

Fragility Under Pressure: Evaluating the Iterative Stability of LLMs in Constrained Interactive Coding

Zhuoheng Han¹ Jin-Ge Yao² Zailong Tian¹

Xi Yang² Houfeng Wang^{1*}

¹State Key Laboratory for Multimedia Information Processing, Peking University

²BAAI FlagEval

2501112032@stu.pku.edu.cn

Abstract

Interactive coding has become an important mode of software development, where humans and large language models (LLMs) iteratively refine code under evolving requirements. Unlike static code generation, this setting requires models to preserve functional correctness while continuously incorporating new constraints across multiple turns. However, existing evaluations are largely single-turn and therefore provide limited insight into how models behave under iterative constraint pressure. To study this gap, we introduce STRIDE, a controlled diagnostic for feedback-driven code repair under a fixed set of verifiable constraints. Using STRIDE, we evaluate a broad set of state-of-the-art models and find that, across these models, single-turn coding performance is informative but insufficient for characterizing behavior under fixed-set iterative repair. In particular, models show a consistent asymmetry between different types of constraints: they handle avoidance-style requirements more robustly than prescriptive structural modifications. We also observe functional regression in individual STRIDE traces, including cases where a later output satisfies a larger fraction of the same checked repair subset while passing fewer hidden tests. These results characterize model behavior within fixed-set iterative repair and motivate evaluation beyond single-turn code generation.

1 Introduction

The ability of Large Language Models (LLMs) to iteratively refine code is increasingly viewed as an important step toward autonomous software engineering. In practice, however, software development is rarely a one-shot generation problem. Developers typically work through repeated cycles of editing, testing, and debugging, where code must be revised to satisfy evolving requirements while preserving existing functionality. In such settings, success depends not only on algorithmic reasoning, but also on robust *state maintenance*: a model must keep track of previously satisfied constraints, identify which parts of the program should change in response to new feedback, and implement local revisions without degrading the global behavior of the solution.

Despite rapid progress in code generation, current understanding of this capability remains heavily shaped by static evaluation. Widely used benchmarks such as HumanEval (Chen et al., 2021), LiveCodeBench (Jain et al., 2024) or SWE-bench (Jimenez et al., 2024) primarily evaluate success on predefined tasks, often emphasizing the final correctness of a generated solution or code modification rather than the process of iterative refinement. While these benchmarks are valuable for assessing standalone coding ability, they provide limited visibility into a different challenge that arises in interactive settings: whether a model can preserve correctness while incorporating new constraints over multiple rounds of feedback. In realistic collaborative workflows, the central difficulty is often not generating a solution

*Corresponding author: wanghf@pku.edu.cn.

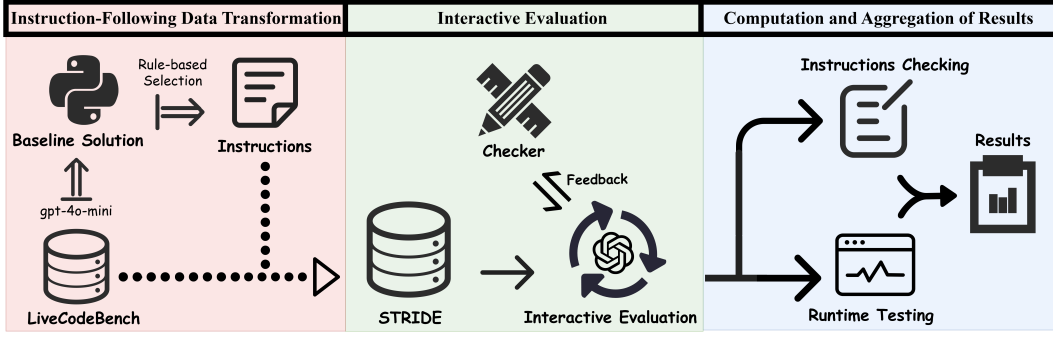


Figure 1: Overview of the STRIDE framework. The pipeline constructs multi-turn coding tasks by combining algorithmic problems with verifiable constraints.

from scratch, but modifying an existing solution under accumulating restrictions without introducing regressions elsewhere in the code.

This distinction is also motivated by conversational coding data. Resources such as ShareGPT (Chiang et al., 2023) suggest that multi-turn interaction is common in practice, with many coding sessions consisting of iterative clarification, revision, and repair rather than isolated code generation. These interactions place a distinct form of pressure on the model: it must respond to new instructions while maintaining consistency with prior edits, previously active requirements, and the overall logic of the program. Whether current LLMs can perform this kind of iterative refinement robustly remains an open question.

In this paper, we introduce **STRIDE**, a controlled diagnostic for feedback-driven code repair. STRIDE pairs an algorithmic problem with a fixed set of verifiable constraints and asks a model to revise its own solution from deterministic feedback. STRIDE complements one-shot code-generation benchmarks and open-ended agent evaluations. The protocol does not model requirement discovery, unrestricted human feedback, or tool-augmented software development.

Using STRIDE, we evaluate a broad set of leading models and find that single-turn coding performance is informative but insufficient for characterizing behavior under fixed-set iterative repair. In particular, we observe a consistent asymmetry in constraint handling: models are generally more robust at satisfying avoidance-style requirements than at implementing prescriptive structural changes. We also observe individual repair traces in which checker success increases on an unchanged repair subset while hidden-test performance falls.

In summary, our contributions are:

- We introduce **STRIDE**, a controlled and verifiable diagnostic for feedback-driven code repair under a fixed constraint set.
- We conduct a systematic evaluation of leading code-capable LLMs and find that single-turn coding ability is an informative but incomplete reference for behavior under fixed-set iterative repair.
- We document constraint-type asymmetries and functional regression in STRIDE traces, including repairs where checker success increases on an unchanged repair subset while hidden-test performance falls.

2 Related Work

2.1 Static Evaluation: From Functionality to Constraint Satisfaction

The evaluation of LLMs for programming has evolved from simple functional correctness to complex constraint adherence. Foundational benchmarks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) established the *pass@k* metric for single-turn

generation. Later efforts such as LiveCodeBench (Jain et al., 2024) tried to address data contamination and difficulty saturation, providing robust snapshots of model capabilities. Parallel to this, general instruction-following benchmarks like IFEval (Zhou et al., 2023) and FollowBench (Jiang et al., 2024) assess adherence to formatting constraints. More recent code-specific benchmarks, including CodeIF (Yan et al., 2025), CodeIF-Bench (Wang et al., 2025), and MultiCodeIF (Duan et al., 2025), move toward verifiable and fine-grained constraint following in coding settings. However, existing literature still largely treats coding capability and instruction following as separate competencies. These mostly static evaluations also miss how executable correctness and constraint adherence vary across repair turns.

2.2 Dynamics of Iterative Refinement and State Tracking

A key promise of LLMs is the ability to refine through multi-turn interaction. The recent popularity of agentic coding has drawn attention to evaluation settings (Jimenez et al., 2024; Merrill et al., 2026; Li et al., 2026) in which an LLM, equipped with an agentic scaffold, adapts its strategy in response to automatic environmental feedback rather than follow-up user instructions. Benchmarks like MT-Bench (Zheng et al., 2023) and MINT (Wang et al., 2023) emulate these conversational dynamics, often relying on LLM-as-a-judge or loose feedback mechanisms. Recent work such as ConvCodeWorld (Han et al., 2025) further formalizes this evaluation in coding scenarios by introducing reproducible environments with diverse feedback signals (e.g., compilation, execution, and verbal feedback) to assess code refinement capabilities. Related code-specific benchmarks such as CodeFlowBench (Wang et al., 2026) and FeedbackEval (Dai et al., 2026) show that iterative generation and repair remain fragile even with structured reuse or external feedback. While Multi-IF (He et al., 2024) scales constraint testing to dialogue, it prioritizes the quantity of accumulated instructions over the logical consistency of the resulting artifact.

Furthermore, recent studies on self-correction challenge the assumption that iterative feedback monotonically improves output quality. Laban et al. (2025) identify a significant unreliability gap in multi-turn interactions, finding that models frequently get lost and fail to recover from early assumptions, leading to substantial performance drops compared to single-turn settings. Our work connects these areas through a controlled, verifiable diagnostic that measures non-monotonic hidden-test outcomes during feedback-driven fixed-set repair.

3 Experimental Methodology: Simulating Multi-Turn Programmatic Interactions

To study whether code-capable LLMs preserve functional correctness during repeated repair, we introduce STRIDE, a controlled diagnostic built around a fixed set of verifiable constraints.

3.1 Task Formulation

We define each interactive coding session as $T = (P, I, H)$, where P is an algorithmic problem, $I = \{c_1, \dots, c_n\}$ is the complete fixed constraint set presented at Turn 1, and H is the conversation history. At Turn 0, the model produces an unconstrained solution S_0 . We set $A_1 = I$, and the model produces the first constrained repair S_1 . For $t \geq 1$, the generation-side selector computes an implementation-derived repair subset $A_{t+1} = G(S_t, A_t; I)$ and lists it in the next user message. The selector chooses constraints from I , while H retains the Turn-1 prompt and each later message asks the model to preserve earlier requirements. The selector checks every member of I on each turn. The protocol permits up to five constrained repair turns, Turns 1 through 5.

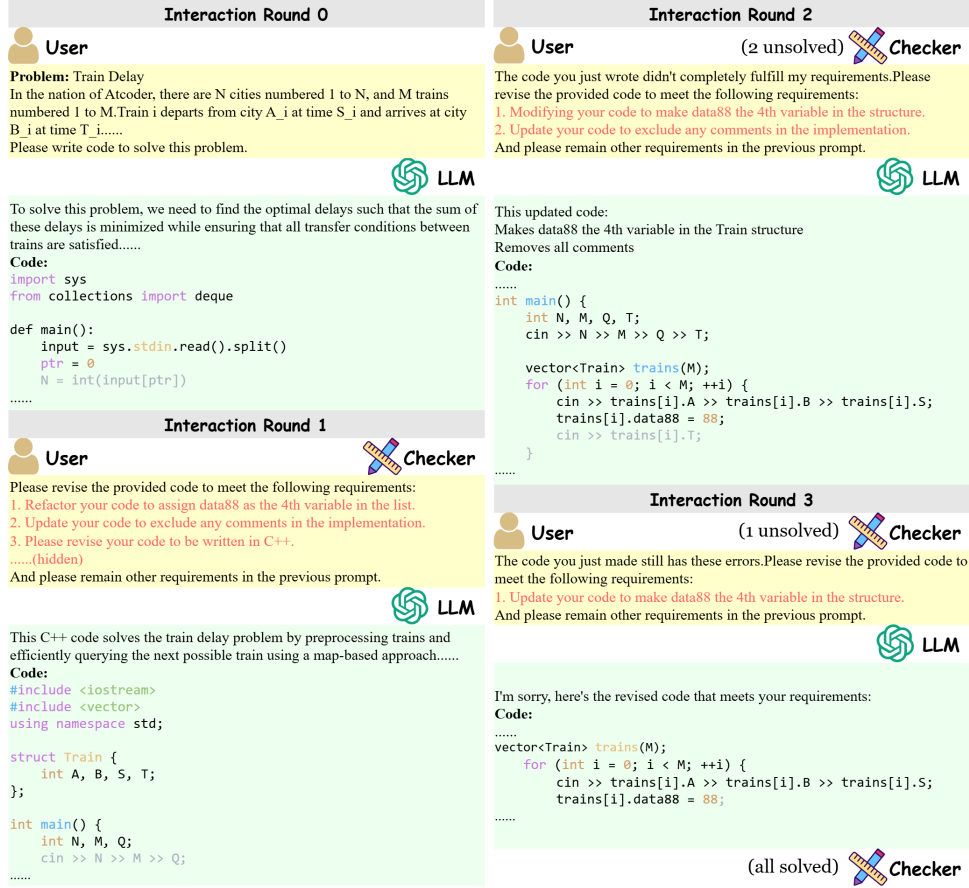


Figure 3: An interaction example produced by DeepSeek-V3. Model outputs appear in green boxes, and checker feedback appears in yellow boxes. Some details are omitted for clarity.

target set remains fixed. Figure 3 shows one interaction, and Appendix A.7 describes the implementation.

A manual audit of 200 sampled solution–constraint pairs yielded Cohen’s $\kappa = .89$, category-averaged precision of .962, and category-averaged recall of .937 for our automated checker; Appendix A.7 describes the audit scope and reporting limits.

3.4 Evaluation Setup

For each model, let N be the number of saved sessions accepted by the metric aggregator. A constraint-bearing record has a nonzero Turn index, a nonempty active constraint list, and a type other than runtime evaluation. We report five complementary metrics. Instruction Solving Number (ISN) counts sessions whose final constraint-bearing record satisfies every active constraint, and ISN Rate normalizes this count by N . Instruction-Following (IF) Pass Rate micro-averages individual active-constraint checker outcomes across sessions and constraint-bearing records. Average maximum recorded Turn (AvgTurn) is the mean of the largest stored Turn index in each session. Test Case Pass Rate (TCPR) micro-averages passed hidden tests at the selected runtime record, whereas Turn-0 Pass@1 is the session-level success rate of the unconstrained initial output. For session i , let A_{it} be the active constraints checked in record t , let $z_{itc} \in \{0, 1\}$ be the checker outcome for $c \in A_{it}$, and let ℓ_i be the final

Model	ISN	Avg. max		Test Case Pass Rate	Pass@1
		Turn	IF Pass Rate		
Gemma 3 27B IT	170 (19.32%)	2.26	55.27%	36.08%	22.16%
Llama 3.3 70B					
Instruct	292 (33.18%)	2.82	60.93%	24.83%	22.27%
GPT-4o mini	248 (28.18%)	3.38	53.87%	29.30%	24.09%
Qwen3-Coder					
30B-A3B-Instruct	274 (31.13%)	3.77	46.86%	55.24%	46.53%
DeepSeek-V3.2	317 (36.02%)	3.38	56.48%	60.86%	56.74%
Claude Sonnet 4.6	385 (43.75%)	3.63	66.83%	68.67%	69.32%
Gemini 3.1 Flash					
Lite Preview	295 (33.52%)	3.35	59.27%	73.50%	70.97%
GPT-5.4	358 (40.68%)	3.11	64.84%	65.89%	71.25%

Table 1: Point estimates for the eight main-text models on STRIDE. ISN is shown as count (rate), and Pass@1 refers only to the unconstrained Turn-0 output. Rows follow ascending Turn-0 Pass@1. **Bold** marks the largest observed point estimate for ISN, IF Pass Rate, Test Case Pass Rate, and Pass@1 and does not imply statistical significance. We do not rank average maximum recorded Turn because early completion and incomplete records can both reduce it. Appendix Table 5 reports the available two-sided 95% intervals and their saved-record scope.

constraint-bearing record in stored order. We define

$$\begin{aligned}
 \text{ISN} &= \sum_{i=1}^N \mathbf{1} \left[A_{i\ell_i} \neq \emptyset \wedge \bigwedge_{c \in A_{i\ell_i}} z_{i\ell_i c} = 1 \right], & \text{ISNRate} &= \text{ISN}/N, \\
 \text{IF} &= \frac{\sum_i \sum_t \sum_{c \in A_{it}} z_{itc}}{\sum_i \sum_t |A_{it}|}, & \text{AvgTurn} &= \frac{1}{N} \sum_i u_i, \\
 \text{TCPR} &= \frac{\sum_i p_{i\tau_i}}{\sum_i m_{i\tau_i}}, & \text{Pass@1} &= \frac{1}{N} \sum_i r_{i0}.
 \end{aligned}$$

Here u_i is the largest integer Turn stored for session i , or zero if none is present. The runtime index τ_i selects the last stored constrained runtime record and falls back to Turn 0 when no constrained runtime record exists; $p_{i\tau_i}$ and $m_{i\tau_i}$ are its passed and total hidden-test counts. A selected record with no tests contributes zero to both TCPR sums. The indicator r_{i0} equals one when the unconstrained Turn-0 output passes its complete hidden-test suite; if its saved pass rate is unavailable, the aggregator uses the Turn-0 success flag.

ISN uses the nonempty checker vector in the final stored constraint-bearing record and does not require runtime-test success. IF Pass Rate micro-averages active checker outcomes, so a constraint checked at several Turns contributes one observation per Turn. Average maximum recorded Turn includes all saved sessions and does not condition on ISN success; early completion and incomplete records can both reduce it. Test Case Pass Rate (TCPR) is a hidden-test micro-average, whereas Turn-0 Pass@1 is a session-level rate.

Uncertainty. We report two-sided 95% intervals over saved task sessions. We use Wilson score intervals for ISN and Turn-0 Pass@1 when the reported point estimate has matching session-level binary records. For average maximum recorded Turn, IF Pass Rate, and Test Case Pass Rate, we use a 10,000-replicate percentile session-cluster bootstrap with seed 2026. Each replicate resamples saved sessions within a model and recomputes IF Pass Rate and Test Case Pass Rate as ratios of summed numerators and denominators. The intervals describe across-task variation conditional on each stated saved-record cohort; they do not cover model-rerun variation, checker error, or missing-record bias. Appendix Table 5 reports the intervals and cohort scope.

We report the eight models in the main-text comparison in Table 1, and we list the full evaluated set in Table 4. We access GPT-5.4, GPT-4o mini, Claude Sonnet 4.6, DeepSeek-V3.2, and Gemini 3.1 Flash Lite Preview through OpenRouter. We run Llama 3.3 70B Instruct,

Qwen3-Coder 30B-A3B-Instruct, and Gemma 3 27B IT locally with vLLM on NVIDIA A800 hardware. For all eight main-text models, we set temperature to 0 and max_tokens to 8,192, and we allow up to five constrained repair turns per session. We provide the corresponding access identifiers or model repositories and retained generation-artifact dates in Appendix Table 8.

For each problem, the evaluated model first produces the unconstrained solution S_0 at Turn 0. Turn 1 presents the complete fixed constraint set I and elicits the first constrained repair S_1 . For Turns 2 through 5, the generation-side selector derives a repair subset from the preceding output and lists it in the next user message. Each request retains the full conversation history and asks the model to preserve earlier requirements.

3.5 Evaluation Results

Table 1 reports Turn-0 Pass@1 alongside ISN, IF Pass Rate, average maximum recorded Turn, and Test Case Pass Rate for the eight main-text models. Pass@1 provides a single-turn reference; the other metrics measure distinct aspects of behavior under fixed-set iterative repair. Section 4 examines constraint-type asymmetry and non-monotonic repair outcomes.

Claude Sonnet 4.6 has the largest observed ISN and IF Pass Rate in Table 1; Gemini 3.1 Flash Lite Preview has the largest observed Test Case Pass Rate, and GPT-5.4 has the largest observed Turn-0 Pass@1. DeepSeek-V3.2 has the largest observed ISN among the four reported open-weight models. We report these as observed point-estimate differences and do not conduct pairwise significance tests. Section 4 analyzes checker and runtime-test outcomes as separate behavioral measurements.

4 The Failure Modes of Iterative Stability

STRIDE complements static coding benchmarks by recording code, checker outcomes, and hidden-test outcomes across repair turns. We use these records to analyze three observed patterns: asymmetry across constraint types, non-monotonic repair outcomes, and cases where checker success and hidden-test performance move in opposite directions. Additional breakdowns by instruction type and difficulty are provided in the appendix.

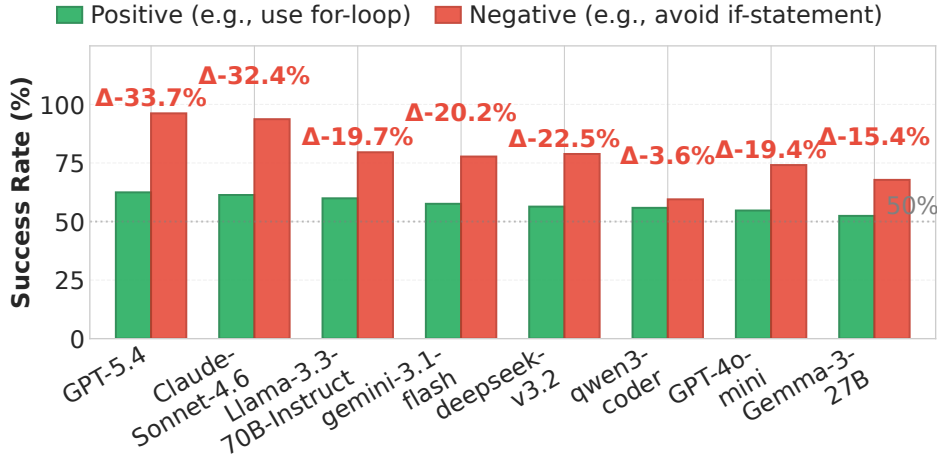


Figure 4: Performance comparison between Negative (Red) and Positive (Green) constraints. Models consistently handle avoidance-style constraints more robustly than prescriptive structural ones.

4.1 Constraint-Type Asymmetry Under Iterative Refinement

Our first finding is that robustness is not uniform across instruction types. As shown in Figure 4, models are consistently more successful on Negative Constraints, which ask them

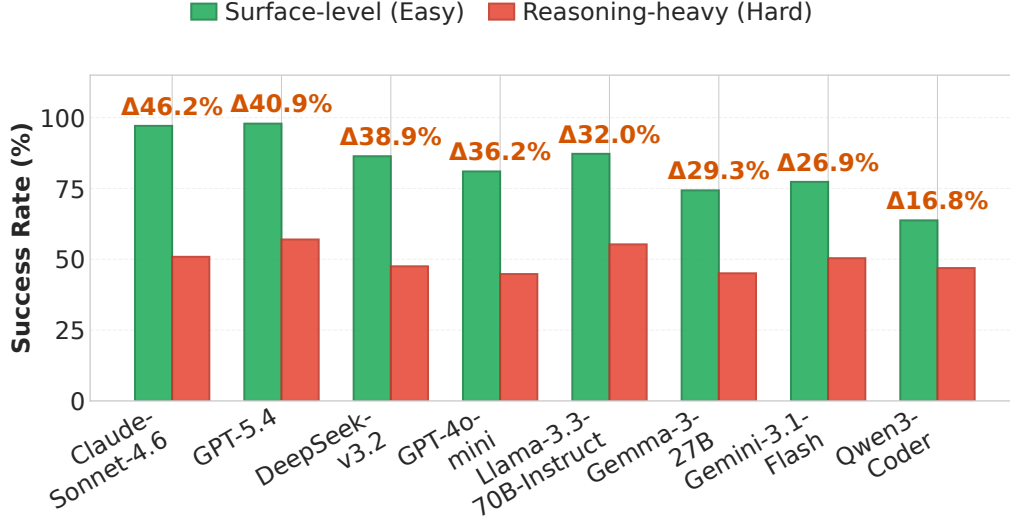


Figure 5: Mean checker success rates for constraint groups categorized as surface-level and reasoning-heavy, aggregated by model. The labels describe the constraint taxonomy rather than a measurement of internal reasoning.

to avoid specific patterns, than on Positive Constraints, which require explicit structural modification. This gap appears across nearly all evaluated models, with some families showing particularly large asymmetries. For example, GPT-5.4 exhibits a substantial gap between the two categories, indicating that prescriptive edits are far more fragile than avoidance-style revisions.

This result shows that checker success varies across the two constraint groups. The experiment does not isolate edit invasiveness or an internal state-tracking mechanism, so we treat the gap as a descriptive difference between the evaluated groups.

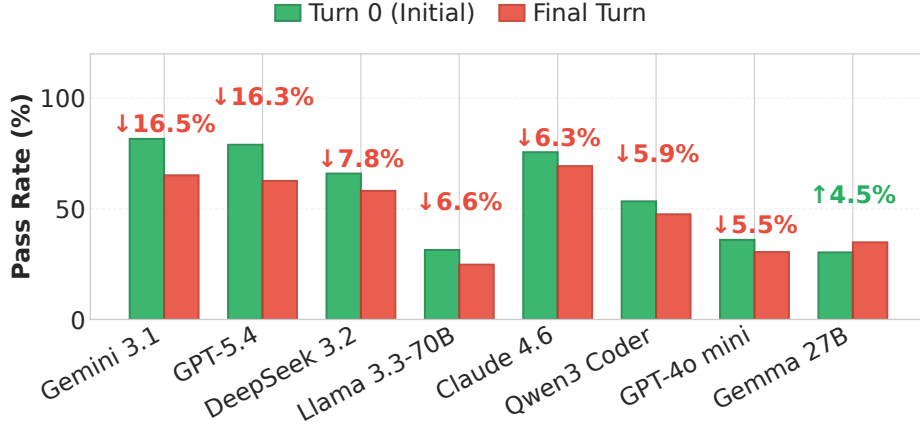
4.2 Non-Monotonic Outcomes During Fixed-Set Repair

STRIDE records hidden-test outcomes for each output, allowing us to inspect changes across repair turns. Some later outputs pass fewer hidden tests than earlier outputs. Figure 6a provides an aggregate Turn 0/Turn 5 view, but its bars use the outputs available at each turn and therefore do not constitute a paired estimate of within-session change.

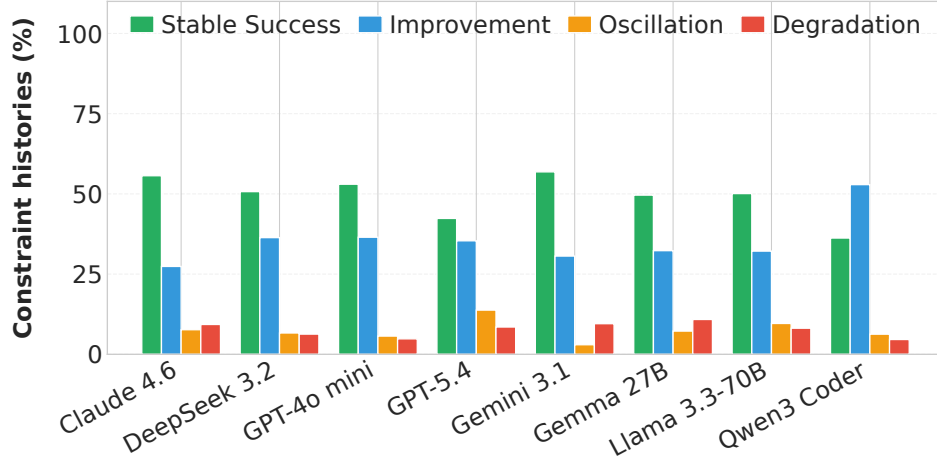
A raw-verified Gemma 3 27B IT trace provides a paired example. For task `abc302_a`, the active repair subset contained the same keyword for check at Turns 2 and 3. The checker result changed from failure to success, while hidden-test performance changed from 12/12 to 9/12. This trace records co-occurring functional regression and increased checker success; it does not identify constraint satisfaction as the cause of the test regression. Appendix A.11 presents the paired outcomes and the minimal code change.

Figure 6b summarizes checker-status histories for the same fixed constraint across repeated observations. Oscillation and degradation are sequence categories defined by observed pass/fail transitions. The plot excludes all-failure histories and normalizes within each model. It neither shows that satisfying one constraint caused another to fail nor identifies a memory or state-maintenance mechanism.

We use *instruction forgetting* as behavioral shorthand for an observed pass-to-fail transition. Within one fixed-set session, the same constraint passes at one constrained turn and fails at its next observed check. The two checks need not occur in adjacent turns because the active feedback subset may omit that constraint between observations. This label describes checker outcomes rather than model memory, training, or another internal mechanism.



(a) Hidden-test pass rates at Turn 0 and Turn 5, computed over outputs available at each turn. Because not all sessions reach Turn 5, the bars use different cohorts.



(b) Per-model distribution of checker-status histories for the same fixed constraint observed at least twice in one session. Percentages are normalized within each model after excluding histories that failed at every observation (3,047 of 7,796 histories; 4,749 remain). Oscillation contains both pass-to-fail and fail-to-pass transitions; degradation contains at least one pass-to-fail transition and no fail-to-pass transition.

Figure 6: Aggregate hidden-test outcomes and checker-status patterns across iterative repair.

4.3 Descriptive Relationship Between Checker and Test Outcomes

STRIDE records checker and hidden-test outcomes for each repair output. The two measures sometimes move in opposite directions within a trace, as in the Gemma example above. We treat this as a co-occurring outcome pattern rather than evidence that constraint satisfaction caused the functional change.

Figure 5 reports checker success across two constraint groups. Average checker success is lower for the reasoning-heavy group in each plotted model. The grouping describes observed checker outcomes and does not measure a model’s internal reasoning process.

Figure 4 and Figure 5 group the evaluated constraint types along different axes. The first separates avoidance-style from prescriptive requirements; the second uses the paper’s surface-level and reasoning-heavy taxonomy. In both views, checker success is lower for the prescriptive or reasoning-heavy group. Read together, the figures locate requirement groups that models satisfy less consistently during repair. They do not identify edit invasiveness, memory, or another internal mechanism as the source of the gap.

Together, the paired trace and aggregate checker plots show that checker success alone does not characterize executable correctness during fixed-set repair. STRIDE reports both outcomes so that later repair behavior can be assessed without treating either measure as a proxy for the other.

5 Conclusion

We introduced STRIDE, a controlled diagnostic for feedback-driven code repair under a fixed set of verifiable constraints. Within this evaluated snapshot, single-turn coding performance is informative but insufficient for characterizing behavior under fixed-set iterative repair: models are more robust to avoidance-style constraints than to prescriptive edits, and some later repair outputs pass fewer hidden tests than earlier outputs. In raw-verified cases, this decline can coincide with increased success on the same checked repair subset; the co-occurrence does not establish causality. STRIDE’s diagnostic value comes from separating baseline generation, checked repair compliance, and executable correctness. Turn-0 Pass@1 describes the starting output; IF and ISN summarize active checker outcomes during repair; and Test Case Pass Rate summarizes hidden-test outcomes at the selected runtime record. Reporting these measurements separately makes divergent trajectories visible without forcing them into a single definition of success. This separation supports analysis within the evaluated protocol while leaving causal explanations and claims about unrestricted software engineering outside the evidence. The present evidence does not establish reliability in unrestricted, tool-augmented software-development workflows.

6 Limitations

Although derived from authentic user interaction logs, the data samples used for this work are still synthesized for better controllability and modulation that specifically targets at multi-turn instruction following without bringing in additional complexity from more realistic agentic usage. As a result, STRIDE may not fully capture the open-ended, ambiguous, or tool-augmented interactions that characterize real-world collaborative coding sessions.

Additionally, our evaluation is restricted to Python, C++, and Java, and the verifiable constraint taxonomy, while derived from ShareGPT, may not cover the full diversity of user intents encountered in production environments. The five-turn budget, though empirically motivated, also limits observation of longer-horizon degradation patterns.

Finally, while STRIDE uses a deterministic simulator to ensure reproducibility, real users often provide richer, more context-sensitive feedback. The gap between rule-based checking and human judgment may cause some failure modes to be under- or over-represented in our analysis.

Acknowledgments

This work was supported by the Beijing Natural Science Foundation (Grant No. L253020).

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, H. Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, et al. Program synthesis with large language models. *arXiv.org*, 2021.

- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv.org*, pp. arXiv-2107, 2021.
- Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL <https://lmsys.org/blog/2023-03-30-vicuna/>.
- Dekun Dai, Mingwei Liu, Anji Li, Jialun Cao, Yanlin Wang, Chong Wang, Xin Peng, and Zibin Zheng. Feedbackeval: A benchmark for evaluating large language models in feedback-driven code repair tasks, 2026. URL <https://arxiv.org/abs/2504.06939>.
- Guoliang Duan, Mingwei Liu, Yanlin Wang, Chong Wang, Xin Peng, and Zibin Zheng. A hierarchical and evolvable benchmark for fine-grained code instruction following with multi-turn feedback, 2025. URL <https://arxiv.org/abs/2507.00699>.
- Hojae Han, Seung won Hwang, Rajhans Samdani, and Yuxiong He. Convcodeworld: Benchmarking conversational code generation in reproducible feedback environments, 2025. URL <https://arxiv.org/abs/2502.19852>.
- Yun He, Di Jin, Chaoqi Wang, Chloe Bi, Karishma Mandyam, Hejia Zhang, Chen Zhu, Ning Li, Tengyu Xu, Hongjiang Lv, et al. Multi-if: Benchmarking llms on multi-turn and multilingual instructions following. *Proceedings of ArabicNLP 2023*, 2024.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, 2024. doi: 10.48550/arXiv.2403.07974.
- Yuxin Jiang, Yufei Wang, Xingshan Zeng, Wanjuan Zhong, Liangyou Li, Fei Mi, Lifeng Shang, Xin Jiang, Qun Liu, and Wei Wang. Followbench: A multi-level fine-grained constraints following benchmark for large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4667–4688. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.257.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, 2024. doi: 10.48550/arXiv.2310.06770. URL <https://openreview.net/forum?id=VTF8yNQm66>.
- Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. Llms get lost in multi-turn conversation, 2025. URL <https://arxiv.org/abs/2505.06120>.
- Junlong Li, Wenshuo Zhao, Jian Zhao, Weihao Zeng, Haoze Wu, Xiaochen Wang, Rui Ge, Yuxuan Cao, Yuzhen Huang, Wei Liu, et al. The tool decathlon: Benchmarking language agents for diverse, realistic, and long-horizon task execution. In *arXiv.org*, 2026. doi: 10.48550/arXiv.2510.25726. URL <https://openreview.net/forum?id=z53s5p0qhf>.
- Mike A. Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, I. Bercovich, Lin Shi, J. Shin, Thomas Walshe, E. K. Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *arXiv.org*, 2026. doi: 10.48550/arXiv.2601.11868. URL <https://openreview.net/forum?id=a7Qa4CcHak>.
- Peiding Wang, Li Zhang, Fang Liu, Lin Shi, Minxiao Li, Bo Shen, and An Fu. Codeif-bench: Evaluating instruction-following capabilities of large language models in interactive code generation, 2025. URL <https://arxiv.org/abs/2503.22688>.
- Sizhe Wang, Zhengren Wang, Dongsheng Ma, Yongan Yu, Rui Ling, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. Codeflowbench: A multi-turn, iterative benchmark for complex code generation, 2026. URL <https://arxiv.org/abs/2504.21751>.

- Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *International Conference on Learning Representations*, 2023. doi: 10.48550/arXiv.2309.10691.
- Kaiwen Yan, Hongcheng Guo, Xuanqing Shi, Jingyi Xu, Yaonan Gu, and Zhoujun Li. Codeif: Benchmarking the instruction-following capabilities of large language models for code generation. *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pp. 1272–1286, 2025. doi: 10.18653/v1/2025.acl-industry.89.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, E. Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Neural Information Processing Systems*, 36:46595–46623, 2023. doi: 10.52202/075280-2020.
- Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models. *arXiv.org*, 2023. doi: 10.48550/arXiv.2311.07911.

A Appendix

A.1 ShareGPT data analysis

Initial analysis revealed a long-tail distribution of conversation turns, with outliers exceeding 1,000 turns due to users switching topics without starting new conversations. To characterize this tail, we analyzed user-model interaction rounds in the ShareGPT coding data and excluded interactions exceeding 10 rounds before the sampling analysis below.

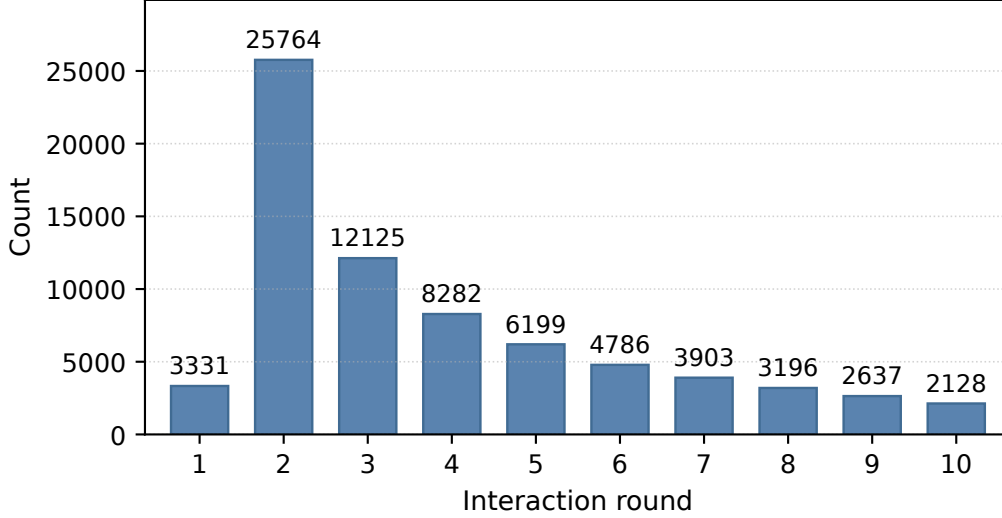


Figure 7: Interaction round distribution.

Figure 7 summarizes the observed interaction-round distribution. After excluding dialogues longer than ten turns, we sampled 500 dialogues with weights based on the remaining turn-count distribution. Two annotators labeled each dialogue independently without seeing each other’s labels. They recorded topic drift, whether a dialogue contained rule-verifiable coding requests, the turn positions of those requests, and how many appeared in each dialogue. They resolved disagreements by consensus.

We used GPT-4o mini to support a second-stage screen and fine-grained categorization of ShareGPT coding requests, and we removed requests classified as irrelevant. The authors reviewed category-stratified samples of the resulting classifications, corrected affected labels, and merged or removed categories. They resolved review disagreements by consensus. Figure 8 shows the resulting category distribution.

In this annotated sample, more than half of the dialogues longer than five turns exhibited topic drift. Of the 500 dialogues, 312 (62.4%) contained at least one rule-verifiable coding request, and 83% of those requests appeared in turns 2–4. Dialogues with exactly three turns contained 1.8 such requests on average. These observations informed STRIDE’s controlled five-turn budget. Our study did not test whether this five-turn budget was universally sufficient or optimal.

A.2 Code Execution Results

We also recorded whether each evaluated output passed its complete hidden-test suite at every STRIDE Turn; Table 3 summarizes these records.

Table 4 extends the eight-model main-text comparison with nine additional model rows.

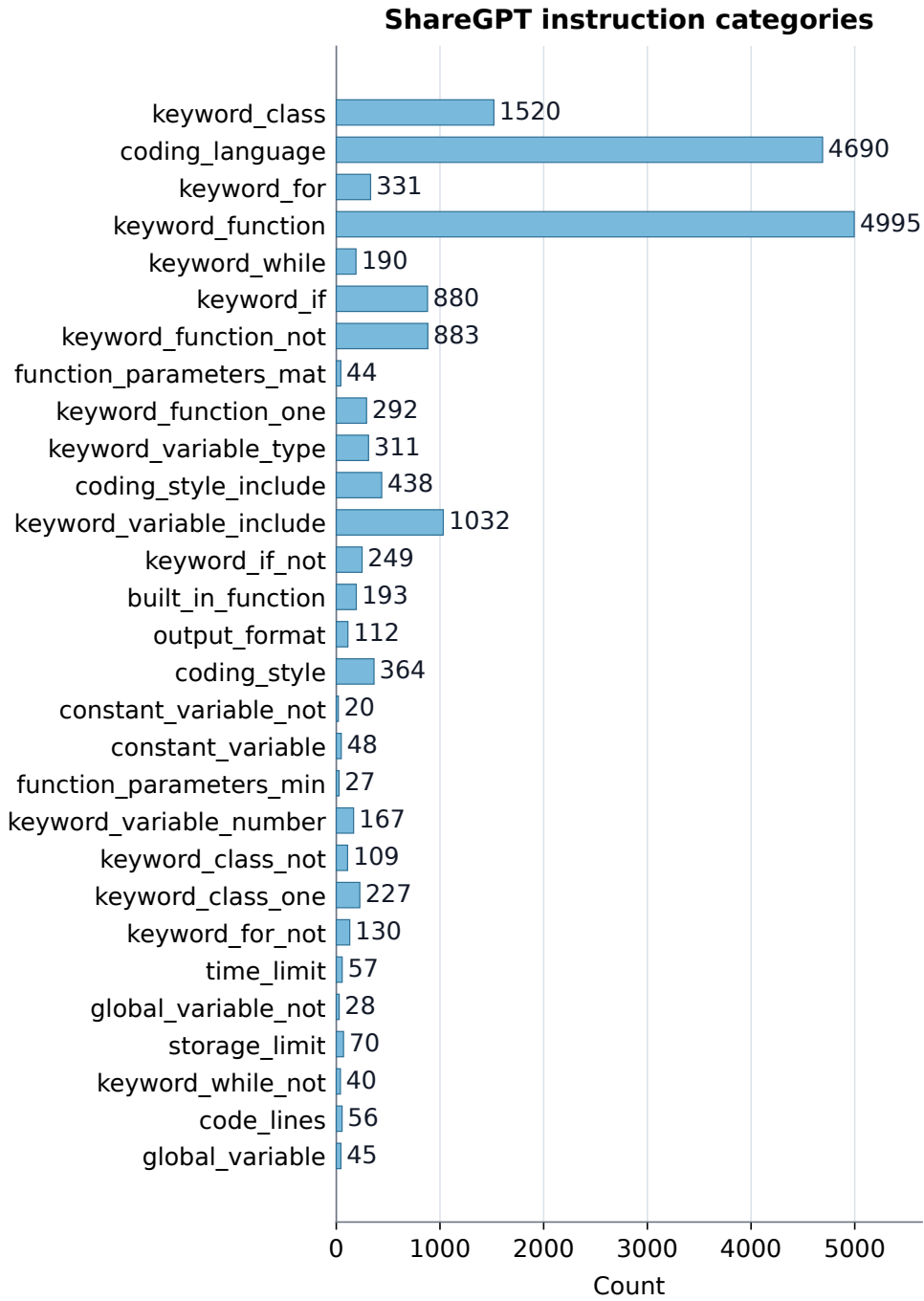


Figure 8: Distribution of different instruction types on ShareGPT.

A.3 Instruction Difficulty Analysis

Completion rates alone do not characterize instruction difficulty. Figure 9 reports the average number of recorded repair Turns used for the displayed instruction types. We selected these types from those with observed feedback-reactivation forgetting events; they do not represent the full constraint inventory.

Table 2: Categories of coding instructions with examples.

Category Group	Category	Example
Variables & Data Types	keyword_variable_include	Modify code to use var87 as a variable name.
	keyword_variable_number	Revise code to position tab18 as the 4th variable.
	keyword_variable_type	Update code to define the 3rd variable as float.
	constant_variable	Update code to include a constant variable.
	constant_variable_not	Refactor code to remove constant variables.
	global_variable global_variable_not	Revise code to use a global variable. Modify code to avoid global variables.
Control Structures	keyword_for	Refactor code to include a for loop.
	keyword_for_not	Update code to exclude for loops.
	keyword_while	Modify code to include at least one while loop.
	keyword_while_not keyword_if keyword_if_not	Adjust code to omit while loops. Revise code to include an if statement. Modify code to avoid if statements.
Functions & Classes	keyword_function	Update code to include 3 functions.
	keyword_function_not	Revise code to exclude functions.
	keyword_function_one	Adjust your code to include function.
	keyword_class	Modify code to include 6 class(es).
	keyword_class_not	Refactor code to remove classes.
	keyword_class_one function_parameters_max function_parameters_min	Refactoring your code to integrate class. Limit function parameters to 3. Ensure no function has fewer than 2 parameters.
Programming Lang & Built-ins	built_in_function	Restrict function usage to built-in functions.
	coding_language	Revise code to be written in Java.
Code Style & Standards	coding_style	Adjust code to exclude comments.
	coding_style_include code_lines	Refactor code to include comments. Revise code to have at most 60 lines.
Performance & Output	time_limit	Modify code to run within 2048 ms.
	storage_limit	Optimize code to use less than 10240 KB.
	output_format	Adjust code to output in { Output } format.

Table 3: Per-turn full-test-suite success rate among evaluated records without a saved execution error (%). Turn 0 is unconstrained; Turns 1–5 are constrained repairs.

Model	Turn 0	Turn 1	Turn 2	Turn 3	Turn 4	Turn 5
GPT-4.1-mini	55.05%	43.28%	38.88%	37.27%	37.33%	37.85%
GPT-4o mini	28.19%	22.16%	20.69%	20.34%	20.82%	21.56%
Claude Sonnet 4.6	71.43%	64.64%	66.51%	65.58%	65.68%	65.96%
Claude-Sonnet-4	60.17%	54.25%	52.58%	52.19%	55.14%	51.83%
Claude-Sonnet-3.7	56.94%	57.71%	55.12%	53.47%	54.40%	55.66%
DeepSeek-v3.1	55.02%	49.17%	45.17%	43.84%	43.39%	43.27%
DeepSeek-v3	51.58%	44.51%	41.08%	38.85%	39.01%	39.45%
Gemini-2.5-Flash	54.74%	48.62%	48.78%	47.36%	47.95%	48.62%
Gemma 3 27B IT	37.14%	30.22%	27.89%	27.25%	26.22%	26.76%
GPT-4.1	48.93%	49.87%	48.99%	48.02%	47.63%	47.89%
Llama 3.3 70B Instruct	28.78%	22.72%	19.64%	17.92%	19.32%	19.15%
GPT-5.4-mini	57.96%	38.49%	33.07%	31.08%	30.55%	30.08%
GPT-5.4	76.84%	64.29%	64.54%	62.58%	63.26%	61.19%
Qwen2.5-32B	30.36%	25.35%	21.75%	22.00%	20.93%	21.17%
Qwen2.5-Coder-32B	36.71%	27.78%	25.83%	24.76%	25.68%	26.08%

Model	ISN	Avg. max			
		Turn	IF Pass Rate	Test Case Pass Rate	Pass@1
GPT-5.4	358 (40.68%)	3.11	64.84%	65.89%	71.25%
Gemini 3.1 Flash					
Lite Preview	295 (33.52%) [†]	3.35	59.27%	73.50%	70.97%
Claude Sonnet 4.6	385 (43.75%)	3.63	66.83%	68.67%	69.32%
DeepSeek-V3.2	317 (36.02%) [†]	3.38	56.48%	60.86%	56.74%
GPT-5.4-mini	322 (36.59%)	3.61	57.49%	44.83%	56.25%
DeepSeek-v3.1	267 (30.34%)	3.41	52.67%	51.79%	50.45%
GPT-4.1-mini	290 (32.95%)	3.72	55.62%	49.37%	51.36%
Qwen3-Coder					
30B-A3B-Instruct	274 (31.13%) [†]	3.77	46.86%	55.24%	46.53%
Claude-Sonnet-4	355 (40.34%)	2.82	67.60%	63.89%	48.07%
DeepSeek-v3	287 (32.61%)	3.14	57.27%	48.09%	44.55%
GPT-4.1	305 (34.66%)	3.29	59.44%	57.09%	43.98%
Gemini-2.5-Flash	229 (26.02%)	2.91	54.47%	52.18%	43.30%
Qwen2.5-32B	245 (27.84%)	3.21	55.66%	33.19%	25.91%
Qwen2.5-Coder-32B	201 (22.84%)	2.71	54.51%	31.45%	25.91%
GPT-4o mini	248 (28.18%)	3.38	53.87%	29.30%	24.09%
Llama 3.3 70B					
Instruct	292 (33.18%)	2.82	60.93%	24.83%	22.27%
Gemma 3 27B IT	170 (19.32%)	2.26	55.27%	36.08%	22.16%

Table 4: Extended STRIDE point estimates. We reproduce the point estimates from Table 1 for its eight models and report the available base-output aggregates for nine additional models. [†] We have 772, 775, and 821 saved evaluator records for DeepSeek-V3.2, Gemini 3.1 Flash Lite Preview, and Qwen3-Coder 30B-A3B-Instruct, respectively. The marked ISN entries reproduce Table 1 and do not represent reconstructions from complete 880-session raw outputs. **Bold** marks the largest observed point estimate for ISN, IF Pass Rate, Test Case Pass Rate, and Turn-0 Pass@1 in this table and does not imply statistical significance. Average maximum recorded Turn remains unranked because early completion and incomplete records can both lower it.

(a) Session-level binary metrics				
Model	Cohort	ISN rate 95% interval	Turn-0 Pass@1 95% interval	
Gemma 3 27B IT	complete N = 880	19.32% [16.85, 22.06]	22.16% [19.54, 25.02]	
Llama 3.3 70B	complete N = 880	33.18% [30.15, 36.36]	22.27% [19.65, 25.14]	
Instruct	complete N = 880	28.18% [25.31, 31.24]	24.09% [21.38, 27.03]	
GPT-4o mini	complete N = 880	31.13% [N/A]	46.53% [43.14, 49.95]	
Qwen3-Coder	available N = 821	36.02% [N/A]	56.74% [53.22, 60.19]	
30B-A3B-Instruct	available N = 772	43.75% [40.51, 47.05]	69.32% [66.19, 72.28]	
DeepSeek-V3.2	complete N = 880	33.52% [N/A]	70.97% [67.67, 74.05]	
Claude Sonnet 4.6	complete N = 775	40.68% [37.48, 43.96]	71.25% [68.17, 74.14]	
Gemini 3.1 Flash	complete N = 880			
Lite Preview				
GPT-5.4				
(b) Turn and micro-averaged metrics				
Model	Cohort	Avg. max Turn 95% interval	IF Pass Rate 95% interval	Test Case Pass Rate 95% interval
Gemma 3 27B IT	complete N = 880	2.26 [2.11, 2.41]	55.27% [53.09, 57.52]	36.08% [31.79, 40.49]
Llama 3.3 70B	complete N = 880	2.82 [2.68, 2.96]	60.93% [58.99, 62.87]	24.83% [21.43, 28.23]
Instruct	complete N = 880	3.38 [3.24, 3.51]	53.87% [52.13, 55.66]	29.30% [26.04, 32.56]
GPT-4o mini	complete N = 880	3.77 [3.65, 3.89]	46.86% [44.96, 48.78]	55.24% [51.26, 59.27]
Qwen3-Coder	available N = 821	3.38 [3.24, 3.52]	56.48% [54.44, 58.54]	60.86% [56.79, 64.88]
30B-A3B-Instruct	available N = 772	3.63 [3.51, 3.75]	66.83% [65.28, 68.42]	68.67% [65.02, 72.23]
DeepSeek-V3.2	complete N = 880	3.35 [3.21, 3.49]	59.27% [57.23, 61.29]	73.50% [69.94, 76.94]
Claude Sonnet 4.6	complete N = 775	3.11 [2.98, 3.23]	64.84% [63.19, 66.52]	65.89% [62.18, 69.60]
Gemini 3.1 Flash	complete N = 880			
Lite Preview				
GPT-5.4				

Table 5: Two-sided 95% intervals for the eight main-text models. Each eligible entry gives the point estimate followed by its interval in brackets; average maximum recorded Turn uses Turn units, and the other intervals use percentage points. “Complete” denotes the full 880-session cohort. “Available” denotes a partial saved-record cohort, so those intervals do not cover missing-record bias. The frozen reported ISN values for Qwen3-Coder 30B-A3B-Instruct, DeepSeek-V3.2, and Gemini 3.1 Flash Lite Preview lack matching 880-session binary records; we mark their ISN intervals as N/A. These marginal intervals do not provide pairwise significance tests.

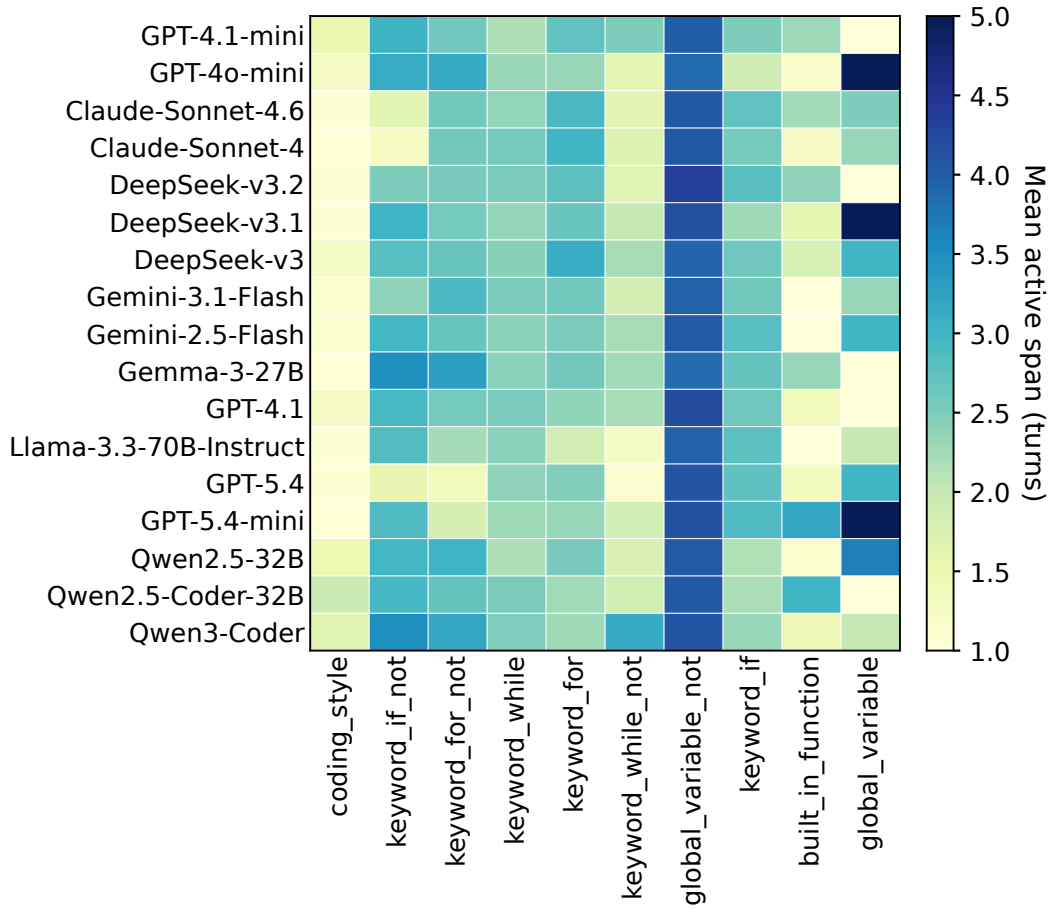


Figure 9: Turns required to resolve different instructions.

We also analyzed the token distribution of the total context length in the final-round input across different models to support our argument in the text. The distribution is shown in Figure 10 and Figure 11. It can be observed that some smaller models, such as Llama and Qwen, tend to generate longer outputs, which also reflects their efficiency in solving problems.

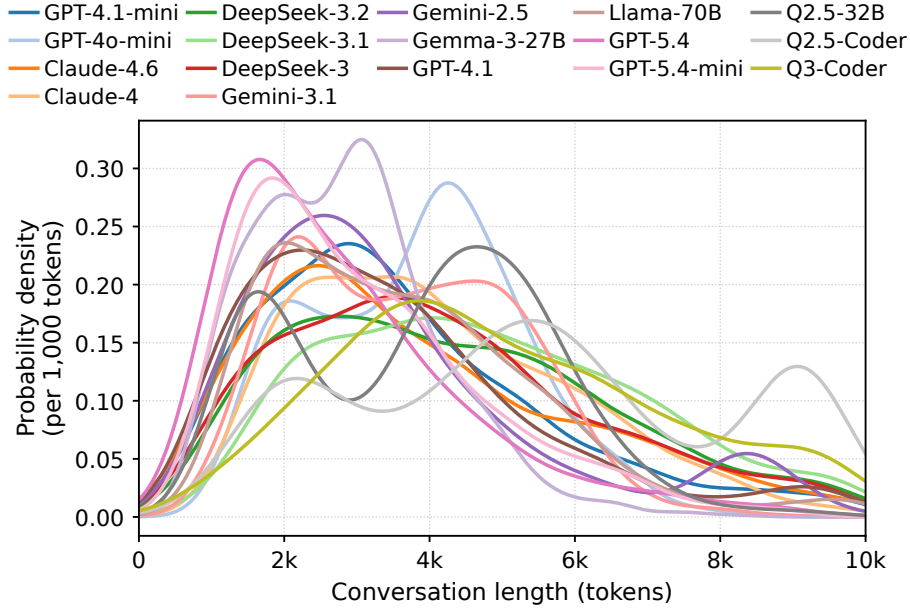


Figure 10: Final round tokens distribution (0-10k).

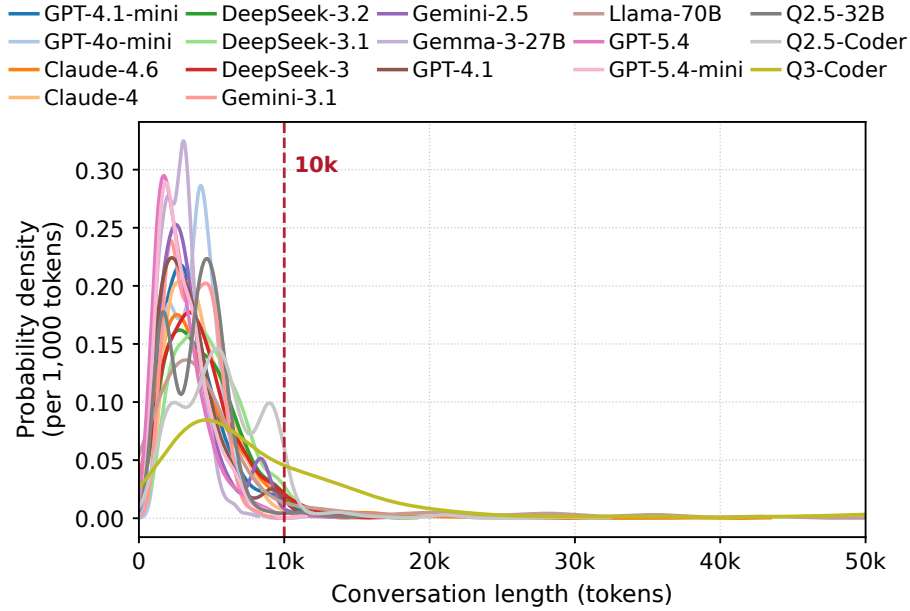


Figure 11: Final round tokens distribution.

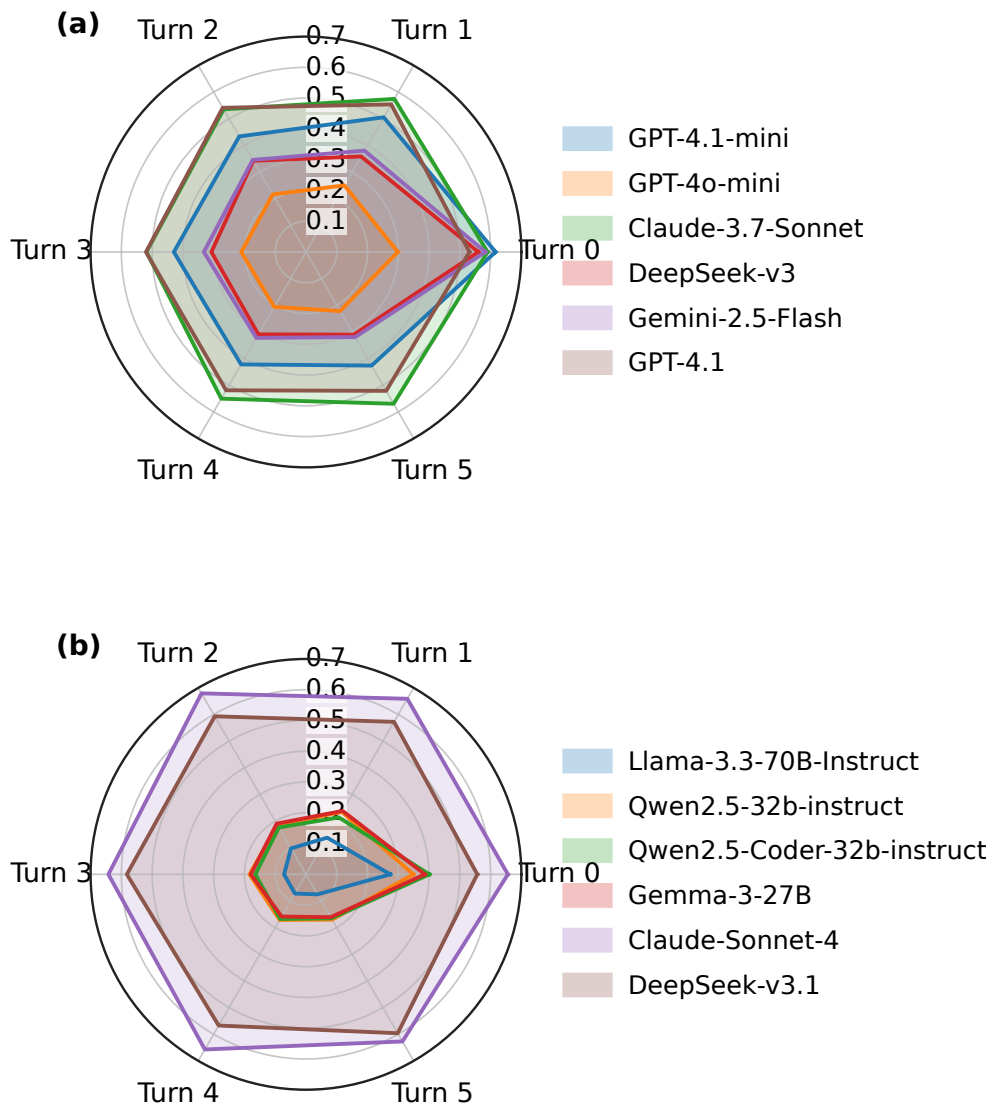


Figure 12: Per-turn full-test-suite success rates for the displayed models.

A.4 Executable performance across repair turns

Figure 12 reports the saved turn-level execution metric for several models. The shapes vary across models and turns. Each plotted turn summarizes the outputs available at that turn, so the figure is descriptive rather than a paired estimate of within-session change.

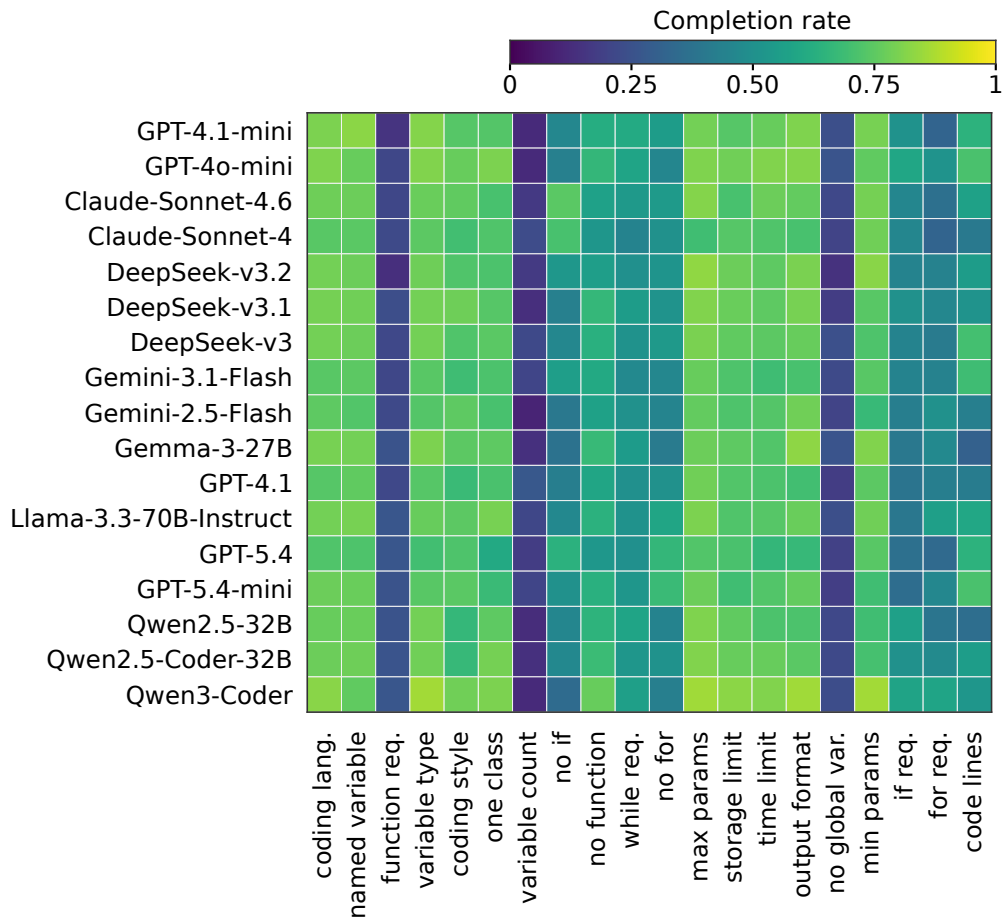


Figure 13: Constraint-type resolution proxy based on active-feedback membership. For each type present at Turn 1, the plot records whether it is absent from the last nonempty active feedback subset. This statistic differs from IF Pass Rate, which micro-averages checker outcomes across constrained Turns.

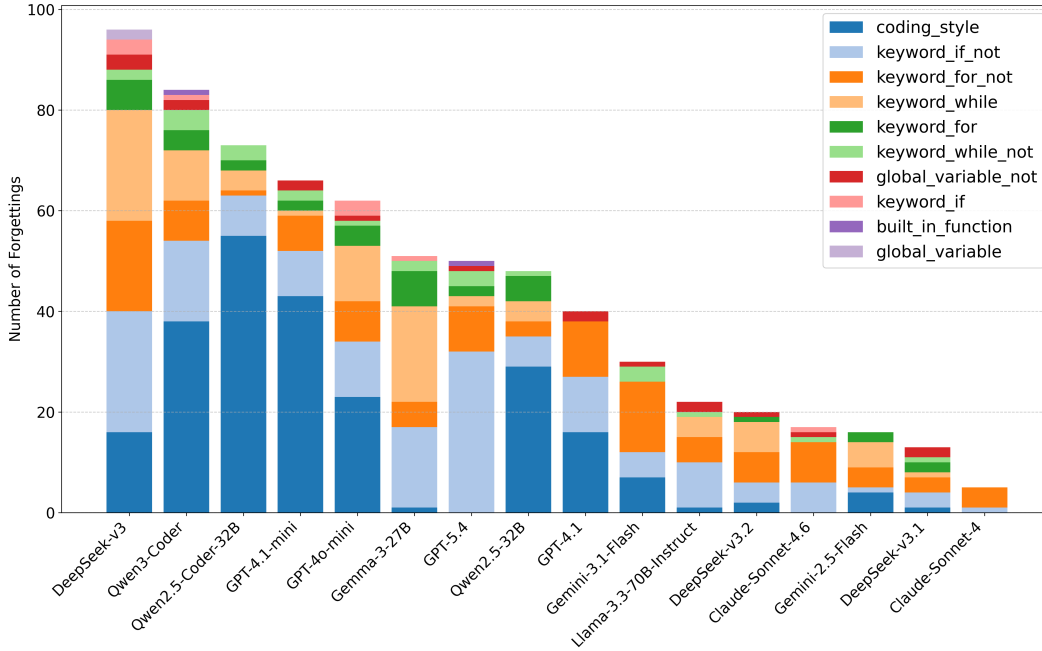


Figure 14: Raw counts of feedback-reactivation forgetting events for the ten constraint types with the largest pooled counts across models. For these constraint types, the selector recomputes active status after each output. We count an event when the same fixed constraint type appeared earlier, was absent from the immediately preceding active feedback subset, and then reappeared, indicating a satisfied-to-unsatisfied reversal. Each model contributes 880 sessions; the counts are not normalized by constraint exposure or eligible reactivation opportunities.

A.5 Analysis on instruction difficulty

Figure 13 compares this final-feedback-subset disappearance proxy across constraint types and models. It describes end-of-session active-feedback status rather than the per-Turn checker micro-average used for IF Pass Rate. Simple instructions, such as specifying a programming language, have high completion rates, while complex ones, like specifying the number of functions or the name of the n -th variable, have low completion rates. This disparity likely arises because models struggle with precise counting or indexing, such as identifying the “third variable,” which requires robust internal reasoning. Additionally, instructions like specifying function counts demand dynamic state tracking, which is challenging in long code generation. Assigning specific variable names, such as requiring the “second variable to be var_88,” further complicates matters by necessitating precise mapping and conflict avoidance.

These challenges highlight deficiencies in precise logical control, state tracking and numerical reasoning. Take DeepSeek-v3 as an example, the input length distribution for the final round averages 4,380 tokens, with extremes reaching 17,500+ tokens. This places the instruction set in the long-context regime, contributing to its difficulty. For detailed information, please refer to the Appendix A.3 for other models’ response length distribution.

A.6 Feedback-Reactivation Forgetting

For the ten constraint types shown in Figure 14, the generation-side selector recomputes active status after each output. A constraint type leaves the active feedback when its corresponding check passes and reappears if a later output fails that check. We call this present-absent-present sequence within one session a *feedback-reactivation forgetting* event. Since A_{t+1} is derived from S_t , the reactivation appears in the user message following the output that reintroduced the violation.

Figure 14 reports raw event counts for the ten constraint types with the largest pooled counts. Each model contributes 880 sessions, but constraint exposure and the number of eligible reactivation opportunities vary; one history may also contribute more than one event. We use the plot as a descriptive category breakdown and do not infer model rankings, memory mechanisms, training-data effects, or fine-tuning strategies from its bar heights.

Figure 6b uses explicit turn-level checker outcomes instead of feedback membership. It groups repeated observations of the same fixed constraint into checker-status histories. The two figures use different units, filters, and denominators and are not pooled.

Model	Histories	All-fail	Plotted
Claude Sonnet 4.6	1,106	310	796
DeepSeek-V3.2	937	365	572
Gemini 3.1 Flash Lite Preview	845	342	503
Gemma 3 27B IT	700	313	387
GPT-4o mini	1,087	506	581
GPT-5.4	866	278	588
Llama 3.3 70B Instruct	842	301	541
Qwen3-Coder 30B-A3B-Instruct	1,413	632	781
Total	7,796	3,047	4,749

Table 6: Denominators for Figure 6b. Each history contains at least two observations of the same fixed constraint. The plotted denominator excludes histories that failed at every observation.

A.7 Details of Experiment Setting

Here, we describe the construction-side filtering rules and the evaluator-side checkers. STRIDE retains constraints for which the implementation provides a rule-based or syntax-aware checker. Checker coverage establishes whether the evaluator can test the stated code property; feasibility is addressed by the construction procedure rather than by checker availability alone.

For each problem, we draw a target set size uniformly from $\{4, 6, 8\}$. We then sample constraint types without replacement from candidates that pass construction-reference and conflict filtering, using fixed type weights. We built this benchmark inventory from LiveCodeBench `code_generation_lite`, release `release_v5`. Researchers can instantiate the construction and evaluation stages from a later compatible release that provides the expected input fields.

Filtering with the construction-time reference. Consider a construction reference that contains no loop. A candidate that asks the model to add a for loop can remain in the pool because satisfying it requires a code change. If the reference already contains a for loop, the same candidate is removed as already satisfied. The conflict table rejects a pair that both requires and forbids a for loop. These examples illustrate the filters; the table covers predefined direct conflicts rather than an exhaustive set of possible incompatibilities.

The generation loop stops when the selector produces no valid cases for the next turn, when the five-turn repair budget is reached, or when a prompt or API error interrupts the session.

Constraint checking combines regular expressions and Abstract Syntax Tree (AST) parsing. For Python code, we use the `ast` module; for C++ and Java, we use `clang` and `javalang`, respectively, ensuring accurate structural analysis. Taking `keyword_if` as an example, we first categorize the code generated by the model into three types: Python, Java, and C++. After classification, we use regular expressions to remove comments and string literals. Then, within the remaining code, we apply regex checks again to verify the presence of if statements.

To evaluate code functionality, we execute generated code with a 5-second timeout and 64MB memory limit in isolated subprocesses, capturing output, errors, execution time,

Table 7: Statistics for the reported STRIDE benchmark-input snapshot. The category distribution counts initial constraint-type assignments ($N = 5,256$); percentages are rounded independently.

Snapshot field	Value	Coarse constraint group	Assignments (share)
Upstream data	LiveCodeBench	Variables & Data Types	1,366 (25.99%)
Upstream release	code_generation_lite release_v5	Control Structures	867 (16.50%)
Problem inventory	880	Functions & Classes	1,404 (26.71%)
Target set-size draw	Uniform over 4, 6, and 8	Programming Language & Built-ins	712 (13.55%)
Realized tasks by target size	294 / 304 / 282 for 4 / 6 / 8	Code Style & Standards	411 (7.82%)
Constraint taxonomy	26 types represented in 6 coarse groups	Performance & Output	496 (9.44%)
Languages	Python, C++, and Java		
Interaction budget	Turn 0 plus up to 5 constrained repair turns		
Verification	Rule and syntax checks plus hidden-test execution		

Table 8: Model access identifiers or repositories, execution routes, and dates of retained generation artifacts for the eight main-text models. We omit provider-side snapshot identifiers and local checkpoint revisions because the retained records do not contain them.

Display name	Model ID / repository	Execution route	Artifact date
Gemma 3 27B IT	google/gemma-3-27b-it	local vLLM, NVIDIA A800	2025-05-06
Llama 3.3 70B Instruct	meta-llama/Llama-3.3-70B-Instruct	local vLLM, NVIDIA A800	2025-05-06
GPT-4o mini	openai/gpt-4o-mini	OpenRouter	2025-04-29
Qwen3-Coder	Qwen/	local vLLM, NVIDIA A800	2026-03-30
30B-A3B-Instruct	Qwen3-Coder-30B-A3B-Instruct	A800	
DeepSeek-V3.2	deepseek-ai/DeepSeek-V3.2	local vLLM, NVIDIA A800	2026-03-30
Claude Sonnet 4.6	anthropic/claude-sonnet-4.6	OpenRouter	2026-03-18
Gemini 3.1 Flash Lite	google/gemini-3.1-flash-lite-preview	OpenRouter	2026-03-30
Preview			
GPT-5.4	openai/gpt-5.4	OpenRouter	2026-03-18

memory usage, and test case pass rates. To ensure efficient evaluation, we set a 90-second total time limit per code.

For multi-language constraints (e.g., coding language: Java/C++), we compile and execute submissions using OpenJDK 17 and GCC 12.2 respectively, with identical time (5s) and memory (64MB) limits. Test cases are language-agnostic I/O pairs, so the same hidden test suite applies regardless of implementation language. We verified parity by cross-compiling a random sample of 50 problems and confirming identical pass/fail outcomes across Python, Java, and C++ reference solutions.

We use *Artifact date* for the date of the retained generation artifact rather than a model-release or checkpoint date. We evaluated the generated code from March 24 to March 31, 2026, and report the results as a dated experimental snapshot. The retained records for DeepSeek-V3.2, Gemini 3.1 Flash Lite Preview, and Qwen3-Coder 30B-A3B-Instruct contain an evaluator start time but no recoverable completion timestamp.

Checker audit. Two authors independently annotated 200 randomly sampled (solution, constraint) pairs drawn from four structurally challenging categories: `keyword_if_not`, `keyword_for`, `keyword_function`, and `coding_style`. Cohen’s $\kappa = .89$ measured inter-annotator agreement. Comparing checker outputs with the manual audit labels yielded category-averaged precision of .962 and category-averaged recall of .937. The audit estimated that checker error changes the reported IF Pass Rate by less than 2% under its reporting convention. These values serve as aggregate validation estimates for the rule-based checker. Individual constraint decisions can still differ from human judgments.

A.8 Token Count in Experiment

For reproducibility, Table 9 reports the total token counts recorded for all models in our experiments.

Table 9: Token count summary. Each Turn 0–5 cell aggregates 880 saved sessions.

Model	Turn 0	Turn 1	Turn 2	Turn 3	Turn 4	Turn 5	Total Tokens
GPT-4.1-mini	723,360	1,378,080	1,921,920	2,438,480	2,947,120	3,449,600	9,995,997
GPT-4o mini	723,360	1,298,000	1,843,600	2,378,640	2,918,080	3,470,720	9,754,275
Claude Sonnet 4.6	723,360	1,664,960	2,403,280	2,967,360	3,546,400	4,167,680	11,191,053
Claude-Sonnet-4	723,360	1,636,800	2,241,360	2,841,520	3,422,320	4,007,520	10,381,103
Claude-3.7-Sonnet	723,360	1,723,920	2,392,720	3,061,520	3,712,720	4,370,960	11,072,009
DeepSeek-v3.1	723,360	2,225,520	2,838,880	3,443,440	4,035,680	4,620,000	13,119,522
DeepSeek-v3	723,360	1,625,360	2,266,000	2,901,360	3,509,440	4,133,360	10,138,178
Gemini-2.5-Flash	723,360	1,239,920	1,652,640	2,094,400	2,528,240	2,970,000	7,765,452
Gemma 3 27B IT	723,360	999,680	1,382,480	1,796,080	2,236,960	2,698,080	6,395,458
GPT-4.1	723,360	1,191,520	1,687,840	2,166,560	2,656,720	3,151,280	8,289,061
Llama 3.3 70B Instruct	723,360	1,400,080	2,052,160	2,891,680	4,011,920	5,441,040	11,572,044
GPT-5.4	723,360	1,139,600	1,606,000	2,079,440	2,537,040	2,996,400	7,050,508
Qwen2.5-32B	723,360	1,116,720	1,599,840	2,134,880	2,798,400	3,617,680	8,926,231
Qwen2.5-Coder-32B	723,360	1,338,480	1,989,040	2,688,320	3,488,320	4,441,360	9,090,314

A.9 Instruction-set size and problem difficulty

STRIDE groups instruction sets by size (4, 6, or 8 instructions), while LiveCodeBench provides difficulty labels for the underlying problems. We examine code quality across instruction-set sizes and instruction completion across problem-difficulty groups. Figures 15, 16, and 17 summarize these descriptive comparisons.

Figure 15 and Figure 16 present code-quality measurements across instruction-set sizes of 4, 6, and 8 for each plotted model. We report these values as descriptive group differences; the current analysis does not provide a significance test or isolate instruction-set size from other task differences.

Figure 17 shows no clear descriptive association between problem difficulty and instruction-following effectiveness in the plotted data.

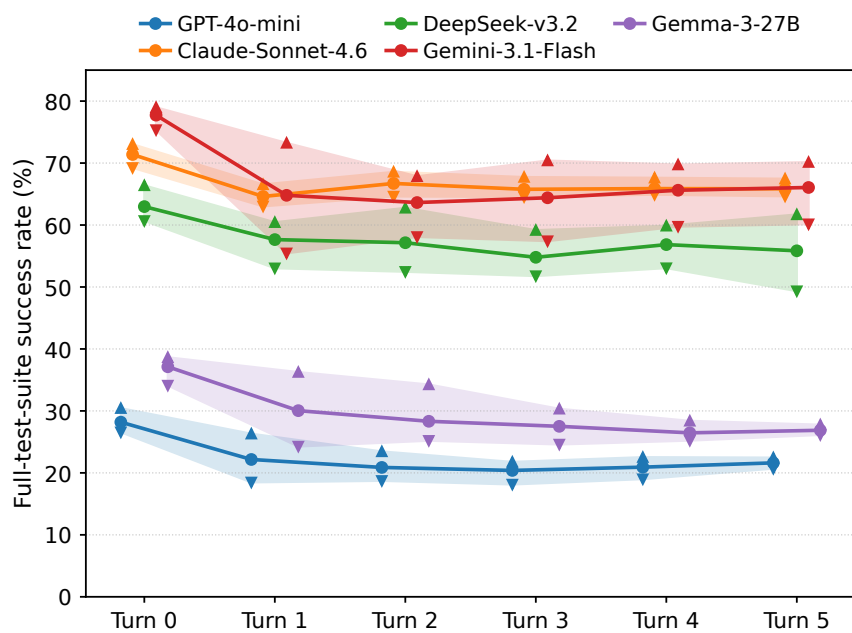


Figure 15: Per-turn full-test-suite success rates across instruction-set sizes for Group 1 models.

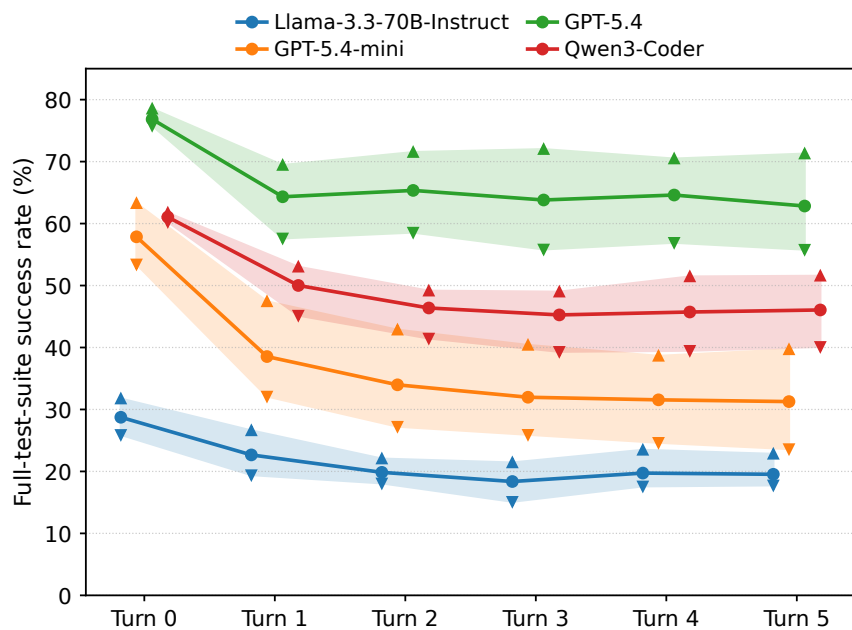


Figure 16: Per-turn full-test-suite success rates across instruction-set sizes for Group 2 models.

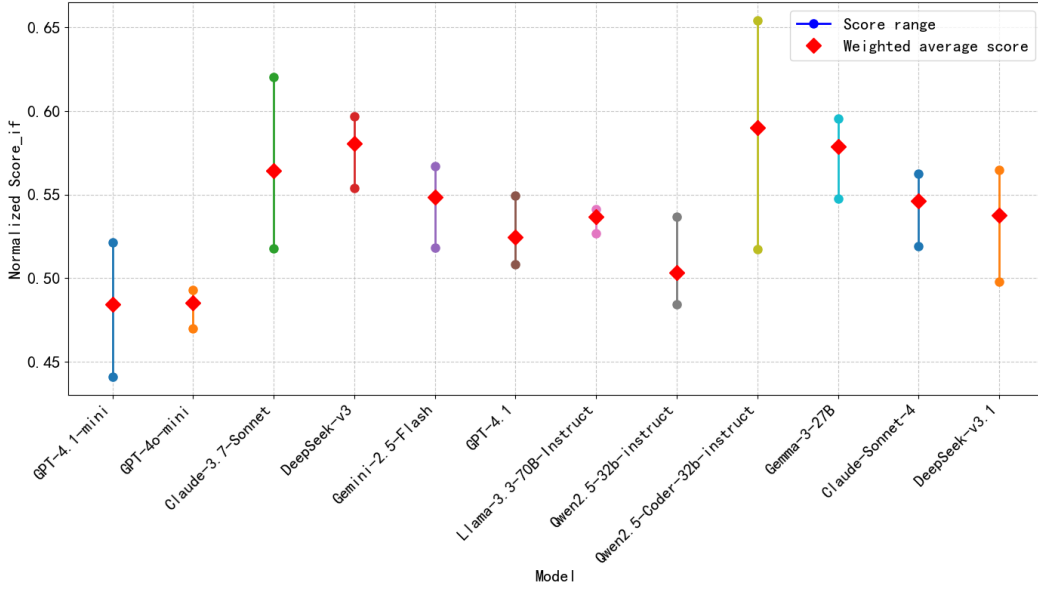


Figure 17: Standardized instruction-following score ranges and weighted averages across coding-problem difficulty groups for the plotted models.

A.10 Use of Language Models

We used GPT-4o mini to support a second-stage screen and fine-grained categorization of ShareGPT coding requests. During benchmark construction, we used GPT-4o mini to produce one construction-time reference solution for each of the 880 LiveCodeBench problems in STRIDE. Construction-time static checks used this reference to omit some structural constraint candidates that the code already satisfied. This reference was separate from each evaluated model’s Turn 0 output. GPT-4o mini was also evaluated as one of the eight main models on the 880-task cohort. Authors reviewed category-stratified samples of the resulting classifications, corrected affected labels, and merged or removed categories; they resolved review disagreements by consensus. Two annotators independently labeled the separate 500-dialogue sample without seeing each other’s labels and resolved annotation disagreements by consensus. For manuscript preparation, LLM assistance was limited to minor language polishing.

A.11 Case Study: Divergent Checker and Test Outcomes

We inspect the raw-verified Gemma 3 27B IT trace highlighted in Section 4.2. AtCoder task `abc302_a` (“Attack”) asks for the minimum number of attacks needed to reduce stamina A by B per attack, which is $\lceil A/B \rceil$, with $1 \leq A, B \leq 10^{18}$. At both Turns 2 and 3, the complete active repair subset consisted only of the requirement checked by `keyword_for`. Table 10 reports the paired outcomes.

Turn	Active repair subset	Checker	Hidden tests
2	<code>keyword_for</code>	Fail	12/12
3	<code>keyword_for</code>	Pass	9/12

Table 10: Paired outcomes from the archived Gemma 3 27B IT trace for task `abc302_a`.

The relevant code change is compact. Turn 2 computes the answer directly, whereas Turn 3 inserts a literal loop to satisfy the active structural requirement:

Minimal Turn-2 to Turn-3 code change

```
// Turn 2
count17 = (a + b - 1) / b;

// Turn 3
for (long long i = 0; a > 0; ++i) {
    a -= b;
    count17++;
}
```

This edit replaces constant-time ceiling division with a loop requiring $\lceil A/B \rceil$ iterations, which can be extremely large under the task bounds. The visible complexity change is consistent with the stored runtime regression, while the checker transition confirms that the requested structural property was added. This single trace illustrates why STRIDE reports checker compliance and executable correctness separately; it does not estimate the frequency of this pattern or establish a general causal relationship between constraint satisfaction and functional regression.

A.12 Code and Data Availability

We release the retained experiment code under Apache License 2.0 at <https://github.com/hzh030430/stride-codebase>. It includes constraint construction, an OpenAI-compatible multi-turn generation client, code extraction, Python/C++/Java evaluation, and final metric calculation. We release the STRIDE metadata overlay under CC BY 4.0 at <https://huggingface.co/datasets/Han03430/stride-metadata-overlay>. The overlay contains `task_index.jsonl` and `constraint_metadata.jsonl`. It references LiveCodeBench `code_generation_lite`, release `release_v5`, by task ID; users obtain the upstream problem content and tests from LiveCodeBench. The metadata contains no model outputs or results. It contains 879 records with complete initial parameters and one types-only record, task 3233, whose canonical parameter values are marked as missing.

A.13 Prompts in our Experiments

Here we provide examples of the prompt used to generate construction-time reference solutions and the prompts used during evaluation.

Construction-Time Reference Solution Prompt

Prompt:

System: You are an expert Python programmer. You will be given a question (problem specification) and will generate a correct Python program that matches the specification and passes all tests. Your code should not only handle the test cases but also any possible input. The test cases are merely provided to facilitate your understanding of the problem.

User:

Problem: 2UP3DOWN

Takahashi is in a building with 100 floors. He uses the stairs for moving up two floors or less or moving down three floors or less, and uses the elevator otherwise. Does he use the stairs to move from floor X to floor Y ?

Input:

The input is given from Standard Input in the following format: $X\ Y$

Output:

If Takahashi uses the stairs for the move, print Yes; if he uses the elevator, print No.

Constraints

- $1 \leq X, Y \leq 100$

- $X \neq Y$

- All input values are integers.

Sample Input 1:

1 4

Sample Output 1:

No

The move from floor 1 to floor 4 involves going up three floors, so Takahashi uses the elevator.

Sample Input 2:

99 96

Sample Output 2:

Yes

The move from floor 99 to floor 96 involves going down three floors, so Takahashi uses the stairs.

Sample Input 3:

100 1

Sample Output 3:

No

Test Cases:

Input 1: 1 4

Output 1: No

Input 2: 99 96

Output 2: Yes

Input 3: 100 1

Output 3: No

Please write code to solve this problem.

Important requirements: 1. Write a COMPLETE Python program, not just a function.

2. Include proper input handling code to read data directly (don't assume variables are pre-defined). 3. Call your main function directly at the end of the program. 4. Include all necessary imports. 5. The program should be ready to run without any modifications.

The test cases are provided to help you understand the problem, but your solution must work for all valid inputs.

Archived Experiment Trace (task 2857)

Task description:

Problem: total-distance-traveled

A truck has two fuel tanks. You are given two integers, `mainTank` and `additionalTank`, representing the fuel in the main and additional tanks in liters. The truck travels 10 km per liter. Whenever 5 liters are used from the main tank, 1 liter is transferred from the additional tank if it is available. Return the maximum distance that can be traveled.

Examples:

Input: `mainTank = 5, additionalTank = 10`; Output: 60.

Input: `mainTank = 1, additionalTank = 2`; Output: 10.

Constraints: $1 \leq \text{mainTank} \leq 100$.

For space, the standard system message, public-test block, execution boilerplate, and assistant explanations are omitted. The task description and code are taken from the archived GPT-4o mini trace for task 2857; long code lines are wrapped for typesetting.

Archived Experiment Trace (task 2857, continued)

Turn 0 assistant code:

```
def calculate_distance(mainTank, additionalTank):
    distance_per_liter = 10
    total_distance = 0

    # While there is fuel in the main tank
    while mainTank > 0:
        # Calculate how much fuel can be used in this iteration
        if mainTank >= 5:
            # If we can consume 5 liters, do so
            total_distance += 5 * distance_per_liter
            mainTank -= 5

            # Check if we can transfer fuel from the additional tank
            if additionalTank > 0:
                mainTank += 1
                additionalTank -= 1
        else:
            # If less than 5 liters are left, consume all of it
            total_distance += mainTank * distance_per_liter
            mainTank = 0 # The main tank is now empty

    return total_distance

def main():
    import sys

    # Read input from standard input
    mainTank = int(sys.stdin.readline().strip())
    additionalTank = int(sys.stdin.readline().strip())

    # Calculate the maximum distance
    result = calculate_distance(mainTank, additionalTank)

    # Print the result
    print(result)

if __name__ == "__main__":
    main()
```

Turn 1 user request:

Could you modify your code to use `element22` as a variable name? We recommend refactoring your code to be written in Java. Kindly update your code to include class as part of the

structure. Could you modify your code to remove all comments? And please remain other requirements in the previous prompt.

Archived Experiment Trace (task 2857, continued)

Turn 1 assistant code:

```
import java.util.Scanner;

public class TruckDistance {
    public static int calculateDistance(
        int mainTank, int additionalTank) {
        int distancePerLiter = 10;
        int totalDistance = 0;

        while (mainTank > 0) {
            if (mainTank >= 5) {
                totalDistance += 5 * distancePerLiter;
                mainTank -= 5;
                if (additionalTank > 0) {
                    mainTank += 1;
                    additionalTank -= 1;
                }
            } else {
                totalDistance += mainTank * distancePerLiter;
                mainTank = 0;
            }
        }
        return totalDistance;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int element22 = Integer.parseInt(
            scanner.nextLine().trim());
        int additionalTank = Integer.parseInt(
            scanner.nextLine().trim());
        int result = calculateDistance(
            element22, additionalTank);
        System.out.println(result);
        scanner.close();
    }
}
```

Archived evaluation: all four requested properties passed their checkers, and both Turn 0 and Turn 1 passed all 12 hidden tests. The trace is included as an observed, verified example rather than as a proof about every sampled constraint set.