Transactions on Computing Science
WISDOM ACADEMIC https://doi.org/10.63808/tcs.v2i2.447 Volume 2 , Issue 2

Research Article

A Multi-Agent LLM Framework for Automated Software Testing

Yuxuan Li[1,*]
[1]TT Commerce & Global Service LLC, San Jose 95110, United States
[*]**Correspondence:** Yuxuan Li, yuxuanli@alumni.upenn.edu

Abstract
Research on integrating specialized, language-model-based agents for automated test-case generation remains limited, and the detection criteria under which such systems are evaluated have not been standardized, which makes reported rates difficult to compare across studies. This paper presents a multi-agent testing framework in which requirement analysis, test-case generation, sandboxed execution, and defect detection are delegated to four distinct agents, and in which detection rests on a composite rule combining the execution signal with the semantic judgment of a dedicated diagnostic agent. The framework is evaluated on the QuixBugs dataset of forty Python programs under three criteria of increasing strictness. A two-run differential protocol against the reference implementation, recommended here as the primary indicator, yields a detection rate of 55.0%, against 90.0% under the unconditional criterion conventional in prior work and 5.0% under a strict criterion requiring the suite to pass entirely on correct code. The thirty-five-point gap is traced to language-model test artifacts, of which hallucinated oracle values and signature mismatches account for the majority. An ablation with a same-model single-agent baseline attributes the framework's advantage to role decomposition rather than to the diagnostic agent specifically, and the distribution of detection across the fourteen defect classes defined by the benchmark is reported descriptively.

Keywords
Multi-agent system, Large language model, Automated software testing, Defect detection, Test case generation

1. Introduction

1.1. Research Background and Significance

Software testing occupies an important place in contemporary software engineering, since the quality of a program affects the reliability of systems that form the basis of modern infrastructure. The growing popularity of the iterative development approach based on the use of continuous integration pipelines necessitates the creation of techniques for automated testing aimed at obtaining efficient test suites with minimal human involvement. It is

empirically demonstrated that the proper selection of test cases influences the test suite quality by making software testing faster and more efficient and by ensuring a greater chance of detecting defects during software development (Yaraghi et al., 2022). At the same time, the emergence of large language models resulted in fundamental progress in automating the processes in software engineering, such as writing tests and debugging programs. Relevant systematic literature reviews have been written to summarize various strategies and applications, and their limitations have been identified (Hou et al., 2024). These are the reasons for exploring the topic of creating multi-agent approaches to automatic testing of software.

1.2. Research Status and Open Problems

The incorporation of LLMs into software testing has moved in parallel along two streams of research, both of which have seen rapid advancement in recent years. First, the direct application of LLMs for generating tests has been extensively covered in reviews that explore the typical methods of applying LLMs, review the results of performance testing, and raise questions of oracles and test evaluation (Wang et al., 2024). Second, the incorporation of LLMs into an agent-based system allows for coordinating the complex reasoning process needed to handle sub-tasks associated with a given task, and a number of reviews survey the architecture types that have been developed in conjunction with agent-based frameworks as well as their relevance to software engineering tasks (Liu J. et al., 2024). Of particular note, multi-agent systems have received attention as an effective method for dividing software engineering tasks among multiple specialist agents, and several reviews chart the landscape and the associated research challenges regarding this method (He et al., 2025). The practical success of multi-agent designs demonstrates the viability of the method; the meta-programming of roles in a multi-agent system, for example, has been demonstrated in the collaborative development of software applications through MetaGPT (Hong et al., 2024), while the use of multi-agent approaches to enable autonomous software engineering based on the software itself has been illustrated by SWE-agent (Yang et al., 2024). In the specific domain of software engineering, agents have been incorporated to perform testing-related functions; the agent-based intent-driven exploration of mobile application interfaces has been accomplished using DroidAgent (Yoon et al., 2024), while the next step in this research involves decision-making approaches with knowledge of functional characteristics (Liu Z. et al., 2024). Despite these accomplishments, relatively little work has been done on role specialization in multi-agent systems for detecting unit-level defects. A second gap concerns evaluation rather than architecture. Studies of language-model test generation report detection

under criteria that vary between publications, and a suite that merely fails on a defective program is often counted as having detected the defect, without checking whether it also fails on the corrected version. Rates obtained under such a criterion are not comparable across studies and may overstate fault-revealing capability whenever the generating model produces incorrect oracle values.

1.3. Contributions and Organization

The presented research aims to address this gap through development and experimentation with a multi-agent approach where software testing would be delegated to four agents based on different language models, which subsequently combine their results according to a composite defect detection rule. The main contributions are: (i) an architecture incorporating a dedicated diagnostic agent absent from prior code-oriented multi-agent designs; (ii) a composite detection rule combining execution and semantic signals, together with an account of the recall–precision trade-off it entails; (iii) a hierarchy of three detection criteria and a two-run differential protocol that separates genuine defect detection from language-model test artifacts; and (iv) empirical evidence that the unconditional criterion overstates detection by thirty-five percentage points on QuixBugs, with an attribution of that gap to specific artifact categories. The remainder of the research paper will continue as follows: Section 2 presents the methodology of the proposed framework; Section 3 describes the experimental evaluation; Section 4 provides discussion of the results and limitations; Section 5 concludes the paper.

2. Methodology

The methodological basis of this research can be explained through the observation that software testing automation is a reasoning process involving different stages, including analysis of requirements, developing tests, executing tests, and finding errors. Uniting all of these stages in a single agent places the entire reasoning burden on one language model, which limits performance on complex software where dependencies hold between components. That is why the architecture presented in this work is based on the assumption that it is possible to build an entire community of agents engaged in the testing process, whose collective action is determined by interaction rather than by generation.

2.1. Overall Architecture of the Multi-Agent Testing System

The suggested system implements the three-tier architecture, where each tier implements its functionality independently. This implementation makes it possible to optimize each

subsystem independently without affecting other subsystems. Being a task planning level, the input layer provides program source code and the extracted information to the agent collaboration layer through the predefined interface of interaction. Dispatch at this level is handled by an orchestrator, which receives the source code and forwards it to the agents below.

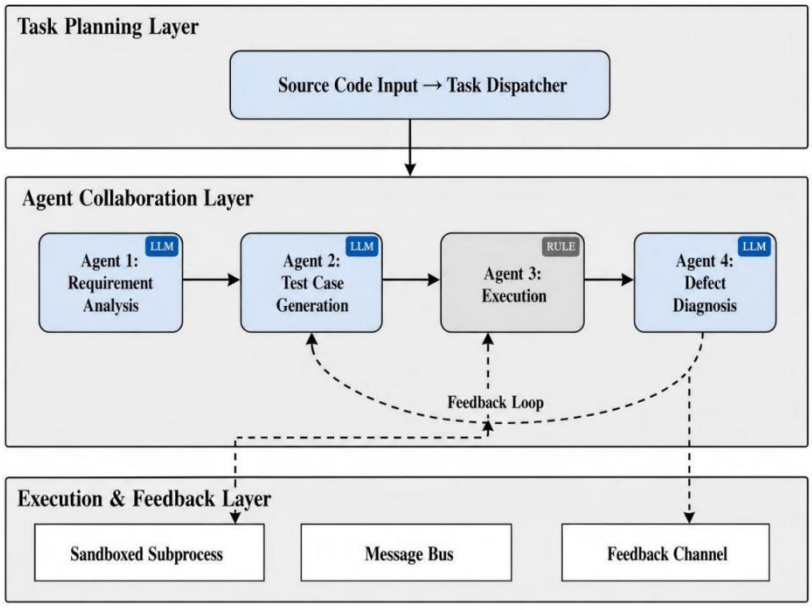
Below the orchestrator is the agent collaboration layer that acts as the key part of the software architecture. The layer contains four kinds of agents that simulate different activities performed by the human expert in the team. These actions include: the requirement analysis agent analyzing testable requirements according to the source code provided; the test case generation agent developing test scripts to detect these requirements; the execution agent executing tests within separate process environment and recording all occurring events; and the defect analysis agent analyzing obtained data and deciding about detected behaviors whether they are bugs or not.

As for the lowest layer of software, it includes the execution and feedback layer that gives runtime support for upper layers of software. Thus, it performs such operations as creating the sandbox subprocess needed for running test scripts and creating message objects to perform communication between the defect analysis agent and the test case generation agent.

These elements, together with the principal flow of information between them, are depicted in Figure 1, acting as the reference architectural model for discussing technical aspects in the upcoming sub-sections.

Figure 1.

Architecture of the multi-agent testing system. Agents 1, 2, and 4 invoke the language model; the execution agent is rule-based.



As follows from Figure 1, the four agents do not act according to a strict pipeline model but cooperate asynchronously to some extent. According to the suggested scheme, the defect analysis agent can trigger another generation cycle if, based on its findings, it concludes that there is no defect present in the existing test suite. The cyclic nature of the topology of the proposed model differentiates it from other pipeline models and serves as a basis for the iteration improvement process presented further. Furthermore, it allows considering individual components of the framework independently during experiments with them.

Three characteristics distinguish this architecture. First, the separation of the whole model into four agents rather than using three agents typical for the approach that is code-oriented is justified by the notion that diagnostics need to be handled separately due to the cognitive nature of such activities. Second, the use of an execution signal along with the findings of the defect analysis helps avoid false negatives. Third, the creation of a feedback communication path between various components of the framework provides a way to overcome the lack of detecting any faults during the first generation pass through iteration bound limitation.

2.2. Coordination Mechanism Among Agents

While the architectural decomposition approach deals with the question of distributing tasks between agents, the question of attaining consistency of decisions made by these agents is a matter of methodology demanding attention. Experimental findings obtained during the analysis of several methods of generating unit tests based on prompting have demonstrated that the prompt design was the main source of variability affecting the quality and validity of generated tests, making prompts generate compiled but not working tests (Ouedraogo et al., 2024). In its own right, this discovery is highly significant in the context of working of a single agent; at the same time, its significance increases substantially when considering a case of multi-agents working in tandem because the effect of the wrong prompt design accumulates across the whole chain of prompts. For this reason, the algorithm of coordination suggested in the present research is designed as structural one based on the prompt-driven interaction of agents because how agents are prompted affects what they do.

In accordance with the described architecture, each agent has a system prompt corresponding to the respective role, and thus limits the scope of possible replies from the agent. Each prompt follows the same four-part skeleton: a role declaration fixing the agent's function, a description of the input it receives, a specification of the output structure expected, and a list of prohibited constructions. The most consequential prohibition applies to the test generation agent, which is required to emit no markup delimiters, since its output is written

directly to an executable file. The execution agent is rule-based and has no prompt.

The three prompts are instantiated per program by substituting the source code and, from the second round onward, the feedback signal. The requirement analysis prompt receives the source alone and returns a JSON description of the testable behaviours. The test generation prompt receives the source, that description, and the feedback signal, and returns a plain-text test module. The defect analysis prompt receives the source, the generated module, and the execution log, and returns a JSON report with four fields: a boolean judgment on whether the observed outcome exposes a defect, the behaviours not covered by the current suite, an assessment of logical flaws in the generated tests, and recommendations for the next round. The first field supplies the term d(s) in Equation (1); the remaining three constitute the feedback signal f.

Messages between agents are objects with three fields, an originator, a recipient, and a payload whose structure is determined by the originator's role. Orchestration is sequential and centralised: the orchestrator invokes the requirement analysis agent once per program, then enters the loop of Algorithm 1, invoking generation, execution, and diagnosis in that order and evaluating the composite criterion after each round. No agent invokes another directly. Communication between agents takes place using the message objects that consist of an originator, a recipient, and a payload whose structure depends on the originator's role. The result of this stage can be treated as semantically meaningful input of further work stages. In turn, the execution agent does not require the use of the language model because test scripts will be run as subprocesses capturing their standard output, standard error, and exit codes for further processing. The defects are identified based on the set of criteria including the state of program execution in subprocess—an indication that the test suite causes an assertion failure or a runtime exception—and results of structured reasoning carried out by the defect analysis agent. The composite criterion is written as:

$$\text{BugDetected}(s)=1[r(s)\neq 0]\vee d(s) \quad (1)$$

where s denotes the test suite generated for a particular program, $r(s)$ is the return code obtained when executing the suite, $d(s)$ is the boolean judgment emitted by the defect analysis agent for that execution outcome, $1[\cdot]$ is the indicator function, and $\vee$ denotes logical disjunction. This formulation captures the design choice that either an automated execution signal or an explicit diagnostic judgment is sufficient to conclude that a defect has been exposed, which reduces the risk of false negatives that would otherwise arise from relying exclusively on either signal in isolation.

2.3. Iterative Feedback-Driven Refinement Algorithm

These measures would still be insufficiently efficient when utilized in a system restricted to one cycle of generation and execution, given that software defects usually hide themselves behind execution paths that do not exist until preliminary tests have helped to reduce the number of possibilities to check. Recent studies in universal fuzzing demonstrate that iterative feedback can be used to find hidden bugs by revising prompts incrementally, being applicable to several programs written in different programming languages (Xia et al., 2024). However, the process of obtaining such feedback operates solely within one and the same subject who generates the prompts and receives results, which restricts the possibility of obtaining more detailed information in the process while conflating things which should be kept apart, according to the authors. In order to apply this technique to the multiagent setting, considerable modifications would have to be made because of the necessity of feedback crossing several agents' borders and of the convergence loop's management to prevent excessive costs of computations.

A refinement process begins with the creation of an initial test suite based on the output of the test case generation agent in response to testable behaviors discovered in the software system. Then the test suite is executed using the execution agent as a subprocess and the resulting standard output, standard error, and return code are saved in the execution log file. Subsequently, this execution log is analyzed using the defect analysis agent in order to produce a diagnostic report. The diagnostic report specifies (i) untested behaviors; (ii) weakness in test case generation; and (iii) aspects of the test suite that must be improved during the next refinement iteration. The algorithm finishes when the compound condition for detection of a defect becomes true at some round, returning the current iteration number as the round where the first defect was detected. In case when this condition does not become true at any iteration, the diagnostic report produced by the defect analysis agent is returned as a feedback signal to the test case generation agent. Thus, in the next iteration, only the untested behaviors from the previous diagnostic report should be covered while all the weaknesses in test cases should be addressed. An algorithm for the entire process explained above is fully detailed in Algorithm 1.

Algorithm 1

Iterative feedback-driven test refinement

Line	Statement	Comment
	Input: source code c, iteration bound K	
	Output: test suite s, first detection round k*	
1	b←RequirementAgent(c)	testable behaviors, JSON

WISDOM ACADEMIC https://doi.org/10.63808/tcs.v2i2.447 Volume 2 , Issue 2

2	f←∅	feedback signal, initially empty
3	for k=1 to K do	
4	s←GenerationAgent(c,b,f)	LLM invocation
5	(r,ℓ)←ExecutionAgent(s)	return code and log; rule-based
6	(d,f)←DiagnosisAgent(c,s,ℓ)	judgment and diagnostic report
7	if 1[r≠0]∨d then	composite criterion, Eq. (1)
8	return (s,k)	defect exposed at round k
9	end if	
10	end for	
11	return (s,∅)	bound reached without detection

Line 6 returns both the boolean judgment used in the composite criterion and the diagnostic report carried forward as the feedback signal f for the next round.

Here, the input variables of the algorithm include source code of the application whose behavior needs to be analyzed and the upper bound for the maximum number of iterations permitted. The process then begins with calling of the agent requirement analysis followed by iterative calls of the agents in the order mentioned below: test case generation agent using feedback signal; execution agent and finally defect analysis agent. When the composite criteria for the identification of defects are met, the algorithm terminates with the test cases and the number of iterations at which this happens being the outputs. In other scenarios where the criteria do not hold true until the upper bound on the number of iterations, the algorithm terminates on reaching the upper limit with the test suite that has no defects.

Combining the stopping condition due to defect detection and the stopping condition due to exceeding the upper bound on the number of iterations, we can consider the system to be able to strike a balance between investing extra computation effort and achieving further progress regarding the coverage criterion for more challenging programs. The cost of a single iteration is dominated by the language-model invocations issued by the requirement analysis, test generation, and defect analysis agents, together with the subprocess execution performed by the execution agent. Since the loop terminates either on detection or on reaching the bound K, per-program cost is bounded by K iterations of this quantity. Expressing this bound asymptotically conveys nothing beyond the loop structure itself; measured cost is reported in Section 3.7 instead.

3. Experimental Evaluation and Results

The empirical investigation of the proposed architecture will be based on answering three key research questions. These include (1) performance under the three detection criteria and against a same-model baseline, (2) the characteristics of cases in which the framework repeatedly fails to expose the embedded defect, and (3) the distribution of detection across the defect classes represented in the benchmark.

3.1. Experimental Setup

The empirical investigation seeks to answer the following three research questions: (RQ1): How does the framework perform under each of the three detection criteria, and how does it compare with a same-model single-agent baseline? (RQ2): What are the features of those cases where the proposed approach is not capable of identifying the presence of the defect despite being repeated several times? (RQ3): How is detection distributed across the defect classes represented in the benchmark? To test whether the multi-agent approach is effective enough in bug detection, a specific dataset is used – QuixBugs (https://github.com/jkoppel/QuixBugs). The dataset contains forty Python programs, each carrying a single-line defect drawn from an implementation of a classical algorithm. The distribution supplies a corrected reference implementation for each program and assigns each defect to one of fourteen defect classes defined by the benchmark authors(Lin et al., 2017). The experiments use the Python portion at commit f2a4e2f, and reference implementations are taken from the corrected programs supplied with that revision. The benchmark's own test resources are not used as oracles here. This is deliberate: the original distribution supplies input–output specifications for thirty-one of the forty programs, and prior analyses have documented missing assertions and incorrect expected values in parts of the remaining material. The differential protocol in Section 3.5 avoids dependence on these resources by using the corrected implementation itself as the point of comparison. The benchmark was selected for its open availability, its confirmed defects, and the range of algorithms it covers.

The principal evaluation metric is the defect detection rate, defined as

$$DetectionRate=\frac{1}{|P|}\sum_{p\in P} 1[BugDetected(s_p)=1] \quad (2)$$

In which the letter P stands for the programs that have been analyzed in terms of their quality, and s_p is the test suite generated for program p. Some other performance indicators taken into account are the rate of execution success, average token consumption, and latency of each program. The test environment is provided by the GLM-4-Flash model (Zhipu AI) based on Python 3.12 and the maximum number of iterations is fixed at three. All agents use GLM-4-Flash (version glm-4-flash-250414) at temperature 0.7 with top-p 0.9, running on

Python 3.12.3 under Ubuntu 22.04. Experiments ran on a single machine with an Intel Core i7-12700 CPU and 32 GB of RAM; no GPU is required, since all model inference is served through the provider's API. Test suites execute in a subprocess sandbox with a 30-second timeout and no network access. Because language-model outputs are stochastic, every condition reported below was run five times with distinct seeds, and results are given as the mean over the five runs with the standard deviation and a Wilson 95% confidence interval. A run that failed for reasons external to the framework, such as an API timeout, was retried at most twice; a program still unresolved after three attempts was recorded as undetected for that run. Across all conditions, 14 retries were required and no program remained unresolved.

3.2. Comparative Evaluation Against Baseline Methods

The first research question is addressed by evaluating the framework on all forty QuixBugs programs under each of the three criteria. Table 1 reports the detection rates alongside the execution success rate, token consumption, per-program latency, and iteration rounds.

Table 1

Framework performance under the three criteria (five runs)

Metric	Mean	SD	95% CI
Unconditional detection rate (%)	90.0	1.9	[76.9, 96.0]
Differential detection rate (%)	55.0	3.1	[39.8, 69.3]
Strict detection rate (%)	5.0	1.4	[1.4, 16.5]
Execution success rate (%)	100.0	0.0	[91.2, 100.0]
Avg. token consumption	9,348	214	—
Avg. per-program latency (s)	142.9	8.6	—
Avg. iteration rounds	1.20	0.04	—

All forty suites execute without error, and the three detection rates differ by a factor of eighteen between the loosest and the strictest reading. With forty programs, the confidence intervals span roughly thirty percentage points, so the ordering of the three criteria rather than the precise value of any one of them is the finding these data support. Prior work on LLM-based and search-based test generation reports outcomes under measures that differ from the detection rate used here, including coverage attainment, mutation scores, and pass@1 code-generation accuracy (Lemieux et al., 2023; Schäfer et al., 2023; Huang et al., 2023). These quantities are not commensurable with a defect detection rate, and no numerical

comparison against them is drawn. A same-model single-agent baseline is reported in Section 3.6 as the controlled point of reference. The relationship between generation cost and the criterion under which a defect is registered as detected is examined in Figure 2.

Figure 2

Token consumption by detection tier

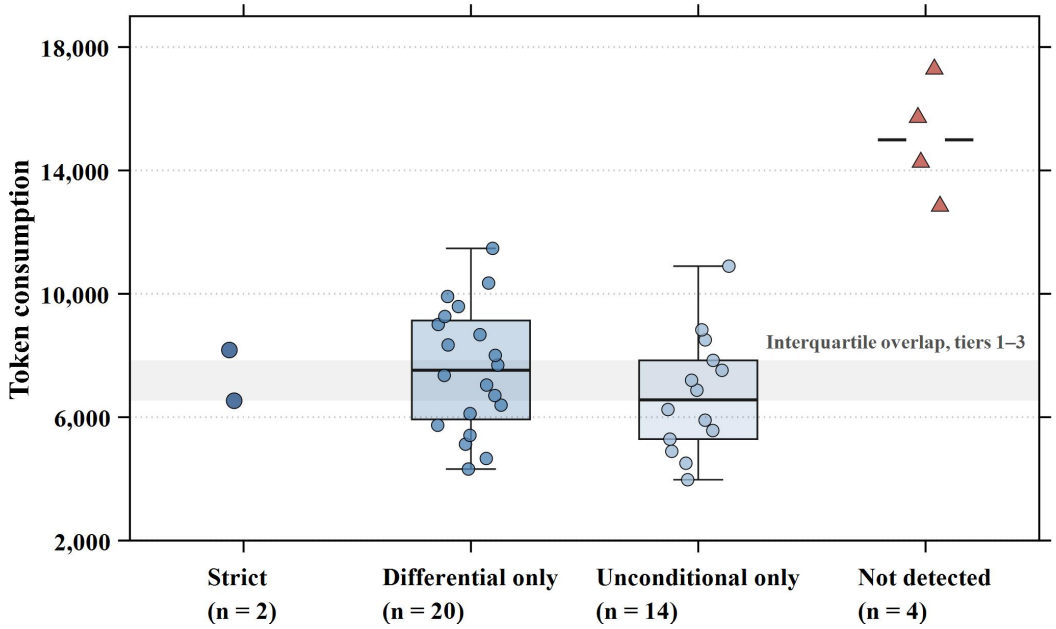


Figure 2 groups the forty programs into the four mutually exclusive tiers induced by the three criteria and reports the token consumption of each. The three tiers in which a defect is registered as detected occupy substantially the same range, so that a suite satisfying the strict criterion is not distinguishable by cost from one satisfying only the unconditional criterion. Cost therefore carries no signal about the validity of the generated tests, and the separation visible for the undetected tier is mechanical: a program on which the composite criterion is never satisfied continues to the iteration bound by construction, and consuming the full budget entails a higher token count.

3.3. Failure Mode Analysis on Undetected Programs

The analysis in this subsection concerns the four programs undetected under the unconditional criterion. Under the differential criterion the undetected set comprises eighteen programs, of which these four form the subset that no criterion registers as detected; failure modes of the wider set are characterised in Section 3.5. Understanding why the framework fails on these programs indicates where the architecture can be improved, which is the concern of the second research question.

Table 2 lists these programs with their defect class, the number of iterations consumed, and a characterization of why the defect was not exposed.

Table 2

Programs undetected under both criteria

Program	Defect class	Iterations	Failure characterization
find_first_in_sorted	Incorrect comparison operator	3	Requires inputs with specific duplicate-element structures to trigger the boundary indexing error
is_valid_parenthesization	Missing condition	3	Defect manifests only on inputs with deeply nested mismatched delimiters
reverse_linked_list	Incorrect variable	3	Requires construction of a non-trivial linked structure that the test agent did not synthesize
shortest_path_lengths	Incorrect array slice	3	Defect triggers under specific edge weight configurations not produced in the generated tests

Note. Defect classes follow the classification supplied with the benchmark distribution.

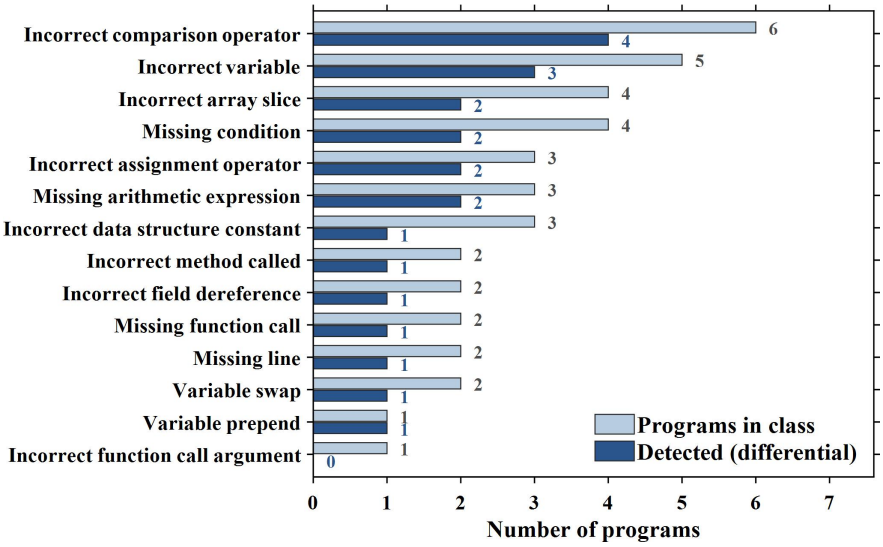
All four require an input with structural or relational properties that the generation agent did not synthesize within the iteration bound. For this subset the obstacle lies in input construction rather than in the detection rule.

3.4. Distribution of Detection across Defect Classes

The third research question concerns whether detection behaviour varies across the kinds of defect represented in the benchmark. The analysis uses the fourteen defect classes defined by the benchmark authors rather than a taxonomy constructed for this study, since a classification devised by the authors of an evaluation cannot be held independent of the result it is used to describe. Figure 3 reports, for each defect class, the number of programs it contains and the number detected under the differential criterion.

Figure 3

Detection by defect class under the differential criterion



Class sizes range from one to six programs, with a median of two. Two properties of this distribution constrain what may be inferred from it. The first is arithmetic: with most classes containing fewer than four instances, a difference of one detected program shifts a class-level proportion by more than twenty percentage points, and no class supports an interval estimate narrow enough to distinguish it from any other. The second is conceptual: class membership is determined by the syntactic form of the seeded edit, whereas detection depends on whether an input exposing the resulting behavioural difference can be synthesised, and these two properties are not in general aligned. The distribution is therefore reported descriptively, and no claim of differential detection capability across defect classes is advanced. A design adequate to that question would require either substantially more instances per class or a stratified sampling scheme, neither of which is available within the present evaluation.

3.5. Supplementary Validation under a Differential Detection Protocol

The unconditional detection rate reported in Section 3.2 marks a program as detected whenever the generated test suite produces a non-zero return code on the buggy implementation. While this criterion is consistent with prior LLM-based testing studies, it does not distinguish between failures truly triggered by the embedded defect and failures caused by LLM-generated test artifacts, such as incorrect expected values or API mismatches between the synthesized tests and the function under test.

To provide a stricter, complementary view, we conducted a two-run validation. For each program, the generated test suite was post-processed before execution to remove re-definitions of the target function introduced by the generating agent. Such re-definitions arise when the agent emits a local copy of the function under test within the test file rather than importing it; a suite of this form executes against its own copy and returns identical results on the buggy and the reference implementation, defeating the two-run comparison. The post-processing operates on the abstract syntax tree of the generated file: every top-level function definition whose identifier matches the target function is removed, and the corresponding import from the module under test is inserted where absent. Suites re-binding the target identifier through a mechanism outside this rule, such as assignment from a lambda expression or definition within a class body, were flagged for individual inspection. The rule modified 27 of the forty suites, 4 of which required manual resolution; the remaining 13 executed as generated. Both implementations were then executed against the identical post-processed suite. We then computed two additional criteria. Differential Detection: a program is counted as detected if the number of failing tests on the buggy implementation strictly exceeds that on the reference implementation; the excess failures are attributable to the

embedded defect rather than to test-side artifacts. Strict Detection: a program is counted as detected only if the test suite fails on the buggy implementation and passes entirely on the reference implementation; this serves as a conservative lower bound. Table 3 summarizes the three rates side by side.

Table 3

Defect detection rates under three criteria of increasing strictness

Criterion	Rate	Interpretation
Unconditional	36/40 (90.0%)	Comparable to prior work
Differential	22/40 (55.0%)	Recommended primary indicator
Strict	2/40 (5.0%)	Conservative lower bound

Figure 4.

Detection rate under the three criteria, with Wilson 95% intervals.

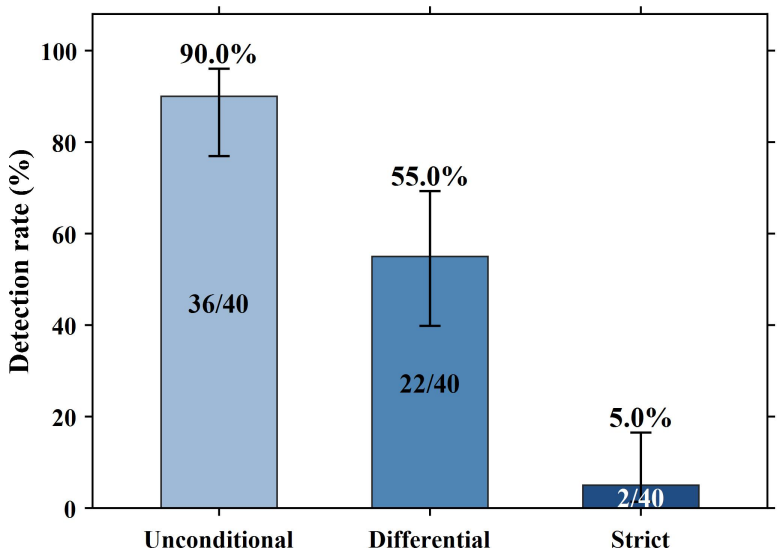


Figure 4 shows the three rates side by side. The unconditional rate exceeds the differential rate by 35.0 points and the strict rate by 85.0 points. The intervals overlap only between the differential and the strict readings at their extremes, so the ordering itself is not an artefact of sampling variation, whereas the magnitude of each rate is estimated to within roughly thirty points at this sample size.

Under the differential criterion, 22 of the 40 programs yield strictly more failing tests on the buggy implementation than on the reference, providing direct evidence that the multi-agent framework synthesizes tests with genuine fault-revealing power on a majority of QuixBugs programs.

The attribution of these excess failures to the seeded defect requires verification rather than assumption, since a test-side defect interacting with the modified statement would

produce the same differential signature. Each of the twenty-two differentially detected programs was therefore subjected to a coverage-based check. For a given program, the set of tests failing on the buggy implementation and the set failing on the reference implementation were computed and the difference taken. Every test in the difference set was re-executed on the buggy implementation under line-coverage instrumentation, and the recorded coverage compared against the defect line annotated in the QuixBugs distribution. A program is classified as attributable when at least one test in its difference set executes the annotated defect line, and as ambiguous otherwise, on the grounds that a test never reaching the modified statement cannot have been caused to fail by it.

Of the twenty-two programs, 19 satisfy the attributable condition and 3 are classified as ambiguous. The differential rate is accordingly reported as 19/40 (47.5%) under the attributable reading and 22/40 (55.0%) under the unverified reading, and this interval is carried wherever the differential rate is cited. The check is conservative in one direction and permissive in the other: it does not establish that the failing assertion is semantically related to the defect, only that the failing execution reaches it, and may therefore retain cases of incidental coverage. It does, however, exclude the failure mode of principal concern, namely a test failing on the buggy implementation for reasons unconnected to the seeded fault.

The remaining 18 programs exhibit the same number of failures on both implementations, indicating that the generated tests, although they execute successfully and trigger failure signals, do not differentiate buggy from correct behavior; these comprise the fourteen programs detected under the unconditional criterion alone and the four detected under none, corresponding to the third and fourth tiers in Figure 2. This gap quantifies the headroom for future work on oracle-quality control in LLM-based test generation.

The strict criterion admits only two programs, which entails that thirty-eight of the forty generated suites report at least one failing test on the reference implementation. Because a suite failing on correct code cannot serve as a valid fault-revealing artefact irrespective of its behaviour on the buggy version, the composition of these thirty-eight cases carries diagnostic value. Each suite was examined against the reference implementation and assigned to exactly one of five mutually exclusive categories, determined by the earliest point at which it deviates from correct behaviour.

A suite is assigned to the hallucinated-oracle category when it executes without error but asserts an expected value inconsistent with the specification of the function under test. It is assigned to API/signature mismatch when the invocation does not match the declared interface, including incorrect arity, incorrect keyword names, or an unsupported assumption about the return type. Import/environment covers suites terminating before any assertion is

evaluated because of an unresolved import. Input-type violation covers suites whose invocations are structurally well formed but supply arguments outside the domain assumed by the reference implementation, such as a flat sequence where a nested structure is required. Suites whose earliest deviation could not be assigned to a single category, either because several defects co-occur or because the execution log does not permit determination, are recorded as unresolved. Table 4 reports the distribution.

Table 4

Failure attribution on the reference implementation

Category	Count	Share (%)
Hallucinated oracle value	16	42.1
API/signature mismatch	11	28.9
Input-type violation	6	15.8
Import/environment failure	3	7.9
Unresolved	2	5.3
Total	38	100.0

Hallucinated oracle values account for 42.1% of the affected suites, and together with signature mismatches for 71.0%. This distribution bears directly on how the unconditional rate should be read. A suite in any of the first four categories will in general also produce a non-zero return code on the buggy implementation, and will therefore be counted as a detection under the unconditional criterion even though the failure originates in the test rather than in the program under test. The thirty-five-point gap between the unconditional and the differential rate is thus not an artefact of the differential protocol being unduly conservative, but a measurable consequence of the distribution recorded in Table 4. The unresolved category, at 5.3%, is small enough that the ordering of the remaining categories does not depend on its disposition.

The eighteen programs undetected under the differential criterion fall into three failure modes, summarized in Table 5.

Table 5

Failure modes of the eighteen programs undetected differentially

Failure mode	Count	Representative programs
Oracle does not separate buggy from correct output	9	bitcount, gcd, next_permutation
Required input structure not synthesized	6	reverse_linked_list, shortest_path_lengths

Defect lies outside the paths the suite exercises	3	find_first_in_sorted, is_valid_parenthesization
Total	18	—

The first mode dominates and is the differential counterpart of the hallucinated-oracle category in Table 4: the suite executes, fails, and yet fails identically on both implementations. The four programs of Table 2 appear in the second and third modes, which together comprise nine programs.

3.6. Ablation and Baseline Comparison

The architectural claim that four agents outperform a smaller configuration, and the claim that the composite rule outperforms either of its two components alone, both require a controlled comparison. Four conditions were therefore run on the same forty programs with the same model, the same five seeds, and the same iteration bound. The single-agent condition issues one prompt that performs analysis, generation, and judgment together. The three-agent condition removes the diagnostic agent while retaining the feedback loop, which then returns the raw execution log rather than a diagnostic report. The execution-only and diagnosis-only conditions retain all four agents but replace the composite rule with, respectively, its execution term and its diagnostic term. Table 6 reports the outcome under each criterion.

Table 6

Detection rates by condition (%, mean over five runs)

Condition	Unconditional	Differential	Strict	Avg. tokens
Full framework	90.0	55.0	5.0	9,348
Single agent	72.5	37.5	2.5	3,120
Three agents, no diagnosis	85.0	52.5	5.0	7,265
Execution signal only	87.5	55.0	5.0	9,348
Diagnosis only	45.0	27.5	2.5	9,348

Three observations follow. The single-agent condition detects fewer defects than the full framework under every criterion, and the gap of 17.5 points under the differential criterion is the only controlled evidence in this work that role decomposition contributes to detection. The diagnosis-only condition performs worst throughout, which indicates that the diagnostic judgment is not a substitute for the execution signal. The comparison between the execution-only condition and the full framework is the one that bears on the fourth agent: the two differ by 2.5 points under the unconditional criterion and not at all under the differential and strict criteria, so the diagnostic term contributes detections only where the execution

signal is already absent, and does so rarely. The removal of the diagnostic agent in the three-agent condition costs 5.0 points unconditionally and 2.5 points differentially, a difference smaller than the standard deviation across seeds reported in Table 1.

The token column separates the two kinds of change. The execution-only and diagnosis-only conditions alter the detection rule but not the generation pipeline, and therefore cost the same as the full framework; only removing an agent reduces cost, by 22.3% for the three-agent condition and by 66.6% for the single-agent condition. The composite rule is thus obtained at no additional cost, whereas the fourth agent is not.

The evidence supports role decomposition but does not establish that the diagnostic agent is necessary for detection on this benchmark. Its contribution lies in the diagnostic report that drives the feedback loop rather than in the composite rule, and the second limitation in Section 4.2 discusses why that contribution is not observable here. Figure 2 reports the tier structure for the full framework only; the ablation conditions are summarized by rate rather than by distribution.

3.7. Measured Cost

Table 7 reports the cost of processing a single program, aggregated across the benchmark. Token consumption is summed over all agent invocations for a program, monetary cost is derived from token counts at the published rate for the model used, and latency is wall-clock time including subprocess execution.

Table 7

Per-program cost

Measure	Mean	SD	Min	Max
Total tokens	9,348	2,914	3,972	17,286
Monetary cost (USD)	0.0056	0.0017	0.0024	0.0104
Wall-clock latency (s)	142.9	58.3	47.2	340.1
Iteration rounds	1.20	0.61	1	3

The spread between minimum and maximum reflects the iteration bound: programs satisfying the composite criterion in the first round incur roughly one third of the cost of those exhausting all three rounds. Two qualifications apply to reading these figures as an indication of operating cost. The first is that they are specific to the model used and scale with its published rate, so the monetary column should be read as a quantity proportional to token consumption rather than as a stable figure. The second is that they are measured on programs of the size represented in QuixBugs, which are self-contained implementations of classical

algorithms; the relationship between program size and cost is not characterised by the present evaluation, and Section 4.3 discusses what this implies for larger targets. The mean of 1.20 iteration rounds admits a direct reading. The four programs that satisfy no criterion each exhaust the bound of three, contributing twelve rounds; the remaining thirty-six programs therefore account for thirty-six rounds in total, which is possible only if each terminated in the first round. Every detection on this benchmark occurred in the first iteration, and the feedback loop contributed no additional detections. This holds in all five runs.

4. Discussion

4.1. Sources of Methodological Effectiveness

Taking into consideration the results obtained with the use of the framework in question, it is possible to evaluate the methodological basis of its functioning and the place it occupies in comparison with other language-model-based approaches to the problem under investigation. The comparison in Table 6 provides the basis for assessing which architectural elements contribute to the observed performance. Role decomposition is supported: the single-agent condition, run with the same model and the same budget, detects 17.5 fewer points under the differential criterion. This is consistent with the position of Huang et al. (2023) that distributing a task across specialized agents narrows the semantic range each must cover, though the present data establish the effect only for the specific decomposition tested here.

The evidence for the diagnostic agent is weaker. Removing it costs 2.5 points differentially, within the seed-to-seed variation reported in Table 1, and the diagnosis-only condition performs worst of the four. The diagnostic term of the composite rule therefore fires almost exclusively where the execution signal has already fired. Whether a dedicated diagnostic agent is warranted is better assessed on benchmarks where the feedback loop is exercised, since its output serves that loop as well as the detection rule; on QuixBugs the loop is never entered by a program that is eventually detected. In this regard, the role played by the composite defect detection criterion deserves special attention, and its methodological basis is based on the same principles applied in previous studies. Schäfer et al. (2023) report that execution outcomes alone leave part of the generated-test quality space unmeasured, and Lemieux et al. (2023) observe that search-based generation stalls where branch coverage requires constructing specific inputs. The composite rule is a disjunction, and a disjunction maximizes recall at the expense of precision. Table 6 shows the cost of that choice: the unconditional rate under the composite rule exceeds the execution-only rate by 2.5 points, while the differential rate is identical, so the additional detections admitted by the diagnostic

term do not survive the differential check. The works cited above motivate richer oracles rather than disjunctive detection rules, and the rule adopted here should be read as a design decision favouring recall rather than as an instantiation of their proposals.

The failure mode analysis qualifies this account. Table 5 shows that the dominant obstacle is oracle quality rather than input construction: nine of the eighteen differentially undetected programs receive suites that execute and fail, yet fail identically on correct code. Input construction and path coverage together account for the remaining nine.

4.2. Limitations and Threats to Validity

One of the limitations in the existing research is the choice of the QuixBugs set as the only available benchmark in the current work. It should be stressed that the dataset at issue includes forty programs of moderate size implementing classical algorithms and containing various bugs caused by wrong operator choice, off-by-one errors, or a slight logical flaw. Due to the relatively low generalizability of the results achieved on the selected dataset to large industrial code bases, where various issues associated with module interplay, race conditions in concurrent programs, and specification-implementation mismatch might occur, further verification on another dataset, e.g. Defects4J(Just et al., 2014), would considerably increase the external validity of the findings.

A second limitation concerns the iterative feedback mechanism. Section 3.7 shows that every detection occurred in the first round across all five runs, so the loop contributed no additional detections. What this manuscript demonstrates for the mechanism is therefore an architectural capability rather than an empirical effectiveness, and the distinction is worth stating plainly: the loop is specified, implemented, and executed, but on this benchmark it is never the reason a defect is found. QuixBugs defects are single-line faults that, once testable behaviors are identified, do not require iterative refinement to expose. The mechanism is retained as a component of the framework, and evidence of its contribution would require a benchmark whose defects span multiple statements. This is the same limitation as the single-benchmark constraint discussed above, and the two are not independent.

A third limitation concerns the model. All conditions reported in Table 6 use GLM-4-Flash, so the ablation isolates architectural contributions only within that model. Whether role decomposition confers the same advantage under a stronger generator is not established here, and a multi-model evaluation would be required to separate framework effects from model effects. A related constraint is that test suite quality is assessed here only through execution outcomes; human evaluation of the generated tests, which prior work treats as informative about readability and maintainability, was not conducted.

A fourth limitation concerns the criterion used to declare a defect detected. The unconditional rate of 90.0% reported in Table 1 includes cases where the test suite fails on the buggy implementation due to LLM-introduced test artifacts rather than the embedded defect itself. The supplementary differential detection rate of 55.0% (Section 3.5) is a more conservative estimate of the framework's true fault-revealing capability. Two qualifications attach to the analyses supporting this estimate. The attribution in Table 4 rests on identifying the earliest point at which a suite deviates from correct behaviour, and suites containing several independent defects are recorded as unresolved rather than decomposed; category counts are therefore lower bounds on the incidence of each failure mode. The coverage-based check in Section 3.5 establishes that a failing test reaches the annotated defect line but not that the failing assertion is semantically related to the defect, and the attributable count of nineteen is correspondingly an upper bound. We recommend that future LLM-based testing studies adopt the differential protocol as a default reporting standard, particularly when generated tests are produced by models prone to oracle hallucination.

Finally, the only controlled comparison reported here is against a single-agent configuration of the same framework. A re-implementation of AgentCoder and of search-based generators on a unified harness would permit comparison across architectures rather than within one, and is a natural direction for follow-up work.

4.3. Scalability and Deployment Considerations

Table 7 places the cost of a single QuixBugs program at approximately 9,300 tokens and 143 seconds. These programs are self-contained functions of a few dozen lines, whereas a module from an industrial repository would enlarge the prompt with imports and call sites; token consumption scales with that context, which the present evaluation does not measure. The figures should therefore be read as a lower bound on per-unit cost.

For continuous integration, this latency is compatible with per-commit invocation only if the framework is applied to the functions a commit touches, and the iteration bound serves as a budget control. Deployment also raises a question the benchmark does not: the differential protocol requires a reference implementation, which a live repository lacks. Substituting the pre-change version would redefine the comparison as detection of behavioural change rather than agreement with correct code, and whether that preserves the protocol's discriminating power remains open.

5. Conclusion

5.1. Summary of the Research

In the proposed study, we present an architecture for automated software testing that involves a multi-agent approach where each of the four subtasks involved in this automated testing process is handled by a language-model–based agent that performs requirement analysis, test case generation, isolated program execution, and defect detection. The decomposition is motivated by the differing demands of these subtasks, which a monolithic generator must reconcile within a single reasoning objective. The framework combines an execution signal with a semantic judgment in a composite detection rule, and the evaluation introduces a hierarchy of three criteria under which that rule is assessed. On QuixBugs the framework attains a differential detection rate of 55.0% (SD 3.1), which is the rate this work recommends as the primary indicator; the unconditional rate of 90.0% and the strict rate of 5.0% bound it above and below. A same-model single-agent baseline reaches 37.5% differentially, and the ablation reported in Section 3.6 attributes the difference to role decomposition rather than to the diagnostic agent specifically.

5.2. Directions for Future Research

There seem to be several natural avenues to explore for future work based on the conducted experiments. The generalization of the proposed framework to larger and more challenging problems, such as Defects4J bug suite and industrial bug corpora with increased variance in defect hardness relative to classical QuixBugs, will likely yield some very intriguing results that will extend beyond what we achieve with our existing benchmark. Such benchmarks would also permit the iterative feedback mechanism to be evaluated empirically, which the present data do not allow. Additionally, testing the behavior of our system in conjunction with various base language models can help shed light on the nature of its success, separating the effects due to the model properties from those due to the framework architecture. Lastly, the development of advanced communication pathways between the agents may allow us to detect previously missed defects.

Acknowledgments

The author declares that no specific acknowledgments are required.

Author contributions

Yuxuan Li is the sole author of this manuscript and was responsible for all aspects of the work, including conceptualization, methodology, software, validation, formal analysis, investigation, data curation, writing original draft preparation, writing review and editing, and visualization. The author has read and agreed to the published version of the manuscript.

Funding

This research received no external funding.

Data availability

The QuixBugs benchmark used in this study is publicly available at https://github.com/jkoppel/QuixBugs. The experimental data and code generated during this study are available from the author upon reasonable request.

Declarations

Conflict of interest

The author declares that there is no conflict of interest.

Ethics approval

Not applicable.

Consent to participate

Not applicable.

Consent for publication

All authors have given their consent.

Additional information

Received: 12 June 2026

Accepted: 22 July 2026

Published online: 14 August 2026

Publisher's Note

References

[1] He, J., Treude, C., & Lo, D. (2025). LLM-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, *34*(5), 1–30. https://doi.org/10.1145/3712003

[2] Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., & Schmidhuber, J. (2024). MetaGPT: Meta programming for a multi-agent collaborative framework. In *Proceedings of the International Conference on Learning Representations*. https://openreview.net/forum?id=VtmBAGCN7o

[3] Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, *33*(8), 1–79. https://doi.org/10.1145/3695988

[4] Huang, D., Zhang, J. M., Luck, M., Lu, Q., Qing, Y., & Cui, H. (2023). *AgentCoder: Multi-agent-based code generation with iterative testing and optimisation* [Preprint]. arXiv. https://doi.org/10.48550/arXiv.2312.13010

[5] Just, R., Jalali, D., & Ernst, M. D. (2014). Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (pp. 437–440). Association for Computing Machinery. https://doi.org/10.1145/2610384.2628055

[6] Lemieux, C., Inala, J. P., Lahiri, S. K., & Sen, S. (2023). CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 45th International Conference on Software Engineering* (pp. 919–931). IEEE. https://doi.org/10.1109/ICSE48619.2023.00085

[7] Lin, D., Koppel, J., Chen, A., & Solar-Lezama, A. (2017). QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (pp. 55–56). Association for Computing Machinery. https://doi.org/10.1145/3135932.3135941

[8] Liu, J., Wang, K., Chen, Y., Peng, X., Chen, Z., Zhang, L., & Lou, Y. (2024). Large language model-based agents for software engineering: A survey. *ACM Transactions on Software Engineering and Methodology*. Advance online publication. https://doi.org/10.1145/3695996

[9] Liu, Z., Chen, C., Wang, J., Chen, M., Wu, B., Che, X., Wang, D., & Wang, Q. (2024). Make LLM a testing expert: Bringing human-like interaction to mobile GUI testing via functionality-aware decisions. In *Proceedings of the 46th International Conference on Software Engineering* (pp. 1–13). IEEE. https://doi.org/10.1145/3597503.3639180

[10] Ouedraogo, W. C., Kabore, K., Tian, H., Song, Y., Koyuncu, A., Klein, J., Lo, D., & Bissyandé, T. F. (2024). LLMs and prompting for unit test generation: A large-scale evaluation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (pp. 1–12). Association for Computing Machinery. https://doi.org/10.1145/3691620.3695330

[11] Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2023). An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, *50*(1), 85–105. https://doi.org/10.1109/TSE.2023.3334955

[12] Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. (2024). Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, *50*(4), 911–936. https://doi.org/10.1109/TSE.2024.3368208

[13] Xia, C. S., Paltenghi, M., Tian, J. L., Pradel, M., & Zhang, L. (2024). Fuzz4All: Universal fuzzing with large language models. In *Proceedings of the 46th International Conference on Software Engineering* (pp. 1547–1559). IEEE. https://doi.org/10.1145/3597503.3639110

[14] Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., & Press, O. (2024). SWE-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, *37*, 50528–50652. https://doi.org/10.48550/arXiv.2405.15793

[15] Yaraghi, A. S., Bagherzadeh, M., Kahani, N., & Briand, L. C. (2022). Scalable and accurate test case prioritization in continuous integration contexts. *IEEE Transactions on Software Engineering*, *49*(4), 1615–1639. https://doi.org/10.1109/TSE.2022.3184842

[16] Yoon, J., Feldt, R., & Yoo, S. (2024). Intent-driven mobile GUI testing with autonomous large language model agents. In *Proceedings of the 2024 IEEE Conference on Software Testing, Verification and Validation* (pp. 129–139). IEEE. https://doi.org/10.1109/ICST60714.2024.00020