Do LLMs Understand Backend Data-Layer Semantics? A Controlled Study of Cross-Endpoint Issues

Francesco Dente[1*], Dario Satriani[2*], Donatello Santoro[2], Enzo Veltri[2], Paolo Papotti[1]
[1]EURECOM, Biot, France
[2]University of Basilicata, Potenza, Italy
{francesco.dente,paolo.papotti}@eurecom.fr
{dario.satriani,donatello.santoro,enzo.veltri}@unibas.it

Abstract

Evaluating Large Language Models (LLMs) on code review requires moving beyond local syntax to assess system-wide semantic reasoning. We investigate whether LLMs can detect semantic data-layer issues: latent bugs requiring cross-endpoint reasoning over persistent state. We introduce a controlled benchmark of 216 minimal database-backed tasks, derived from 18 independent hand-written scenarios, to evaluate five frontier and open-weights models under two settings: open-ended review versus targeted detection. Our evaluation reveals a stark *discovery gap*: while models reliably recognize explicitly described issues, they struggle to discover them unprompted, with average F_1 dropping from 0.91 to 0.85 for the easiest category and from 0.73 to 0.03 for the hardest. Furthermore, models exhibit a hierarchy of detectability across issue types, and for retention issues, a *compliance bias* where explicit definitions lead to over-flagging of structurally similar but semantically correct code. Our findings demonstrate that targeted defect detection substantially overestimates true open-ended review capabilities, underscoring the need to evaluate LLMs on latent intent recovery. Code

1 Introduction

Large Language Models (LLMs) are used to generate and review code. A code reviewer must not only recognize local syntax patterns but also determine whether an implementation satisfies the intended behavior of the system from a mixture of source code, endpoint names, and natural language (NL) comments (Tambon et al., 2025; Nguyen et al., 2025; Jin and Chen, 2025). This paper studies a form of this challenge: whether LLMs detect *semantic data-layer issues* in database-backed applications. Semantic bugs are elusive because the relevant oracle is often not a local syntactic property, but an implicit program invariant (Shi et al., 2010; Dossche and Coppens, 2024).

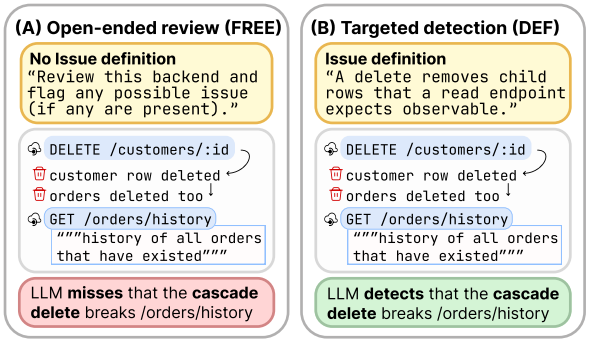


Figure 1: **FREE vs. DEF review on a cascading-delete violation.** Both panels show the same backend: a delete endpoint that cascades to order rows, and a read endpoint contracted to return all historical orders. **(A)** Without an issue definition (FREE), the model misses the violation. **(B)** Given the definition (DEF), it identifies it.

We focus on the data layer since it is where application-level semantics become persistent, shared, and observable across multiple endpoints. However, intended data semantics are often split across database schemas, endpoint logic, ORM mappings, queries, and NL cues such as docstrings. Data-layer issues may not crash the program, but emerge only from interactions across these components, affecting reliability, maintainability, and usability (Huang et al., 2023; Wang et al., 2015; Dente et al., 2026).

We use this setting as a diagnostic probe for LLM code review. Data-layer errors are small enough to be controlled, but irreducible to a single syntactic trigger. For example, a cascading delete is not inherently wrong, it becomes problematic only if a second endpoint's documented contract commits to preserving those deleted rows, as shown in Figure 1. Detecting such problems requires connecting schema constraints, endpoint behavior, and latent application intent, a setting where rule-based and code-centric approaches can miss such issues

*These authors contributed equally.

(Yang and Chen, 2026).

We study three representative dimensions of backend data semantics: *retention* (`cascade_delete`), *consistency* (`stale_aggregate`), and *visibility* (`exposed_record`). We construct a manually validated benchmark of 216 minimal tasks, obtained by rewriting 18 independent hand-written scenarios into 12 language and framework variants each, where all relevant schema and code evidence is provided in a single prompt, isolating semantic reasoning from retrieval and repository navigation.

Our evaluation compares an unguided, open-ended code review setting (FREE) against targeted detection where the model receives an explicit natural language definition of the target issue (DEF). This separation lets us distinguish *recognizing a named problem* from *discovering an unnamed semantic issue*. Testing five frontier and open-weight models reveals three core findings:

- Discovery gap: Removing the issue definition causes a substantial drop in average F_1: visibility 0.91→0.85, consistency 0.94→0.56, and retention 0.73→0.03, even though all tasks are minimal and validated with human reviewers.
- Hierarchy of detectability: Visibility issues are frequently discovered unprompted, consistency issues moderately so, and retention issues almost never. This suggests models recognize familiar security-like practices more readily than latent data-integrity invariants.
- Compliance bias: Given explicit definitions, models tend to flag structurally similar cases regardless of whether the semantic invariant is actually violated, achieving perfect recall but low precision on retention issues.

In summary, we contribute (1) a taxonomy of latent semantic data-layer issues requiring cross-endpoint reasoning; (2) a controlled benchmark of 216 instances that isolates semantic reasoning from retrieval; and (3) empirical evidence that targeted detection evaluations substantially overestimate open-ended review ability, highlighting the need to evaluate LLMs on latent intent recovery.

2 Related Work

LLMs for code review. Recent work studies LLMs' ability to review and assess code (Zhang et al., 2026; Khan et al., 2026; Wu et al., 2024; Nguyen et al., 2025; Tang et al., 2024; Dinu et al., 2026; Hu et al., 2025). These works cover recurring bug patterns, code smells, missing corner cases, and incomplete implementations in both human- and LLM-generated code (Tambon et al., 2025; Sadik and Govind, 2025). However, these approaches target localized code inspection rather than latent cross-code and data semantics (Lu et al., 2021, 2023; Lin et al., 2024; Zhao et al., 2023), while we isolate semantic review problems in database-backed APIs.

Code and natural language specifications. Prior work studies whether LLMs can verify code against NL specifications, showing systematic failures when judging whether code satisfies a given requirement (Jin and Chen, 2025; Hu et al., 2025). Instead of assuming that the relevant specification is explicitly given, we ask whether an LLM reviewer can discover semantic data-layer issues when the intent is latent in code, schema, and endpoint names.

Semantic bugs and implicit specifications. Semantic bug detection has been studied through invariants and implicit specifications (Shi et al., 2010; Dossche and Coppens, 2024). We adopt this view, but focus on LLM-based review retention, consistency, and visibility of persistent records across API endpoints. Other work extracts semantic constraints from requirements to guide LLM-based unit-test generation for business-logic bugs (Liu et al., 2025; Yang and Chen, 2026); we instead study lightweight source-code review, without relying on test generation to expose bugs.

3 Methodology

Issue Taxonomy. We define three data-layer issues targeting persistent-state semantics: *retention*, *consistency*, and *visibility* (Table 1). None of these patterns is inherently a bug: each becomes problematic only when cross-endpoint reasoning reveals that a second endpoint observes the side effect.

The categories instantiate three core dimensions along which persistent state can violate intent: whether data survives (retention), stays coherent (consistency), and is visible to the right readers (visibility). We select these because they are highly representative of real-world backend bugs. First, mapping the fine-grained root causes reported in an empirical bug study (Liu et al., 2024) onto our dimensions, more than half of the 423 reported bugs fall into these three categories (e.g., transaction misuse under retention; inconsistent entity state under consistency). Second, each category corresponds to a failure mode actively documented and mitigated by major engineering organizations: silent

Table 1: Issue taxonomy. Every issue requires reasoning across at least two endpoints and the database schema.

Issue	Definition	Example scenario
`exposed_record` (visibility)	A write changes a record's status but a read endpoint still serves it without checking.	A product manager discontinues a product, but the storefront endpoint lacks a status filter; customers can still browse and attempt to purchase it.
`stale_aggregate` (consistency)	A write modifies source-of-truth rows but fails to update a denormalized aggregate that another endpoint returns.	A warehouse operator updates a line item's price, but the precomputed order total is never recalculated; a customer viewing their order sees an outdated amount.
`cascade_delete` (retention)	A delete removes child rows that a read endpoint expects to remain observable (e.g., historical records).	An admin deletes a user account and the system silently removes all associated orders; a reporting dashboard now returns incomplete historical data.

foreign-key cascades (Thirumalai, 2026), cache and aggregate consistency (Pan, 2022), and soft-delete visibility filtering (Jerry Nixon, 2026). The three categories are intended as representative of recurring data-layer failure modes, not as an exhaustive taxonomy. Each defect admits a small, local repair, as reported in Appendix L.

Task Construction. For each issue, we manually construct three positive and three negative tasks, yielding 18 base tasks. Each task consists of an SQLite schema paired with a minimal Flask backend (avg. 53 LoC) exposing 2–3 endpoints over raw SQL, scoped so that only the target issue is reachable. Negative tasks share the positive surface structure but include a mechanism that precludes the issue, e.g., a `JOIN` that filters orphaned rows (Appendix A).

We rewrite each base task into 11 additional language/framework/ORM combinations with GPT-5.4 (Appendix C.1), yielding $18 \times 12 = 216$ instances across three languages (Python, JavaScript, Go), two frameworks per language (Flask/FastAPI, Express/Fastify, Gin/Fiber), and two data-access styles (raw SQL and ORM). Rewrites preserve endpoint semantics, URL paths, docstrings, query structure, and are manually verified.

We compare FREE, an open-ended data-layer review without an issue definition, against DEF, where the model receives a NL definition of the target issue. This distinguishes recognizing a described problem from independently discovering a semantic issue (prompts in Appendix C.2).

Evaluation. Each model is queried in a single-shot setting with the prompt containing the database schema, application code, and task instructions. DEF outputs are structured JSON labels compared deterministically against the ground truth; fewer than 0.3% of DEF outputs failed JSON parsing. FREE produces free-form text; one of the authors manually classified all FREE outputs against the ground truth. A second author independently re-annotated all the instances, yielding near-perfect agreement (99.7%, Cohen's $\kappa = 0.99$). A post-hoc LLM judge (GPT-5.4) agrees with both annotators on $\geq$92% of instances ($\kappa \geq 0.78$), suggesting that this step could also be automated for larger-scale evaluations. We report precision, recall, and F_1 computed against the manual annotations for FREE. Full evaluation details are provided in Appendix B.

Human Validation. To validate that the tasks are solvable from code and schema alone, 10 backend developers (3+ years of experience) annotated 39 distinct positive instances under the same FREE conditions given to models. Each issue category was independently identified by at least three experts, confirming that the tasks are well-formulated and the issues are discoverable by humans. The overall detection rate remained 16/39 (issues detected), showing that the tasks are non-trivial even for experienced developers (Appendix D).

4 Results

Models perform much worse in open-ended review than in targeted detection, showing that issue discovery is a major bottleneck even when all relevant code and schema evidence is present (Table 2). Based on the performance disparities between executions and the semantic complexity of issues' dimensions, we identified three core findings.

Discovery Gap. Our empirical evaluation reveals a performance disparity between open-ended semantic issue discovery and targeted detection, a phenomenon that we formalize as the *discovery gap*. When models evaluate implementations in the unguided FREE setting (no semantic issue definitions are provided), they struggle to surface latent semantic anomalies. For instance, on the `cascade_delete` category, all models outside the

Table 2: Detection performance by model, prompt type, and issue category. P = Precision, R = Recall. Colored deltas on DEF rows show $F1_{DEF} - F1_{FREE}$.

		EXPOSED RECORD (visibility)			STALE AGGREGATE (consistency)			CASCADE DELETE (retention)		
Model	**Prompt**	P	R	F1	P	R	F1	P	R	F1
GPT-5.4	FREE	1.00	0.81	0.89	0.90	0.78	0.84	0.75	0.08	0.15
	DEF	0.82	1.00	0.90 (+.01)	1.00	1.00	1.00 (+.16)	0.61	1.00	0.76 (+.61)
Kimi-K2.6	FREE	1.00	0.89	0.94	1.00	0.44	0.62	0.00	0.00	0.00
	DEF	0.84	1.00	0.91 (–.03)	0.95	1.00	0.97 (+.35)	0.57	1.00	0.73 (+.73)
DeepSeek-V3.1	FREE	1.00	0.58	0.74	1.00	0.14	0.24	0.00	0.00	0.00
	DEF	0.61	1.00	0.76 (+.02)	0.72	1.00	0.84 (+.60)	0.55	1.00	0.71 (+.71)
Gemma-4-31b	FREE	1.00	0.94	0.97	1.00	0.67	0.80	0.00	0.00	0.00
	DEF	1.00	1.00	1.00 (+.03)	1.00	1.00	1.00 (+.20)	0.60	1.00	0.75 (+.75)
Qwen3.5-397B-A17B	FREE	1.00	0.53	0.69	1.00	0.19	0.33	0.00	0.00	0.00
	DEF	1.00	1.00	1.00 (+.31)	0.82	1.00	0.90 (+.57)	0.55	1.00	0.71 (+.71)
Average	FREE	1.00	0.75	0.85	0.98	0.44	0.56	0.15	0.02	0.03
	DEF	0.85	1.00	0.91 (+.06)	0.90	1.00	0.94 (+.38)	0.57	1.00	0.73 (+.70)

GPT family fail to discover the underlying issue entirely ($F_1 = 0.00$), while GPT-5.4 attains only $F_1 = 0.15$. By contrast, models identify the same issue with average $F_1 = 0.73$ in DEF. The results are supported by a 95% bootstrap confidence interval analysis (details in Appendix K) in which the paired DEF−FREE gain is $+0.70$ $[0.62, 0.86]$ for retention and $+0.38$ $[0.18, 0.60]$ for consistency, whereas for visibility the interval includes zero ($+0.06$ $[-0.00, 0.17]$): the discovery gap is significant exactly where models fail to discover the issue unprompted. The gap is not a matter of the target issue being crowded out by other findings: FREE reviews report 2.02 findings on average, and 40% of GPT-5.4's reviews on positive instances are empty (Appendix H). This suggests that models are substantially better at applying an explicit semantic rule than at recovering the relevant invariant from contextual cues.

Hierarchy of Detectability. The results outline a distinct hierarchy of the detectability rate across the three semantic categories, from (i) visibility (`exposed_record`), highest rate, to (ii) consistency (`stale_aggregate`), moderate rate, and finally to (iii) retention (`cascade_delete`), lowest rate. **Visibility** (`exposed_record`) represents the highest baseline performance in unguided settings. In the FREE setting, SOTA models such as GPT-5.4, Kimi-K2.6, and Gemma-4-31b demonstrate exceptional intrinsic detection capabilities, achieving high F_1 (see Table 2).This likely reflects the models' alignment with common security-centric review patterns, where missing filters are a well-documented defect. **Consistency** (`stale_aggregate`) proves to be moderately elusive under open-ended conditions as all models, with the exception of GPT-5.4 and Gemma-4-31b, achieve a F_1 score below 0.7 in the FREE setting, but results improve in the DEF scenario. This highlights the models' ability to trace implicit state dependencies across different resources when explicitly stated. **Retention** (`cascade_delete`) marks the worst result in semantic issue discovery. In the FREE setting, Kimi-K2.6, DeepSeek-V3.1, Gemma-4-31b and Qwen3.5-397B suffer an outright failure, returning an F_1 score of 0.00, while GPT-5.4 yields a score of 0.15. However, providing a declarative definition in the DEF setting yields an average F_1 gain of $+0.70$, consistently distributed across all models. This indicates that despite being able to identify the issue, a cascading semantic side effect is structurally indistinguishable from normal database behavior to an unprompted model. Qualitative analysis of these results reveals that models rarely detect latent persistent-state semantic violations. Instead, their evaluations are heavily biased toward identifying issues that resemble common, established code patterns. Appendix J further shows that the hierarchy is not an artifact of LLM-based augmentation. On the 18 hand-written base tasks alone, it persists and is starker than in the full set under FREE (visibility 0.92 > consistency 0.68 > retention 0.00). A difference-in-differences analysis shows no consistent self-preference by GPT-5.4 on the instances it generated.

Compliance Bias. Explicit natural language

definitions enhances model metrics, but the `cascade_delete` scenario exposes a contrasting phenomenon: compliance bias. Given a declarative issue definition, models often rely on superficial structural heuristics rather than semantic reasoning, acting as simple rule-based static analyzers. This explains the recall of 1.00 scored by all the models: instead of checking whether data invariants are violated, they over-match the target definition, flagging cases that superficially mirrors the structural layout of the definition, such as the presence of database-level cascade delete. This suggests that models are effective at identifying specific issues, but sometimes lack the capacity to distinguish correct and incorrect implementation when the latter is structurally similar. For instance, as detailed in Appendix E, models routinely flag a benign customer-deletion endpoint as a violation because it executes a safe `CASCADE DELETE`. For example, GPT-5.4 flags a semantically sound endpoint with: *"the explicit delete* `DELETE FROM orders WHERE o_custkey = ?` *matches an application-level cascade delete, and it causes observable data loss because the* `/api/orders` *endpoint depends on rows in* `orders` *existing."* The model acts as a rigid static analyzer, failing to infer that the read endpoint's active-customer filter renders the deletion both safe and intended. This is not reducible to generic instruction-following: adding a statement to the prompt that explicitly licenses a negative answer leaves the over-flagging almost intact (details in Appendix E.1).

These patterns are stable across languages, data-access styles (Appendix F), and sampling temperature ($\sigma \leq 0.04$; Appendix G), and no single base scenario drives them (Appendix I), confirming that the dominant factor is the issue category, not the implementation idiom or sampling temperature.

5 Conclusion

We investigated the capacity of LLMs to identify semantic data-layer issues, a class of latent errors characterized not by local faults but by violations of system-wide invariants and persistent-state logic. By constructing a suite of controlled, self-contained backend programs, we assessed the capacity of models to recognize semantic issues. Even in this idealized setting, where all relevant evidence fits in a single prompt, open-ended discovery lags far behind targeted detection. Our findings suggest that advancing the state of the art in LLM-based code review requires moving toward models that can synthesize implicit requirements across disparate system components.

Limitations

Self-contained prompts. By design, each task is delivered as a single prompt containing the full schema and application code. This isolates semantic reasoning from retrieval and navigation, but real-world code review requires locating relevant files across a repository. Our results therefore characterize reasoning ability in an idealized setting and may overestimate performance in practice, where the model must additionally decide *what* to read.

Taxonomy scope. We study three issue types targeting retention, consistency, and visibility. These are intended to represent recurring failure modes in the data-layer of the backend. However, the space of semantic data-layer issues is broader; our findings may not transfer to categories that involve different reasoning patterns. Adding more semantically distinct scenarios per category is a natural extension of the benchmark, and our released task design easily allows it.

Single-shot, non-agentic evaluation. Models are evaluated in a single inference call with no tool use, iterative refinement, or multi-turn interaction. Agentic pipelines that can query the database, run tests, or request clarification may achieve higher performance.

Ethics Considerations

This work evaluates existing models on synthetic tasks and does not introduce new risks. A practical concern is that high DEF performance may encourage over-reliance on LLM reviewers for data-layer correctness; our discovery gap results caution against this. All tasks are synthetic and contain no proprietary or personal data.

Acknowledgment

This work was partially supported by the AI4SWEng project, funded by the EU's Horizon program (grant agreement No. 101189909), and by the 3IA Côte d'Azur Investments in the IA-cluster project managed by the ANR (ANR-23-IACL-0001).

References

Moonshot AI. 2026. Kimi k2.6: Advancing open-source coding.

Google Deepmind. 2026. Gemma 4.

DeepSeek-AI. 2024. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Francesco Dente, Dario Satriani, and Paolo Papotti. 2026. Constraint decay: The fragility of llm agents in backend code generation. *arXiv preprint arXiv:2605.06445*.

Ion George Dinu, Marian Cristian Mihăescu, and Traian Rebedea. 2026. Smellbench: Evaluating llm agents on architectural code smell repair. *Preprint*, arXiv:2605.07001.

Niels Dossche and Bart Coppens. 2024. Inference of error specifications and bug detection using structural similarities. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association.

Ruida Hu, Xinchen Wang, Xin-Cheng Wen, Zhao Zhang, Bo Jiang, Pengfei Gao, Chao Peng, and Cuiyun Gao. 2025. Benchmarking llms for fine-grained code review with enriched context in practice. *CoRR*, abs/2511.07017.

Haochen Huang, Bingyu Shen, Li Zhong, and Yuanyuan Zhou. 2023. Protecting data integrity of web applications with database constraints inferred from application code. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 632–645. ACM.

Jerry Nixon. 2026. Enable soft delete in Azure SQL. Azure SQL Dev Corner, Microsoft DevBlogs.

Haolin Jin and Huaming Chen. 2025. Uncovering systematic failures of llms in verifying code against natural language specifications. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 3819–3823. IEEE Press.

Taufiqul Islam Khan, Shaowei Wang, Haoxiang Zhang, and Tse-Hsun Chen. 2026. A survey of code review benchmarks and evaluation practices in pre-llm and LLM era. *CoRR*, abs/2602.13377.

Hong Yi Lin, Patanamon Thongtanunam, Christoph Treude, and Wachiraphan Charoenwet. 2024. Improving automated code reviews: Learning from experience. In *21st IEEE/ACM International Conference on Mining Software Repositories, MSR 2024, Lisbon, Portugal, April 15-16, 2024*, pages 278–283. ACM.

Kaibo Liu, Zhenpeng Chen, Yiyang Liu, Jie M. Zhang, Mark Harman, Yudong Han, Yun Ma, Yihong Dong, Ge Li, and Gang Huang. 2025. Llm-powered test case generation for detecting bugs in plausible programs. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 430–440. Association for Computational Linguistics.

Wei Liu, Shouvick Mondal, and Tse-Hsun Chen. 2024. An empirical study on the characteristics of database access bugs in java applications. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–25.

Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023*, pages 647–658. IEEE.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, and 3 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Dung Manh Nguyen, Thang Chau Phan, Nam Le Hai, Tien-Thong Doan, Nam V. Nguyen, Quang Pham, and Nghi D. Q. Bui. 2025. CodeMMLU: A multi-task benchmark for assessing code understanding & reasoning capabilities of codeLLMs. In *The Thirteenth International Conference on Learning Representations*.

OpenAI. 2026. Introducing GPT-5.4.

Lu Pan. 2022. Cache made consistent. Engineering at Meta.

Qwen Team. 2026. Qwen3.5: Towards native multimodal agents.

Ahmed R. Sadik and Siddhata Govind. 2025. Benchmarking llm for code smells detection: Openai gpt-4.0 vs deepseek-v3. In *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*, EASE '25, page 969–975, New York, NY, USA. Association for Computing Machinery.

Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. 2010. Do i use the wrong definition? defuse: definition-use invariants for detecting concurrency and sequential bugs. *SIGPLAN Not.*, 45(10):160–174.

Florian Tambon, Arghavan Moradi-Dakhel, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Giuliano Antoniol. 2025. Bugs in large language

models generated code: an empirical study. *Empirical Softw. Engg.*, 30(3).

Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé F. Bissyandé. 2024. CodeAgent: Autonomous communicative agents for code review. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11279–11313, Miami, Florida, USA. Association for Computational Linguistics.

Prabakaran Thirumalai. 2026. No more silent foreign key cascades: MySQL 9.7 lets child triggers speak up. The Oracle MySQL Blog.

Peipei Wang, Daniel Joseph Dean, and Xiaohui Gu. 2015. Understanding real world data corruptions in cloud systems. In *2015 IEEE International Conference on Cloud Engineering, IC2E 2015, Tempe, AZ, USA, March 9-13, 2015*, pages 116–125. IEEE Computer Society.

Di Wu, Fangwen Mu, Lin Shi, Zhaoqiang Guo, Kui Liu, Weiguang Zhuang, Yuqi Zhong, and Li Zhang. 2024. ismell: Assembling llms with expert toolsets for code smell detection and refactoring. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1345–1357, New York, NY, USA. Association for Computing Machinery.

Chen Yang and Junjie Chen. 2026. Uncovering business logic bugs via semantics-driven unit test generation. *CoRR*, abs/2604.23509.

Yuntong Zhang, Zhiyuan Pan, Imam Nur Bani Yusuf, Haifeng Ruan, Ridwan Shariffdeen, and Abhik Roychoudhury. 2026. Code review agent benchmark. *CoRR*, abs/2603.23448.

Yunhua Zhao, Kostadin Damevski, and Hui Chen. 2023. A systematic survey of just-in-time software defect prediction. *ACM Comput. Surv.*, 55(10):201:1–201:35.

A Task Construction Details

Each task is a minimal database-backed REST API consisting of an SQLite schema and a backend implementation with 2–3 endpoints. The schemas adopt TPC-H-inspired naming conventions and entity relationships[1] (e.g., `customer`–`orders`, `orders`–`lineitem`, `product`), but are not TPC-H subsets: they are hand-designed to isolate each issue type with minimal tables and columns.

Base task statistics. Table 3 summarizes the dimensions of the 18 base tasks (Python/Flask/raw SQL). Tasks are deliberately minimal: 1–2 tables, 4–13 columns, 2–3 endpoints, and 41–93 non-empty lines of code.

[1] https://www.tpc.org/tpch/default5.asp

Table 3: Base task statistics (18 tasks, Python/Flask/raw SQL).

	Min	Max	Avg
Tables per schema	1	2	1.8
Columns per schema	4	13	8.7
Foreign keys per schema	0	1	0.8
Endpoints per task	2	3	2.2
Lines of code	41	93	53.4

Augmentation diversity. Each base task is rewritten into 11 additional variants via GPT-5.4 (see Appendix C.1), yielding $18 \times 12 = 216$ evaluation instances. The 12 variants span three languages (Python, JavaScript, Go), two frameworks per language (Flask/FastAPI, Express/Fastify, Gin/Fiber), and two data-access patterns (raw SQL and ORM: SQLAlchemy, Sequelize, GORM). Table 4 reports lines of code across all 216 instances. Go variants are the most verbose (avg. 134 LoC), Python the most concise (avg. 73 LoC); ORM variants are consistently longer than their raw-SQL counterparts.

Augmentation reliability. The augmentation is unlikely to alter task semantics for two reasons. First, the semantic signal that makes an issue detectable, or confirms its absence, is embedded in endpoint docstrings and inline comments (e.g., "*Return the summary of all existed historical orders*"), and the augmentation prompt explicitly requires preserving all comments unchanged. Second, the base tasks are deliberately small (avg. 53 LoC), making the translation a straightforward code-rewriting task in well-known languages and frameworks, well within the capability frontier of GPT-5.4.

Table 4: Lines of code across all 216 evaluation instances.

Grouping		Min	Max	Avg
Language	Python	41	190	72.9
	JavaScript	35	210	95.8
	Go	78	325	134.2
Data access	Raw SQL	35	265	83.8
	ORM	52	325	118.1
All 216 instances		35	325	101.0

Positive task examples. Below we show one positive base task per issue category. In each case, the issue arises from the interaction between two endpoints.

cascade_delete. *Schema:* `customer` and `orders` (with a foreign key to `customer`). *Endpoints:* `DELETE /api/customers/:id`, which removes the customer together with all of their orders via a cascading delete; and `GET /api/orders/summary`, which aggregates over all rows in `orders` and is explicitly documented as returning a summary of *all orders that have existed*. *Issue:* after a customer deletion, the corresponding order rows are physically removed, so `/api/orders/summary` silently omits them from its aggregate. The endpoint therefore violates its documented contract: rather than reporting over all historical orders, it reports only over those whose owning customer has not yet been deleted. Detection requires the model to recognize that the cascading delete in the writer endpoint destroys precisely the rows that the reader endpoint's documented contract commits to preserving.

stale_aggregate. *Schema:* `orders` and `lineitem` (with a foreign key to `orders`). *Endpoints:* `PUT /api/orders/:id/lineitems/:num`, which updates the pricing fields of a line item (`l_extendedprice`, `l_discount`, `l_tax`); `GET /api/orders/:id`, which reads `orders.o_totalprice` directly; and a third endpoint for order creation that materializes `o_totalprice` from the corresponding line-item pricing fields at insertion time. *Issue:* the update endpoint mutates line-item prices without recomputing the cached `o_totalprice` on the parent order, so subsequent reads via `GET /api/orders/:id` return a stale aggregate. Detecting the issue requires the model to recover the semantic relationship between line-item pricing fields and `o_totalprice` from the creation endpoint, where this relationship is the only place it is made explicit, and to recognize that the update path fails to preserve it.

exposed_record. *Schema:* `product`. *Endpoints:* `POST /api/products/:id/discontinue`, which sets `p_status` to `'discontinued'`; and `GET /api/products/:id`, which is explicitly documented as returning only purchasable products. *Issue:* the read endpoint retrieves the product row without filtering on `p_status`, so a product that has been discontinued via the first endpoint continues to be served by the second as if it were available. The endpoint therefore violates its documented contract, and detection requires the model to connect the state transition introduced by `/discontinue` to the eligibility predicate implied by "purchasable" in the read endpoint's specification.

In all cases, each endpoint is locally plausible; the issue emerges only from cross-endpoint reasoning about the intended semantics of the persistent state.

B Evaluation Details

Task–label structure. Each of the 18 base tasks targets exactly one issue category (`cascade_delete`, `stale_aggregate`, or `exposed_record`). For each category we construct 3 positive tasks (issue present) and 3 negative tasks (issue absent), yielding 6 tasks per category. After augmentation into 12 variants, this produces $6 \times 12 = 72$ evaluation instances per category, for a total of 216. The positive/negative split is balanced: each category contains 36 positive and 36 negative instances.

Scoring pipeline. The two prompting regimes produce different output formats and are scored differently:

- **DEF.** The model is instructed to return a JSON array of `{name, detected, description}` objects. We parse the JSON output and compare each `detected` field deterministically against the ground-truth label. If the model's output cannot be parsed as valid JSON, the instance is excluded from metrics calculation.
- **FREE.** The model produces free-form text (also requested as JSON, but with model-chosen issue names). Because the model may use different terminology or framing, deterministic matching is not possible. Instead, two of the authors independently classified all 1,080 FREE outputs (216 per model $\times$ 5 models) as described below.

FREE annotation procedure. All 1,080 FREE outputs (216 instances × 5 models) were classified by two of the authors, working independently. Each annotator was presented with a spreadsheet containing, for each instance: the task name, the variant, the ground-truth label (whether the issue is present or absent, together with an explanation of why), and the raw model output. For each row the annotator marked TRUE if the model's output describes the target issue, or FALSE if it does not; no partial credit was given. The annotation is straightforward: each ground-truth finding is a short, specific statement (e.g., "deleting a department cascades to employees"), and model outputs are typically well-structured JSON or clearly itemised lists, leaving

little room for interpretation. This made exhaustive annotation of all instances by both annotators feasible. The two annotators agreed on 99.7% of instances (Cohen's k = 0.99); disagreements were solved by discussion and the resulting adjudicated labels are used for all reported FREE metrics. All annotations, including both annotators' raw labels, are released with the benchmark.

LLM judge validation. To assess whether the manual classification step could be automated, we ran a post-hoc comparison against an LLM judge. We prompted GPT-5.4 (temperature 0) with each expected ground-truth finding (name, expected polarity, and description) together with the model's raw output, and asked it to determine whether the model's response matches the expected finding. The judge matched on root cause rather than terminology, and returned a binary `detected` label per finding. Across all 1,080 instances, the LLM judge (GPT-5.4) agrees with both annotators on $\geq$92% of instances ($\kappa \geq 0.78$). All disagreements go in the same direction: the judge over-accepts borderline model outputs that both human annotators rejected (FP), while it never misses an issue that either human found (FN = 0 across all models and categories). This suggests that an LLM judge provides a reliable upper-bound approximation of human classification and could serve as a practical substitute for manual annotation in larger-scale evaluations.

Metrics. For each (model, prompt regime, issue category) triple, we compute binary classification counts over the corresponding 72 instances: true positives (issue present and detected), false positives (issue absent but flagged), true negatives (issue absent and not flagged), and false negatives (issue present but missed). From these counts we derive precision, recall, and F_1.

Models. We evaluate five models spanning five providers: GPT-5.4 (OpenAI, 2026), Kimi-K2.6 (AI, 2026), DeepSeek-V3.1 (DeepSeek-AI, 2024), Gemma-4-31b (Deepmind, 2026) and Qwen3.5-397B-A17B (Qwen Team, 2026).

DEF Output Parsing In the DEF setting, models are instructed to return a structured JSON array. If the output cannot be parsed as valid JSON, the instance is excluded from metrics. Table 5 reports exclusion rates per model.

Across all models, 99.7% of instances was successfully parsed. Only Kimi-K2.6 produced 3 unparseable outputs, all on `stale_aggregate` negative tasks. As a sensitivity check, we recomputed metrics treating these 3 exclusions as false positives (worst case): Kimi-K2.6 `stale_aggregate` F_1 drops from 0.97 to 0.94, leaving all reported findings unchanged.

Table 5: DEF instances successfully parsed per model (out of 216 expected).

Model	CD	SA	ER	Total
GPT-5.4	72/72	72/72	72/72	216/216
Kimi-K2.6	72/72	69/72	72/72	213/216
DeepSeek-V3.1	72/72	72/72	72/72	216/216
Qwen3.5-397B-A17B	72/72	72/72	72/72	216/216
Gemma-4-31b	72/72	72/72	72/72	216/216

C LLM prompts

C.1 Task Augmentation Prompt

Each base task (Python/Flask/raw SQL) is rewritten into 11 variants using GPT-5.4 (temperature 0.0).

```
You are an expert backend software engineer.

Rewrite the application below from Python / Flask / raw SQL into {target}.

## Strict requirements
- Preserve every endpoint exactly: same URL paths, HTTP methods, query parameters, and JSON request/response shapes.
- Preserve database access semantics without exception: same rows returned, same cascade deletion order, same JOINs and filters.
- Use SQLite as the underlying database engine.
- Output a single complete, runnable source file. No markdown fences, no prose.
- Keep all comments from the original file.

⟨ database schema (SQLite) ⟩
⟨ original implementation (Python / Flask / raw SQL) ⟩

Output only the rewritten source file for {target}.
```

C.2 Detection Prompts

Both regimes share the same output contract: a JSON array of `{name, detected, description}` objects. They differ only in the system instructions.

FREE prompt.

```
You are an expert backend data-layer code reviewer. Your job is to analyze application code and flag any possible issue.

## Ground rules
```

```
- The DB schema is assumed correct. Do
NOT flag schema design issues.
- Do NOT flag syntactic bugs, style
issues, missing input validation,
generic performance or concurrency
concerns.
- If multiple independent issues exist,
report each as its own item.
- If no issues are found, return an empty
array.
```

⟨ *database schema* ⟩
⟨ *application code* ⟩

⟨ *JSON output schema* ⟩

DEF prompt. Identical to FREE, except the ground rules instruct the model to analyze *only* the specified issue, and an issue definition block is prepended to the inputs:

```
## Issue definition
```
⟨ *natural-language definition from Table 1* ⟩

The exact definitions injected into the DEF prompt are:

- `cascade_delete` — *"A delete operation that removes rows from a child table (via explicit application-level DELETEs or ON DELETE CASCADE) in a way that causes observable data loss for another endpoint that needs those child rows to exist even after the parent is gone."*
- `stale_aggregate` — *"A write endpoint modifies source-of-truth rows in a child table, but a denormalized/pre-computed aggregate value in a parent table is not recalculated. Another endpoint returns the stale aggregate, producing incorrect results."*
- `exposed_record` — *"A write endpoint changes the state of a record (or a related record) such that it should no longer be visible or available to clients, but a read endpoint still serves it without checking."*

D Human Validation Details

Participants and assignment. We recruited 10 backend developers (unpaid volunteers), each with at least 3 years of professional experience building data-intensive applications. Six experts participated in two annotation rounds, yielding 16 expert–round assignments in total. Each assignment consisted of exactly three positive tasks: one per issue category (`cascade_delete`, `stale_aggregate`, `exposed_record`) drawn from a distinct language/framework variant so that every annotation covers a unique (task, variant) pair. The first round used Python/Flask (base tasks); subsequent rounds assigned FastAPI, Express, Fastify, and Go/Gin variants. In total, experts annotated 39 distinct positive instances out of 108, spanning 5 framework variants.

Procedure. Experts were made aware of the use of the data. Then they received the same inputs as models in the FREE setting: the database schema and application code, with no issue definitions, no hints about which categories to look for, and no time limit. They were asked to report any data-layer issue they could identify.

Results. Table 6 reports detection rates per issue category. Every category was independently identified by at least three experts, confirming that the tasks are well-formulated and the issues are discoverable from code and schema alone. At the same time, the overall detection rate of 41% (16/39) underscores that semantic data-layer reasoning is a genuinely difficult task, even for experienced developers.

Table 6: Human expert detection rates on positive instances, by issue category.

Issue category	Detected	Total	Recall
`cascade_delete`	3	13	23%
`stale_aggregate`	4	13	31%
`exposed_record`	9	13	69%
Overall	16	39	41%

Notably, the human recall hierarchy—visibility (69%) > consistency (31%) > retention (23%)—mirrors the LLM detectability ranking observed in Table 2.

Design rationale: coverage over redundancy. Because the ground truth is established by construction (tasks are authored with a known target issue), the purpose of the human study is to confirm *solvability*, not to establish labels via annotator consensus. We therefore prioritized breadth: each expert annotated a different (task, variant) pair, covering 39 of the 108 positive instances across 5 framework variants, rather than assigning multiple annotators to the same instance. This design validates that the issues remain discoverable across augmented variants and languages, which is the relevant property for our benchmark.

E Compliance Bias Insights

Within the DEF setting, models demonstrate a pronounced tendency to classify all `cascade_delete` tasks uniformly as positive instances. This behavior

suggests that models primarily operate via surface-level heuristic pattern matching rather than executing deep semantic reasoning over the application context. To empirically illustrate this phenomenon, we present a representative case study below.

Consider a standard one-to-many relation between two entities, `customer` and `orders`. The application layer exposes two distinct endpoints, each implementing a single functional requirement: (i) `delete_customer`, which deletes a target customer record alongside all corresponding order entries and (ii) `orders`, which retrieves historical order data of a targeted active customer. To explicitly state the semantic constraints of the data retrieval operation, the `orders` function is annotated with the following docstring: *Return orders tied to active customers.*

This scenario does not constitute a valid instance of a `cascade_delete` semantic issue. Because the `orders` endpoint strictly filters data based on customer existence, and given that the system provides no additional functionality to access legacy or orphaned orders, cascading the deletion from the parent to the child table is semantically sound. There is no operational requirement to persist these records post-deletion; hence, no functional data loss occurs.

Conversely, models exhibit a conservative false-positive bias when evaluating this setup. They misclassify this safe architectural pattern as a vulnerability due to a superficial reliance on the presence of the cascading delete mechanism itself. The models fail to evaluate the composition of application-level invariants that differentiate this benign instance from an actual semantic defect.

An illustrative example of an LLM-generated false positive (GPT-5.4) for the given scenario is the following:

```
In delete_customer, the explicit
child-table delete DELETE FROM orders
WHERE o_custkey = ? removes all orders
before deleting the parent customer.
This matches an application-level
cascade delete, and it causes observable
data loss because the /api/orders
endpoint depends on rows in orders
existing to return order data.
```

E.1 Neutrality Ablation

An alternative explanation for the compliance bias is generic instruction-following: having been told to look for a specific issue, the model may report it because it was asked to, independently of the code. Two observations argue against this account. First, if the effect were driven by instruction-following alone, it should appear uniformly once any definition is supplied; instead DEF precision is 0.85 for visibility and 0.90 for consistency, but only 0.57 for retention (Table 2), the category whose negatives are structurally closest to its positives.

Second, we test the account directly. We reran the full DEF evaluation on `cascade_delete` (72 instances, temperature 0) with a neutrality cue added to the ground rules, which explicitly licenses a negative answer:

```
- The issue may or may not be present
in the code below. Concluding that the
issue is absent is an equally valid
and expected outcome; do not assume the
issue must be present.
```

Everything else in the prompt is unchanged. We ran the ablation on the strongest closed-weight and open-weight models.

Table 7: Neutrality ablation on `cascade_delete` (DEF, 36 positive and 36 negative instances). TN is the number of negatives correctly rejected.

Model	Prompt	P	R	F1	TN
GPT-5.4	original	0.61	1.00	0.76	13/36
	+ neutrality	0.61	1.00	0.76	13/36
Kimi-K2.6	original	0.57	1.00	0.73	9/36
	+ neutrality	0.62	1.00	0.77	14/36

For GPT-5.4 the outcome is identical on every instance. For Kimi-K2.6 five additional negatives are rejected, moving precision from 0.57 to 0.62, while recall stays at 1.00 for both models. Explicitly licensing a negative answer therefore leaves the over-flagging almost intact.

F Detectability Across Implementation Idioms

To assess whether implementation idioms influence detectability, we report F_1 broken down by language, data-access pattern, and issue category across all five models.

Language. Table 8 reports aggregate F_1 by language. In the FREE setting, JavaScript and Go yield similar aggregate F_1 (0.49), with Python slightly lower (0.45). In the DEF setting, performance is remarkably stable across languages (0.85–0.87), confirming that the discovery gap and detectability hierarchy are not artifacts of a single language.

Data-access pattern. Table 9 reports aggregate F_1 by data-access style. In the FREE setting,

Table 8: Aggregate F_1 by language (all models, all issue categories).

	CASC. DEL.		STALE AGG.		EXP. REC.	
Lang.	Free	Def	Free	Def	Free	Def
Python	0.03	0.71	0.49	0.92	0.83	0.91
JS	0.03	0.74	0.57	0.94	0.87	0.89
Go	0.03	0.73	0.58	0.95	0.86	0.92

raw SQL variants are slightly easier to review ($F_1 = 0.50$) than ORM variants (0.45), with the gap concentrated in `cascade_delete` (0.06 vs. 0.00): ORM abstractions appear to obscure the cascading side effect. In the DEF setting, the gap vanishes (0.86 vs. 0.85), indicating that explicit definitions compensate for the added abstraction layer.

Table 9: Aggregate F_1 by data-access pattern (all models, all issue categories).

	CASC. DEL.		STALE AGG.		EXP. REC.	
Access	Free	Def	Free	Def	Free	Def
Raw SQL	0.06	0.73	0.58	0.95	0.87	0.91
ORM	0.00	0.73	0.51	0.93	0.84	0.90

Summary. The dominant factor is issue category, not implementation idiom. Language and data-access pattern introduce only minor variation ($\leq$0.09 F_1), and this variation disappears almost entirely under DEF.

G Prompt Sensitivity Ablation

To verify that DEF results are not an artifact of one specific definition wording, we re-ran all DEF evaluations with *paraphrased* definitions (P2) that preserve the same semantics but alter vocabulary and sentence structure. For example, the original `cascade_delete` definition is framed around "a delete operation that removes rows from a child table … causes observable data loss"; the paraphrase reframes it as "a parent record is deleted, and its dependent child records are also removed … a separate read endpoint now returns incomplete or missing data." We ran P2 on two models with contrasting profiles: GPT-5.4 (strongest closed-weight) and Kimi-K2.6 (strongest open-weight).

Table 10 shows that DEF performance is stable across phrasings: the mean absolute F_1 difference is 0.03, and the maximum is 0.10 (GPT-5.4 on `exposed_record`, where P2 actually *improves* precision from 0.82 to 1.00). Recall is identical across all conditions, and the compliance bias pattern on `cascade_delete` persists regardless of wording. These results confirm that the findings reported in Table 2 are robust to definition phrasing and not driven by a specific lexical choice.

Table 10: Prompt sensitivity: original (P1) vs. paraphrased (P2) DEF definitions (temperature 0).

		P1 (orig.)		**P2 (paraph.)**		
Model	**Issue**	P	F1	P	F1	$\lvert\Delta\rvert$
GPT-5.4	CD	0.61	0.76	0.60	0.75	0.01
	SA	1.00	1.00	1.00	1.00	0.00
	ER	0.82	0.90	1.00	1.00	0.10
Kimi-K2.6	CD	0.57	0.73	0.57	0.73	0.00
	SA	0.95	0.97	0.82	0.90	0.07
	ER	0.84	0.91	0.86	0.92	0.01

Temperature and sampling variance. All main experiments use temperature 0. To verify that this choice does not bias the results, we re-ran GPT-5.4 and Qwen3.5-397B in the DEF setting and GPT-5.4 in the FREE setting at temperature 0.7 with 3 independent runs each. Table 11 reports mean F_1 and standard deviation (σ) alongside the original temperature-0 result.

Table 11: Temperature sensitivity: T=0 vs. T=0.7 (3 runs). σ is the standard deviation across runs.

Model	**Prompt**	**Issue**	T=0	T=0.7	σ
GPT-5.4	DEF	CD	0.76	0.76	0.00
		SA	1.00	1.00	0.00
		ER	0.90	0.92	0.02
	FREE	CD	0.15	0.18	0.04
		SA	0.84	0.84	0.01
		ER	0.89	0.83	0.02
Qwen3.5-397B	DEF	CD	0.71	0.71	0.01
		SA	0.90	0.90	0.01
		ER	1.00	0.99	0.00

DEF results are nearly deterministic: across both models and all categories, $\sigma \leq 0.01$ and the maximum $|\Delta F_1|$ between T=0 and the T=0.7 mean is 0.02. FREE shows slightly more variance (σ up to 0.04 on the low-base-rate `cascade_delete` category), but the discovery gap and detectability hierarchy remain fully intact. These results confirm that the findings are robust to sampling temperature.

H What Models Report Under FREE

One reading of the discovery gap is that it merely reflects a larger search space: under FREE the

Table 12: Content of FREE reviews. "Empty" is the share of reviews reporting no issue at all; the last column restricts this to positive instances, where an issue is present.

Model	Findings/review	Empty	Empty (pos.)
GPT-5.4	0.50	56%	40%
Kimi-K2.6	3.07	1%	0%
DeepSeek-V3.1	4.25	0%	0%
Gemma-4-31b	0.88	14%	4%
Qwen3.5-397B-A17B	1.42	5%	1%
All	2.02	15%	9%

model must decide what to look for, so the target issue could be present in the review but diluted among many other findings, or crowded out by them. Two properties of our setup already bound this effect. The FREE prompt restricts the review to data-layer issues and explicitly excludes syntax, style, input validation, performance, and concurrency concerns (Appendix C.2), and each task exposes only 2–3 endpoints, which leaves few candidate cross-endpoint interactions.

The reviews themselves confirm it. Across them, models report 2.02 findings per review on average, so the target issue is not lost in a flood of candidates. The pattern is clearest for GPT-5.4, the strongest FREE model: 40% of its reviews on positive instances are *empty*. It does not bury the issue among distractors, it finds nothing at all. Table 12 reports the per-model figures.

A pure search-space account also predicts that an explicit definition should help all categories comparably. It does not: the average gain is $+0.06$ for visibility, $+0.38$ for consistency, and $+0.70$ for retention (Table 2). We therefore read the gap as primarily a matter of which semantic failure modes a model brings to an unguided review, which is also what makes an explicit taxonomy useful in practice.

I Per-Base-Task Results

Reporting by base task shows whether a headline number is carried by a single scenario. Table 13 counts, for each of the 18 base tasks, how many of its 60 outcomes (12 variants $\times$ 5 models) are correct: for a positive task (Pi) a correct outcome is detecting the target issue, for a negative task (Ni) it is not flagging one.

No reported effect depends on one scenario. Every positive retention task is discovered in at most 2 of 60 FREE outcomes and recognized in 60 of 60 DEF outcomes, and every positive visibility task is discovered in at least 40 of 60 FREE outcomes, so the two ends of the detectability hierarchy do not hinge on a single task.

The one scenario that crosses categories is consistency P3 (43/60), which falls into the visibility range. Under FREE, consistency P3 (72%) is discovered far more often than P1 (35%) and P2 (27%): in P3 the aggregate's name makes explicit that it derives from the updated field, whereas P1 and P2 require inferring that relationship from domain semantics through a third endpoint.

Additionally, under DEF, retention N1 and N2 are falsely flagged in nearly every outcome (0% and 2% correct rejections) while N3 is mostly rejected correctly (75%). In N1 and N2 a cascade delete does occur but is harmless, since no other endpoint reads the deleted rows; in N3 no cascade delete occurs at all. Models thus key on the presence of the surface pattern named in the definition rather than on its downstream observability, which is the mechanism we describe as compliance bias (Section E).

Table 13: Correct outcomes per base task, out of 60 (12 variants $\times$ 5 models). Pi = positive task, Ni = negative task.

Category	Task	FREE	DEF
Visibility	P1	40/60 (67%)	60/60 (100%)
	P2	51/60 (85%)	60/60 (100%)
	P3	44/60 (73%)	60/60 (100%)
	N1	60/60 (100%)	44/60 (73%)
	N2	60/60 (100%)	60/60 (100%)
	N3	60/60 (100%)	38/60 (63%)
Consistency	P1	21/60 (35%)	60/60 (100%)
	P2	16/60 (27%)	60/60 (100%)
	P3	43/60 (72%)	60/60 (100%)
	N1	59/60 (98%)	58/60 (97%)
	N2	60/60 (100%)	40/60 (67%)
	N3	58/60 (97%)	55/60 (92%)
Retention	P1	1/60 (2%)	60/60 (100%)
	P2	2/60 (3%)	60/60 (100%)
	P3	0/60 (0%)	60/60 (100%)
	N1	60/60 (100%)	0/60 (0%)
	N2	60/60 (100%)	1/60 (2%)
	N3	59/60 (98%)	45/60 (75%)

J Hand-Crafted vs. Augmented Tasks

198 task instances are generated via code augmentation using GPT-5.4, which is also one of the evaluated models. To demonstrate that the findings hold on the base set alone, and that GPT-5.4 isn't favored by its own rewrites, we compared the results

of the hand-crafted tasks against the augmented instances.

Findings replicate on the hand-crafted subset. Table 14 splits the results by origin. All three findings hold on the base tasks: `cascade_delete` FREE F_1 is 0.00 for all five models (vs. 0.03 on the augmented instances) against 0.72 under DEF; the detectability hierarchy is preserved under FREE (visibility 0.92 $>$ consistency 0.68 $>$ retention 0.00); and DEF recall is 1.00 in every cell, with `cascade_delete` precision at 0.56 on base and 0.58 on augmented instances. The largest base–augmented difference under DEF is 0.04 F_1.

No consistent self-preference for the generator. If augmentation favored the generator, GPT-5.4's change from base to augmented instances should exceed that of the other models. Table 15 reports the difference-in-differences $\text{DiD} = \Delta_{\text{GPT-5.4}} - \overline{\Delta}_{\text{others}}$, with $\Delta = F_1^{\text{aug}} - F_1^{\text{base}}$. The signal is neither consistent nor large. In one cell (`cascade_delete`, FREE) GPT-5.4 scores higher on its own rewrites; in another (`stale_aggregate`, FREE) the positive DiD is driven entirely by the other models scoring *lower* on the augmented instances ($\overline{\Delta} = -0.17$) rather than by GPT-5.4 scoring higher ($\Delta = +0.04$); and in both `exposed_record` cells GPT-5.4 does *worse* on its own rewrites.

Table 14: Hand-written base instances (18) vs. GPT-5.4 augmentations (198), macro-averaged over the five models.

		Base (n=18)			**Augmented** (n=198)		
Issue	**Prompt**	P	R	F1	P	R	F1
ER	FREE	1.00	0.87	0.92	1.00	0.74	0.84
	DEF	0.92	1.00	0.95	0.85	1.00	0.91
SA	FREE	1.00	0.53	0.68	0.98	0.44	0.55
	DEF	0.90	1.00	0.94	0.90	1.00	0.94
CD	FREE	0.00	0.00	0.00	0.15	0.02	0.03
	DEF	0.56	1.00	0.72	0.58	1.00	0.73

K Statistical Significance

The 12 variants of a base task share its scenario, its labels, its docstrings, and its query structure, so they are not independent observations: the independent experimental unit is the base task, of which there are 18 (3 positive and 3 negative per issue category). We therefore quantify uncertainty with a cluster bootstrap over base tasks. In each of $B = 10{,}000$ iterations we resample base tasks with replacement, stratified by polarity (3 positive and 3 negative per category), each sampled task contributing all 12 of its variant outcomes, and recompute F_1 on the resample; we report 95% percentile intervals. For the DEF−FREE difference we apply the *same* task resample to both regimes, so that scenario difficulty affects both equally and the interval isolates the effect of the prompt. The *Average* row macro-averages the five per-model F_1 values within each iteration, matching the averaging convention of Table 2.

Table 15: Difference-in-differences on F_1 ($\Delta = F_1^{\text{aug}} - F_1^{\text{base}}$).

Issue	**Prompt**	Δ GPT-5.4	$\overline{\Delta}$ others	DiD
ER	FREE	−0.12	−0.07	−0.05
	DEF	−0.11	−0.02	−0.09
SA	FREE	+0.04	−0.17	+0.21
	DEF	+0.00	−0.00	+0.00
CD	FREE	+0.16	+0.00	+0.16
	DEF	+0.01	+0.02	−0.01

Table 16 reports the resulting intervals. The discovery gap is significant for retention (`cascade_delete`) for all five models and for consistency (`stale_aggregate`) for four of five, whereas for visibility (`exposed_record`) the paired interval includes zero for four of five models and on average. This is precisely the ordering claimed by the detectability hierarchy: the categories that models discover unprompted are the ones for which an explicit definition adds nothing measurable.

L Representative Repairs

The analyzed issues are detection targets, not open-ended design problems: each admits small, local repairs. We list representative ones per category, both to characterize the issues concretely and to support future work that uses the benchmark for repair rather than detection.

Retention (`cascade_delete`). Replace the physical deletion of child rows with a soft delete (a status or `deleted_at` column) and filter on it where the rows should no longer be visible. Alternatively, remove the cascade and require explicit child cleanup, or archive the child rows into a history table that the reading endpoint queries directly.

Table 16: Cluster bootstrap over the 18 base tasks (10,000 iterations, stratified by polarity, 95% percentile intervals). ER = `exposed_record`, SA = `stale_aggregate`, CD = `cascade_delete`. * = the 95% interval of the paired DEF−FREE difference excludes zero.

Model	**Issue**	FREE F_1	[95% CI]	DEF F_1	[95% CI]	Δ DEF−FREE	[95% CI]
GPT-5.4	ER	0.89	$[0.74, 1.00]$	0.90	$[0.75, 1.00]$	$+0.01$	$[-0.15, 0.20]$
	SA	0.84	$[0.71, 0.93]$	1.00	$[1.00, 1.00]$	$+0.16^*$	$[0.07, 0.29]$
	CD	0.15	$[0.00, 0.28]$	0.76	$[0.67, 1.00]$	$+0.61^*$	$[0.43, 0.86]$
Kimi-K2.6	ER	0.94	$[0.86, 1.00]$	0.91	$[0.86, 1.00]$	-0.03	$[-0.12, 0.07]$
	SA	0.62	$[0.40, 0.91]$	0.97	$[0.92, 1.00]$	$+0.36^*$	$[0.06, 0.60]$
	CD	0.00	$[0.00, 0.00]$	0.73	$[0.67, 0.89]$	$+0.73^*$	$[0.67, 0.89]$
DeepSeek-V3.1	ER	0.74	$[0.40, 1.00]$	0.76	$[0.67, 1.00]$	$+0.02$	$[-0.24, 0.37]$
	SA	0.24	$[0.15, 0.40]$	0.84	$[0.67, 1.00]$	$+0.59^*$	$[0.41, 0.82]$
	CD	0.00	$[0.00, 0.00]$	0.71	$[0.67, 0.80]$	$+0.71^*$	$[0.67, 0.80]$
Gemma-4-31b	ER	0.97	$[0.96, 1.00]$	1.00	$[1.00, 1.00]$	$+0.03$	$[0.00, 0.04]$
	SA	0.80	$[0.59, 1.00]$	1.00	$[1.00, 1.00]$	$+0.20$	$[0.00, 0.41]$
	CD	0.00	$[0.00, 0.00]$	0.75	$[0.67, 1.00]$	$+0.75^*$	$[0.67, 1.00]$
Qwen3.5-397B-A17B	ER	0.69	$[0.50, 0.80]$	1.00	$[1.00, 1.00]$	$+0.31^*$	$[0.20, 0.50]$
	SA	0.33	$[0.00, 0.74]$	0.90	$[0.75, 1.00]$	$+0.57^*$	$[0.19, 1.00]$
	CD	0.00	$[0.00, 0.00]$	0.71	$[0.67, 0.80]$	$+0.71^*$	$[0.67, 0.80]$
Average	ER	0.85	$[0.77, 0.89]$	0.91	$[0.86, 1.00]$	$+0.06$	$[-0.00, 0.17]$
	SA	0.56	$[0.37, 0.79]$	0.94	$[0.88, 1.00]$	$+0.38^*$	$[0.18, 0.60]$
	CD	0.03	$[0.00, 0.06]$	0.73	$[0.67, 0.90]$	$+0.70^*$	$[0.62, 0.86]$

Consistency (`stale_aggregate`). Recompute the denormalized aggregate in every write path that touches its source rows, ideally inside the same transaction. Otherwise stop materializing it and derive it on read from the source-of-truth rows, or enforce the invariant below the application, with a database trigger or a generated column.

Visibility (`exposed_record`). Add the missing status predicate to the read endpoints. Alternatively, centralize the visibility filter in a single query helper or repository method so that new endpoints inherit it, or push it into the data-access layer as a default scope or view, so that serving a hidden record requires an explicit opt-out.

The last option in each category is the one that generalizes: the issues arise because a constraint on persistent state is enforced at some access paths and not others, so repairs that move the constraint to a single choke point prevent recurrence, whereas per-endpoint patches do not.