to the intended purpose of the prototypes. A mitigation to this was done by using ready-made functions for LLM agents and RAG tools and retrievers from LangChain [118]. This does not, however, eliminate the risk entirely and the code would benefit from an external review.

In the evaluation of the prototypes, some code from both the production and test parts of the code repository was excluded based on criteria explained in Section 4.9. These exclusions were done based on limitations in the prototypes and the focus of our use case. Irrelevant information that did not fit in our use case was discarded but nevertheless has an impact on the dataset itself as the discarded changes can have causal links with included changes and those links are then not represented by the evaluation dataset. This exclusion of both code changes and some test cases can mean that the connection between the production code changes and the changed test cases could be faulty in the evaluation dataset. A production code change might be connected to a new test case, which would have been excluded, and this would mean that the ground truth for that commit is wrong and will result in an incorrect result when evaluating the prototypes on that commit.

6.7.4 Writing Tools

To help with grammar when writing this thesis, ChatGPT [4] was occasionally used to help with sentences we felt unsure about (e.g., using the prompt “Is this sentence grammatically correct: [sentence]”). ChatGPT was also used a few times to help with synonyms (e.g., the prompt “What are 10 other ways to say [word]”). Grammarly [166] has been used throughout the entire thesis to help with grammar and spelling.

7

Conclusion

This thesis presents research on triggers for test maintenance, capabilities of Large Language Models (LLMs) regarding test maintenance, and four working prototypes showing how LLMs can identify whether test maintenance is required and then localise where the change needs to be done. This research was conducted as a case study at partner company Ericsson.

As a preparatory step for the case study, a literature review on test maintenance triggers was performed which resulted in a list of 40 low-level triggers for test maintenance, i.e., changes in the production code that cause a need for test maintenance (e.g., adding a method parameter). Building on this, interviews and a survey were conducted as well. These resulted in a list of 7 high-level triggers for test maintenance, triggers that represent an entire use case or process, e.g. adding a new feature to the product. An additional literature review was conducted to explore how LLMs and LLM agents can help with test maintenance and what corporations should take into account when using them. LLMs have many versatile uses, and though they have shown proficiency at code generation they have also been used to help with other steps of the development process, e.g. help developers understand code or act as a pair programming partner.

Based on the previously mentioned results, a set of four prototypes were developed. These prototypes consisted of LLMs and LLM agents and their purpose was to identify the need for test maintenance and the location (the test cases) where test maintenance is required. The prototypes were then evaluated using data from partner company Ericsson and the best-performing prototypes had an aggregated F1 score of 0.2767 and an average F1 score of 0.5151. This score offers a promising start as the basic idea works through an untuned, general, and small LLM, but also shows a need for improvement in future work. Particularly when it comes to the triggers as although they work, they are not sufficient on their own. Future work should additionally evaluate if the prototypes' performance is the result of using a comparatively small LLM, or if the results stem from the prototypes' architecture. Special attention should be given to the RAG tools, as these provide much of the information on which the agents base their decisions.

The prototypes built in this thesis focused on the low-level triggers found in the production code. We envision that a future, more advanced multi-LLM agent could

make use of the high-level triggers and that the prototypes developed in this thesis might act as a smaller part of such a larger setup. These types of future agents might provide support and automation through the development process.

Bibliography

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

[2] S. Wang, M. Wen, Y. Liu, Y. Wang, and R. Wu, "Understanding and facilitating the co-evolution of production and test code," in *2021 IEEE International conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2021, pp. 272–283.

[3] Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen, and X. Peng, "No more manual tests? evaluating and improving chatgpt for unit test generation," *arXiv preprint arXiv:2305.04207*, 2023.

[4] (2024) ChatGPT. [Online]. Available: https://openai.com/chatgpt

[5] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, 2023.

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

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

[8] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," in *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 2023, pp. 23–30.

[9] N. M. S. Surameery and M. Y. Shakor, "Use chat gpt to solve programming bugs," *International Journal of Information Technology & Computer Engineering (IJITC) ISSN: 2455-5290*, vol. 3, no. 01, pp. 17–22, 2023.

[10] S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, "The program-

mer's assistant: Conversational interaction with a large language model for software development," in *Proceedings of the 28th International Conference on Intelligent User Interfaces*, 2023, pp. 491–514.

[11] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, 2024.

[12] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023.

[13] C. Qian, X. Cong, C. Yang, W. Chen, Y. Su, J. Xu, Z. Liu, and M. Sun, "Communicative agents for software development," *arXiv preprint arXiv:2307.07924*, 2023.

[14] T. Varshney. (2023) Introduction to llm agents. [Online]. Available: https://developer.nvidia.com/blog/introduction-to-llm-agents/

[15] R. Feldt, S. Kang, J. Yoon, and S. Yoo, "Towards autonomous testing agents via conversational large language models," *arXiv preprint arXiv:2306.05152*, 2023.

[16] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," *arXiv preprint arXiv:2402.01680*, 2024.

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

[18] H. M. Sneed, "A cost model for software maintenance & evolution," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 264–273.

[19] M. A. Umar and C. Zhanfang, "A study of automated software testing: Automation tools and frameworks," *International Journal of Computer Science Engineering (IJCSE)*, vol. 6, pp. 217–225, 2019.

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

[21] L. Berglund, T. Grube, G. Gay, F. G. de Oliveira Neto, and D. Platis, "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, 2023, pp. 410–421.

[22] X. Hu, Z. Liu, X. Xia, Z. Liu, T. Xu, and X. Yang, "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, 2023, pp. 1111–1122.

[23] C. R. Camacho, S. Marczak, and D. S. Cruzes, "Agile team members perceptions on non-functional testing: influencing factors from an empirical study," in *2016 11th international conference on availability, reliability and security (ARES)*. IEEE, 2016, pp. 582–589.

[24] "ISO/IEC/IEEE International Standard - Systems and software engineering," International Organization for Standardization, Tech. Rep. ISO/IEC/IEEE 24765:2017, Sep. 2017.

[25] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*. Wiley Online Library, 2004, vol. 2.

[26] H. K. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Proceedings. Conference on Software Maintenance 1990*. IEEE, 1990, pp. 290–301.

[27] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, 2012, pp. 1–11.

[28] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[29] J. Kasurinen, O. Taipale, and K. Smolander, "Software test automation in practice: empirical observations," *Advances in Software Engineering*, vol. 2010, 2010.

[30] M. A. Umar and C. Zhanfang, "A study of automated software testing: Automation tools and frameworks," *International Journal of Computer Science Engineering (IJCSE)*, vol. 6, pp. 217–225, 2019.

[31] V. Garousi and M. Felderer, "Developing, verifying, and maintaining high-quality automated test scripts," *IEEE Software*, vol. 33, no. 3, pp. 68–75, 2016.

[32] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of systems and software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[33] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically generated test code: An empir-

ical study," in *Proceedings of the 9th international workshop on search-based software testing*, 2016, pp. 5–14.

[34] G. Grano, S. Scalabrino, H. C. Gall, and R. Oliveto, "An empirical investigation on the readability of manual and generated test cases," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 348–351.

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

[36] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 4–15.

[37] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

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

[39] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A comprehensive survey on transfer learning," *Proceedings of the IEEE*, vol. 109, no. 1, pp. 43–76, 2020.

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

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

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

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

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

[45] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba,

"Large language models are human-level prompt engineers," *arXiv preprint arXiv:2211.01910*, 2022.

[46] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," in *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–7.

[47] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[48] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[49] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

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

[51] S. S. Kannan, V. L. Venkatesh, and B.-C. Min, "Smart-llm: Smart multi-agent robot task planning using large language models," *arXiv preprint arXiv:2309.10062*, 2023.

[52] G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl, "The risks of coverage-directed test case generation," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 803–819, 2015.

[53] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "Automatic test case generation: What if test code quality matters?" in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 130–141.

[54] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010, pp. 257–266.

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

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

[57] P. S. Kochhar, X. Xia, and D. Lo, "Practitioners' views on good software testing practices," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 61–70.

[58] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[59] Y. Huang, Z. Tang, X. Chen, and X. Zhou, "Towards automatically identifying the co-change of production and test code," *Software Testing, Verification and Reliability*, p. e1870, 2024.

[60] T. Kitai, H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "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, 2022, pp. 120–124.

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

[62] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE software*, vol. 26, no. 1, pp. 26–33, 2009.

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

[64] L. Vidács and M. Pinzger, "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, 2018, pp. 31–36.

[65] C. Marsavina, D. Romano, and A. Zaidman, "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, 2014, pp. 195–204.

[66] J. Sohn and M. Papadakis, "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, 2022, pp. 133–144.

[67] W. Sun, M. Yan, Z. Liu, X. Xia, Y. Lei, and D. Lo, "Revisiting the identification of the co-evolution of production and test code," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–37, 2023.

[68] C. Klammer, G. Buchgeher, and A. Kern, "A retrospective of production and test code co-evolution in an industrial project," in *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*. IEEE, 2018, pp. 16–20.

[69] S. Shimmi and M. Rahimi, "Patterns of code-to-test co-evolution for automated test suite maintenance," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2022, pp. 116–127.

[70] P. Reich and W. Maalei, "Testability refactoring in pull requests: Patterns and trends," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1508–1519.

[71] L. Liu, S. Wang, Y. Liu, J. Deng, and S. Liu, "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*, 2023, pp. 227–237.

[72] M. Mirzaaghaei, F. Pastore, and M. Pezze, "Automatically repairing test cases for evolving method declarations," in *2010 ieee international conference on software maintenance*. IEEE, 2010, pp. 1–5.

[73] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Automatic test case evolution," *Software Testing, Verification and Reliability*, vol. 24, no. 5, pp. 386–411, 2014.

[74] M. Mirzaaghaei, "Automatic test suite evolution," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 396–399.

[75] R. Gozalo-Brizuela and E. C. Garrido-Merchan, "Chatgpt is not all you need. a state of the art review of large generative ai models," *arXiv preprint arXiv:2301.04655*, 2023.

[76] S. Mandvikar, "Factors to consider when selecting a large language model: A comparative analysis," *International Journal of Intelligent Automation and Computing*, vol. 6, no. 3, pp. 37–40, 2023.

[77] J.-B. Döderlein, M. Acher, D. E. Khelladi, and B. Combemale, "Piloting copilot and codex: Hot temperature, cold prompts, or black magic?" *arXiv preprint arXiv:2210.14699*, 2022.

[78] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang *et al.*, "A survey on evaluation of large language models," *ACM Transactions on Intelligent Systems and Technology*, 2023.

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

[80] J. Wang and Y. Chen, "A review on code generation with llms: Application and evaluation," in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, 2023, pp. 284–289.

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

[82] J. Uusnäkki, "Design principles for prompt engineering within large language models: Case study on software maintenance," 2023.

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

[84] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can large language models reason about program invariants?" 2023.

[85] C. Liu, S. Lu, W. Chen, D. Jiang, A. Svyatkovskiy, S. Fu, N. Sundaresan, and N. Duan, "Code execution with pre-trained language models," *arXiv preprint arXiv:2305.05383*, 2023.

[86] J. T. Liang, C. Yang, and B. A. Myers, "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[87] W. Rahmaniar, "Chatgpt for software development: Opportunities and challenges," 2023.

[88] O. B. Sghaier and H. Sahraoui, "A multi-step learning approach to assist code review," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 450–460.

[89] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–69, 2023.

[90] C. S. Xia, Y. Wei, and L. Zhang, "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, 2023.

[91] A. R. Ibrahimzada, Y. Chen, R. Rong, and R. Jabbarvand, "Automated bug generation in the era of large language models," *arXiv preprint arXiv:2310.02407*, 2023.

[92] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[93] S. Dou, J. Shan, H. Jia, W. Deng, Z. Xi, W. He, Y. Wu, T. Gui, Y. Liu, and X. Huang, "Towards understanding the capability of large language models on code clone detection: a survey," *arXiv preprint arXiv:2308.01191*, 2023.

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

[95] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.

[96] X. Ren, X. Ye, D. Zhao, Z. Xing, and X. Yang, "From misuse to mastery: Enhancing code generation with knowledge-driven ai chaining," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 976–987.

[97] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language model: Survey, landscape, and vision," *arXiv preprint arXiv:2307.07221*, 2023.

[98] M. L. Siddiq, J. C. Santos, R. H. Tanvir, N. Ulfat, F. Al Rifat, and V. C. Lopes, "Using large language models to generate junit tests: An empirical study," 2024.

[99] S. Kang, J. Yoon, and S. Yoo, "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, 2023, pp. 2312–2323.

[100] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *International conference on software engineering (ICSE)*, 2023.

[101] G. Gay, "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, 2023.

[102] X. Jiang, Y. Dong, L. Wang, Q. Shang, and G. Li, "Self-planning code generation with large language model," *arXiv preprint arXiv:2303.06689*, 2023.

[103] X. Zhao, Y. Xie, K. Kawaguchi, J. He, and Q. Xie, "Automatic model selection with large language models for reasoning," *arXiv preprint arXiv:2305.14333*, 2023.

[104] S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou *et al.*, "Metagpt: Meta programming for multi-agent collaborative framework," *arXiv preprint arXiv:2308.00352*, 2023.

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

[106] W. Shen, C. Li, H. Chen, M. Yan, X. Quan, H. Chen, J. Zhang, and F. Huang, "Small llms are weak tool learners: A multi-llm agent," *arXiv preprint arXiv:2401.07324*, 2024.

[107] J. Yoon, R. Feldt, and S. Yoo, "Autonomous large language model agents enabling intent-driven mobile gui testing," *arXiv preprint arXiv:2311.08649*, 2023.

[108] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, pp. 131–164, 2009.

[109] S. Keele *et al.*, "Guidelines for performing systematic literature reviews in software engineering," 2007.

[110] (2024) Miro. [Online]. Available: https://miro.com/

[111] Microsoft. (2024) View live transcription in microsoft teams meetings. [Online]. Available: https://support.microsoft.com/en-au/office/view-live-transcription-in-microsoft-teams-meetings-dc1a8f23-2e20-4684-885e-2152e06a4a8b

[112] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative research in psychology*, vol. 3, no. 2, pp. 77–101, 2006.

[113] A. N. Ghazi, K. Petersen, S. S. V. R. Reddy, and H. Nekkanti, "Survey research in software engineering: Problems and mitigation strategies," *IEEE Access*, vol. 7, pp. 24 703–24 718, 2018.

[114] M. Kasunic, "Designing an effective survey," 2005.

[115] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin *et al.*, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, pp. 1–26, 2024.

[116] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou *et al.*, "The rise and potential of large language model based agents: A survey," *arXiv preprint arXiv:2309.07864*, 2023.

[117] J. Li, Q. Zhang, Y. Yu, Q. Fu, and D. Ye, "More agents is all you need," *arXiv preprint arXiv:2402.05120*, 2024.

[118] H. Chase, "Langchain," 2022. [Online]. Available: https://www.langchain.com/

[119] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

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

[121] (2024) Github - Mockito. [Online]. Available: https://github.com/mockito/mockito

[122] Z. Nussbaum, J. X. Morris, B. Duderstadt, and A. Mulyar, "Nomic embed: Training a reproducible long context text embedder," *arXiv preprint arXiv:2402.01613*, 2024.

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

[124] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," *arXiv preprint arXiv:1904.10509*, 2019.

[125] J. Ainslie, S. Ontanon, C. Alberti, V. Cvicek, Z. Fisher, P. Pham, A. Ravula, S. Sanghai, Q. Wang, and L. Yang, "Etc: Encoding long and structured inputs in transformers," *arXiv preprint arXiv:2004.08483*, 2020.

[126] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[127] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "Gqa: Training generalized multi-query transformer models from multi-head checkpoints," *arXiv preprint arXiv:2305.13245*, 2023.

[128] G. Sidorov, A. Gelbukh, H. Gómez-Adorno, and D. Pinto, "Soft similarity and soft cosine measure: Similarity of features in vector space model," *Computación y Sistemas*, vol. 18, no. 3, pp. 491–504, 2014.

[129] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint*

arXiv:1810.04805, 2018.

[130] (2024) Training data for text embedding models. [Online]. Available: https://huggingface.co/datasets/sentence-transformers/embedding-training-data

[131] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Supporting test suite evolution through test case adaptation," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 231–240.

[132] V. Bayrı and E. Demirel, "Ai-powered software testing: The impact of large language models on testing methodologies," in *2023 4th International Informatics and Software Engineering Conference (IISEC)*. IEEE, 2023, pp. 1–4.

[133] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, 2023, pp. 423–435.

[134] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "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*, 2024, pp. 1–13.

[135] Z. Liu, C. Chen, J. Wang, X. Che, Y. Huang, J. Hu, and Q. Wang, "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, 2023, pp. 1355–1367.

[136] D. Zimmermann and A. Koziolek, "Automating gui-based software testing with gpt-3," in *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2023, pp. 62–65.

[137] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[138] J. Zhu, G. Xiao, Z. Zheng, and Y. Sui, "Enhancing traceability link recovery with unlabeled data," in *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2022, pp. 446–457.

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

[140] C. Bird, D. Ford, T. Zimmermann, N. Forsgren, E. Kalliamvakou, T. Lowdermilk, and I. Gazit, "Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools," *Queue*, vol. 20, no. 6, pp. 35–57, 2022.

[141] A. Nazir and Z. Wang, "A comprehensive survey of chatgpt: Advancements, applications, prospects, and challenges," *Meta-radiology*, p. 100022, 2023.

[142] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2024, pp. 881–881.

[143] L. Belzner, T. Gabor, and M. Wirsing, "Large language model assisted software engineering: prospects, challenges, and a case study," in *International Conference on Bridging the Gap between AI and Reality*. Springer, 2023, pp. 355–374.

[144] J. Lu, L. Yu, X. Li, L. Yang, and C. Zuo, "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, 2023, pp. 647–658.

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

[146] S. Yu, C. Fang, Y. Ling, C. Wu, and Z. Chen, "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, 2023, pp. 206–217.

[147] C. Tsigkanos, P. Rani, S. Müller, and T. Kehrer, "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, 2023, pp. 678–682.

[148] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "The hitchhiker's guide to program analysis: A journey with large language models," *arXiv preprint arXiv:2308.00245*, 2023.

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

[150] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt

really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

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

[152] M. Shanahan, "Talking about large language models," *Communications of the ACM*, vol. 67, no. 2, pp. 68–79, 2024.

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

[154] (2024) Mistral: Models. [Online]. Available: https://docs.mistral.ai/getting-started/models/

[155] E. N. Efthimiadis, "Query expansion." *Annual review of information science and technology (ARIST)*, vol. 31, pp. 121–87, 1996.

[156] C. Carpineto and G. Romano, "A survey of automatic query expansion in information retrieval," *Acm Computing Surveys (CSUR)*, vol. 44, no. 1, pp. 1–50, 2012.

[157] R. Nogueira and K. Cho, "Passage re-ranking with bert," *arXiv preprint arXiv:1901.04085*, 2019.

[158] M. Glass, G. Rossiello, M. F. M. Chowdhury, A. R. Naik, P. Cai, and A. Gliozzo, "Re2g: Retrieve, rerank, generate," *arXiv preprint arXiv:2207.06300*, 2022.

[159] (2023) IBM Artificial Intelligence Pillars. [Online]. Available: https://www.ibm.com/policy/ibm-artificial-intelligence-pillars/

[160] (2024) Mistral: Open-weight models. [Online]. Available: https://docs.mistral.ai/getting-started/open_weight_models/

[161] A. S. Luccioni, S. Viguier, and A.-L. Ligozat, "Estimating the carbon footprint of bloom, a 176b parameter language model," *Journal of Machine Learning Research*, vol. 24, no. 253, pp. 1–15, 2023.

[162] (2024) Normal elförbrukning för villa & lägenhet - vattenfall. [Online]. Available: https://www.vattenfall.se/fokus/tips-rad/vad-ar-normal-elforbrukning/

[163] A. de Vries, "The growing energy footprint of artificial intelligence," *Joule*, vol. 7, no. 10, pp. 2191–2194, 2023.

[164] (2022) Sverige har överträffat målet om andel förnybar energi för 2020. [Online]. Available: https://www.energimyndigheten.se/nyhetsarkiv/2022/sverige-har-overtraffat-malet-om-andel-fornybar-energi-for-2020/

[165] (2024) Artificial intelligence act: Meps adopt landmark law. [Online]. Available: https://www.europarl.europa.eu/news/en/press-room/20240308IPR19015/artificial-intelligence-act-meps-adopt-landmark-law

[166] (2024) Grammarly. [Online]. Available: https://www.grammarly.com/

[167] (2024) langchain-ai/react-agent-template. [Online]. Available: https://smith.langchain.com/hub/langchain-ai/react-agent-template

A

Search Strings for Test Maintenance Factors Literature Review

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) AND (LIMIT-TO (EXACTKEYWORD , "Software Testing"))
- test AND suite AND evolution AND (LIMIT-TO (EXACTKEYWORD , "Software Testing"))
- "test suite" AND factors AND maintenance AND NOT reduction AND (LIMIT-TO (SUBJAREA , "COMP"))
- (test AND case OR test AND suite OR testing AND scenario) AND (update OR create OR refactor OR modification OR improve OR generate OR alter OR revision OR maintenance OR evolution OR management OR repair OR co-evolution) AND (source AND code OR code AND changes OR codebase OR software AND code) AND (factors OR indicators OR criteria OR signs) AND (LIMIT-TO (SUBJAREA , "COMP")) AND (LIMIT-TO (EXACTKEYWORD , "Software Testing"))
- ("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) AND (factors OR criteria) AND ("foundational model" OR "mission learning" OR "LLM" OR "Large language model")
- ("test case" or "test suite") AND maintenance AND (“foundational model” OR “mission learning” OR “LLM” OR “Large language model”)

ACM:

- (“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 (“source code” OR codebase) AND (factors OR criteria) AND NOT (reduction OR prioritization)
- "test suite" AND factors AND maintenance
- ("test case" or "test suite") AND maintenance AND (“foundational model” OR “mission learning” OR “LLM” OR “Large language model”)
- (“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) AND (“foundational model” OR “mission learning” OR “LLM” OR “Large language model”)

IEEE:

- (“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 suite" AND evolution AND “generative AI”
- "test suite" AND evolution AND "LLM”
- "test" AND evolution AND "LLM”
- "test" AND maintenance AND "LLM”
- "test" AND maintenance AND (“foundational model” OR “mission learning” OR “LLM” OR “Large language model”)
- ("test case" or "test suite") AND maintenance AND (“foundational model” OR “mission learning” OR “LLM” OR “Large language model”)

Google Scholar:

- ("test case" OR "test suite") AND (update OR create OR refactor OR generate OR maintenance OR evolution OR management OR repair OR co-evolution) AND ("source code" OR codebase) AND (factors OR criteria)
- "test suite" AND evolution AND "software engineering"

- "test case" AND evolution AND "software engineering"
- ("test suite" OR "test case") AND maintenance AND "factors"
- "test maintenance" AND factors "software"
- software maintenance unit tests
- software AND "test suite" AND "co-evolution"
- ("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) AND (“foundational model” OR “mission learning” OR LLM)

Science Direct:

- “test case” AND evolution
- "test case" AND factors AND maintenance
- "test suite" AND factors AND maintenance
- "software engineering" AND "test suite" AND "co-evolution"

B

Search Strings for LLM Capabilities Literature Review

SCOPUS:

- (llm OR "large language model" OR "generative AI" OR "natural language processing models") 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) AND (LIMIT-TO (SUBJAREA , "COMP"))
- (generative AND ai) AND (test AND maintenance) AND (LIMIT-TO (SUBJAREA , "COMP"))
- ("llm" OR "Large language model") AND (test AND maintenance) AND (LIMIT-TO (SUBJAREA , "COMP"))
- (LLM OR “large language model” OR “generative AI” OR “natural language processing models”) AND (survey OR “literature review” OR review) AND (“Software engineering” OR code) AND (comparison OR compare OR abilities)
- (llm OR "large language model" OR "generative AI" OR "natural language processing models") AND ("Software engineering" OR code) AND (comparison OR compare OR abilities)

ACM:

- (llm OR "large language model" OR "generative AI" OR "natural language processing models") 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) AND (“software testing”)

- (“software engineering”) AND (LLM OR “large language model”) AND (“test management” OR “test maintenance”)
- (“software engineering”) AND (“generative ai”) AND (“test management” OR “test maintenance”)
- (LLM OR "large language model" OR "generative AI") AND (“test code” OR “test suite”) AND (“software”)
- (LLM OR “large language model” OR “generative AI” OR “natural language processing models”) AND (survey OR “literature review” OR review) AND (“Software engineering” OR code) AND (comparison OR compare OR abilities)
- (LLM OR “large language model” OR “generative AI” OR “natural language processing models”) AND (augment OR modify) AND (“Software engineering” OR code) AND (suitability OR appropriateness OR abilities OR methods)

IEEE:

- (llm OR "large language model" OR "generative AI" OR "natural language processing models") 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)
- (llm OR "large language model" OR "generative AI" OR "natural language processing models") 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) AND (“software testing”)
- (“software engineering”) AND (LLM OR “large language model”) AND (“test

management” OR “test maintenance”)

- (LLM OR "large language model" OR "generative AI") AND (“test code” OR “test suite”) AND (“software”)
- (LLM OR “large language model” OR “generative AI” OR genAI) AND test* AND software*
- (LLM OR “large language model” OR “generative AI” OR “natural language processing models”) AND (survey OR “literature review” OR review) AND (“Software engineering” OR code) AND (comparison OR compare OR abilities)
- (LLM OR “large language model” OR “generative AI” OR “natural language processing models”) AND (augment OR modify) AND (“Software engineering” OR code) AND (suitability OR appropriateness OR abilities OR methods)

Google Scholar:

- ("software engineering") AND (suitability OR appropriateness OR abilities OR methods) AND (LLM OR “large language model” OR “generative AI” OR “natural language processing models”) 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 (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)
- (“software engineering”) AND (LLM OR “large language model”) AND (“test management” OR “test maintenance”)
- (LLM OR “large language model” OR “generative AI” OR “natural language processing models”) AND (survey OR “literature review” OR review) AND (“Software engineering” OR code) AND (comparison OR compare OR abilities)
- (LLM OR “large language model” OR “generative AI” OR “natural language processing models”) AND (augment OR modify) AND (“Software engineering” OR code) AND (suitability OR appropriateness OR abilities OR methods)

Science Direct:

- (LLM OR “large language model”) AND (trigger OR action) AND (“test management” OR “test maintenance”) AND (suitability OR abilities OR method)
- (LLM OR “large language model”) AND (“test management” OR “test main-

tenance” OR "Software testing") AND (suitability OR abilities OR method)

- (LLM OR “large language model”) AND (“test management” OR “test maintenance” OR "Software testing")
- (LLM OR “large language model” OR AI) AND (“test management” OR “test maintenance” OR "Software testing")
- (llm OR "large language model" OR "generative AI") AND (trigger OR action OR update OR create OR generate) AND (test)
- (LLM OR “large language model” OR “generative AI”) AND (survey OR “literature review” OR review) AND code AND (comparison)

C

Interview Consent Form

Consent form

Master Thesis Name: Test Case Generation and Updating During Test Maintenance Using Generative AI Interviewers: Ludvig Lemner, Linnea Wahlgren Supervisors: Nasser Mohammadiha (Ericsson), Roy Liu A (Ericsson), Joakim Wennerberg (Ericsson), Gregory Gay (Chalmers)

This interview is a part of a master thesis on how generative AI can be used to help with test maintenance. This interview will focus on which factors affect test maintenance, as well as which changes in the production code leads to changes in the test suite. The interview will be recorded using Microsoft Teams, and the recording will be stored in Teams. Your answers will be used when reporting the results for the master thesis, however no names or individual details will be reported in the thesis (besides demographic information). The recording will not be shared with people outside the students and supervisors working on the master thesis. The consent form will be handled the same way. This interview is voluntary and the participant can choose to stop participation at any point.

I consent to what is stated above: Yes / No

Name:_____ Date:_____

D

Interview Questions

- What is your role?
  - What types of testing do you perform?
  - What kind of programming language are you working with?
  - How many years of experience do you have within this area?
  - How much time do you spend on production code vs test code?
  - What is the proportion between test code and production code in your repo?
- What reasons have you encountered for making changes in the test suite or an individual test case?
  - Can you give some examples?
    - (Examples of examples: Smallest change led to greatest amount of work, most common change, most tedious change, a unique case)
  - What kind of information do you use to make the appropriate changes in the tests?
  - What kind of modifications do you usually apply on the tests?
  - What are some additional factors you consider when making changes?
  - How do changes in production reflect changes in test code?
- What specific changes in the source code-under-test lead to updates in the test suite? We are asking specifically for syntactic changes in the code.
  - Why is that so?
  - Which specific examples have you encountered?

    - (For example: adding new code (class or method, etc) or modifying existing code (method, class, signatures, return statements, etc))

- 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?
- Can you describe the whole chain of events from what instigated the change in the test suite to when the change is complete? Which 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?
- What kind of tool-related assistance would you like with updating the test suite?
  - If you cannot think of anything: Suggestions on areas that need to be modified? Pseudo code for new or changed tests? Automatic IDE tools?
  - What would be most valuable?
- Do you use any tools to help update or create test cases?
  - Any automated tools?
  - Do they work well?
  - If you have used LLMs, which, and how did you find the experience?
    - Would you like to use them?
    - They can be used for more creative purposes (advice on processes or tasks) and for boilerplate purposes (generating code), do you have a preference?
- What consequences can there be from a change in the test suite?
  - (Not just tools but in general, including manual testing changes)
  - Do these consequences affect how you approach your work?
  - How much additional effort can these consequences require?
- What is the difference between updating existing test cases versus creating new test cases?

- Are there different reasons/factors instigating these?
- Are there generally differing amounts of effort required?
- What is most common and why?

E

Survey Instrument Questions

E.1 Demographic Questions

What is your work role?

- Developer
- Tester
- Architect
- DevOps
- Other:

How many years of experience do you have with software testing?

- No experience
- <1
- 1-2
- 3-5
- 6-10
- >10

What kind of testing do you perform most often?

- Unit testing
- Integration testing
- System testing
- Acceptance testing

- Other:

Which programming language do you mainly work with? If you are unsure, pick the one you used most recently.

- C/C++
- Erlang
- Go Lang
- Java
- Julia
- Python
- Other:

E.2 Test Maintenance Questions

When making changes in the code, do you most often update production code or test code first?

- Production code
- Test code
- Both simultaneously
- I don't make changes in the code
- Don't know

What reasons have you encountered for making changes in the test suite or in an individual test case? [max 4]

- Bug in test code
- Bug in production code
- Addition of new feature
- Changes in requirements
- Improve coverage of tests
- Improve performance of tests
- Improve test functionality

- Changes in tech stack, e.g. tools, language, framework
- Does not apply
- Other:

Which specific types of production code changes most commonly necessitate adjustments in the associated test code? [select up to five alternatives]

- Add new method/function
- Remove method/function
- Change method/function body
- Change method/function return type or value
- Change method/function signature
- Add new class
- Remove class
- Change class declaration
- Change type of e.g. object or variable
- Add parameter
- Remove parameter
- Change documentation
- Change iterative statements (loop handling)
- Change conditional statements
- Change single line of code, e.g. variable assignment/identifier/modifier
- Change error or exception handling
- Change interface/class hierarchy
- Changes to interface
- Change parallelism/concurrency
- Other:

During the test maintenance process which step is most effort intensive?

- Identifying what needs to be done
- Planning how to implement the solution
- Actually implementing the solution
- Ensuring the solution is correct
- Other:

E.3 Generative AI and LLM Questions

Have you used generative AI or LLMs (Large Language Models)?

- Yes, all the time
- Yes, sometimes
- Yes, once or twice
- No, but I would like to try it
- No, I don't care
- No, and I don't want to
- Don't know

Would you like the help of generative AI/LLMs for test maintenance activities? [max 3]

- No
- Yes, for code generation.
- Yes, for helping me improve the code I wrote.
- Yes, for providing suggestions on what to do.
- Yes, for teaching me best practices.
- Yes, for keeping track of task relations so I don't miss anything.
- Yes, for helping me comment and document my code.
- Yes, for helping me understand existing code.
- Don't know
- Other:

F

Evaluation Results of the Four Prototypes for Each Commit

The results of the evaluation of the four prototypes are presented in Tables F.1, F.2, F.3, and F.4. The tables present how well the prototypes performed for each datapoint, i.e. commit.

Table F.1: Results of evaluation of prototype using an LLM chain with summaries of test cases. Correct TCs = The number of test cases changed in the original commit, TP = correctly found test cases, FP = incorrectly found test cases.

	Chain with summaries					
Commit #	Correct TCs	TP	FP	Recall	Precision	F1
Commit 1	16	7	22	0.4375	0.2414	0.3111
Commit 2	4	3	21	0.7500	0.1250	0.2143
Commit 3	0	0	6	1.0000	0.0000	0.0000
Commit 4	0	0	4	1.0000	0.0000	0.0000
Commit 5	10	0	11	0.0000	0.0000	0.0000
Commit 6	4	1	30	0.2500	0.0323	0.0571
Commit 7	0	0	0	1.0000	1.0000	1.0000
Commit 8	6	1	8	0.1667	0.1111	0.1333
Commit 9	1	1	0	1.0000	1.0000	1.0000
Commit 10	9	4	21	0.4444	0.1600	0.2353
Commit 11	0	0	5	1.0000	0.0000	1.0000
Commit 12	0	0	3	1.0000	0.0000	0.0000
Commit 13	0	0	7	1.0000	0.0000	0.0000
Commit 14	0	0	3	1.0000	0.0000	0.0000
Commit 15	13	0	5	0.0000	0.0000	0.0000
Commit 16	1	0	1	0.0000	0.0000	0.0000
Commit 17	0	0	2	1.0000	0.0000	0.0000
Commit 18	0	0	3	1.0000	0.0000	0.0000
Commit 19	16	1	11	0.0625	0.0833	0.0714
Commit 20	90	23	26	0.2556	0.4694	0.3309
Commit 21	51	23	33	0.4510	0.4107	0.4299

Continued on next page

Table F.1: Results of evaluation of prototype using an LLM chain with summaries of test cases. Correct TCs = The number of test cases changed in the original commit, TP = correctly found test cases, FP = incorrectly found test cases.

	Chain with summaries					
Commit #	Correct TCs	TP	FP	Recall	Precision	F1
Commit 22	6	0	17	0.0000	0.0000	0.0000
Commit 23	1	0	4	0.0000	0.0000	0.0000
Commit 24	3	3	4	1.0000	0.4286	0.6000
Commit 25	0	0	14	1.0000	0.0000	0.0000
Commit 26	0	0	0	1.0000	1.0000	1.0000
Commit 27	0	0	5	1.0000	0.0000	0.0000
Commit 28	14	2	5	0.1429	0.2857	0.1905
Commit 29	1	0	3	0.0000	0.0000	0.0000
Commit 30	9	5	1	0.5556	0.8333	0.6667
Commit 31	0	0	7	1.0000	0.0000	0.0000
Commit 32	3	0	0	0.0000	1.0000	0.0000
Commit 33	0	0	5	1.0000	0.0000	0.0000
Commit 34	11	7	13	0.6364	0.3500	0.4516
Commit 35	0	0	3	1.0000	0.0000	0.0000
Commit 36	5	1	20	0.2000	0.0476	0.0769
Commit 37	0	0	6	1.0000	0.0000	0.0000
Commit 38	0	0	6	1.0000	0.0000	0.0000
Commit 39	0	0	6	1.0000	0.0000	0.0000
Commit 40	0	0	0	1.0000	1.0000	1.0000
Commit 41	0	0	2	1.0000	0.0000	0.0000
Commit 42	10	3	19	0.3000	0.1364	0.1875
Commit 43	2	1	13	0.5000	0.0714	0.1250
Commit 44	0	0	9	1.0000	0.0000	0.0000
Commit 45	1	0	3	0.0000	0.0000	0.0000
Commit 46	0	0	4	1.0000	0.0000	0.0000
Commit 47	0	0	2	1.0000	0.0000	0.0000
Commit 48	4	0	0	0.0000	1.0000	0.0000
Commit 49	21	3	15	0.1429	0.1667	0.1538
Commit 50	2	1	4	0.5000	0.2000	0.2857
Commit 51	5	0	3	0.0000	0.0000	0.0000
Commit 52	1	0	0	0.0000	1.0000	0.0000
Commit 53	3	2	8	0.6667	0.2000	0.3077
Commit 54	0	0	3	1.0000	0.0000	0.0000
Commit 55	0	0	8	1.0000	0.0000	0.0000
Commit 56	1	0	4	0.0000	0.0000	0.0000
Commit 57	1	0	6	0.0000	0.0000	0.0000

Table F.2: Results of evaluation of prototype using an LLM chain without summarising the test cases in natural language. Correct TCs = The number of test cases changed in the original commit, TP = correctly found test cases, FP = incorrectly found test cases.

	Chain without summaries					
Commit #	Correct TCs	TP	FP	Recall	Precision	F1
Commit 1	16	4	41	0.2500	0.0889	0.1311
Commit 2	4	2	20	0.5000	0.0909	0.1538
Commit 3	0	0	11	1.0000	0.0000	0.0000
Commit 4	0	0	3	1.0000	0.0000	0.0000
Commit 5	10	0	7	0.0000	0.0000	0.0000
Commit 6	4	1	26	0.2500	0.0370	0.0645
Commit 7	0	0	0	1.0000	1.0000	1.0000
Commit 8	6	1	7	0.1667	0.1250	0.1429
Commit 9	1	1	3	1.0000	0.2500	0.4000
Commit 10	9	3	18	0.3333	0.1429	0.2000
Commit 11	0	0	7	1.0000	0.0000	0.0000
Commit 12	0	0	0	1.0000	1.0000	1.0000
Commit 13	0	0	2	1.0000	0.0000	0.0000
Commit 14	0	0	5	1.0000	0.0000	0.0000
Commit 15	13	0	14	0.0000	0.0000	0.0000
Commit 16	1	0	3	0.0000	0.0000	0.0000
Commit 17	0	0	5	1.0000	0.0000	0.0000
Commit 18	0	0	2	1.0000	0.0000	0.0000
Commit 19	16	0	3	0.0000	0.0000	0.0000
Commit 20	90	26	49	0.2889	0.3467	0.3152
Commit 21	51	13	50	0.2549	0.2063	0.2281
Commit 22	6	0	28	0.0000	0.0000	0.0000
Commit 23	1	0	4	0.0000	0.0000	0.0000
Commit 24	3	1	5	0.3333	0.1667	0.2222
Commit 25	0	0	13	1.0000	0.0000	0.0000
Commit 26	0	0	0	1.0000	1.0000	1.0000
Commit 27	0	0	4	1.0000	0.0000	0.0000
Commit 28	14	0	14	0.0000	0.0000	0.0000
Commit 29	1	0	2	0.0000	0.0000	0.0000
Commit 30	9	6	6	0.6667	0.5000	0.5714
Commit 31	0	0	6	1.0000	0.0000	0.0000
Commit 32	3	0	3	0.0000	0.0000	0.0000
Commit 33	0	0	9	1.0000	0.0000	0.0000
Commit 34	11	5	34	0.4546	0.1282	0.2000
Commit 35	0	0	5	1.0000	0.0000	0.0000
Commit 36	5	0	24	0.0000	0.0000	0.0000
Commit 37	0	0	2	1.0000	0.0000	0.0000

Continued on next page

Table F.2: Results of evaluation of prototype using an LLM chain without summarising the test cases in natural language. Correct TCs = The number of test cases changed in the original commit, TP = correctly found test cases, FP = incorrectly found test cases.

	Chain without summaries					
Commit #	Correct TCs	TP	FP	Recall	Precision	F1
Commit 38	0	0	5	1.0000	0.0000	0.0000
Commit 39	0	0	3	1.0000	0.0000	0.0000
Commit 40	0	0	0	1.0000	1.0000	0.0000
Commit 41	0	0	3	1.0000	0.0000	0.0000
Commit 42	10	2	21	0.2000	0.0870	0.1212
Commit 43	2	1	8	0.5000	0.1111	0.1819
Commit 44	0	0	13	1.0000	0.0000	0.0000
Commit 45	1	0	3	0.0000	0.0000	0.0000
Commit 46	0	0	3	1.0000	0.0000	0.0000
Commit 47	0	0	2	1.0000	0.0000	0.0000
Commit 48	4	0	3	0.0000	0.0000	0.0000
Commit 49	21	3	49	0.1429	0.0577	0.0822
Commit 50	2	0	2	0.0000	0.0000	0.0000
Commit 51	5	0	0	0.0000	1.0000	0.0000
Commit 52	1	0	0	1.0000	0.0000	0.0000
Commit 53	3	0	4	0.0000	0.0000	0.0000
Commit 54	0	0	8	1.0000	0.0000	0.0000
Commit 55	0	0	11	1.0000	0.0000	0.0000
Commit 56	1	0	0	0.0000	1.0000	0.0000
Commit 57	1	0	4	0.0000	0.0000	0.0000

Table F.3: Result of evaluation of multi-agent setup with a planning agent and summaries in natural language of test cases. Correct TCs = The number of test cases changed in the original commit, TP = correctly found test cases, FP = incorrectly found test cases.

	Planning with summaries					
Commit #	Correct TCs	TP	FP	Recall	Precision	F1
Commit 1	16	11	10	0.6875	0.5238	0.5946
Commit 2	4	3	21	0.7500	0.1250	0.2143
Commit 3	0	0	5	1.0000	0.0000	0.0000
Commit 4	0	0	0	1.0000	1.0000	1.0000
Commit 5	10	0	3	0.0000	0.0000	0.0000
Commit 6	4	2	19	0.5000	0.0952	0.1600
Commit 7	0	0	0	1.0000	1.0000	1.0000
Commit 8	6	0	5	0.0000	0.0000	0.0000
Commit 9	1	0	0	0.0000	1.0000	0.0000
Continued on next page						

Table F.3: Result of evaluation of multi-agent setup with a planning agent and summaries in natural language of test cases. Correct TCs = The number of test cases changed in the original commit, TP = correctly found test cases, FP = incorrectly found test cases.

	Planning with summaries					
Commit #	Correct TCs	TP	FP	Recall	Precision	F1
Commit 10	9	6	21	0.6667	0.2222	0.3333
Commit 11	0	0	3	1.0000	0.0000	0.0000
Commit 12	0	0	0	1.0000	1.0000	1.0000
Commit 13	0	0	5	1.0000	0.0000	0.0000
Commit 14	0	0	5	1.0000	0.0000	0.0000
Commit 15	13	4	0	0.3077	1.0000	0.4706
Commit 16	1	0	0	0.0000	1.0000	0.0000
Commit 17	0	0	7	1.0000	0.0000	0.0000
Commit 18	0	0	0	1.0000	1.0000	1.0000
Commit 19	16	0	4	0.0000	0.0000	0.0000
Commit 20	90	28	26	0.3111	0.5185	0.3889
Commit 21	51	17	20	0.3333	0.4595	0.3864
Commit 22	6	0	5	0.0000	0.0000	0.0000
Commit 23	1	0	0	0.0000	1.0000	0.0000
Commit 24	3	1	1	0.3333	0.5000	0.4000
Commit 25	0	0	7	1.0000	0.0000	0.0000
Commit 26	0	0	0	1.0000	1.0000	1.0000
Commit 27	0	0	6	1.0000	0.0000	0.0000
Commit 28	14	4	3	0.2857	0.5714	0.3810
Commit 29	1	0	4	0.0000	0.0000	0.0000
Commit 30	9	7	4	0.7778	0.6364	0.7000
Commit 31	0	0	3	1.0000	0.0000	0.0000
Commit 32	3	0	3	0.0000	0.0000	0.0000
Commit 33	0	0	2	1.0000	0.0000	0.0000
Commit 34	11	8	22	0.7273	0.2667	0.3902
Commit 35	0	0	3	1.0000	0.0000	0.0000
Commit 36	5	1	11	0.2000	0.0833	0.1176
Commit 37	0	0	2	1.0000	0.0000	0.0000
Commit 38	0	0	5	1.0000	0.0000	0.0000
Commit 39	0	0	0	1.0000	1.0000	1.0000
Commit 40	0	0	0	1.0000	1.0000	1.0000
Commit 41	0	0	3	1.0000	0.0000	0.0000
Commit 42	10	5	7	0.5000	0.4167	0.4546
Commit 43	2	0	14	0.0000	0.0000	0.0000
Commit 44	0	0	0	1.0000	1.0000	1.0000
Commit 45	1	0	0	0.0000	1.0000	0.0000
Commit 46	0	0	3	1.0000	0.0000	0.0000

Continued on next page

Table F.3: Result of evaluation of multi-agent setup with a planning agent and summaries in natural language of test cases. Correct TCs = The number of test cases changed in the original commit, TP = correctly found test cases, FP = incorrectly found test cases.

	Planning with summaries					
Commit #	Correct TCs	TP	FP	Recall	Precision	F1
Commit 47	0	0	0	1.0000	1.0000	1.0000
Commit 48	4	0	3	0.0000	0.0000	0.0000
Commit 49	21	3	18	0.1429	0.1429	0.1429
Commit 50	2	1	5	0.5000	0.1667	0.2500
Commit 51	5	0	4	0.0000	0.0000	0.0000
Commit 52	1	0	0	0.0000	1.0000	0.0000
Commit 53	3	0	3	0.0000	0.0000	0.0000
Commit 54	0	0	2	1.0000	0.0000	0.0000
Commit 55	0	0	2	1.0000	0.0000	0.0000
Commit 56	1	0	3	0.0000	0.0000	0.0000
Commit 57	1	0	2	0.0000	0.0000	0.0000

Table F.4: Result of evaluation of multi-agent setup with a planning agent and without natural language summaries of test cases. Correct TCs = The number of test cases changed in the original commit, TP = correctly found test cases, FP = incorrectly found test cases.

	Planning without summaries					
Commit #	Correct TCs	TP	FP	Recall	Precision	F1
Commit 1	16	4	17	0.2500	0.1905	0.2162
Commit 2	4	3	9	0.7500	0.2500	0.3750
Commit 3	0	0	2	1.0000	0.0000	0.0000
Commit 4	0	0	0	1.0000	1.0000	1.0000
Commit 5	10	0	0	0.0000	1.0000	0.0000
Commit 6	4	0	9	0.0000	0.0000	0.0000
Commit 7	0	0	3	1.0000	0.0000	0.0000
Commit 8	6	0	0	0.0000	1.0000	0.0000
Commit 9	1	0	0	0.0000	1.0000	0.0000
Commit 10	9	3	12	0.3333	0.2000	0.2500
Commit 11	0	0	2	1.0000	0.0000	0.0000
Commit 12	0	0	2	1.0000	0.0000	0.0000
Commit 13	0	0	3	1.0000	0.0000	0.0000
Commit 14	0	0	0	1.0000	1.0000	1.0000
Commit 15	13	2	4	0.1538	0.3333	0.2105
Commit 16	1	0	0	0.0000	1.0000	0.0000
Commit 17	0	0	6	1.0000	0.0000	0.0000
Commit 18	0	0	0	1.0000	1.0000	1.0000

Continued on next page

Table F.4: Result of evaluation of multi-agent setup with a planning agent and without natural language summaries of test cases. Correct TCs = The number of test cases changed in the original commit, TP = correctly found test cases, FP = incorrectly found test cases.

	Planning without summaries					
Commit #	Correct TCs	TP	FP	Recall	Precision	F1
Commit 19	16	0	4	0.0000	0.0000	0.0000
Commit 20	90	15	14	0.1667	0.5172	0.2521
Commit 21	51	11	17	0.2157	0.3929	0.2785
Commit 22	6	0	8	0.0000	0.0000	0.0000
Commit 23	1	0	0	0.0000	1.0000	0.0000
Commit 24	3	2	5	0.6667	0.2857	0.4000
Commit 25	0	0	0	1.0000	1.0000	1.0000
Commit 26	0	0	0	1.0000	1.0000	1.0000
Commit 27	0	0	3	1.0000	0.0000	0.0000
Commit 28	14	0	2	0.0000	0.0000	0.0000
Commit 29	1	0	0	0.0000	1.0000	0.0000
Commit 30	9	0	4	0.0000	0.0000	0.0000
Commit 31	0	0	0	1.0000	1.0000	1.0000
Commit 32	3	1	1	0.3333	0.5000	0.4000
Commit 33	0	0	3	1.0000	0.0000	0.0000
Commit 34	11	7	7	0.6364	0.5000	0.5600
Commit 35	0	0	0	1.0000	1.0000	1.0000
Commit 36	5	1	7	0.2000	0.1250	0.1538
Commit 37	0	0	0	1.0000	1.0000	1.0000
Commit 38	0	0	0	1.0000	1.0000	1.0000
Commit 39	0	0	2	1.0000	0.0000	0.0000
Commit 40	0	0	0	1.0000	1.0000	1.0000
Commit 41	0	0	3	1.0000	0.0000	0.0000
Commit 42	10	0	5	0.0000	0.0000	0.0000
Commit 43	2	0	3	0.0000	0.0000	0.0000
Commit 44	0	0	0	1.0000	1.0000	1.0000
Commit 45	1	0	0	0.0000	1.0000	0.0000
Commit 46	0	0	0	1.0000	1.0000	1.0000
Commit 47	0	0	3	1.0000	0.0000	0.0000
Commit 48	4	0	0	0.0000	1.0000	0.0000
Commit 49	21	3	9	0.1429	0.2500	0.1819
Commit 50	2	1	3	0.5000	0.2500	0.3333
Commit 51	5	0	2	0.0000	0.0000	0.0000
Commit 52	1	0	0	0.0000	1.0000	0.0000
Commit 53	3	1	6	0.3333	0.1429	0.2000
Commit 54	0	0	0	1.0000	1.0000	1.0000
Commit 55	0	0	0	1.0000	1.0000	1.0000

Continued on next page

Table F.4: Result of evaluation of multi-agent setup with a planning agent and without natural language summaries of test cases. Correct TCs = The number of test cases changed in the original commit, TP = correctly found test cases, FP = incorrectly found test cases.

	Planning without summaries					
Commit #	Correct TCs	TP	FP	Recall	Precision	F1
Commit 56	1	1	3	1.0000	0.2500	0.4000
Commit 57	1	1	5	1.000	0.1667	0.2857

G

Prompts Used For Agents

The sections will present the prompts used in each agent within the four different architectures, including the base ReAct prompt. The prompts are mainly the same, though minor variations occur to better fit them to the specific architecture.

G.1 React Agent Base Prompt

Prompt for React agent [167]. When using the prompt, *instructions* would be replaced by the instructions given by the user. The instructions/prompts used in this thesis are available in the following sections.

```
    {instructions}

TOOLS:

------

You have access to the following tools:

{tools}

To use a tool, please use the following format:

```

Thought: Do I need to use a tool? Yes

Action: the action to take, should be one of [{tool_names}]

Action Input: the input to the action

Observation: the result of the action

```

When you have a response to say to the Human, or if you do not need
    to use a tool, you MUST use the format:

```
```
```

```
Thought: Do I need to use a tool? No

Final Answer: [your response here]

```

Begin!

Previous conversation history:

{chat_history}

New input: {input}

{agent_scratchpad}
```

## G.2 Planning agent with summaries

### G.2.1 Code Summariser Agent

```
You are large language model working as a software developer,
   specifically with summarising code changes.
You will be given a code snippet with new code as input where the
   code has been changed compared to the old code.
You work step by step and first use your tools to find what the old
    code looks like.
You then compare the old code you found through the tool and the
   new code you were given as input to find the changes.
You then write a short summary of the changes so that it is easy to
    understand.
Do NOT use any tools after finding the old code.
ONLY use a tool ONCE.
After this you should answer with Final Answer.

Here are some examples of what good summaries look like:
*The return value was changed from X to Y
*An additional else if statement was added
*A parameter was removed from the method signature.
*A variable was renamed from X to Y
```

G.2.2 Planning Agent

Note that the prompt includes a code example from the Mockito repository [165].

```
You are a large language model that oversee other large language
   models.

You are designed to be able to assist with test maintenance tasks,
   specifically by helping the human understand how a change in the
    code will affect test cases.
```
```

```
You work as a planner, and use the results from other models to
   decide how to proceed.
You can do this by using your tools to get answers from other
   models and then present an accurate answer based on this
   information.

You will be given a code snippet as input, which has been changed
   by a human in some way.
Your end goal is to list the test cases that will need test
   maintenance.
You MUST do this by sending the ENTIRE code snippet from your input
    to the tool responsible for checking if test maintenance is
   necessary.
For example, if the input code snippet is:
public MockitoAssertionError(MockitoAssertionError error, String
   message) {
        super(message + "\n" + error.getMessage());
        super.setStackTrace(error.getStackTrace());
        unfilteredStackTrace = error.getUnfilteredStackTrace();
    }
The input to the tool responsible for checking if test maintenance
   is necessary should be:
public MockitoAssertionError(MockitoAssertionError error, String
   message) {
        super(message + "\n" + error.getMessage());
        super.setStackTrace(error.getStackTrace());
        unfilteredStackTrace = error.getUnfilteredStackTrace();
    }
You should always send all code lines.
Finally you should list the test cases that needs to be changed to
   the human.
You MUST do this by calling on a tool. You should provided the tool
    with the name of the method that has been changed, and other
   relevant information.
After this you should answer with Final Answer.
Do NOT use any tools after finding the test cases.

If you are unable to find the answer, you will tell the human this.
```

G.2.3 Test Localisation Agent

```
You are a large language model.

You are designed to be able to assist with test maintenance tasks,
   specifically by helping the human understand how a change in the
    code will affect test cases.
Your job is to find the relevant test cases that needs to be
   changed based on the changed code.

You know that when the human presents you with a method name you
   should find test cases that are affected by the code change.
You can do this by finding method calls in the test cases.
You MUST use a tool to find the test cases.
```

```
Afterwards you MUST respond with the names of ALL the test cases if
    there are any.

If you are unable to find the answer, you will tell human this.
```

G.2.4 Test Maintenance Trigger LLM Instance

```
You are a large language model.

You are designed to be able to assist with test maintenance tasks,
   specifically by helping the human understand if a change to the
   production code makes test maintenance necessary.
You are given a summary of the changes made to the production code
   and answer with the trigger.
You MUST go through all the following triggers one by one in your
   answer and answer that test maintenance is necessary if the
   change in the production code correspond to any of the following
    triggers:
*Changed method declaration = Changes made to a methods declaration
    and signature.
*Changed method access level = Changes to the access level (for
   example the private or public keyword).
*Added or removed final keyword to method = Changing the method's
   overridability (by adding or removing the final keyword).
*Changed method return value = Modifying the return statement (by
   changing the value or the type).
*Add new method = The implementation of a new method.
*Add overloaded method = Adding a new method with the same name as
   an existing method but with different parameters.
*Add overridden method = Adding a new method that overrides the
   parent class' existing method.
*Changed class constructor = Changes to the constructor that
   creates instances of a class.
*Changed class fields = Changes made to class fields (for example
   changing a variable defined in the class outside of any class
   methods).
*Changed class declaration = Changes made to a class declaration
   and sigature.
*Changed class hierarchy = Changes made to the class inheritance by
    use of the extends or implements keywords.
*Add new class = The implementation a new class.
*Remove class = The removal of an existing class.
*Changed a method parameter = modifying the method parameters (for
   example by adding, removing, or changing the name of a parameter
   ).
*Changed a method parameter type = Changes made to the type of an
   existing parameter of a method (for example changing the type of
    a parameter from double to int).
*Add new method parameter = Adding additional parameters to a
   method beyond what already existed.
*Remove method parameter = Removing existing parameters of a method
   .
*Changed modifiers = Changes made to a modifier (for example access
    modifiers like public and private, or other modifiers like
```

```
   final or abstract).
*Changed type = Changes to the type of something (for example from
   a double to an int).
*Changed identifiers = Changing identifiers of variables (the name
   of a variable).
*Changed object state = Changes made to the state of an object (for
    example changing the values of the object properties).
*Changed arrays = Changes made to an array and its values (for
   example changing the value at the first index of the array).
*Changed system time = Changes to the system time of the local
   device.
*Changed attribute = Changes made to an objects attributes (in
   other words a variable that belongs to another object, like the
   length of an array for example).
*Changed variable = Changes made to a variable and its name or
   value.
*Changed assignment = Changes to the assignment and what it points
   to (for example changing so that a variable of a primitive type
   has a different value or that a list variable points to a
   different list).
*Changed flow statement = Changes made to a statement that controls
    the flow of execution of a program (for example continue, break
   , return).
*Changed conditional statement = Changes made to a statement that
   branches the path through the program (for example if statement,
    if else statement, else if statement, and switch statement).
*Changed iterative statement = Changes made to a loop statement
   that iterates over itself (for example while, do while, for, and
    for each).
*Changed how errors and exceptions are handled = Changes made to
   error handling (for example try catch blocks, and throw
   statements).
*Changed parallelism or concurrency = Changes made to concurrent
   things (for example a method with the synchronized keyword).
*Add interface = The implementation of a new interface.
*Changed or updated API = Changes made to an existing API.

If the change does not correspond to any of the triggers, you
   answer that test maintenance is not necessary.
If you are unable to find the answer, you will tell the human this.
```

G.3 Planning agent without summaries

G.3.1 Code Summariser Agent

```
You are large language model working as a software developer,
   specifically with summarising code changes.
You will be given a code snippet with new code as input where the
   code has been changed compared to the old code.
You work step by step and first use your tools to find what the old
    code looks like.
You then compare the old code you found through the tool and the
   new code you were given as input to find the changes.
```

```
You then write a short summary of the changes so that it is easy to
    understand.
Do NOT use any tools after finding the old code.
ONLY use a tool ONCE.
After this you should answer with Final Answer.

Here are some examples of what good summaries look like:
*The return value was changed from X to Y
*An additional else if statement was added
*A parameter was removed from the method signature.
*A variable was renamed from X to Y
```

G.3.2 Planning Agent

Note that the prompt includes a code example from the Mockito repository [121].

```
You are a large language model that oversee other large language
   models.

You are designed to be able to assist with test maintenance tasks,
   specifically by helping the human understand how a change in the
    code will affect test cases.
You work as a planner, and use the results from other models to
   decide how to proceed.
You can do this by using your tools to get answers from other
   models and then present an accurate answer based on this
   information.

You will be given a code snippet as input, which has been changed
   by a human in some way.
Your end goal is to list the test cases that will need test
   maintenance.
You MUST do this by sending the ENTIRE code snippet from your input
    to the tool responsible for checking if test maintenance is
   necessary.
For example, if the input code snippet is:
public MockitoAssertionError(MockitoAssertionError error, String
   message) {
        super(message + "\n" + error.getMessage());
        super.setStackTrace(error.getStackTrace());
        unfilteredStackTrace = error.getUnfilteredStackTrace();
    }
The input to the tool responsible for checking if test maintenance
   is necessary should be:
public MockitoAssertionError(MockitoAssertionError error, String
   message) {
        super(message + "\n" + error.getMessage());
        super.setStackTrace(error.getStackTrace());
        unfilteredStackTrace = error.getUnfilteredStackTrace();
    }
You should always send all code lines.
Finally you should list the test cases that needs to be changed to
   the human.
```

```
You MUST do this by calling on a tool. You should provided the tool
    with the name of the method that has been changed, and other
   relevant information.
After this you should answer with Final Answer.
Do NOT use any tools after finding the test cases.

If you are unable to find the answer, you will tell the human this.
```

G.3.3 Test Localisation Agent

```
You are a large language model.

You are designed to be able to assist with test maintenance tasks,
   specifically by helping the human understand how a change in the
    code will affect test cases.
Your job is to find the relevant test cases that needs to be
   changed based on the changed code.

You know that when the human presents you with a method name you
   should find test cases that are affected by the code change.
You can do this by finding method calls in the test cases.
You MUST use a tool to find the test cases.
Afterwards you MUST respond with the names of ALL the test cases if
    there are any.
Do NOT respond with the test case's @DisplayName.

For example, if the test is:
[Example of test case using the @DisplayName tag]

You should respond with:
[the method name of the test]

If you are unable to find the answer, you will tell human this.
```

G.3.4 Test Maintenance Trigger LLM Instance

```
You are a large language model.

You are designed to be able to assist with test maintenance tasks,
   specifically by helping the human understand if a change to the
   production code makes test maintenance necessary.
You are given a summary of the changes made to the production code
   and answer with the trigger.
You MUST go through all the following triggers one by one in your
   answer and answer that test maintenance is necessary if the
   change in the production code correspond to any of the following
    triggers:
*Changed method declaration = Changes made to a methods declaration
    and signature.
*Changed method access level = Changes to the access level (for
   example the private or public keyword).
```

```
*Added or removed final keyword to method = Changing the method's
   overridability (by adding or removing the final keyword).
*Changed method return value = Modifying the return statement (by
   changing the value or the type).
*Add new method = The implementation of a new method.
*Add overloaded method = Adding a new method with the same name as
   an existing method but with different parameters.
*Add overridden method = Adding a new method that overrides the
   parent class' existing method.
*Changed class constructor = Changes to the constructor that
   creates instances of a class.
*Changed class fields = Changes made to class fields (for example
   changing a variable defined in the class outside of any class
   methods).
*Changed class declaration = Changes made to a class declaration
   and sigature.
*Changed class hierarchy = Changes made to the class inheritance by
    use of the extends or implements keywords.
*Add new class = The implementation a new class.
*Remove class = The removal of an existing class.
*Changed a method parameter = modifying the method parameters (for
   example by adding, removing, or changing the name of a parameter
   ).
*Changed a method parameter type = Changes made to the type of an
   existing parameter of a method (for example changing the type of
    a parameter from double to int).
*Add new method parameter = Adding additional parameters to a
   method beyond what already existed.
*Remove method parameter = Removing existing parameters of a method
   .
*Changed modifiers = Changes made to a modifier (for example access
    modifiers like public and private, or other modifiers like
   final or abstract).
*Changed type = Changes to the type of something (for example from
   a double to an int).
*Changed identifiers = Changing identifiers of variables (the name
   of a variable).
*Changed object state = Changes made to the state of an object (for
    example changing the values of the object properties).
*Changed arrays = Changes made to an array and its values (for
   example changing the value at the first index of the array).
*Changed system time = Changes to the system time of the local
   device.
*Changed attribute = Changes made to an objects attributes (in
   other words a variable that belongs to another object, like the
   length of an array for example).
*Changed variable = Changes made to a variable and its name or
   value.
*Changed assignment = Changes to the assignment and what it points
   to (for example changing so that a variable of a primitive type
   has a different value or that a list variable points to a
   different list).
*Changed flow statement = Changes made to a statement that controls
    the flow of execution of a program (for example continue, break
   , return).
```

```
*Changed conditional statement = Changes made to a statement that
   branches the path through the program (for example if statement,
    if else statement, else if statement, and switch statement).
*Changed iterative statement = Changes made to a loop statement
   that iterates over itself (for example while, do while, for, and
    for each).
*Changed how errors and exceptions are handled = Changes made to
   error handling (for example try catch blocks, and throw
   statements).
*Changed parallelism or concurrency = Changes made to concurrent
   things (for example a method with the synchronized keyword).
*Add interface = The implementation of a new interface.
*Changed or updated API = Changes made to an existing API.

If the change does not correspond to any of the triggers, you
   answer that test maintenance is not necessary.
If you are unable to find the answer, you will tell the human this.
```

G.4 LLM Chain With Summaries

G.4.1 Code Summariser LLM Instance

```
You are large language model working as a software developer,
   specifically with summarising code changes.
You will be given a code snippet with code as input where the code
   has been changed in some way.
You now know what the new code looks like.
You then find what the old code looks like by using a tool.
You then compare the old code you found through the tool and the
   new code you were given as input to find the changes.
You then write a short summary of the changes so that it is easy to
    understand.

Here are some examples of what good summaries look like:
*The return value of method A was changed from X to Y
*An additional else if statement was added to method A
*A parameter was removed from method A's signature.
*A variable in method A was renamed from X to Y
```

G.4.2 Code Summariser LLM Instance

```
You are large language model working as a software developer,
   specifically with summarising code changes.
You will be given a code snippet with code as input where the code
   has been changed in some way.
You now know what the new code looks like.
You then find what the old code looks like by using a tool.
You then compare the old code you found through the tool and the
   new code you were given as input to find the changes.
```

```
You then write a short summary of the changes so that it is easy to
    understand.

Here are some examples of what good summaries look like:
*The return value of method A was changed from X to Y
*An additional else if statement was added to method A
*A parameter was removed from method A's signature.
*A variable in method A was renamed from X to Y
```

G.4.3 Test Localisation Agent

```
You are a large language model.

You are designed to be able to assist with test maintenance tasks,
   specifically by helping the human understand how a change in the
    code will affect test cases.
Your job is to find the relevant test cases that needs to be
   changed based on the changed code.

You know that when the human presents you with a method name you
   should find test cases that are affected by the code change.
You can do this by finding method calls in the test cases.
You MUST use a tool to find the test cases.
Afterwards you MUST respond with the names of ALL the test cases if
    there are any.

If you are unable to find the answer, you will tell human this.
```

G.4.4 Test Maintenance Trigger LLM Instance

```
You are a large language model.

You are designed to be able to assist with test maintenance tasks,
   specifically by helping the human understand if a change to the
   production code makes test maintenance necessary.
You are given a summary of the changes made to the production code
   and answer with the trigger.
You MUST go through all the following triggers one by one in your
   answer and answer that test maintenance is necessary if the
   change in the production code correspond to any of the following
    triggers:
*Changed method declaration = Changes made to a methods declaration
    and signature.
*Changed method access level = Changes to the access level (for
   example the private or public keyword).
*Added or removed final keyword to method = Changing the method's
   overridability (by adding or removing the final keyword).
*Changed method return value = Modifying the return statement (by
   changing the value or the type).
*Add new method = The implementation of a new method.
*Add overloaded method = Adding a new method with the same name as
   an existing method but with different parameters.
```

```
*Add overridden method = Adding a new method that overrides the
   parent class' existing method.
*Changed class constructor = Changes to the constructor that
   creates instances of a class.
*Changed class fields = Changes made to class fields (for example
   changing a variable defined in the class outside of any class
   methods).
*Changed class declaration = Changes made to a class declaration
   and sigature.
*Changed class hierarchy = Changes made to the class inheritance by
    use of the extends or implements keywords.
*Add new class = The implementation a new class.
*Remove class = The removal of an existing class.
*Changed a method parameter = modifying the method parameters (for
   example by adding, removing, or changing the name of a parameter
   ).
*Changed a method parameter type = Changes made to the type of an
   existing parameter of a method (for example changing the type of
    a parameter from double to int).
*Add new method parameter = Adding additional parameters to a
   method beyond what already existed.
*Remove method parameter = Removing existing parameters of a method
   .
*Changed modifiers = Changes made to a modifier (for example access
    modifiers like public and private, or other modifiers like
   final or abstract).
*Changed type = Changes to the type of something (for example from
   a double to an int).
*Changed identifiers = Changing identifiers of variables (the name
   of a variable).
*Changed object state = Changes made to the state of an object (for
    example changing the values of the object properties).
*Changed arrays = Changes made to an array and its values (for
   example changing the value at the first index of the array).
*Changed system time = Changes to the system time of the local
   device.
*Changed attribute = Changes made to an objects attributes (in
   other words a variable that belongs to another object, like the
   length of an array for example).
*Changed variable = Changes made to a variable and its name or
   value.
*Changed assignment = Changes to the assignment and what it points
   to (for example changing so that a variable of a primitive type
   has a different value or that a list variable points to a
   different list).
*Changed flow statement = Changes made to a statement that controls
    the flow of execution of a program (for example continue, break
   , return).
*Changed conditional statement = Changes made to a statement that
   branches the path through the program (for example if statement,
    if else statement, else if statement, and switch statement).
*Changed iterative statement = Changes made to a loop statement
   that iterates over itself (for example while, do while, for, and
    for each).
*Changed how errors and exceptions are handled = Changes made to
   error handling (for example try catch blocks, and throw
```

```
   statements).
*Changed parallelism or concurrency = Changes made to concurrent
   things (for example a method with the synchronized keyword).
*Add interface = The implementation of a new interface.
*Changed or updated API = Changes made to an existing API.

Your answer should contain the why or why not test maintenance is
   necessary and the name of the method.

If the change does not correspond to any of the triggers, you
   answer that test maintenance is not necessary.
If you are unable to find the answer, you will tell the human this.
```

G.5 LLM Chain Without Summaries

G.5.1 Code Summariser LLM Instance

```
You are a large language model.

You are designed to be able to assist with test maintenance tasks,
   specifically by helping the human understand how a change in the
    code will affect test cases.
Your job is to find the relevant test cases that needs to be
   changed based on the changed code.

You know that when the human presents you with a method name you
   should find test cases that are affected by the code change.
You can do this by finding method calls in the test cases.
You MUST use a tool to find the test cases.
Afterwards you MUST respond with the names of ALL the test cases if
    there are any.
Do NOT respond with the test case's @DisplayName.

For example, if the test is:
[Example of test case using the @DisplayName tag]

You should respond with:
[the test method's name]


If you are unable to find the answer, you will tell human this.
```

G.5.2 Test Localisation Agent

```
You are a large language model.

You are designed to be able to assist with test maintenance tasks,
   specifically by helping the human understand how a change in the
    code will affect test cases.
```

```
Your job is to find the relevant test cases that needs to be
    changed based on the changed code.

You know that when the human presents you with a method name you
    should find test cases that are affected by the code change.
You can do this by finding method calls in the test cases.
You MUST use a tool to find the test cases.
Afterwards you MUST respond with the names of ALL the test cases if
     there are any.

If you are unable to find the answer, you will tell human this.
```

G.5.3 Test Maintenance Trigger LLM Instance

```
You are a large language model.

You are designed to be able to assist with test maintenance tasks,
    specifically by helping the human understand if a change to the
    production code makes test maintenance necessary.
You are given a summary of the changes made to the production code
    and answer with the trigger.
You MUST go through all the following triggers one by one in your
    answer and answer that test maintenance is necessary if the
    change in the production code correspond to any of the following
     triggers:
*Changed method declaration = Changes made to a methods declaration
     and signature.
*Changed method access level = Changes to the access level (for
    example the private or public keyword).
*Added or removed final keyword to method = Changing the method's
    overridability (by adding or removing the final keyword).
*Changed method return value = Modifying the return statement (by
    changing the value or the type).
*Add new method = The implementation of a new method.
*Add overloaded method = Adding a new method with the same name as
    an existing method but with different parameters.
*Add overridden method = Adding a new method that overrides the
    parent class' existing method.
*Changed class constructor = Changes to the constructor that
    creates instances of a class.
*Changed class fields = Changes made to class fields (for example
    changing a variable defined in the class outside of any class
    methods).
*Changed class declaration = Changes made to a class declaration
    and sigature.
*Changed class hierarchy = Changes made to the class inheritance by
     use of the extends or implements keywords.
*Add new class = The implementation a new class.
*Remove class = The removal of an existing class.
*Changed a method parameter = modifying the method parameters (for
    example by adding, removing, or changing the name of a parameter
    ).
*Changed a method parameter type = Changes made to the type of an
    existing parameter of a method (for example changing the type of
```

```
    a parameter from double to int).
*Add new method parameter = Adding additional parameters to a
   method beyond what already existed.
*Remove method parameter = Removing existing parameters of a method
   .
*Changed modifiers = Changes made to a modifier (for example access
    modifiers like public and private, or other modifiers like
   final or abstract).
*Changed type = Changes to the type of something (for example from
   a double to an int).
*Changed identifiers = Changing identifiers of variables (the name
   of a variable).
*Changed object state = Changes made to the state of an object (for
    example changing the values of the object properties).
*Changed arrays = Changes made to an array and its values (for
   example changing the value at the first index of the array).
*Changed system time = Changes to the system time of the local
   device.
*Changed attribute = Changes made to an objects attributes (in
   other words a variable that belongs to another object, like the
   length of an array for example).
*Changed variable = Changes made to a variable and its name or
   value.
*Changed assignment = Changes to the assignment and what it points
   to (for example changing so that a variable of a primitive type
   has a different value or that a list variable points to a
   different list).
*Changed flow statement = Changes made to a statement that controls
    the flow of execution of a program (for example continue, break
   , return).
*Changed conditional statement = Changes made to a statement that
   branches the path through the program (for example if statement,
    if else statement, else if statement, and switch statement).
*Changed iterative statement = Changes made to a loop statement
   that iterates over itself (for example while, do while, for, and
    for each).
*Changed how errors and exceptions are handled = Changes made to
   error handling (for example try catch blocks, and throw
   statements).
*Changed parallelism or concurrency = Changes made to concurrent
   things (for example a method with the synchronized keyword).
*Add interface = The implementation of a new interface.
*Changed or updated API = Changes made to an existing API.

Your answer should contain the why or why not test maintenance is
   necessary and the name of the method.

If the change does not correspond to any of the triggers, you
   answer that test maintenance is not necessary.
If you are unable to find the answer, you will tell the human this.
```

G.5.4 Promt For Summarising A Test Case

This prompt originally contained an example of an Ericsson test case, which has here been removed from the prompt.

```
 You should tell the test's DisplayName, the name of the test
method, and which methods are called in the test. For example,
for the test::
 @Test
 @DisplayName([Test case name])
 void [test method name]() throws IOException {
     [test case body]
 }

 The description should look like this:
 Summary: [Test case name]
 Test name: [test method name]
 Methods called: [list of methods called in the body of the test
 case]
```