Evaluating Long Range Dependency Handling in Code Generation LLMs

Anonymous authors
Paper under double-blind review

Abstract

As language models support larger and larger context sizes, evaluating their ability to make effective use of that context becomes increasingly important. We analyze the ability of several code generation models to handle long range dependencies using a suite of multi-step key retrieval tasks in context windows up to 8k tokens in length. The tasks progressively increase in difficulty and allow more nuanced evaluation of model capabilities than tests like the popular needle-in-the-haystack test. We find that performance degrades significantly for many models (up to 2x) when a function references another function that is defined later in the prompt. We also observe that models that use sliding window attention mechanisms have difficulty handling references further than the size of a single window. We perform simple prompt modifications using call graph information to improve multi-step retrieval performance up to 3x. Our analysis highlights that long-context performance needs more consideration than just retrieval of single facts within a document.

1 Introduction

Long context inference is an increasingly important differentiator of LLMs, with model releases supporting larger and larger context windows (Anthropic, 2023; OpenAI, 2023; Google, 2024; Jiang et al., 2023; Touvron et al., 2023). This is enabled by advances in efficient attention implementation such as FlashAttention (Dao et al., 2022), Grouped-Query Attention (Ainslie et al., 2023) and Paged Attention (Kwon et al., 2023), as well as scaling attention by introducing sparsity via windowing (Child et al., 2019; Beltagy et al., 2020).

Applications, such as in-editor code completion from tools like GitHub's Copilot (Choi, 2024) and SourceGraph's Cody (Isken & Hill, 2024), benefit from long context support as they leverage retrieval-augmented generation strategies (Gao et al., 2024) to incorporate code snippets from across the user's project into a single prompt. This allows completions to be driven by the user's context rather than just by what the model learned at training time. Works such as Zhang et al. (2023) and Shrivastava et al. (2023) have proposed various approaches to improve cross-repository code completion.

These applications rely heavily on the model's ability to effectively use everything in its context window, and thus evaluation of long context usage has been a topic of interest in the community. Khandelwal et al. (2018) showed that early LSTM-based language models only roughly modelled information from more distant tokens. Recently, the "needle-in-the haystack" test, which measures recall for chat-style LLMs (Kamradt, 2024) has grown popular, and a parallel key-retrieval method has been used by Rozière et al. (2024) to measure recall in code generation models. However, we believe that desirable long context inference also includes the ability to reason over multiple pieces of information in the input, and needle-in-haystack evaluation approaches do not capture a model's ability to do this. In the paper we contribute:

1. A set of multi-step key retrieval tasks that progressively increase in difficulty and allow evaluation of long-range dependency handling and basic reasoning through function calls.
2. Empirical evaluation of several open source and proprietary code generation models. We find that model performance varies greatly depending on the number of steps involved, and the distinctiveness of the target fact compared to the rest of the context. We also discover that the order of function

declarations has a large effect on model ability to complete these tasks. We further observe that sliding window mechanisms degrade models' ability to resolve references beyond the size of the window.

3. An investigation of methods for improving multi-step recall that rely only on prompt modification using information obtainable via existing non-LLM techniques, in particular we add annotations of function dependencies in the form of comments.

2 Related Work

Symbolic reasoning: Zhang et al. (2022) study neural networks' ability to handle indirection by introducing the Pointer Value Retrieval (PVR) task. They train models to take in a sequence of tokens (typically digits), where the first token acts as pointer to the value to be retrieved. The synthetic nature of this task provides full control over the complexity of the problem. Abnar et al. (2023) extend this work to add recursive steps to the PVR task, evaluate it in the context of vision transformers, and present a new architecture that supports adaptive compute.

Multi-hop QA: Min et al. (2019) study a popular multi-hop reasoning benchmark (HotpotQA) and find that many of the questions can be answered with single hop reasoning. They find many questions embed contextual info that make the task easier, and that weak distractors may make picking the right answer 'obvious' without need for the desired reasoning steps. Chen & Durrett (2019) further explore dataset design choices to induce multi-hop reasoning.

Long context retrieval: Liu et al. (2024) Examine instruction-tuned LLM performance for synthetic key-retrieval tasks as well as multi-document QA and describe a "lost-in-the-middle" phenomenon where information becomes harder to retrieve if it is not near the beginning or end of the prompt.

Long context dependencies: Yu et al. (2024) present a code completion benchmark named CoderEval, where problem solutions have varying levels of dependencies on code in the surrounding context. They find that model performance for functions with dependencies is significantly worse than performance for standalone functions. In the context of natural-language Kuratov et al. (2024), Yuan et al. (2024) and Levy et al. (2024) develop benchmarks to evaluate how well language models extract and reason over distributed facts in the presence of noise text. Hsieh et al. (2024) develop a rich set of tasks to evaluate long context performance with different dependency structures in natural language.

Chain-of-thought prompting: Wei et al. (2023) and Kojima et al. (2023) explore prompting strategies to induce multi-step reasoning in instruction tuned models. They find that models perform better on various tasks when prompted to output intermediate steps in the reasoning chain.

While most of the existing literature focuses on chat-style natural language generation, our work focuses on autocomplete-style code generation, where latency requirements often constrain the use of 'scratchpad' methods such as chain-of-thought to improve reasoning. We situate our work in between the PVR work of Zhang et al. (2022), which is highly controllable but uses a fairly abstract task, and the code-with-dependencies benchmarking work exemplified by Yu et al. (2024), which is more realistic but also makes it more difficult to run controlled experiments that allow discovery of specific failure modes.

3 Long Context Multi-Step Key Retrieval

3.1 Task Design

To study long-range dependency handling we propose four multi-step key retrieval tasks of increasing complexity (one-step, two-step, three-step, and concatenation retrieval). These extend the key-retrieval task in Rozière et al. (2024) and test models' ability to integrate multiple pieces of information spread throughout a long context window to make a completion.

One-step retrieval is equivalent to the key-retrieval task in Rozière et al. (2024). We differ from their design by using random strings to construct function names rather than fixed function names, and return

(a) One Step

```
# ...
def key():
    return "xdfgew"
# ...
assert key() ==
```

(b) Two Step

```
# ...
def value():
    return "xdfgew"
# ...
def key():
    return value()
# ...
assert key() ==
```

(c) Three Step

```
# ...
def value_2():
    return "xdfgew"
# ...
def value_1():
    return value_2()
# ...
def key():
    return value_1()
# ...
assert key() ==
```

(d) Concatenation

```
# ...
def value_1():
    return "xdfgew"
# ...
def value_2():
    return "asdahj"
# ...
def key():
    return value_1() +
     value_2()
# ...
assert key() ==
```

Figure 1: Key retrieval tasks with increasing levels of difficulty. Function names and return values are randomized in the actual prompt. See Appendix G for a complete example.

values that are string literals rather than integer literals. Using a simple template system, we first generate a *key function* that returns a random string. The model is then asked to complete an *assert* statement on the return value of this function. All return strings that are 10 characters long and all function names are between 13 and 20 characters long.

In **two-step retrieval**, the key function calls a *value function* that returns the string. **Three-step retrieval** adds an additional function call between the key function and the value function that returns the string. In the **concatenation retrieval** task the key function calls two value functions and returns the concatenation of their returned strings Figure 1.

To turn each of these into a long context problem we insert varying amounts *irrelevant* code into the context window, we detail this in Section 3.3. In all cases the assert statement is on the result of the key function and is placed at the end of the prompt.

3.2 Avoiding Reliance on Parametric Knowledge and Trivial Solutions

To make sure the model is not solely relying on parametric knowledge (i.e., knowledge stored in the weights) to solve the task, we construct function names from two or three random sequences of lowercase characters or digits, joined with underscores (e.g., `zcxjdz_309521_xcdgfp`). Return values are random strings of lowercase characters (e.g., `pczjdfeyxc`). This makes it extremely unlikely that such functions exist in the model's training data, and therefore to complete the task successfully, the model must use the information from the prompt.

In our early experiments we observed that models may produce trivial solutions such as `assert foo() == foo()`. While these are technically correct , such solutions prevent us from testing the models' ability to retrieve information from the context. To prevent this, we *guide the decoding* of tokens to make sure that the response starts as a string literal[1]. As we sample output tokens, we use the `prefix_allowed_tokens` functionality in the HuggingFace transformers library Wolf et al. (2020) to ensure that the output starts with any number of spaces followed by a single or a double quote. Once an initial quote mark is produced, generation proceeds unrestricted.

3.3 Long Context Construction

To turn the tasks described above into long context inference problems we add irrelevant snippets to the context window from two sources. Motivated by findings in Min et al. (2019) and Chen & Durrett (2019) that show that the presence of good distractor functions are important when measuring multi-step reasoning, we first generate a number of synthetic *distractor* functions that use the same template as the the key and value functions that contain the task information. Then we sample standalone Python functions from the

[1]We are not able to do this guided decoding for the hosted GPT4o models

HumanEval dataset (Chen et al., 2021) to fill out the context window to our desired size. Algorithm 1 describes this dataset generation process in more detail.

Algorithm 1 Generate Long Context Retrieval Tasks

$n_k \leftarrow$ number of unique key functions
$n_d \leftarrow$ number of synthetic distractor functions
$n_t \leftarrow$ maximum number of tokens
$n_p \leftarrow$ maximum number of position combinations
repeat n_k **times**
  snippets $\leftarrow$ Generate key and value function(s) for task
  assert $\leftarrow$ Generate assert statement
  irrelevant $\leftarrow$ Generate n_d distractor functions + randomly sampled functions from HumanEval such than TokenCount(*snippets* + *irrelevant* + *assert*) $\lesssim n_t$.
  positions $\leftarrow$ All Len(*snippets*) combinations of integers in $1 \ldots$ Len(*snippets* + *irrelevant*)
    ▷ If $> n_p$ randomly sample n_p of these combinations
  for each *position combination* in *positions* **do**
    for each permutation of *snippets* **do**
      prompt $\leftarrow$ *irrelevant*
      Insert each function in *snippets* into *prompt* using the positions in *position combination*
      Add *prompt* + *assert* to the list of generated prompts
end

This generates a dataset that allows us to compare the effect of the following variables: **position** of task related snippets within the input context, **relative order** of the task-related snippets, and the **spread** of task related snippets. We vary these factors while keeping the irrelevant functions used constant for each key function we generate. To aid reproducibility we fix the random seed during generation so that the prompt set can easily be re-generated.

In our experiments, we set n_k to 100 for one-step task, and 20 for other tasks; n_d to 0, 1, and 5 for three distractor conditions; n_p to ∞, 150, 50, 50 for one-step, two-step, three-step, and concatenation tasks respectively (this results in a maximum of 6000 prompts for each condition). We set n_t to 2k, 4k and 8k tokens.

3.4 Models

We evaluate these tasks on six open source models: StarCoderBase-1B, StarCoderBase-7B, StarCoderBase-15.5B (Li et al., 2023), StarCoder2-7B (Lozhkov et al., 2024), Mistral-7B-v0.1 (Jiang et al., 2023) and DeepSeekCoder-6.7B-base(Guo et al., 2024). We also evaluate on GPT-4o-mini and GPT-4o (OpenAI, 2024). The StarCoder family of models are competitive on code generation benchmarks and available in a number of different parameter counts. StarCoder2-7B and DeepSeekCoder-6.7B-base models have been pre-trained on "repository level" data with the explicit aim of improving performance in contexts where there are dependencies between code units. Mistral is a general purpose LLM that is nonetheless competitive on code generation benchmarks. Similarly the GPT-4o models display strong performance on code generation tasks even though they are not specific to code. We use implementations from the HuggingFace transformers library(Wolf et al., 2020) for the open source models.

We are primarily interested in seeing how the performance of a given model *changes* as **task difficulty** (i.e., number of hops needed) and **context size** varies. We evaluate multiple models to see how consistent effects are across multiple models.

[1]Specifically gpt-4o-mini-2024-07-18 and gpt-4o-2024-11-20

[2]HumanEval scores for StarCoder* models are from Lozhkov et al. (2024), StarcoderBase-1B scores are from `https://huggingface.co/bigcode/starcoderbase-1b`. Mistral-7B scores are from Jiang et al. (2023). GPT scores are from `https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/`

[3]Mistral-7B was trained with 8192 context size but reports support for up to 131k tokens.

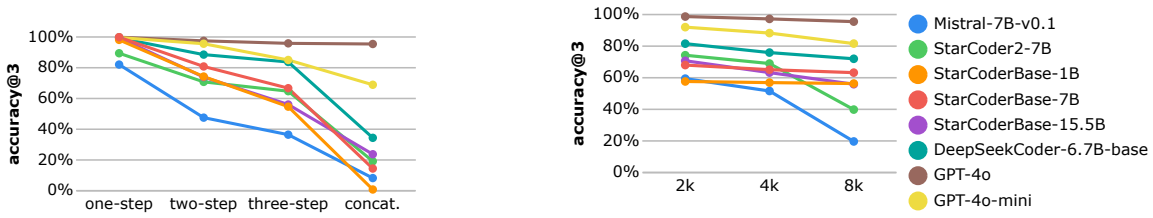
Table 1: Models evaluated.

Model	HumanEval[2]	Max Context Size[3]	Sliding Window Size
Mistral-7B	30.5	8192 / 131k	4096
StarCoder2-7B	35.4	16384	4096
StarCoderBase-1B	15.1	8192	N/A
StarCoderBase-7B	30.5	8192	N/A
StarCoderBase-15.5B	29.3	8192	N/A
DeepSeekCoder-6.7B-base	49.4	16384	N/A
GPT-4o-mini	87.2	128k	N/A
GPT-4o	90.2	128k	N/A

3.5 Results

3.5.1 Overall Task Performance

Our primary metric is **accuracy@k**, which is similar to the **pass@k** metric introduced in Chen et al. (2021), the only difference is instead of running unit tests, we simply check if the string literal produced is the expected string. We compute **accuracy@3** over **10** generations for each input prompt. Hyperparameters for generation are in Appendix D



(a) Task performance by model.

(b) Task performance by prompt length.

Figure 2: Overall task performance. For detailed scores, see Table 5 in appendix.

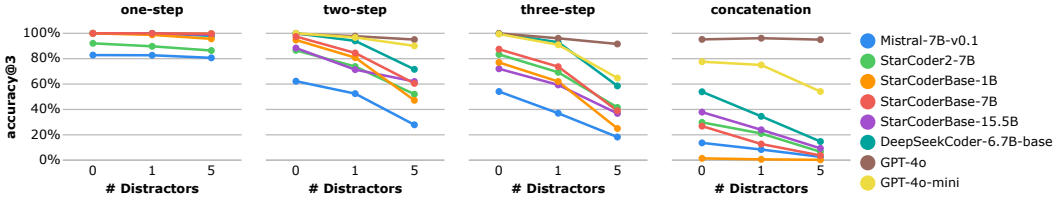


Figure 3: Effect of number of distractors by task variant. For detailed scores, see Table 6 in appendix.

Task difficulty: Figure 2a shows that tasks increase in difficulty in the following order: one-step, two-step, three-step, and concatenation. We note particularly weak performance on the concatenation task. Another factor affecting difficulty is the number of **distractor functions**. For the more difficult tasks we see a consistent degradation in performance as distractors are added (Figure 3).

Context size: In Figure 2b we observe a small performance drop as context sizes increases for StarCoderBase-1B, StarCoderBase-7B, StarCoderBase-15, DeepSeekCoder-6.7B, GPT-4o-mini and GPT-4o. For StarCoder2-7B and Mistral-8B, which both use sliding window attention, a large drop in performance occurs when moving from 4k to 8k context length, we hypothesize that this is related to the size of the sliding window used and will explore this in more detail in Section 3.5.3.

3.5.2 Incorrect Responses

We analyzed the incorrect responses from models and note a number of distinct failure modes. Firstly a noticeable amount of incorrect generations are return values of distractor functions. Considering the 1 and 5 distractor conditions, approximately 10% of all responses are distractor answers across models (min=1.8%, max=16.2%). Within just the incorrect responses, approximately 20% are distractors (min=10.9%, max=33.4%). Full details are in Table 7 and Table 8 in the appendix.

For the concatenation task a common failure mode is returning a partial answer (i.e. one of the two values to be concatenated). Approximately 33.1% of all incorrect responses errors for this task are partial answers from one of the two strings value functions. See Table 9 for details.

To get further insight into other incorrect responses we also conduct an edit distance analysis. This allows us to see if models are mostly getting the answer right and maybe failing on just a few tokens. We compute Levenshtein distance between model responses and ground truth answers for the incorrect responses and find the mean distance to be 10.97. We note that the length of the expected string is 11 in all tasks. This indicates that when the model is wrong it is generally completely wrong rather than just incorrectly generating one or two tokens. Details of this analysis can be found in Table 10

3.5.3 Effect of Task Snippet Position

We focus the rest of our analysis on the five distractor condition as it allows us to see a good variation in task performance.

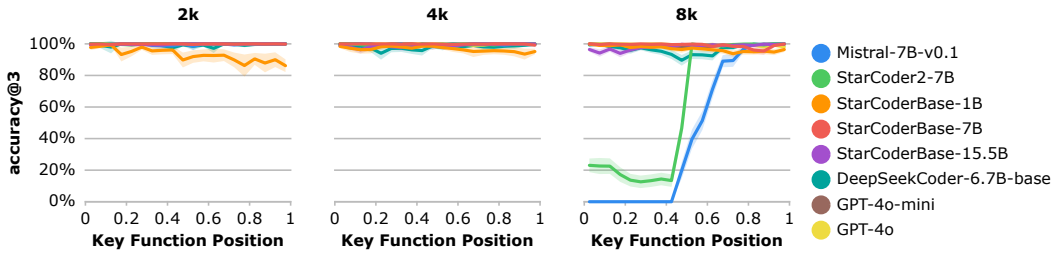


Figure 4: Effect of key function position on one-step task with 5 distractors. Position is defined as index of the first token in the function normalized by the total number of tokens in the prompt, and grouped into 20 bins. 0 is the beginning of the prompt and 1 is at the end.

In the **one-step** retrieval task we see good performance with respect to *key function position* across the entire context window with two notable exceptions. At the 8k context size **Mistral-7B-v0.1** and **StarCoder2-7B** are unable to perform the one-step task when the key function is more than ~4k tokens away from the generation site Figure 4. Both models use a sliding window attention mechanism with a 4k window size. While this mechanism theoretically allows the model to scale very large context sizes (Jiang et al. (2023) reports a theoretical attention span of 131k tokens), our results indicate that the model fails to retrieve precise information at a distance greater than the sliding window.

Mistral-7B-v0.1 allows for changing the sliding window size at run time. To futher confirm the result above, we measured one-step retrieval performance with sliding windows of 2048 and 8192 (in addition to the default 4096) at the 8k context size, and find that the drop in performance relative to key function position is generally consistent with this result. However we note that even at the 8192 sliding window size, performance begins to drops when the key function is approximately 60% of the size of the prompt away from the generation site, it however

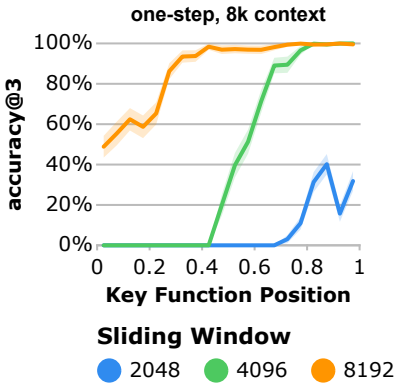


Figure 5: Varied sliding window sizes for Mistral-7B-v0.1 on the one-step task with 5 distractors.

does not drop to zero as in the 2048 and 4096 case. Figure 5 shows this in detail.

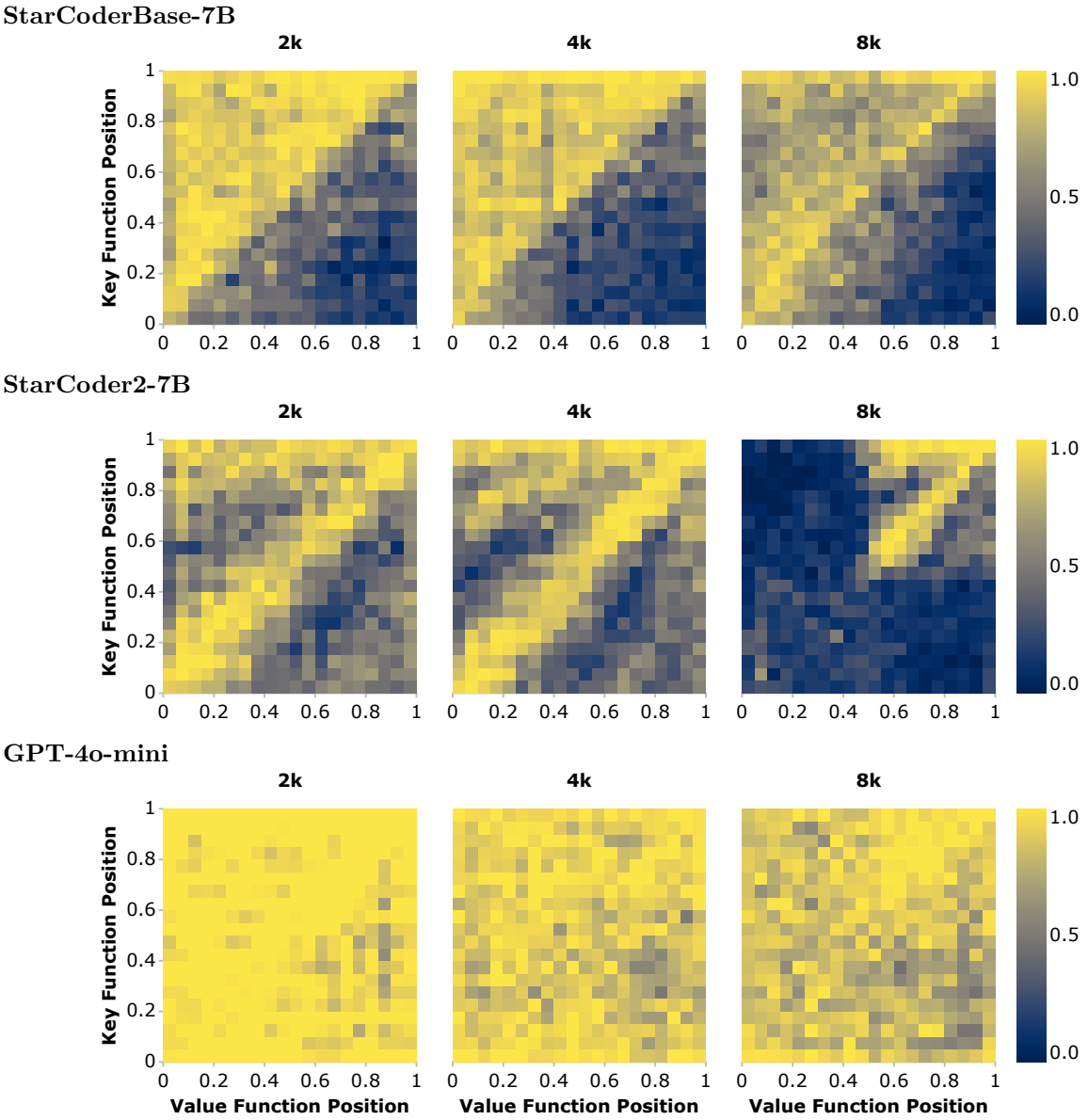


Figure 6: Effect of key and value function position for two-step task with 5 distractors. Top-to-bottom StarCoderBase-7B, StarCoder2-7B, GPT-4o-mini. Color represents accuracy@3 score.

The **two-step** task has two relevant task snippet positions, Figure 6 shows heatmaps of task performance vs. both key function position and value function position. We see that performance is worse when the key function *appears before* the value function in the prompt for the first two models. We call this a **forward reference** and it turns out this has a large effect on performance in most models. We also observe that performance is generally higher along the diagonal, i.e., when the functions are closer to each other. We elaborate on this further in Section 3.5.4

For **three-step** and **concatenation** tasks, it becomes harder to directly analyze position effects, as there are three task-relevant code snippets with independent positions. We instead explore performance by number forward references, and by spread between task-relevant snippets in the following sections.

3.5.4 Effect of Forward References

Table 2: Model accuracy (%) vs. number of forward references. 5 distractors. Performance generally drops as number of forward references increases. Blue indicates exceptions to this behavior, indicating a condition that performs worse than the condition with one more forward reference.

Two-step Task

Context Size		2k		4k		8k
# Forward References	0	1	0	1	0	1
Mistral-7B-v0.1	60.1	21.1	46.4	22.5	8.0	9.2
StarCoder2-7B	73.5	57.6	69.4	56.5	28.1	26.8
StarCoderBase-1B	55.3	39.4	49.7	33.9	65.6	39.3
StarCoderBase-7B	90.0	38.1	87.2	34.1	76.0	37.7
StarCoderBase-15.5B	85.5	63.4	82.3	49.2	63.4	28.3
DeepSeekCoder-6.7B-base	83.7	75.2	74.4	67.7	70.2	58.5
GPT-4o-mini	98.4	95.7	91.4	87.2	87.4	80.3
GPT-4o	99.8	98.3	98.6	93.6	96.2	83.9

Three-step Task

Context Size			2k			4k			8k
# Forward References	0	1	2	0	1	2	0	1	2
Mistral-7B-v0.1	28.1	24.4	16.8	19.8	22.3	16.9	2.7	11.7	10.3
StarCoder2-7B	61.5	60.5	45.7	46.2	48.1	37.5	14.4	23.1	16.0
StarCoderBase-1B	32.3	29.5	18.1	29.5	28.5	14.9	19.8	23.8	8.2
StarCoderBase-7B	63.4	41.3	18.8	65.4	42.0	20.5	41.9	35.6	15.9
StarCoderBase-15.5B	57.2	50.6	37.2	54.0	39.7	24.9	34.5	21.0	10.5
DeepSeekCoder-6.7B-base	69.4	73.1	56.2	52.2	61.3	52.6	42.9	50.5	40.1
GPT-4o-mini	68.0	81.8	70.9	53.3	67.1	62.8	39.4	56.3	50.4
GPT-4o	95.7	96.7	89.6	88.7	91.3	86.6	85.2	91.0	86.8

Concatenation Task

Context Size			2k			4k			8k
# Forward References	0	1	2	0	1	2	0	1	2
Mistral-7B-v0.1	11.8	4.3	3.1	2.4	0.6	0.6	0.8	0.2	0.2
StarCoder2-7B	17.7	14.7	7.9	6.4	5.4	4.2	1.8	0.7	0.7
StarCoderBase-1B	1.6	0.3	0.0	0.5	0.1	0.0	0.0	0.0	0.0
StarCoderBase-7B	13.9	9.3	4.0	2.0	1.6	1.2	0.6	0.6	0.3
StarCoderBase-15.5B	22.9	14.2	11.0	10.1	6.1	5.8	6.8	4.5	2.6
DeepSeekCoder-6.7B-base	40.3	24.7	20.2	14.2	8.5	7.2	9.6	4.6	3.2
GPT-4o-mini	79.8	71.2	63.6	64.9	52.7	45.5	46.1	35.4	27.7
GPT-4o	98.5	96.7	97.3	97.4	96.0	94.9	93.6	91.7	88.9

We define a **forward reference** as a function calling a not-yet-defined function (i.e. one that will be defined later in the prompt). We find that forward references have a negative impact on task performance in all models, in most conditions we tested. The exceptions to this are highlighted in blue in Table 2, and mainly occur in the three-step task where we sometimes observe an increase in performance when going from zero to one forward reference followed by a drop going from two to three forward references. Table 2 shows these results in detail. We see drops as large as 44.9% in models like StarCoderBase-7B (three-step task at 4k), 19.4% for GPT-4o-mini (concatenation task at 4k) and 12.3% for GPT4o (two-step task at 8k).

3.5.5 Effect of Task Snippet Spread

We saw in Figure 6 that the order of functions in multi-step tasks and the distance between functions affects performance. We define *spread* as the distance between the first token of the first task-related snippet and the last token of the last task-related snippet, *not including the assert statement* at the end of the prompt.

In the two-step task we broadly see three behaviors with respect to spread, the GPT4o models show little variation as spread increases. DeepSeekCoder-6.7B-base, StarCoder2-7B and Mistral-7B-v0.1 show performance drops at moderate amount of spreads but perform well when snippets are very close to each

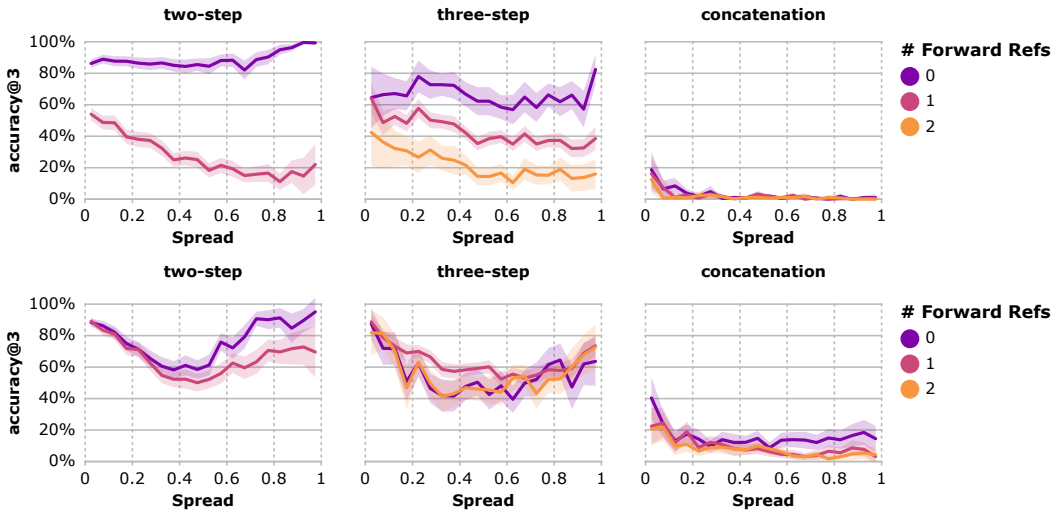


Figure 7: Effect of spread (normalized to the number of tokens in the context) for StarCoderBase-7B (top) and DeepSeekCoder-6.7B-base (bottom) with 4k context size and 5 distractors. For data on other models and context sizes, see Figure 10, Figure 11, and Figure 12.

other or very far from each other. Snippets with maximal spread are located close to the beginning and end of the prompt and result in higher performance. The other models (StarCoderBase-1B, StarCoderBase-7B, and StarCoderBase-15B) show a drop in performance as spread increases, but only in the presence of forward references. Figure 7 shows the effect of task snippet spread for StarCoderBase-7B DeepSeekCoder-6.7B-base with 4k context length on the two-step retrieval task with results for the other models in the appendix (Figure 10).

For the three-step task similar, effects are clearly visible at 2k and 4k context size but are as not consistent across different models Figure 11. For the concatenation task, the effect is less pronounced as model's performance is generally low regardless of spread and the number of forward references and there is not much room for separation Figure 12.

4 Improving Retrieval Performance with Call Graph Comments

We saw in Section 3.5.4 that performance degrades for the multi-step tasks in the presence of forward references. We considered if injecting information about function call relationships into the prompt would help performance. This information is easy to obtain from static analysis of code (or in our case at prompt construction time), and if beneficial, provides a lightweight approach to improving performance at inference time. We focus on our most difficult tasks, namely the three-step and concatenation retrieval task with 5 distractors using the same models as in our previous experiment.

4.1 Experiment Setup

We add comments above each function that contain a lightweight description of the *call graph* associated with that function. We can construct these comments by annotating which functions are **"called by"** other functions, or annotating which functions **"call"** a given function, or a combination of **both**. In the three-step retrieval task we can do this transitively and add annotation of *all the functions* that will be called to compute the result of a given function.

We considered two templates for these comments, in one variation ("names only") we simply included a comma-separated list of function names *and no other information*. In the other ("full sentence") we use the phrases: *"This function is called by "* and *"This function calls "* followed by the comma-separated list of function names. See Appendix H for examples.

4.2 Results

Table 3: Call graph comment performance across all models. Accuracy@3

(a) By template

Template	Comment Types		
	Calls	Called By	Both
Full-Sentence	41.4	53.5	**56.6**
Names-Only	36.0	51.7	**49.4**

(b) By task (full sentence template only)

Task	Comment Types			
	None	Calls	Called By	Both
Three-step	46.9	55.3	74.2	**76.8**
Concatenation	23.3	27.6	32.8	**36.4**

We observe that the addition of call-graph comments has a positive effect on task performance. Table 3b shows a ~1.5x improvement on the concatenation task, and a ~1.6x improvement on the three-step retrieval task with the addition of call-graph comments aggregated across models. We do note that StarCoderBase-1B sees no improvement on the concatenation task. See Appendix C for detailed results.

Full sentence vs. function names only: The *full-sentence* template performs better than the *names-only* version (Table 3a). This suggests that the models are able to take some advantage of the more complete natural language description and that there may be room for further improvement by tuning the template. However it is notable that simply including the names of the referenced functions before they function definition has such a large performance boost. We focus the rest of our analysis on the full-sentence condition as it was the best performing template.

Call graph directions: Table 3b shows that while most of the benefit comes from the "X is called by Y" type of comment, using both directions produces the best improvement overall.

Full graph vs. next-hop comments: We noted earlier that in the three-step case we add comments describing the full call graph (i.e., up to two steps away). We experimented with only adding comments about the *next function in the call graph* and found that the full graph version performs best (Table 4).

Table 4: Call graph comment performance by depth across all models. Accuracy@3

# Forward References	Comment Type		
	None	Next Hop	Full
0	48.6	70.4	**73.9**
1	48.8	71.9	**79.0**
2	37.8	58.9	**71.0**

Context size, spread, and forward references: Figure 8 shows that call-graph comments improve performance across all context lengths, number of forward references, and task-snippet spreads for StarCoderBase-7B (this trend holds for other models where the call graph comments make a difference). Full plots for all the models are found in Appendix C. We note that with respect to *spread* for the concatenation task, most of the improvement appears when the task-relevant snippets are closer together.

5 Discussion

We found that models were able to perform well at the simpler one-step recall task over their entire context lengths (Figure 2a). However our experiments highlight the importance of distinguishing between *long-context support* and the ability to handle *long-range dependencies*. This is illustrated by models like Mistral and StarCoder2 that appear to make a trade-off between the two. While sliding window attention greatly expands their supported context sizes, our experiments show that when distance between relevant snippets is greater than the sliding window size used by these models, performance degrades quickly. Our tasks require precise recall over a long distance and it appears to be difficult to pass on information from earlier sliding windows in a way that allows successful completion of the task.

As different approaches to scaling attention to larger context windows proliferate, practitioners should take care to understand and mitigate these effects if their task relies on precise recall of long-range dependencies.

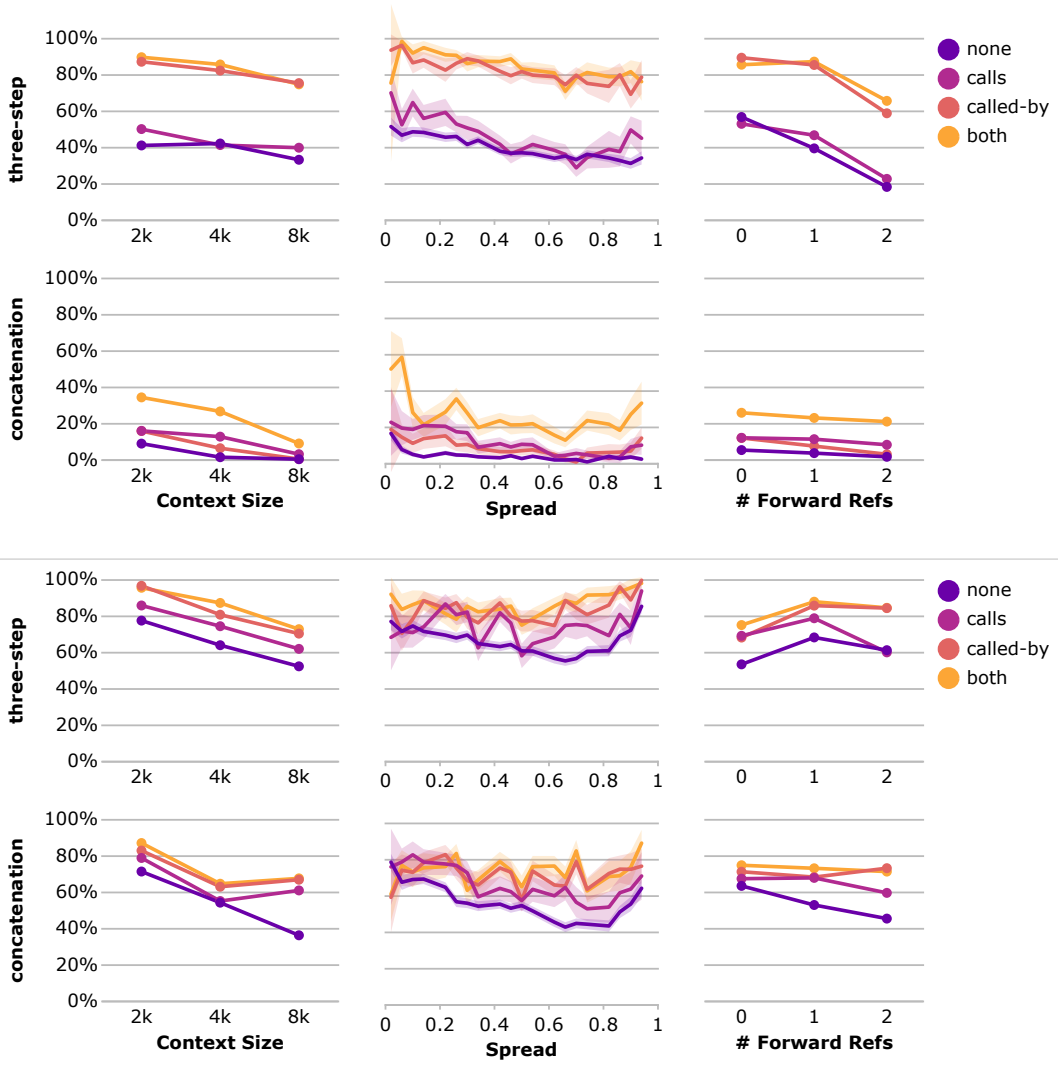


Figure 8: Effect of call graph comments on StarcoderBase-7B (top) and GPT-4o-mini (bottom) performance across context sizes, spread of task snippets and number of forward references.

In the context of code generation, builders of retrieval augmented generation systems may find it desirable to rewrite code to bring code snippets that depend on each other closer together.

Our multi-step tasks test the ability of models to reason through indirection introduced by simple yet common operations like function calling and concatenation. We find that models perform significantly worse on these tasks than the simpler one-step recall task (Figure 2a). This is particularly true in the presence of distractors, which we believe are important for appropriately pressure testing the ability of models to do the required reasoning (Figure 3).

We note that models we tested struggled even more in the presence of forward references (Table 2). In this situation all the information in the prompt is the same, but is presented in a different order. Humans and compilers are able to resolve this problem and it would be ideal if models could do the same. While we do not know what causes this behaviour, one hypothesis is that the causal attention used during training favors attending backwards in the context window and propagating information from earlier to later tokens.

Finally we found that adding simple natural-language text annotations that describe the call graph, or even just mention referenced function names before defining a function, greatly improves performance on the more

difficult multi-step tasks in all cases, including where forward references are present. Most of the benefit seems to come from just having the function name of the forward-reference appear first, though there also appears to be room to explore tuning the comment text to improve performance (Table 3).

For researchers these findings shed light on challenges LLMs face when modeling dependency relationships between tokens. Future work could investigate what mechanisms causes the discrepancy in performance when dependent function orders are changed.

These findings suggest a couple of avenues for practitioners to explore when working with models that exhibit similar failures to those demonstrated in this paper. Firstly it may be possible to rewrite code to reduce forward references while maintaining the semantics of the code, and secondly that extra information that can be gleaned from existing methods such as static analysis can be injected as comments to help models reason over the code.

6 Limitations

Our experiments uses synthetically generated functions that are not as realistic as real code in the wild. Suggestions such as re-ordering functions to bring dependent code closer together or adding annotations like call-graph comments would need to be tested in real retrieval-augmented code generation environments. We do not do in-depth investigation of potential internal mechanisms that lead to the behaviors we observed, we describe some hypotheses but otherwise leave that to future work.

References

Samira Abnar, Omid Saremi, Laurent Dinh, Shantel Wilson, Miguel Angel Bautista, Chen Huang, Vimal Thilak, Etai Littwin, Jiatao Gu, Josh Susskind, and Samy Bengio. Adaptivity and Modularity for Efficient Generalization Over Task Complexity, October 2023. URL http://arxiv.org/abs/2310.08866. arXiv:2310.08866 [cs].

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints, December 2023. URL http://arxiv.org/abs/2305.13245. arXiv:2305.13245 [cs].

Anthropic. Introducing 100K Context Windows, May 2023. URL https://www.anthropic.com/news/100k-context-windows.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The Long-Document Transformer, December 2020. URL http://arxiv.org/abs/2004.05150. arXiv:2004.05150 [cs].

Jifan Chen and Greg Durrett. Understanding Dataset Design Choices for Multi-hop Reasoning. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4026–4032, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1405. URL https://aclanthology.org/N19-1405.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, July 2021. URL http://arxiv.org/abs/2107.03374. arXiv:2107.03374 [cs].

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating Long Sequences with Sparse Transformers, April 2019. URL http://arxiv.org/abs/1904.10509. arXiv:1904.10509 [cs, stat].

Nicole Choi. What is retrieval-augmented generation, and what does it do for generative AI?, April 2024. URL https://github.blog/2024-04-04-what-is-retrieval-augmented-generation-and-what-does-it-do-for-generative-ai/.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness, June 2022. URL http://arxiv.org/abs/2205.14135. arXiv:2205.14135 [cs].

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-Augmented Generation for Large Language Models: A Survey, March 2024. URL http://arxiv.org/abs/2312.10997. arXiv:2312.10997 [cs].

Google. Our next-generation model: Gemini 1.5, February 2024. URL https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence, January 2024. URL http://arxiv.org/abs/2401.14196. arXiv:2401.14196.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What's the Real Context Size of Your Long-Context Language Models?, April 2024. URL http://arxiv.org/abs/2404.06654. arXiv:2404.06654 [cs].

Alex Isken and Corey Hill. How Cody understands your codebase, February 2024. URL https://sourcegraph.com/blog/how-cody-understands-your-codebase#.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7B, October 2023. URL http://arxiv.org/abs/2310.06825. arXiv:2310.06825 [cs].

Gregory Kamradt. GitHub - gkamradt/LLMTest_needleinahaystack, April 2024. URL https://github.com/gkamradt/LLMTest_NeedleInAHaystack.

Urvashi Khandelwal, He He, Peng Qi, and Dan Jurafsky. Sharp Nearby, Fuzzy Far Away: How Neural Language Models Use Context, May 2018. URL http://arxiv.org/abs/1805.04623. arXiv:1805.04623 [cs].

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large Language Models are Zero-Shot Reasoners, January 2023. URL http://arxiv.org/abs/2205.11916. arXiv:2205.11916 [cs].

Yuri Kuratov, Aydar Bulatov, Petr Anokhin, Dmitry Sorokin, Artyom Sorokin, and Mikhail Burtsev. In Search of Needles in a 11M Haystack: Recurrent Memory Finds What LLMs Miss, February 2024. URL http://arxiv.org/abs/2402.10790. arXiv:2402.10790 [cs].

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention, September 2023. URL http://arxiv.org/abs/2309.06180. arXiv:2309.06180 [cs].

Mosh Levy, Alon Jacoby, and Yoav Goldberg. Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models, February 2024. URL http://arxiv.org/abs/2402.14848. arXiv:2402.14848 [cs].

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder: may the source be with you!, 2023. URL https://arxiv.org/abs/2305.06161. Version Number: 2.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, February 2024. ISSN 2307-387X. doi: 10.1162/tacl_a_00638. URL https://doi.org/10.1162/tacl_a_00638.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder 2 and The Stack v2: The Next Generation, February 2024. URL http://arxiv.org/abs/2402.19173. arXiv:2402.19173 [cs].

Sewon Min, Eric Wallace, Sameer Singh, Matt Gardner, Hannaneh Hajishirzi, and Luke Zettlemoyer. Compositional Questions Do Not Necessitate Multi-hop Reasoning. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4249–4257, Florence, Italy, 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1416. URL https://www.aclweb.org/anthology/P19-1416.

OpenAI. New models and developer products announced at DevDay, November 2023. URL https://openai.com/blog/new-models-and-developer-products-announced-at-devday.

OpenAI. GPT-4o mini: advancing cost-efficient intelligence, July 2024. URL https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code Llama: Open Foundation Models for Code, January 2024. URL http://arxiv.org/abs/2308.12950. arXiv:2308.12950 [cs].

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-Level Prompt Generation for Large Language Models of Code. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 31693–31715. PMLR, July 2023. URL https://proceedings.mlr.press/v202/shrivastava23a.html. ISSN: 2640-3498.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton

Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models, July 2023. URL http://arxiv.org/abs/2307.09288. arXiv:2307.09288 [cs].

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models, January 2023. URL http://arxiv.org/abs/2201.11903. arXiv:2201.11903 [cs].

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. HuggingFace's Transformers: State-of-the-art Natural Language Processing, July 2020. URL http://arxiv.org/abs/1910.03771. arXiv:1910.03771 [cs].

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 1–13, February 2024. doi: 10.1145/3597503.3623322. URL http://arxiv.org/abs/2302.00288. arXiv:2302.00288 [cs].

Tao Yuan, Xuefei Ning, Dong Zhou, Zhijie Yang, Shiyao Li, Minghui Zhuang, Zheyue Tan, Zhuyu Yao, Dahua Lin, Boxun Li, Guohao Dai, Shengen Yan, and Yu Wang. LV-Eval: A Balanced Long-Context Benchmark with 5 Length Levels Up to 256K, February 2024. URL http://arxiv.org/abs/2402.05136. arXiv:2402.05136 [cs].

Chiyuan Zhang, Maithra Raghu, Jon Kleinberg, and Samy Bengio. Pointer Value Retrieval: A new benchmark for understanding the limits of neural network generalization, February 2022. URL http://arxiv.org/abs/2107.12580. arXiv:2107.12580 [cs, stat].

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.151. URL https://aclanthology.org/2023.emnlp-main.151.

A Appendix

A Detailed Results: Experiment 1

We include detailed tables of accuracy@3 scores (percent scale). Table 5 shows overall scores by task and context length (2k, 4k, and 8k); Table 6 shows scores by the number of distractors; and Table 2 shows scores by the number of forward references.

Table 5: Overall accuracy@3 scores for each model, task, and context size. Each score in the context length columns is the mean across all three distractor conditions. The "mean" column shows the average score for the three context sizes.

Task	Model	2k	4k	8k	mean
one-step	Mistral-7B-v0.1	99.8±0.6	99.8±0.6	46.4±0.5	82.0
	StarCoder2-7B	100.0±0.2	99.8±0.3	68.1±0.3	89.3
	StarCoderBase-1B	97.1±0.5	98.3±0.6	98.5±0.7	98.0
	StarCoderBase-7B	100.0±0.4	99.9±0.3	99.6±0.1	99.8
	StarCoderBase-15.5B	99.8±0.6	99.5±0.5	99.0±0.3	99.5
	DeepSeekCoder-6.7B-base	99.7±0.6	99.2±0.5	98.8±0.4	99.2
	GPT-4o-mini	99.9±0.1	99.8±0.1	99.3±0.0	99.6
	GPT-4o	100.0±0.5	99.9±0.4	99.6±0.3	99.9
two-step	Mistral-7B-v0.1	63.1±0.1	58.3±0.2	16.7±0.2	46.1
	StarCoder2-7B	81.9±0.0	82.6±0.1	46.4±0.1	70.3
	StarCoderBase-1B	73.3±0.1	73.5±0.1	73.7±0.1	73.5
	StarCoderBase-7B	81.6±0.1	80.1±0.1	79.2±0.9	80.3
	StarCoderBase-15.5B	82.2±0.0	74.7±0.1	64.0±0.8	73.7
	DeepSeekCoder-6.7B-base	90.1±0.1	88.0±0.1	85.6±0.1	87.9
	GPT-4o-mini	98.8±0.3	95.8±0.2	92.2±0.1	95.6
	GPT-4o	99.1±0.0	97.9±0.0	95.5±0.1	97.5
three-step	Mistral-7B-v0.1	53.1±0.4	41.5±0.4	14.7±0.5	36.5
	StarCoder2-7B	78.9±0.2	75.3±0.3	40.0±0.3	64.7
	StarCoderBase-1B	55.9±0.4	55.2±0.5	52.9±0.6	54.7
	StarCoderBase-7B	67.0±0.6	66.9±0.5	66.2±0.4	66.7
	StarCoderBase-15.5B	65.8±0.5	57.1±0.5	45.3±0.6	56.1
	DeepSeekCoder-6.7B-base	88.3±0.6	83.9±0.6	79.0±0.6	83.7
	GPT-4o-mini	90.8±0.6	86.5±0.6	77.8±0.6	85.0
	GPT-4o	97.9±0.6	95.6±0.5	94.1±0.6	95.9
concatenation	Mistral-7B-v0.1	17.3±0.4	6.7±0.4	0.7±0.4	8.2
	StarCoder2-7B	34.7±0.1	18.3±0.2	4.5±0.3	19.2
	StarCoderBase-1B	1.7±0.2	0.6±0.3	0.1±0.4	0.8
	StarCoderBase-7B	21.9±0.7	13.4±0.6	8.0±0.5	14.4
	StarCoderBase-15.5B	34.4±0.5	21.6±0.5	15.2±0.6	23.7
	DeepSeekCoder-6.7B-base	46.2±0.5	32.5±0.5	24.6±0.6	34.4
	GPT-4o-mini	78.6±0.6	70.9±0.5	57.2±0.5	68.9
	GPT-4o	97.9±0.5	95.5±0.5	92.9±0.5	95.4

Table 6: Overall model performance vs. number of distractors.

One-step Task

Context Size			2k			4k			8k
# Distractors	0	1	5	0	1	5	0	1	5
Mistral-7B-v0.1	100.0	99.9	99.6	100.0	100.0	99.5	48.5	48.2	42.8
StarCoder2-7B	100.0	100.0	99.9	100.0	100.0	99.6	76.3	69.3	59.8
StarCoderBase-1B	100.0	99.1	93.2	99.9	98.4	96.7	99.9	98.8	97.0
StarCoderBase-7B	100.0	100.0	99.9	100.0	100.0	99.9	100.0	99.7	99.1
StarCoderBase-15.5B	99.9	99.8	99.8	100.0	99.5	99.0	99.8	99.3	98.1
DeepSeekCoder-6.7B-base	100.0	100.0	99.3	100.0	100.0	98.0	100.0	99.7	96.8
GPT-4o-mini	100.0	99.6	100.0	100.0	99.7	99.7	100.0	98.7	99.2
GPT-4o	100.0	100.0	100.0	100.0	100.0	99.9	99.9	99.7	99.4

Two-step Task

Context Size			2k			4k			8k
# Distractors	0	1	5	0	1	5	0	1	5
Mistral-7B-v0.1	87.4	74.5	40.6	76.7	63.9	34.4	22.7	18.9	8.6
StarCoder2-7B	99.2	85.5	65.5	97.9	86.8	63.0	62.6	49.1	27.5
StarCoderBase-1B	98.5	80.9	47.3	96.2	82.5	41.8	89.5	79.2	52.4
StarCoderBase-7B	99.2	86.3	64.0	97.9	81.9	60.6	95.4	85.3	56.8
StarCoderBase-15.5B	93.6	80.9	74.4	89.6	68.8	65.8	82.0	64.3	45.8
DeepSeekCoder-6.7B-base	100.0	96.5	79.4	100.0	92.9	71.0	100.0	92.6	64.4
GPT-4o-mini	100.0	99.4	97.1	100.0	98.1	89.3	100.0	92.7	83.9
GPT-4o	100.0	98.2	99.1	100.0	97.6	96.1	99.0	97.4	90.0

Three-step Task

Context Size			2k			4k			8k
# Distractors	0	1	5	0	1	5	0	1	5
Mistral-7B-v0.1	81.4	54.0	23.8	61.4	42.3	21.0	19.5	14.7	10.0
StarCoder2-7B	95.6	83.0	58.2	94.5	85.4	46.0	60.0	39.5	20.5
StarCoderBase-1B	80.7	59.0	28.1	70.9	68.3	26.4	79.6	58.7	20.5
StarCoderBase-7B	91.7	68.0	41.2	82.6	75.9	42.3	88.0	77.1	33.4
StarCoderBase-15.5B	81.4	66.5	49.5	69.7	62.1	39.6	65.0	49.5	21.5
DeepSeekCoder-6.7B-base	100.0	95.3	69.7	99.9	93.6	58.3	99.3	90.0	47.5
GPT-4o-mini	100.0	94.8	77.7	99.8	95.7	64.1	98.5	82.3	52.5
GPT-4o	100.0	98.4	95.3	100.0	96.7	90.1	99.8	93.1	89.4

Concatenation Task

Context Size			2k			4k			8k
# Distractors	0	1	5	0	1	5	0	1	5
Mistral-7B-v0.1	27.0	18.5	6.4	12.8	6.2	1.2	1.1	0.6	0.4
StarCoder2-7B	54.3	36.2	13.4	26.5	23.0	5.4	8.6	3.9	1.1
StarCoderBase-1B	2.9	1.5	0.6	1.1	0.5	0.2	0.3	0.1	0.0
StarCoderBase-7B	37.5	19.2	9.1	26.9	11.7	1.6	16.1	7.3	0.5
StarCoderBase-15.5B	50.5	36.6	16.1	35.6	21.9	7.3	27.5	13.3	4.6
DeepSeekCoder-6.7B-base	60.1	50.1	28.4	54.2	33.3	10.0	47.4	20.4	5.8
GPT-4o-mini	85.4	78.8	71.5	74.4	84.0	54.4	73.0	62.2	36.4
GPT-4o	97.6	98.7	97.5	93.2	97.3	96.1	94.8	92.4	91.4

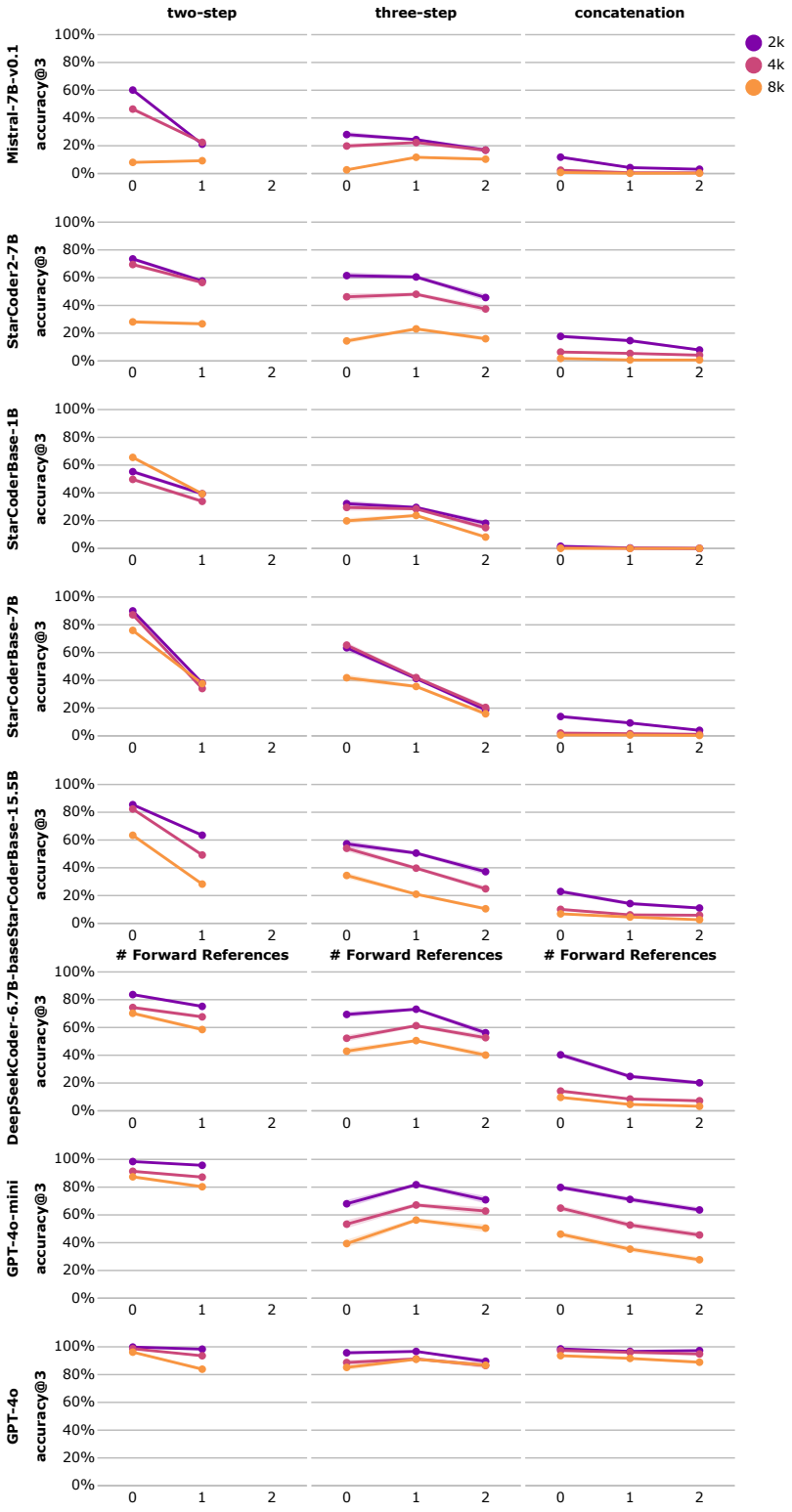


Figure 9: Effect of forward references by task variant, with 5 distractors. The data is showin Table 2

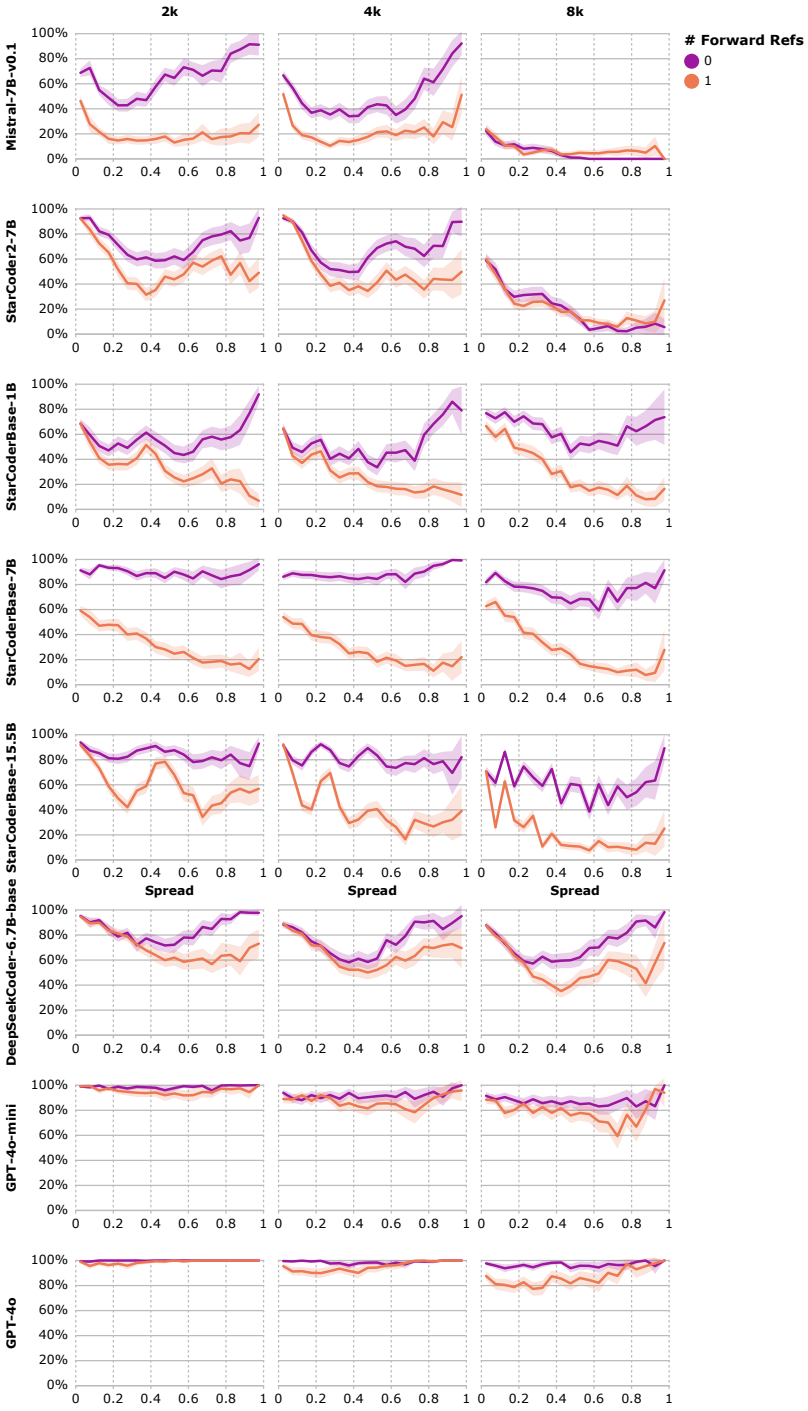


Figure 10: Effect of spread for two step tasks for each model, with 5 distractors.

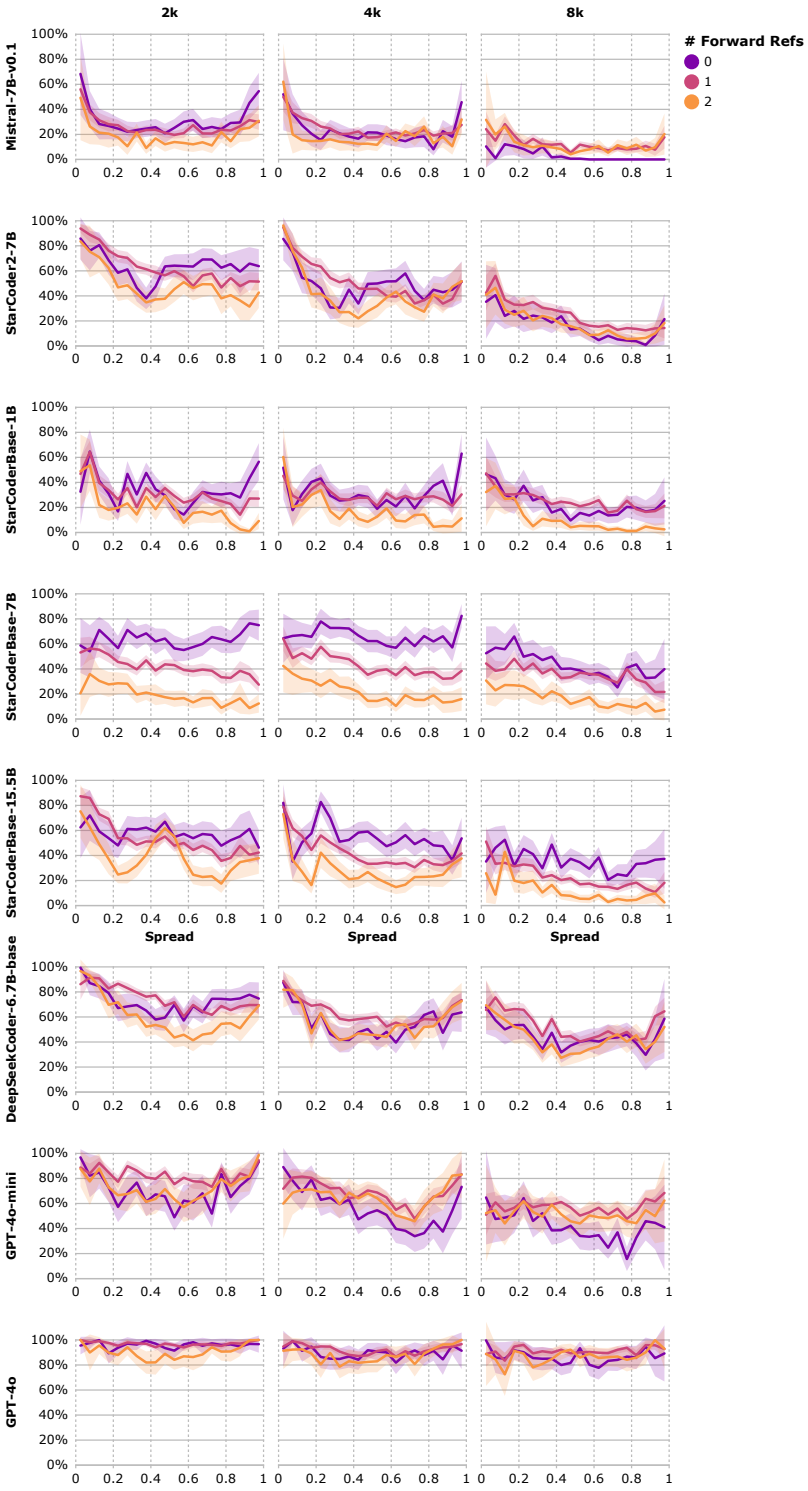


Figure 11: Effect of spread for three step tasks for each model, with 5 distractors.

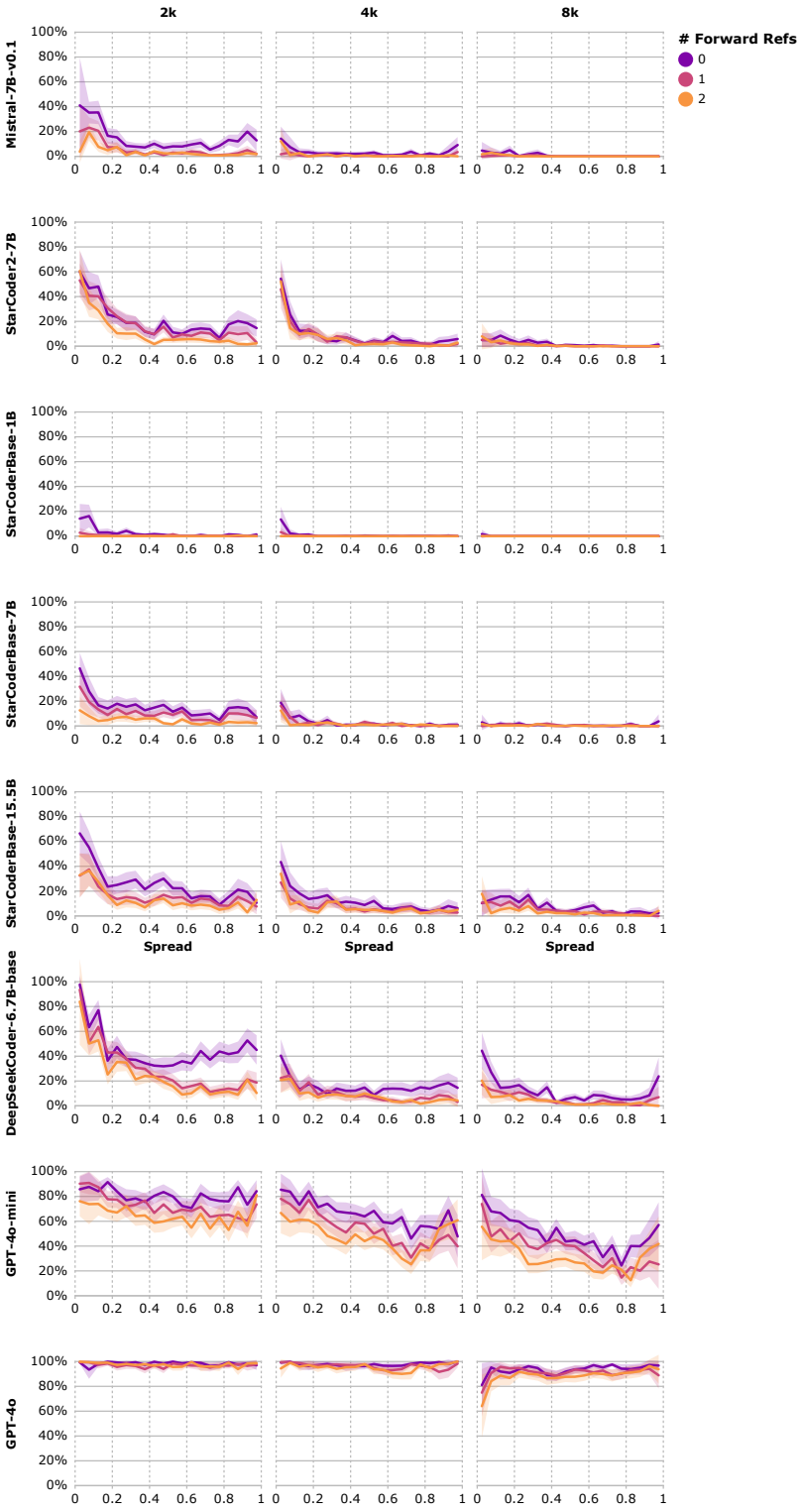
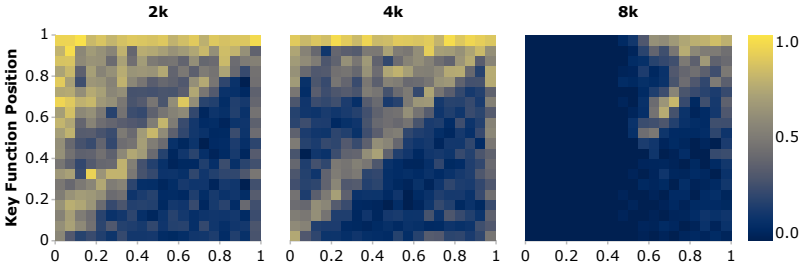
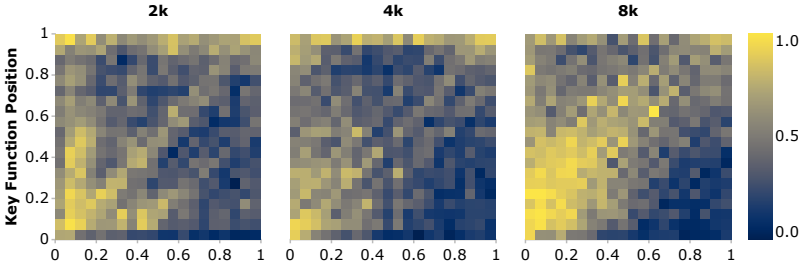


Figure 12: Effect of spread for concatenation tasks for each model, with 5 distractors.

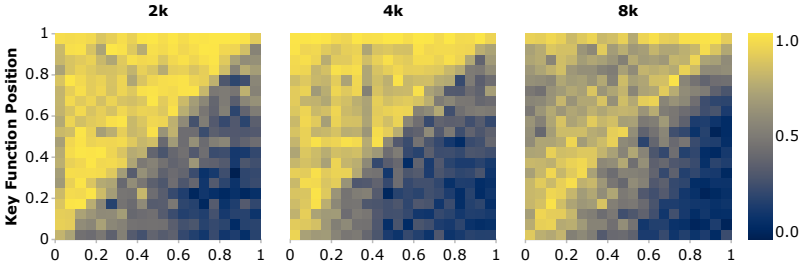
Mistral-7B-v0.1



StarCoderBase-1B



StarCoderBase-7B



StarCoderBase-15.5B

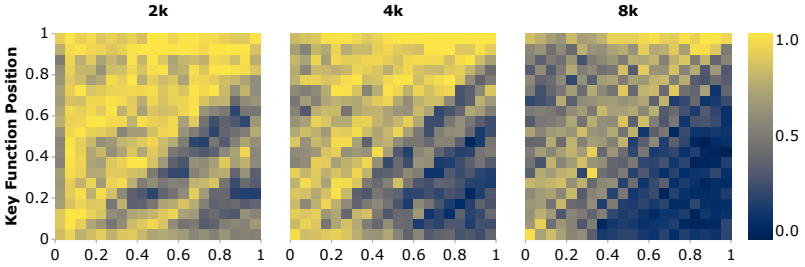
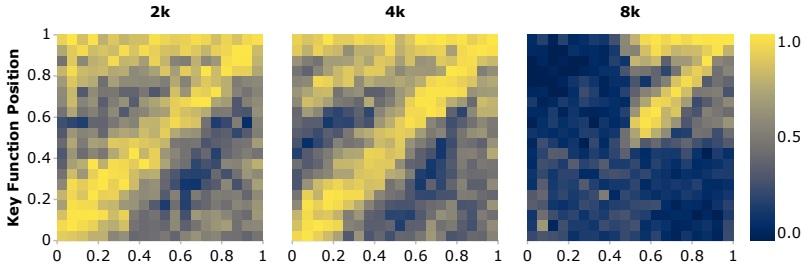
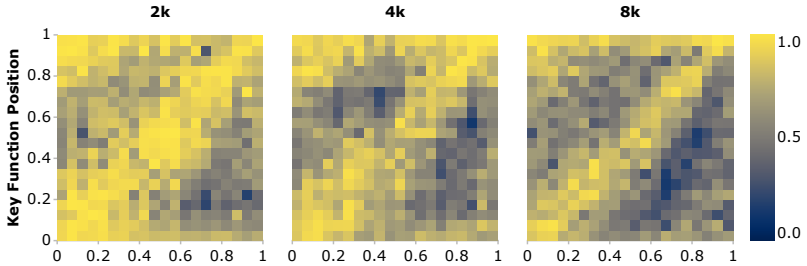


Figure 13: Effect of key and value function position for two-step tasks, with 5 distractors. Color represents accuracy@3 score.

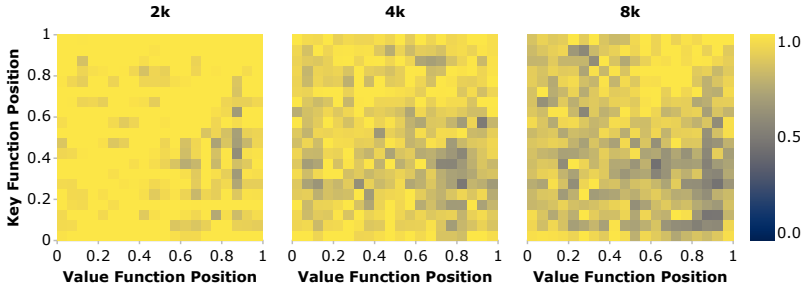
StarCoder2-7B



DeepSeekCoder 6.7B-base



GPT-4o-mini



GPT-4o

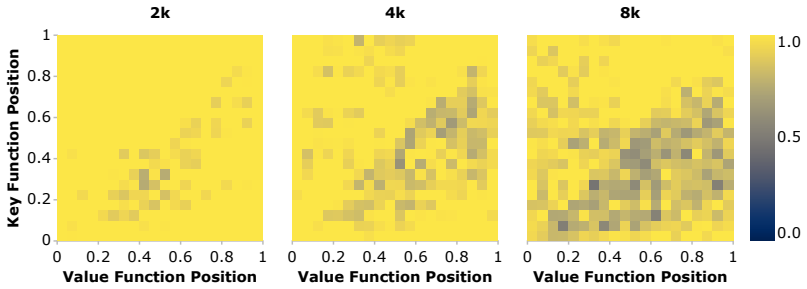


Figure 14: Effect of key and value function position for two-step tasks, with 5 distractors. Color represents accuracy@3 score.

B Incorrect Response Analysis

Here we present analyses of incorrect responses broken down by model. We also present details of the edit distance analysis.

Table 7: Percentage of distractor answers across all experiments with distractors

Prompt Size	2k	4k	8k	Mean
Mistral-7B-v0.1	10.5	9.4	5.3	8.4
StarCoder2-7B	16.7	19.4	12.9	16.3
StarCoderBase-1B	11.5	11.0	14.2	12.2
StarCoderBase-7B	7.7	9.5	11.7	9.6
StarCoderBase-15.5B	11.5	11.3	10.1	11.0
DeepSeekCoder-6.7B-base	16.0	17.5	15.1	16.2
GPT-4o-mini	5.2	7.3	11.1	7.9
GPT-4o	0.9	1.9	2.7	1.8

Table 8: Percentage of incorrect answers that are distractor values

Prompt Size	2k	4k	8k	Mean
Mistral-7B-v0.1	14.3	12.5	6.0	10.9
StarCoder2-7B	30.8	35.0	16.9	27.6
StarCoderBase-1B	15.9	16.2	22.2	18.1
StarCoderBase-7B	12.5	15.6	20.8	16.3
StarCoderBase-15.5B	19.8	17.9	15.5	17.7
DeepSeekCoder-6.7B-base	36.4	34.6	29.5	33.5
GPT-4o-mini	31.0	33.8	35.5	33.4
GPT-4o	25.8	29.1	24.2	26.3

Table 9: Percentage of incorrect responses in the concatenation task where the response is one of the two strings that should have been concatenated.

Prompt Size	2k	4k	8k	Mean
Mistral-7B-v0.1	20.7	26.7	11.5	19.6
StarCoder2-7B	21.8	33.5	30.0	28.5
StarCoderBase-1B	44.9	43.0	44.2	44.0
StarCoderBase-7B	40.9	50.9	51.6	47.8
StarCoderBase-15.5B	30.3	33.1	32.5	32.0
DeepSeekCoder-6.7B-base	23.4	30.4	34.7	29.5
GPT-4o-mini	28.1	26.7	32.3	29.0
GPT-4o	13.7	7.6	8.1	9.8

Table 10: Levenshtein Distance for incorrect responses.

Prompt Size	2k	4k	8k	Mean
Mistral-7B-v0.1	10.8	12.3	15.1	12.7
StarCoder2-7B	10.5	10.5	11.4	10.8
StarCoderBase-1B	9.6	9.4	9.7	9.6
StarCoderBase-7B	9.5	9.7	9.7	9.6
StarCoderBase-15.5B	11.1	11.9	12.3	11.8
DeepSeekCoder-6.7B-base	9.3	9.9	10.2	9.8
GPT-4o-mini	8.5	10.0	12.1	10.2
GPT-4o	9.5	11.4	12.4	11.1

C Detailed Results: Call Graph Comment Experiment

We include details of model performance (accuracy@3 percent scores) for the full-sentence template variant of the call graph experiment.

Table 11: Call graph comment performance by model.

Task	Model	Comment Type			
		None	Calls	Called-By	Both
Three Step	Mistral-7B-v0.1	18.2	16.7	33.8	39.6
	StarCoder2-7B	41.6	57.4	80.2	90.8
	StarCoderBase-1B	25.0	30.5	46.5	47.4
	StarCoderBase-7B	39.0	48.7	86.0	87.2
	StarCoderBase-15.5B	36.9	49.5	85.3	87.5
	DeepSeekCoder-6.7B-base	58.5	72.6	86.5	88.2
	GPT-4o-mini	64.8	74.2	82.7	85.4
	GPT-4o	91.6	95.0	99.0	97.8
Concatenation	Mistral-7B-v0.1	2.7	3.8	5.6	8.9
	StarCoder2-7B	6.6	14.3	27.4	31.2
	StarCoderBase-1B	0.3	0.3	0.3	0.4
	StarCoderBase-7B	3.7	17.6	16.0	30.2
	StarCoderBase-15.5B	9.3	20.0	33.7	40.6
	DeepSeekCoder-6.7B-base	14.7	17.9	27.2	35.8
	GPT-4o-mini	54.1	65.1	71.1	73.3
	GPT-4o	95.0	95.3	96.1	94.0

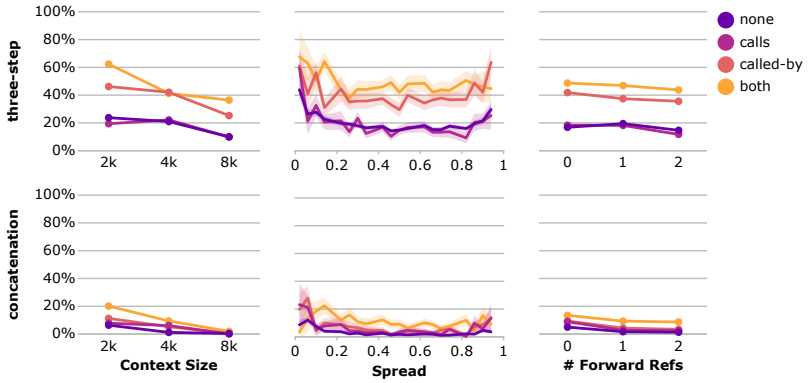


Figure 15: Effect of call graph comments on Mistral-7B-v0.1.

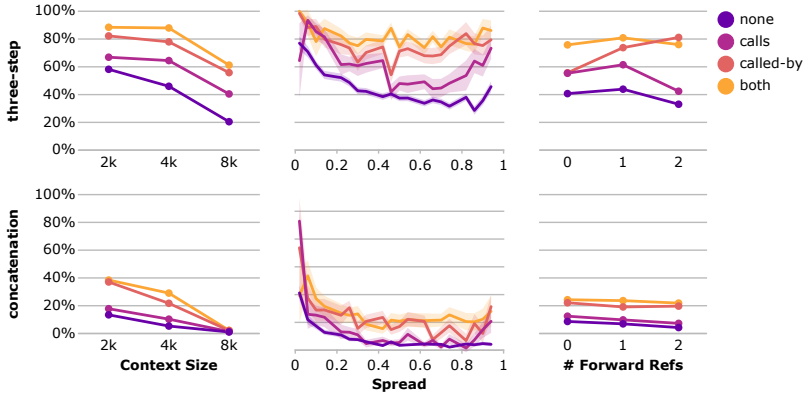


Figure 16: Effect of call graph comments on Starcoder2-7B performance.

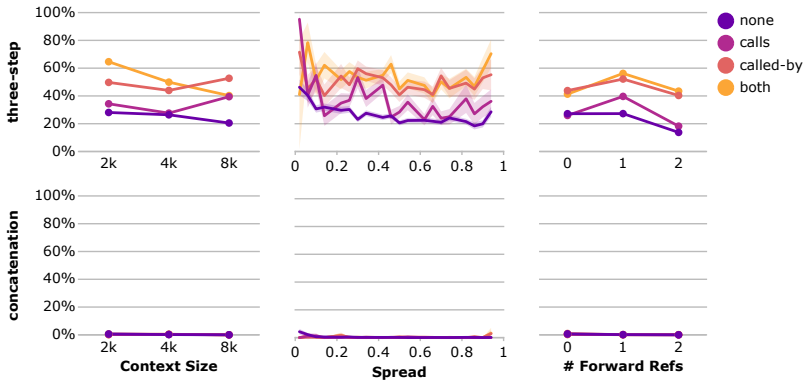


Figure 17: Effect of call graph comments on StarCoderBase-1B performance.

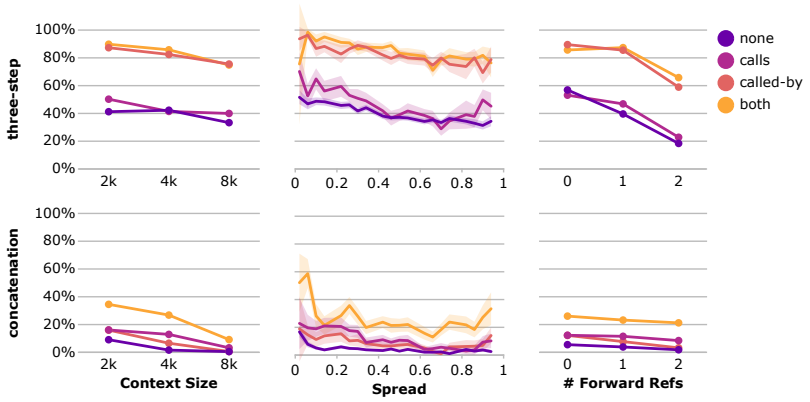


Figure 18: Effect of call graph comments on StarCoderBase-7B performance.

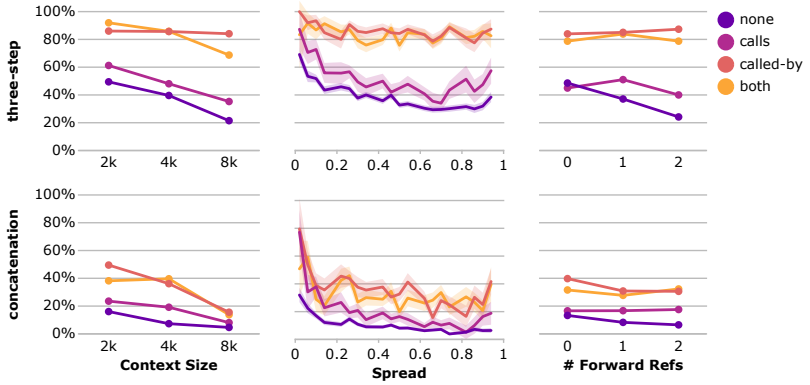


Figure 19: Effect of call graph comments on StarCoderBase-15.5B performance.

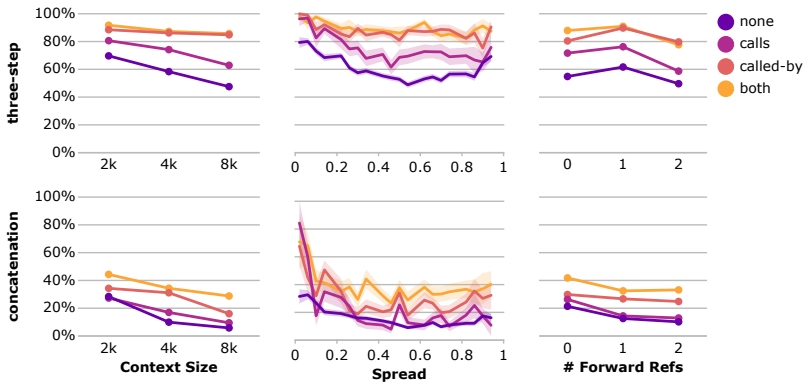


Figure 20: Effect of call graph comments on DeepSeekCoder-6.7B-base performance.

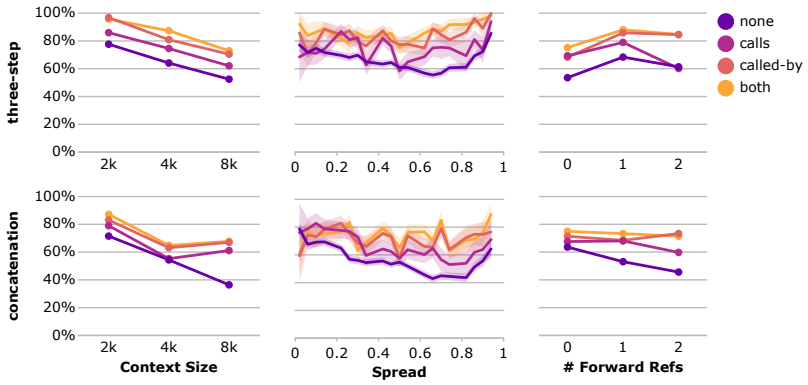


Figure 21: Effect of call graph comments on GPT-4o-mini performance.

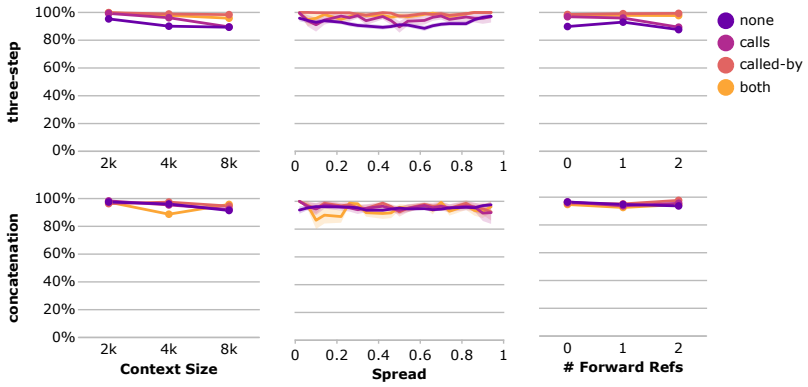


Figure 22: Effect of call graph comments on GPT-4o performance.

Table 12: Next-hop vs. full graph comments performance on three-step retrieval.

Model	# Forward References	Comment Type		
		None	Next Hop	Full Graph
Mistral-7B-v0.1	0	16.8	39.6	53.4
	1	19.5	29.9	39.8
	2	14.7	17.1	25.3
StarCoder2-7B	0	40.7	83.2	90.9
	1	43.9	80.5	92.5
	2	33.1	61.1	84.3
StarCoderBase-1B	0	27.2	28.4	32.9
	1	27.3	46.3	52.1
	2	13.8	29.9	43.4
StarCoderBase-7B	0	56.9	88.7	92.5
	1	39.6	78.9	91.2
	2	18.4	46.8	66.0
StarCoderBase-15.5B	0	48.6	66.6	76.6
	1	37.1	78.7	89.3
	2	24.2	80.2	91.5
DeepSeekCoder-6.7B-base	0	54.8	84.5	87.9
	1	61.7	86.1	91.0
	2	49.6	66.7	77.7
GPT-4o-mini	0	53.6	70.0	75.2
	1	68.4	84.3	88.1
	2	61.4	78.7	84.7
GPT-4o	0	89.8	98.1	97.8
	1	93.0	98.3	97.9
	2	87.6	96.9	97.7

D Hyperparameters for Generation

For all experiments we use nucleus sampling with the following parameters.

Table 13: Generation hyperparameters.

Hyperparameter	Value
Temperature	0.8
Top p	0.95
Top k	0
Batch size	1
Output samples per input prompt	10

E Confidence Intervals

All confidence intervals (error bars and plus/minus values) are calculated as 95% confidence intervals of the estimated mean following assumptions of normal distribution and independent samples.

F Compute Information

We ran all experiments on machines with a single A100 GPU with 80GB of VRAM on a cloud provider. Experiments were run over the course of about a month and we estimate ~1000 GPU hours were used including runs that had bugs in them and additional experiments that are not part of the paper. We use FlashAttention Dao et al. (2022) to improve memory usage and latency of generation.

G Prompt Samples for Multi Step Key Retrieval

```
def iaizjb_184360_440195():
    return "wxmbrnpokw"

def string_to_md5(text):
    """
    Given a string 'text', return its md5 hash equivalent string.
    If 'text' is an empty string, return None.

    >>> string_to_md5('Hello world') == '3e25960a79dbc69b674cd4ec67a72c62'
    """
    import hashlib
    return hashlib.md5(text.encode('ascii')).hexdigest() if text else None

def egllun_467846():
    return "eooyfwmxln"

def count_upper(s):
    """
    Given a string s, count the number of uppercase vowels in even indices.

    For example:
    count_upper('aBCdEf') returns 1
    count_upper('abcdefg') returns 0
    count_upper('dBBE') returns 0
    """
    count = 0
    for i in range(0,len(s),2):
        if s[i] in "AEIOU":
            count += 1
    return count

def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements that are in even positions
     .


    Examples
    solution([5, 8, 7, 1]) ==> 12
    solution([3, 3, 3, 3, 3]) ==> 9
    solution([30, 13, 24, 321]) ==>0
    """
    return sum([x for idx, x in enumerate(lst) if idx%2==0 and x%2==1])

def choose_num(x, y):
    """This function takes two positive numbers x and y and returns the
    biggest even integer number that is in the range [x, y] inclusive. If
    there's no such number, then the function should return -1.

    For example:
    choose_num(12, 15) = 14
    choose_num(13, 12) = -1
    """
    if x > y:
        return -1
    if y % 2 == 0:
        return y
    if x == y:
        return -1
    return y - 1

def digitSum(s):
    """Task
    Write a function that takes a string as input and returns the sum of the upper characters only'
    ASCII codes.

    Examples:
        digitSum("") => 0
        digitSum("abAB") => 131
        digitSum("abcCd") => 67
        digitSum("helloE") => 69
        digitSum("woArBld") => 131
        digitSum("aAaaaXa") => 153
    """
    if s == "": return 0
    return sum(ord(char) if char.isupper() else 0 for char in s)

def multiply(a, b):
    """Complete the function that takes two integers and returns
```

```
    the product of their unit digits.
    Assume the input is always valid.
    Examples:
    multiply(148, 412) should return 16.
    multiply(19, 28) should return 72.
    multiply(2020, 1851) should return 0.
    multiply(14,-15) should return 20.
    """
    return abs(a % 10) * abs(b % 10)

def right_angle_triangle(a, b, c):
    '''
    Given the lengths of the three sides of a triangle. Return True if the three
    sides form a right-angled triangle, False otherwise.
    A right-angled triangle is a triangle in which one angle is right angle or
    90 degree.
    Example:
    right_angle_triangle(3, 4, 5) == True
    right_angle_triangle(1, 2, 3) == False
    '''
    return a*a == b*b + c*c or b*b == a*a + c*c or c*c == a*a + b*b

def add_elements(arr, k):
    """
    Given a non-empty array of integers arr and an integer k, return
    the sum of the elements with at most two digits from the first k elements of arr.

    Example:

        Input: arr = [111,21,3,4000,5,6,7,8,9], k = 4
        Output: 24 # sum of 21 + 3

    Constraints:
        1. 1 <= len(arr) <= 100
        2. 1 <= k <= len(arr)
    """
    return sum(elem for elem in arr[:k] if len(str(elem)) <= 2)

def vskfby_510934():
    return "thwtyqwjws"

def qgtsin_336194_iwdghb():
    return iaizjb_184360_440195()

def awdpgq_293061_vwetvu():
    return rbwofb_803321_331141()

def even_odd_count(num):
    """Given an integer. return a tuple that has the number of even and odd digits respectively.

     Example:
        even_odd_count(-12) ==> (1, 1)
        even_odd_count(123) ==> (1, 2)
    """
    even_count = 0
    odd_count = 0
    for i in str(abs(num)):
        if int(i)%2==0:
            even_count +=1
        else:
            odd_count +=1
    return (even_count, odd_count)

def minSubArraySum(nums):
    """
    Given an array of integers nums, find the minimum sum of any non-empty sub-array
    of nums.
    Example
    minSubArraySum([2, 3, 4, 1, 2, 4]) == 1
    minSubArraySum([-1, -2, -3]) == -6
    """
    max_sum = 0
    s = 0
    for num in nums:
        s += -num
        if (s < 0):
            s = 0
        max_sum = max(s, max_sum)
    if max_sum == 0:
        max_sum = max(-i for i in nums)
    min_sum = -max_sum
```

```
    return min_sum

def qvkrlp_701464_787942():
    return vskfby_510934()

def rdcxoi_135343():
    return egllun_467846()

def iscube(a):
    '''
    Write a function that takes an integer a and returns True
    if this ingeger is a cube of some integer number.
    Note: you may assume the input is always valid.
    Examples:
    iscube(1) ==> True
    iscube(2) ==> False
    iscube(-1) ==> True
    iscube(64) ==> True
    iscube(0) ==> True
    iscube(180) ==> False
    '''
    a = abs(a)
    return int(round(a ** (1. / 3))) ** 3 == a

def special_factorial(n):
    """The Brazilian factorial is defined as:
    brazilian_factorial(n) = n! * (n-1)! * (n-2)! * ... * 1!
    where n > 0

    For example:
    >>> special_factorial(4)
    288

    The function will receive an integer as input and should return the special
    factorial of this integer.
    """
    fact_i = 1
    special_fact = 1
    for i in range(1, n+1):
        fact_i *= i
        special_fact *= fact_i
    return special_fact

def is_equal_to_sum_even(n):
    """Evaluate whether the given number n can be written as the sum of exactly 4 positive even numbers
    Example
    is_equal_to_sum_even(4) == False
    is_equal_to_sum_even(6) == False
    is_equal_to_sum_even(8) == True
    """
    return n%2 == 0 and n >= 8

assert rdcxoi_135343() ==
```

Listing 1: Sample prompt for two-step retrieval task with 2000 token context size and 5 distractors.

H Prompt Samples for Call Graph Comments Experiment

```
# This function calls oswgtr_325169_862229
# This function is called by aokwfl_208971_hwmofh
def gqpvbp_138573():
    return oswgtr_325169_862229()

# This function calls gqpvbp_138573 and oswgtr_325169_862229
def aokwfl_208971_hwmofh():
    return gqpvbp_138573()

# This function calls lhezee_508969 and hjdnwl_724283
def oftoyy_286138():
    return lhezee_508969()

# This function is called by gqpvbp_138573 and aokwfl_208971_hwmofh
def oswgtr_325169_862229():
    return "kyfgholcrg"

# This function calls gjobme_651008_tymmij
# This function is called by wwzfoa_904885
def bweckw_860527_nykiyp():
    return gjobme_651008_tymmij()

# This function is called by lhezee_508969 and oftoyy_286138
def hjdnwl_724283():
    return "pwincnzyqh"

# This function is called by bweckw_860527_nykiyp and wwzfoa_904885
def gjobme_651008_tymmij():
    return "axxtrhucug"

# This function calls bweckw_860527_nykiyp and gjobme_651008_tymmij
def wwzfoa_904885():
    return bweckw_860527_nykiyp()
```

Listing 2: Full sentence call graph comments for three step retrieval. 5 distractors. HumanEval functions excluded for brevity.

```
# oswgtr_325169_862229
# aokwfl_208971_hwmofh
def gqpvbp_138573():
    return oswgtr_325169_862229()

# gqpvbp_138573, oswgtr_325169_862229
def aokwfl_208971_hwmofh():
    return gqpvbp_138573()

# lhezee_508969, hjdnwl_724283
def oftoyy_286138():
    return lhezee_508969()

# gqpvbp_138573, aokwfl_208971_hwmofh
def oswgtr_325169_862229():
    return "kyfgholcrg"

# gjobme_651008_tymmij
# wwzfoa_904885
def bweckw_860527_nykiyp():
    return gjobme_651008_tymmij()

# lhezee_508969, oftoyy_286138
def hjdnwl_724283():
    return "pwincnzyqh"

# bweckw_860527_nykiyp, wwzfoa_904885
def gjobme_651008_tymmij():
    return "axxtrhucug"

# bweckw_860527_nykiyp, gjobme_651008_tymmij
def wwzfoa_904885():
    return bweckw_860527_nykiyp()
```

Listing 3: Function names only call graph comments for three step retrieval. 5 distractors. HumanEval functions excluded for brevity.