Sense and Sensitivity: Examining the Influence of Semantic Recall on Long Context Code Understanding

Anonymous ACL submission

Abstract

Large language models (LLMs) are increasingly deployed for understanding large codebases, but whether they understand operational semantics of long code context or rely on pattern matching shortcuts remains unclear. We distinguish between lexical recall (retrieving code verbatim) and semantic recall (understanding operational semantics). Evaluating 10 state-of-the-art LLMs, we find that while frontier models achieve near-perfect, position-independent lexical recall, semantic recall degrades severely when code is centrally positioned in long contexts. We introduce semantic recall sensitivity to measure whether tasks require understanding of code's operational semantics vs. permit pattern matching shortcuts. Through a novel counterfactual measurement method, we show that models rely heavily on pattern matching shortcuts to solve existing code understanding benchmarks. We propose a new task `SemTrace`, which achieves high semantic recall sensitivity through unpredictable operations; LLMs' accuracy exhibits severe positional effects, with median accuracy drops of 92.73% versus `CRUXEval`'s 53.36% as the relevant code snippet approaches the middle of the input code context. Our findings suggest current evaluations substantially underestimate semantic recall failures in long context code understanding.

1 Introduction

Large Language Models (LLMs) are increasingly applied to industry coding tasks (Edwards, 2024) that demand understanding of large codebases (Jimenez et al., 2024). Recent advances (Dao et al., 2022; Peng et al., 2024; Su et al., 2024) enable these models to process extremely long inputs, up to millions of tokens (OpenAI, 2025). However, a fundamental question remains unanswered: when models solve code understanding tasks, are they processing the specific code provided in context, or applying memorized patterns from pretraining? This distinction becomes critical as LLMs are deployed in production environments where they must handle novel, project-specific code that cannot be solved through pattern matching alone. Pattern matching shortcuts may also result in LLMs missing subtle vulnerabilities (Ding et al., 2025).

We introduce a key distinction between two capabilities for code understanding in long contexts: *lexical recall*, meaning the ability to locate and reproduce code verbatim, and *semantic recall*, meaning the ability to remember what code does when it is run, i.e., its operational semantics (Winskel, 1993). These capabilities are distinct; models can have perfect lexical recall yet fail at semantic recall, demonstrating they can access relevant code but not understand its effects. While needle-in-the-haystack (NIAH) benchmarks (Liu et al., 2024a,c) measure the lexical recall, the relationship between lexical and semantic recall is not well understood.

Moreover, code understanding tasks like output prediction are designed to measure semantic recall, but can often be solved through pattern-matching shortcuts (recognizing familiar algorithms, applying memorized correlations) without requiring semantic recall of the specific implementation. This conflation poses a fundamental evaluation challenge: existing benchmarks may allow shortcuts that mask semantic recall failures. We introduce *semantic recall sensitivity* as a property of tasks–the degree to which solving the task requires semantically recalling specific code details rather than pattern matching.

To investigate whether lexical and semantic recall rely on different mechanisms, we leverage positional variation in long contexts as a diagnostic lens for how models integrate information, rather than as an end in itself. Positional variation naturally arises as code length scales, and understanding of code should remain stable regardless of where relevant details occur. Prior work has shown that LLM

performance varies based on where information appears in the input (Liu et al., 2024b; Lu et al., 2022), but here we treat position as a controlled perturbation to probe representational differences between lexical and semantic recall. Our evaluation across 10 state-of-the-art LLMs reveals a clear dissociation: frontier models achieve near-perfect, position-independent lexical recall of relevant code, yet semantic recall on the same code degrades when centrally positioned within the code context.

On `CRUXEval` (Gu et al., 2024), a popular input-output prediction benchmark, semantic recall shows moderate position-dependent degradation (median accuracy drop of 53.36%) compared to negligible lexical recall degradation (2.39%). However, we hypothesize this substantially underestimates semantic recall fragility due to low semantic recall sensitivity, as `CRUXEval` permits pattern matching that compensates for semantic recall failures. To test this, we propose a novel counterfactual measurement method: systematically removing lines from code and measuring the LLMs' performance degradation. Code that requires semantic recall should cause the LLMs' performance to fall sharply (like the Python interpreter), since the code's operational semantics eventually changes; LLMs using pattern matching degrade gradually. `CRUXEval` shows gradual degradation (only 44.15-59.74% accuracy loss at 50% line removal), confirming low sensitivity.

To isolate semantic recall, we introduce `SemTrace`, an output prediction task that achieves high semantic recall sensitivity through unpredictable operations. `SemTrace` reveals dramatically more severe degradation: median accuracy drops of 92.73% versus `CRUXEval`'s 53.36%, with some frontier models reaching zero accuracy when code is centrally positioned. Even GPT 4.1 shows clear position-dependence when scaled to higher-digit arithmetic that cannot be memorized, confirming that moderate degradation on existing benchmarks masks severe semantic recall fragility.

Contributions. (1) We introduce the distinction between lexical recall (accessing code verbatim) and semantic recall (understanding operational semantics), demonstrating they dissociate in long contexts: models maintain near-perfect, position-independent lexical recall while semantic recall exhibits severe position-dependent failures. (2) We propose semantic recall sensitivity as a task property with a counterfactual measurement method, revealing existing benchmarks like `CRUXEval` have low sensitivity, allowing pattern matching to mask failures. (3) We introduce `SemTrace`, an output prediction task isolating semantic recall via unpredictable operations, revealing 92.73% median accuracy degradation vs. 53.36% on `CRUXEval`, demonstrating current evaluations substantially underestimate semantic recall challenges.

2 Related Work

Our work intersects a number of research areas, namely long context evaluation, positional effects in long contexts, and memorization.

2.1 Long Context Evaluation

Significant effort has been devoted to long-context evaluation, expanding from needle-in-haystack retrieval (Mohtashami and Jaggi, 2023; Shaham et al., 2023; Greg Kamradt, 2023) to comprehensive benchmarks (Bai et al., 2024, 2025), extended context lengths (Zhang et al., 2024a), and complex reasoning tasks (Kuratov et al., 2024; Hsieh et al., 2024; Levy et al., 2024; Modarressi et al., 2025). For code specifically, prior work has focused on repository-level retrieval and code completion (Guo et al., 2023; Liu et al., 2024c), semantic search (Liu et al., 2024a), and code repository QA (Bai et al., 2025). However, these tasks primarily evaluate *code retrieval* (locating relevant code given queries) rather than understanding *operational semantics* (predicting execution behavior or input-output mappings). While prior work evaluates retrieval and understanding as separate tasks, we test both capabilities on identical code in long contexts to understand their relationship. We also introduce semantic recall sensitivity as a property for characterizing code understanding benchmarks.

2.2 Position Effects in Long Context

Prior work has identified positional biases in LLMs, ranging from long context scenarios (Zhang et al., 2024b) to in-context learning (Fang et al., 2025; Min et al., 2022; Lu et al., 2022), with empirical and theoretical evidence suggesting the important role of positional encoding, loss functions, and data distribution (Wu et al., 2025; Gu et al., 2025). Particularly relevant to our work is the 'lost-in-the-middle' effect (Liu et al., 2024b; Gao et al., 2024), where performance significantly degrades when relevant information is centrally positioned within the natural language input. However, prior work has not examined position effects on LLMs' understanding of operational semantics. We are

the first to investigate how position affects code understanding, and, critically, to distinguish how it differentially impacts lexical vs. semantic recall.

2.3 Memorization

LLMs have been shown to have memorized numerous algorithms, facts, and patterns from pretraining (Hartmann et al., 2023; Chang et al., 2023; Nanda et al., 2023), which can mask their true reasoning ability and hinder evaluation (Huang and Chang, 2023). To counter memorization, prior work perturbs problems such that their solutions change (e.g., 1-indexed Python (Wu et al., 2024) or altered math problems (Huang et al., 2025)). We propose a complementary counterfactual approach: rather than perturbing code to alter results, we systematically remove information (one line at a time) from code snippets to ultimately render them unsolvable. If accuracy remains stable despite missing critical information, the model is relying on memorized patterns rather than the provided code.

3 Lexical and Semantic Recall

We distinguish between two types of recall capabilities that become critical when evaluating code understanding in long contexts. The first is *Lexical Code Recall (R^L)*, defined as the ability to retrieve or reproduce code snippets from the input context verbatim, without necessarily understanding their meaning or behavior. This capability can be directly assessed through retrieval or cloze-style tasks, and modern LLMs achieve near-perfect performance on such benchmarks (Yang et al., 2025; OpenAI, 2025).

The second is *Semantic Code Recall (R^S)*, defined as the ability to understand what a code snippet from the input context does when it is run: its operational semantics (Winskel, 1993). R^S measures the *usability* of the model's representation for performing downstream reasoning tasks that depend on the provided code. For instance, given an in-context implementation of quicksort and an unsorted array, $R^L(quicksort)$ corresponds to regenerating the quicksort code verbatim, whereas $R^S(quicksort)$ corresponds to leveraging understanding of that code to correctly predict a sorted output array by tracing quicksort's execution steps.

These capabilities are distinct: models can have perfect lexical recall yet fail at semantic recall, demonstrating they can access relevant code but not process it. While needle-in-the-haystack benchmarks (Liu et al., 2024a,c) measure lexical recall, no prior work has systematically investigated semantic recall (understanding operational semantics through tasks like input-output prediction) in long contexts. Moreover, the relationship between these capabilities remains poorly understood.

3.1 Semantic Recall Sensitivity

We introduce *semantic recall sensitivity* as a property of a code understanding task quantifying the degree to which a task requires understanding operational semantics of a given code snippet rather than permitting pattern matching shortcuts. *Pattern matching shortcuts* refer to solving tasks by applying memorized algorithmic patterns or spurious input-output correlations from pretraining, thereby bypassing semantic recall of the specific provided code. The semantic recall sensitivity of a benchmark is the average sensitivity across its examples.

Consider an output prediction task where a model must predict what a function returns given some input. If the function implements a standard algorithm like quicksort, a model might predict the sorted output by recognizing the algorithm's structure and applying memorized sorting behavior, without semantically recalling the specific implementation details. Such a task exhibits *low semantic recall sensitivity*. Conversely, if the function implements a novel or arbitrary computation (e.g., a specific non-standard permutation of the input array defined only within the context), memorized patterns provide no shortcuts; instead, the model must semantically recall the specific code lines. This task would have *high semantic recall sensitivity*, as pre-trained knowledge alone is insufficient.

This distinction has critical implications for evaluation. Low semantic recall sensitivity is inherently problematic for two reasons. First, in production environments, models encounter novel, project-specific code where pattern matching shortcuts are unavailable. Benchmarks permitting shortcuts fail to assess this critical capability. Second, the difference between correct and vulnerable code often lies in a single line: Ding et al. (2025) show models struggle to distinguish correct from vulnerable implementations when differences are subtle, suggesting pattern matching may be dangerous when small implementation details matter.

Moreover, low sensitivity masks semantic recall failures in long contexts. If pattern matching compensates for position-dependent semantic recall degradation, benchmarks will systematically under-

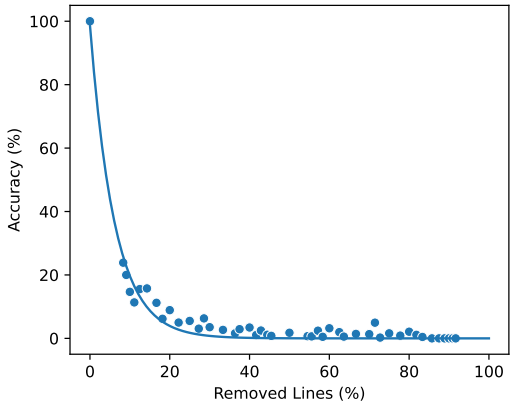

Figure 1: Python interpreter performance on incomplete `CRUXEval` functions. Accuracy drops sharply and approaches zero beyond 20% removal, establishing the expected pattern when output prediction genuinely requires understanding all provided code. This provides a reference for evaluating LLM semantic recall sensitivity. Exponential trend fitted with bootstrapped 95% CI.

estimate the severity of understanding failures. We investigate this empirically in §5.1, demonstrating that low-sensitivity benchmarks exhibit smoother degradation curves that obscure severe underlying semantic recall fragility.

3.2 Measuring Semantic Recall Sensitivity

While semantic recall sensitivity characterizes a fundamental property of code understanding tasks, measuring it presents a challenge: how do we determine whether a model's success depends on understanding specific code versus applying pattern matching shortcuts?

We propose estimating semantic recall sensitivity by measuring the impact of withholding information from the code snippet. The key intuition is that for code reasoning tasks like output prediction, removing lines fundamentally changes the code such that it can no longer produce the correct output. Therefore, if a model continues to provide correct answers despite missing critical information, it cannot be reasoning about the specific provided code; instead, it must be relying on memorized patterns. Conversely, if performance degrades sharply as information is removed, the task requires genuine semantic recall of the specific code details.

Formally, let $Inc(C)$ denote a set of incomplete versions C^{inc} from the original snippet C (e.g., by removing specific lines). We define the sensitivity of task $\mathcal{T}$ with respect to C, denoted $Sens^{\mathcal{T}}(C)$, as the average normalized performance degradation

```python
def f(x):
    arr = [0, 0, 0, 0]
    arr[0] = x - 43
    arr[2] = x - 65
    arr[1] = x + 88
    arr[3] = x - 74
    return arr
```

Listing 1: Example from `SemTrace`. Given input $x = 81$, the model must predict the output $[38, 169, 16, 7]$ by correctly recalling and applying each assignment line.

across these incomplete versions (Equation 1):

$$\frac{1}{|Inc(C)|} \sum_{C^{inc} \in Inc(C)} \frac{R^{\mathcal{T}}(C) - R^{\mathcal{T}}(C^{inc})}{R^{\mathcal{T}}(C) + \epsilon} \quad (1)$$

Here, $R^{\mathcal{T}}(C)$ is the model's reasoning performance on the original snippet, and $R^{\mathcal{T}}(C^{inc})$ is its performance on an incomplete version. The term inside summation captures the relative performance caused by the incomplete information in $C^{inc} \in Inc(C)$ (with $\epsilon > 0$ added for numerical stability). Geometrically, $Sens^{\mathcal{T}}(C)$ measures the area under the performance degradation curve as information is progressively removed from C. A higher $Sens^{\mathcal{T}}(C)$ indicates a steeper degradation, meaning that successfully performing task $\mathcal{T}$ on snippet C heavily depends on semantically recalling the specific details within C, thus signifying high semantic recall sensitivity.

To validate this approach and establish a reference point, we measure the Python interpreter's performance on incomplete CRUXEval functions. As expected, accuracy drops sharply–76 percentage points after removing just one line–and approaches zero beyond 20% removal (Figure 1). This exponential decay confirms that output prediction fundamentally requires semantic understanding of all code lines. Models showing gradual rather than sharp degradation therefore demonstrate low semantic recall sensitivity, indicating reliance on pattern matching rather than true code understanding.

3.3 Measuring Semantic Recall: `SemTrace`

Having established how to measure semantic recall sensitivity, we now introduce a task designed to achieve high sensitivity by construction. To prevent pattern matching shortcuts, the task must require understanding implementation-specific details that cannot be inferred from algorithmic patterns or memorized correlations. We propose `SemTrace`, an output prediction task where errors can be directly attributed to semantic recall failures rather than general reasoning deficiencies. In `SemTrace`,

the model receives a Python function that takes an integer x as input, initializes a list, and populates it through assignment statements before returning the list. Crucially, each assignment modifies a distinct list element by adding a value y (drawn uniformly from $[-100, 99]$) to the input x, and the order of assignment lines is randomized. An example is shown in Listing 1.

We use simple 2-digit addition and subtraction operations ($x+y$) to minimize reasoning confounds while preventing pattern matching. These operations are simple enough that frontier LLMs can perform them reliably in isolation, ensuring that reasoning difficulty does not obscure semantic recall effects (we verify this empirically in Table 1). However, because operations follow no predictable pattern, models cannot rely on pretrained heuristics without memorizing solutions for all possible combinations. The probability of guessing a single operation correctly is $\frac{1}{200}$; for functions with k assignments (k drawn uniformly from $[4, 10]$), the probability of guessing the entire output array by chance, even assuming knowledge of the generation process, is negligible (at most $(\frac{1}{200})^4 = 6.25 \times 10^{-10}$). Correctly predicting the output therefore necessitates accurate R^S of all assignment lines, rendering `SemTrace` highly sensitive to R^S.

Since `SemTrace`'s lines are mutually independent and comparable in difficulty, we can measure the degree of R^S through partial matches: counting how many list positions the model predicted correctly. This metric allows us to distinguish semantic recall failures (some positions correct, indicating partial understanding) from complete breakdowns (near-random performance), confirming that errors reflect failures to recall specific lines rather than inability to perform the underlying operations.

4 Experimental Setup

Models. We evaluate 10 state-of-the-art LLMs including GPT 4.1 (2025/04/14) and five open-weight frontier models (DeepSeek Coder V2 Instruct, Gemma 3 27B It, Llama 3.3 70B Instruct, Qwen 2.5 Coder 32B Instruct, and Codestral 22B v0.1), and their smaller counterparts (Gemma 3 12B It, Llama 3.1 8B Instruct, Qwen 2.5 Coder 7B Instruct, DeepSeek Coder V2 Lite Instruct) to examine scaling effects. Main results focus on the six frontier models for clarity; we include results on smaller models in §A. Open-weight models were served on a `p5e.48xlarge` AWS EC2 instance using vLLM (Kwon et al., 2023). Following prior work (Liu et al., 2024b), we use greedy decoding for reproducibility. To manage costs, GPT 4.1 experiments use one-eighth of each dataset, randomly sampled.

Experimental Design. To isolate positional effects, we embed target code within contexts of irrelevant distractor code. Distractor functions are randomly selected Python functions from `CodeSearchNet` (Husain et al., 2019), filtered by character count (25th–75th percentile, ∼200 tokens each) to exclude outliers. We use 20, 40, 60, or 80 distractor functions (∼4k to ∼16k total tokens), systematically varying the target code's position across 11 equally spaced locations within each context. Crucially, distractors are unrelated to the tasks, minimizing potential confounds from semantic interference while allowing us to cleanly measure position and context length effects.

Tasks. We evaluate three capabilities: (1) semantic recall via `CRUXEval` (Gu et al., 2024) input and output prediction (800 Python functions each); (2) lexical recall via function-level retrieval on the same `CRUXEval` functions, where models must reproduce entire function bodies verbatim; (3) high-sensitivity semantic recall via `SemTrace` output prediction (800 generated functions).

For lexical recall, we prepend each line with a unique 6-digit hexadecimal key following key-value retrieval approaches (Li* et al., 2023; Liu et al., 2024b). This provides unambiguous reference markers and prevents models from identifying targets through naming patterns, which could confound position effects. For semantic recall sensitivity measurement, we generate incomplete versions of `CRUXEval` functions by systematically removing lines. We report zero-shot, exact match accuracy for all tasks. In Figure 2, results are shown as relative accuracy, normalized to each model's maximum performance at that context length, to facilitate comparison across models with different baseline capabilities.

Prompting. Following Liu et al. (2024b), we use query-aware contextualization–placing the query both before and after the code–to enable decoder-only models to attend to the query when processing code. Prompt templates appear in §E.

5 Results

Table 1 shows baseline performance without long-context distractors. Notably, frontier models

achieve higher median accuracy on SemTrace (86.19%) than on CRUXEval input (45.06%) or output prediction (55.38%), indicating SemTrace is not inherently more difficult. This establishes an important baseline: the dramatic position-dependent degradation we observe on SemTrace cannot be attributed to task difficulty alone.

Model	CRUXEval-I/O	SemTrace
Codestral 22B	44.62 / 50.38	23.25
GPT 4.1	79.00 / 77.00	100.00
Deepseek Coder V2	36.50 / 54.00	96.00
Gemma 3 27B	50.62 / 25.87	12.25
Llama 3.3 70B	18.25 / 56.75	87.12
Qwen 2.5 Coder 32B	45.50 / 61.75	85.25

Table 1: Baseline performance (% accuracy) without distractors. CRUXEval-I/O shows input/output prediction accuracy. Median accuracy is higher for SemTrace (86.19%) than CRUXEval (45.06% - 55.38%), indicating its severe position-dependent degradation (Figure 2 (c)) reflects high semantic recall sensitivity rather than inherent task difficulty.

5.1 Lexical Recall Remains Stable While Semantic Recall Degrades with Position

Figure 2 presents our central finding: a stark dissociation between lexical and semantic recall as code position varies in long contexts.

Lexical recall is position-independent. Figure 2a shows frontier models maintain over 95% accuracy on function retrieval regardless of where target code appears, demonstrating they can reliably access code from any position within the tested context windows.

Semantic recall degrades with position, moderately on CRUXEval. Figure 2b reveals accuracy dropping as target code moves toward the center, with relative accuracy decreasing 16.25-84.29% from maximum. Performance declines most severely when code appears at the 60-80% position, while remaining relatively stable near context boundaries. This pattern is already present with only 20 distractor contexts (4k tokens), becoming more pronounced as the code context size increases. CRUXEval input prediction exhibits similar pattern (see §B for details).

Semantic recall degrades severely on SemTrace. Figure 2c shows dramatically intensified degradation compared to CRUXEval. Despite being easier at baseline (Table 1), SemTrace accuracy plummets when target code is centrally positioned. For example, Qwen 2.5 Coder 32B, the second-best performer on CRUXEval-O with only 23.31% relative degradation in accuracy for an 80 distractor code context (16k tokens), suffers 91.38% relative accuracy loss on SemTrace, dropping 53 percentage points; Codestral 22B's and Gemma 3 27B's accuracy reaches zero. This severe degradation appears even at shorter contexts and deepens substantially as context grows. Partial match analysis (§ C) confirms this represents gradual semantic recall degradation rather than complete failure.

Retrieval failures cannot explain semantic degradation. Models demonstrably access code near-perfectly (Figure 2a) yet fail to understand its operational semantics (Figure 2b-c). The remaining question is: why does semantic recall show such different positional sensitivity between CRUXEval (moderate) and SemTrace (severe)?

5.2 CRUXEval's Low Semantic Recall Sensitivity Masks Position Effects

Why does semantic recall show such different positional sensitivity between CRUXEval (moderate degradation) and SemTrace (severe degradation)? We hypothesize that CRUXEval has low semantic recall sensitivity, meaning models can partially solve output prediction tasks by applying pattern matching shortcuts from pretraining rather than semantically recalling the specific provided code.

To test this, we generate incomplete versions of each CRUXEval function by removing all possible combinations of lines while preserving function signatures, creating 71,994 incomplete functions. We measure accuracy aggregated by percentage of lines removed. If models truly require semantic recall of all provided code, we should observe sharp exponential decay similar to the Python interpreter (Figure 1). Conversely, gradual degradation indicates models are compensating through pattern matching shortcuts instead.

Counterfactual measurement reveals gradual degradation instead of sharp decline. Figure 3 shows the results across all 10 evaluated models. Removing 50% of lines–making the code fundamentally incomplete and unexecutable–causes only 44.15% accuracy drop for input prediction and 59.74% for output prediction, contrasting sharply with the Python interpreter's exponential decay.

Larger models show slightly higher sensitivity but remain resilient. Larger models (GPT 4.1, DeepSeek Coder V2, Llama 3.3 70B, Qwen 2.5 Coder 32B) exhibit sharper performance declines than smaller models, suggesting their greater re-

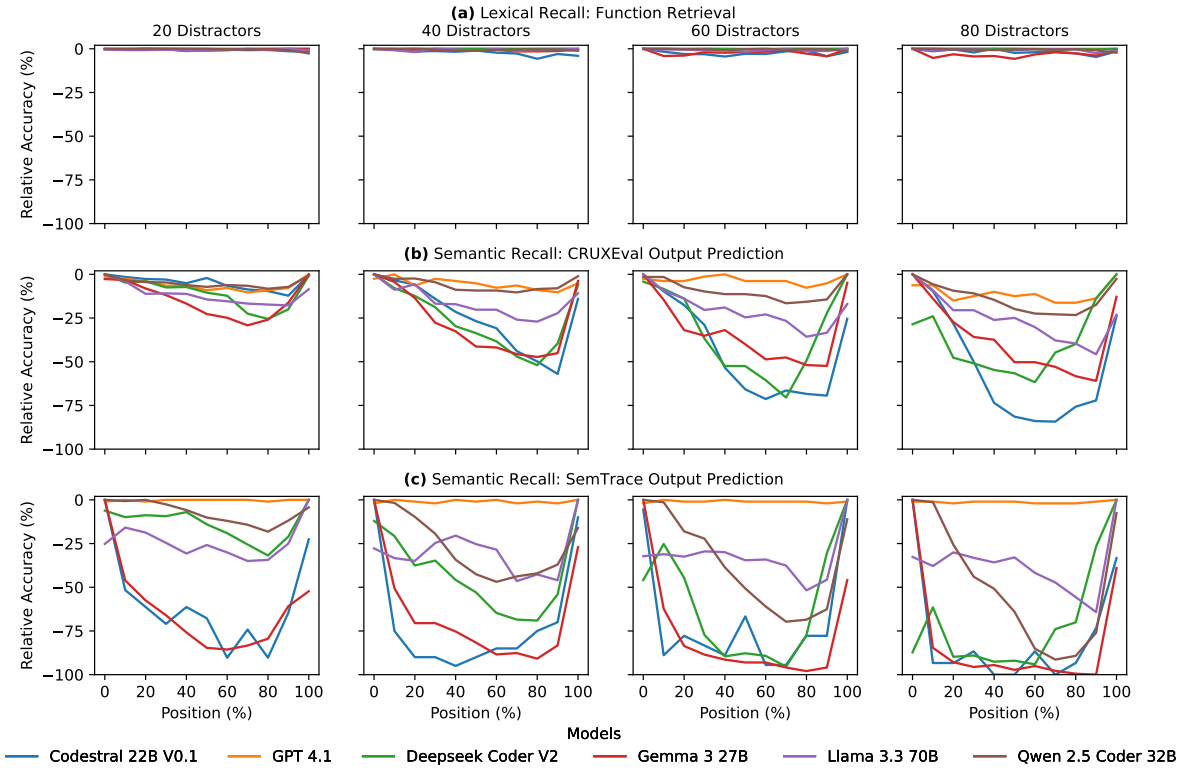

Figure 2: Dissociation between lexical and semantic recall across code positions in long contexts. (a) Lexical recall (function retrieval) remains near-perfect and position-independent across all context lengths for frontier models. (b) Semantic recall on `CRUXEval` output prediction (`CRUXEval-O`) exhibits moderate lost-in-the-middle effects, with median accuracy decreases of 53.36% as context length increases and target code moves toward the middle. (c) High-sensitivity semantic recall (`SemTrace`) shows severe position dependence, with median accuracy plummeting up to 92.73% when target code is centrally positioned.

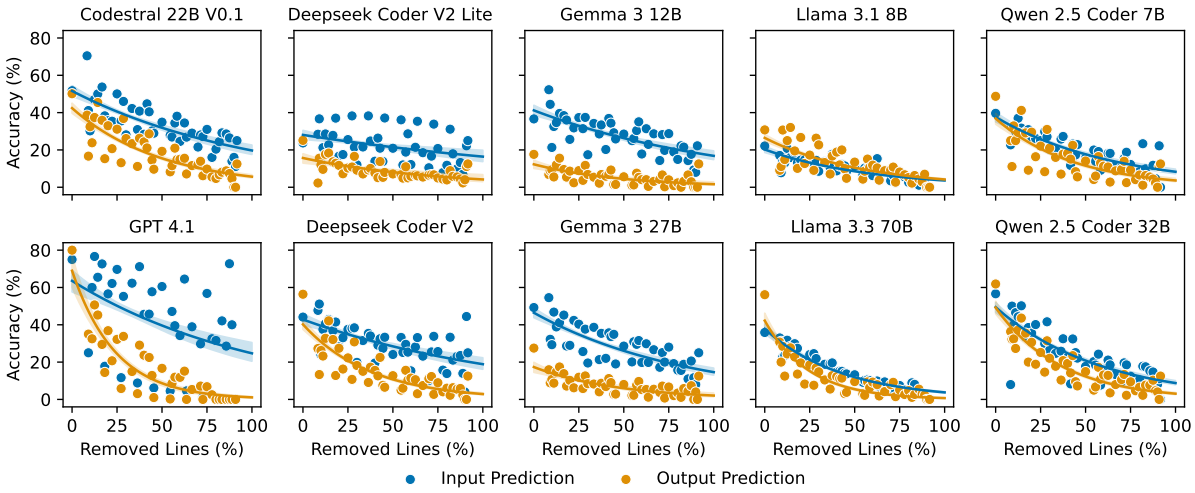

Figure 3: `CRUXEval` semantic recall sensitivity measurement across all models. Performance on input (blue) and output (orange) prediction tasks as lines are progressively removed from functions. We fit exponential trend lines using non-linear least squares and and leverage bootstrapping to compute 95% confidence intervals. Even with 50% of lines removed, models' accuracy drops only 44.15% and 59.74% for input and output prediction, respectively, showing gradual degradation that contrasts sharply with the Python interpreter's exponential decay (Figure 1). This gradual degradation indicates low semantic recall sensitivity, meaning models compensate for missing code through pattern matching from pretraining rather than relying on semantic understanding of the specific provided code.

liance on semantic recall. However, even these models degrade far more gradually than the Python interpreter, maintaining substantial accuracy despite fundamentally incomplete code.

The low sensitivity to missing information confirms low semantic recall sensitivity. Models leverage knowledge of common algorithmic patterns–sorting, string manipulation, basic arithmetic–to predict plausible outputs without fully understanding the specific implementation, masking semantic recall failures. This explains why CRUXEval shows only moderate position effects (Figure 2b) while SemTrace reveals severe degradation (Figure 2c).

5.3 GPT 4.1's Exception Reveals Memorization of Simple Arithmetic

Figure 2 reveals an anomaly: GPT 4.1 achieves near-perfect accuracy on SemTrace regardless of position or context length, while still exhibiting clear position-dependent degradation on CRUXEval. This makes GPT 4.1 the only model where the pattern inverts, performing better on SemTrace than CRUXEval, opposite to all other models. If GPT 4.1 had truly superior semantic recall capabilities, why would it fail on the less sensitive task (CRUXEval) while succeeding on the more sensitive one (SemTrace)?

We hypothesize that GPT 4.1 has memorized 2-digit arithmetic, effectively converting SemTrace from a semantic recall task into a lexical recall task by simply retrieving memorized addition/subtraction results rather than reasoning about code. To test this, we evaluate GPT 4.1 on SemTrace variants using 3-, 4-, 5-, and 6-digit operations.

Figure 4 shows the results with 80 distractors (∼16k tokens). GPT 4.1 maintains near-perfect performance on 2- and 3-digit SemTrace across all positions. However, position-dependent degradation emerges clearly for 4-digit operations, with accuracy dropping up to 31 percentage points when code is centrally positioned. This degradation intensifies for 5- and 6-digit operations, with 6-digit SemTrace showing up to 43% relative accuracy loss and pronounced U-shaped curves.

This finding reveals that even high-sensitivity tasks remain vulnerable to shortcuts if they involve operations models can solve without processing code. GPT 4.1's perfect 2-digit performance appeared to demonstrate superior semantic recall but exhibited position-independence characteristic of retrieval rather than processing: a pattern consistent with memorized arithmetic. The emergence of position-dependence at higher digits suggests that when such shortcuts become unavailable, even GPT 4.1 must rely on genuine semantic recall, which exhibits the same fragility as other frontier models. This underscores a key design principle: measuring semantic recall requires preventing both pattern matching shortcuts and operations that models can solve through mechanisms other than processing the specific provided code.

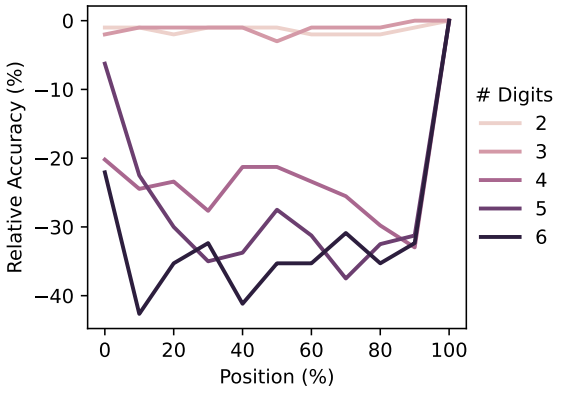

Figure 4: GPT 4.1's higher-digit SemTrace performance reveals arithmetic memorization. Testing with 2-6 digit operations at 80 distractors shows position-independent performance for 2-3 digits, but clear position-dependent degradation for 4+ digits (31pp drop at 4 digits, 43% relative loss at 6 digits). This demonstrates GPT 4.1's perfect 2-digit performance (Figure 2c) reflects memorized arithmetic rather than superior semantic recall.

6 Conclusion

We introduced the distinction between lexical code recall (verbatim retrieval) and semantic code recall (understanding what code does), demonstrating that while frontier LLMs achieve near-perfect position-independent lexical recall, semantic recall exhibits severe lost-in-the-middle effects with median accuracy drops of 92.73%. We proposed semantic recall sensitivity as a property of benchmarks and showed that existing code reasoning benchmarks like CRUXEval have low sensitivity compared to SemTrace, allowing pattern matching to mask the true severity of position-dependent failures. Our findings reveal that current long-context evaluations may substantially underestimate the challenges models face when reasoning about novel, unfamiliar code. Future work should develop benchmarks with high semantic recall sensitivity to more accurately assess models' true code understanding capabilities in long contexts.

7 Limitations

Language Coverage. Our evaluation focuses exclusively on Python code. While Python is widely used in both research and industry, code understanding capabilities may vary across programming languages with different syntactic and semantic properties. Future work should extend this analysis to other languages to determine whether our findings generalize.

Distractor Code Design. We use randomly sampled, semantically unrelated Python functions as distractors to isolate positional effects. While this design enables clean measurement of position-dependent degradation, real-world codebases contain semantically related functions that may introduce interference effects not captured in our study. While we anticipate this would lead to further deterioration, the relationship between semantic similarity of surrounding code and semantic recall failures remains an open question.

Task Scope. Our evaluation focuses on input-output prediction and code retrieval tasks. While these tasks directly measure semantic and lexical recall, production code understanding involves additional capabilities such as code generation, debugging, and complex multi-hop reasoning. The extent to which our findings apply to these broader tasks requires further investigation.

`SemTrace` Design Constraints. `SemTrace` uses simple arithmetic operations to isolate semantic recall from reasoning difficulty. While this design choice enables clear attribution of errors to semantic recall failures, it may not capture semantic understanding challenges in more complex algorithmic contexts. Additionally, our discovery that GPT 4.1 has memorized arithmetic operations highlights an inherent limitation: as models grow more capable, designing tasks that prevent all possible shortcuts becomes increasingly difficult.

Context Length Range. Our experiments evaluate contexts up to approximately 16k tokens. While this range covers common code understanding scenarios, recent models support significantly longer contexts (up to millions of tokens). While our results suggest increasing context length leads to increased semantic recall degradation, whether the pattern continues at extreme context lengths is left for future work.

Benchmark Scale. Our evaluations use 800 examples per task. While sufficient to observe consistent patterns across models, larger-scale evaluation could reveal more subtle effects and enable more fine-grained analysis of failure modes.

References

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2024. LongBench: A bilingual, multitask benchmark for long context understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3119–3137, Bangkok, Thailand. Association for Computational Linguistics.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2025. Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. *Preprint*, arXiv:2412.15204.

Kent Chang, Mackenzie Cramer, Sandeep Soni, and David Bamman. 2023. Speak, memory: An archaeology of books known to ChatGPT/GPT-4. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7312–7327, Singapore. Association for Computational Linguistics.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Re. 2022. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2025. Vulnerability detection with code language models: How far are we? In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1729–1741.

Benj Edwards. 2024. Google CEO says over 25% of new Google code is generated by AI.

Lizhe Fang, Yifei Wang, Khashayar Gatmiry, Lei Fang, and Yisen Wang. 2025. Rethinking invariance in in-context learning. In *The Thirteenth International Conference on Learning Representations*.

Muhan Gao, TaiMing Lu, Kuai Yu, Adam Byerly, and Daniel Khashabi. 2024. Insights into LLM long-context failures: When transformers know but don't tell. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 7611–7625, Miami, Florida, USA. Association for Computational Linguistics.

Greg Kamradt. 2023. Needle In A Haystack - Pressure Testing LLMs. Original-date: 2023-11-11T00:50:02Z.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. CRUXEval: A benchmark for code reasoning, understanding and execution. In *Forty-first International Conference on Machine Learning*.

Xiangming Gu, Tianyu Pang, Chao Du, Qian Liu, Fengzhuo Zhang, Cunxiao Du, Ye Wang, and Min Lin. 2025. When attention sink emerges in language models: An empirical view. In *The Thirteenth International Conference on Learning Representations*.

Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. Longcoder: a long-range pre-trained language model for code completion. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Valentin Hartmann, Anshuman Suri, Vincent Bindschaedler, David Evans, Shruti Tople, and Robert West. 2023. Sok: Memorization in general-purpose large language models. *Preprint*, arXiv:2310.18362.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, and Boris Ginsburg. 2024. RULER: What's the real context size of your long-context language models? In *First Conference on Language Modeling*.

Jie Huang and Kevin Chen-Chuan Chang. 2023. Towards reasoning in large language models: A survey. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 1049–1065, Toronto, Canada. Association for Computational Linguistics.

Kaixuan Huang, Jiacheng Guo, Zihao Li, Xiang Ji, Jiawei Ge, Wenzhe Li, Yingqing Guo, Tianle Cai, Hui Yuan, Runzhe Wang, Yue Wu, Ming Yin, Shange Tang, Yangsibo Huang, Chi Jin, Xinyun Chen, Chiyuan Zhang, and Mengdi Wang. 2025. MATH-perturb: Benchmarking LLMs' math reasoning abilities against hard perturbations. In *Forty-second International Conference on Machine Learning*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Yuri Kuratov, Aydar Bulatov, Petr Anokhin, Ivan Rodkin, Dmitry Igorevich Sorokin, Artyom Sorokin, and Mikhail Burtsev. 2024. BABILong: Testing the limits of LLMs with long context reasoning-in-a-haystack. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same task, more tokens: the impact of input length on the reasoning performance of large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15339–15353, Bangkok, Thailand. Association for Computational Linguistics.

Dacheng Li*, Rulin Shao*, Anze Xie, Ying Sheng, Lianmin Zheng, Joseph E. Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. 2023. How long can open-source llms truly promise on context length?

Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. 2024a. RepoQA: Evaluating long context code understanding. In *First Workshop on Long-Context Foundation Models @ ICML 2024*.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024b. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2024c. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*.

Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2022. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8086–8098, Dublin, Ireland. Association for Computational Linguistics.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the role of demonstrations: What makes in-context learning work? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11048–11064, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Ali Modarressi, Hanieh Deilamsalehy, Franck Dernoncourt, Trung Bui, Ryan A. Rossi, Seunghyun Yoon, and Hinrich Schütze. 2025. Nolima: Long-context evaluation beyond literal matching. *Preprint*, arXiv:2502.05167.

Amirkeivan Mohtashami and Martin Jaggi. 2023. Random-access infinite context length for transformers. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. 2023. Progress measures for grokking via mechanistic interpretability. In *The Eleventh International Conference on Learning Representations*.

OpenAI. 2025. Introducing gpt-4.1 in the api.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. 2024. YaRN: Efficient context window extension of large language models. In *The Twelfth International Conference on Learning Representations*.

Uri Shaham, Maor Ivgi, Avia Efrat, Jonathan Berant, and Omer Levy. 2023. ZeroSCROLLS: A zero-shot benchmark for long text understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 7977–7989, Singapore. Association for Computational Linguistics.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomput.*, 568(C).

Glynn Winskel. 1993. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA.

Xinyi Wu, Yifei Wang, Stefanie Jegelka, and Ali Jadbabaie. 2025. On the emergence of position bias in transformers. In *Forty-second International Conference on Machine Learning*.

Zhaofeng Wu, Linlu Qiu, Alexis Ross, Ekin Akyürek, Boyuan Chen, Bailin Wang, Najoung Kim, Jacob Andreas, and Yoon Kim. 2024. Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 1819–1862, Mexico City, Mexico. Association for Computational Linguistics.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 23 others. 2025. Qwen2.5 technical report. *Preprint*, arXiv:2412.15115.

Xinrong Zhang, Yingfa Chen, Shengding Hu, Zihang Xu, Junhao Chen, Moo Hao, Xu Han, Zhen Thai, Shuo Wang, Zhiyuan Liu, and Maosong Sun. 2024a. ∞Bench: Extending long context evaluation beyond 100K tokens. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15262–15277, Bangkok, Thailand. Association for Computational Linguistics.

Zhenyu Zhang, Runjin Chen, Shiwei Liu, Zhewei Yao, Olatunji Ruwase, Beidi Chen, Xiaoxia Wu, and Zhangyang Wang. 2024b. Found in the middle: How language models use long contexts better via plug-and-play positional encoding. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

A Results on Smaller Models

Model	CRUXEval-I/O	SemTrace
Deepseek Coder V2 Lite	20.00/21.00	4.12
Gemma 3 12B	39.50/19.25	15.88
Llama 3.1 8B	19.50/32.62	16.00
Qwen2.5 Coder 7B	31.62/46.62	22.25

Table 2: Baseline performance (% accuracy) without distractors. `CRUXEval-I/O` shows input/output prediction accuracy. TBD

We evaluate four smaller models (DeepSeek Coder V2 Lite, Gemma 3 12B, Llama 3.1 8B, and Qwen 2.5 Coder 7B) across all tasks. While these models exhibit lower absolute performance than their frontier counterparts (Table 2), they demonstrate qualitatively similar patterns. Figure 5 shows that smaller models maintain the key dissociation between lexical and semantic recall: they achieve relatively stable lexical recall across positions while showing moderate and severe position-dependent semantic recall degradation on both `CRUXEval` and `SemTrace`, respectively.

However, smaller models exhibit notable differences in lexical recall stability compared to frontier models. Figure 5a reveals greater positional variance and some anomalous accuracy drops, particularly DeepSeek Coder V2 Lite showing degradation in the middle of the input context, and both DeepSeek Coder V2 Lite and Llama 3.1 8B exhibiting drops at the beginning of the context. Despite these irregularities, the lexical recall patterns remain qualitatively distinct from the systematic lost-in-the-middle degradation observed in semantic recall tasks Figure 5b-c). Notably, the smaller models exhibit more pronounced performance variability and sharper degradation on `SemTrace`, suggesting their semantic recall capabilities are more fragile. However, the fundamental pattern (position-independent lexical recall vs. position-dependent semantic recall) remains consistent across model scales, confirming that our findings generalize beyond frontier models.

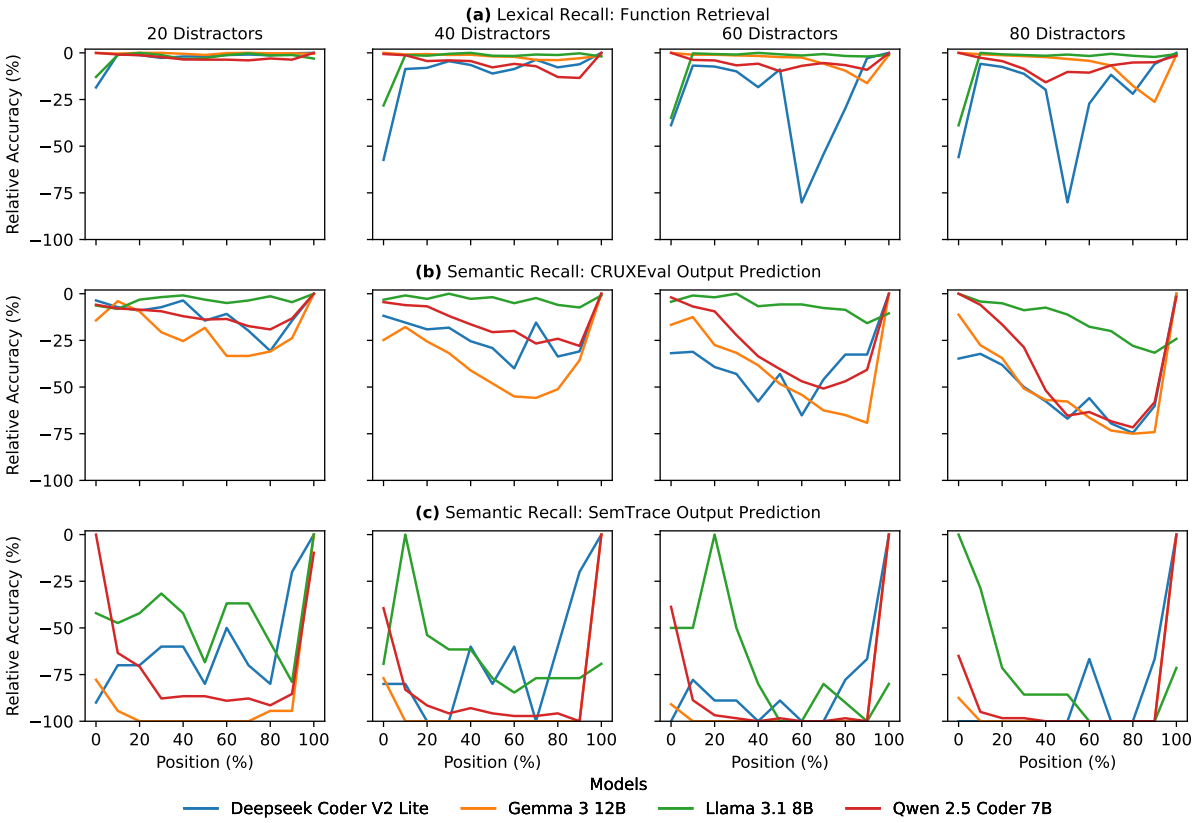

Figure 5: Dissociation between lexical and semantic recall across code positions in long contexts for smaller models. Positional effects across lexical recall (a), semantic recall on `CRUXEval` output prediction (`CRUXEval-O`) (b), and high-sensitivity semantic recall on `SemTrace` (c). Despite lower absolute performance, smaller models exhibit the same fundamental dissociation as frontier models: position-independent lexical recall and position-dependent semantic recall, with more severe degradation on `SemTrace`.

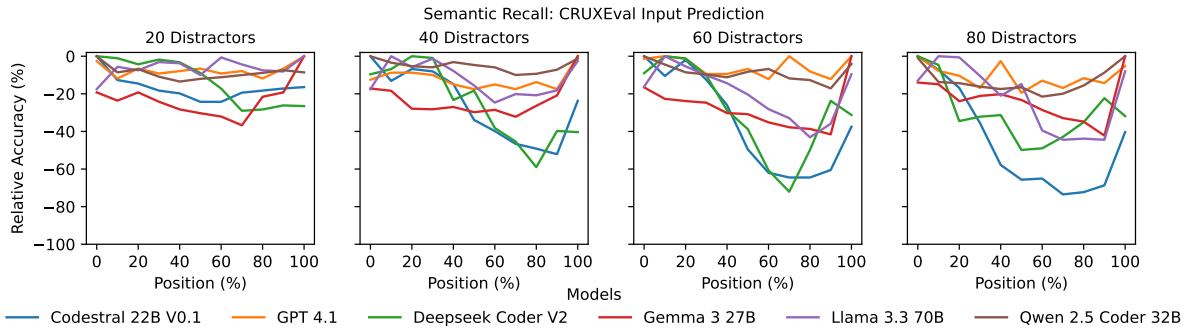

Figure 6: `CRUXEval` input prediction (`CRUXEval-I`) positional effects for frontier models. Similar to output prediction (Figure 2b), all models show moderate position-dependent degradation with the steepest declines occurring when target code appears at 60-80% positions.

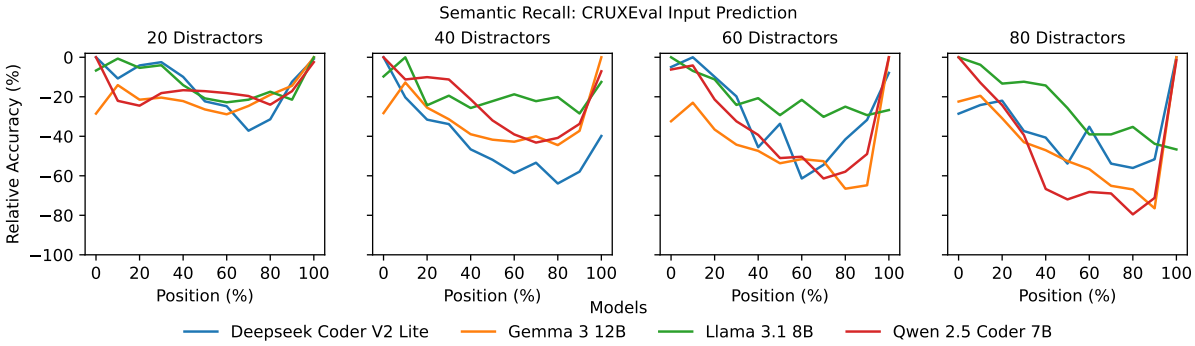

Figure 7: `CRUXEval` input prediction (`CRUXEval-I`) positional effects for smaller models. Performance patterns mirror those observed in `CRUXEval` output prediction (Figure 7b), with moderate lost-in-the-middle effects becoming more pronounced as context length increases.

B `CRUXEval` Input Prediction (`CRUXEval-I`) Results

We present complete results for `CRUXEval` input prediction (`CRUXEval-I`) to complement the output prediction (`CRUXEval-O`) results shown in § 5.1. Figure 6 and Figure 7 demonstrate that models exhibit nearly identical positional sensitivity patterns on input prediction as they do on output prediction (Figure 2b and Figure 5b, respectively). Both tasks show moderate lost-in-the-middle effects, with performance degrading as target code moves toward central positions, particularly at the 60-80% position range. This similarity is expected given that both tasks measure semantic recall on the same `CRUXEval` functions and are equally susceptible to pattern matching shortcuts, as confirmed by our counterfactual measurement (Figure 3).

C `SemTrace` Detailed Analysis

Partial Match Accuracy To further investigate whether the severe accuracy drops on `SemTrace` reflect complete task breakdown or gradual semantic recall failures, we measure partial match accuracy: counting how many individual list positions models predict correctly rather than requiring the entire output to be exact. Figure 8 reveals that partial match accuracy follows qualitatively similar positional patterns to exact match accuracy (Figure 2c), with performance degrading as target code moves toward central positions. However, the degradation curves are notably smoother, indicating that many errors in exact match accuracy stem from partial failures where models correctly recall some assignment lines but not others. This pattern confirms that position-dependent degradation reflects gradual semantic recall failures rather than complete comprehension breakdown, as models maintain the ability to semantically recall and apply some–but not all–assignment operations even under challenging positional conditions.

Resolution Errors Despite explicitly prohibiting partial results in the prompt, we find that several models still at times generate answers such as $81 - 43$ instead of 38. Since these expressions are interpreted by Python as equivalent, we don't penalize such responses. However, our supplementary analysis (Figure 9) reveals that especially Gemma and Llama family models provide these "unresolved" outputs much more frequently when the target function appears near the middle of the input context. We hypothesize that this represents an attempt to circumvent semantic recall limitations through lexical recall.

D Model and Dataset Licenses

We include the licenses for models and datasets in Table 3 and Table 4, respectively.

Table 3: Model licensing information.

Model Name	License
Codestral 22B	MNLP-0.1
GPT 4.1	Proprietary
DeepSeek Coder V2	Deepseek-license
DeepSeek Coder V2 Lite	Deepseek-license
Llama 3.1 8B	Llama3.3
Llama 3.3 70B	Llama3.3
Qwen 2.5 Coder 32B	Apache-2.0
Qwen 2.5 Coder 7B	Apache-2.0
Gemma 3 12B	Gemma
Gemma 3 27B	Gemma

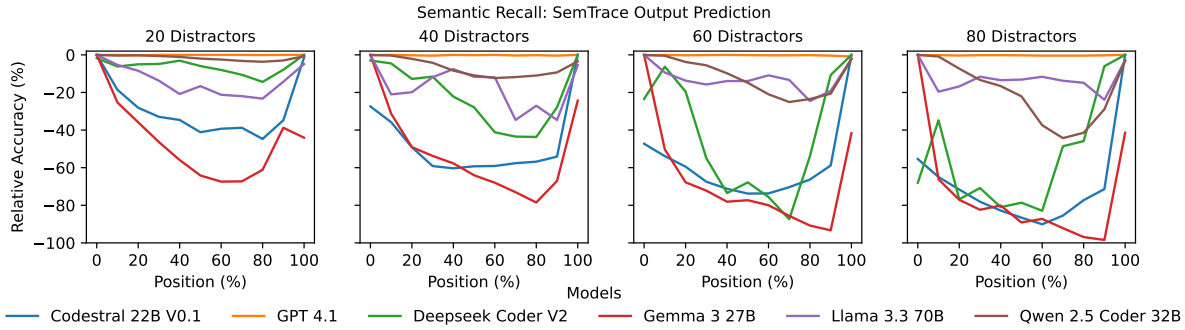

Figure 8: Partial match accuracy on `SemTrace` across code positions, measuring the percentage of list elements predicted correctly rather than requiring exact full-output matches. Despite following similar positional trends to exact match accuracy (Figure 2c), degradation curves are smoother, indicating that accuracy drops result from gradual semantic recall failures where models correctly process some assignment lines but not others, rather than complete task comprehension breakdown.

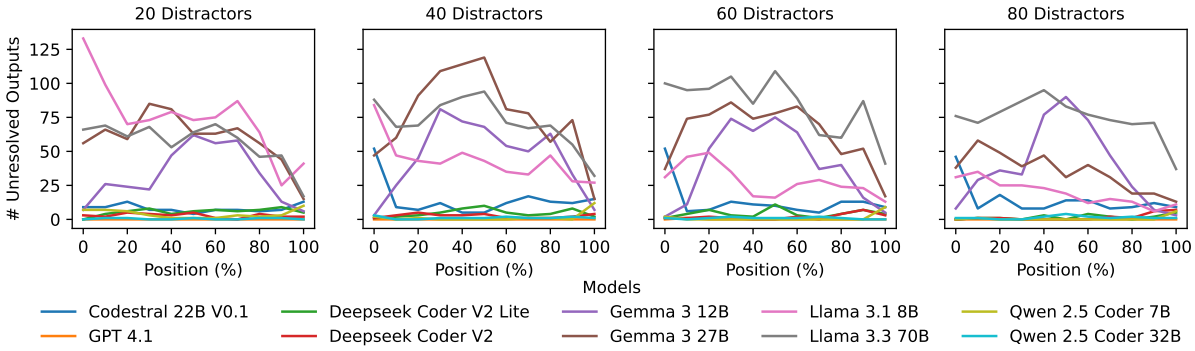

Figure 9: Number of unresolved outputs on `SemTrace`. Unresolved outputs are expressions like $81 - 43$ instead of the computed result 38. Models from the Gemma and Llama families produce substantially more unresolved outputs when target functions appear near the middle of the context, suggesting an attempt to circumvent semantic recall limitations through lexical recall of the operations without performing the computation.

Table 4: Dataset licensing information.

Dataset Name	License
CRUXEval	MIT License
CodeSearchNet	MIT License

E Prompt Templates

In this section we include templates for input prediction (Figure 10), output prediction (Figure 11), and function-level retrieval (Figure 12) tasks. We leverage assistant prefill such that each model provides a predictable and easy-to-parse response.

F Detailed Setup Information

Machine Details. We performed all experiments using AWS EC2 `p5e.48xlarge` instance equipped with 192 cores, 2048GB RAM, and eight NVIDIA H200 GPUs on Ubuntu 22.04 with CUDA 12.4. To serve open-source LLMs, we use vllm 0.7.1 (Kwon et al., 2023). We access GPT 4.1 through OpenAI API.

Execution Time. For the final evaluation runs, we spent total of 47 GPU-days on open-source model inference. We have spent about 2 days running experiments on GPT 4.1 through OpenAI API. We estimate that total usage, including reruns and development, might be 1.5-2 times higher than our evaluation runs.

```
User:
You are given a number of Python functions
and an assertion containing an output of
one of the functions. Find any input such
that executing that function on the input
leads to the given output. There may be
multiple answers, but you should only
output one.

[ASSERTION]
assert {output} == f(??)
[/ASSERTION]

[FUNCTIONS]
{code_block}
[/FUNCTIONS]

[ASSERTION]
assert {output} == f(??)
[/ASSERTION]

You are given a number of Python functions
and an assertion containing an output of
one of the functions. Find any input such
that executing that function on the input
leads to the given output. There may be
multiple answers, but you should only
output one.

Assistant:
Sure! Here is the corresponding input:

“‘python
assert {output} == f(
```

Figure 10: Input Prediction Chat Completion Prompt Template.

```
User:
You are given a number of Python functions and an assertion containing an input to one of the functions. Complete the assertion with a literal (no unsimplified expressions, no function calls) containing the output when executing the provided code on the given input, even if the function is incorrect or incomplete. Do NOT output any extra information.

[ASSERTION]
assert f({input}) == ??
[/ASSERTION]

[FUNCTIONS]
{code_block}
[/FUNCTIONS]

[ASSERTION]
assert f({input}) == ??
[/ASSERTION]

You are given a number of Python functions and an assertion containing an input to one of the functions. Complete the assertion with a literal (no unsimplified expressions, no function calls) containing the output when executing the provided code on the given input, even if the function is incorrect or incomplete. Do NOT output any extra information.

Assistant:
Sure! Here is the corresponding output:

“‘python
assert f({input}) ==
```

Figure 11: Output Prediction Chat Completion Prompt Template.

```
User:
Each line in the code block below starts with a random key. I’m looking for a function starting at key ‘{start}‘ and ending at key ‘{end}‘ in the code snippet below. Can you help me find it?

“‘python
{code_block}
“‘

Each line in the code block above starts with a random key. I’m looking for a function starting at key ‘{start}‘ and ending at key ‘{end}‘ in the code snippet above. Can you help me find it?

Assistant:
Sure! Here is the full function starting at key ‘{start}‘ and ending at key ‘{end}‘:

“‘python
```

Figure 12: Function-level Retrieval Chat Completion Prompt Template.