Improving Static Code Analysis Using a Consensus-Based Multi-LLM*

Augustinas Labutis[1,*,†], Jonas Čeponis[1,†], Šarūnas Grigaliūnas[1,†] and Rasa Brūzgienė[1,†]

[1]Department of Computer Sciences, Kaunas University of Technology, Kaunas LT-51368, Lithuania

Abstract

Static code analysis is a prevailing technique for enhancing software quality and security. It facilitates the identification of defects, vulnerabilities, and maintainability issues without requiring program execution, thereby ensuring the integrity and reliability of the software under scrutiny. However, its practical application is still constrained by false positives, limited semantic understanding, and reduced precision when applied to large and evolving codebases. This research proposes a consensus-based approach to improving static code analysis with the support of multiple large language models (LLMs). The proposed approach diverges from the conventional treatment of automated code repair and warning interpretation as the output of a single model. Instead, it frames these processes as a comparative process in which several independently generated repair candidates are assessed in relation to the same diagnostic issue. The approach commences with repository-level diagnostics extracted from the target codebase and transformed into structured repair tasks. Subsequently, multiple LLMs are engaged in the generation of alternative candidate fixes for the identified problem. Furthermore, the method incorporates a historical grounding mechanism in which previously faulty and corrected code pairs are compared to estimate the expected scale of code modification for a given diagnostic type. This grounding percentage functions as an empirical reference point for evaluating the proportionality and structural plausibility of generated fixes. Candidate repairs are further examined through validation criteria such as compilation outcome, diagnostic resolution, and the absence of newly introduced warnings or errors. The final result is determined through a consensus-based ranking that combines grounding consistency and convergence between repairs generated independently. This approach is intended to provide a more reliable basis for static analysis support.

Keywords

Static code analysis, large language models, consensus algorithms, multi-agent systems, automated program repair, software quality, code diagnostics

CEUR-WS.org/Vol-4309/paper11.pdf

1. Introduction

Large language models are increasingly being applied to software engineering tasks, including source code generation, defect correction, and support for static code analysis. However, when employed to address compiler warnings and diagnostics, single-model solutions frequently prove unreliable. Such solutions may produce inconsistent outputs, invalid code modifications, unnecessarily broad changes, or fixes that introduce new issues into the codebase. This challenge is particularly important in the context of static code analysis improvement, where the usefulness of a generated correction depends not only on its plausibility, but also on its ability to resolve the identified issue in a proportionate and technically valid manner. Prior research has shown that static analysis remains limited by incomplete semantic understanding and practical precision constraints in complex systems [1, 2], while automated repair methods and LLM-based correction can struggle when defects require broader contextual reasoning or stronger validation [3, 4].

The scientific problem addressed in this paper is how to improve the reliability of LLM-assisted remediation of static code analysis findings when single-model generated fixes may appear plausible, yet remain technically invalid, disproportionately broad, or inconsistent with historically observed

TKTP 2026: Annual Doctoral Symposium of Computer Science, 8.—9.6.2026 Turku, Finland

*Corresponding author.

†These authors contributed equally.

augustinas.labutis@ktu.lt (A. Labutis); jonas.ceponis@ktu.lt (J. Čeponis); sarunas.grigaliunas@ktu.lt (Š. Grigaliūnas); rasa.bruzgiene@ktu.lt (R. Brūzgienė)

0009-0006-6775-8424 (A. Labutis); 0009-0006-9206-8361 (J. Čeponis); 0000-0001-9268-9244 (Š. Grigaliūnas); 0000-0002-0816-8700 (R. Brūzgienė)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-02

repair behavior. In response to this problem, the present paper proposes a consensus-based multi-LLM approach for improving static code analysis. The novelty of the proposed approach lies in the integration of four elements into a single structured workflow: (i) historical grounding percentages derived from before-and-after correction pairs for a selected diagnostic type, (ii) independent candidate fix generation by multiple large language models for the same diagnostic issue, (iii) recompilation- and diagnostic-based validation of each generated candidate, and (iv) consensus-driven final selection based on both technical validity and grounding alignment. In this way, this work contributes a validation-oriented and proportion-aware framework for more trustworthy static-analysis remediation. The use of consensus is motivated by recent work showing that multi-agent and consensus-based decision mechanisms can improve robustness when several independently generated outputs must be compared and selected [5, 6].

To investigate this idea, a prototype system was implemented that retrieves repositories, extracts diagnostics, generates multiple candidate repairs, validates them through recompilation, and evaluates whether sufficient agreement exists to support a reliable final fix. In this paper, we present a structured methodology for LLM-assisted remediation of static analysis findings. We have implemented a prototype that realizes the proposed workflow. In addition, we have developed an experimental setup for assessing the feasibility of validation- and consensus-based repair selection. The study was conducted as a collaborative research endeavor, with the authors contributing to the conceptualization of the method, the design and implementation of the prototype, the preparation of the experimental framework, and the analysis and presentation of the obtained results.

The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the proposed method. Section 4 describes the experimental setup and discusses the obtained results. Finally, Section 5 concludes the paper and outlines future work.

2. Related Work

Static code analysis is a software quality and security enhancement technique that involves the examination of source code without executing the program. The program's strengths lie in its capacity for early defect detection and its ability to provide extensive code coverage, often through representations such as abstract syntax trees, control-flow graphs, and value-flow models [3, 4]. However, prior studies also demonstrate that static analysis is limited by conservative approximations, which may result in false positives and incomplete semantic understanding, particularly in large and complex systems [3, 4]. Empirical evaluations further indicate that no single static analysis tool provides consistently complete detection, and that practical effectiveness often decreases when methods are applied to real-world projects rather than curated benchmarks [2, 7].

Consequently, researchers have explored complementary approaches, including dynamic and hybrid analysis. Dynamic analysis, on the other hand, observes runtime behavior and can reveal issues that remain invisible to purely static reasoning. However, it is itself limited by test coverage and monitoring overhead [5]. Hybrid approaches endeavor to integrate the extensive scope of static analysis with the enhanced contextual precision of runtime validation [5, 6]. This general direction is salient because it demonstrates that analysis results become more reliable when they are not accepted directly, but rather subjected to an additional validation stage.

The field of automated program repair provides a significant foundation for this research. Recent literature demonstrates the efficacy of APR methods in addressing recurring and structurally simple defects, particularly when correction patterns exhibit repetitive characteristics and are adequately represented [8]. Concurrently, the efficacy of repairs declines for semantically intricate issues necessitating profound contextual or non-local reasoning [8]. Large language models have expanded this area significantly, as they are capable of generating code fixes from both source context and textual diagnostics [9]. However, extant studies also emphasize that LLM-generated repairs remain uncertain. They are sensitive to prompting, may produce disproportionate changes, and cannot be assumed correct without external validation [9].

Recent research has also begun to integrate LLMs more directly into software analysis. LLM-enhanced static analysis has been proposed as a means to enhance semantic reasoning beyond that achievable with traditional rule-based tools. However, extant approaches still face challenges with long-range dependencies, complex logic, and real-world scale [10]. In a similar vein, studies that integrate structured static information with dynamic symbolic traces have indicated that more robust results can be attained by incorporating multiple sources of evidence [11]. These studies suggest that richer evidence can enhance analysis, but they also demonstrate that identifying the most trustworthy outcome remains a challenging problem.

This challenge has thus motivated interest in multi-agent and consensus-based systems. LLM-based multi-agent architectures distribute reasoning across multiple agents and have been proposed for software engineering tasks such as code generation, testing, and repair [12]. As demonstrated by related studies, the implementation of voting and consensus-oriented decision protocols has been shown to enhance performance by aggregating multiple outputs produced independently [13]. In contrast, more advanced mechanisms are designed to achieve greater reliability and fault tolerance in coordinated agent systems [8]. However, the extant literature does not yet provide a sufficiently established framework for applying consensus-based multi-LLM reasoning specifically to static-analysis repair selection under validation-driven criteria. Consequently, the proposed work is situated at the intersection of static analysis, automated repair, LLM-based code generation, and consensus-oriented decision making.

3. Proposed method

3.1. Method overview

The proposed method is a multi-stage workflow designed to improve static code analysis remediation by using a consensus-based evaluation of multiple, independently generated code fixes (Figure 1). Rather than relying on a single model response, the method treats automated repair as a comparative process. Several candidate solutions are generated, validated, and ranked before a final decision is made. The method's central objective is to combine historical repair evidence, repository-level diagnostic extraction, candidate generation, empirical validation, and consensus-oriented selection into one structured correction process.

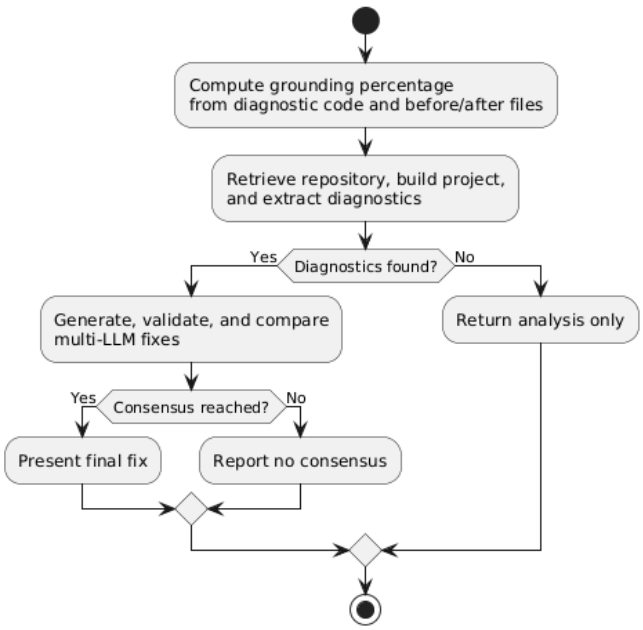


Figure 1: Overall proposed method workflow.

The method is based on the assumption that a generated fix should not be accepted solely because it appears plausible at the textual level. In practical software correction scenarios, especially those related to static code analysis findings, the usefulness of a repair depends on more than just readability. A candidate solution must be proportionate to the issue being addressed, not introduce unnecessary modifications, and be technically verifiable within the analyzed codebase. For this reason, the method introduces several control stages that transform repair generation into a traceable decision-making process rather than a one-step generation task.

At a high level, the method begins by establishing a baseline for a selected diagnostic type. Then, it retrieves and analyzes a repository, extracts compiler diagnostics, and constructs a structured repair task. The same task is provided to multiple large language models, which generate candidate fixes independently. These candidates are then applied and validated through recompilation and diagnostic checking. Finally, the system evaluates whether the results satisfy the consensus-based acceptance requirements.

In this architecture, each stage plays a specific role. Historical grounding provides a baseline for expected change magnitude. Repository analysis ensures that repair generation is based on actual diagnostics and code context. Multi-model generation produces several alternative correction strategies for the same issue. Validation filters out technically unsuccessful or harmful candidates. Consensus evaluation determines if the remaining solutions exhibit sufficient agreement and reliability to justify selecting a final result. By combining these stages, the method aims to enhance the reliability of LLM-assisted static analysis remediation.

3.2. Historical grounding and repository analysis

The initial phase of the method involves the establishment of a historical grounding model, which functions as a reference point for the anticipated scale of code modification associated with a specific diagnostic type (Figure 2). This stage necessitates three essential inputs: a diagnostic or warning code, a source file containing the issue prior to correction, and the corresponding corrected version of the same file post-repair. By comparing the two file versions, the method determines the extent to which the source code has been altered during a known valid correction process.

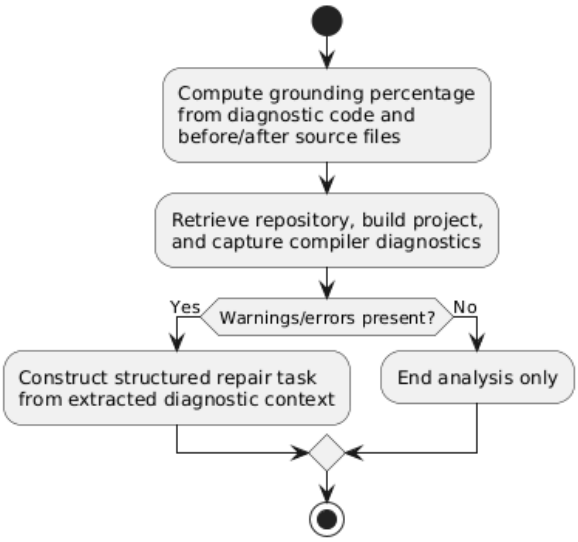


Figure 2: Historical grounding and repository analysis

The result of this comparison is expressed as a grounding percentage. This percentage indicates the proportion of code modifications made to the original version in comparison with the corrected version. Its purpose is not to demonstrate the semantic correctness of a future repair. Instead, it functions as a

historical baseline, indicating the typical magnitude of structural change associated with a repair for the selected diagnostic category.

Subsequent to the establishment of the grounding baseline, the method advances to the analysis of repositories and the extraction of diagnostic elements. A repository is retrieved, built in a controlled environment, and examined for compiler output. The resulting diagnostics, including warning or error codes, file paths, and relevant source locations, are then used to form a structured repair task that combines the identified issue with its source code context.

This integrated stage, therefore, introduces the historical and repository-specific context necessary for the execution of repairs. The concept of a proportionate fix is defined by its historical underpinnings, while repository analysis serves to identify the present issue that necessitates resolution. Consequently, these two elements provide the informational basis on which all subsequent candidate generation and evaluation depend.

3.3. Multi-LLM candidate generation and validation

Subsequent to the formulation of a structured repair task, the method advances to the generation of candidates through the utilization of multiple large language models (Figure 3). Each selected model undergoes the same repair task, including the diagnostic information and the corresponding code context. Each model then generates its own candidate fix, which may differ in implementation details, modification scope, or repair strategy.

This stage of the process conceptualizes code repair as a comparative generation problem. Rather than presupposing the reliability of a single response, the method deliberately solicits multiple answers, which can subsequently be evaluated against objective standards. The diversity of outputs is advantageous because it reveals whether independent model responses converge toward similar repair patterns or differ substantially.

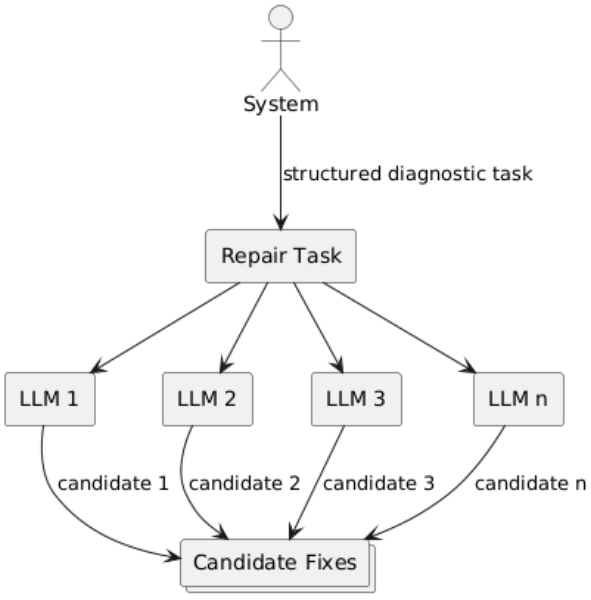


Figure 3: Multi-LLM candidate generation and validation.

Subsequent to the generation of candidate fixes, the method undertakes a validation stage to evaluate each of them. For each candidate, the generated modification is applied to the repository, and the project is subsequently rebuilt. The validation process meticulously documents three key outcomes: first, whether the build successfully completes; second, whether the target diagnostic is reduced or eliminated; and third, whether any novel warnings or errors emerge as a consequence of the change.

Candidates who do not successfully complete the compilation process are considered unsuccessful and are subsequently excluded from further selection or assigned a very low priority. Candidates that

compile successfully but do not improve the original diagnostic are also considered weak solutions. In this manner, validation functions (Figure 4) as the primary quality filter in the workflow, thereby ensuring that the subsequent consensus stage encompasses only technically meaningful outputs.

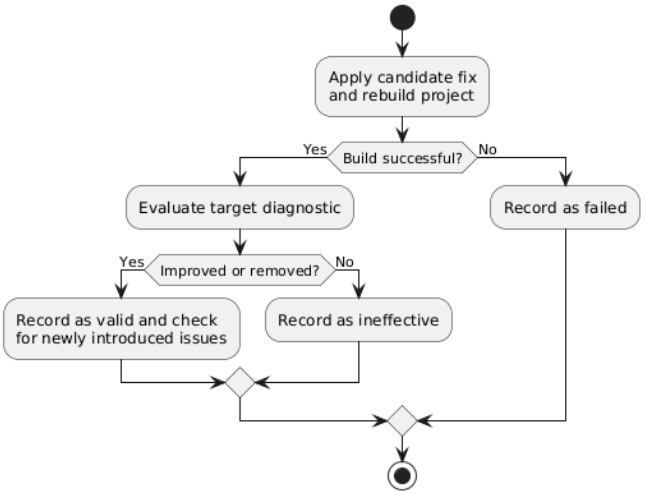


Figure 4: Candidate validation process.

3.4. Grounding-based comparison, consensus, and final selection

Subsequent to validation, each remaining candidate is evaluated according to its modification magnitude. This is achieved by calculating the percentage of code altered by the generated fix and comparing that value with the historical grounding percentage previously established (Figure 5). The objective of this comparison is to ascertain whether the structural scale of the candidate repair remains aligned with previously observed valid corrections for the same diagnostic type.

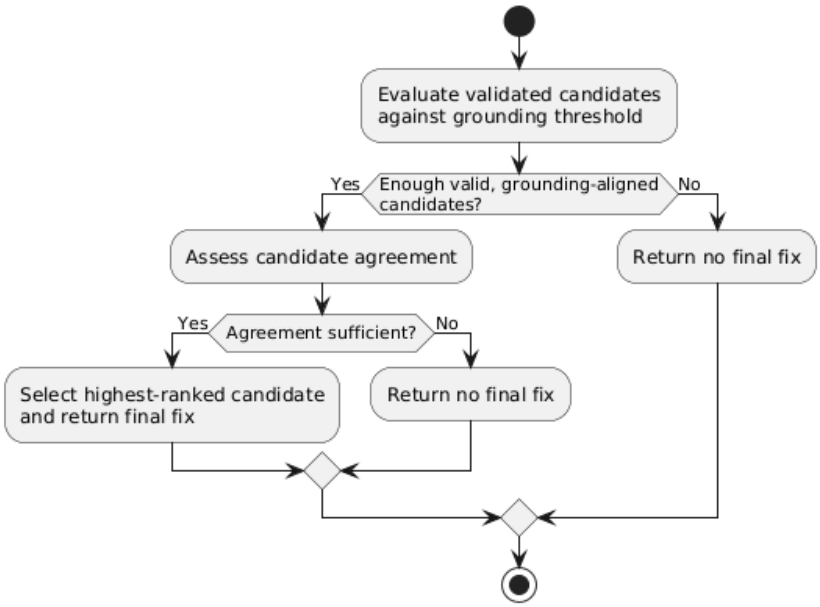


Figure 5: Grounding comparison and consensus selection.

A candidate whose modification percentage remains within the acceptable range suggested by the grounding baseline is considered structurally consistent with historical repair behavior. Conversely, if a

candidate introduces a substantially greater amount of change than expected, the method interprets this as a sign that the repair may be overextended and therefore less suitable for final selection.

The culminating stage of the method is consensus-based selection. The term ”consensus” is not defined as exact textual identity between candidate fixes; rather, it is defined as convergence among candidates according to shared acceptability criteria. A candidate contributes to consensus when it passes build-based validation, remains aligned with grounding expectations, and does not strongly contradict the successful correction patterns exhibited by other validated candidates.

In the event that multiple candidates, generated independently, satisfy these conditions, the method interprets this convergence as a signal of increased reliability and selects the highest-ranked candidate as the final result. In instances where the candidates exhibit significant disparities in terms of build success, diagnostic improvement, or modification magnitude, it is deemed that a consensus has not been reached. Consequently, the system generates a report indicating that no adequately reliable final fix could be identified under the prevailing criteria.

4. Experimental Setup and Results

4.1. Experimental objective

The experimental evaluation was designed to assess the practical operation of the proposed consensus-based method for improving static code analysis findings. The primary objective of this study was to ascertain the capacity of the implemented prototype to execute the complete workflow, commencing with historical grounding calculation and culminating in repository retrieval, diagnostic extraction, multi-LLM candidate generation, recompilation-based validation, and final consensus determination. Consequently, the experiment appraises not solely the conceptual framework of the method, but also its viability in an operational prototype environment.

A secondary objective was to ascertain whether the proposed method facilitates a more meticulous repair process than relying on a solitary generated fix. Consequently, the evaluation emphasizes observable technical outcomes, including successful repository analysis, accurate extraction of compiler diagnostics, generation of multiple candidate fixes for the same issue, successful recompilation of those fixes, and their alignment with the expected grounding range. The experiment, therefore, treats repair quality as a combination of technical correctness, structural proportionality, and agreement across independently generated candidates.

4.2. Experimental environment

The experiment was conducted using the implemented prototype in a local execution environment. The prototype was designed as an integrated system supporting grounding pattern ingestion, repository analysis, compiler diagnostic extraction, candidate fix generation, recompilation-based validation, and consensus evaluation. This configuration enabled the implementation of the same workflow on a repetitive basis for selected repositories and diagnostic cases under controlled conditions.

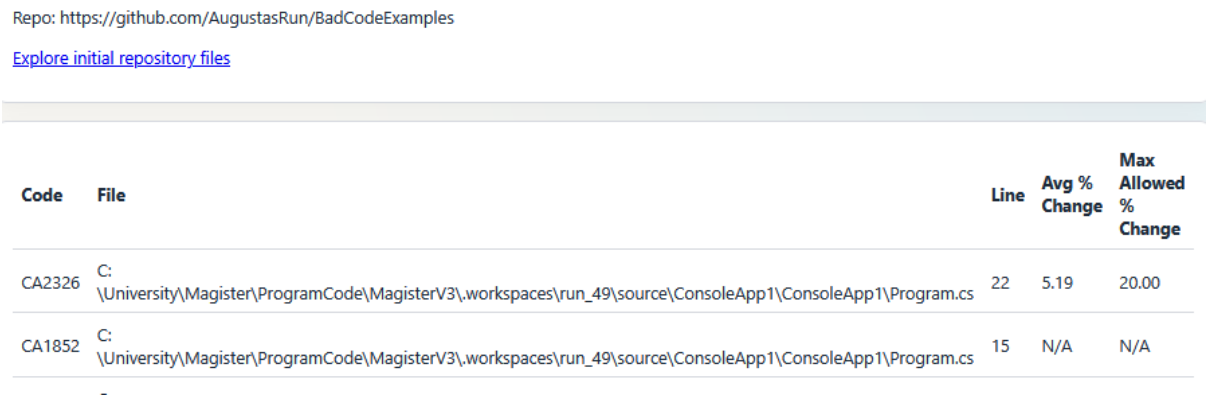
The experimental setup consisted of two main input categories. The initial category encompassed historical correction exemplars employed to ascertain grounding percentages. These examples were derived from source code files prior to and following known corrections for a selected diagnostic code. The second category comprised source code repositories submitted for analysis. Subsequent to the provision of a repository, the prototype retrieved the repository, built the project, extracted compiler warnings and errors, and prepared structured repair tasks for further processing by multiple large language models.

4.3. Experimental procedure

The experiment was conducted in accordance with a series of steps that corresponded to the primary stages of the proposed method. Initially, a historical repair pattern was established by submitting

a diagnostic code along with the original and corrected versions of a source file. Subsequently, the prototype calculated the grounding percentage and stored it as the historical baseline for subsequent comparison. This step ensured that future generated fixes could be assessed not only by compilation outcome, but also by the relative magnitude of code modification.

Subsequently, a repository was submitted for analysis. The prototype retrieved the repository, initiated the build process, and collected compiler diagnostics from the build output. Preliminary diagnostic findings were used to identify the issue type, the associated source file, and the relevant code context (Figure 6). These elements were then combined into a structured repair task, which served as the input for automated code correction.



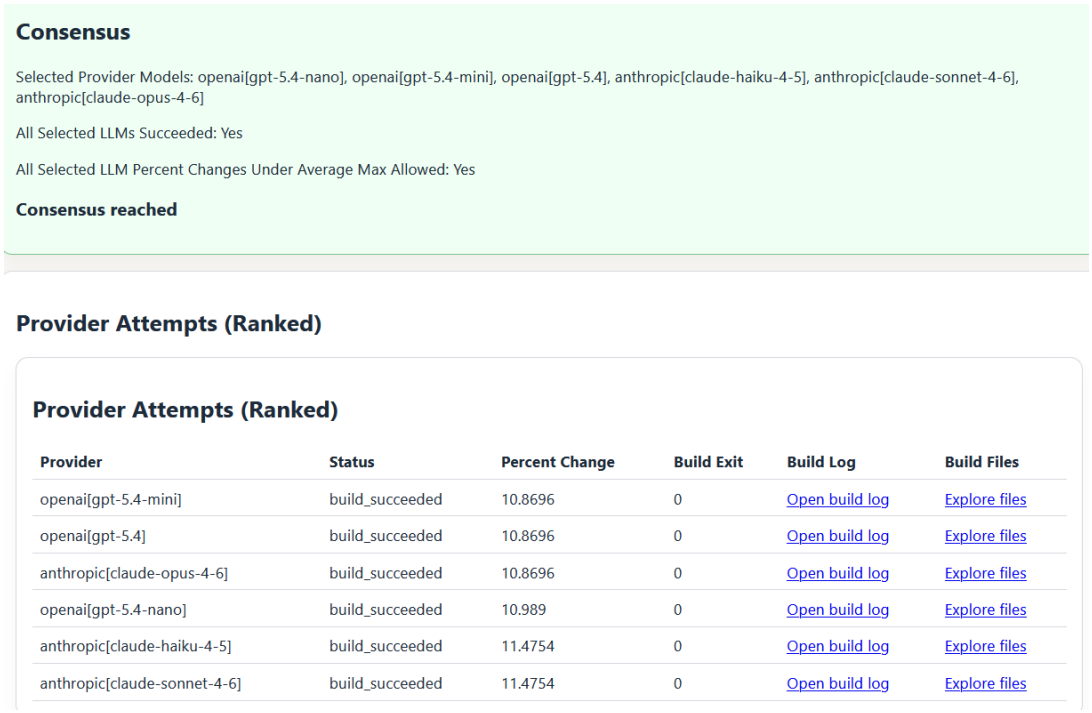
Repo: https://github.com/AugustasRun/BadCodeExamples

Explore initial repository files

Code	File	Line	Avg % Change	Max Allowed % Change
CA2326	C:\University\Magister\ProgramCode\MagisterV3\.workspaces\run_49\source\ConsoleApp1\ConsoleApp1\Program.cs	22	5.19	20.00
CA1852	C:\University\Magister\ProgramCode\MagisterV3\.workspaces\run_49\source\ConsoleApp1\ConsoleApp1\Program.cs	15	N/A	N/A

Figure 6: Compiler build and warning gathering results.

The structured repair task was disseminated to a number of large language models that had been selected for this purpose. Each model independently generated a candidate fix for the same diagnostic problem. Each generated candidate was then validated (Figure 7). by applying the proposed change to the repository and rebuilding the project. The validation process documented three outcomes: i) the



Consensus

Selected Provider Models: openai[gpt-5.4-nano], openai[gpt-5.4-mini], openai[gpt-5.4], anthropic[claude-haiku-4-5], anthropic[claude-sonnet-4-6], anthropic[claude-opus-4-6]

All Selected LLMs Succeeded: Yes

All Selected LLM Percent Changes Under Average Max Allowed: Yes

Consensus reached

Provider Attempts (Ranked)

Provider Attempts (Ranked)

Provider	Status	Percent Change	Build Exit	Build Log	Build Files
openai[gpt-5.4-mini]	build_succeeded	10.8696	0	Open build log	Explore files
openai[gpt-5.4]	build_succeeded	10.8696	0	Open build log	Explore files
anthropic[claude-opus-4-6]	build_succeeded	10.8696	0	Open build log	Explore files
openai[gpt-5.4-nano]	build_succeeded	10.989	0	Open build log	Explore files
anthropic[claude-haiku-4-5]	build_succeeded	11.4754	0	Open build log	Explore files
anthropic[claude-sonnet-4-6]	build_succeeded	11.4754	0	Open build log	Explore files

Figure 7: LLM suggestion and consensus result.

success of the compilation process; ii) the improvement or resolution of the target diagnostic; and iii) the appearance of additional warnings or errors following modification. The percentage of code change introduced by each candidate was calculated and compared with the historical grounding percentage.

The prototype's evaluation process sought to ascertain the potential for consensus among the participants. The determination of consensus was made by evaluating the technical success of the generated candidates and the modification percentages of these candidates, which were compared to the acceptable range defined by the grounding baseline. In the event that the stipulated conditions were met, the system issued a report indicating that consensus had been achieved (Figure 7). Conversely, the prototype indicated that a consensus had not been reached.

4.4. Evaluation criteria and metrics

The experimental evaluation was based on four practical criteria derived from the intended behavior of the proposed method. The initial criterion pertained to the successful execution of grounding calculation, which was employed to ascertain the system's capacity to accurately compute and store the grounding percentage under specific conditions. These conditions entailed the provision of valid historical input files and a diagnostic code. The second criterion was successful repository analysis, which included repository retrieval, build execution, and the extraction of compiler diagnostics. These criteria establish whether the prototype can create the historical and repository-level context required for the generation of repairs. The third criterion pertained to the successful candidate generation and validation. The objective of this criterion was to ascertain whether multiple large language models could independently generate fixes for the same issue and whether those fixes could be validated by recompilation. The primary metrics at this stage included the success of candidate generation, the success of the build following modification, the degree of diagnostic improvement, and the presence or absence of newly introduced issues. The fourth criterion pertained to grounding alignment and consensus achievement. For each validated candidate, the experiment measured the percentage of code changed and compared it with the historical grounding threshold. Candidates whose change magnitude remained within the acceptable range were treated as structurally aligned with historical repair patterns.

The final metric was determined by the presence or absence of consensus among the selected candidates. Consequently, the evaluation metrics encompassed not solely the efficacy of the repair, but also its proportionality and the extent to which it was supported by multiple independent model outputs.

4.5. Experiment Results

The results of the prototype demonstrate the successful execution of the proposed workflow for all selected diagnostic cases. As demonstrated in (Table 1), the evaluation of the models revealed that they

Table 1
Multi-LLM repair results for selected C# diagnostics.

Diagnostic code	Model	Build ok	Fixed issue	New diagnostic code	Change %
CA1826	gpt-5.4	+	+	1	10.86%
CA1826	claude-sonnet-4-6	+	+	1	10.86%
CA1826	claude-opus-4-6	+	+	1	11.47%
CA2100	gpt-5.4	+	+	0	31.01%
CA2100	claude-sonnet-4-6	+	+	0	14.64%
CA2100	claude-opus-4-6	+	+	0	15.01%
CA2326	gpt-5.4	+	+	1	22.01%
CA2326	claude-sonnet-4-6	+	+	0	6.98%
CA2326	claude-opus-4-6	+	+	0	2.41%

were capable of producing compilable fixes for the selected C# diagnostics.

In every instance where the models were tested, the target issue was marked as resolved. This finding suggests that, at the fundamental technical level, the prototype exhibited the capacity to retrieve the issue context, generate candidate repairs, apply them, and validate them through recompilation. The findings substantiate the hypothesis that the implemented system is capable of facilitating end-to-end automated remediation for static analysis findings.

Concurrently, the findings also demonstrate substantial disparities in the structural behavior exhibited by the generated fixes. For diagnostic code CA1826, the three models exhibited comparable change percentages, ranging from 10.86% to 11.47%, with all remaining within the accepted grounding range. Consequently, a consensus was reached for this case, as demonstrated in (Table 2). This finding indicates that, for relatively straightforward diagnostics, independently generated fixes tend to converge towards a similar scale of correction. Consequently, these fixes can be regarded as mutually reinforcing repair candidates.

Table 2
Consensus evaluation results for selected C# diagnostics.

Diagnostic code	Avg grounding %	Successful models	Models within threshold	Consensus
CA1826	11.06%	gpt-5.4, claude-sonnet-4-6, claude-opus-4-6	gpt-5.4, claude-sonnet-4-6, claude-opus-4-6	+
CA2100	20.22%	gpt-5.4, claude-sonnet-4-6, claude-opus-4-6	claude-sonnet-4-6, claude-opus-4-6	-
CA2326	10.47%	gpt-5.4, claude-sonnet-4-6, claude-opus-4-6	claude-sonnet-4-6, claude-opus-4-6	-

It has been observed that there is a discrepancy in the behavior exhibited by diagnostic codes CA2100 and CA2326. While all models successfully generated compilable fixes and resolved the target issue, the modification percentages exhibited greater variability. In the CA2100 case, the GPT-5.4 result reached 31.01%, which exceeded the average grounding value of 20.22%. In contrast, the Claude Sonnet 4.6 and Claude Opus 4.6 candidates remained within the acceptable range. A similar pattern emerges for CA2326, where GPT-5.4 produced a substantially larger modification percentage of 22.01%, while the other two models remained much closer to the historical baseline of 10.47%. As not all successful candidates remained within the required threshold, consensus could not be reached in these cases.

The findings indicate that the success of the build and the resolution of issues are not adequate in determining the final repair selection. A candidate may compile successfully and remove the target diagnostic, yet still differ substantially from historically observed repair behavior. The grounding-based comparison, therefore, provides an additional control mechanism that helps distinguish proportionate fixes from overly broad modifications. In this sense, the initial results of the prototype support the primary concept of the method: that automated repair should be evaluated not only by its technical functionality, but also by the consistency of the scale of change with the anticipated correction patterns.

The obtained results suggest that frontier models perform adequately for most tested cases, particularly for simpler or more localized diagnostics. However, the findings also suggest that these methods may not be adequate for more complex diagnostic codes, where generated fixes can become less proportionate even when they remain technically valid. Consequently, the establishment of a multi-LLM consensus is imperative, as it introduces an additional layer of reliability by comparing independently generated candidates and ensuring that the final selection does not rely on a single, overextended solution.

5. Conclusions and Future Work

This paper presented a consensus-based method for improving static code analysis remediation through the combined use of multiple large language models, historical grounding percentages, recompilation-based validation, and agreement-based final selection. The method was designed to address a key limitation of single-model repair generation, namely that a plausible fix may still be technically invalid, disproportionately broad, or inconsistent with alternative solutions. By treating repair as a comparative and validation-driven process, the proposed approach provides a more controlled framework for automated correction of static analysis findings.

The implemented prototype showed that this workflow can be realized in practice. The system was able to calculate grounding percentages from historical correction examples, retrieve and analyze repositories, extract compiler diagnostics, generate multiple candidate fixes for the same issue, validate them through recompilation, and determine whether consensus could be reached. The results suggest that this approach offers a broader and more transparent basis for repair selection than relying on a single generated output.

At the same time, the current study has several limitations. The evaluation was performed on a prototype and focused mainly on compiler diagnostics and build-based validation. Although recompilation and diagnostic improvement provide meaningful technical evidence, they do not guarantee full semantic correctness or preservation of intended behavior. In addition, the usefulness of grounding depends on the availability and representativeness of historical repair examples. Another important limitation is that the current validation is not yet sufficient for practical cost measurement. While the prototype can indicate whether a generated repair compiles and whether diagnostics are reduced, further validation is needed before the approach can be reliably used to estimate real-world remediation costs, developer effort reduction, model execution cost, review overhead, or the economic value of automated repair in production environments.

Future work should therefore extend the evaluation to a wider range of repositories, diagnostics, and repair scenarios, and strengthen validation through additional semantic checks such as automated tests or post-analysis. This stronger validation layer would also be necessary for future practical cost measurement, because cost-related conclusions should be based not only on successful recompilation, but also on whether generated fixes are behaviorally correct, reviewable, maintainable, and acceptable in realistic development workflows. It would also be valuable to refine the grounding model using larger historical datasets and to explore more advanced consensus strategies, such as weighted or confidence-aware selection. Together, these directions could help develop the proposed method into a more robust framework for trustworthy LLM-assisted static code analysis improvement and provide a more reliable basis for assessing its practical cost-effectiveness.

Declaration on Generative AI

During the preparation of this paper, the authors used generative AI tool (ChatGPT 5.4) for the revision of the English language. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

References

[1] J. Salet, K. Rakholia, O. Rahul, K. Jignesh, K. Jay, Navigating the Evolution: Current Trends and Future Directions in Programming Languages, International Journal of Innovative Research in Computer Science and Technology 12 (2024) 43–46. URL: https://ijircst.org/view_abstract.php?title=Navigating-the-Evolution:-Current-Trends-and-Future-Directions-in-Programming-Languages&year=2024&vol=12&primary=QVJULTEyODg=. doi:10.55524/ijircst.2024.12.4.7.

[2] Bug detection in Java code: An extensive evaluation of static analysis tools using Juliet Test Suites - Amankwah - 2023 - Software: Practice and Experience - Wiley Online Library, ???? URL: https://onlinelibrary.wiley.com/doi/10.1002/spe.3181.

[3] Y. Smaragdakis, N. Grech, S. Lagouvardos, K. Triantafyllou, I. Tsatiris, Symbolic value-flow static analysis: deep, precise, complete modeling of Ethereum smart contracts, Symbolic Value-Flow Static Analysis: Deep, Precise, Complete Modeling of Ethereum Smart Contracts (Artifact) 5 (2021) 163:1–163:30. URL: https://dl.acm.org/doi/10.1145/3485540. doi:10.1145/3485540.

[4] P. Diamantakis, T. Avgerinos, Y. Smaragdakis, Desyan: A Platform for Seamless Value-Flow and Symbolic Analysis, 2025. URL: http://arxiv.org/abs/2508.00508. doi:10.48550/arXiv.2508.00508, arXiv:2508.00508 [cs].

[5] J. Mertz, I. Nunes, Tigris: A DSL and framework for monitoring software systems at runtime, Journal of Systems and Software 177 (2021) 110963. URL: https://www.sciencedirect.com/science/article/pii/S0164121221000601. doi:10.1016/j.jss.2021.110963.

[6] F. A. Aboaoja, A. Zainal, F. A. Ghaleb, B. A. S. Al-rimy, T. A. E. Eisa, A. A. H. Elnour, Malware Detection Issues, Challenges, and Future Directions: A Survey, Applied Sciences 12 (2022) 8482. URL: https://www.mdpi.com/2076-3417/12/17/8482. doi:10.3390/app12178482.

[7] Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java | Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ???? URL: https://dl.acm.org/doi/10.1145/3611643.3616262.

[8] S. Dikici, T. T. Bilgin, Advancements in automated program repair: a comprehensive review, Knowledge and Information Systems 67 (2025) 4737–4783. URL: https://doi.org/10.1007/s10115-025-02383-9. doi:10.1007/s10115-025-02383-9.

[9] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, Z. Chen, A Critical Review of Large Language Model on Software Engineering: An Example from ChatGPT and Automated Program Repair, 2024. URL: http://arxiv.org/abs/2310.08879. doi:10.48550/arXiv.2310.08879, arXiv:2310.08879 [cs].

[10] H. Li, Y. Hao, Y. Zhai, Z. Qian, Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach, Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach (Artifact) 8 (2024) 111:474–111:499. URL: https://dl.acm.org/doi/10.1145/3649828. doi:10.1145/3649828.

[11] H. Wang, Z. Tang, S. H. Tan, J. Wang, Y. Liu, H. Fang, C. Xia, Z. Wang, Combining Structured Static Code Information and Dynamic Symbolic Traces for Software Vulnerability Prediction, in: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, Association for Computing Machinery, New York, NY, USA, 2024, pp. 1–13. URL: https://dl.acm.org/doi/10.1145/3597503.3639212. doi:10.1145/3597503.3639212.

[12] J. He, C. Treude, D. Lo, LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision, and the Road Ahead, ACM Trans. Softw. Eng. Methodol. 34 (2025) 124:1–124:30. URL: https://dl.acm.org/doi/10.1145/3712003. doi:10.1145/3712003.

[13] L. B. Kaesberg, J. Becker, J. P. Wahle, T. Ruas, B. Gipp, Voting or Consensus? Decision-Making in Multi-Agent Debate, in: W. Che, J. Nabende, E. Shutova, M. T. Pilehvar (Eds.), Findings of the Association for Computational Linguistics: ACL 2025, Association for Computational Linguistics, Vienna, Austria, 2025, pp. 11640–11671. URL: https://aclanthology.org/2025.findings-acl.606/. doi:10.18653/v1/2025.findings-acl.606.