Security of Infocommunication Systems and Internet of Things

2026 Vol 4, No 1

https://doi.org/10.31861/sisiot2026.1.01005

Received 10 March 2026; revised 06 April 2026; accepted 08 June 2026; published 30 June 2026

LLM-Assisted Repair Gating for Improving Hidden-Test Success in Automated Code-Validation Pipelines

Roman Zaiats[1], Myroslav Strynadko[2] and Halyna Lastivka[1,*]

[1]Department of the Radio Engineering and Information Security, Yuriy Fedkovych Chernivtsi National University, Chernivtsi, Ukraine
[2]Departmet of Correlation Optics, Yuriy Fedkovych Chernivtsi National University, Chernivtsi, Ukraine

*Corresponding author (E-mail: g.lastivka@chnu.edu.ua)

ABSTRACT Large language models (LLMs) are increasingly explored as tools for code generation, debugging, and automated program repair. However, their reliable use in software-validation pipelines remains constrained by the risk of accepting plausible but incorrect patches, especially in environments where final correctness is determined by hidden evaluation tests rather than visible checks alone. This paper investigates an LLM-assisted repair framework integrated into a gated code-validation pipeline. Instead of treating the LLM as an unconstrained code generator, the proposed approach introduces a repair stage into a controlled workflow with sequential validation checks and final hidden-test assessment. The framework was evaluated on a benchmark of fifty programming tasks under two paired conditions: a baseline execution pipeline without automated repair and an LLM+gating configuration in which a candidate repair was applied before validation. The results show a substantial improvement in final correctness: in the baseline condition, only 5 of 50 tasks passed hidden tests, whereas the LLM+gating configuration achieved hidden-test success on all 50 tasks. The observed difference was highly significant under paired statistical analysis, while runtime measurements indicated no meaningful computational overhead compared with the baseline workflow. These findings show that LLM-assisted repair can become an effective component of automated validation pipelines when deployed within an explicit repair-and-gating workflow. The study also provides a reproducible experimental protocol and supplementary research package for independent verification of the reported results.

KEYWORDS large language models, automated program repair, hidden tests, code-validation pipeline, AI-assisted debugging.

I. INTRODUCTION

Large language models (LLMs) have rapidly evolved from code-completion assistants into general-purpose software engineering tools capable of code generation, explanation, debugging, and automated repair. Recent surveys show that research in this area now covers not only model architectures and prompting strategies, but also evaluation methodologies, benchmark design, deployment limitations, and practical usage scenarios in software development [1, 2]. At the same time, practitioner-oriented studies indicate that LLM-based coding assistants are increasingly viewed as viable instruments for everyday development workflows, particularly for repetitive programming tasks, rapid prototyping, debugging, and maintenance support [3].

The growing adoption of LLMs for programming has made evaluation a central research problem. Classical code-generation benchmarks are no longer sufficient for judging modern code-oriented models, especially in settings where the model can exploit benchmark contamination or optimize for visible tests alone. Recent benchmark efforts therefore emphasize contamination-resistant, execution-based, and reasoning-oriented evaluation. In particular, LiveCodeBench introduces a dynamic and contamination-aware benchmark that extends beyond standard code generation to include code execution, self-repair, and test-output prediction [4], while ProBench focuses on competitive programming and highlights the growing importance of code reasoning in advanced models [5]. In addition, MultiPL-E demonstrates the need for multilingual and cross-language benchmarking when assessing neural code generation systems [6]. Complementing benchmark design, critical reviews of code-generation evaluation stress that correctness alone is an insufficient criterion and that software-oriented assessment should also consider efficiency, maintainability, and broader quality-related dimensions [7-9].

This issue becomes especially important in automated execution pipelines for infocommunication and Internet of Things (IoT) software systems. Such pipelines often operate under strict correctness and validation requirements, where the final decision is determined not only by visible tests but also by hidden evaluation suites, regression checks, and deployment constraints. In these environments, an LLM-generated patch may appear plausible and locally acceptable while still failing concealed validation or introducing undesired behavior. Accordingly, the main challenge is no longer whether LLMs can generate code, but how they can be integrated into software-validation workflows in a manner that is reliable, reproducible, and operationally safe.

A closely related body of research addresses automated program repair (APR). Recent work shows that LLMs can support both bug localization and patch generation, often improving repair effectiveness when compared with earlier template-based or search-based methods [10]. More broadly, studies on the impact of

code language models on APR and on APR in the era of large pre-trained language models demonstrate that language-model-based repair has become a major direction in software engineering research [11, 12]. Surveys of APR techniques and literature reviews dedicated specifically to LLM-based APR further show that the field is now moving toward more structured repair workflows that incorporate contextual retrieval, validation strategies, and benchmark-aware evaluation [13-16].

Another important development is the emergence of AI-assisted debugging and interactive repair support. Rather than using LLMs solely as code generators, recent systems integrate them into debugging environments and collaborative maintenance workflows. ChatDBG, for example, augments conventional debuggers with LLM-driven conversational analysis of failures and program state [17], while studies in embedded-system development and debugging show that LLMs can already support practical hardware-software debugging scenarios relevant to edge and IoT settings [18]. These works suggest that the most promising role of LLMs in engineering practice is not unrestricted generation, but structured assistance embedded into explicit validation and interaction loops.

Despite these advances, an important systems-level gap remains. Existing studies have largely focused on code generation, repair effectiveness, or benchmark design in isolation. Comparatively less attention has been devoted to how an LLM-assisted repair stage can be systematically embedded into an automated validation pipeline with sequential gates and hidden-test-based final acceptance. In particular, while prior work demonstrates that LLMs can generate or refine patches, it does not fully address the architectural question of how such repair should be integrated into reproducible execution workflows suitable for validation-oriented environments.

This paper addresses that gap by investigating an LLM-assisted repair stage integrated into a gated code-validation pipeline. Instead of treating the LLM as an unconstrained code generator, the proposed framework introduces a repair stage into a controlled validation loop that includes sequential execution gates and final hidden-test verification. The aim of the study is not to benchmark a specific commercial or open-source model, but to evaluate how a controlled repair mechanism affects final correctness outcomes in an automated validation setting.

The main contributions of this work are as follows:

- **End-to-end framework**: the reproducible pipeline architecture is proposed for integrating an LLM-assisted repair stage into an automated gated validation workflow;
- **Repair gating policy**: the structured gating mechanism is introduced to control when candidate repairs are applied and how they are verified within the execution pipeline;
- **Experimental protocol**: the controlled paired benchmark of fifty programming tasks is designed to compare a baseline execution pipeline with an LLM-gating configuration;
- **Quantitative evaluation**: the study reports hidden-test success, runtime characteristics, and paired statistical significance of repair outcomes;
- **Reproducible supplementary package**: the structured research package is provided to support independent replication of the experimental workflow, logs, and results.

The remainder of this paper is organized as follows. Section II reviews related work on LLMs for programming, automated program repair, and code-evaluation benchmarks. Section III introduces the system model and the LLM-in-the-loop pipeline architecture. Section IV describes the experimental setup. Section V presents the experimental results and discussion. Section VI addresses threats to validity. Finally, Section VII concludes the paper and outlines directions for future work.

II. RELATED WORK

Research on large language models for programming has developed along several closely connected directions, including code generation, benchmarking and evaluation, automated program repair, and AI-assisted debugging. Recent survey papers show that LLM-based software engineering has already evolved from isolated code-completion tasks toward broader workflows involving synthesis, explanation, transformation, testing, and maintenance support [1, 2]. In addition to general technical surveys, practitioner-oriented studies indicate that LLM-based tools are increasingly viewed as useful assistants in real software-development settings, especially for repetitive coding, rapid prototyping, debugging, and productivity enhancement [3]. These observations confirm that the role of LLMs in software engineering is no longer limited to experimental use, but is gradually shifting toward integration into practical development pipelines.

A substantial portion of recent literature has focused on the evaluation of LLMs for code generation and programming tasks. In this area, a major challenge is the design of benchmarks that fairly assess model capability without contamination or overfitting to known datasets. LiveCodeBench addresses this issue by introducing a contamination-resistant benchmark that evaluates not only code generation, but also execution, self-repair, and related code-oriented tasks [4]. ProBench extends this perspective to competitive programming and highlights the importance of advanced reasoning and problem-solving ability in modern code LLMs [5]. MultiPL-E further broadens the benchmarking landscape by showing that multilingual and cross-language evaluation is essential when studying neural code generation across diverse programming ecosystems [6].

At the same time, the literature increasingly recognizes that correctness alone is not sufficient for evaluating generated programs. Critical reviews of code-generation evaluation point out that software-oriented assessment should account for multiple dimensions, including efficiency, maintainability, and reliability [7]. This view is reinforced by recent work on efficient code generation [8] and by multidimensional benchmark proposals that explicitly move beyond functional correctness as the sole metric of model performance [9].

Taken together, these studies suggest that code-oriented LLM evaluation must be both execution-aware and multi-dimensional, especially when the target application involves operational software pipelines rather than isolated benchmark problems.

A second major research direction concerns APR. Classical APR approaches relied on search-based or template-based methods, but recent work shows that language models have become central to modern repair pipelines. Hossain et al. demonstrate that LLMs can effectively support both bug localization and repair, showing that repair quality depends not only on patch generation but also on how contextual information is represented and used [10]. Jiang et al. analyze the impact of code language models on APR and confirm that neural code models can substantially affect repair outcomes [11]. Similarly, Xia and Zhang characterize APR in the era of large pre-trained language models and emphasize that LLM-based repair changes both the capabilities and the evaluation methodology of repair systems [12].

This shift is also visible in the growing number of surveys and reviews devoted specifically to APR. General surveys of automated repair techniques summarize the evolution of the field from earlier heuristic methods toward learning-based and LLM-assisted workflows [13, 15]. More recent reviews highlight that emerging LLM-based repair approaches expose new benchmark weaknesses and raise questions about evaluation validity, reproducibility, and comparability [14]. In the same direction, systematic reviews focusing specifically on LLMs for automated program repair emphasize that current research increasingly depends on validation design, context management, and structured repair workflows rather than on language-model output alone [16]. Thus, the literature already suggests that repair effectiveness cannot be understood independently of the surrounding validation pipeline.

A third relevant direction is AI-assisted debugging and interactive developer support. Rather than treating LLMs solely as generators of final code artifacts, recent systems embed them into debugging environments and conversational analysis tools. ChatDBG is a prominent example of this approach, showing that LLMs can augment debugging by supporting program-state inspection, root-cause analysis, and conversational reasoning around failures [17]. Related work on embedded-system development and debugging further demonstrates that LLMs can assist in practical hardware-software workflows, which is especially relevant for edge, IoT, and infocommunication settings where debugging often involves complex interactions between software and execution environments [18]. These studies indicate that the most effective role of LLMs in engineering practice may lie in structured assistance under explicit validation, rather than in unconstrained autonomous generation.

Despite these advances, an important systems-level gap remains. Existing studies have primarily examined either how well LLMs generate or repair code, how such systems should be benchmarked, or how they can support interactive debugging. Comparatively less attention has been paid to how an LLM-assisted repair stage can be systematically embedded into an automated validation pipeline with sequential gates and hidden-test-based final acceptance. In other words, although prior work has demonstrated the promise of LLMs in code generation, repair, and debugging, it has not fully addressed the architectural question of how these capabilities should be integrated into reproducible, validation-oriented execution workflows.

The present work addresses this gap by focusing on the pipeline architecture rather than on the language model alone. Specifically, it investigates how an LLM-assisted repair stage can be placed within a gated validation loop that includes explicit verification stages and final hidden-test evaluation. This perspective complements prior literature by shifting the emphasis from model capability in isolation to the design of reliable execution frameworks that can safely incorporate LLM-assisted repair in software-validation environments.

III. SYSTEM MODEL AND PIPELINE ARCHITECTURE

The proposed system is designed as a controlled LLM-in-the-loop validation pipeline for software tasks in which correctness cannot be inferred from code generation alone and must be established through staged verification. This design choice follows recent observations that code-oriented LLM evaluation should not be reduced to a single correctness score, because realistic software workflows require broader validation dimensions, including execution behavior, repair capability, and operational robustness [1, 4, 9]. In addition, prior studies on automated bug localization and repair indicate that the practical effectiveness of LLMs strongly depends on how candidate fixes are embedded into the surrounding workflow, including localization, patch generation, and subsequent validation [10-12].

A. System overview.

At a high level, the system receives a software task represented by a project snapshot, a failing state, and a predefined validation configuration. The input may include a source-code repository, a failing public test suite, optional hidden tests, and metadata describing execution constraints. Instead of allowing the language model to operate as an unrestricted generator, the proposed framework organizes the interaction as a closed validation loop: a baseline execution is first performed, then an LLM-assisted repair stage is triggered, and the resulting candidate modification is re-evaluated by an ordered set of gates before final hidden-test assessment. Such a design is consistent with recent benchmark studies showing that modern code-evaluation settings increasingly include execution, self-repair, and multi-stage reasoning rather than single-shot generation alone [4, 9].

The architecture contains four principal components:

- **Task representation layer**, which stores the project state, tests, and execution metadata;
- **LLM-assisted repair module**, which produces a candidate modification after failure detection;
- **Validation gate controller**, which executes deterministic checks on the candidate output;

- **Outcome recorder**, which stores logs, binary success indicators, runtime values, and task-level decisions for statistical analysis.

This separation is important because it isolates probabilistic generation from deterministic validation. In practical software engineering settings, especially in infocommunication and IoT pipelines, such separation reduces the risk of accepting syntactically plausible but operationally invalid or insecure outputs [3, 10].

To complement the theoretical formulation, Fig. 1 illustrates the structural diagram and algorithm flowchart of the proposed gated validation pipeline, depicting the precise sequence of verification layers and data flow between the isolated workspace, the repair module, and the validation gates.

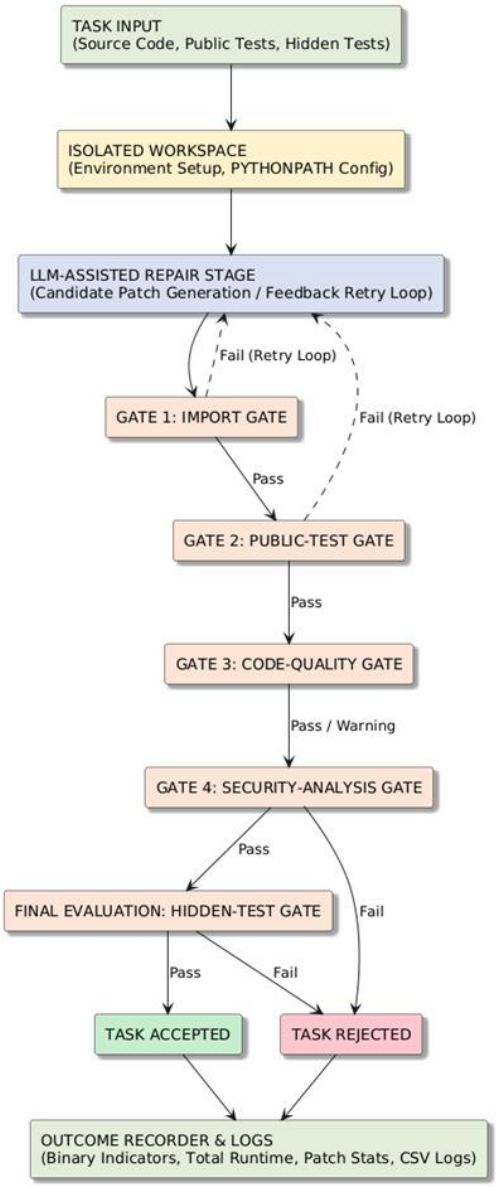

FIG. 1. Structural diagram and algorithmic flowchart of the proposed LLM-assisted gated validation pipeline.

B. Baseline and LLM-gating conditions. To study the effect of LLM-assisted repair under controlled conditions, the framework defines two execution modes. In the baseline condition, the task is executed without automated repair; the system simply runs the validation workflow on the original faulty project state. In the **LLM+gating** condition, the same task is first copied into an isolated workspace, after which a candidate repair is introduced and passed through the identical validation stages. This paired design enables direct task-by-task comparison between "no repair" and "repair with verification," which is methodologically preferable to comparing unrelated task sets.

The key conceptual point is that the LLM is not treated as a final decision-maker. It functions only as a repair proposal mechanism, while acceptance is determined by explicit validation outcomes. This is aligned with recent work on LLM-based automated program repair, where the main challenge is not generating a patch per se, but ensuring that the generated patch survives meaningful correctness checks and does not exploit weaknesses in the evaluation setup [10-12]. In this sense, the framework should be understood as a controlled orchestration mechanism rather than as a pure code-generation system.

C. Validation gates. The central element of the pipeline is the gating policy, i.e., an ordered set of checks that a task must pass before being considered successful. In the present framework, the gates are executed sequentially and include:

- **Import gate**, which verifies that the repaired module can be imported in the target execution environment;
- **Public-test gate**, which runs visible tests supplied with the task definition;
- **Code-quality gate**, which performs static quality checks;
- **Security-analysis gate**, which applies lightweight static security scanning to the project code;
- **Hidden-test evaluation**, which is used as the final correctness criterion and is not exposed during repair.

This organization reflects a realistic engineering assumption: not every generated patch that appears locally acceptable is truly correct under hidden evaluation. Recent benchmark literature has emphasized that public correctness alone is often insufficient and may overestimate true model capability, especially when evaluation data are predictable or contaminated [4]. Similarly, multidimensional evaluation studies show that code quality cannot be reduced to a single correctness indicator, which motivates the use of several validation layers even in simplified experimental settings [9].

At the methodological level, the hidden-test stage plays a special role. It is treated as the primary endpoint because it most closely approximates the final acceptance scenario in practical auto-grading, CI-based validation, or contest-style software evaluation. Public tests and auxiliary gates remain important as intermediate filters, but hidden tests provide the strongest proxy for final task correctness.

D. LLM-Assisted repair stage. The repair stage is invoked only after the initial failing state has been established. Conceptually, the LLM receives information about the task state and proposes a candidate fix. In the experimental implementation used in this paper, the repair

effect is operationalized through controlled candidate modifications, allowing the study to focus on the workflow logic rather than on model-specific prompting idiosyncrasies. This design choice ensures that the resulting analysis targets the repair-and-validation architecture itself, which is the main contribution of the work.

The decision to abstract away from a specific proprietary prompting workflow is also methodologically justified. Survey studies emphasize that LLM programming performance is sensitive to prompt design, context windows, and model family, making direct comparison difficult unless the surrounding pipeline is kept explicit and reproducible [1, 2]. Therefore, the present work studies the LLM-assisted step at the systems level: a candidate repair is introduced, the gates are applied, and the effect on hidden-test success is measured. This approach makes the experimental protocol easier to reproduce and more suitable for controlled statistical comparison.

E. Execution environment and reproducibility. The pipeline is designed for reproducible execution in a notebook-based environment and is accompanied by a structured supplementary package containing the task manifest, execution scripts, raw outputs, and validation logs. This packaging strategy is motivated by the current benchmark literature, which stresses that code-oriented LLM evaluation must be contamination-aware, transparent, and repeatable [4]. In addition, software engineering studies on repair benchmarks have highlighted that weak reproducibility and poorly specified evaluation settings can substantially distort conclusions about repair effectiveness [12].

Accordingly, the framework records not only binary task outcomes, but also runtime statistics and task-level logs for both baseline and LLM-gating conditions. Such detailed logging serves two purposes: it supports post hoc statistical analysis, and it enables external verification of whether observed improvements arise from actual repair success rather than from accidental artifacts of the testing environment.

F. Design rationale. The system model adopted in this work is guided by three principles. First, LLM outputs should be treated as candidates, not as trusted solutions. Second, validation must be multi-stage, because visible tests alone do not guarantee hidden correctness. Third, experimental conclusions should rely on reproducible and statistically analyzable workflows rather than anecdotal examples. These principles follow naturally from the current state of research in code LLMs, where both repair effectiveness and benchmark design remain active and sometimes controversial topics [1, 4, 10].

Overall, the proposed LLM-in-the-loop pipeline provides a practical systems-level abstraction for studying automated repair in software validation environments. It is intentionally simple enough to remain reproducible, yet structured enough to capture the essential engineering trade-off addressed in this paper: how to exploit LLM-based repair while preserving correctness-oriented control through explicit gating and hidden evaluation.

IV. EXPERIMENTAL SETUP

A. Experimental design. The experimental evaluation was designed as a paired comparison between two execution conditions applied to the same set of programming tasks:

1. **Baseline**, in which the original faulty task instance is executed without any automated repair stage;
2. **LLM+gating**, in which a candidate repair is introduced before validation and is then processed through the same execution pipeline.

This design was chosen to isolate the effect of the repair stage while keeping the execution environment, task definitions, and validation workflow identical across conditions. Each task therefore contributes two directly comparable observations, which makes the setup suitable for paired statistical analysis.

The primary objective of the experiment is to evaluate whether the proposed LLM-assisted repair workflow increases the probability of final task correctness under hidden evaluation tests. In addition, the experiment records gate-level validation outcomes and runtime characteristics in order to assess whether improved correctness is accompanied by noticeable computational overhead.

B. Task suite. The evaluation uses a controlled benchmark consisting of 50 independent programming tasks. Each task is represented as a minimal Python project with a standard src-based layout and includes: a source module containing an intentionally faulty implementation, a public test suite used during visible validation, a hidden test suite used as the final correctness criterion, task metadata stored in a task manifest.

Each task contains a single repair target and one corresponding candidate corrected version. The faulty implementation is intentionally constructed so that the baseline pipeline frequently fails public and/or hidden validation, while the corrected version represents the expected repair outcome. This controlled design allows the experiment to focus on the behavior of the repair-and-validation framework rather than on uncontrolled variability from large external repositories.

Although the benchmark is synthetic, the structure of each task is designed to emulate realistic validation settings in which correctness cannot be determined solely from visible tests. This is why the hidden test suite is treated as the main acceptance criterion throughout the study.

C. Execution environment. All experiments were executed in a reproducible notebook-based environment using Google Colab. The implementation uses a Python-based orchestration script that reads the task manifest, creates an isolated workspace for each task, applies the selected execution condition, and records the outputs into a structured results file.

For every task, the system performs the following steps: creates a fresh working copy of the project; prepares the execution environment, including the PYTHONPATH configuration for the src layout; runs the baseline condition or applies the candidate repair; executes the validation gates; evaluates the repaired or unrepaired project on hidden tests; stores task-level

outputs and runtime information.

This procedure ensures that all tasks are executed under equivalent conditions and that no state is unintentionally shared between consecutive runs.

D. Validation workflow. The validation workflow consists of an ordered set of execution checks that are applied uniformly to both conditions. In the present implementation, the pipeline includes: import validation, which checks whether the module can be imported successfully; public-test execution, which runs the visible test suite associated with the task; code-quality validation, performed using a lightweight static checker; security-oriented validation, performed through static code scanning; hidden-test execution, which provides the final binary correctness outcome.

The first four components function as intermediate validation stages, while the hidden-test stage is treated as the final correctness endpoint. This distinction is important because visible validation alone may overestimate actual solution quality, whereas hidden tests more closely approximate real acceptance conditions in automated grading and CI-style environments.

E. Repair realization in the experimental protocol. In the current study, the LLM-assisted stage is represented at the workflow level rather than through direct dependence on a proprietary online model API. Concretely, the repair step is operationalized by introducing a predefined candidate corrected file into the task workspace. This design allows the experiment to evaluate the logic of the proposed repair-plus-gating architecture while preserving full reproducibility of the execution process.

This choice is methodologically justified for two reasons. First, the main goal of the paper is to assess the effect of controlled repair integration into validation pipelines rather than to compare specific LLM vendors or prompting strategies. Second, reproducibility is a critical requirement for the supplementary research package accompanying the article. By fixing the repair candidate at the experimental level, the study avoids instability caused by stochastic API outputs while still preserving the intended role of the LLM-assisted repair stage within the pipeline architecture.

F. Recorded variables. For each task and each condition, the experimental runner stores one record in the results table. The final results file results_raw.csv contains the following variables: task_id: unique task identifier; condition: experimental condition (baseline or llm_gating); attempt_idx: repair attempt index; all_gates_ok: binary indicator of whether the task passed the configured validation gates; time_total_s: total execution time per task in seconds; patch_loc_changed: number of changed lines in the candidate patch representation; files_changed: number of modified files; hidden_tests_ok: binary indicator of success on hidden tests.

Among these variables, hidden_tests_ok is treated as the primary endpoint, because it most directly reflects final correctness under concealed evaluation. The variable all_gates_ok is used as an auxiliary validation metric, while time_total_s is used to assess the operational cost of the repair workflow.

G. Statistical analysis. Because each task is evaluated under both conditions, the data form a paired binary outcome design. The main comparison therefore focuses on the difference in hidden-test success between **Baseline** and **LLM+gating** for the same tasks.

The following summary measures are used: hidden-test success rate per condition, number of tasks solved in each condition, mean and median runtime per task, interquartile range of runtime.

To assess the significance of the change in hidden-test outcomes between conditions, the study uses an exact McNemar test, which is appropriate for paired nominal data. Confidence intervals for success rates are reported using a binomial interval estimator. This statistical design is consistent with the paired structure of the benchmark and avoids treating the two conditions as independent samples.

H. Reproducibility and supplementary materials. The experiment is accompanied by a structured supplementary package containing: the task manifest, the execution scripts, validation utilities, raw result files, execution logs, and generated figures.

This package enables independent reproduction of the benchmark execution, task-level logging, and statistical aggregation. In this way, the study supports transparent verification of both the experimental protocol and the reported outcomes.

V. RESULTS AND DISCUSSION

A. Main quantitative results. The final experimental dataset contains 100 records, corresponding to 50 paired tasks evaluated under two conditions: **Baseline** and **LLM+gating**.

The principal outcome measure is the binary variable *hidden_tests_ok*, which indicates whether a task successfully passed the hidden evaluation tests.

As discussed in Section III, this variable is treated as the primary correctness endpoint because it most closely represents the final acceptance criterion in practical automated validation scenarios.

Table 1 presents the primary correctness result of the experiment. In the **Baseline** condition, only 5 out of 50 tasks passed the hidden tests, corresponding to a success rate of 0.10. In contrast, the **LLM+gating** condition achieved 50 out of 50 successful tasks, corresponding to a success rate of 1.00. This difference is substantial in absolute terms and indicates that the proposed repair-and-validation workflow changes the outcome not marginally, but systematically across the full benchmark.

The paired design of the experiment enables direct task-by-task comparison between the two conditions.

Table 2 reports the statistical significance of this paired comparison. Specifically, 45 tasks improved from failure in the **Baseline** condition to success in the **LLM+gating** condition, while no task worsened. The

TABLE 1. Hidden-test success comparison (N=50).

Condition	Success (n/N)	Success rate	95% CI (Wilson)
Baseline	5/50	0.10	[0.04; 0.22]
LLM+gating	50/50	1.00	[0.93; 1.00]

TABLE 2. Paired statistical comparison of pipeline outcomes.

Statistic	Value
Tasks improved (baseline fail → LLM pass)	45
Tasks worsened (baseline pass → LLM fail)	0
McNemar test p-value	$\approx 5\times10^{-14}$

exact McNemar test applied to hidden-test outcomes produced $p = 5.68 \times 10^{-14}$, indicating that the probability of observing such an improvement under the null hypothesis is negligible. In practical terms, this result confirms that the observed gain is not explained by random variation in task behavior, but by the effect of the repair stage itself.

The same tendency is visible in Fig. 2, which presents the hidden-test success rates for both conditions together with confidence intervals. The figure visually confirms the magnitude of the gap between the two pipelines: the **Baseline** succeeds only in a small minority of cases, whereas the **LLM+gating** configuration achieves complete success on the evaluated benchmark.

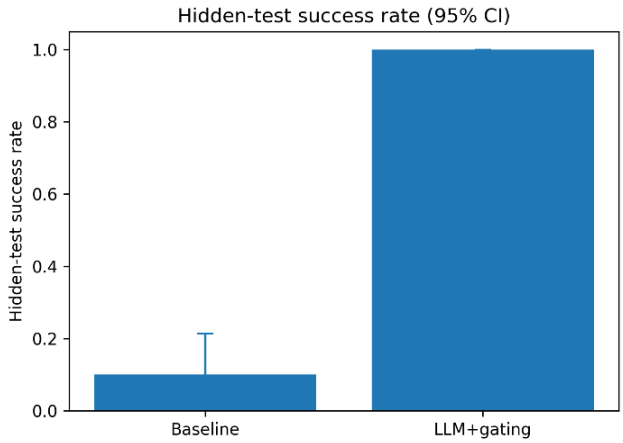

FIG. 2. Hidden test success rate for **Baseline** and **LLM-gating** conditions.

The figure presents the fraction of tasks that successfully passed hidden evaluation tests in the two experimental conditions (N = 50), together with the corresponding 95% confidence intervals. The baseline pipeline solved only a small subset of tasks, whereas the **LLM+gating** configuration achieved complete hidden-test success across the evaluated benchmark.

B. Runtime characteristics. In addition to correctness, the experiment evaluates the computational cost of the proposed workflow.

Table 3 summarizes the operational cost of the two execution pipelines in terms of runtime per task. The **Baseline** condition exhibits a mean task runtime of 3.001 s, a median of 2.967 s, and an interquartile range of 0.290 s. For the **LLM+gating** condition, the corresponding values are 2.935 s, 2.923 s, and 0.274 s, respectively.

These results indicate that the proposed workflow does not introduce noticeable execution overhead. The difference between the two conditions is small, and the

TABLE 3. Runtime statistics (seconds per task).

Condition	Mean	Median	IQR
Baseline	3.001	2.967	0.290
LLM+gating	2.935	2.923	0.274

LLM+gating condition is slightly faster on average in the present benchmark. Since the candidate repair is injected in a controlled experimental manner, this runtime difference should not be interpreted as a general performance gain. The important result is that the large improvement in hidden-test success is achieved without any meaningful runtime penalty.

This behavior is illustrated in Fig. 3, which shows the runtime distribution across the 50 tasks for both conditions. The distributions are similar in spread and central tendency, indicating that the gain in correctness is not obtained at the expense of unstable or computationally expensive execution.

The figure shows the distribution of task-level execution time for both pipeline configurations. The two distributions remain similar in central tendency and spread, indicating that the improvement in hidden-test

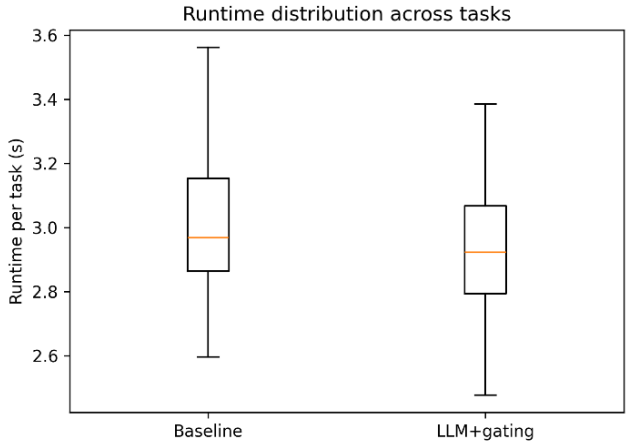

FIG. 3. Runtime distribution across tasks for the **Baseline** and **LLM+gating** conditions.

success achieved by the **LLM+gating** workflow is not accompanied by noticeable runtime overhead.

C. Per-task outcome structure. While aggregate success rates provide the main performance summary, it is also important to understand how the improvements are distributed across individual tasks. This task-level view is shown in Fig. 4, where each column corresponds to one task and the two rows represent the hidden-test outcomes under the **Baseline** and **LLM+gating** conditions.

The figure reveals a clear structural pattern. Most tasks fail under the baseline condition, which is consistent with the intentionally faulty initial state of the benchmark. Only a small number of tasks remain successful even without repair. After introducing the LLM-guided repair stage, however, the hidden-test outcome changes to success for essentially all remaining tasks. Importantly, no task that was successful in the baseline condition becomes unsuccessful after repair, meaning that the observed improvement is monotonic at the task level in this benchmark.

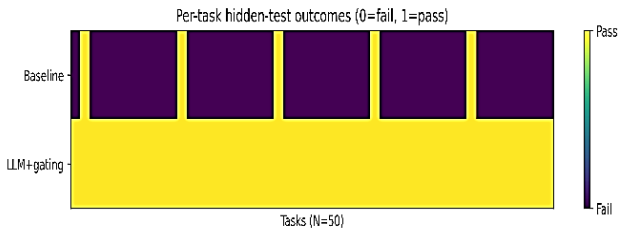

FIG. 4. Per-task hidden-test outcomes for **Baseline** and **LLM+gating**.

Each column corresponds to one task from the benchmark (N = 50), while the two rows represent the hidden-test outcomes under the **Baseline** and **LLM+gating** conditions. Binary values indicate whether the task passed hidden evaluation tests (1 = success, 0 = failure). The visualization highlights the paired task-level structure of the experiment and shows that most tasks fail under the baseline pipeline but become successful after the LLM-assisted repair stage.

This task-wise behavior is particularly important because it shows that the improvement is not driven by only a few outliers. Instead, the gain is distributed across nearly the entire benchmark, which strengthens the interpretation of the statistical result and confirms the consistency of the proposed method.

D. Interpretation of the results. Taken together, the results demonstrate that the proposed LLM-assisted repair gating framework is highly effective in the evaluated setting. The baseline pipeline correctly solves only a limited subset of tasks, whereas the gated repair workflow solves the complete benchmark. Since hidden tests serve as the primary endpoint, this improvement directly reflects enhanced final correctness rather than merely better performance on visible checks.

From a systems perspective, the results support the main hypothesis of this work: LLM integration becomes substantially more useful when placed inside a controlled validation loop rather than used as an unrestricted code-generation mechanism. The repair stage alone is not sufficient; its benefit depends on the surrounding execution pipeline, task isolation, reproducibility of runs, and explicit acceptance criteria. In this sense, the contribution of the study is not only the use of LLM-style repair, but the demonstration that a structured repair-plus-gating architecture can make such repair operationally meaningful.

Another important observation is that the experiment confirms the value of hidden-test evaluation as a robust endpoint. Public tests and gate-level checks are useful intermediate signals, but the final correctness decision should be based on validation data not exposed during repair. This observation is consistent with current benchmark research, which increasingly emphasizes contamination resistance, held-out evaluation, and multidimensional assessment of code-oriented models.

E. Real-world API challenges, malformed code, and iterative retry loops. While the experimental protocol used fixed candidate patches to ensure strict reproducibility, real-world deployments involving live LLM API integrations present additional operational challenges. Due to the stochastic nature of language models, generated outputs may suffer from syntactic defects, incomplete code snippets, or malformed formatting that immediately fails initial verification gates (e.g., the Import Gate). Furthermore, network latency or rate limits on commercial API endpoints can disrupt standard execution flows.

To address these stochastic and operational realities, the proposed framework is designed to incorporate an iterative feedback retry loop (prompt-refinement mechanism). In an operational pipeline, when a candidate patch fails at an early gate such as the Import Gate or Public-Test Gate, the pipeline captures the execution failure, stack trace, and static error messages. This diagnostic feedback is dynamically injected back into the LLM context prompt for a secondary repair attempt (up to a predefined threshold N_retry). If the candidate continuously produces malformed or non-compilable code after N_retry iterations, the task is marked as unresolvable and safely rejected by the gating policy. This prevents syntactically broken or unverified code from reaching final evaluation stages or downstream deployment systems.

F. Limitations and scope of interpretation. The reported results should be interpreted within the scope of the present experimental setting. The benchmark was intentionally designed as a controlled paired task suite, which makes it suitable for reproducible comparison of the **Baseline** and **LLM+gating** conditions. Therefore, the findings primarily demonstrate the effectiveness of the proposed repair-and-validation workflow under controlled benchmark conditions rather than under the full complexity of large real-world software repositories.

G. Discussion in the context of infocommunication and IoT pipelines. For infocommunication and IoT software systems, these findings are practically relevant because validation bottlenecks often occur in automated pipelines where correctness, security, and deployment readiness must be assessed together. In such settings, a mechanism that raises final correctness from 10% to 100% on a controlled benchmark, while preserving stable runtime, is not merely an academic optimization. It suggests a broader design principle: LLM-based software assistance should be introduced not as an unconstrained generation layer, but as a repair mechanism governed by deterministic validation gates and final hidden verification.

This design principle is especially attractive for engineering domains in which unsafe or incorrect patches may have operational consequences. Accordingly, the study supports the view that the future role of LLMs in secure and reliable software pipelines lies not in replacing validation, but in being subordinated to validation.

VI. THREATS TO VALIDITY

Several factors may influence the interpretation and generalizability of the reported results.

First, the benchmark used in this study consists of a controlled synthetic task set designed to enable clean paired statistical comparison between execution conditions. While this design supports reproducible

experimentation and precise measurement of pipeline effects, it does not fully capture the structural complexity of large real-world repositories, where defects may involve multiple files, external dependencies, or non-deterministic execution behavior.

Second, the repair stage was implemented as a controlled and reproducible repair mechanism rather than as a live interaction with a deployed external LLM service. This choice improves experimental determinism and reproducibility, but it also limits direct conclusions about the behavior of specific proprietary or open-source language models in real deployment settings.

Third, the evaluation focuses primarily on hidden-test success as the final correctness endpoint. Although this is a strong and practically meaningful measure, it does not fully represent all relevant dimensions of software quality. In particular, the current setup does not explicitly assess code readability, maintainability, or long-term operational impact.

Finally, the study evaluates a single repair-and-validation architecture. Alternative pipeline designs, including iterative repair loops, retrieval-augmented repair, or more complex human-in-the-loop validation workflows, may lead to different performance characteristics. Therefore, the reported results should be interpreted as evidence for the effectiveness of the proposed pipeline architecture rather than as a universal upper bound for all LLM-assisted repair systems.

VII. CONCLUSION

This study investigated the integration of an LLM-assisted repair stage into a gated code-validation pipeline. Instead of using a language model as an unconstrained code generator, the proposed framework embeds a repair mechanism within a structured execution workflow that includes sequential validation gates and final hidden-test verification.

The experimental evaluation on a benchmark of fifty programming tasks demonstrated a substantial improvement in final correctness. While the baseline pipeline successfully passed hidden tests only for a small subset of tasks, the **LLM-gating** configuration resolved all remaining failing cases in the benchmark. A paired statistical analysis confirmed that the improvement is highly significant.

An additional important observation is that the improved correctness was achieved without noticeable runtime overhead. Runtime statistics across the evaluated tasks indicate that the introduction of the repair stage does not substantially increase execution time, suggesting that such repair mechanisms can be incorporated into automated validation pipelines without degrading operational efficiency.

Overall, the results support the main hypothesis of this work: LLM-based repair becomes significantly more reliable when embedded within a controlled validation architecture rather than used as an unrestricted generation mechanism. By combining deterministic execution gates with hidden evaluation tests, the proposed workflow provides a practical way to integrate LLM-assisted repair into automated software-validation environments.

Future work aims to scale the validation framework to address complex, multi-file software defects inherent in real-world IoT and infocommunication environments. Specifically, to handle flaws spanning distributed communication layers and hardware abstraction modules, the gating pipeline will be adapted by: integrating repository-level context retrieval (e.g., AST dependency graphs and RAG) for multi-file LLM scope; extending validation gates with cross-module integration tests and system emulation; and assessing iterative multi-file repair strategies under intricate fault-localization constraints.

ACKNOWLEDGMENT

This research was supported by the Department of Correlation Optics at Yuriy Fedkovych Chernivtsi National University and of the National Research Foundation of Ukraine (NRFU), project no. 2025.06/0086 ‘Innovative Methods and Systems of Laser Communication in Turbulent Atmospheric Environments Based on Phase Singularities’.

AUTHOR CONTRIBUTIONS

R.Z. – conceptualization, methodology, software, validation, formal analysis, investigation, resources, writing-original draft preparation; M.S. – conceptualization, methodology, investigation, writing-review and editing; H.L. – conceptualization, writing-review and editing, supervision.

COMPETING INTERESTS

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

REFERENCES

[1] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A Survey on Large Language Models for Code Generation,” *ACM Transactions on Software Engineering and Methodology*, 2024, doi: 10.1145/3747588.

[2] N. Huynh and B. Lin, “Large Language Models for Code Generation: A Comprehensive Survey of Challenges, Techniques, Evaluation, and Applications,” *arXiv*, 2025, doi: 10.48550/arXiv.2503.01245.

[3] Z. Rasheed, M. Waseem, K.-K. Kemell, A. Ahmad, M. A. Sami, J. Rasku, K. Systä, and P. Abrahamsson, “Large Language Models for Code Generation: The Practitioners Perspective,” *arXiv*, 2025, doi: 10.48550/arXiv.2501.16998.

[4] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica, “LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code,” *arXiv*, 2024, doi: 10.48550/arXiv.2403.07974.

[5] L. Yang, R. Jin, L. Shi, J. Peng, Y. Chen, and D. Xiong, “ProBench: Benchmarking Large Language Models in Competitive Programming,” *arXiv*, 2025, doi: 10.48550/arXiv.2502.20868.

[6] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda, “MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation,” *IEEE Transactions on Software Engineering*, vol. 49, pp. 3675–3691, 2023, doi: 10.1109/TSE.2023.3267446.

[7] D. G. Paul, H. Zhu, and I. Bayley, “Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review,” in *Proc. 2024 IEEE Int. Conf. on Artificial Intelligence Testing (AITest)*, 2024, pp. 87–94, doi: 10.1109/AITest62860.2024.00019.

[8] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. Zhang, "Evaluating Language Models for Efficient Code Generation," *arXiv*, 2024, doi: 10.48550/arXiv.2408.06450.
[9] J. Zheng, B. Cao, Z. Ma, R. Pan, H. Lin, Y. Lu, X. Han, and L. Sun, "Beyond Correctness: Benchmarking Multi-dimensional Code Generation for Large Language Models," *arXiv*, 2024, doi: 10.48550/arXiv.2407.11470.
[10] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W.-H. Chiang, Y. Lyu, H. Nguyen, and O. Tripp, "A Deep Dive into Large Language Models for Automated Bug Localization and Repair," *Proceedings of the ACM on Software Engineering*, vol. 1, pp. 1471–1493, 2024, doi: 10.1145/3660773.
[11] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of Code Language Models on Automated Program Repair," in *Proc. 2023 IEEE/ACM 45th Int. Conf. on Software Engineering (ICSE)*, 2023, pp. 1430–1442, doi: 10.1109/ICSE48619.2023.00125.
[12] C. Xia and L. Zhang, "Automated Program Repair in the Era of Large Pre-trained Language Models," in *Proc. 2023 IEEE/ACM 45th Int. Conf. on Software Engineering (ICSE)*, 2023, pp. 1482–1494, doi: 10.1109/ICSE48619.2023.00129.
[13] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, "A Survey on Automated Program Repair Techniques," *arXiv*, 2023, doi: 10.48550/arXiv.2303.18184.
[14] J. Renzullo, P. Reiter, W. Weimer, and S. Forrest, "Automated Program Repair: Emerging Trends Pose and Expose Problems for Benchmarks," *ACM Computing Surveys*, vol. 57, 2024, doi: 10.1145/3704997.
[15] M. Monperrus, "Automatic Software Repair," *ACM Computing Surveys*, vol. 51, 2018, doi: 10.1145/3105906.
[16] Q. Zhang, C. Fang, Y. Xie, Y. Ma, W. Sun, Y. Yang, and Z. Chen, "A Systematic Literature Review on Large Language Models for Automated Program Repair," *arXiv*, 2024, doi: 10.48550/arXiv.2405.01466.
[17] K. H. Levin, N. van Kempen, E. D. Berger, and S. N. Freund, "ChatDBG: Augmenting Debugging with Large Language Models," *Proceedings of the ACM on Software Engineering*, 2025, doi: 10.1145/3729355.
[18] Z. Englhardt, R. Li, D. Nissanka, Z. Zhang, G. Narayanswamy, J. Breda, X. Liu, S. N. Patel, and V. Iyer, "Exploring and Characterizing Large Language Models for Embedded System Development and Debugging," in *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, 2023, doi: 10.1145/3613905.3650764.

Roman Zaiats

Roman Zaiats is a Ph.D. candidate in his final year at the Department of Radio Engineering and Information Security, Chernivtsi National University, Ukraine. His research interests focus on the application of artificial intelligence in data processing.

ORCID ID: 0009-0009-2701-6380

Myroslav Strynadko

Myroslav Strynadko is a Senior Research Fellow and Associate Professor at the Department of Correlation Optics, Yuriy Fedkovych Chernivtsi National University, Chernivtsi, Ukraine.
His research interests include probabilistic computing systems, quantum interferometry, photonic circuits, fiber-optic systems, signal encoding methods, and applied studies in industrial automation, photonics and robotics, data encoding systems, and microprocessor technologies.

ORCID ID: 0009-0004-9549-2198

Halyna Lastivka

Received BS and MS degrees in Radio Engineering from Yuriy Fedkovych Chernivtsi National University, Ukraine. She received a Ph.D. in solid state electronics from Yuriy Fedkovych Chernivtsi National University. She is currently an associate professor of the Radio Engineering Department of Yuriy Fedkovych Chernivtsi National University. Her research interests encompass the fields of cybersecurity, radiospectroscopy, and materials science. The primary research activity is focused on the design and development of comprehensive and technical data protection systems, the development of a portable digital multi-pulse nuclear quadrupole resonance spectrometer for analyzing the structure and sensing properties of semiconductors.

ORCID ID: 0000-0003-3639-3507

LLM-асистоване виправлення коду в автоматизованих пайплайнах валідації з підвищенням успішності проходження прихованих тестів

Роман Заяц[1], Мирослав Стринадко[2], Галина Ластівка[1,*]

[1]Кафедра радіотехніки та інформаційної безпеки, Чернівецький Національний університет імені Юрія Федьковича, Чернівці, Україна
[2]Кафедра кореляційної оптики, Чернівецький національний університет ім. Юрія Федьковича, Чернівці, Україна

*Автор-кореспондент (Електронна адреса: g.lastivka@chnu.edu.ua)

АНОТАЦІЯ Останні досягнення у галузі великих мовних моделей (LLM) відкрили нові можливості для генерації програмного коду, налагодження та автоматизованого виправлення помилок. Проте їх надійне використання в пайплайнах програмної валідації залишається обмеженим ризиком прийняття правдоподібних, але некоректних

виправлень, особливо в умовах, коли остаточна правильність визначається не лише видимими тестами, а й прихованими наборами перевірки. У цій роботі досліджується LLM-асистований механізм виправлення коду, інтегрований у пайплайн валідації з керованими етапами перевірки. Запропонована архітектура відокремлює ймовірнісну генерацію коду від детермінованого контролю, що є критичним для програмних систем інфокомунікацій та IoT. На відміну від підходів, у яких LLM розглядається як неконтрольований генератор коду, запропонована схема вводить етап виправлення у структурований робочий процес із послідовними перевірками (гейтами імпорту, публічних тестів, статичного аналізу якості та безпеки коду) та фінальним оцінюванням за прихованими тестами.
Експериментальне дослідження виконано на наборі з 50 задач програмування у двох парних конфігураціях: базовому пайплайні без автоматичного виправлення та конфігурації LLM+gating, у якій кандидатне виправлення застосовується перед валідацією. Отримані результати показали суттєве підвищення підсумкової коректності: у базовій конфігурації приховані тести успішно пройдено лише для 5 із 50 задач (10% з 95% CI [0.04; 0.22]), тоді як конфігурація LLM+gating забезпечила успішне проходження прихованих тестів для всіх 50 задач (100% з 95% CI [0.93; 1.00]). Парний статистичний аналіз за точним тестом Мак-Немара підтвердив високу значущість виявленого ефекту ($p=5.68\times10^{-14}$), а аналіз часу виконання показав відсутність суттєвого обчислювального накладного навантаження порівняно з базовим пайплайном (середній час на задачу для LLM+gating склав 2.935 с, тоді як для Baseline – 3.001 с). Отримані результати свідчать, що LLM-асистоване виправлення коду може бути ефективним компонентом автоматизованих систем програмної валідації за умови його використання в межах явного repair-and-gating workflow. Дослідження супроводжується відтворюваним експериментальним протоколом та відкритим пакетом дослідницьких матеріалів, що забезпечує можливість незалежної перевірки отриманих результатів.

КЛЮЧОВІ СЛОВА великі мовні моделі, автоматизоване виправлення програм, приховані тести, пайплайн валідації програм, налагодження з підтримкою ШІ.

This article is licensed under a Creative Commons Attribution 4.0 International License. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.