Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

WriterKnowsBest: Evidence-Bound Writing for Faithful Repository-Level Documentation

Yaokai Zu
University of Electronic Science and Technology of China
Tongtong Wang
ICBC Software Development Center
Danfeng Zhang
ICBC Software Development Center
Jiayu Zhang
zhangjy02@sdc.icbc.com.cn

ICBC Software Development Center

Research Article

Keywords:

Posted Date: November 27th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11565916/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

WriterKnowsBest: Evidence-Bound Writing for Faithful Repository-Level Documentation

Yaokai Zu[1,2†], Tongtong Wang[2†], Danfeng Zhang[2], Jiayu Zhang[2*]

[1]University of Electronic Science and Technology of China, Chengdu, Sichuan, China.

[2]ICBC Software Development Center, Hangzhou, Zhejiang, China.

*Corresponding author(s). E-mail(s): zhangjy02@sdc.icbc.com.cn;
Contributing authors: yaokaizu@std.uestc.edu.cn;
wangtt2@sdc.icbc.com.cn; zju_zdf@163.com;
†These authors contributed equally to this work.

Abstract

Repository documentation must describe implemented behavior and cover the public application programming interfaces (APIs) developers need. Detailed descriptions may go beyond the available evidence, while cautious ones may omit useful content. We present WriterKnowsBest, an evidence-bound approach that requires explicit claim attribution and coverage during generation. The system records repository reads, and the writer binds checkable claims to these records during drafting. Separate coverage obligations identify the content to address, and typed checks validate structural facts and usage examples before delivery. Claim–evidence associations are available for inspection throughout generation. We evaluate the approach on four Java repositories comprising 31 matched components, 62 documents, and 3,808 atomic facts. Compared with RepoDoc, weighted atomic support increases from 0.844 to 0.924 and public-API coverage from 87.3% to 99.0%. When the same generator is required to cover repository facts and link its claims to evidence, the atomic support rate rises from 0.871 to 0.904. Of 223 audited bindings, 88.8% have full or partial evidentiary support. These results suggest that explicit evidence and coverage constraints can improve factual support and completeness together within the studied setting.

Keywords: Repository-level documentation, Large language models, Evidence-bound writing, Code documentation generation, Factuality evaluation

1 Introduction

A method that returns `null` can receive an elaborate description. In our evaluation corpus, an authentication stub is described as providing the application's core authentication and registration logic, with usage advice for behavior absent from the implementation. The description is plausible for an authentication service but misrepresents the repository. A developer relying on it would expect capabilities the software does not provide.

Repository-level documentation must explain the interfaces and relationships needed to understand a component and ground its claims in source. Adjusting prompt strictness alone does not readily satisfy both requirements. Asking for detail can lead to unsupported elaboration; asking for caution can leave public interfaces undocumented or replace substantive sections with `N/A`. We therefore ask how to improve factual support while preserving useful coverage.

Recent systems use repository context, retrieval, and verification to improve documentation generation (Luo et al. 2024; Yang et al. 2025; Hoang et al. 2026; Xu et al. 2026; Panchal et al. 2026). Even when evidence is available, a reviewer still needs to know which evidence is associated with each claim. Without a recorded association, the reviewer must reconstruct it from the completed document and the repository. Recording source access and claim attribution during generation makes these associations available as the draft develops.

We introduce WriterKnowsBest, an approach we call *evidence-bound writing*. Each repository read produces a system-issued record, or receipt. The writer binds checkable claims to receipts obtained during drafting. Separate coverage obligations identify the enumerable facts the document must address. Before delivery, the system checks these obligations and uses symbol resolution or compilation where the claim type permits it. The writer is free to explore the repository and compose an explanation, but generation must satisfy explicit attribution and coverage requirements.

We evaluate factual support, coverage, and the evidentiary quality of the bindings. Across four Java repositories and 31 matched components, WriterKnowsBest increases weighted atomic support from 0.844 to 0.924 relative to RepoDoc, while public-API coverage increases from 87.3% to 99.0%. Comparing the two variants on the same components, we find that adding the full set of requirements increases the atomic support rate from 0.871 to 0.904. Of 223 audited bindings, 88.8% have full or partial support. These results support the use of explicit evidence and coverage constraints for faithful repository documentation.

We make three contributions:

- **A generation contract for evidence attribution.** We define how checkable claims are associated with system-recorded source reads before a document can be delivered.
- **Separate controls for support and coverage.** We combine claim bindings with coverage obligations and typed verification to check claim attribution and detect omissions of required repository facts.

- **Evaluation of factual support, coverage, and attribution.** We evaluate 62 documents containing 3,808 atomic facts, compare obligation variants on matched components, and audit the evidentiary support of 223 claim–evidence bindings.

Section 2 positions the approach within prior work. Section 3 defines the problem, Section 4 presents the method, and Section 5 reports the evaluation. Section 6 discusses the implications and limitations.

2 Related Work

2.1 Repository-level documentation generation

Documentation systems based on large language models (LLMs) range from single-function explanations (Khan and Uddin 2023; Ahmed and Devanbu 2022; Lomshakov et al. 2024; Makharev and Ivanov 2025) to repository-wide documentation (Luo et al. 2024; Yang et al. 2025; Hoang et al. 2026; Bae et al. 2026a; Venigalla and Chimalakonda 2025; Xu et al. 2026; Panchal et al. 2026), with benchmarks for evaluating their outputs (Wang et al. 2026; Hoang et al. 2026). LEDGE (Panchal et al. 2026) combines dependency graphs with retrieval to incorporate structural relationships into documentation. RepoDoc (Xu et al. 2026) uses a knowledge graph to generate documentation and support incremental updates. We study how claims are associated with recorded source access during generation. Retrieval supplies the evidence available to the writer; evidence-bound writing specifies the attribution and coverage requirements the draft must satisfy.

2.2 Factuality evaluation

For code-summary evaluation, SIDE uses contrastive learning to measure alignment between a summary and the semantics of its code, without relying on a reference summary (Mastropaolo et al. 2024). Atomic-fact decomposition (Min et al. 2023) and its long-form extensions (Wei et al. 2024; Bishop et al. 2024; Rajendhran et al. 2025) support fine-grained factuality assessment. Related approaches include natural language inference (NLI) and claim-based evaluation (Scirè et al. 2024; Iqbal et al. 2024; Wang et al. 2024) and code-specific detectors (Tian et al. 2025; Maharaj et al. 2025; Bae et al. 2026b). These methods rely heavily on model-based evaluators (Liu et al. 2023; Zheng et al. 2023; Chen et al. 2024; Wang et al. 2025; Crupi et al. 2025; Zhuge et al. 2025; Feng et al. 2025; Degiovanni et al. 2026). We use atomic-fact evaluation to measure source support. During generation, WriterKnowsBest also applies deterministic checks to enumerable code facts before delivery.

2.3 Attribution and citation

Retrieval-augmented generation supplies evidence as context (Lewis et al. 2020), while citation generation asks models to attribute claims to sources (Gao et al. 2023). Retrieved context alone does not establish support for an individual claim. During drafting, WriterKnowsBest records the association between each claim and system-issued

read records. At delivery, it checks these associations together with coverage obligations. Enforcing both recorded attribution and coverage distinguishes this approach.

Retrieval determines what evidence to consult, factuality assessment evaluates the explanation, and attribution associates its statements with sources. Evidence-bound writing connects them through a generation contract: repository reads are recorded, claims are bound to those reads, and delivery requires both bindings and resolved coverage obligations. We accordingly evaluate document support, interface coverage, and binding quality separately.

3 Problem Definition

3.1 Faithfulness and completeness

We consider documentation for a component in a repository frozen at a specified revision. The input consists of the component's source, related repository context, and available compiler artifacts. The output describes its responsibilities, public interfaces, relationships, and usage. A faithful document should not describe unimplemented behavior as an existing capability. Within our bounded task, a complete document should address the component's enumerable public interface and relevant structural obligations.

The authentication example illustrates a failure of faithfulness. In enterprise-banking-system, `AuthServiceImpl` is a stub whose only method returns `null`. The baseline document nevertheless describes authentication and registration logic and dependencies absent from the source. Of its 56 atomic facts, 26 concern future, unimplemented behavior, and 17 are fully supported; weighted atomic support is 0.393. The document describes a plausible application design in place of the actual implementation.

A document can also avoid unsupported statements yet omit existing public methods or mark applicable sections as `N/A`. We therefore assess support and coverage separately (Figure 1) to determine whether both can improve in the same setting.

3.2 The missing claim–evidence connection

Let E^v denote repository evidence at revision v, D a generated document, and O^v the enumerable facts to be addressed. Comparing D with E^v can reveal incorrect or unsupported statements. Without a generation trace, however, it cannot show which source material was presented to the writer for a particular claim. Our task is therefore to record an attribution relation between checkable claims and system-recorded repository reads, as well as to resolve O^v.

A read record identifies the evidence presented to the writer, and a binding associates that record with a claim. These records allow the system to detect missing claim bindings and unresolved coverage before delivery. Section 4.4 formalizes these requirements as a generation contract.

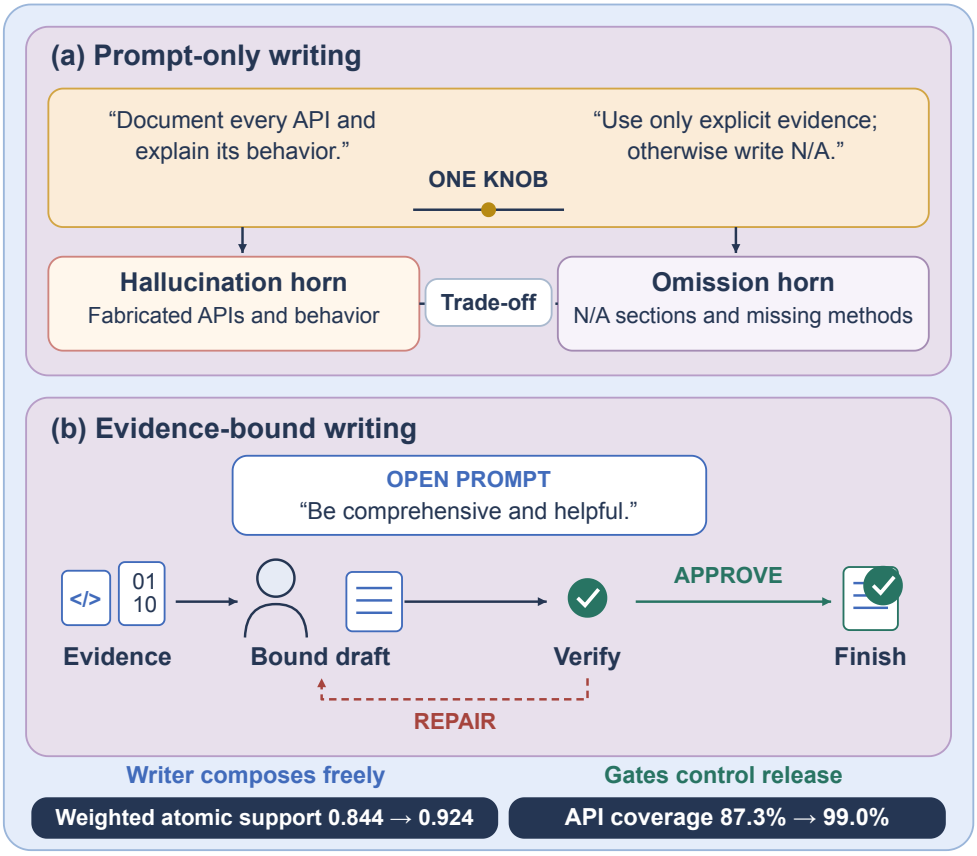

Fig. 1 Closing two failure modes without a trade-off. (a) Prompt-based controls can leave unsupported elaboration or omissions. (b) Evidence-bound writing associates claims with consulted evidence and separately tracks the repository facts that should be addressed

4 Methodology

4.1 Architecture overview

Figure 2 shows how WriterKnowsBest connects the writer's evidence, the claims in its draft, and the repository facts it must cover. A frozen repository supplies source files, compiler artifacts, and a graph of structural relations. While the writer explores this evidence and drafts the document, the system issues receipts for reads, the writer binds claims to receipts, and coverage obligations identify content that remains unaddressed. Delivery checks assess the draft against the generation contract.

Bindings record the evidence associated with written claims; coverage obligations identify what still needs to be documented, preventing omission from becoming a shortcut to delivery. Typed verification checks structural facts and usage examples against compiler artifacts or compilation results. Appendix A describes how the execution layer enforces these requirements.

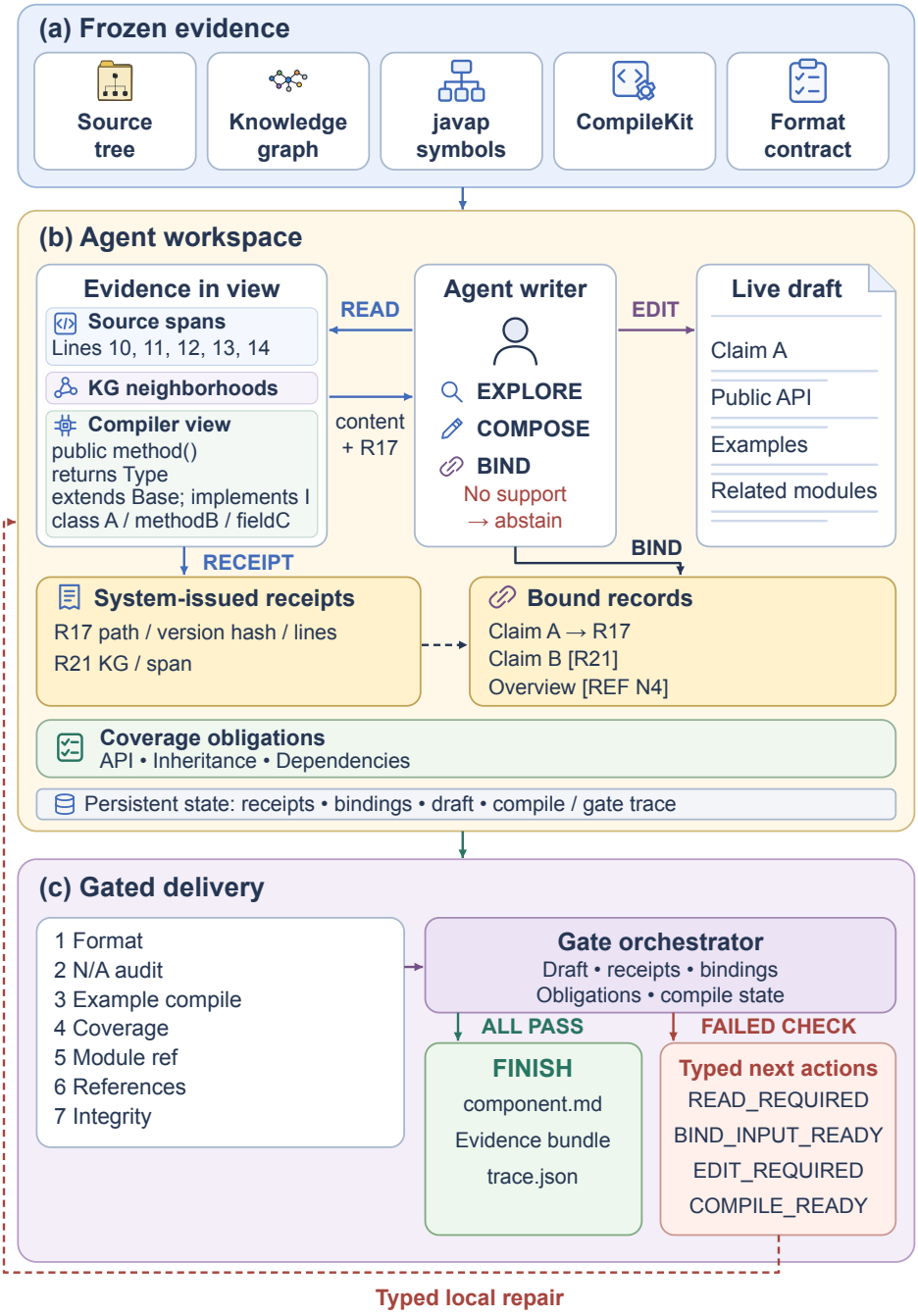

Fig. 2 Evidence-bound writing in WriterKnowsBest. (a) The frozen repository supplies evidence. (b) Repository reads produce receipts; the draft binds claims to those receipts and addresses coverage obligations. (c) Typed checks determine delivery eligibility and return repair requests when conditions remain unsatisfied. The dashed return path denotes typed local repair

4.2 Recording and binding repository evidence

When the writer accesses a source range, knowledge-graph neighborhood, or compiler artifact, the system returns the content with a receipt containing its path, revision, and line interval. A system-issued receipt attests only that the specified material entered the model context. Verification relies on these observable reads and the generated text, rather than on model-reported reasoning.

Receipts are versioned. A change to the underlying evidence invalidates the affected receipts and bindings so that re-verification and repair can be confined to the affected content.

Claim attribution.

The writer produces two types of binding. For precise, checkable claims, such as a method signature, return type, or inheritance relation, it produces a *provenance record* that binds a quoted claim to one or more receipt identifiers. For passages that synthesize information from multiple sources, such as overviews and responsibility summaries, it produces a *reference note* identifying the read materials that informed the passage. If no evidence binding is available, the writer either abstains or marks the statement as unsupported.

The binding gate rejects sections whose checkable claims lack evidence bindings. Structural claims are checked separately against compiler artifacts (§4.5), while the bindings preserve a claim-level evidence record for subsequent audit.

During drafting, the writer can bind a statement to evidence already in view, request another read, or revise the statement. A method signature can be bound to its source or compiler record; an explanation spanning several methods can retain multiple receipts. This allows the writer to synthesize an explanation and preserve the evidence associated with each claim. Reviewers can inspect the associations directly, without reconstructing them from the finished document.

4.3 Coverage obligations

An undocumented public method has no claim whose binding can be checked. Coverage obligations identify these omissions. The system enumerates the facts a component document must address from the parsed repository: the public API surface from compiler symbol tables rather than regular expressions, explicit inheritance and implementation relations, and knowledge-graph edges for direct dependencies and call paths. Each obligation contains a fact identifier, a recommended section, and source spans. The coverage gate blocks delivery while obligations remain unresolved. The `N/A` audit rejects sections marked inapplicable when repository evidence exists. Together, these checks track a bounded set of content requirements.

A binding starts with a document claim and identifies its associated evidence. An obligation starts with a repository fact and identifies where the document must address it. For example, when the writer describes a public method, it binds that description to a recorded read; if the method is omitted, the corresponding obligation remains unresolved. The writer can keep the explanation concise while accounting for the interface it must cover.

4.4 Generation contract

We define the conditions under which a generated document is eligible for delivery. Let E^v be the repository evidence frozen at revision v, D_t the current draft, and R_t the receipts present at the start of step t. A receipt can be issued only by a system-mediated repository read and records the returned path, revision, and line interval. Let C_t^+ denote the checkable claims in D_t that are presented as supported, $B_t(c)$ the receipt set bound to claim c, and O^v the enumerable coverage obligations derived from E^v.

A binding may cite only receipts already present in the current system state:

$$B_t(c) \subseteq R_t. \tag{1}$$

The writer may revise a tentative sentence before establishing a binding, but it cannot deliver that sentence as supported until the relevant evidence has been read and bound. An invented receipt identifier therefore cannot satisfy the contract.

Let s_t contain the draft, receipts, bindings, obligations, and verification results, and let g_k be an applicable delivery gate. The delivery predicate is

$$\begin{aligned} \text{Eligible}_t \iff\ & \forall c \in C_t^+ : B_t(c) \neq \varnothing, \\ & \forall o \in O^v : \text{resolved}_t(o), \\ & \forall k : g_k(s_t) = \mathsf{pass}. \end{aligned} \tag{2}$$

Here, an obligation is resolved when the draft addresses it or the `N/A` audit accepts it as inapplicable. The contract governs delivery eligibility, not semantic entailment. A binding establishes an observable record of evidence use, while deterministic gates verify only the claim types covered by symbol tables, name resolution, or compilation. Other prose remains traceable to the evidence that was in view and can be audited separately.

4.5 Typed verification and delivery

The evidence used for verification depends on the claim type. Structural facts, including member existence, signatures, visibility, and inheritance, are checked against class-member information extracted with `javap` (Oracle n.d.). Usage examples must compile in the repository's build environment and resolve the documented component's symbols. Module references must resolve to repository classes or packages. Behavioral explanations retain their evidence bindings but require separate semantic assessment.

A method reference names a declared program element, whereas a usage example also specifies how program elements are combined. Name resolution and compilation thus check different aspects of the document. Selecting a check for each claim type allows a repair request to identify the affected content and the condition it must satisfy. In each case, drafting with evidence is followed by the applicable checks and repair of the affected content.

A failed check returns a repair request identifying the affected content. The document can be delivered only after the applicable binding, coverage, and structural conditions are satisfied. Delivery includes both the document and an evidence bundle for inspection. Appendix A describes the seven implementation gates and their state transitions.

5 Evaluation and Results

5.1 Evaluation design

Repositories. We evaluate 31 matched components from four Java repositories: MyBatis, jsoup, nflow, and enterprise-banking-system. Each component has one WriterKnowsBest and one RepoDoc document, yielding 62 documents and 3,808 extracted atomic facts. MyBatis and jsoup are mature, widely documented libraries, whereas nflow and enterprise-banking-system are less common. Including the latter pair reduces, but does not eliminate, the risk of parametric-memory contamination.

Baseline. We compare with RepoDoc (Xu et al. 2026), a knowledge-graph-based documentation pipeline that performs reliability checks after generation.

Models. deepseek-v4-flash writes the documents through native tool calls over the evidence space.

LLM-as-a-judge protocol. A single GLM-5.2 judge implements FActScore-style extraction and verification (Min et al. 2023). It splits prose, tables, and code examples into independently checkable facts, then compares each fact with the frozen repository source and records a label, rationale, and source locations. Labels are *correct* (clearly supported), *partial* (the core is supported but a qualifier or subclaim is inaccurate), *incorrect* (contradicted or asserting absent behavior), and *unverifiable* (neither supported nor contradicted). We report Atomic Support Rate (ASR) and Weighted Atomic Support (WAS):

$$\mathrm{ASR} = \frac{n_{\mathrm{correct}}}{N}, \qquad \mathrm{WAS} = \frac{n_{\mathrm{correct}} + 0.5\, n_{\mathrm{partial}}}{N},$$

where n_{correct} and n_{partial} count correct and partial facts, and N counts all evaluated facts. We also report method and public-API coverage: the fraction of source-declared public methods (and methods plus constructors and fields) mentioned by the document.

We ask four research questions: (RQ1) how factual support compares with the baseline; (RQ2) whether improved factual support is accompanied by higher API coverage; (RQ3) how obligation variants compare within the same generator; and (RQ4) how well cited evidence supports the bound text. The baseline comparison evaluates the complete systems. The variant comparison and binding audit examine narrower aspects of the approach.

5.2 Factual support (RQ1)

We compare atomic support for matched WriterKnowsBest and RepoDoc component documents. Table 1 reports the results by repository, and Table 2 lists the verdicts for all 3,808 facts.

WriterKnowsBest achieves higher weighted atomic support than RepoDoc on all four repositories. The largest absolute difference is on enterprise-banking-system: WAS increases from 0.586 to 0.793, and unverifiable facts decrease from 141 to 47. On MyBatis and jsoup, baseline WAS is approximately 0.93 and the gains are approximately three points. Across all repositories, incorrect facts decrease from 61 to 21 (65.6%), while correct facts increase from 1,624 to 1,639. This comparison covers the complete systems;

Table 1 Atomic support and coverage by repository. WAS = Weighted Atomic Support; ASR = Atomic Support Rate; MCov/API Cov = method/API coverage. Best values in bold.

	WriterKnowsBest		RepoDoc	
Repo	WAS	ASR	WAS	ASR
MyBatis	**0.962**	**0.940**	0.934	0.908
jsoup	**0.963**	**0.940**	0.930	0.905
nflow	**0.893**	**0.874**	0.837	0.799
bank	**0.793**	**0.780**	0.586	0.552
Overall	**0.924**	**0.904**	0.844	0.814
MCov (%)	**98.8**		85.3	
API Cov (%)	**99.0**		87.3	

Table 2 Atomic-fact verdict distribution over all 62 documents (3,808 facts).

System	Correct	Partial	Incorrect	Unverif.	Facts
RepoDoc	1,624	119	61	190	1,994
WriterKnowsBest	**1,639**	75	**21**	79	1,814

RQ3 examines the coverage, reference, and integrity obligation variants within the same generator.

The verdict counts help explain the higher support scores. Despite producing fewer facts in total, WriterKnowsBest produces more fully correct facts and fewer incorrect or unverifiable ones. Its improvement combines an increase in supported content with a reduction in statements that conflict with, or cannot be established from, the repository. Higher support is observed in both the mature libraries and the two less common projects.

The support advantage appears across all four repositories, with the largest difference in the banking project.

5.3 Coverage alongside factual support (RQ2)

Interface coverage also improves (Table 1). WriterKnowsBest documents 98.8% of public methods and 99.0% of the public API surface, compared with 85.3% and 87.3% for RepoDoc. The largest baseline coverage deficits are in the MyBatis and jsoup documents, where method coverage is as low as 82.5%.

WriterKnowsBest documents contain fewer atomic facts than the baseline (1,814 vs. 1,994) but cover more methods and API elements. Most of the difference in fact count comes from unverifiable content, including prospective behavior and generic advice unsupported by source: WriterKnowsBest contains 79 such facts, compared with 190 for RepoDoc.

Atomic support measures how well the document's statements are grounded in source, whereas API coverage measures how much of the public interface it addresses. The increase in both metrics answers the question in Section 3.1: a document can make more selective claims while covering the interface more completely. Coverage obligations guide that selection toward content defined by the repository.

Within this evaluation, higher support is accompanied by higher interface coverage. Fewer total statements therefore need not mean less coverage of the repository.

5.4 Contribution of generation obligations (RQ3)

The WriterKnowsBest–RepoDoc comparison measures end-to-end performance. We therefore compare two variants of the same generator on the same 31 components, differing in whether the coverage, reference, and integrity obligations are enabled. T0 retains the format, `N/A`, example-compilation, and module-reference gates; T1 is the full system. Both variants use deepseek-v4-flash with a 24-turn budget, and facts are assessed under the same four-way rubric as in RQ1.

Table 3 Matched comparison of generation-time obligation variants on 31 components. WAS and ASR are percentages; Δ values are percentage points and follow the reported three-decimal scores.

Arm	Facts	WAS	ASR	Inc.	Unver.
T0 (shared gates)	1,731	90.3	87.1	1.8	4.8
T1 (+ obligations)	1,814	**92.4**	**90.4**	**1.2**	**4.4**
Δ	+83	+2.1	+3.2	−0.6	−0.4

T1 increases WAS by 2.1 points and ASR by 3.2 points over T0 (Table 3). Its incorrect-fact rate is 1.2%, compared with 1.8% for T0, and its unverifiable rate is 4.4%, compared with 4.8%. T1 contains 83 more facts (+4.8%) and 131 more correct facts, while the counts of partial, incorrect, and unverifiable facts are each lower.

Relative to RepoDoc, the full system produces fewer facts and covers more of the interface. Relative to T0, it produces more facts with higher atomic support. Both patterns are consistent with selecting supported content, rather than with favoring shorter or longer documents. The combined obligations require the writer to address repository facts and record the evidence associated with its claims.

Compared with T0, T1 produces more facts and a higher proportion of correct facts. The comparison evaluates the combined coverage, reference, and integrity obligations.

5.5 Evidentiary sufficiency of bindings (RQ4)

We audit 223 bindings sampled from generation traces: at most five structured provenance records and five free-text reference notes per component, selected at evenly spaced positions. Under a separate prompt, the evaluator receives only each sampled claim or note and its cited receipt contents, then labels the binding *fully supported*,

partially supported, or *irrelevant* when the evidence supports all, some, or none of the associated text.

Table 4 Audit of 223 sampled evidence bindings (percent of each binding type).

Binding type	Full	Partial	Irrelevant	n
Structured provenance	18.3	80.2	**1.5**	131
Free-text ref. notes	19.6	55.4	25.0	92
All bindings	18.8	70.0	11.2	223

Of the 223 sampled bindings, 18.8% are fully supported, 70.0% are partially supported, and 11.2% are irrelevant to the associated text (Table 4). Structured provenance records have a 1.5% irrelevant rate, compared with 25.0% for free-text reference notes. These results indicate that most sampled bindings have some evidentiary relevance while motivating stronger entailment checks for passages that synthesize information from multiple sources.

Among sampled bindings, the irrelevant rate is 1.5% for structured provenance records and 25.0% for free-text reference notes; overall, 88.8% have at least partial evidentiary support.

5.6 Case study: Where errors remain

The matched `NflowEngine` documents illustrate how factual errors affect a component explanation. The baseline contains 68 facts and has weighted atomic support of 0.787. Its overview and structural descriptions are mostly correct, but its usage examples and cross-module explanations contain most of the errors. Four usage examples contain 18 compilation errors after repeated reports at the same code position are aggregated.

The errors include nonexistent types, invalid calls, fabricated APIs, and unresolved imports. For example, the baseline instantiates `WorkflowState` as a class, although it is an interface. The prose around the example may be plausible, but the example fails a compiler-backed check. Appendix B gives the full error account.

The WriterKnowsBest document contains 47 fully correct facts and one partially correct fact, with no incorrect facts and weighted atomic support of 0.99. This case shows the value of checking a claim against an appropriate verification source.

A reader needs both an account of what a component does and examples of how to use it. The overview explains its broad purpose; an example specifies the types and calls used to realize that purpose. An example that instantiates an interface as a class fails to connect the overview to usable code, even when the overview is accurate. Source support therefore needs to extend to the concrete statements developers act on.

6 Discussion

6.1 Implications for evidence-grounded generation

The results suggest assessing factual support and coverage separately. In our sample, the system with fewer extracted facts covers more of the public API, which document length alone would not reveal. Coverage obligations specify what the explanation must address, and claim attribution records the evidence associated with its statements. Using both controls makes completeness and faithfulness explicit requirements throughout generation.

The binding audit also distinguishes the two forms of attribution. Structured provenance records have a 1.5% irrelevant rate, compared with 25.0% for free-text reference notes. Explicit claim–evidence associations let reviewers inspect the cited basis of each statement and assess whether the two correspond. Table 4 lists the support categories used in this assessment.

The approach connects generation to verification through an explicit evidence record. The writer produces an explanation and records its associations with consulted evidence. Verification then inspects that record and applies claim-specific checks where available. Reviewers can examine the document and its recorded evidence use without access to the model's internal reasoning.

Repository exploration determines which evidence the writer can consult. The generation contract records how claims are associated with that evidence and tracks the content still required. Stronger retrieval can supply more evidence, and bindings preserve its association with the explanation. We integrate claim–evidence links and coverage checks into the writing process.

6.2 Practical implications

In practice, checks should reflect the statement being made. A signature can be compared with a symbol table, an example can be compiled, and a module name can be resolved. Design intent or runtime behavior requires different evidence. Distinguishing these cases helps reviewers understand what a successful check establishes.

With the evidence bundle, reviewers can follow a document claim to its associated source locations and inspect the proposed basis for the explanation. Typed repair targets the affected content, while persistent state allows interrupted runs to resume.

6.3 Limitations

Limitations of deterministic coverage. Typed verification decides only enumerable structural facts: member existence, signatures, visibility, inheritance, compilable examples, and resolvable references. Behavioral semantics, performance characteristics, and design intent are not decidable from symbol tables or a compiler. For such claims, the writer cites evidence, marks the claim as unsupported, or abstains; the post-hoc evaluation applies the source-grounded atomic-fact protocol above. Thus, deterministic guarantees apply to enumerable structural properties, not to behavioral meaning.

Limitations of language and toolchain generality. Our deterministic verifiers build on the JVM toolchain (`javap` symbol tables and `javac` compilation). Porting

the approach to another language requires an equally authoritative symbol-table and compiler front-end. For languages with less mature tooling, fewer claim types may be verified deterministically and more may require source-grounded assessment.

Limitations of binding sufficiency. A binding attests that evidence was read and cited, not that the evidence entails the bound claim. In our audit, 80.2% of structured bindings are only partially supported, and 25.0% of free-text reference notes are irrelevant to the associated text. Recording evidence consumption is insufficient to establish entailment by itself, and the audit motivates stronger checks over bound evidence.

6.4 Future work

WriterKnowsBest documents a frozen repository snapshot. Keeping documentation in step with code evolution—the incremental setting that RepoDoc targets—is beyond the present scope. Future work will examine incremental regeneration in which a code change invalidates only the affected receipts, bindings, verification results, and documents.

7 Conclusion

WriterKnowsBest requires explicit evidence attribution and coverage during repository documentation generation. It records source reads, binds checkable claims to those records, and checks enumerable obligations before delivery. Across four repositories, weighted atomic support improves from 0.844 to 0.924 while public-API coverage improves from 87.3% to 99.0%. The variant comparison shows higher factual support with the full requirements, while the binding audit finds full or partial evidence support for 88.8% of sampled bindings. These results support recording evidence use during generation and checking enumerable claims before delivering repository documentation.

Appendix A Implementation and execution details

This appendix describes how the prototype enforces the generation contract in Section 4.4. These execution details support inspection and reproduction; the empirical claims concern the document-level results in Section 5.

A.1 Execution semantics

The generation contract specifies the conditions for delivery. The execution layer controls which operations may update the state used to check delivery conditions. It enforces the same contract at the system-mediated tool boundary, rather than defining a separate reliability objective.

We model execution as

$$\mathcal{M} = (P, X, A, \Gamma, T), \tag{A1}$$

where P is a finite set of execution phases, X is the persistent data-state space, and A contains Read, Edit, Compile, BindEvidence, and Finish. The tool policy

$\Gamma : P \times X \to 2^A$ returns the actions available in the current phase and state, and $T : (P \times X) \times A \to P \times X$ applies an accepted action. At step t, the data state is

$$x_t = (R_t, B_t, O^v, D_t, K_t), \tag{A2}$$

where K_t contains compilation records and gate verdicts. Every accepted action must satisfy

$$a_t \in \Gamma(p_t, x_t), \qquad (p_{t+1}, x_{t+1}) = T(p_t, x_t, a_t). \tag{A3}$$

The writer proposes tool calls, but the execution layer rejects calls outside $\Gamma(p_t, x_t)$ without applying a state transition.

A failed gate narrows the actions available in the next phase. `READ_REQUIRED` enables additional repository reads and evidence binding. `EDIT_REQUIRED` restricts editing to the failing block and accepts a repair only when its content hash changes. `COMPILE_READY` enables recompilation of a repaired example. When an input changes, the system invalidates the affected verification results and retains unaffected results for reuse. Receipts, bindings, obligations, the draft, and verification records persist across transitions, allowing an interrupted run to continue from the recorded state without regenerating unchanged content.

Delivery is available only when the contract is satisfied:

$$\texttt{FINISH} \in \Gamma(p_t, x_t) \Longrightarrow \text{Eligible}_t\,. \tag{A4}$$

Within this mediated interface, the writer cannot bypass the contract by requesting an out-of-phase action. These semantics do not establish semantic truth or general agent safety. They ensure that receipts originate from system reads, bindings refer to receipts already in state, and delivery remains unavailable while the contract is unsatisfied. Compilation is one verifier within this execution layer, rather than the organizing principle of the method.

A.2 Gate implementation

The system routes claims by type. Compiler or repository artifacts verify claims that admit deterministic checks; behavioral prose must cite receipts or be marked unsupported:

- **Structural facts** (member existence, signatures, visibility, inheritance) are checked against class-member information extracted with `javap` (Oracle n.d.), with bridge and synthetic members filtered;
- **Usage examples** must compile under `javac` in the repository's own build environment and must resolve the documented component's symbols;
- **Module references** are resolved against repository classes and packages through a fully qualified class name (FQCN) gate;
- **Behavioral prose** must cite associated receipts; when adequate evidence is unavailable, the writer abstains or marks the statement as unsupported.

Symbol-table, compilation, and name-resolution gates return deterministic pass/fail results. The binding gate checks the presence of claim–receipt associations, not whether the cited evidence entails the claim.

Seven gates control delivery: format, `N/A` audit, coverage, module reference, reference completeness, example compilation, and integrity. The system records each verdict in the execution state and either activates the corresponding typed repair phase (§A.1; Figure 3) or makes `FINISH` available. Once all gates pass, `FINISH` returns the document, coverage manifest, provenance records, reference notes, compile records, and generation trace.

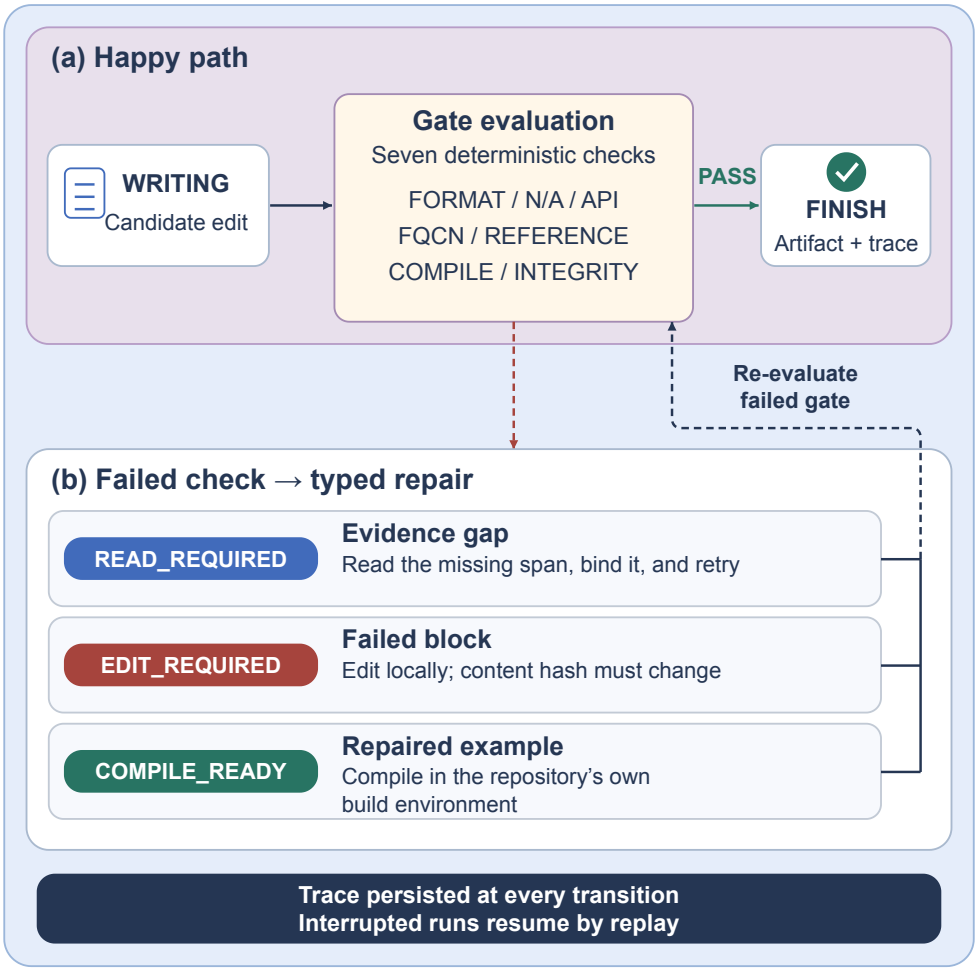

Fig. 3 Typed diagnostics make every failure actionable. (a) `FINISH` becomes available only after all seven gates pass. (b) A failed check selects a typed repair phase, followed by re-evaluation of the failed gate

A.3 Operational observations

Generation traces show how the system repairs failed checks and resumes interrupted runs. A compilation failure returns `EDIT_REQUIRED` for the affected block. The writer edits that block, re-reads it, and resubmits it; the repair is accepted only if the code hash changes. Examples for `NodeIterator`, `Parser`, `Safelist`, `Element`, and `XMLConfigBuilder` reach compiled-and-verified states through this process.

One operational trace records a run interrupted by API rate limiting at round 34, after it has discharged five of six coverage obligations and compiled four examples. The run resumes from persisted state and finishes 11 rounds later without regenerating unchanged text or re-reading evidence.

Appendix B Detailed error analysis

B.1 Error distribution in `NflowEngine`

In the baseline document, the section-level correct-fact counts are Overview 6/6, Key Responsibilities 5/5, Constructor 9/10, Public Methods 11/12, Attributes 8/8, Usage Examples 0/6, Related Modules 2/6, and Common Issues 0/3. These counts supplement the document-level comparison in Section 5.6.

Across the four baseline usage examples, 18 compilation errors remain after aggregating repeated reports at the same code position. These errors fall into four categories. **(1) Invalid types:** the nonexistent `H2Variants` is instantiated in all four examples, and the interface `WorkflowState` is instantiated as a class. **(2) Invalid calls:** `execute` is absent from `WorkflowDefinition`, while `stopInState()` is called with no arguments. **(3) Fabricated APIs:** `WorkflowInstance.builder()` and `MaintenanceConfiguration.builder()` do not exist; the examples also substitute `createWorkflowInstance` for `insertWorkflowInstance`, `checkHealth` for `checkDatabaseConnection`, and `maintainWorkflows` for `cleanupWorkflows`.
(4) Missing imports: unresolved imports make the examples uncompilable outright.

Statements and Declarations

Funding

No funding was received for conducting this study.

Competing interests

The authors declare no competing interests.

Author contributions

Yaokai Zu conducted the main experiments. Tongtong Wang and Danfeng Zhang provided experimental guidance. Jiayu Zhang provided guidance throughout the experimental work.

Data and code availability

The enterprise data and code used in this study are not publicly available due to confidentiality restrictions.

References

Ahmed T, Devanbu P (2022) Few-shot training llms for project-specific code-summarization. In: Proceedings of the 37th IEEE/ACM international conference on automated software engineering, pp 1–5, https://doi.org/10.1145/3551349.3559555

Bae S, Lee J, Choi C, et al (2026a) Remember your trace: Memory-guided long-horizon agentic framework for consistent and hierarchical repository-level code documentation. arXiv preprint arXiv:2605.14563 https://doi.org/10.48550/arXiv.2605.14563

Bae S, Na C, Lee J, et al (2026b) Referee: Reference-free and fine-grained method for evaluating factual consistency in real-world code summarization. In: Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp 14352–14377, https://doi.org/10.18653/v1/2026.acl-long.653

Bishop JA, Ananiadou S, Xie Q (2024) Longdocfactscore: Evaluating the factuality of long document abstractive summarisation. In: Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024), pp 10777–10789

Chen GH, Chen S, Liu Z, et al (2024) Humans or llms as the judge? a study on judgement bias. In: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pp 8301–8327, https://doi.org/10.18653/v1/2024.emnlp-main.474

Crupi G, Tufano R, Velasco A, et al (2025) On the effectiveness of llm-as-a-judge for code generation and summarization. IEEE Transactions on Software Engineering 51(8):2329–2345. https://doi.org/10.1109/tse.2025.3586082

Degiovanni R, Morales S, Coccia M, et al (2026) Robust llm-as-a-judge validators for assessing the quality of educational exams. In: Proceedings of the 41st ACM/SIGAPP Symposium on Applied Computing, pp 95–102, https://doi.org/10.1145/3748522.3779926

Feng Y, Wang S, Cheng Z, et al (2025) Are we on the right way to assessing llm-as-a-judge? arXiv preprint arXiv:2512.16041 https://doi.org/10.48550/arXiv.2512.16041

Gao T, Yen H, Yu J, et al (2023) Enabling large language models to generate text with citations. In: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pp 6465–6488, https://doi.org/10.18653/v1/2023.emnlp-main.398

Hoang AN, Le-Anh M, Le B, et al (2026) Codewiki: Evaluating ai's ability to generate holistic documentation for large-scale codebases. In: Findings of the Association for Computational Linguistics: ACL 2026, pp 5812–5827, https://doi.org/10.18653/v1/2026.findings-acl.288

Iqbal H, Wang Y, Wang M, et al (2024) Openfactcheck: A unified framework for factuality evaluation of llms. In: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp 219–229, https://doi.org/10.18653/v1/2024.emnlp-demo.23

Khan JY, Uddin G (2023) Automatic code documentation generation using gpt-3. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. Association for Computing Machinery, New York, NY, USA, ASE '22, https://doi.org/10.1145/3551349.3559548, URL https://doi.org/10.1145/3551349.3559548

Lewis P, Perez E, Piktus A, et al (2020) Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in neural information processing systems 33:9459–9474

Liu Y, Iter D, Xu Y, et al (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, pp 2511–2522, https://doi.org/10.18653/v1/2023.emnlp-main.153

Lomshakov V, Podivilov A, Savin S, et al (2024) Proconsul: Project context for code summarization with llms. In: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track, pp 866–880, https://doi.org/10.18653/v1/2024.emnlp-industry.65

Luo Q, Ye Y, Liang S, et al (2024) Repoagent: An llm-powered open-source framework for repository-level code documentation generation. In: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp 436–464, https://doi.org/10.18653/v1/2024.emnlp-demo.46

Maharaj K, Munigala V, Tamilselvam SG, et al (2025) Etf: An entity tracing framework for hallucination detection in code summaries. In: Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp 30639–30652, https://doi.org/10.18653/v1/2025.acl-long.1480

Makharev V, Ivanov V (2025) Code summarization beyond function level. In: 2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code), IEEE, pp 153–160, https://doi.org/10.1109/llm4code66737.2025.00024

Mastropaolo A, Ciniselli M, Di Penta M, et al (2024) Evaluating code summarization techniques: A new metric and an empirical characterization. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pp 1–13, https://doi.org/10.1145/3597503.3639174

Min S, Krishna K, Lyu X, et al (2023) Factscore: Fine-grained atomic evaluation of factual precision in long form text generation. In: Proceedings of the 2023 conference on empirical methods in natural language processing, pp 12076–12100, https://doi.org/10.18653/v1/2023.emnlp-main.741

Oracle (n.d.) The javap command. URL https://docs.oracle.com/en/java/javase/17/docs/specs/man/javap.html, java SE 17 and JDK 17. Accessed 12 October 2026

Panchal M, Deo A, Prabhu V, et al (2026) Ledge: Leveraging dependency graphs for enhanced context aware documentation generation. Automated Software Engineering 33(2):51. https://doi.org/10.1007/s10515-026-00596-y

Rajendhran R, Zadeh A, Sarte M, et al (2025) VeriFastScore: Speeding up long-form factuality evaluation. In: Christodoulopoulos C, Chakraborty T, Rose C, et al (eds) Findings of the Association for Computational Linguistics: EMNLP 2025. Association for Computational Linguistics, Suzhou, China, pp 9234–9259, https://doi.org/10.18653/v1/2025.findings-emnlp.491, URL https://aclanthology.org/2025.findings-emnlp.491/

Scirè A, Ghonim K, Navigli R (2024) Fenice: Factuality evaluation of summarization based on natural language inference and claim extraction. In: Findings of the Association for Computational Linguistics: ACL 2024, pp 14148–14161, https://doi.org/10.18653/v1/2024.findings-acl.841

Tian Y, Yan W, Yang Q, et al (2025) Codehalu: Investigating code hallucinations in llms via execution-based verification. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp 25300–25308, https://doi.org/10.1609/aaai.v39i24.34717

Venigalla ASM, Chimalakonda S (2025) Docfetch-towards generating software documentation from multiple software artifacts. arXiv preprint arXiv:2508.17719 https://doi.org/10.48550/arXiv.2508.17719

Wang R, Guo J, Gao C, et al (2025) Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering. Proceedings of the ACM on Software Engineering 2(ISSTA):1955–1977. https://doi.org/10.1145/3728963

Wang X, Hu R, Gao C, et al (2026) Evaluating repository-level software documentation via question answering and feature-driven development. arXiv preprint arXiv:2604.06793 https://doi.org/10.48550/arXiv.2604.06793

Wang Y, Reddy RG, Mujahid ZM, et al (2024) Factcheck-bench: Fine-grained evaluation benchmark for automatic fact-checkers. In: Findings of the Association for Computational Linguistics: EMNLP 2024, pp 14199–14230, https://doi.org/10.18653/v1/2024.findings-emnlp.830

Wei J, Yang C, Song X, et al (2024) Long-form factuality in large language models. Advances in Neural Information Processing Systems 37:80756–80827. https://doi.org/10.52202/079017-2567

Xu D, Liu M, Wang X, et al (2026) Repodoc: A knowledge graph-based framework to automatic documentation generation and incremental updates. arXiv preprint arXiv:2604.26523 https://doi.org/10.48550/arXiv.2604.26523

Yang D, Simoulin A, Qian X, et al (2025) Docagent: A multi-agent system for automated code documentation generation. In: Proceedings of the 63rd Annual Meeting of the

Association for Computational Linguistics (Volume 3: System Demonstrations), pp 460–471, https://doi.org/10.18653/v1/2025.acl-demo.44

Zheng L, Chiang WL, Sheng Y, et al (2023) Judging llm-as-a-judge with mt-bench and chatbot arena. Advances in neural information processing systems 36:46595–46623. https://doi.org/10.52202/075280-2020

Zhuge M, Zhao C, Ashley DR, et al (2025) Agent-as-a-judge: Evaluate agents with agents. In: Singh A, Fazel M, Hsu D, et al (eds) Proceedings of the 42nd International Conference on Machine Learning, Proceedings of Machine Learning Research, vol 267. PMLR, pp 80569–80611, URL https://proceedings.mlr.press/v267/zhuge25a.html