ADR-Control: Do LLM Coding Agents Follow Architectural Decisions? An Evaluation of ADR-Grounded Code Generation

Pham The Phong[1], **Phan Duy Hung**[1,*],
[1]FPT University, Hanoi, Vietnam,
Co-author Emails: Phong24MSE13184@fsb.edu.vn, HungPD2@fe.edu.vn
Correspondence: HungPD2@fe.edu.vn

Abstract

Large language models can generate code at repository scale, but they frequently violate architectural intent even when relevant design decisions are available as context. This gap mirrors a broader faithfulness problem in natural language generation: models retrieve correct evidence yet fail to apply it faithfully in their outputs. We propose *ADR-Control*, a framework that elevates Architecture Decision Records (ADRs) from passive documentation to active control artifacts through retrieval, constraint normalization, verification, and typed trace emission. We evaluate one bounded component of this framework: canonical ADR-constraint violation identification on fixed candidate patches under four reviewer-context conditions across five tasks from one .NET repository. Across two canonical repetitions, Structured-ADR context achieves mean canonical identification F1 of 0.746 ± 0.145, compared with 0.675 ± 0.194 for ADR-prose context. This result provides pilot evidence for bounded ADR-grounded constraint identification; it does not establish end-to-end generation superiority, verifier-gated repair success, executable correctness, or generalization beyond the pilot.

1 Introduction

Large language models are rapidly moving from snippet-level code completion toward repository-level software generation, issue resolution, and tool-mediated software maintenance (Chen et al., 2021; Jimenez et al., 2024; Liu et al., 2025). However, success on implementation-oriented tasks does not guarantee preservation of a system's architectural intent. Generated code may satisfy local functional requirements while violating architectural decisions about layering, dependencies, integration style, persistence, messaging, or process boundaries (Su et al., 2026). This gap is especially costly in multi-developer and outsourced environments, where the rationale behind prior architectural choices is often under-documented, fragmented, or lost over time (Buchgeher et al., 2023; Ahmeti et al., 2024).

Architecture Decision Records (ADRs) are a lightweight mechanism for capturing architecturally significant choices, their context, alternatives, and consequences (Tyree and Akerman, 2005; Zimmermann, 2011; Buchgeher et al., 2023). Prior work has shown that design decisions should be treated as first-class architectural knowledge, because architecture is not merely structure but also the documented reasoning that explains why particular structures were chosen and which alternatives were rejected (Jansen and Bosch, 2005; ISO/IEC/IEEE, 2011). Yet in most AI-assisted software generation settings, ADRs often remain passive documents: they are archived, occasionally retrieved, and seldom enforced (Buchgeher et al., 2023; Ahmeti et al., 2024).

Retrieval-augmented generation (RAG) partially addresses the grounding problem by injecting external evidence into model inference. In software engineering, retrieval has improved code understanding and repository-level generation by supplying models with localized files, APIs, documentation, and tests (Tao et al., 2026; Zhang et al., 2023; Lu et al., 2022). Still, retrieval alone is not equivalent to control. Models can retrieve the correct evidence yet fail to apply it faithfully, omit key constraints, or contradict the retrieved source in the final output (Ji et al., 2023; Huang and Huang, 2024). This faithfulness gap is well-documented in natural language generation: Es et al. (2024) show that reference-free evaluation reveals systematic failures in grounding generated text to retrieved context, while Asai et al. (2024) demonstrate that models can learn to self-reflect on whether their outputs are supported by evidence. Recent work on guideline adherence further demonstrates that even when GPT models follow human summariza-

Proceedings of the 19th International Natural Language Generation Conference: Long Papers, pages 301–309
October 17-21, 2026 ©2026 Association for Computational Linguistics

tion guidelines for targeted communication goals, automatic metrics like ROUGE and BERTScore fail to capture guideline adherence, misaligning with human judgment (Zhou et al., 2025). Similarly, our work evaluates whether generated artifacts follow explicit human-authored architectural guidelines rather than only optimizing local correctness. That gap motivates a different design: treat ADRs as retrieval targets *and* control artifacts inside the generation loop, not as passive documentation. Recent studies have demonstrated both the feasibility of ADR generation and the detection of ADR violations, but not the enforcement of ADR constraints during generation (Dhar et al., 2024; Su et al., 2026).

To close this gap, this paper argues that ADRs should be moved from the periphery of agentic software generation into the control loop itself. We propose *ADR-Control*, an architecture-grounded agentic RAG framework in which approved ADRs are (i) retrieved as decision evidence, (ii) normalized from declared constraint records into explicit obligations and prohibitions, (iii) checked by a verifier after generation using explicit detector rules, and (iv) linked through typed traces to requirements, code edits, tests, and verification outcomes. In this framing, ADRs are neither generic context nor post hoc explanation; they are operationalized as bounded, machine-checkable control artifacts. The central question is not only whether generated code works, but whether the system can justify its outputs against explicit architectural evidence.

This paper does not claim novelty in RAG, ADR documentation, architecture conformance checking, or requirements traceability as isolated techniques. Instead, it combines these established lines of work into an ADR-centered control mechanism for agentic software generation. The paper makes three contributions. First, it introduces a compact structured ADR artifact containing accepted constraints, rejected alternatives, verification obligations, source references, and status information. Second, it specifies *ADR-Control*, a proposed loop connecting ADR-aware retrieval, constraint normalization, patch generation, rule-based verification, and typed trace emission. Third, it reports a controlled pilot of one component of that loop: fixed-patch reviewer-context evidence over five purposively selected tasks. The pilot evaluates whether structured ADR evidence improves detection of annotated architectural constraint violations; it does not claim that the complete loop generates better patches or repairs them more effectively.

2 Background and Related Work

2.1 Faithful and Grounded Generation

Retrieval-augmented generation combines parametric generation with explicit retrieval over external knowledge bases (Lewis et al., 2020; Guu et al., 2020; Karpukhin et al., 2020). Surveys emphasize that retrieval quality and grounding behavior are central to downstream trustworthiness (Gao et al., 2023; Huang and Huang, 2024), yet even well-grounded RAG systems exhibit faithfulness failures (Ji et al., 2023). RAGAS provides reference-free evaluation metrics (Es et al., 2024), RAGTruth provides annotated evidence for analysing groundedness failures (Niu et al., 2024), while Self-RAG integrates retrieval with self-reflection during generation (Asai et al., 2024). Together, these lines of work motivate our use of an explicit control representation for architectural evidence. In software engineering, retrieval-augmented code generation systems like RepoCoder and ReACC improve repository-level completion (Zhang et al., 2023; Lu et al., 2022) but optimize for implementation-level correctness without explicit architectural control.

2.2 Constrained and Structured Output Generation

Constrained text generation has been formulated through energy-based frameworks (Qin et al., 2022), strict constraint satisfaction with rejection sampling (Kim et al., 2025), and grammar-constrained decoding via finite-state machines (Willard and Louf, 2023). Multi-dimensional evaluation frameworks such as MulDimIF (Ye et al., 2025) demonstrate that constraint adherence degrades significantly with difficulty. Prior constrained-generation work commonly operationalizes lexical, grammatical, structural, or decoding-time constraints. ADR-Control instead focuses on project-specific semantic software constraints encoded in approved decision records, including obligations, prohibitions, and rejected alternatives.

2.3 Architecture-Aware Evaluation

Architectural decisions have long been recognized as a core unit of architectural knowledge (Tyree and Akerman, 2005; Jansen and Bosch, 2005; Zimmermann, 2011). Recent work explores whether LLMs can draft or validate ADRs (Dhar et al., 2024, 2025;

Gupta et al., 2026; Su et al., 2026), but focuses on ADR generation or violation detection rather than enforcing constraints during downstream generation. Reflexion modeling (Murphy et al., 2001) is the closest analogue to our verifier, but assumes pre-authored architecture models and post-hoc auditing. ADR-Control derives constraints from approved ADR records and applies verification inside the generation loop.

Evaluation beyond local correctness is increasingly recognized as critical (Liu et al., 2023; Zheng et al., 2024). HumanEval and SWE-bench established code-centric standards (Chen et al., 2021; Jimenez et al., 2024), while ArchBench (Adnan et al., 2026) argues that architecture-specific dimensions remain underrepresented. Requirements traceability research argues that artifacts should remain linked across requirements, design, code, and test levels (Gotel and Finkelstein, 1994; Cleland-Huang et al., 2012). ADR-Control contributes an architecture-aware evaluation protocol measuring ADR retrieval quality, obligation fulfillment, violation rate, rejected-alternative reintroduction, and trace F1.

3 ADR-Control System Design

3.1 Overview

ADR-Control wraps software generation with four operations: ADR-aware retrieval, constraint normalization, architecture-aware verification, and typed trace emission. The central idea is that approved ADRs should not remain passive documentation but should become active control artifacts that constrain generation at runtime.

The runtime is driven by explicit artifacts rather than hidden prompt state. The **requirement record** defines the requested behavior, acceptance criteria, affected components, and source reference. The **ADR constraint record** encodes accepted decision constraints, rejected alternatives, verification obligations, affected components, rule identifiers, and detector hints. The **task record** binds the requirement to a repository snapshot and runtime-visible retrieval inputs. The **trace edge record** stores typed links among requirements, ADRs, files, tests, and verification evidence. This artifact contract is also the anti-leakage boundary: model-visible inputs contain the requirement, retrieved repository context, retrieved ADR evidence, and normalized constraints, while evaluator-only labels (gold ADR sets and gold trace edges) remain hidden during generation.

3.2 Control Pipeline

Figure 1 contrasts ADR-Control with the Basic-RAG baseline. Basic-RAG retrieves repository context and sends it directly to the generator. ADR-Control adds a separate ADR-control path that retrieves accepted ADRs, normalizes declared constraints, verifies generated artifacts, and emits typed trace links. For fairness, Basic-RAG must not parse ADR constraint records or run the verifier. ADR-Control loads constraints only from ADRs retrieved at runtime, avoiding retrieval leakage.

ADR-aware retrieval uses a hybrid design combining dense and sparse ADR retrieval with metadata overlap and ADR-status weighting. **Constraint normalization** converts declared ADR constraint records into verifier-ready bundles via deterministic section parsing over `Constraint Record`, `Rejected Alternatives`, and `Verification`.

The control path is intentionally separated from ordinary repository retrieval. Repository retrieval provides implementation context such as relevant files, APIs, and tests. ADR-aware retrieval instead searches over accepted decision records and returns decision evidence with status metadata, affected components, and declared constraints. This separation matters for evaluation: Basic-RAG may still retrieve useful repository text, but it does not receive the normalized obligation/prohibition representation used by the verifier. Conversely, ADR-Control cannot use gold ADR labels directly; it can only enforce constraints from ADRs retrieved at runtime.

Constraint normalization makes the governing decision explicit enough to check without relying on a second LLM judge. A normalized bundle contains constraint identifiers, rule identifiers, severity, affected scopes, rejected-alternative detectors, and verification obligations. In the callback tasks, for example, this converts a prose decision such as "dispatch callbacks asynchronously using a pooled HTTP client" into a must-use rule requiring enqueue-and-dispatch behavior, a prohibition on inline synchronous callbacks, a rule requiring a client factory, and tests that should be present or planned. The bundle is still bounded and human-authored: ADR-Control does not invent new architecture rules beyond those recorded in the accepted ADR.

Generation then receives three kinds of infor-

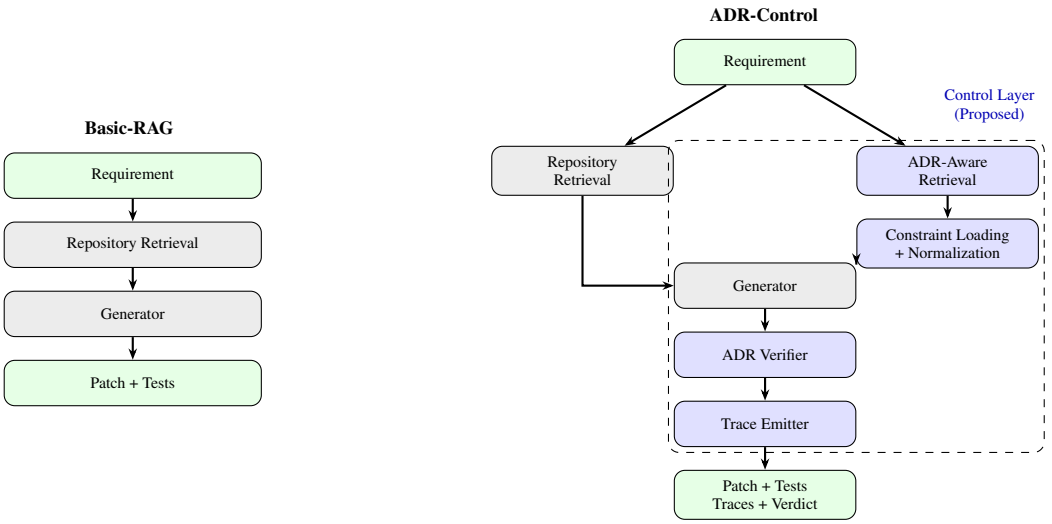


Figure 1: Proposed ADR-Control loop, compared with Basic-RAG. Gray blocks are adapted components; blue blocks are proposed ADR-Control components. The camera-ready pilot evaluates only the effect of reviewer-visible ADR evidence on fixed-patch constraint identification; generation, verifier gating, repair, and trace emission are not evaluated end-to-end.

mation: the user requirement, retrieved repository context, and retrieved ADR evidence plus its normalized constraints. The generator is asked to produce a patch or implementation plan that satisfies the requirement while respecting the loaded constraints. The verifier runs after generation and can reject outputs that appear functionally plausible but violate the controlling decision. This design follows the NLG distinction between retrieval and faithful use of retrieved evidence: a model may see the right evidence yet still omit, weaken, or contradict it in the generated artifact.

3.3 Verifier and Trace Emitter

The **ADR-Control runtime verifier** is implemented as an auditable detector pass over generated artifacts, following the spirit of architecture conformance checking but grounded in ADR-derived rules rather than in a separate architecture model (Murphy et al., 2001). Each obligation or prohibition is associated with an explicit detector specification, such as required file changes, required API usage, forbidden dependency patterns, forbidden synchronous calls, required tests, or rejected-alternative keywords. The runtime verifier records evaluated constraint IDs, violated constraint IDs, per-obligation pass/fail results, and rejected-alternative checks. This makes its operational output reproducible from stored task, constraint, patch, and verifier-result artifacts. Within the proposed ADR-Control runtime, verifier findings are derived from explicit detector rules over loaded constraint records. The canonical experiment in Section 4 is separate: an LLM reviewer inspects a fixed candidate patch under different evidence conditions, and an evaluator-side scorer compares the reviewer findings against gold constraint annotations.

The detector rules are deliberately conservative and artifact-specific. For obligations, the verifier checks whether expected tests, implementation references, or design commitments are present. For prohibitions, it checks for patterns such as direct `HttpClient` construction, synchronous dispatch from request-handling code, cross-partition discovery, or rejected retry loops. For rejected alternatives, the detector records a separate reintroduction flag so that a patch can be distinguished between merely incomplete and actively returning to a rejected design. This distinction is important for architecture-aware generation: a patch that omits a test is different from one that revives a decision explicitly ruled out by an ADR.

Typed trace emission constructs typed confidence-scored edges connecting requirement, ADR, code, test, and verification artifacts. The trace emitter emits a trace graph for auditability, linking requirements to ADRs, ADRs to code edits, requirements to tests, and verification records to their sources. The intended effect is not to prove architectural correctness in a formal methods sense. Its practical goal is narrower: improve the odds that generation remains consistent with governing decisions, surface deviations early, and produce

enough structured evidence for human review.

Trace edges are evaluated separately from constraint compliance because they answer a different question. A generated patch can satisfy an ADR yet provide little audit trail, or it can mention many artifacts while violating a hard constraint. The trace graph therefore records typed links such as REQ-TO-ADR, ADR-TO-FILE, ADR-TO-TEST, and VERIFICATION-TO-CONSTRAINT. Precision, recall, and F1 are computed against evaluator-only gold trace edges at confidence ≥ 0.70. This gives the reviewer a way to inspect not only whether the generated artifact appears compliant, but also whether the system can explain which requirement, decision, file, test, and verification evidence support that compliance.

4 Experiments and Results

4.1 Pilot Dataset and Task Selection

We report a feasibility pilot on five purposively selected change tasks from one real .NET repository, linked to five accepted ADRs: ADR-0002 (off-peak scheduling), ADR-0003 (company ID registry authorization), ADR-0004 (asynchronous webhook dispatch), ADR-0005 (webhook callback pattern), and ADR-0007 (stale batch reconciliation). The tasks cover distinct architectural constraint types rather than forming a statistically representative benchmark. The primary annotation universe contains 19 evaluator-annotated ADR constraints.

4.2 Reviewer-Context Protocol

The canonical experiment evaluates canonical ADR-constraint violation identification on fixed candidate patches, not patch generation. For each task, the same Basic-RAG candidate patch is reviewed under four evidence conditions; only the evidence supplied to the reviewer changes. The conditions are *Requirement-only* (historical label: No-RAG; requirement and patch), *Repository-context* (Basic-RAG; repository context and patch), *ADR-prose* (ADR-as-context; repository context, ADR prose, and patch), and *Structured-ADR* (repository context, normalized constraints, and patch). Across five tasks, four conditions, and two complete repetitions, the experiment produces 40 review outputs. Gold constraint labels remain evaluator-only. We retain the historical labels in parentheses to identify the provenance of each evidence packet, not to imply four end-to-end system configurations.

The bounded task packet contains the requirement, candidate patch, selected repository evidence where applicable, and either ADR prose or normalized constraints. It does not expose gold labels, verifier results, or hidden trace annotations. The preserved configuration records the runtime identifier `opencode-go/gpt-5.6-luna` and a maximum output of 4000 tokens; the provider-side model mapping and unretained sampling parameters were not separately retained and are not reconstructed. This design separates evidence supplied to the reviewer from evaluator-only scoring artifacts.

4.3 Conditions and Metrics

The primary metric is constraint-only F1 over the 19 annotated ADR constraints. For each repetition, true positives, false positives, and false negatives are pooled across the five tasks before computing precision, recall, and F1. We report the mean $\pm$ sample standard deviation across the two complete repetitions and make no significance claim. More precisely, this metric measures exact recovery of evaluator-annotated architectural constraint IDs, or *canonical ADR-constraint violation identification*. It does not measure semantic constraint discovery independent of the canonical labels, patch quality, verifier-gated repair, build success, test success, verification-obligation fulfillment, or complete traceability.

4.4 Reviewer-Context Ablation

To isolate the contribution of reviewer-visible architectural evidence, we hold each candidate patch fixed and vary only the evidence supplied to the reviewer. The four conditions progressively expose the requirement and patch alone, repository context, ADR prose, and normalized ADR constraints. Gold constraint labels remain evaluator-only. This design isolates the association between evidence representation and canonical ADR-constraint identification; it is not an end-to-end component ablation of generation, verifier gating, repair, execution, or trace emission.

Table 1 shows that Structured-ADR context achieved the highest observed mean canonical identification F1 (0.746 ± 0.145), compared with 0.675 ± 0.194 for ADR-prose. Precision was perfect in both ADR-informed conditions; the difference arose mainly from recall. This result is consistent with the hypothesis that normalized constraint evidence helps a reviewer recover more annotated constraint IDs than ADR prose alone. Because

Table 1: Reviewer-context ablation for canonical ADR-constraint identification over 19 annotated constraints, five tasks, and two canonical repetitions. The same fixed candidate patch is reviewed under each evidence condition. Values are mean ± sample SD across repetitions.

Condition	Precision	Recall	F1
Requirement-only	0.000 ± 0.000	0.000 ± 0.000	0.000 ± 0.000
Repository-context	0.000 ± 0.000	0.000 ± 0.000	0.000 ± 0.000
ADR-prose	1.000 ± 0.000	0.526 ± 0.223	0.675 ± 0.194
Structured-ADR	1.000 ± 0.000	0.605 ± 0.186	0.746 ± 0.145

Table 2: Task-level canonical constraint recovery. Each cell reports correctly identified constraints in repetitions 1/2; N is the number of evaluator-annotated constraints for the task.

Task	ADR	N	ADR-prose	Structured-ADR
T1	0002	3	1/3	3/3
T2	0003	3	3/1	0/3
T3	0004	3	0/0	1/0
T4	0005	3	3/2	3/3
T5	0007	7	0/7	7/0
Total		19	7/13	14/9

the candidate patch is fixed, the result concerns reviewer-context identification rather than generation or repair.

A full runtime component ablation remains future work. It would require a genuinely distinct No-verifier condition and an ADR-Control condition that executes the verifier-gated accept/reject or repair path; the present artifacts do not support attributing an effect to verifier gating.

The zero Requirement-only and Repository-context scores mean that those conditions recovered no exact evaluator-annotated constraint IDs under the scorer. They do not imply that reviewers identified no generic architectural problems. The scorer deliberately measures recovery of declared ADR constraints, which makes the result auditable but narrower than general defect detection. The two repetitions also support descriptive comparison only; no significance claim is warranted.

Table 2 exposes substantial task- and repetition-level variation hidden by the aggregate mean. Structured-ADR recovered at least as many constraints as ADR-prose on T1 and T4 in both repetitions, but neither condition dominated every task. On T5, for example, Structured-ADR recovered all seven annotated constraints in repetition 1 and none in repetition 2, while ADR-prose showed the reverse pattern. The seven T5 constraints span stale-job thresholds, timeout handling, API-failure degradation, a per-run limit, partial-result recovery,

Table 3: Scope of preserved experimental evidence. Generation and execution evidence are reported separately from the canonical fixed-patch reviewer experiment. Only the canonical reviewer experiment contributes to the primary quantitative result.

Evidence set	Artifacts	Evidence supported
Canonical reviewer	40 review outputs	Fixed-patch review only
Phase-2 generation	20 patch artifacts	Validation stopped before build/test
ADR-0003 extension	Bounded subset	C01/C02 partial execution; C03 static audit

immediate failure, and webhook notification. Thus, explicit constraint records can support exhaustive canonical identification, but they do not ensure stable reviewer attention. This qualitative case supports the narrower interpretation of Table 1: the observed aggregate difference is preliminary and coexists with high run-level variability.

The earlier Trace F1 value of 0.42 is retained only as a secondary historical diagnostic: no edge-type breakdown was preserved, and the canonical reviewer experiment does not evaluate complete traceability. Likewise, the phase-2 generation artifacts do not support clean-worktree success claims because validation stopped with `test_project_not_found`. A bounded ADR-0003 extension provides partial executable evidence for C01 and C02 and a static audit for C03, but it is not a comparative end-to-end evaluation. These evidence boundaries prevent the fixed-patch result from being misread as proof that the complete ADR-Control loop generates or repairs better implementations.

4.5 Analysis and Discussion

The result supports a narrow context association: the observed mean was higher for exact recovery of annotated constraint IDs under normalized ADR constraints than under ADR prose in this pilot. It does not isolate the effects of ADR retrieval, generation, verifier gating, repair, or trace emission because the candidate patch was held fixed and the reviewer evidence was varied. The runtime verifier and the evaluator-side scorer serve different roles. The former operationalizes ADR constraints for the proposed control loop, whereas the latter scores reviewer findings against annotated constraints. Neither constitutes an independent oracle of architectural correctness, and the scorer's exact-ID matching can miss semantically equivalent findings. Because the Structured-ADR packet exposes canoni-

cal constraint representations while the scorer requires canonical IDs, the comparison should be interpreted as evidence about ADR-specific constraint identification, not as evidence that generic reviewers cannot detect architectural defects.

This bounded result reinforces the paper's NLG framing. Retrieved evidence is not necessarily applied evidence, and prose context may leave constraints implicit. Normalization makes obligations and prohibitions more explicit for a reviewer, but the two-repetition design and the difficult multi-constraint tasks show why the effect should be treated as preliminary. A larger study should add independent annotation, repeated end-to-end generation, valid execution evidence for every comparison condition, and human assessment by software engineers or architects.

5 Conclusion

ADR-Control is an ADR-centered control framework that treats accepted decisions as structured runtime artifacts rather than passive documentation. The corrected pilot evaluates one bounded component of this proposal: fixed-patch canonical constraint identification under reviewer-context conditions. Structured-ADR context achieved the highest observed mean canonical identification F1 (0.746 ± 0.145), compared with 0.675 ± 0.194 for ADR-prose context. This result supports the feasibility of making architectural constraints more explicit for review, but it does not establish end-to-end generation superiority, verifier-gated repair success, executable correctness, complete traceability, or cross-repository generalization.

Limitations

This study is a pilot and should not be interpreted as benchmark-scale evidence. It covers five purposively selected tasks from one repository and one architectural domain. The canonical experiment holds candidate patches fixed and varies reviewer context; it therefore measures ADR-grounded constraint identification rather than end-to-end code generation quality or repair success. The runtime verifier operationalizes declared ADR constraints through explicit detector rules, whereas the evaluator-side scorer compares LLM-reviewer findings with annotated labels; neither is an independent oracle of architectural correctness. The evidence lacks second-annotator adjudication, repeated end-to-end runs, full clean-worktree execution across all compared conditions, and human assessment by software engineers. These limitations motivate larger studies across repositories, ADR families, programming languages, and organizational contexts.

Ethical Considerations

This work does not involve human subjects, personal data, or harmful content generation. The framework operates on developer-authored Architecture Decision Records and repository code. Execution-only validation was performed locally and sent no source code to a remote model. Reviewer-context inference used only the bounded experimental packet described in Section 4.2; it did not expose evaluator-only labels or hidden annotations.

References

Bushra Adnan, Ankit Gupta, Srinivas Akshathala, et al. 2026. ArchBench: Benchmarking generative-AI for software architecture tasks. arXiv preprint arXiv:2603.17833. *Preprint*, arXiv:2603.17833.

Blerta Ahmeti, Matthias Linder, Roger Groner, et al. 2024. Architecture decision records in practice: An action research study. In *European Conference on Software Architecture*, pages 333–349. Springer.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2024. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *The Twelfth International Conference on Learning Representations*.

Georg Buchgeher, Stefan Schoberl, Viktor Geist, et al. 2023. Using architecture decision records in open source projects: An MSR study on GitHub. *IEEE Access*, 11:63725–63740.

Mark Chen, Jerry Tworek, Heewoo Jun, et al. 2021. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374. *Preprint*, arXiv:2107.03374.

Jane Cleland-Huang, Oliver Gotel, and Andrea Zisman, editors. 2012. *Software and Systems Traceability*. Springer.

Rishabh Dhar, Ankit Kakran, Arjun Karan, et al. 2025. DRAFT-ing architectural design decisions using LLMs. arXiv preprint arXiv:2504.08207. *Preprint*, arXiv:2504.08207.

Rishabh Dhar, Kripa Vaidhyanathan, and Vasudeva Varma. 2024. Can LLMs generate architectural design decisions? an exploratory empirical study. arXiv preprint arXiv:2403.01709. *Preprint*, arXiv:2403.01709.

Shahul Es, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. 2024. RAGAS: Automated evaluation of retrieval augmented generation. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*. Association for Computational Linguistics.

Yunfan Gao, Yun Xiong, Xinyu Gao, et al. 2023. Retrieval-augmented generation for large language models: A survey. arXiv preprint arXiv:2312.10997. *Preprint*, arXiv:2312.10997.

Oliver C. Z. Gotel and Anthony C. W. Finkelstein. 1994. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering*, pages 94–101.

Ankit Gupta, Rishabh Dhar, Daniel Feitosa, et al. 2026. Context matters: Evaluating context strategies for automated ADR generation using LLMs. arXiv preprint arXiv:2604.03826. *Preprint*, arXiv:2604.03826.

Kelvin Guu, Kenton Lee, Zora Tung, et al. 2020. REALM: Retrieval-augmented language model pretraining. In *Proceedings of the 37th International Conference on Machine Learning*, pages 3929–3938.

Yujian Huang and Jiajun Huang. 2024. A survey on retrieval-augmented text generation for large language models. arXiv preprint arXiv:2404.10981. *Preprint*, arXiv:2404.10981.

ISO/IEC/IEEE. 2011. ISO/IEC/IEEE 42010:2011: Systems and software engineering—architecture description. Technical report, International Organization for Standardization.

Anton Jansen and Jan Bosch. 2005. Software architecture as a set of architectural design decisions. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pages 109–120.

Ziwei Ji, Nayeon Lee, Rita Frieske, et al. 2023. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):248:1–248:38.

Carlos E. Jimenez, John Yang, Alexander Wettig, et al. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, et al. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*, pages 6769–6781.

Minbeom Kim, Thibaut Thonet, Jos Rozen, Hwaran Lee, Kyomin Jung, and Marc Dymetman. 2025. Guaranteed generation from large language models. In *The Thirteenth International Conference on Learning Representations*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, et al. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474.

Jiawei Liu, Kai Wang, Yifan Chen, et al. 2025. Large language model-based agents for software engineering: A survey. arXiv preprint arXiv:2409.02977. *Preprint*, arXiv:2409.02977.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. G-Eval: NLG evaluation using GPT-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2511–2522.

Shuai Lu, Nan Duan, Hojae Han, et al. 2022. ReACC: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*. ArXiv:2203.07722.

Gail C. Murphy, David Notkin, and Kevin J. Sullivan. 2001. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380.

Cheng Niu, Yuanhao Wu, Juno Zhu, Siliang Xu, Kashun Shum, Randy Zhong, Juntong Song, and Tong Zhang. 2024. RAGTruth: A hallucination corpus for developing trustworthy retrieval-augmented language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Lianhui Qin, Sean Welleck, Daniel Khashabi, and Yejin Choi. 2022. COLD decoding: Energy-based constrained text generation with Langevin dynamics. In *Advances in Neural Information Processing Systems*, volume 35, pages 9689–9701.

Rui Su, Anton Bakhtin, Nisar Ahmad, et al. 2026. Evaluating large language models for detecting architectural decision violations. arXiv preprint arXiv:2602.07609. *Preprint*, arXiv:2602.07609.

Yifan Tao, Yujia Qin, and Yiming Liu. 2026. Retrieval-augmented code generation: A survey with focus on repository-level approaches. arXiv preprint arXiv:2510.04905. *Preprint*, arXiv:2510.04905.

Jeff Tyree and Arnie Akerman. 2005. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(2):19–27.

Brandon T. Willard and Rémi Louf. 2023. Efficient guided generation for large language models. arXiv preprint arXiv:2307.09702. *Preprint*, arXiv:2307.09702.

Junjie Ye, Caishuang Huang, Zhuohan Chen, Wenjie Fu, Chenyuan Yang, Leyang Yang, Yilong Wu, Peng Wang, Meng Zhou, Xiaolong Yang, Tao Gui,

Qi Zhang, Zhongchao Shi, Jianping Fan, and Xuanjing Huang. 2025. MulDimIF: A multi-dimensional constraint framework for evaluating and improving instruction following in large language models. arXiv preprint arXiv:2505.07591. *Preprint*, arXiv:2505.07591. To appear at ACL 2026.

Fengji Zhang, Bei Chen, Yan Zhang, et al. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484.

Jiasheng Zheng, Boxi Cao, Zhengzhao Ma, Ruotong Pan, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. 2024. Beyond correctness: Benchmarking multi-dimensional code generation for large language models. arXiv preprint arXiv:2407.11470. *Preprint*, arXiv:2407.11470.

Yongxin Zhou, Fabien Ringeval, and François Portet. 2025. Can GPT models follow human summarization guidelines? A study for targeted communication goals. In *Proceedings of the 18th International Natural Language Generation Conference*, pages 249–273. Association for Computational Linguistics. Best Evaluation Award, INLG 2025.

Olaf Zimmermann. 2011. Architectural decisions as reusable design assets. *IEEE Software*, 28(1):64–69.