International Journal of Engineering Innovation and Computational Science

Retrieval Augmented Generation for Reliable Software Engineering: A Critical Review of Repository Context and Evaluation

Ethan Vance [1], **Maya Lin** [2*]
[1] Associate Professor, Department of Computer Science, University of Texas at Arlington, Arlington, TX, USA
[2] Assistant Professor, Department of Computer Science & Software Engineering, Miami University, Oxford, OH, USA
*** Corresponding Author:** linm2@miamioh.edu

Citation: E. Vance and M. Lin, "Retrieval Augmented Generation for Reliable Software Engineering: A Critical Review of Repository Context and Evaluation," *International Journal of Engineering Innovation and Computational Science*, vol. 1, no. 1, pp. 1–16, 2026.

Received: 10 Dec 2025
Accepted: 02 Mar 2026

Abstract

Repository-aware coding assistants need access to project knowledge, yet retrieving that knowledge does not establish the correctness of generated software. This critical integrative review examines how retrieval-augmented generation supports code completion, repository-level generation, and issue resolution. A purposively selected corpus of 27 technical publications, supported by one methodological reference, is analysed through structured extraction of retrieval design, task setting, evaluation evidence, and limitations. The technical literature spans 2020–2026, with emphasis on recent repository-oriented methods and benchmarks. Four themes emerge. First, useful context depends on dependency structure, task requirements, and interactions between retrieved fragments, rather than similarity alone. Second, retrieval accuracy, textual agreement, and executable correctness provide different kinds of evidence. Third, repository freshness and the trustworthiness of retrieved material affect reliability independently of model capability. Fourth, additional retrieval and agentic steps require justification against latency, cost, and human verification effort. The review develops a reference architecture and a staged evaluation framework that connect context selection to independently assessed software behaviour. It argues for selective retrieval, version-controlled evidence, explicit failure analysis, and testing that remains separate from model confidence. Current research supports carefully bounded assistance, while broad claims of dependable autonomous maintenance require stronger evidence across repositories, languages, and operational conditions.

Keywords: Retrieval-Augmented Generation, Software Engineering, Repository-Level Code Generatio, Large Language Models, Software Reliability.

INTRODUCTION

A coding assistant can produce plausible code while misunderstanding the project in which that code must run. A function name may be familiar, but its local contract, import path, return type, or error behaviour may differ from examples encountered during model training. The resulting patch can look reasonable and still fail at the first integration boundary. For practical software engineering, access to repository-specific knowledge is therefore a central design problem. The difficulty is deciding which knowledge matters and establishing whether the generated change respects it.

Retrieval-augmented generation, abbreviated RAG, combines a language model with information obtained from an external collection. The foundational formulation linked parametric generation with non-parametric retrieval for knowledge-intensive language tasks [1]. Software applications adapt this principle to source files, application programming interface documentation, tests, and other project artefacts. RepoCoder demonstrated an iterative connection between retrieving repository context and generating code [2]. The attraction is straightforward: a project can supply information that a general model cannot reliably reconstruct from its parameters, and the external collection can change without retraining the entire generator.

That attraction should not be mistaken for a reliability guarantee. An assistant can retrieve a relevant function but use it incorrectly. It can copy an obsolete calling pattern, overlook a configuration constraint, or produce a patch that passes a narrow test while violating another requirement. RepoBench separates retrieval,

Copyright © 2026 by Author/s. IJEICS is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

completion, and pipeline evaluation [3], while CrossCodeEval examines completion that requires cross-file information [4]. These distinctions matter because a successful intermediate operation does not establish successful software behaviour. The code-oriented evaluation in CodeRAG-Bench makes the problem particularly visible: both locating useful material and using it effectively remain obstacles [5].

The move from completion to repository maintenance expands the problem further. SWE-bench evaluates changes made in response to real issue descriptions, using problems derived from existing repositories and their pull requests [6]. Such tasks require more than continuation of a local sequence. They involve identifying relevant components and coordinating a change with existing behaviour. At the same time, the Agentless preprint shows why elaborate autonomous workflows should face strong, simpler baselines: localization, repair, and validation can be organised without unrestricted model-directed action [7]. Architectural complexity is a hypothesis about usefulness, not evidence of it.

This review examines a focused question: what does the available literature establish about making retrieval-supported software engineering reliable, and what remains untested? Reliability here means producing changes that satisfy stated requirements and preserve relevant existing behaviour under identifiable operating conditions. The definition is deliberately broader than matching a reference completion and narrower than claiming universal correctness. Security, reproducibility, and the cost of checking a change enter the analysis because they influence whether an apparently successful assistant can be used responsibly in an engineering workflow.

The contribution is an integrative interpretation of retrieval methods, software benchmarks, and selected evidence on context use and security. Rather than rank systems evaluated under incompatible conditions, the paper connects design choices to the claims their evaluations can support. The resulting framework distinguishes evidence about context selection from evidence about code behaviour and operational usefulness. This distinction provides a practical basis for choosing evaluations, diagnosing failures, and deciding where human judgement remains necessary.

The focus is deliberately on repository context rather than general conversational assistance. A repository provides a concrete setting in which information has locations, versions, dependencies, and observable consequences when used incorrectly. It also offers a useful meeting point between information retrieval and software engineering. Retrieval research asks whether the right material reaches the generator; software engineering asks whether the resulting change behaves as required. Bringing these questions together makes it possible to examine a common source of overstatement without dismissing the substantial progress in either field.

LITERATURE REVIEW

Repository Context as an Engineering Resource

Repository retrieval differs from retrieving general prose because source code contains both textual and structural relationships. Similar names may indicate related functionality, but dependency edges can expose relationships that lexical overlap misses. A completion may require a type definition in one file and the behaviour of a wrapper in another. Conversely, a highly similar implementation can be unsuitable because it belongs to a different subsystem or assumes a different library version. A retrieval design therefore makes an implicit choice about what counts as relevant evidence.

Recent methods expose several ways to make that choice. Repoformer learns when retrieval is likely to help, addressing the inefficiency and possible harm of retrieving for every completion [8]. DraCo uses dataflow analysis to construct repository-specific context and retrieve code entities related to the completion target [9]. CodeRAG combines query construction, multiple retrieval paths, and reranking aligned with the generator [10]. RepoShapley extends the discussion to interactions among context fragments, treating their usefulness as dependent on the selected combination [11]. These approaches address different decisions within a pipeline; their published evaluations do not establish a universal ordering among them.

The progression is conceptually useful. Retrieval has moved beyond finding a nearby-looking example towards deciding whether information is needed and how several pieces should be combined. However, training a selector against one generator or benchmark can also tie its decisions to that setting. A method that

identifies useful context for completion may need different supervision for debugging, migration, or feature implementation. The appropriate unit of evidence changes with the task.

Context Availability and Context Use

Increasing the input window allows more repository material to be presented to a model, but does not settle how well that material is used. Liu et al. found position-sensitive performance in long-context question answering and key-value retrieval [12]. Their experiments concern language tasks rather than repository repair, so they cannot establish a specific failure rate for coding assistants. They nevertheless provide a reason to test ordering and distractors instead of assuming that available context is effectively understood.

The comparison by Li et al. further complicates a simple preference for retrieval. With sufficient resources, long-context models performed better on average in their evaluated tasks, while RAG retained a cost advantage; their hybrid routing method sought to use both [13]. Again, the result is not a ranking of repository engineering systems. It establishes that context strategies have to be compared under an explicit resource budget. For a small project, supplying a larger coherent region may be reasonable. A large repository with generated files, duplicated utilities, and historical examples creates a different selection problem.

Self-RAG offers a related mechanism for deciding when to retrieve and for assessing support during generation [14]. RAGAs separates aspects of retrieval and generation evaluation, including the use of retrieved information [15]. Both contribute useful concepts, but language-based assessments of support cannot replace execution-based checks on software. A faithful explanation of an incorrect implementation remains an incorrect implementation. The literature therefore supplies transferable design ideas whose software-specific validity still has to be tested.

Evaluation Beyond Plausible Completions

CoderEval examines pragmatic code generation with dependencies beyond isolated self-contained functions [16]. DevEval brings manually annotated requirements and dependencies into evaluation based on real repositories [17]. Together, these benchmarks make project context more visible than a standalone programming exercise does. They also expose an interpretive challenge: success on one task unit does not automatically transfer to a larger maintenance workflow, even when both tasks involve source code from a repository.

Test adequacy introduces another source of uncertainty. EvalPlus found that expanding the tests used to assess generated programs exposes incorrect solutions missed by the original checks and can change model rankings [18]. LiveCodeBench addresses a different concern through time-segmented evaluation on newly collected programming problems and through multiple coding capabilities [19]. Its competition setting is not equivalent to maintaining an established codebase, but its attention to evaluation timing is relevant wherever public repositories or benchmark solutions may enter training data.

These studies caution against treating one score as a stable measure of engineering ability. Exact match, identifier agreement, test success, and issue resolution are observations at different levels. Even an execution-based result depends on the environment and the checks applied. A stronger evaluation identifies what the task requires, what the available tests cover, and which assumptions remain outside those tests. The meaning of the denominator also matters: a set of short completions should not be described as if it represented arbitrary repository changes.

Retrieval within Software Engineering Workflows

Retrieval can be embedded within a larger interaction loop. SWE-agent studies interfaces through which a model navigates repositories, edits files, and runs programs [20]. AutoCodeRover combines program-aware search with repair and can use test-derived fault localization to sharpen the context [21]. These systems make retrieval an activity that changes as the assistant observes results. An unsuccessful test can create a more informative search target than the original natural-language request.

Validation itself is also becoming a generated artefact. SWT-Bench evaluates the construction of tests for real bug reports, including issue reproduction and their ability to help distinguish fixes [22]. Feature implementation creates additional demands: FEA-Bench evaluates additions that must work with existing

repository code [23]. M2RC-EVAL broadens repository-level completion assessment to 18 programming languages and more detailed completion scenarios [24]. This expansion is welcome, although greater benchmark variety also increases the need to state exactly which settings support a general claim.

The engineering value of an assistant is partly determined by how easily its mistakes can be found. A modest candidate patch accompanied by traceable context and a reproducible failing test may be more useful than a larger, confident patch that leaves its assumptions hidden. This interpretation makes reviewability a relevant question for interface evaluation, although the cited studies do not provide a direct comparison of those two situations.

Security and the Boundaries of Retrieved Evidence

Retrieved material can carry more than useful project information. Greshake et al. demonstrated indirect prompt injection through content encountered by language-model applications [25]. PoisonedRAG examines attacks that corrupt a retrieval collection to influence generated answers [26]. The latter study concerns RAG knowledge tasks, so its attack outcomes should not be transferred numerically to software repositories. Both studies identify a relevant boundary: information admitted into context may influence model behaviour even when it should have no authority over instructions or tools.

Generated code also needs security evaluation independently of retrieval quality. Pearce et al. documented insecure suggestions in an early evaluation of GitHub Copilot [27]. That study is retained as foundational evidence about the separation between plausible code and secure code, not as an estimate for present systems. A repository-aware assistant may inherit vulnerable patterns from retrieved examples just as a parameter-only model may reproduce patterns from training. Whether a particular retrieval design increases or reduces that risk remains an empirical question.

Table 1 distinguishes the task settings considered in this review. The differences explain why evidence must be interpreted within its original scope. Completion methods, repair workflows, and general RAG security studies contribute to the same engineering discussion, but they do not answer the same experimental question.

Table 1. Task Settings and The Boundaries of their Evidence

Task Setting	Information Needed	Evidence Obtained	Main Boundary
Cross-file completion [3], [4]	Definitions and local usage	Retrieval and completion quality	Does not establish issue resolution
Repository generation [16], [17]	Requirements and dependencies	Behaviour under supplied tests	Coverage remains task-specific
Repair and features [6], [23]	Issue context and affected code	Patch or feature test outcomes	Long-term maintenance is unresolved
General RAG [12]–[15], [26]	External textual evidence	Context use and trust mechanisms	Transfer to software is indirect

METHODOLOGY

The study uses one method, a critical integrative literature review. This approach combines evidence from related research strands to develop a conceptual account rather than estimate a pooled treatment effect. Snyder's methodological discussion supports selecting a review design that matches the question and making

the selection and synthesis logic explicit [28]. Here, the purpose is to relate retrieval decisions to evidence of software reliability. A meta-analysis would be inappropriate because the publications vary in task units, models, retrieval collections, resource budgets, and outcome definitions.

The literature search was conducted on 23 October 2026 through a web-search interface, followed by checks against primary publication records. Sources included the ACL Anthology, the proceedings of ICLR, NeurIPS, and ICML, ACM and IEEE records, USENIX proceedings, and author-posted manuscripts. Searches combined retrieval-augmented code generation, repository-level code completion, software engineering agents, code generation evaluation, and RAG security. Exact-title follow-up searches resolved bibliographic details and retrieved full texts. References and related-work discussions in the located papers informed additional title searches. These were targeted searches, not exhaustive exports from bibliographic databases.

The analytical corpus contains 27 technical publications [1]–[27]; the methodological reference [28] is separate. Publications from 2023–2026 receive priority. Two earlier technical works supply the foundational RAG formulation and evidence on code security. A work was retained when it provided an identifiable method, benchmark, or empirical result relevant to repository context, software evaluation, or a clearly transferable reliability mechanism. Primary records were checked for title, authorship, year, venue or preprint status, and an identifier or stable publication URL. The published version was preferred where it could be established. Agentless is explicitly treated as a preprint.

The review excluded product marketing, unsupported implementation claims, duplicate versions of the same work, and studies whose relevance depended only on using a language model. General RAG studies were included selectively when they clarified context use, evaluation, or information trust. Their results were coded as indirect evidence for software engineering. The resulting sample is purposive. It cannot support claims about the prevalence of a method across all publications, and no database-wide screening counts or completeness claim are inferred from it.

For organising the synthesis, each publication received one principal role. Five concern repository retrieval methods [2], [8]–[11]; nine provide repository-oriented evaluation resources [3]–[6], [16], [17], [22]–[24]; and three study repair workflows [7], [20], [21]. Two address broader code evaluation [18], [19], five supply general RAG or context-use foundations [1], [12]–[15], and three inform security analysis [25]–[27]. These assignments describe use within this review, since individual publications can contribute to several research areas. They provide a check on corpus composition without treating the categories as independent experimental replications.

For each technical publication, structured extraction recorded its task, information source, retrieval or interaction mechanism, comparator, evaluation outcome, and principal limits on interpretation. Table 2 defines the extraction fields. A second reading of the relevant methods and results checked whether a claim concerned retrieval quality, generated code, executed behaviour, or the surrounding workflow. When a publication did not establish an operational property, the synthesis treated that property as unresolved rather than assuming either success or failure. No experiments were rerun, and no unreported performance values were reconstructed.

Table 2. Fields Used For Structured Evidence Extraction

Field	Question Applied to Each Technical Publication
Task and context	What is generated, and what repository or external information is available?
Mechanism	How is context retrieved, selected, ordered, or updated during interaction?
Comparator	Which baseline is used, and which other components change?
Outcome	Does the measure concern retrieval, text, execution, or workflow performance?

Field	Question Applied to Each Technical Publication
Validity limits	Which languages, datasets, versions, budgets, or evaluation assumptions restrict interpretation?
Evidence role	Is the result direct software evidence, an indirect mechanism, or a foundational concept?

The analysis grouped the extracted evidence by the engineering decision it informed. Initial categories concerned context selection, correctness assessment, trust and freshness, and resource use. Comparison across publications refined these into the four themes reported below. Conflicting implications were retained, including the benefits of larger context in some language evaluations and the benefits of selective retrieval in code completion. This is a qualitative synthesis within the integrative review, not a separate experiment or an independently replicated coding study.

Interpretive confidence was assessed narratively. Direct code experiments and executable repository benchmarks carried more weight for software claims than language-only studies. A component ablation could support a narrower causal interpretation than an end-to-end comparison that changed several elements simultaneously. Preprint status, restricted language coverage, shared benchmarks, and limited operational evidence reduced the breadth of conclusions. The review reports no inter-rater agreement because independent duplicate screening and coding were not conducted. Its architecture and evaluation framework are proposals derived from the synthesis and require empirical validation.

RESULTS

Useful Retrieval Depends on the Task and the Selected Context

The first theme concerns the gap between retrieving related material and retrieving material that helps produce a correct change. RepoCoder uses a generated continuation to improve subsequent retrieval, while DraCo uses program relationships to identify relevant context [2], [9]. These mechanisms address different weaknesses in the initial query. One improves its textual representation; the other supplies structural information. Their common implication is that repository retrieval should be conditioned on the particular missing information rather than treated as a generic nearest-neighbour search.

The distinction becomes clearer when considering negative cases. RepoCoder reports limited effectiveness where a repository contains little code duplication [2]. This is an informative boundary, since a method that benefits from recurring patterns may contribute less when the task introduces genuinely new behaviour. A repository can be large without containing a useful analogue. Adding more retrieved material under those conditions can increase the burden on the generator while leaving the missing specification unresolved.

Selective and interaction-aware methods make this boundary explicit. Repoformer evaluates whether retrieval is useful before committing to it [8]. CodeRAG attempts to improve alignment between retrieved information and generation [10]. RepoShapley evaluates the contributions of combinations of fragments [11]. The combined interpretation is that context utility depends on the generator, task, and surrounding context. A fragment that appears unnecessary by itself may define a type needed to interpret another fragment. A repeated implementation may add little once an equivalent example is already present.

This does not imply that every application requires a learned selector or a graph. Table 3 summarises the different design choices and the limits on their interpretation. A lightweight lexical approach can remain a useful baseline, especially when identifiers are distinctive and the repository is small. Structural indexing and learned filtering introduce their own construction, update, and supervision requirements. Their value has to appear in the downstream outcome under a comparable budget.

Table 3. Contrasting Repository Retrieval Designs

Method	Central Decision	Context Mechanism	Interpretive Limit
RepoCoder [2]	Refine the query	Iterative retrieval and generation	Benefits depend partly on recurring code patterns
Repoformer [8]	Decide when to retrieve	Learned selective retrieval	Policy utility depends on workload and supervision
DraCo [9]	Follow program relationships	Dataflow-guided context graph	Completion evidence; wider workflows untested
CodeRAG [10]	Align context with generation	Multiple paths and BestFit reranking	Residual retriever–generator misalignment
RepoShapley [11]	Select useful combinations	Interaction-aware filtering	Completion agreement is not behavioural proof

Figure 1 presents the reference architecture derived from this theme. It separates the repository snapshot, retrieval policy, generator, and validation environment. The separation allows a failed candidate to trigger a targeted information request without treating every failure as evidence that the language model needs to generate more text. It also preserves a record of the context used for each attempt. The figure describes a proposed organisation of components, not a system implemented or benchmarked in this review.

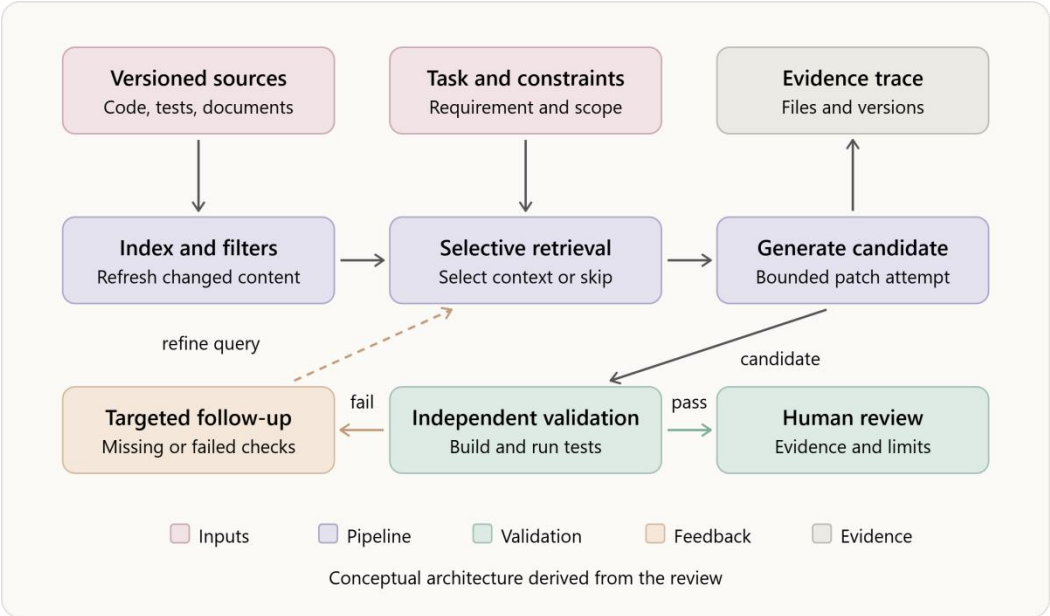


Figure 1. Proposed Architecture Linking Versioned Repository Context, Selective Retrieval, Generation, and Independent Validation

Evidence of Correctness Remains Separate from Evidence of Relevance

The second theme is a persistent mismatch between the property being measured and the broader claim attached to it. RepoBench's distinct retrieval and pipeline tasks help expose where a system can fail [3].

CrossCodeEval tests the use of cross-file context [4]. Neither a retrieval hit nor agreement with a reference line, considered alone, proves that a change meets an issue's behavioural requirements. Textual metrics can be informative for completion, but their interpretation needs to remain attached to that purpose.

The word verification also needs care. RepoShapley's post-verification concerns completion outcomes measured through edit similarity and exact match, and the paper identifies limitations tied to its decoding and selection setting [11]. This is different from running an independent behavioural test suite. Treating both activities as interchangeable would erase an important distinction between validating a context-selection decision and validating the resulting program. The same caution applies to using an automated language judge to assess an explanation of code.

Executable benchmarks provide a stronger connection to behaviour, while preserving their own boundaries. CoderEval and DevEval retain dependencies and repository context [16], [17]. SWE-bench evaluates issue-directed changes, and FEA-Bench addresses feature implementation [6], [23]. The latter includes 1,401 task instances from 83 repositories, while DevEval contains 1,825 samples from 115 repositories. These figures describe dataset coverage, not interchangeable sample sizes for a common performance estimate. The tasks differ in what a successful solution must accomplish.

Test strength remains consequential even within executable evaluation. EvalPlus demonstrates that more demanding checks can reveal failures hidden by a smaller test set [18]. SWT-Bench adds evidence about generating issue-reproducing tests and using them to assess candidate fixes [22]. The synthesis supports keeping test construction and patch generation distinguishable. A test that merely reproduces the assumptions embedded in a generated patch offers less independent assurance than one derived from the requirement and checked against the original failure.

Table 4 maps representative benchmarks to the claims they can inform. Figure 2 then organises these claims into progressively broader evaluation questions. It is a conceptual ordering of scope, not a numerical reliability scale. Evidence at a narrower level remains useful, but an additional claim needs additional checks. In particular, language coverage is not the same as workflow coverage: M2RC-EVAL expands the former without purporting to evaluate every maintenance activity [24].

The information available to the system is another part of the outcome definition. A model supplied with the correct dependency has a different task from one required to locate that dependency in an entire repository. Combining those results would hide the contribution of retrieval. Similarly, allowing access to the reference implementation or to tests that expose the intended solution changes what success demonstrates. Evaluation reports should therefore describe the permitted inputs alongside the metric. This recommendation follows from comparing the task structures across the corpus; it does not imply that the reviewed benchmarks all provide the same information or suffer from the same leakage mechanism.

Table 4. Evaluation Resources and the Claims they can Support

Resource	Primary Evaluation Unit	Useful Evidence	Boundary
RepoBench [3]	Retrieval and next-line completion	Component and pipeline diagnosis	Not a full repair workflow
CrossCodeEval [4]	Cross-file completion	Use of repository dependencies	Completion-focused outcomes
CoderEval and DevEval [16], [17]	Functions within project context	Dependency-aware generation	Test adequacy still matters
SWE-bench [6]	Issue-directed patch	Repository test outcomes	Resolution follows the test oracle

Resource	Primary Evaluation Unit	Useful Evidence	Boundary
EvalPlus [18]	Generated program	Failures under stronger tests	No complete correctness guarantee
SWT-Bench [22]	Issue-reproducing test	Reproduction and patch filtering	Not general edge-case coverage
FEA-Bench [23]	Feature implementation	Integration of new code	Python and selected repositories
M2RC-EVAL [24]	Multilingual completion	Language and scenario coverage	Does not assess all maintenance tasks

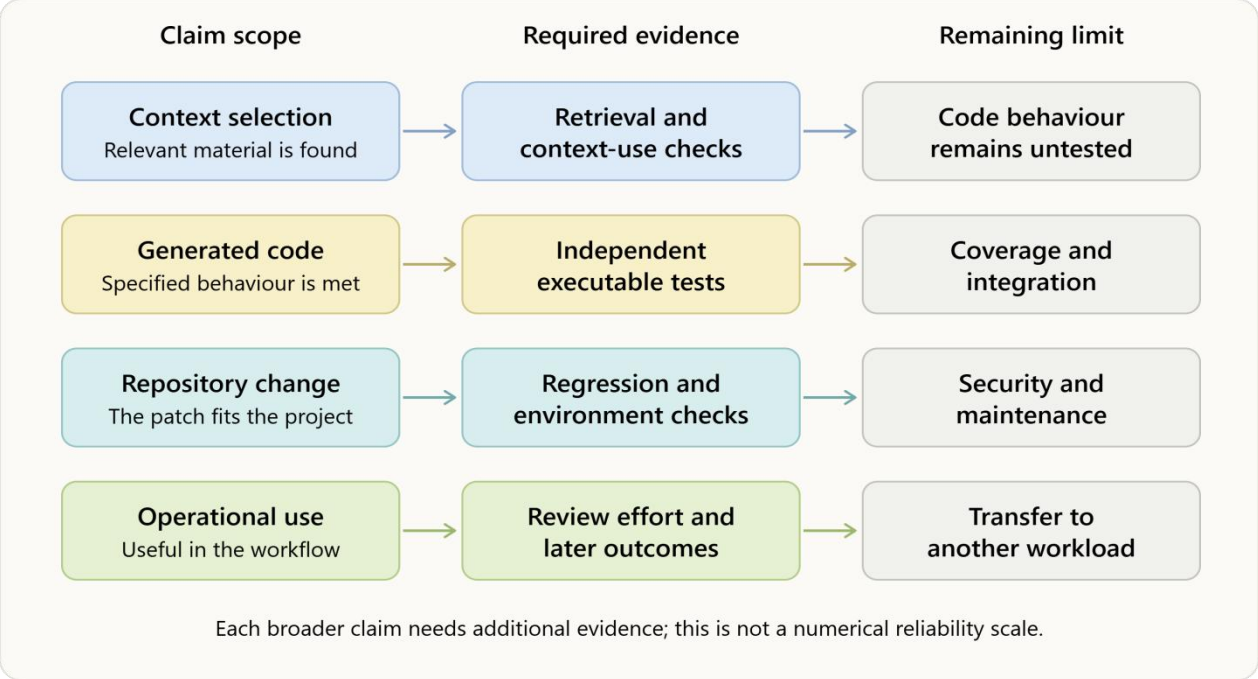


Figure 2. Evaluation Questions Needed as Claims Expand from Context Retrieval to Operational Usefulness

Freshness and Trust Constrain The Use of Repository Knowledge

The third theme concerns conditions under which retrieved information should be used. The reviewed methods largely evaluate context within defined datasets or repository states. Their results do not establish dependable behaviour when an index is stale, permissions change, or conflicting versions coexist. That absence limits deployment claims; it does not show that the methods necessarily fail under those conditions. The practical inference is that repository state must become an explicit variable in both system design and evaluation.

Consider a hypothetical interface migration in which a function changes from returning a value to returning a structured object. A retriever may select a close textual match from an earlier revision. The generator can then produce code consistent with that example while inconsistent with the current interface. Nothing about high similarity resolves the version conflict. Recording a commit identifier, refreshing

affected index entries, and checking the current definition would make the failure easier to detect. These are proposed controls motivated by the mechanism, rather than measured outcomes from the reviewed experiments.

Trust introduces a distinct problem. The indirect-injection and knowledge-corruption studies show that retrieved content can redirect or contaminate generation [25], [26]. Applied to coding assistants, the relevant design question is whether comments, documentation, and external examples are treated as data or as instructions with authority over actions. An accurate search result can still carry an untrusted instruction. Improving relevance therefore does not, by itself, resolve the security boundary.

The review consequently separates code quality from context integrity. Tests can reveal some faulty behaviour, yet they may not detect disclosure through an unnecessary tool call or inappropriate access to another project. Conversely, restricting access does not establish that an authorised patch is correct. Table 5 pairs these distinct risks with proposed controls and the evidence needed to evaluate them. The controls are intentionally testable: each should have a defined failure condition rather than rely on a general claim that the assistant is trustworthy.

Access control also applies to derived representations. If source files are removed or a user's permissions change, previously created snippets and indexes should not silently preserve access. A complete operational evaluation would check the retrieval layer as well as the visible source browser. This is an architectural implication of maintaining an external information collection. The selected literature does not establish how consistently deployed coding assistants enforce such changes, so the review makes no prevalence claim about the problem.

Table 5. Proposed Controls and Evidence Needed to Assess them

Risk	Proposed Control	Evaluation Evidence
Stale context	Pin repository and dependency versions	Failure and detection rates as index age changes
Missing dependencies	Trace definitions and imports	Outcomes when relevant context is withheld
Untrusted content	Separate content from instructions; restrict tools	Adversarial tests and access-control logs
Weak validation	Use independently specified behavioural tests	Issue reproduction, regressions, and held-out checks
Excessive effort	Set budgets and explicit stopping rules	Total attempts, latency, cost, and review time

Note: The controls are synthesis-derived recommendations, not experimentally validated results of this review.

Workflow Value Depends on Output Acquisition and Verification Costs

The fourth theme concerns how additional computation translates into usable engineering work. Iterative retrieval, reranking, repeated generation, and test execution can improve opportunities to find a solution, but each adds work. Repoformer's selective design supplies direct evidence that retrieval policy can affect efficiency in completion [8]. The RAG versus long-context comparison supplies complementary, indirect evidence that context strategy must be considered alongside resource use [13]. Neither result establishes a fixed cost advantage across all repository sizes and interaction patterns.

The contrast between SWE-agent, AutoCodeRover, and Agentless also argues against counting tool use as a contribution in itself [7], [20], [21]. These systems organise information gathering and repair differently. Their value lies in the resolved tasks and the conditions under which they achieve them. A comparison that gives one system more attempts, a stronger generator, or better execution feedback cannot isolate the benefit of its retrieval design. Such a comparison may still be useful as an assessment of complete systems, provided that the claim remains at that level.

An additional cost is the developer's effort to inspect the result. Most of the evidence considered here centres on benchmark performance and computational behaviour; it does not establish a general reduction in maintenance workload. A reasonable interpretation is that reviewability belongs in the evaluation target. The assistant should expose the files and definitions on which it relied, the tests actually run, and unresolved assumptions. A polished narrative without those details can make checking harder rather than easier.

An assistant saves little if its output transfers all uncertainty to the person reviewing it. The size of that burden cannot be inferred from the benchmark results considered here. It motivates measuring time to an acceptable change, the work spent rejecting unsuitable candidates, and later corrections. These outcomes connect the computational system to the engineering task more directly than token counts or generation speed alone.

DISCUSSION

Reliability Requires Evidence Across Component Boundaries

The synthesis supports treating retrieval-supported software engineering as a sequence of claims that must remain connected. The retriever claims that selected material is useful to the task. The generator claims, implicitly, that a proposed change uses that material appropriately. The validation stage supplies evidence about specified behaviours. A deployment decision adds assumptions about access, maintainability, and the consequences of a mistake. Reliability weakens when one stage's evidence is silently used to justify another stage's claim.

This interpretation changes the role of context provenance. A list of retrieved files is useful for tracing a decision, but it does not establish that those files were sufficient or correctly interpreted. Likewise, a successful build establishes compatibility with the build process under the recorded environment, not satisfaction of every requirement. The proposed architecture therefore retains intermediate artefacts and makes the transitions inspectable. Engineers should be able to identify which version of a definition informed a patch and which test would fail if that interpretation were wrong.

Figure 3 illustrates a diagnostic process for an unsuccessful candidate. It separates missing evidence, misuse of available evidence, and inadequate validation before recommending a new attempt. This decomposition is a proposal arising from the review. It has not been shown to improve repair performance. Its immediate value is analytical: it prevents an undifferentiated label such as model failure from hiding a stale index, a poor query, or a test that never exercised the changed path.

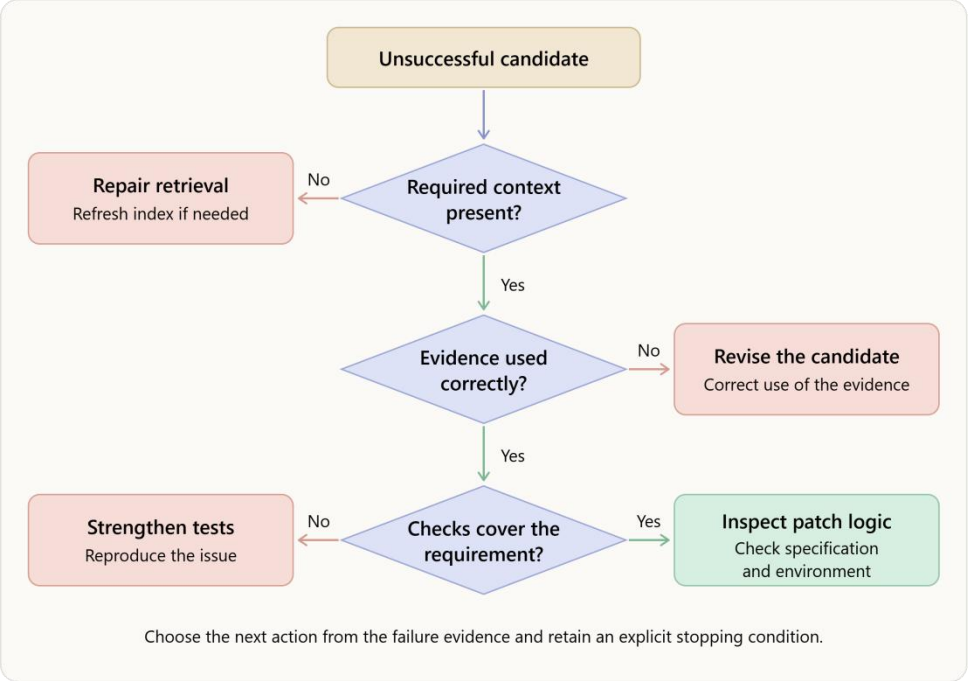


Figure 3. Proposed Diagnosis of an Unsuccessful Candidate Before Repeating Retrieval or Generation

The same distinction helps interpret successful cases. When a system solves an issue, the result may depend on a good initial retrieval, an effective later search, favourable test coverage, or an easily localised defect. Reporting these pathways would reveal more than a single aggregate percentage. It would also make negative results useful. A retrieval approach that fails only when dependency resolution is ambiguous raises a more specific engineering question than an unexplained decline in an overall score.

Designing Comparisons that Support Practical Decisions

Evaluation should begin with the intended action. Inline completion, answering a repository question, and modifying a service have different time budgets and error consequences. For completion, the central comparison may involve acceptance and correction of a short suggestion. For issue resolution, the unit is a patch applied to a recorded repository state. An experiment should specify this unit before choosing a benchmark or metric. Otherwise, the evaluation can become a convenient measurement of a property the application does not principally need.

A useful experimental design would hold the generator, task set, and resource budget constant while comparing no retrieval, a simple retrieval baseline, and the proposed retrieval mechanism. A matched long-context condition could determine whether careful selection adds value beyond supplying more coherent context. An oracle-context condition can help estimate the room for improvement in retrieval, provided that it is reported as diagnostic information unavailable in normal use. These comparisons are recommendations from the synthesis, not experiments conducted here.

Attempts and stopping rules also require explicit reporting. A system allowed many candidates has a different opportunity to succeed from one restricted to a single attempt. If all attempts count towards cost but only the best patch is assessed, the selection procedure must be described. Failed environment setup and timeouts should be retained in the accounting rather than silently removed. Statistical uncertainty should reflect clustering within repositories where appropriate, because many tasks from one project do not provide the same diversity as equally many independent projects.

Cost measurement should specify its boundary as carefully as correctness measurement does. Index construction may occur once, while retrieval and generation occur repeatedly; counting only online calls can conceal an expensive preparation stage. Conversely, charging the full index cost to every task would misrepresent a shared service. A transparent report would separate preparation, refresh, per-attempt inference,

execution, and human review. It would identify the expected reuse of the index and report both successful and unsuccessful attempts. Such accounting permits readers to judge whether an apparent improvement depends on a workload very different from their own, without requiring a single monetary estimate that becomes obsolete when model pricing changes.

Perturbation tests would add information that a fixed benchmark often lacks. Relevant conditions include renamed identifiers, outdated documentation, irrelevant but plausible snippets, missing dependencies, and alternative placement of necessary definitions. Security tests should additionally distinguish corrupted factual content from instruction-bearing material. The purpose is not to assemble the longest possible checklist. It is to identify which changes in the information environment invalidate the claimed capability and whether the assistant detects that loss of support.

Temporal separation needs similar discipline. LiveCodeBench makes evaluation timing visible [19], but applying that principle to repositories requires decisions about commit dates, later fixes, and duplicate code across projects. A repository snapshot should exclude information that would reveal the intended patch. Public availability alone does not demonstrate contamination, and uncertainty about training data should be acknowledged without asserting that a benchmark is either perfectly clean or necessarily compromised. Both extremes conceal the actual evidential limit.

Choosing Retrieval and Validation for the Application

There is no defensible universal prescription to use the largest context, the most elaborate graph, or the greatest number of agent steps. A small, well-structured project may be served adequately by simple search and a strong test suite. A large repository with many related interfaces may justify structural retrieval. Tasks that frequently require several complementary definitions may benefit from context selection that accounts for interactions. These are hypotheses to test against the application's workload, not conclusions that one method dominates across settings.

The first implementation decision should therefore be the information contract. The system needs to know which repository state is authoritative, which external documentation matches its dependencies, and what content it is permitted to access. These choices determine the meaning of a retrieved result. An index that combines several versions without exposing their provenance makes the generator responsible for resolving conflicts it may not recognise. Version labels and selective refresh make the contract easier to check.

The second decision concerns feedback. Execution results can guide further search, but they should be interpreted as observations under a particular environment. An import error may identify a missing dependency rather than an incorrect algorithm. A test failure may indicate a flawed patch, an inadequate reproduction, or an unrelated pre-existing defect. Feeding all failures back as undifferentiated error text can encourage unnecessary edits. A controlled workflow would classify the feedback and constrain the next action to the unresolved part of the task.

The third decision is where to stop. A system should be able to return an unresolved dependency or insufficient specification rather than continue indefinitely. Stopping is especially reasonable when repeated attempts reuse the same unsupported assumption. The evidence reviewed here supports selective computation more clearly than it supports universal autonomous action. For maintainers, an accurate statement of what remains unknown can be a useful output, provided that the assistant identifies the missing evidence precisely and preserves the work already checked.

Research Priorities Emerging from the Evidence

Several specific studies would advance the field more than another comparison under slightly different defaults. One is a controlled investigation of context freshness: keep the generator and tasks fixed, vary the age and consistency of the index, and measure both incorrect outputs and detection of insufficient support. This would test an operational issue that current snapshot-based evaluations cannot settle. The outcome should distinguish a missed solution from a confidently wrong patch, since those failures impose different review costs.

A second priority is to evaluate retrieval choices against independent behavioural outcomes. Completion metrics remain useful, but selection objectives trained only on textual similarity may reward the wrong

information for a repair task. Experiments could compare selectors trained on completion agreement with selectors trained on tests that were constructed independently of generated candidates. The challenge is avoiding expensive or leaky supervision. Any claimed gain would need to include the cost and availability of that feedback.

A third priority is to follow changes beyond initial test success. Maintenance studies could record whether accepted patches require later correction, introduce new dependencies, or complicate subsequent changes. Such work would be slower and less tidy than benchmark evaluation, but would directly address the practical claims attached to coding assistants. A patch's usefulness is not fully visible at the moment it first passes. Longitudinal evidence would help establish whether an immediate saving persists or reappears as work elsewhere in the maintenance process.

Language and repository diversity remain part of this agenda. Completion coverage across many languages is valuable [24], but repair and feature development also depend on build systems, frameworks, and project conventions. Evaluation should report where a result holds rather than convert breadth in one dimension into a general claim about all software engineering. Carefully chosen negative cases, including unfamiliar project structure and incomplete issue descriptions, would make that boundary easier to understand.

Limitations of the Review

The review's purposive corpus prioritises influential methods and complementary evidence rather than exhaustive coverage. Web-indexed discovery can miss relevant studies, particularly those described with different terminology or unavailable through accessible primary records. The literature also changes quickly. Including a recent context-selection study does not make the corpus complete through the search date. The review offers an auditable conceptual synthesis of identified works, not a census of the field.

The evidence is heterogeneous. Some papers evaluate completion, others evaluate complete repair workflows, and several investigate general RAG behaviour. Transfer from the last group is explicitly inferential. Shared models, datasets, and repository families also limit the independence of apparent agreement across studies. The paper therefore avoids pooled accuracy estimates and does not interpret repeated use of a benchmark as repeated validation of production readiness.

Finally, the synthesis was not independently double-coded, and the proposed architecture was not implemented. Its design recommendations remain open to revision through controlled evaluation. The most defensible conclusions concern distinctions that recur across the evidence: relevant context is not sufficient proof of correctness, executable checks depend on their coverage, and operational usefulness requires more than a favourable generation score. The scale of any benefit in a particular organisation remains unresolved.

CONCLUSION

Retrieval-augmented generation provides a practical way to connect coding models with repository-specific knowledge. Recent work improves how systems identify useful context, decide when retrieval is necessary, and combine information across files. The strongest interpretation of this progress is that context can be managed more deliberately. It does not establish that retrieved information has been interpreted correctly or that a generated change is safe to accept without further evidence.

The critical integrative review identifies four connected requirements for reliable use: task-sensitive context selection, independent assessment of software behaviour, control over the freshness and trust of retrieved material, and evaluation of the complete cost of obtaining an acceptable result. The proposed architecture and evaluation framework make these requirements explicit while preserving the distinction between published findings and engineering recommendations.

For research, the priority is clearer attribution of gains to specific components under matched conditions, followed by evaluation across changing repository states and meaningful maintenance tasks. For implementation, the immediate opportunity lies in bounded assistance with traceable context, reproducible execution, and explicit stopping conditions. A useful coding assistant should help an engineer understand why a change is supported and where uncertainty remains. That is a firmer basis for practical adoption than treating a benchmark score as a general certificate of software reliability.

REFERENCES

[1] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 9459–9474. [Online]. Available: https://papers.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[2] F. Zhang et al., "RepoCoder: Repository-level code completion through iterative retrieval and generation," in *Proc. 2023 Conf. Empirical Methods in Natural Language Processing*, 2023, pp. 2471–2484. doi: 10.18653/v1/2023.emnlp-main.151.

[3] T. Liu, C. Xu, and J. McAuley, "RepoBench: Benchmarking repository-level code auto-completion systems," in *Proc. 12th Int. Conf. Learning Representations*, 2024. [Online]. Available: https://proceedings.iclr.cc/paper_files/paper/2024/hash/d191ba4c8923ed8fd8935b7c98658b5f-Abstract-Conference.html

[4] Y. Ding et al., "CrossCodeEval: A diverse and multilingual benchmark for cross-file code completion," in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 46701–46723. doi: 10.52202/075280-2023.

[5] Z. Z. Wang et al., "CodeRAG-Bench: Can retrieval augment code generation?," in *Findings of the Association for Computational Linguistics: NAACL 2025*, 2025, pp. 3199–3214. doi: 10.18653/v1/2025.findings-naacl.176.

[6] C. E. Jimenez et al., "SWE-bench: Can language models resolve real-world GitHub issues?," in *Proc. 12th Int. Conf. Learning Representations*, 2024. [Online]. Available: https://proceedings.iclr.cc/paper_files/paper/2024/hash/edac78c3e300629acfe6cbe9ca88fb84-Abstract-Conference.html

[7] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying LLM-based software engineering agents," *arXiv preprint arXiv:2407.01489*, 2024. doi: 10.48550/arXiv.2407.01489.

[8] D. Wu, W. U. Ahmad, D. Zhang, M. K. Ramanathan, and X. Ma, "Repoformer: Selective retrieval for repository-level code completion," in *Proc. 41st Int. Conf. Machine Learning*, vol. 235, 2024, pp. 53270–53290. [Online]. Available: https://proceedings.mlr.press/v235/wu24a.html

[9] W. Cheng, Y. Wu, and W. Hu, "Dataflow-guided retrieval augmentation for repository-level code completion," in *Proc. 62nd Annu. Meeting Association for Computational Linguistics*, vol. 1, 2024, pp. 7957–7977. doi: 10.18653/v1/2024.acl-long.431.

[10] S. Zhang, Y. Ding, S. Lian, S. Song, and H. Li, "CodeRAG: Finding relevant and necessary knowledge for retrieval-augmented repository-level code completion," in *Proc. 2025 Conf. Empirical Methods in Natural Language Processing*, 2025, pp. 23278–23288. doi: 10.18653/v1/2025.emnlp-main.1187.

[11] Y. Huo et al., "RepoShapley: Shapley-enhanced context filtering for repository-level code completion," in *Findings of the Association for Computational Linguistics: ACL 2026*, 2026, pp. 10390–10412. doi: 10.18653/v1/2026.findings-acl.505.

[12] N. F. Liu et al., "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024. doi: 10.1162/tacl_a_00638.

[13] Z. Li, C. Li, M. Zhang, Q. Mei, and M. Bendersky, "Retrieval augmented generation or long-context LLMs? A comprehensive study and hybrid approach," in *Proc. 2024 Conf. Empirical Methods in Natural Language Processing: Industry Track*, 2024, pp. 881–893. doi: 10.18653/v1/2024.emnlp-industry.66.

[14] A. Asai, Z. Wu, Y. Wang, A. Sil, and H. Hajishirzi, "Self-RAG: Learning to retrieve, generate, and critique through self-reflection," in *Proc. 12th Int. Conf. Learning Representations*, 2024. [Online]. Available: https://proceedings.iclr.cc/paper_files/paper/2024/hash/25f7be9694d7b32d5cc670927b8091e1-Abstract-Conference.html

[15] S. Es, J. James, L. Espinosa Anke, and S. Schockaert, "RAGAs: Automated evaluation of retrieval augmented generation," in *Proc. 18th Conf. European Chapter of the Association for Computational Linguistics: System Demonstrations*, 2024, pp. 150–158. doi: 10.18653/v1/2024.eacl-demo.16.

[16] H. Yu et al., "CoderEval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proc. IEEE/ACM 46th Int. Conf. Software Engineering*, 2024, pp. 1–12. doi: 10.1145/3597503.3623316.

[17] J. Li et al., "DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories," in *Findings of the Association for Computational Linguistics: ACL 2024*, 2024, pp. 3603–3614. doi: 10.18653/v1/2024.findings-acl.214.

[18] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation," in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 21558–21572. doi: 10.52202/075280-0943.

[19] N. Jain et al., "LiveCodeBench: Holistic and contamination free evaluation of large language models for code," in *Proc. 13th Int. Conf. Learning Representations*, 2025. [Online]. Available: https://proceedings.iclr.cc/paper_files/paper/2025/hash/94074dd5a072d28ff75a76dabed43767-Abstract-Conference.html

[20] J. Yang et al., "SWE-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems*, vol. 37, 2024, pp. 50528–50652. doi: 10.52202/079017-1601.

[21] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "AutoCodeRover: Autonomous program improvement," in *Proc. 33rd ACM SIGSOFT Int. Symp. Software Testing and Analysis*, 2024, pp. 1592–1604. doi: 10.1145/3650212.3680384.

[22] N. Mündler, M. N. Müller, J. He, and M. Vechev, "SWT-Bench: Testing and validating real-world bug-fixes with code agents," in *Advances in Neural Information Processing Systems*, vol. 37, 2024, pp. 81857–81887. doi: 10.52202/079017-2601.

[23] W. Li et al., “FEA-Bench: A benchmark for evaluating repository-level code generation for feature implementation,” in *Proc. 63rd Annu. Meeting Association for Computational Linguistics*, vol. 1, 2025, pp. 17160–17176. doi: 10.18653/v1/2025.acl-long.839.

[24] J. Liu et al., “M2RC-EVAL: Massively multilingual repository-level code completion evaluation,” in *Proc. 63rd Annu. Meeting Association for Computational Linguistics*, vol. 1, 2025, pp. 15661–15684. doi: 10.18653/v1/2025.acl-long.763.

[25] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz, “Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection,” in *Proc. 16th ACM Workshop on Artificial Intelligence and Security*, 2023, pp. 79–90. doi: 10.1145/3605764.3623985.

[26] W. Zou, R. Geng, B. Wang, and J. Jia, “PoisonedRAG: Knowledge corruption attacks to retrieval-augmented generation of large language models,” in *Proc. 34th USENIX Security Symp.*, 2025, pp. 3827–3844. [Online]. Available: https://www.usenix.org/conference/usenixsecurity25/presentation/zou-poisonedrag

[27] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, “Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions,” in *Proc. IEEE Symp. Security and Privacy*, 2022, pp. 754–768. doi: 10.1109/SP46214.2022.9833571.

[28] H. Snyder, “Literature review as a research methodology: An overview and guidelines,” *Journal of Business Research*, vol. 104, pp. 333–339, 2019. doi: 10.1016/j.jbusres.2019.07.039.