An Exploration of Agentic Information Fusion for Test Maintenance Prediction

Jingxiong Liu
liujing@chalmers.se
Chalmers University of Technology and University of Gothenburg, Ericsson AB
Gothenburg, Sweden

Nasser Mohammadiha
Ericsson AB
Gothenburg, Sweden

Gregory Gay
greg@greggay.com
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden

Abstract

Test maintenance is a critical, yet costly, activity—particularly as codebases rapidly evolve. To assist, we present MAST, a multi-agent framework that predicts which test cases require maintenance following changes to the production code. This identification task is necessary as a precondition to any subsequent maintenance activities, but remains challenging due to the complex relationships between production and test code. MAST advances the state-of-the-art by integrating multiple analyses—including static, lexical, and semantic analyses—through an intelligent fusion and post-check procedure and by focusing on a realistic use and evaluation setting—i.e., standardized input formats, repository-level analyses, and the ability to infer relations between test and production artifacts rather than assuming a pre-existing mapping.

We evaluated MAST on 21 industrial Java repositories from Ericsson AB, considering situations where test maintenance both was and was not required in the ground truth. MAST yielded superior precision to a state-of-the-art baseline—resulting in a higher accuracy, F1, and F2 score—with only some loss in recall. Our ablation study demonstrates the value of each analysis in producing the final recommendations. MAST illustrates the potential of multi-agent systems that can fuse multiple information sources when performing software testing tasks.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; **Software maintenance tools**; • **Computing methodologies** → **Machine learning**.

Keywords

Software Testing, Test Maintenance, Large Language Models, Multi-Agent Systems, Program Analysis

ACM Reference Format:
Jingxiong Liu, Nasser Mohammadiha, and Gregory Gay. 2026. An Exploration of Agentic Information Fusion for Test Maintenance Prediction. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE'26).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

ASE'26, Munich, Germany

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software testing is a crucial, but expensive, stage in the quality assurance process [1]. Although there is an initial cost associated with creating test cases, much of the cost of testing is imposed by the ongoing need for *test maintenance* [32], i.e., the adaptation of the test suite as the project evolves. In most cases, such activities takes place after changes are made to the source code that render the existing test suite obsolete—i.e., some tests may no longer be necessary, some may need adaptations to match the changed behavior of the component-under-test, or new tests may be needed [1]. Test maintenance can also be conducted to improve the quality or efficiency of the test suite—for example, pruning redundant tests to reduce the execution time of the suite [32].

In current practice, test maintenance requires long-term commitment of significant developer effort [32, 38]. Automation could assist in reducing the cost and improving the quality or reliability of the test maintenance process [14, 38]. In particular, in this study, we focus on how a *multi-agent LLM system*—where a task is performed by a team of cooperating, semi-autonomous agents [12, 21]—can assist in the test maintenance process. In a multi-agent LLM system, an overall task is split into sub-tasks that are performed by individual agents, where each agent pairs an LLM with instructions for the sub-task and access to tools, external data sources (e.g., via retrieval-augmented generation), and memory structures [12, 21].

Figure 1: Input and output of the MAST framework.

Regardless of the purpose of a test maintenance activity, one of the first steps in the process is to *identify which existing test cases need to be modified or deleted*—a task we refer to as "test localization". In this study, we propose MAST (**M**ulti-**A**gent Multi-**S**ource **T**est Maintenance), a multi-agent framework that integrates multiple complementary analysis techniques to perform test localization following a change to the project source code.

As shown in Figure 1, MAST takes as input a `git diff` showing a production code change, then outputs a list of test cases to modify or delete. For each test case, an explanation of why a change is needed

is also provided. A tester could then go on to use this information to perform their planned maintenance activities. In the future, MAST could also be expanded to directly perform such actions on the identified test cases. In terms of scoping, we focus on code-based test cases at the unit, integration, and system-level. We have focused on Java-based projects in our evaluation, however, MAST is not conceptually tied to a particular language or testing framework.

We have developed and evaluated MAST in collaboration with Ericsson AB, a major Swedish telecommunications company. MAST offers the following novel contributions over previous work (e.g., [6, 13]), including our own prior work [21]:

Multi-Source Analysis: MAST yields predictions by fusing the results of three complementary analyses of the production code change and the test cases. First, *static analysis*—specifically, call graph analysis—is used to identify which test cases invoke the changed production code. Second, *lexical analysis* is used to tokenize and analyze the similarity of production and test code. Third, MAST performs a *semantic analysis* where natural language summaries of the production code change and test cases are compared.

The results of these analyses are then fused by an LLM agent, then the fused list is subjected to an additional *post-check analysis*, where an agent compares the code of each potential candidate test to the production code change to filter false positives. Collectively, we hypothesize that these four analyses and the fusion procedure will yield more accurate results than previous studies—e.g., our prior work applied semantic analysis alone [21]. While LLM-based analysis fusion has been applied to other aspects of software development (e.g., [8, 16, 20]), it has not been applied in test maintenance.

Realistic Application Setting: MAST has been designed and evaluated in an industrial context, with the aim of producing a tool that could be applied in practice. Unlike, for example, [13], MAST takes a `git diff` as input—a standard method of summarizing code changes—enabling direct invocation following a commit. While [13] was limited to analyzing changes to a single production method, MAST can process repository-level changes. Previous approaches, e.g., [6] assumed a precise, pre-existing mapping of production and test artifacts, while MAST infers that relationship. Finally, we have examined the performance of MAST in both positive (i.e., in the ground truth, test cases were updated by developers following a production code change) and negative (i.e., production code changes did not lead to test maintenance) situations, ensuring that our framework can be used in situations where we do not already know whether maintenance is required.

We evaluated MAST on 21 Java repositories from five teams at Ericsson, where we analyzed the performance of MAST in comparison to a state-of-the-art baseline derived from our previous research [21]. We also conducted an ablation study where we examined the impact of the individual analyses and the fusion on the overall performance of MAST. We observed that:

- In cases where test maintenance was required, MAST achieves a much higher precision than the baseline, leading to improved accuracy, F1, and F2 scores. There is a trade-off between precision and recall, and the baseline achieves higher recall than MAST.
- In cases where test maintenance was not required, MAST yields far fewer false positives than the baseline, resulting in a higher accuracy.
- Each information source offers different maintenance suggestions. The fusion agent effectively merges the sources, and the post-check prunes many false positives from the merged list (albeit at some cost in recall). The superior results of MAST in precision, accuracy, and F1 score over the examined sub-workflows shows that each agent contributes to the overall effectiveness of the framework. However, if recall is highly prioritized over prevision, a tester may also wish to inspect the fusion results prior to the post-check.

MAST demonstrates the potential of multi-agent LLM systems that can fuse multiple information sources when performing software testing tasks, such as test localization. Our findings advance our understanding of how multi-agent LLM systems can be applied within test maintenance and MAST represents a starting point for a future framework that can perform diverse maintenance tasks on the identified test cases. We make MAST available for researchers and practitioners to use and extend (see Section 8).

2 Background, Scoping, and Related Work

Software Testing: Software testing is an activity performed to ensure that software delivers correct functionality and meets quality goals (e.g., performance) [2]. During testing, a *test suite*—one or more *test cases*—is executed against the system-under-test. Each test case performs input actions, then compares the resulting program behavior against expectations (called *test oracles*).

Testing can take place at multiple levels of granularity. While our work is not explicitly tied to a specific level, the projects used in the evaluation contain *unit tests*—i.e., tests targeting an individual class—and *integration tests*—i.e., tests targeting a set of collaborating units. These tests are written in the JUnit framework for Java[1].

Test Maintenance: Test maintenance is the process of updating an existing test suite [1, 21]. It often corresponds to a change in the production code, leading to addition of new tests and removal or adaptation of obsolete tests [1, 21]. Alternatively, maintenance may occur to improve efficiency (e.g., removal of redundant tests), due to changes in the technology stack or requirements, in response to discovery of a fault, or as a result of a deliberate decision to improve coverage or other quality indicators [21].

Test maintenance is understood to be important [17] and can account for a significant proportion of testing effort over the lifespan of a project [1, 32]. However, in practice, few open-source projects implement tests following patterns that promote maintainability [10] and the area has received comparatively little research attention [31]—particularly in industrial settings [14].

Large Language Models and Multi-Agent LLM Systems: LLMs are a form of machine learning model designed to produce open-ended text-based responses to instructions (called *prompts*) [4]. LLMs are trained to infer the semantic meaning of a prompt and use that meaning to produce an appropriate response. LLMs are suited for language analysis and transformation tasks such as translation, summarization, and decision support [43]. Because of their ability

[1] https://junit.org/

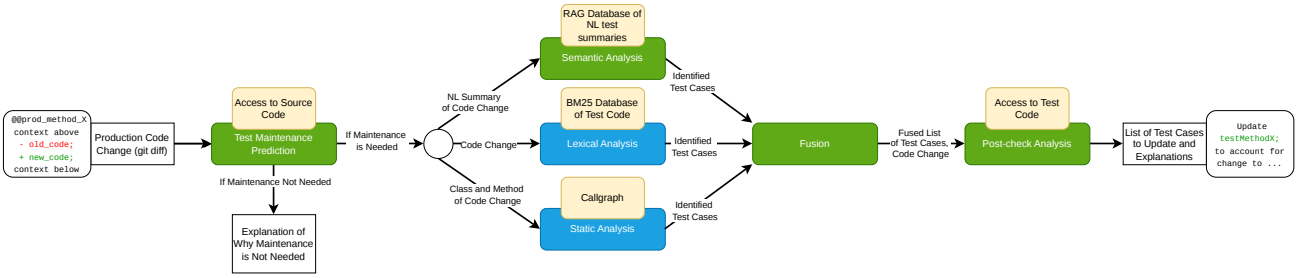

Figure 2: Overview of MAST. Boxes colored green represent agents, while the ones colored blue are scripted components.

to understand and output both natural language and code, LLMs are effective at software development and testing tasks [37].

An *LLM agent* is a software component coupling an LLM and a prompt with tool access—including access to external data sources—and memory capabilities [7, 12]. These mechanisms allow LLM agents to reason, plan, and perceive or interact with an environment [7]. For example, an LLM agent may have access to a database of relevant context (e.g., source code) through Retrieval-Augmented Generation (RAG) [19]. This would allow it to reason about the code, make changes to it, and push those changes through access to the version control system. Multiple agents can work together—*a multi-agent system*—each given different roles or sub-tasks [12]. The collaboration between agents (a *workflow*) can be pre-planned by developers or agent-determined at execution time.

In terms of scoping, our multi-agent system consists only of pre-planned agent collaborations, with the workflow defined as a directed graph where each node represents an LLM agent or a *scripted component*—functionality performed through traditional source code, rather than through invocation of an LLM. To simplify the presentation, we refer to all LLM-based nodes as "agents", even if that node does not include tool or external data access.

Related Work: Automated test maintenance has been under-studied in the research literature [13, 21]. However, test localization and obsolete test code repair have both been a focus of prior work. Traditional approaches are pattern-based, e.g., based on linking tests with production code purely through naming conventions [35] or method signatures and return values [26–28]. Recently, there has been interest in applying LLMs to both localization and repair.

Liu et al. [22], Tu et al. [34], and Xu et al. [40] have developed multi-agent or LLM-based frameworks for test repair. All three studies abstract the test localization problem, assuming that it is already known which tests require maintenance. Liu and Tu both employ static analyses to enhance their frameworks. Liu et al. use static analyses to infer context on the changed code class, how it is invoked, and the environment that the test and code are executed in [22]. Tu et al. use a call graph to trace non-local dependencies [34]. Xu et al. use BM-25-based lexical analysis to retrieve similar code-test evolution pairs to use in formulating the test repair [40].

Of particular relevance are two LLM-related approaches that perform test localization. Hu et al. applied a transformer model to both localize and update tests [13]. They do not separate these tasks into individual predictions, but perform a single repair task. If the final test is identical to the input, they assume that the test has been determined to still be relevant. Their approach is limited to production changes contained within a single method.

Recently, Chi et al. proposed a multi-agent framework for test localization [6]. Their approach is conceptually similar to our "post-check" agent—comparing the test code to the production code change. However, their approach also employs a RAG-based memory mechanism containing natural language summaries of reasons why previous test updates took place. This is implemented in a similar manner to our semantic analysis, but focusing on different contextual information. A limitation of their approach is that it assumes that the set of all tests possibly affected by a production code change is already known, while our approach infers this relationship, and their evaluation dataset has a one-to-one mapping between production and test code changes. Neither approach considers commits where no test maintenance took place at all following a production change, only situations where particular individual test cases were not updated.

While LLM-based information fusion has been studied in other aspects of the development process—e.g., vulnerability detection [8], test report clustering [20], and documentation evaluation [16]—we are the first to apply this concept to test maintenance.

3 MAST Framework

MAST (**M**ulti-**A**gent Multi-**S**ource **T**est Maintenance) is a multi-agent framework that integrates multiple analysis techniques to predict which test cases must be updated following a change to the production code. An overview of MAST is shown in Figure 2, where the green boxes represent LLM agents and blue boxes represent scripted components. MAST executes the following workflow:

- The **test maintenance prediction agent** (Section 3.2) assesses the production code change and predicts whether test maintenance is needed.
- Three analyses are executed in parallel—**semantic analysis**, **lexical analysis**, and **static analysis** (Section 3.3). Each yields a list of candidate test cases.
- The **fusion agent** (Section 3.4) merges these lists into a single, filtered, candidate list.
- Then, the **post-check analysis agent** (Section 3.5) compares the code of each test on this list to the production code change, filtering those without clear relevance.

This process yields a list of tests predicted to need maintenance, along with rationale for each inclusion.

3.1 Input Format

MAST uses the standard `git diff` format [25] as input. This format shows code changes as well as surrounding unchanged lines for context. To offer additional context to MAST, we include nine code lines above and below each change. A single commit may consist of multiple `git diff` chunks. In such cases, we input one chunk at a time to MAST. An example `git diff` is shown at Listing 1.

Listing 1: Example git diff chunk.

```
@@ −1446,20 +1446,20 @@ public final class AsciiString implements
        CharSequence, Comparable<CharSequence>;
          return false;
      }
−     for (int i = 0, j = 0; i &lt; a.length(); ++i, ++j) {
−       if (!equalsIgnoreCase(a.charAt(i),  b.charAt(j))) {
+     for (int i = 0; i &lt; a.length(); ++i) {
+       if (!equalsIgnoreCase(a.charAt(i),  b.charAt(i))) {
          return false;
        }
```

3.2 Test Maintenance Prediction Agent

The test maintenance prediction agent uses an LLM to analyze the `git diff` chunk, extracting a natural language summary of the production code change and using the diff and summary to decide whether the change requires test maintenance. The agent will then offer, as output, its assessment along with the rationale and the code change summary. An example is shown in Listing 2.

If this agent decides that maintenance is not needed, then it will issue its output and conclude the execution of MAST. If it decides that maintenance is needed, the agent will also analyze the file where the diff chunk is situated and extract the method and class names from that file. It will then pass the rationale, change summary, and extracted method and class names to the analyses.

Listing 2: Example maintenance prediction output.

```
maintenance analysis is: {'needs_update': True,
'maintenance_reason': 'The isAndroid() method implementation was changed from
        using a static variable IS_ANDROID to calling PlatformDependent0.
        isAndroid(). This change modifies the behavior and dependencies of the
        method, potentially affecting tests that mock or rely on the previous
        implementation.',
'code_summary': 'File: PlatformDependent.java − Modified the isAndroid() method
        to delegate to PlatformDependent0.isAndroid() instead of returning a static
        IS_ANDROID variable'}
```

3.3 Analysis Nodes

3.3.1 Semantic Analysis Agent. The intention of the semantic analysis is to analyze the underlying intent behind potentially relevant test cases and how that intent relates to the code change—i.e., what purpose does the test serve and what methods from the production code does it invoke? Semantic analysis has been proven to be powerful in solving software engineering problems (e.g., [39]), especially when coupled with the inference capabilities of LLMs [21, 42].

This agent uses an LLM equipped with RAG to retrieve natural language summaries of test cases from an embedding database (based on the `bge-m3` model [5]). These summaries are generated by an LLM for each test case in the suite, based on a commit-id. This database can be generated in advance or when MAST is invoked. An example of a test case and its summary can be seen in Listing 3. The agent retrieves the test summaries with the highest similarity to the natural language summary of the code change—i.e., those above a certain threshold using FAISS [15]. In this case, we retrieve test cases with similarity score above 0.50, with the values chosen through experimentation. We then perform a second filtering step, where we retain a subset with a similarity score $> mean + (0.25 * std.dev.)$, where the mean and standard deviation refer to the set being filtered.

Listing 3: Example of a test case and its summary.

```
  @Test
  public void testGetBytesStringBuilder() {
    final StringBuilder b = new StringBuilder();
    for (int i = 0; i < 1 << 16; ++i) {
      b.append("eaÃã");
    }
    final String bString = b.toString();
    final Charset[] charsets = CharsetUtil.values();
    for (int i = 0; i < charsets.length; ++i) {
      final Charset charset = charsets[i];
      byte[] expected = bString.getBytes(charset);
      byte[] actual = new AsciiString(b, charset).toByteArray();
      assertArrayEquals("failure for " + charset, expected, actual);
    }
  }

Summary: Get bytes from StringBuilder with various charsets
Test name: testGetBytesStringBuilder()
Methods called: StringBuilder.append, StringBuilder.toString, CharsetUtil.values,
        String.getBytes, AsciiString constructor, AsciiString.toByteArray,
        assertArrayEquals
```

3.3.2 Lexical Analysis. Lexical analysis—an analysis based on parsing text into token vectors—is a classic method of similarity assessment [9]. To perform this analysis, we parse the raw source code of the test cases after a pre-processing stage where we convert all code to lower case, remove punctuation, and separate the keyword "test" from other words.

We use the BM25 ranking algorithm [29] to identify tests whose code is syntactically relevant to the code change. Due to its speed and simplicity, BM25 has also been applied for similar purposes in other LLM-based frameworks (e.g., [36, 40, 41]). After parsing, the test code is stored in a BM25 index database. Then, all test cases that fall above a chosen similarity threshold will be retrieved from the index database. We retrieve up to five test cases with a similarity score above 50.00, values chosen through experimentation.

3.3.3 Static Analysis. Static analysis encompasses a broad range of techniques that gather data from processing the production and test code [23]. In particular, in this study, we use call graph analysis [30] to suggest test cases that are relevant to the production code change. Past research has shown that call graphs can reduce LLM hallucinations [11, 24, 34].

Listing 4: Example of the mapping of production methods to test cases from the call graph.

```
"AsciiString.toByteArray": [
  "common/src/test/java/io/netty/util/AsciiStringCharacterTest.java::
       AsciiStringCharacterTest.testGetBytesStringBuilder",
  "common/src/test/java/io/netty/util/AsciiStringCharacterTest.java::
       AsciiStringCharacterTest.testGetBytesString",
  "common/src/test/java/io/netty/util/AsciiStringCharacterTest.java::
       AsciiStringCharacterTest.testGetBytesAsciiString"
]
```

This static analysis component uses the `tree-sitter` tool [3] to parse the repository, and creates a call graph where each production

method is mapped to all test cases that directly invoke that method. An example is shown in Listing 4. Based on the methods extracted by the test maintenance prediction agent, this component extracts the potentially-impacted tests from the call graph.

3.4 Fusion Agent

The fusion agent employs an LLM to intelligently merge the test cases predicted by each of the three analysis nodes into a single list. As noted previously, LLM-based information fusion has shown potential as a flexible alternative to traditional heuristic-based fusion methods [8, 16, 20]. The prompt for this agent explains each information source, but does not suggest which sources may be more or less trustworthy to avoid biasing the model's predictions. The agent then returns a single list of test cases along with rationale to explain how it conducted the fusion process. An example of the rationale is shown in Listing 5.

Listing 5: Example of the fusion rationale.

```
The semantic analysis (Source 1) provides semantically relevant tests related to
version parsing, security manager behavior, and appendable character sequence
operations. From this list, 'testMajorVersion' and
'testMajorVersionFromJavaSpecificationVersion' are further corroborated by
Source 3 (lexical analysis), indicating lexical relevance as well. The append–related
tests ('testAppendStringWithExpand', 'testSimpleAppendWithExpand',
'testAppendString', 'testSimpleAppend') are also part of the semantic analysis
results and fall under the same functional category, suggesting they exercise core
logic likely impacted by changes in string or sequence handling. Source 2 (static
analysis) does not provide additional tests beyond what's already included in
Source 1, so it doesn't expand the result set. 'testIsZero' was only found via
semantic analysis but lacks support from other sources and isn't clearly tied
to the primary focus areas identified. Therefore, we select the six tests that
show strong semantic and/or lexical relevance.
```

3.5 Post-Check Analysis Agent

The post-check analysis agent receives the fused list of test cases, then it extracts the code for each and compares the code to the production code change to assess the relevance of the suggested test. This agent generates a final, refined list of test cases. For each, it also provides rationale for its inclusion. An example of this rationale is shown in Listing 6.

Listing 6: Example of post-check decision and reasoning.

```
{'Update_or_Not': 'False',
'post_check_reason': 'The test `testMajorVersionFromJavaSpecificationVersion` is
        designed to test the behavior of `PlatformDependent.
        majorVersionFromJavaSpecificationVersion()` under a custom `
        SecurityManager` that denies access to the `java.specification.version`
        system property. However, in the provided git diff, the methods `
        javaVersion0`, `majorVersionFromJavaSpecificationVersion`, and `
        majorVersion` have been removed from the `PlatformDependent` class.
        This means the method being tested no longer exists in the codebase. The
        test is no longer relevant and should not be retained without updates.'}
```

3.6 Implementation Details

MAST is implemented using LangGraph [18], a commonly-used open-source framework for creating multi-agent LLM systems. LangGraph represents the workflow of the system as a directed graph, where each node can be an agent or a traditional scripted component. The output of each agent—i.e., the current state—is stored in a common memory structure that can be accessed by all agents, like fetching information from a public information bus. The LangGraph architecture is highly flexible and scalable, which means that practitioners and researchers could add more agents to our existing framework—e.g., adding analyses or extending MAST to update the test code.

All agents that employ an LLM use the Qwen3-Coder-480B-A35B-Instruct-FP8 model [33] model, deployed on Ericsson's internal computing infrastructure. Ericsson must approve models for internal use, and this model was the best available at the time that we implemented MAST. This model is generally effective at coding tasks (e.g., achieving a 44% resolved rate on SWE-rebench[2]).

4 Methods

We have performed an evaluation of MAST on datasets curated from 21 Java repositories at Ericsson AB containing unit and integration test cases. Specifically, in this evaluation, we address the following research questions:

Research Questions

- **RQ1:** What is the performance of MAST on *positive* cases (i.e., where test maintenance was required in the ground truth), compared to the baseline approach?
- **RQ2:** What is the performance of MAST on *negative* cases (i.e., where test maintenance was not required), compared to the baseline approach?
- **RQ3:** How do the individual components of MAST contribute to the performance of the overall framework?

We split our analysis of each dataset into *positive* cases (**RQ1**), where a production code change is followed by subsequent test maintenance, and *negative* cases (**RQ2**), where a production code change did not require test maintenance. This is because a developer may not know in advance whether maintenance will be required, and MAST should yield acceptable performance in both situations. In **RQ3**, we conducted an ablation study where we compare the performance of different sub-workflows (individual analyses, fusion, post-check) within MAST to understand how they contribute to the overall performance of the framework.

4.1 Datasets

We have extracted datasets for this evaluation from 21 Java repositories from five teams at Ericsson AB. A description and metadata for each repository is shown in Table 1. The repositories are sorted by the average size of the test suite across the commits.

To obtain the dataset for each repository, we extracted the commit history until Feb. 24th, 2026 of the main branch. To identify positive commits (**RQ1,3**), we filtered each commit to identify those where test cases were updated. Our aim was to identify cases where test updates are related to a meaningful change to the production code. Therefore, to ensure the quality of the identified commits, we then manually removed all commits where the test changes are unrelated to the source code change or are only superficial changes. Specifically, we removed commits where test changes were only renaming of test cases, changes due to linting requirements, changes imposed by a change in the test support code (rather than in the

[2]https://swe-rebench.com/

Table 1: Overview of projects used to evaluate MAST, including the name, number of positive and negative commits, average test suite size across all commits, and a description. Repository names are anonymized to preserve confidentiality.

Project	Positive Commits	Negative Commits	Avg. Suite Size	Description
*-dataexport-test-stability	2	0	2.00	Support for stability testing of the data export tool.
*-extractor	9	6	6.10	Data extraction utility.
*-dataexport-test-e2e	9	2	8.40	Support for end-to-end testing of the data export tool.
*-extractor-2	13	4	10.00	Data extraction utility.
*-service-authentication	8	5	12.70	Authentication service.
*-parser	13	10	13.20	Data parsing tool.
-datastream--scanner	14	10	13.40	Datastream scanner that scans and moves files and sends events.
*-lib-common	6	10	15.30	Library of common utility functions.
-datastream-dump--extractor	3	2	16.00	Extracts data dumps.
*-datastream-dump-cleaner	16	10	19.30	File system cleanup utility.
*-datastream-archive	16	10	40.40	Tool that archives data streams.
*-service-dataexport-download	8	5	42.60	Download service used to export data.
archiver	54	10	49.80	Archiver service.
*-datastream-dump-scanner	78	10	49.90	File system scanner that detects new files and publishes information about them.
*-service-dataexport-executor	31	10	67.90	Provides an API for a UI to manage data export execution.
*-decrypter	8	6	87.30	Decrypting utility.
*-datastream-common	7	10	93.50	A common library for all datastream-based projects.
common	5	2	97.20	Common library for a data ingestion service.
*-datastream-dump-common	72	10	103.60	A common libray for all the datastream dump projects.
*-datastream-dump-archiver	123	10	114.00	Archiver component that reacts to messages about new files available on the file system.
*-service-dataexport-handler	55	10	118.00	Data export service.

production code), or where the directory structure of the project was changed. After this filtering, we obtained 430 valid commits showing code-test co-evolution across the 21 repositories.

To obtain negative examples to address **RQ2**, we also selected the latest 10 commits from each repository where there was a production code change, but no changes to the test methods. In some cases, there were fewer than 10 commits without test updates. In such cases, we included all relevant commits.

For each commit, we extracted the code of all test methods that were updated by the developers. We extracted the production code changes in `git diff` format, separating each change into different chunks. Each chunk includes, as additional context, the nine lines above and below the code changes.

4.2 Baseline Approach

To compare MAST to the existing state-of-the-art for LLM-based test localization, we utilize a baseline representing our prior research [21]. In our previous study, we proposed a multi-agent LLM system for test localization based on semantic analysis alone. We have derived an equivalent implementation to the framework from our previous study by extracting a portion of the MAST pipeline consisting of the Maintenance Prediction agent and the Semantic Analysis agent from Figure 2.

4.3 Data Collection

To analyze performance, we executed MAST for each commit.

(1) The information stored in the semantic RAG database, BM25 index database, and call graph were updated to reflect the test suite prior to any test changes made in that commit, associated with a unique commit-id. In case of a re-execution, the contents of these databases or call graphs do not need to be regenerated.
(2) Each individual code change (`git diff` chunk) for that commit was input into MAST.
(3) Ground truth is known for a commit, not for each chunk. To assess performance, the identified test cases for all chunks in that commit were aggregated, then compared to the ground truth for the actual commit.

We executed all agents with a temperature setting of 0.00 to help ensure stability and result reproducibility. However, we performed the evaluation three times, as there may still be some variance in results. We present the mean and standard deviation across the three executions. For each commit, we define:

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

We collect results for the full MAST pipeline. In addition, for the baseline and ablation study, we also extract performance for the following sub-workflows within MAST:

(1) **Baseline (Semantic Only):** Maintenance Prediction, Semantic Analysis
(2) **Lexical Only:** Maintenance Prediction, Lexical Analysis
(3) **Static Only:** Maintenance Prediction, Static Analysis
(4) **Without Post-Check:** Maintenance Prediction, Semantic Analysis, Lexical Analysis, Static Analysis, Fusion

Workflows 1–3 represent the three information sources alone, with the first representing the framework in our prior work [21]. Workflow 4, then, represents the framework without the final post-check analysis, with the three information sources and the fusion agent. Isolating these four workflows allows us to understand how accurate each individual analysis is and how each contributes to the overall performance of MAST.

4.4 Data Analysis

We collect the number of true and false positives and negatives for MAST and each workflow for each commit. We then aggregate the results to calculate the following metrics:

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

Depending on the research question, we either aggregate TP, FP, TN, and FN across commits for each individual project—i.e., calculate these metrics per-project—or we aggregate TP, FP, TN, and FN across all commits for all projects—i.e., calculate metrics across the full set of commits. In the latter case, we aggregate to account for the imbalance in the number of commits per-project.

We use these measurements to provide multiple viewpoints on performance. In particular, as some of the commits contain a large number of TN (i.e., a large test suite where few tests are updated), the accuracy may be artificially inflated. Therefore, the F1 and F2 scores are useful because they focus on TP, FP, and FN. While the F1 score is commonly used to assess machine learning tasks, we also include the F2 score—which places a heavier weight on recall than precision—because omitting tests that should have been included in a prediction is likely more detrimental than including some FPs. We use these metrics to address the research questions as follows:

- To address **RQ1**, we consider the set of positive commits (i.e., commits where test maintenance was required). We report all metrics for each project individually, as well as aggregated across all projects, for MAST and the baseline.
- To address **RQ2**, we focus on negative commits (i.e., cases where no test maintenance was performed). We only report accuracy, as there can be no TP or FN, only TN and FP. Therefore, the other metrics cannot be calculated. We report accuracy across all projects for MAST and the baseline. We omit results for individual projects due to the relatively small number of negative commits extracted from each individual project.
- To address **RQ3**, we report all five metrics for positive commits and accuracy alone for negative commits— aggregated across all datasets—for MAST and all four workflows (including the baseline) defined in the previous section. We omit results for individual projects due to space constraints, but they are available in our supplemental data (Section 8).

5 Results

5.1 Comparison of MAST and Baseline in Positive Cases (RQ1)

Table 2 lists the mean precision, recall, accuracy, F1 score, and F2 score of MAST and the baseline for the dataset extracted from each project, as well as aggregated across all projects. As we executed each approach three times to assess the variance of each approach, we also list the standard deviation. To answer this question, we focus on the positive commits—i.e., commits where test maintenance was necessary. For each metric, we list the best approach in **bold**.

Overall, from Table 2, we see that MAST achieves a substantially higher precision, accuracy, F1, and F2 score than the baseline. The baseline represents only a single analysis performed in MAST, the semantic analysis, and the results suggest benefits from the additional analyses of information sources as well as the fusion procedure and the post-check analysis. The improvement in performance comes with a trade-off in recall, and the baseline achieves approximately 16% higher recall than MAST. However, this trade-off is not proportional, as MAST achieves 69% higher precision and 28% higher F1 score across the full set of commits.

The overall trends holds across the individual datasets, except for three cases (*-lib-common, *-datastream-dump-*-extractor, and *-service-dataexport-download) where the baseline achieves a higher F1 score. In all three cases, the difference comes down to a major difference in recall between the approaches. In 10 datasets, due to the difference in recall, the baseline also has a higher F2 score than MAST, as the F2 score places a high emphasis on recall. In some situations, recall may be of very high importance to a tester—i.e., they are willing to sift through many false positives to ensure they do not miss any true positives. Overall, MAST achieves the best balance of results, but it may miss some tests in such situations.

MAST versus Baseline on Positive Cases (RQ1)

MAST achieves a much higher precision than the baseline, leading to better accuracy, F1, and F2 scores. However, there is a trade-off between precision and recall, with the baseline yielding higher recall than MAST.

5.2 Comparison of MAST and Baseline in Negative Cases (RQ2)

Table 3 lists results aggregated across all projects. To address RQ2, we focus on negative commits—i.e., where a production code change did not lead to subsequent test maintenance. We also focus on the baseline (first row) and the full MAST framework (final row). In negative cases, there are only FP (tests predicted as needing maintenance, when no such maintenance is needed) and TN (tests correctly predicted as not needing maintenance). Therefore, we only list the number of FP and the resulting accuracy. For negative commits, we see that MAST was much more effective than the baseline. There was an 83% reduction in false positives in such cases, leading to a 6% improvement in accuracy.

MAST versus Baseline on Negative Cases (RQ2)

In cases where test maintenance was not required, MAST yields far fewer false positives than the baseline, resulting in a higher accuracy.

5.3 Ablation Study (RQ3)

The goal of RQ3 is to understand how MAST's individual components and their interaction affect the performance of the full framework. To answer this question, we compare five workflows. First, we look at the three analyses of information sources (semantic analysis—i.e., the baseline—lexical analysis, and static analysis). Then, we examine MAST without applying the post-check analysis—i.e., we assess the fusion of the three information sources. Finally, we can compare to the final output of MAST (i.e., after applying the

Table 2: Comparison of the performance of MAST and the baseline for positive commits (maintenance required) on each project and aggregated across all projects. The best approach is in bold for each metric and repository.

Repository	Approach	Precision	Recall	Accuracy	F1 Score	F2 Score
*-dataexport-test-stability	MAST	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**	**1.000 ± 0.000**
	Baseline	0.667 ± 0.289	**1.000 ± 0.000**	0.667 ± 0.289	0.778 ± 0.192	0.889 ± 0.096
*-extractor	MAST	**0.697 ± 0.022**	**0.929 ± 0.000**	**0.864 ± 0.012**	**0.796 ± 0.014**	**0.871 ± 0.007**
	Baseline	0.460 ± 0.024	**0.929 ± 0.000**	0.667 ± 0.031	0.615 ± 0.022	0.771 ± 0.014
*-dataexport-test-e2e	MAST	**0.775 ± 0.015**	0.885 ± 0.000	**0.836 ± 0.010**	**0.826 ± 0.009**	**0.860 ± 0.004**
	Baseline	0.500 ± 0.000	**0.923 ± 0.000**	0.559 ± 0.000	0.649 ± 0.000	0.789 ± 0.000
*-extractor-2	MAST	**0.631 ± 0.024**	**0.750 ± 0.054**	**0.817 ± 0.017**	**0.685 ± 0.035**	**0.723 ± 0.045**
	Baseline	0.444 ± 0.016	**0.750 ± 0.054**	0.683 ± 0.014	0.558 ± 0.025	0.659 ± 0.038
*-service-authentication	MAST	**0.611 ± 0.048**	0.517 ± 0.029	**0.785 ± 0.015**	**0.559 ± 0.016**	0.532 ± 0.021
	Baseline	0.436 ± 0.041	**0.750 ± 0.050**	0.675 ± 0.042	0.550 ± 0.023	**0.654 ± 0.021**
*-parser	MAST	**0.692 ± 0.031**	0.923 ± 0.038	**0.920 ± 0.013**	**0.791 ± 0.034**	**0.865 ± 0.036**
	Baseline	0.427 ± 0.013	**0.974 ± 0.022**	0.781 ± 0.013	0.594 ± 0.009	0.775 ± 0.004
-datastream--scanner	MAST	**0.777 ± 0.033**	0.658 ± 0.031	**0.878 ± 0.013**	**0.712 ± 0.031**	0.678 ± 0.031
	Baseline	0.420 ± 0.020	**0.856 ± 0.056**	0.696 ± 0.019	0.564 ± 0.029	**0.709 ± 0.041**
*-lib-common	MAST	**0.846 ± 0.037**	0.596 ± 0.067	**0.710 ± 0.038**	0.698 ± 0.049	0.633 ± 0.061
	Baseline	0.687 ± 0.000	**0.885 ± 0.000**	0.707 ± 0.000	**0.773 ± 0.000**	**0.836 ± 0.000**
-datastream-dump--extractor	MAST	**0.733 ± 0.208**	0.615 ± 0.077	**0.826 ± 0.087**	0.666 ± 0.135	0.634 ± 0.098
	Baseline	0.568 ± 0.019	**0.872 ± 0.044**	0.785 ± 0.012	**0.687 ± 0.001**	**0.787 ± 0.021**
*-datastream-dump-cleaner	MAST	**0.754 ± 0.035**	0.667 ± 0.031	**0.873 ± 0.014**	**0.708 ± 0.033**	0.682 ± 0.032
	Baseline	0.425 ± 0.004	**0.884 ± 0.012**	0.697 ± 0.005	0.574 ± 0.004	**0.727 ± 0.006**
*-datastream-archive	MAST	**0.763 ± 0.042**	0.637 ± 0.027	**0.939 ± 0.007**	**0.694 ± 0.032**	0.659 ± 0.028
	Baseline	0.391 ± 0.017	**0.895 ± 0.030**	0.837 ± 0.009	0.544 ± 0.021	**0.711 ± 0.024**
*-service-dataexport-download	MAST	**1.000 ± 0.000**	0.389 ± 0.020	0.776 ± 0.007	0.560 ± 0.020	0.443 ± 0.020
	Baseline	0.809 ± 0.020	**0.684 ± 0.037**	**0.825 ± 0.010**	**0.740 ± 0.020**	**0.705 ± 0.030**
archiver	MAST	**0.650 ± 0.009**	0.635 ± 0.019	**0.868 ± 0.004**	**0.642 ± 0.014**	**0.638 ± 0.017**
	Baseline	0.418 ± 0.010	**0.723 ± 0.006**	0.761 ± 0.007	0.530 ± 0.009	0.631 ± 0.008
*-datastream-dump-scanner	MAST	**0.612 ± 0.021**	0.641 ± 0.027	**0.915 ± 0.005**	**0.626 ± 0.024**	**0.635 ± 0.026**
	Baseline	0.316 ± 0.007	**0.665 ± 0.011**	0.803 ± 0.004	0.429 ± 0.008	0.545 ± 0.010
*-service-dataexport-executor	MAST	**0.449 ± 0.020**	**0.721 ± 0.063**	**0.911 ± 0.005**	**0.554 ± 0.034**	**0.643 ± 0.048**
	Baseline	0.208 ± 0.008	0.716 ± 0.017	0.770 ± 0.008	0.323 ± 0.011	0.481 ± 0.014
*-decrypter	MAST	**0.656 ± 0.028**	0.617 ± 0.041	**0.914 ± 0.007**	**0.635 ± 0.031**	**0.624 ± 0.037**
	Baseline	0.356 ± 0.011	**0.631 ± 0.008**	0.817 ± 0.006	0.455 ± 0.011	0.546 ± 0.010
*-datastream-common	MAST	**0.865 ± 0.045**	0.889 ± 0.048	**0.995 ± 0.002**	**0.877 ± 0.042**	**0.884 ± 0.044**
	Baseline	0.279 ± 0.006	**1.000 ± 0.000**	0.945 ± 0.002	0.436 ± 0.008	0.659 ± 0.007
common	MAST	**0.917 ± 0.061**	0.464 ± 0.005	**0.841 ± 0.009**	**0.616 ± 0.012**	0.515 ± 0.004
	Baseline	0.545 ± 0.011	**0.645 ± 0.009**	0.754 ± 0.006	0.591 ± 0.008	**0.622 ± 0.008**
*-datastream-dump-common	MAST	**0.711 ± 0.016**	0.603 ± 0.004	**0.971 ± 0.001**	**0.653 ± 0.009**	0.622 ± 0.006
	Baseline	0.394 ± 0.006	**0.732 ± 0.014**	0.937 ± 0.001	0.513 ± 0.008	**0.625 ± 0.011**
*-datastream-dump-archiver	MAST	**0.521 ± 0.003**	0.623 ± 0.020	**0.934 ± 0.001**	**0.567 ± 0.009**	**0.599 ± 0.015**
	Baseline	0.320 ± 0.007	**0.678 ± 0.008**	0.878 ± 0.002	0.435 ± 0.008	0.554 ± 0.009
*-service-dataexport-handler	MAST	**0.729 ± 0.011**	0.522 ± 0.022	**0.950 ± 0.002**	**0.608 ± 0.017**	0.554 ± 0.021
	Baseline	0.404 ± 0.008	**0.646 ± 0.025**	0.903 ± 0.002	0.497 ± 0.013	**0.577 ± 0.019**
Overall	MAST	**0.621 ± 0.003**	0.613 ± 0.014	**0.932 ± 0.001**	**0.617 ± 0.009**	**0.615 ± 0.012**
	Baseline	0.367 ± 0.001	**0.708 ± 0.005**	0.865 ± 0.000	0.483 ± 0.002	0.597 ± 0.003

Table 3: Ablation study on positive and negative commits, aggregated across all projects.

	Positive Commits					Negative Commits	
Workflow	Precision	Recall	Accuracy	F1 Score	F2 Score	False Positives	Accuracy
Baseline (Semantic-Only)	0.367 ± 0.001	0.708 ± 0.005	0.865 ± 0.000	0.483 ± 0.002	0.597 ± 0.003	688.700 ± 19.500	0.931 ± 0.004
Lexical-Only	0.395 ± 0.002	0.458 ± 0.003	0.889 ± 0.000	0.424 ± 0.002	0.444 ± 0.002	200.300 ± 25.000	0.980 ± 0.003
Static Analysis-Only	0.502 ± 0.003	0.590 ± 0.005	0.911 ± 0.001	0.542 ± 0.003	0.570 ± 0.004	188.700 ± 4.500	0.981 ± 0.000
Fusion Without Post-Check	0.355 ± 0.004	**0.778 ± 0.018**	0.854 ± 0.001	0.487 ± 0.007	**0.628 ± 0.012**	769.000 ± 8.200	0.923 ± 0.003
MAST (All Analyses)	**0.621** ± 0.003	0.613 ± 0.014	**0.932 ± 0.001**	**0.617 ± 0.009**	0.615 ± 0.012	**113.300 ± 36.500**	**0.989 ± 0.002**

final post-check analysis). Table 3 lists results for each workflow aggregated across all positive and negative commits.

5.3.1 Comparison of Information Sources. Comparing the analyses of three information sources (semantic, lexical, and static analysis) on positive commits, we see that the static analysis yields a higher precision than the other two sources (0.502) and a balanced recall, leading to the highest F1 score of the three analyses. As noted previously, the semantic analysis yields a substantially higher recall—and somewhat higher F2 score—than the other two methods. However, on negative commits, it yields many more false positives than the

Table 4: Comparison of LLM-based and a traditional heuristic-based fusion, aggregated across all projects.

Workflow	Precision	Recall	Accuracy	F1 Score	F2 Score
LLM Fusion w/o Post-Check	0.355 ± 0.004	**0.778 ± 0.018**	0.854 ± 0.001	0.487 ± 0.007	**0.628 ± 0.012**
Heuristic Fusion w/o Post-Check	**0.612 ± 0.002**	0.500 ± 0.002	**0.927 ± 0.000**	**0.550 ± 0.002**	0.519 ± 0.002
MAST (LLM Fusion)	0.621 ± 0.003	**0.613 ± 0.014**	0.932 ± 0.001	**0.617 ± 0.009**	**0.615 ± 0.012**
MAST (Heuristic Fusion)	**0.736 ± 0.002**	0.421 ± 0.005	**0.935 ± 0.000**	0.536 ± 0.004	0.460 ± 0.004

other two information analyses. In negative cases, the lexical and static analysis yield similar performance, with a slight edge to the static analysis.

While the lexical analysis yields lower performance than the other methods, when we examine the tests recommended by each information source qualitatively, there are some TP results suggested uniquely by each of the three analyses. Therefore, we see value in retaining each information source.

5.3.2 Effect of Fusion. From Table 3, in positive cases, we can see that the fusion—before the post-check—yields a lower precision than any of the individual analyses. However, it also has the highest recall (and F2 score) of any workflow, including the full MAST framework. Because we use an LLM to perform the fusion—rather than taking a simple union or intersection of the three sources—we achieve more nuanced results.

The fusion yields a relatively inclusive list of test cases from the three sources, but still filters some cases with limited support. For example, the lexical analysis suggests a large number of FPs (hence its low precision) that were filtered by the fusion step. The fusion results in a high number of true positives—achieving high recall—but also many false positives that must be filtered by the post-check—i.e., the precision is relatively low. We see a similar effect in negative cases, where the fusion yields more false positives and—as a result—a lower accuracy than the individual information analyses because it produces a relatively inclusive list of recommendations from the individual analyses.

5.3.3 Effect of Post-Check. In MAST, the post-check analysis takes each prediction from the fused list and compares the test code to the change in the production code to make one final assessment of the relevance of the suggestion. The intention of this analysis is to filter spurious suggestions from the fused list. From Table 3, and as discussed earlier, we can see that the post-check is successful in this task. In both positive and negative cases, the number of false positives is far lower for the full MAST framework (i.e., after post-check) than after the fusion, resulting in improved precision, accuracy, and F1 score. There is some cost to the recall as well, but less than the relative improvement in the precision.

In cases where recall is highly prioritized, a tester may wish to inspect the fusion results instead of—or in comparison with—the full results of the MAST framework. However, given the number of false positives and the relative similarity in F2 score, the cost of inspecting all cases in the fused list must be considered.

5.3.4 LLM-Based Versus Traditional Fusion Methods. A hypothesis underlying the design of MAST was that LLM-based fusion could yield superior results to traditional, heuristic-based fusion methods. We assessed this hypothesis by also implementing a version of MAST with a traditional fusion method, where a test is included in the merged list as long as it is in more than one of the fused lists. The comparison between the two fusion methods before the post-check, as well as the full MAST framework with each fusion method, is shown in Table 4.

From the results, we can again see the tension between precision and recall. The heuristic fusion yields a higher precision, accuracy, and F1 score than the LLM-based fusion. However, the LLM-based fusion yields a similarly substantial increase in recall and F2 score over the heuristic method. This further shows that the LLM-based fusion is not a simple merging of the individual lists, but the product of a more complex analysis.

We can see the full effect when each fusion is integrated into the full MAST framework, with its subsequent post-check. MAST with the LLM-based fusion yields a substantially higher recall, F1, and F2 score, as well as a similar accuracy—while MAST with the heuristic fusion yields a higher precision. Overall, while neither method yields better results in all measurements, the LLM-based fusion ultimately leads to a better overall performance due to its recall—i.e., its high true positive rate.

Ablation Study (RQ3)

Each information source yields different, but potentially valuable, maintenance suggestions. The fusion agent effectively merges the sources, and the post-check prunes many false positives from the merged list (albeit at some cost in recall). The superior results of MAST in precision, accuracy, and F1 score over the individual workflows show that each agent contributes to the overall effectiveness of MAST. However, in situations where recall is highly prioritized, a tester may also wish to inspect the fusion results prior to the post-check.

6 Discussion

6.1 Implications of the Results

From the results, we can see that MAST achieves a higher precision, accuracy, F1, and F2 score than the baseline on positive commits and a higher accuracy on negative commits—due to substantially fewer false positives. Like in many machine learning tasks, there is a natural tension between precision and recall. The baseline—as well as the fusion results before applying the post-check—offer a higher recall than the full MAST framework. However, this trade-off is not symmetrical. MAST achieves a substantially higher precision with only some loss in recall. Overall, therefore, we recommend the use of MAST over the baseline or the fusion results except in cases where any loss in recall is unacceptable.

Our findings highlight the complementary nature of the different analysis techniques. The semantic analysis captures the intentions underlying each test case, but also suffers from lower precision by

operating on natural language interpretations of the test code. The fusion overcomes this limitation by also incorporating the lexical analysis—which captures syntactic similarity of the code—and the static analysis—which captures structural dependencies between tests and production code. The post-check then successfully filters cases from the fused list of test cases that lack sufficient support for their inclusion. MAST effectively fuses information from different sources, offering evidence that LLM-based analysis fusion could assist in other software engineering tasks as well.

6.2 Recommendations for Practitioners

A natural question is—are the results of MAST good enough to use the framework in practice? To contextualize the current results of MAST, an average of 6.5 test cases are updated in each positive commit in the ground truth. When MAST is executed for a commit, it will generally yield four correct recommendations (TP), two incorrect recommendations (FP), and miss two tests that should have been identified (FN).

These results are promising. About two-thirds of MAST's recommendations are correct, and the number of false positives is not burdensome. The main barrier to relying on MAST are the false negatives—the missed cases. If MAST were used as the sole decision-maker for test maintenance, then the project quality could still suffer. These results suggest that there is a strong potential for applying LLM-based tools in industrial test maintenance, but also as an indication that there is still room for improvement.

Our recommendation is to use MAST as a way to accelerate the test maintenance process, but not as the sole determination of what tests to update and not as a total replacement for human judgment. We envision two primary use-cases for MAST:

- MAST could be manually invoked after the production code is updated, but before it is committed to the repository, by the developer. The developer could use its recommendations, along with their own intuition or other tools, to perform any required test maintenance.
- MAST could be invoked in the continuous integration pipeline after the production changes are committed (along with any planned test maintenance) as a way to identify potentially missed maintenance cases. Its recommendations could then be offered as advice to the developer.

In both cases, MAST can ease the burden of test localization, but its predictions should be taken as recommendations that are validated by the developer.

6.3 Threats to Validity

6.3.1 External Validity. We conducted our evaluation at one company, Ericsson AB, and focused on one programming language, Java. Therefore, some aspects of this study may not generalize beyond this specific context. MAST is not closely coupled to any particular technology or project at Ericsson, nor to specific aspects of the Java language (except for some parsing tools). However, the specific performance trends may not hold on all projects or all languages. We partially mitigated this threat by curating commits from 21 different projects, covering diverse domains and project sizes. Therefore, because the projects used in our evaluation were of a similar scope, we speculate that our results will hold—at least—on other small-to-medium sized Java projects. In the future, we will also consider open-source projects and additional languages.

6.3.2 Internal Validity. We made two assumptions when collecting commits. First, we assumed that tests were updated because of the production code changes in the same commit (i.e., code and tests co-evolved). Second, we assumed that no tests were missed by developers. Both assumptions are in-line with previous research, but may also be incorrect at times. Some tests may have been missed or were not updated until later commits. This means that some false positives may actually be valid suggestions—something we witnessed in our past work [21]. The performance of MAST in real-world situations may differ somewhat from in the evaluation. However, in some cases, "false positives" may still offer valuable suggestions to developers.

6.3.3 Construct Validity. We evaluated MAST against a baseline drawn from our prior work [21], but we did not include the approaches proposed by Hu et al. [13] or Chi et al. [6]. However, their approaches cannot be executed under equivalent conditions (the limitations of prior work were discussed in Sections 1–2), and the effort needed to re-implement their approaches for our context could not be justified.

7 Conclusion

In this study, we presented MAST, a multi-agent framework that performs test localization following a change to the production code. MAST advances the state-of-the-art by integrating multiple code analysis techniques—including static, lexical, and semantic analysis—and through our focus on a realistic use and evaluation setting—i.e., MAST operates on standard input formats, is not limited to single-method changes, does not require an explicit mapping of production and test code, and has been evaluated on cases where test maintenance was and was not required in the ground truth.

MAST was more effective than a baseline from our prior work, achieving higher precision, accuracy, F1, and F2 scores. We witnessed a trade-off between precision and recall. However, MAST still yields a far higher precision than the baseline, with only some loss in recall. Our ablation study demonstrates the value of each integrated analysis in producing the final set of recommendations.

In future work, we plan to improve the performance of MAST through techniques such as prompt optimization and the integration of long-term memory, expand our analysis to open-source repositories, explore languages other than Java, and expand MAST to perform maintenance actions on the localized test cases.

Acknowledgments: Support for this research was provided by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

8 Data Availability Statement

The current version of MAST is available at https://github.com/Roy19921010/MAST. Because this framework may evolve in the future, we also make available a supplementary package at https://doi.org/10.5281/zenodo.19680703 containing a snapshot of the version of MAST used in this study, as well as the experiment results for all datasets. The Ericsson repositories used in our study cannot be made availability due to confidentiality constraints.

References

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

[2] Maurício Aniche. 2022. *Effective Software Testing: A developer's guide.* Simon and Schuster.

[3] Max Brunsfeld. 2018. Tree-sitter: A parser generator tool and incremental parsing library. https://github.com/tree-sitter/tree-sitter. Accessed: 2026-04-10.

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

[5] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation. arXiv:2402.03216 [cs.CL]

[6] Jianlei Chi, Xiaotian Wang, Yuhan Huang, Lechen Yu, Di Cui, Jianguo Sun, and Jun Sun. 2025. REACCEPT: Automated Co-evolution of Production and Test Code Based on Dynamic Validation and Large Language Models. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA055 (June 2025), 23 pages. doi:10.1145/3728930

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

[8] Chaoyang Gao, Xiang Chen, and Guangbei Zhang. 2025. SVA-ICL: Improving LLM-based software vulnerability assessment via in-context learning and information fusion. *Information and Software Technology* 186 (2025), 107803. doi:10.1016/j.infsof.2025.107803

[9] Raji Ghawi and Jürgen Pfeffer. 2019. Efficient hyperparameter tuning with grid search for text categorization using kNN approach with BM25 similarity. *Open Computer Science* 9, 1 (2019), 160–180.

[10] Danielle Gonzalez, Joanna CS Santos, Andrew Popovich, Mehdi Mirakhorli, and Mei Nagappan. 2017. A large-scale study on the usage of testing patterns that address maintainability attributes: patterns for ease of modification, diagnoses, and comprehension. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 391–401.

[11] Piyush Gupta, Sangjae Bae, and David Isele. 2025. Graph-Grounded LLMs: Leveraging Graphical Function Calling to Minimize LLM Hallucinations. arXiv:2503.10941 [cs.AI] https://arxiv.org/abs/2503.10941

[12] Junda He, Christoph Treude, and David Lo. 2025. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.* (Jan. 2025). doi:10.1145/3712003

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

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

[15] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE transactions on big data* 7, 3 (2019), 535–547.

[16] Kiarash Naghavi Khanghah, Hoang Anh Nguyen, Anna C. Doris, Amir Mohammad Vahedi, Daniele Grandi, Faez Ahmed, and Hongyi Xu. 2026. MCERF: Advancing Multimodal LLM Evaluation of Engineering Documentation with Enhanced Retrieval. arXiv:2604.09552 [cs.IR] https://arxiv.org/abs/2604.09552

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

[18] Inc. LangChain. 2024. LangGraph: Building Stateful, Multi-Actor Applications with LLMs. https://github.com/langchain-ai/langgraph. Accessed: 2026-04-10.

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

[20] Ying Li, Ye Zhong, Lijuan Yang, Yanbo Wang, and Penghua Zhu. 2025. LLM-Guided Crowdsourced Test Report Clustering. *IEEE Access* 13 (2025), 24894–24904. doi:10.1109/ACCESS.2025.3530960

[21] Jingxiong Liu, Ludvig Lemner, Linnea Wahlgren, Gregory Gay, Nasser Mohammadiha, and Joakim Wennerberg. 2025. Exploring the Integration of Large Language Models in Industrial Test Maintenance Processes. arXiv:2409.06416 [cs.SE] https://arxiv.org/abs/2409.06416

[22] Jun Liu, Jiwei Yan, Yuanyuan Xie, Jun Yan, and Jian Zhang. 2024. Fix the Tests: Augmenting LLMs to Repair Test Cases with Static Collector and Neural Reranker. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 367–378.

[23] Panagiotis Louridas. 2006. Static code analysis. *Ieee Software* 23, 4 (2006), 58–61.

[24] Yang Luo. 2025. Can we translate code better with LLMs and call graph analysis?. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*. 7625–7633.

[25] Mike McQuaid. 2014. *Git in practice.* Simon and Schuster.

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

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

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

[29] Stephen Robertson and Hugo Zaragoza. 2009. *The probabilistic relevance framework: BM25 and beyond.* Vol. 4. Now Publishers Inc.

[30] B.G. Ryder. 1979. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering* SE-5, 3 (1979), 216–226. doi:10.1109/TSE.1979.234183

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

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

[33] Qwen Team. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[34] Jingxiang Tu, Bo Lin, Yihao Qin, Shangwen Wang, Liqian Chen, and Xiaoguang Mao. 2025. Trace: Test Repair via Agent-based Context Extraction with LLMs. In *2025 32nd Asia-Pacific Software Engineering Conference (APSEC)*. 57–68. doi:10.1109/APSEC66846.2025.00017

[35] Bart Van Rompaey and Serge Demeyer. 2009. Establishing traceability links between unit test cases and units under test. In *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 209–218.

[36] Chaozheng Wang, Zezhou Yang, Shuzheng Gao, Cuiyun Gao, Ting Peng, Hailiang Huang, Yuetang Deng, and Michael Lyu. 2025. Rag or fine-tuning? a comparative study on lcms-based code completion in industry. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*. 93–104.

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

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

[39] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[40] Tangzhi Xu, Jianhan Liu, Yuan Yao, Cong Li, Feng Xu, and Xiaoxing Ma. 2025. Comprehend, Imitate, and then Update: Unleashing the Power of LLMs in Test Suite Evolution. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 469–481. doi:10.1109/ASE63991.2025.00046

[41] Zezhou Yang, Ting Peng, Cuiyun Gao, Chaozheng Wang, Hailiang Huang, and Yuetang Deng. 2025. A deep dive into retrieval-augmented generation for code completion: Experience on wechat. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 608–619.

[42] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.

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

Received 23 April 2026; accepted 21 June 2026