CAN LONG-CONTEXT LANGUAGE MODELS SOLVE REPOSITORY-LEVEL CODE GENERATION?

Anonymous authors
Paper under double-blind review

ABSTRACT

With the advance of real-world tasks that necessitate increasingly long contexts, recent language models (LMs) have begun to support longer context windows. One particularly complex task is repository-level code generation, where retrieval-augmented generation (RAG) has been used as the *de facto* approach. Nonetheless, RAG may not be optimal in processing entire codebases with cross-file dependencies. Therefore, we ask: *can we instead leverage long-context LMs to solve repository-level code generation problems?* To answer this question, we conduct a comparative study of LC and RAG methods using top-performing open-source CODELLAMA 7B and closed CLAUDE-3.5-sonnet models. We evaluate on the repository-level code completion benchmark — RepoEval (Zhang et al., 2023), and find that LC can match or surpass RAG performance when the repository is sufficiently small and well-structured, yet RAG still outperforms LC when the repository grows larger or involves complex structures, dependencies, or domain-specific implementations. We further ablate on context ordering and code snippet chunking, and find that better ordering of input code snippets can boost both LC results, while design choices for code snippet chunking such as size and overlaps do not produce prominent effects. Overall, our work reveals the scenarios where current LC methods are shown effective and fall short in repository-level code generation, potentially offering insights for future method developments.

1 INTRODUCTION

With the recent advances of large language models (LLMs) in solving various tasks, repository-level program generation has emerged as a critical problem in achieving automated software engineering works (Chen et al., 2021; Jimenez et al., 2023). The repository-level code generation task provides a natural language query (NL), and asks the model to generate programs by understanding the entire codebase. Nonetheless, code repositories often contain hundreds of files and feature complex code structures and long-distance dependencies across files. For most traditional LLMs with short context window length constraints (e.g., $1k$ or $4k$), these long codebase content poses pronounced challenges (Liu et al., 2024).

One common approach to accommodate the short context constraints is to leverage the retrieval-augmented generation (RAG) method (Guu et al., 2020; Lewis et al., 2020; Izacard et al., 2022), which first retrieves a few relevant code snippets from the code repository, and then generates programs only based on the query and these pre-selected set of contexts. However, the effectiveness of RAG heavily depends on the relatedness and organization of the retrieved content. When dealing with large-scale repositories containing intricate module dependencies and diverse coding patterns, the retrieval process may lack a general view of the codebase, thus often failing to capture essential relations and producing inconsistent or incorrect programs (Zhang et al., 2023; Wang et al., 2024).

Another emerging approach that potentially avoids this imperfect context retrieval issue, is to directly input the entire code repository to an LLM that supports long context (LC). For code-specific LMs, LC support can usually be achieved in two ways. First, at (pre-)training time by training on longer text sequences, for example, CODELLAMA (Roziere et al., 2023) and STARCODER (Li et al., 2023) support inputs of at most $16k$ tokens. Alternatively, one can apply length-extending modules, such as Unlimiformer (Bertsch et al., 2023), Longformer (Beltagy et al., 2020), Performers (Choromanski et al., 2020) to extrapolate the input context limit at inference time. Given these

techniques in enabling longer inputs for code LMs, however, limited work has explored using LC to solve repository-level code generation problems, thus making it unclear how well LC methods perform compared to RAG.

To answer this question, we first ask: *Can long-context models solve repository-level code generation tasks?* (§3) by comparing RAG and LC methods on the RepoEval benchmark (Zhang et al., 2023) with rigorous execution-based evaluation (Wang et al., 2024). We experiment with (1) the top-performing API model, CLAUDE-3.5-sonnet, and provide retrieved contexts or entire code repository content for RAG and LC methods, respectively. In addition to closed models, we also study the open-weight CODELLAMA model. To enable CODELLAMA for processing (e.g., with millions of tokens) beyond its $16k$-token limit, we extend its context limit to unlimited length using Unlimiformer (Bertsch et al., 2023) and then perform similar experiments as those for Claude.

Among experiments with varied repositories, first, we find that *LC can outperform RAG when the codebase is small* (i.e., less than $40k$ tokens) *and well-structured*, often with tightly coupled modules and consistent coding patterns; this is particularly evident in CODELLAMA compared to its stronger CLAUDE counterpart. Meanwhile, for slightly larger ($80k$–$120k$ tokens) repositories with complex modules yet rather consistent structure and coding styles, LC models can still perform on par with the best RAG setting, suggesting certain potentials for LC to become more effective via better understandings of complex codebases. Nonetheless, for even larger codebases beyond $120k$ tokens, we find the CODELLAMA model often produces near-zero results and CLAUDE degrading severely; both underperform RAG methods by a large margin. Expectedly, LC performance with both models degrades to a greater extent when the codebase structure gets more complex.

Comparing two models under the LC setting, interestingly, we find the weaker, code-specialized CODELLAMA *7B model sometimes outperforms the generally stronger* CLAUDE*-3.5-sonnet model*, particularly on small repositories such as `amazon-science/patchcore-inspection`. However, as the repository grows larger, CODELLAMA tends to degrade severely fast while CLAUDE appears more robust to extended contexts.

We further investigate the advantages offered by two design choices — code chunking strategy and input snippet ordering (§4). We find that ordering contexts based on their lexical or semantically similar to the problem brings substantial performance improvements in LC mode across all repositories. In comparison, the chunking sizes of code snippets and their inter-overlaps do not have prominent effects on final RAG performance, suggesting the need for deeper investigation in designing better RAG methods.

In general, our work aims to systematically compare LC and RAG methods in generating code for repositories with varied sizes, structures, and coding styles. We identify promises in LC models in processing small codebases, and hope to facilitate developments in solving remaining challenges regarding more large, complex codebases.

2 PROBLEM STATEMENT

In this section, we first introduce the RAG and LC methods adopted in our experiments (§2.1), then provide the benchmark, evaluation, and model details (§2.2, §2.3).

2.1 METHODS

We tackle the code completion task, where each example contains an incomplete code file f, of which the last n tokens before the incomplete region is used as the query q for the problem.

Retrieval-Augmented Generation We adopt the RAG pipeline from Zhang et al. (2023) using a sparse bag-of-words retrieval Lu et al. (2022), which given a code repository, first chunks the repository into code snippets as the retrieval pool, performs retrieval, and augments retrieved contexts for generation.

To *create the retrieval database*, we follow Zhang et al. (2023) to chunk each file f into w-line snippets, while keeping a s-line overlap between adjacent snippets. More concretely, for file f we get code snippets $C_f = \{c_i \mid c_i = f[i : i + w], \forall i | (w - s) = 0\}$. The full retrieval database is the union of code snippets from all files: $C_{repo} = \bigcup_{f \in \text{repo}} C_f$.

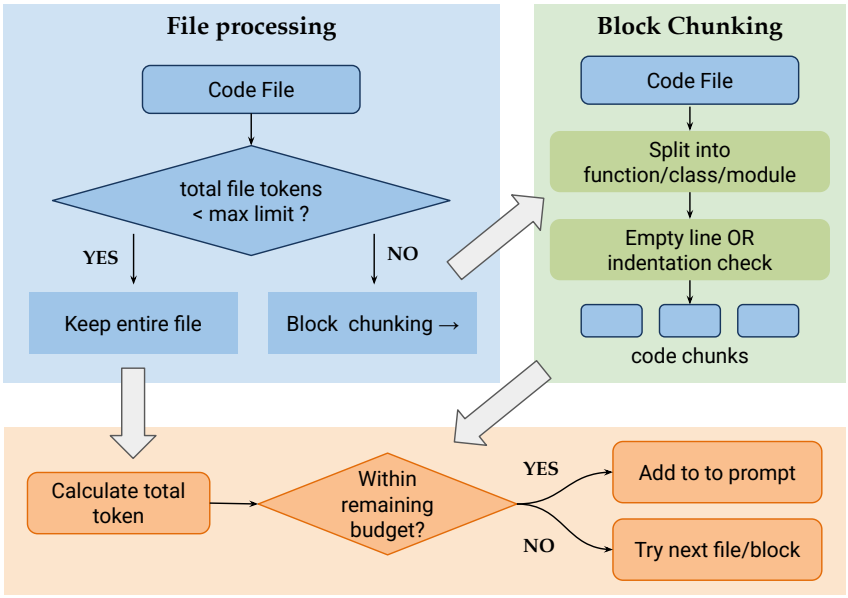

Figure 1: Dynamic prompt construction process for RAG.

To *perform retrieval* for a given problem query q (i.e., the last n tokens before the incomplete region in the target code file), we use q as the retrieval query and find top-k relevant code snippets from the database C_{repo}, $C_{\text{ret}} = \text{top}_k\{(c,\ \text{sim}(q,\ c)) \mid c \in C_{\text{repo}}\}$. $\text{sim}(q,\ c)$ stands for the similarity measure, which can be implemented using token-based Jaccard similarity or cosine similarity of dense vector representations (Guo et al., 2022a).

To *generate programs with retrieved code snippets*, we provide the query q and top-relevant retrieved code snippets C_{ret} to the RAG model. We implement a dynamic prompt construction process for adding retrieved code snippets, as illustrated in Figure 1. More specifically, given a pre-determined maximum token count limit, we continue adding top-ranking code snippets to the input context until it reaches the limit. We experiment with varied context lengths ranging from $4k$, $8k$, $16k$, $32k$, to $64K$ tokens, under the RAG setup, to analyze how context length affects the generation quality.

Long Context Model In the long-context setting, we build the input prompt by aggregating all Python files from the repository. We traverse the repository using `os.walk` to collect all Python files, where for the target file, we only include content before the target function's starting line number (specified in the benchmark's metadata as `context_start_lineno`). To provide a rigorous test of LC method capabilities, we randomly shuffle the collected files to break any potentially semantically correlations if intentionally ordered in other ways. At the same time, to ensure consistent and reproducible experiments, we use a fixed random seed (i.e., 42) throughout the LC experiments. This design choice ensures that models must truly understand and utilize the entire context rather than relying on distance-based ordering, better aligning with LC's fundamental premise of comprehensive context utilization.

For both RAG and LC experiments, we provide the path metadata and for each of the code snippet (either w-line chunks in RAG or full file content in LC) using the standardized prompt format. We provide the exact prompt in §A.

2.2 DATASET

We use the RepoEval (Zhang et al., 2023) benchmark, which features the repository-level code completion task. Unlike many benchmarks that focus on isolated code snippets, RepoEval requires models to understand long-range contexts across multiple files in an entire code repository. RepoEval consists of three test splits — line, API, and function completion tasks — created from Python repositories with diverse coding styles and domains. We adopt the function split for our experiments,

because it is the only test split that supports rigorous execution-based evaluation. The function completion split covers six code repositories of varying lengths, from 29k tokens to 132k tokens. Table 1 shows the repositories covered in the function test split and their respective statistics.

Repository Name	# files	# lines	# tokens	# tasks
amazon-science/patchcore-inspection	16	2,532	29,179	32
deepmind/tracr	56	9,110	110,574	146
facebookresearch/omnivore	66	11,797	132,538	22
google/lightweight_mmm	36	9,676	123,118	64
lucidrains/imagen-pytorch	14	7,324	83,579	67
maxhumber/redframes	49	3,881	40,672	42

Table 1: Statistics of the repositories in RepoEval function set. The repositories cover a range of sizes, from smaller codebases (29k tokens in amazon-science/patchcore-inspection) to larger ones (132k tokens in google/lightweight_mmm), and span different domains including computer vision, machine learning, and data processing.

Evaluation Metrics: Pass@1 We evaluate the functional correctness of model-generated programs using the pass@1 metric (Chen et al., 2021) as implemented by Wang et al. (2024). Compared to traditional similarity metrics like Exact Match (EM) and Edit Similarity (ES), execution metrics can more accurately reflect the functional correctness of code generation. This is because even if the generated code differs superficially from the target code, it is still valuable in real applications if it executes successfully and produces correct results.

2.3 MODELS

We experiment with top-performing code-specific LMs that are open-weight and closed-sourced.

Open-Weight Models We use CODELLAMA 7B (Roziere et al., 2023) as top representatives for the open-weight code LMs. CODELLAMA 7B has a maximum context length of 16k tokens, enabling it to handle longer code snippets and more complex programming contexts. To handle larger repositories that exceed this limit, we incorporate Unlimiformer (Bertsch et al., 2023) to extend its context limit to an unlimited number of tokens.

Close Source Models We use CLAUDE-3.5-sonnet (Anthropic, 2023), one of the strongest API LM. CLAUDE has a context limit of 200k tokens, and readily supports RAG with varying context lengths and full-repository long-context setups.

3 RESULTS AND ANALYSIS

We conducted experiments using both CODELLAMA 7B and CLAUDE-3.5-sonnet models with RAG and LC approaches, and report their pass@1 scores in each repository in Table 2.

First of all, comparing RAG methods with different context lengths, CODELLAMA generally achieves the best performance at 16k-token range, aligning with its default context limit. On the other hand, CLAUDE tends to perform better with shorter contexts, and often scores the best with 4k token contexts.

LC Surpasses RAG on Small, Structured Codebase Surprisingly, for the weaker CODELLAMA, on small, well-organized repositories (i.e., `amazon-science/patchcore-inspection`), LC achieves the highest score 37.5% among all methods, while RAG in the best setup with 16k achieves comparable scores. We also find code in this repository to have better modularity and consistent coding styles. This indicates that, on small, well-structured repositories, CODELLAMA can effectively consume the full repository content and produce correct programs.

We show a representative example in Figure 2a about implementing the forward method of the `Preprocessing` class. While RAG@16k generates a partially correct but incomplete imple-

Repository	Size	RAG			LC
		4k	16k	32k	= Size
CodeLlama-7B					
amazon-science/patchcore-inspection	$29k$	28.1	**37.5**	34.4	**37.5**
maxhumber/redframes	$40k$	**21.4**	**21.4**	19.1	14.3
lucidrains/imagen-pytorch	$83k$	49.3	**59.7**	56.7	1.4
deepmind/tracr	$110k$	34.3	**37.0**	35.6	0.0
google/lightweight_mmm	$123k$	17.2	**29.7**	21.9	0.0
facebookresearch/omnivore	$132k$	18.2	**27.3**	18.2	0.0
Average		31.1	37.5	34.0	5.0
Claude-3.5-Sonnet					
amazon-science/patchcore-inspection	$29k$	**40.6**	28.1	21.9	15.6
maxhumber/redframes	$40k$	33.3	**38.1**	35.7	31.0
lucidrains/imagen-pytorch	$83k$	**43.3**	**43.3**	40.3	38.8
deepmind/tracr	$110k$	**51.4**	43.8	41.8	36.3
google/lightweight_mmm	$123k$	**25.0**	21.9	18.8	9.4
facebookresearch/omnivore	$132k$	**40.9**	18.2	18.2	13.6
Average		41.8	36.5	33.8	28.4

Table 2: Performance of CodeLlama and Claude under RAG and LC settings. Numbers show the percentage of successful code completions. The best scores for each repository are bolded.

mentation that misses the crucial tensor concatenation step, LC successfully produces the correct implementation with proper tensor operations. This case demonstrates how LC's access to the entire codebase helps maintain consistency with the repository's tensor processing patterns — all similar preprocessing modules in the codebase use `torch.cat` for feature aggregation. The RAG approach, despite having access to relevant snippets, fails to capture this consistent pattern, suggesting that *sometimes having the complete context helps maintain global implementation consistency*.

In contrast, CLAUDE-3.5-Sonnet does not operate well in long-context mode, achieving only 15.6% pass@1. RAG performs substantially better — scoring 40.6% with a $4k$-token context; however, as the number of retrieved snippets increases to $16k$ and $32k$ tokens, the scores drop to 28.1% and 21.9% progressively. We manually inspect the failure cases and find that CLAUDE tends to generate more verbose and explanatory code, often adding unnecessary documentation and error handling that deviates from the repository's existing patterns. For example, as shown in Figure 2b, when completing a function in the `BaseSampler` class, CLAUDE adds a progress bar and additional parameter validation that are not present in similar functions within the codebase, suggesting its tendency to over-generalize based on its pre-training rather than adhering to the specific patterns employed by the repository at hand.

When LC Approaches RAG Performance Expectedly according to the base capabilities of the LMs, CODELLAMA completely fails when input contexts reach $80k$ tokens or beyond, in both RAG and LC settings. In contrast, the stronger Claude can still perform reasonably well till at least $110k$ tokens. Particularly on `deepmind/tracr` ($110k$ tokens) and `lucidrains/imagen-pytorch` ($83k$ tokens) repositories, CLAUDE can still operate comparably well, in comparison to RAG methods (36.3% and 38.8% respectively) despite the long context. While these two repositories are larger, they have consistent internal structure and clearer module boundaries, suggesting that well-structured codebases may offer more chances for LC to work.

When RAG Offers Clear Advantages Over LC Beyond traditional advantages in context filtering and selection, RAG exhibits superior performance in domain-specific code generation compared to LC. In our analysis of the `Google/lightweight_mmm` repository, CLAUDE achieves 25.0% success rate under RAG with the shortest context length ($4k$), but only 9.4% in LC mode. We identify two critical patterns in implementation accuracy. First, in mathematical implementations such as the Hill function (Figure 3a), LC produces code that is syntactically valid but se-

Task Input:

```
class Preprocessing(torch.nn.Module):
   def forward(self, features):
      # Process features with each module
          and aggregate using torch.cat
      processed = [module(feature)
         for module, feature in
         zip(self.preprocessing_modules,
             features)]
      return torch.cat(processed, dim=1)
```

CodeLlama-7B (RAG mode) [Pattern deviation]

```
def forward(self, features):
   """
   Args: features: [NxCxWxH]
   """
   processed = [module(feature)
      for module, feature in
      zip(self.preprocessing_modules, features)]
   # Missing tensor concatenation
   return processed
```

CodeLlama-7B (LC mode) [Pattern match]

```
def forward(self, features):
   """
   Args: features: [NxCxWxH]
   """
   return torch.cat([
      module(feature)
      for module, feature in
      zip(self.preprocessing_modules, features)
      ],
      dim=1)
```

(a) Preprocessing module implementation.

Task Input:

```
class BaseSampler(abc.ABC):
   @abc.abstractmethod
   def run(self, features):
      pass
```

Claude-3.5-Sonnet (LC mode) [Over-generalization]

```
class GreedyCoresetSampler(BaseSampler):
   def run(self, features):
      # Progress tracking for sampling
          process
      pbar = tqdm.tqdm(total=num_samples)
      # Parameter validation
      if not isinstance(features, np.ndarray):
         features = np.array(features)
```

CodeLlama-7B (LC mode) [Repository-specific pattern]

```
class GreedyCoresetSampler(BaseSampler):
   def run(self, features):
      # Direct sampling implementation
      samples = self._get_samples(features)
      return samples
```

(b) BaseSampler module implementation.

Figure 2: Example completions from `amazon-science/patchcore-inspection`. Left: shows the implementation of the `Preprocessing` class's forward method using CODELLAMA. In RAG mode, the solution stops after processing the features (highlighted in pink), missing the essential tensor concatenation step; in contrast, the LC mode solution correctly aggregates the features using `torch.cat`. Right: displays implementations for the `BaseSampler` class, where the CLAUDE output unnecessarily adds progress tracking and type validation (highlighted in pink), deviating from the existing class implementations, where CODELLAMA sticks to the available `self._get_samples` function.

mantically flawed: while `data^slope/(data^slope+half_max..^slope)` appears reasonable, it fails to match the canonical form `1/(1+(half_max../data) ^ slope)`. Second, when handling framework-specific code (Figure 3b), LC defaults to basic implementations like `dist.Normal(0, 2)`, overlooking essential framework conventions such as explicit parameter naming (`dist.HalfNormal(scale=2.)`) and distribution scale, potentially due to the difficulty of finding targeted patterns among numerous input code files in the long context.

These findings highlight a key limitation in current code generation approaches: while domain-specific implementations often adhere to established patterns, LC's access to the full codebase proves insufficient for discerning these specialized requirements. RAG, through targeted retrieval of relevant code segments, more effectively captures and reproduces these domain-specific implementations. This suggests that comprehensive context access may actually hinder performance in specialized coding tasks, where precise pattern recognition is crucial.

4 CODE SNIPPET ORDERING IS THE KEY

To investigate the most critical components of RAG that outperform LC method, we systematically examined the key differences between these two approaches. Our RAG retrieved 50-line code

Hill Function Implementation:

LC Implementation

```
def hill(data, half_max_effective_concentration,
      slope):
   return
      data**slope / (data**slope +
      half_max_effective_concentration**slope)
```

RAG Implementation

```
def hill(data, half_max_effective_concentration,
      slope):
   return
      1 / (1 + (half_max_effective_concentration
      / data) ** slope)
```

(a) Mathematical precision: RAG correctly implements the standard Hill function formula pattern, while LC generates a mathematically similar but functionally different implementation.

Numpyro Priors Definition:

LC Implementation

```
return {
   _INTERCEPT: dist.Normal(0, 2),
   _COEF_TREND: dist.Normal(0, 0.1),
   _EXPO_TREND: dist.Normal(0, 0.1)
}
```

RAG Implementation

```
return immutabledict.immutabledict({
   _INTERCEPT: dist.HalfNormal(scale=2.),
   _COEF_TREND: dist.Normal(loc=0., scale=1.),
   _EXPO_TREND: dist.Uniform(low=0.5, high=1.5)
})
```

(b) Framework-specific implementation: RAG follows numpyro's best practices with explicit parameter naming and correct distribution choices, while LC uses basic distribution calls that miss crucial framework-specific details.

Figure 3: Examples of domain-specific implementation discrepancies between LC and RAG methods in `google/lightweight_mmm`. While both methods produce syntactically correct code, RAG's ability to leverage fixed implementation patterns leads to more precise results in both mathematical formulas and framework-specific examples.

snippets with 5-line overlaps (denoted as *RAG/w50-o5*), which differs from LC in three aspects: overlapping lines (5-line in RAG vs. no overlap in LC), smaller snippet lengths (50-line in RAG vs. entire file in LC), and relevance-based ordering (vs. random ordering).

To isolate the impact of each factor, we conducted ablation experiments with the following setups: (1) *w50-o0*: we remove overlapping lines between adjacent code snippets, and chunk files into 50-line disjoint code snippets. (2) *w*inf*-o0/LC-sem*: to bridge the gap between the length of code snippets in RAG and LC settings, we perform retrieval on the file level in RAG. This setting could also be seen as an upgraded version of LC with semantic-based file ordering (i.e., *LC-sem*), compared to the random file ordering in the default LC method (i.e., *LC-rdm*).

Results of all settings are shown in Table 3.

Window Size and Overlap are Not Critical When using 50-line code snippets, comparing 5-line and zero overlap between adjacent snippets, we find that for most repositories and context lengths, there is no significant difference. This indicates that snippet overlap is not critical for chunking code snippets yet increases context lengths by about 20%. Comparing RAG using 50-line snippets and full code files, we also do not find substantial differences between them, suggesting that window size is not a decisive factor for RAG behavior as well.

Context Organization Matters Lastly, to investigate the impact of context organization, we compared file ordering by semantic relevance (*LC-sem*) and random ordering (*LC-rdm*). Based on LC's approach of randomly shuffling files with a fixed seed, we introduced max token constraints to simulate prompts of varying lengths for RAG, enabling a direct and fair comparison between semantic-relevance-based and random file ordering. As shown in Table 3, random ordering leads to large performance degradation across all repositories, implying that proper sequential organization of context information is crucial for code generation quality.

To further validate the importance of context organization, we also compared RAG's performance under the default setting (50-line snippets with 5-line overlap, *RAG/w50-o5*), additionally with randomly-shuffled snippet ordering (RAG (random)). As shown in Table 4, augmenting code snippets in random ordering consistently underperforms its semantic-ordered counterpart, i.e., the de-

Repository	Method	4k	8k	16k	32k	64k
amazon-science/patchcore-inspection	*RAG*	28.1	31.3	**37.5**	**34.4**	25.0
	w50-o0	28.1	**34.4**	34.4	**34.4**	**34.4**
	LC-sem	**31.3**	28.1	28.1	28.1	28.1
	LC-rdm	18.8	28.1	31.3	28.1	25.0
maxhumber/redframes	*RAG*	21.4	**19.1**	19.1	19.1	7.1
	w50-o0	19.1	14.3	19.1	14.3	11.9
	LC-sem	**26.2**	16.7	**26.2**	**21.4**	**16.7**
	LC-rdm	9.5	14.3	9.5	11.9	11.9
lucidrains/imagen-pytorch	*RAG*	49.3	52.2	**59.7**	**56.7**	**44.8**
	w50-o0	**52.2**	**58.2**	**59.7**	44.8	29.9
	LC-sem	43.3	46.3	56.7	41.8	29.9
	LC-rdm	3.0	4.4	7.5	10.4	1.5
deepmind/tracr	*RAG*	34.3	36.3	36.3	35.6	26.0
	w50-o0	33.6	**39.0**	**37.0**	32.2	20.6
	LC-sem	**37.7**	37.0	35.6	**39.7**	**26.7**
	LC-rdm	18.5	19.2	19.9	22.6	18.5
google/lightweight_mmm	*RAG*	15.6	**21.9**	**29.7**	**23.4**	**15.6**
	w50-o0	**17.2**	20.3	21.9	20.3	7.8
	LC-sem	15.6	**21.9**	23.4	18.8	12.5
	LC-rdm	12.5	7.8	17.2	12.5	7.8
facebookresearch/omnivore	*RAG*	18.2	18.2	27.3	18.2	**18.2**
	w50-o0	**22.7**	**31.9**	**31.8**	**22.7**	13.6
	LC-sem	18.2	18.2	18.2	**22.7**	13.6
	LC-rdm	18.2	18.2	13.6	22.7	4.5

Table 3: CodeLlama performance with different ablation setups under varying context lengths from $4k$ to $64k$. We bold the best method for each context length.

fault RAG method. This result reinforces our finding that semantic-based ordering plays a crucial role in code generation, as in §4.

Repository	RAG			RAG (Random)		
	4K	16K	32K	4K	16K	32K
amazon-science/patchcore-inspection	28.13	37.50	34.38	25.00	31.25	28.13
maxhumber/redframes	21.43	21.43	19.05	14.29	7.14	14.29
lucidrains/imagen-pytorch	49.25	59.70	56.72	22.39	38.81	38.81
deepmind/tracr	34.25	36.99	35.61	31.51	31.51	28.77
google/lightweight_mmm	17.19	29.69	21.88	15.63	26.56	18.75
facebookresearch/omnivore	18.18	27.27	18.18	18.18	18.18	22.73

Table 4: CODELLAMA performance when using semantically (left) and randomly ordered code snippets in retrieval-augmented generation.

These ablation studies reveal that proper context ordering can effectively boost long-context method performance across all repositories under various context lengths. The fact that performance is sensitive to context organization but not to window size or overlap suggests that future improvements in repository-level code generation might benefit more from better context planning strategies than from chunking optimizations.

5 Related Work

Repository-Level Code Completion Code generation has gradually shifted from single-snippet to more complex repository-level code generation (Zhang et al., 2023), as the increasing demand for automating software development with neural models (Jimenez et al., 2023). Despite the progress in code LMs such as StarCoder (Li et al., 2023) and DeepseekCoder (Guo et al., 2023) , repo-level code generation still poses numerous challenges, such as processing large-scale codebase content and understanding complex, cross-file dependencies.

Retrieval-Augmented Generation (RAG) RAG (Lewis et al., 2020) has been adopted as an effective approach for augmenting LM knowledge (Karpukhin et al., 2020) or alleviating LLM context limitations (Zhou et al., 2023; Zhang et al., 2023). RAG retrieved only top-relevant code snippets from the large-scale codebase, encouraging the model to focus on presumably relevant content, often boosting model performance while reducing computational costs. However, for long, complex code repositories, existing RAG methods face certain limitations (Wang et al., 2024), suggesting for more explorations in optimal RAG settings and alternative methods such as LC models.

Long-Context Language Models LLMs that support increasingly long contexts are showing increasing promises across a wide range of tasks, including repository-level code generation, that particularly stresses long, complex codebase input. Some works focus on pre-training LMs to support tens of thousands of tokens out-of-the-box (Beltagy et al., 2020; Guo et al., 2022b; Chen et al., 2023), while auxiliary modules are designed to further extrapolate model context limits during inference (Bertsch et al., 2023). Recent advances in API models such as GPT (OpenAI, 2023) or Claude (Anthropic, 2023) further push the context boundary to millions of tokens, offering potential in solving repo-level code generation. Built on these successes in LC models, we investigate the possibility of solving repo-level code generation with LC models.

6 Conclusion

Our comprehensive study demonstrates that long-context models can effectively solve repository-level code generation, but their success is highly contingent upon repository characteristics. In sufficiently small, well-structured repositories, LC models can match or outperform RAG approaches. However, LC performance deteriorates significantly in larger repositories with more complex structures, where RAG's selective context augmentation offers more advantages. Through ablation studies, we find that proper context organization, rather than specific chunking strategies, is fundamental to RAG performance. These findings provide clear guidelines for choosing between LC and RAG approaches based on repository properties, while highlighting the importance of adaptive context processing strategies in repository-level code generation systems.

7 Discussions and Future Directions

While our study provides valuable insights into the capabilities of long-context and retrieval-augmented models for repository-level code generation, there still exists some room for future investigation. First, although our experiments cover 373 code generation tasks, the best available source only allows us to perform analysis on 6 repositories, which may not offer sufficient diversity compared to the full spectrum of code repositories. This limited size also poses somewhat challenges for us in conducting robust, large-scale quantitative testing. Alternatively, we focus on qualitative analysis. We consider the findings in this work as preliminary insights to bootstrap larger-scale studies in the future.

Second, our comparison between LC and RAG approaches relies on semantic-based retrieval methods. While this provides strong baseline performance, more advanced neural retrieval approaches for code repositories developed in the future may yield different results, particularly in capturing complex dependencies across repository components.

The above areas suggest opportunities for future work to validate our findings with larger-scale repository analysis, and extend them with more advanced RAG and LC techniques.

REFERENCES

Anthropic. Claude. https://www.anthropic.com/, 2023.

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. In *ArXiv preprint arXiv:2004.05150*, 2020.

Amanda Bertsch, Uri Alon, Graham Neubig, and Matthew R Gormley. Unlimiformer: Long-range transformers with unlimited length input. *arXiv preprint arXiv:2305.01625*, 2023.

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tomas Sarlos, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020.

Daya Guo, Shuo Ren, Shuai Lu, Zhi Feng, Duyu Tang, Nan Duan, Ming Zhou, Pengcheng Yin, and Daxin Jiang. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, pp. 7212–7225, 2022a.

Daya Guo, Qiang Zhu, Dongyu Yang, et al. DeepSeek-Coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2301.14196*, 2023.

Mandy Guo, Joshua Ainslie, David C Uthus, Santiago Ontanon, Jianmo Ni, Yun-Hsuan Sung, and Yinfei Yang. LongT5: Efficient text-to-text transformer for long sequences. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pp. 724–736, 2022b.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. Retrieval augmented language model pre-training. In *International Conference on Machine Learning*, pp. 3929–3938. PMLR, 2020.

Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Few-shot learning with retrieval augmented language models. *arXiv preprint arXiv:2208.03299*, 2022.

Carlos E Jimenez, John Yang, Alexander Wettig, et al. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2023.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2020. URL https://aclanthology.org/2020.emnlp-main.550/.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pp. 9459–9474, 2020.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. StarCoder: May the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and LINGMING ZHANG. RepoQA: Evaluating long context code understanding. In *First Workshop on Long-Context Foundation Models @ ICML 2024*, 2024. URL https://openreview.net/forum?id=hK9YSrFuGf.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. ReACC: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, May 2022. URL https://aclanthology.org/2022.acl-long.431/.

OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Baptiste Roziere et al. Code Llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Zheng Zhang Wang, Akari Asai, Xinyun Yu, Furu Wei Xu, Yan Xie, Graham Neubig, and Daniel Fried. Coderag-bench: Can retrieval augment code generation? *arXiv preprint arXiv:2406.14497*, 2024.

Fuwen Zhang, Bo Chen, Yuyu Zhang, Jackson Keung, Jie Liu, Di Zan, Yong Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.151. URL https://aclanthology.org/2023.emnlp-main.151.

Shuyan Zhou, Uri Alon, Furu Wei Xu, Zhi Jiang, and Graham Neubig. DocPrompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*, 2023.

A STANDARDIZED PROMPT FORMAT

The prompt consists of two main parts: (1) a context section containing relevant code snippets from the repository, and (2) the incomplete code to be completed. The prompt data is structured as a JSON object with the following format:

```
{
  "prompt": "# Here are some relevant code fragments:
# -----------------------------------------
# the below code fragment can be found in:
# src/utils.py
# -----------------------------------------
# def process_input(data):        (1)
#     return data.to(device)
# -----------------------------------------

def forward(self, x):            (2)
    x = self.layer1(x)",         (3)
}
```

Figure 4: Example prompt format. Notes: (1) Retrieved context with # prefix, (2) Incomplete code without prefix, (3) Completion point.

The prompt contains:

- A context section with relevant code snippets (prefixed with #)
- The incomplete code to be completed (without # prefix)

B LC FAILS UNDER COMPLEX MODULE INTERDEPENDENCIES

On repositories with complex module interactions — such as those with extensive auxiliary documentation and intricate multimodal processing — the LC mode struggles to maintain proper code organization. As shown in Figure 5, when completing the `CkptExcludeKernel` class's `call` method, LC mode mistakenly embeds additional utility function definitions within the class method. This scope confusion highlights LC's fundamental limitation in filtering and selecting relevant context in large, interdependent codebases, whereas RAG mode, by focusing on the essential filtering logic, manages to preserve concise and correctly structured implementations.

Repository Context (Input):

```
class CkptExcludeKernel:
    """
    Removes the keys from the given model state_dict
    that match the key_pattern.
    Args:
        key_pattern: Patterns used to select the keys
        in the state_dict that are eligible for this kernel.
    """
    def __init__(self, key_pattern: List[str]):
        self.key_pattern = key_pattern
    def call(self, state_dict: Dict):
        """
        Args:
            state_dict: A dictionary representing
            the given checkpoint's state dict.
        """
```

Claude-3.5-Sonnet (RAG mode) (1) (Focused implementation)

```
exclude_keys = unix_pattern_to_parameter_names(
    self.key_pattern, state_dict.keys()
)
new_state_dict = {}
for key in state_dict:
    if key not in exclude_keys:
        new_state_dict[key] = state_dict[key]
return new_state_dict
```

Claude-3.5-Sonnet (LC mode) (2) (Unnecessary complexity)

```
def unix_pattern_to_parameter_names(
    constraints: List[str], parameter_names: List[str]
) -> Set[str]:
    """Converts unix patterns to parameter names
    Args:
        patterns: List of unix patterns to match.
        parameter_names: List to match against.
    Returns:
        Set of matched parameter names.
    """
    matched_names = set()
    for pattern in constraints:
        for name in parameter_names:
            if fnmatch.fnmatch(name, pattern):
                matched_names.add(name)
    return matched_names
exclude_keys = unix_pattern_to_parameter_names(
    self.key_pattern, state_dict.keys())
new_state_dict = {}
for key in state_dict:
    if key not in exclude_keys:
        new_state_dict[key] = state_dict[key]
return new_state_dict
```

Figure 5: Comparison of model outputs for the checkpoint kernel implementation. (1) RAG mode directly implements the core filtering logic without unnecessary overhead. (2) LC mode includes redundant function definition and documentation, making the code more complex without adding functionality. This demonstrates how RAG's focused context helps maintain code conciseness.