On the Importance of Context Filtering in Retrieval-Augmented Code Completion

Sergey Sedov
New York University
evolution3000futurama@gmail.com

Vsevolod Savinskiy
Constructor University Bremen
tirlimster@gmail.com

Andrei Arzhantsev
Technical University of Munich
arzhanandrey@gmail.com

***Abstract*—We present a retrieval-augmented pipeline for code completion task developed by our NoMoreActimel team during the JetBrains & Mistral AI Context Collection Competition. Our approach separates offline index pre-processing and online query processing parts. We highlight the importance of asymmetric approach to RAG in code completion tasks, separating index-specific and query-specific heuristics. We argue that stronger embedding models should perform increasingly better than BM-25 baselines when applied on large databases, which leads us to retrieval across all repositories instead of a single one. However, as our experiments show that sometimes less context is better, retrieval over larger code-bases increases the significance of *proper context filtering*. Therefore, we identify the main challenges of model-based RAG in code completion as poor context relevancy and extensive generality of chunk embeddings in particular. We focus on experiments with different chunking strategies, introducing a hole-centered query chunking strategy as our first modification that controls query relevance. We propose several reweighting penalties for similarity scores in order to increase relevancy of in-context chunks, penalizing by length and distance to completion hole. Filtering by simple similarity score thresholds also helps the final model performance. Besides that, we find that generation of short textual descriptions of completion target significantly improves metrics as well. While textual descriptions can be generated with much smaller model (1.5B) and token budget (1-2 sentences), they deal with context-overfitting compared to potential code-completion generations provided in-context. The described approach achieved top results: 1st in Python and 2nd in Kotlin private phases using lightweight Qwen 0.6B embedding model. Heavier Nomic 7B model gave the substantial lead of 13% to the second-best solution in the Python public phase leaderboard. The code is available on github.**

***Index Terms*—code completion, retrieval, RAG, embeddings, FAISS, ChrF, context composition**

I. Introduction

Context information is critical for code completion quality. The JetBrains & Mistral AI Context Collection Challenge evaluates submitted contexts via Fill-in-the-Middle prompting of three code models (Mellum, Codestral, Qwen2.5-Coder) and averages ChrF scores across models. To our understanding, the specific models have 4B, 22B and 14B of parameters correspondingly. This paper documents a compact, reproducible RAG [1] pipeline that builds a global index over all repositories and composes per-query contexts through multi-view retrieval with soft penalties. We highlight that, in case of separation between offline index creation and online query processing stages, mentioned ideas can be applied more extensively by exploiting pre-processing time. For instance, index doc-string enrichment could play a crucial role for retrieval quality, especially if used embedding model was not trained on less common languages, e.g. Kotlin.

Contributions: (1) A two-stage RAG system tailored to code completion task, with short description generation that consistently improves retrieval and final composition; (2) a length- and distance-aware scoring and robustness to context-window truncation; (3) evidence that cross-repository retrieval with strong code embeddings is beneficial compared to single-repository baselines; (4) evidence that completion text-descriptions work more robustly than suggested code completions when generated by a smaller model (5) comparison of time consumption by different parts of pipeline, some engineering notes on how to reduce I/O latency.

II. Method

We denote the code hole location by line index h. The context budget is $B = 16{,}384$ tokens afterwards truncated to model context-length from left.

A. *Offline indexing*

Index Chunking: We construct two chunk types over all repositories: (a) max-length chunks which include up to 1024 tokens, splitted by new-lines; (b) function-level segments separated by syntactic features, i.e. function and class definitions. Some methods (import-level chunks; generated doc-string chunks for functions) were implemented but excluded from final submissions due to pre-processing costs when offline and online parts are computed together.

Embeddings and storage: We embed all chunks using either *nomic-ai/nomic-embed-code* (7B) or *Qwen/Qwen3-Embedding-0.6B*. All embeddings are consolidated into a single FAISS index to avoid small-file I/O overhead. We found that single-file indices substantially reduce online-stage latency compared to per-repository `.npz` files.

Correctness and scope: Public/practice phases include timestamped repository variants. We additionally verified that the proposed cross-repository retrieval does not leak answers from future snapshots, i.e. that different data samples do not include answers for each other, which makes sense in the common application scenario. With strong embedding models, cross-repository retrieval was substantially better than single-repository retrieval, both being ahead of BM25 baseline.

B. *Online query processing: Query chunking*

Given the incomplete file (prefix, suffix), we form several types of chunks excluding semantics-based methods (which could be done in the future):

1) Full-file chunks as a baseline.
2) Max-length chunks that include up to 1024 tokens, splitted by new-lines. Basic approach that covers everything.
3) Centered windows around h that include $[h-m, h+m]$ lines of the incomplete file for various m.
4) Short natural-language task description produced by Qwen2.5-Coder 1.5B Instruct, which we append to every query chunk calling it the combined strategy.

We carefully select prompt and special tokens for description generation, limiting it to 128 token cap. Importantly, we append descriptions to suffices of final contexts, using them for index retrieval at the same time.

C. *Online query processing: Scores reweighting*

Query Chunks Distance Penalty: Common retrieval-augmentation approach is to compute embeddings over the whole query files. Frequently such approach poorly fits the code completion purposes as query files may contain a lot of task-irrelevant information. To deal with this, we introduce *distance-based score weighting* which assigns $[0, 1]$ weight to each query chunk based on the line-wise distance between chunk center and completion hole, normalized by the total number of lines in query. Denoting query as q, query chunk as qc and number of lines as $L(\cdot)$:

$$w_q(q, qc) = \frac{|center(qc) - h|}{L(q)} \tag{1}$$

Query Chunks Length Penalty: Distance-based weighting does not capture the size of query chunk, which generally correlates negatively with its relevance for the completion task. This is especially relevant for centered-chunking which is not affected by distance-based weights. Therefore, looking at final matching scores, we empirically selected *length-based scored weighting* parameters for centered query chunks only:

$$w_q(q) = \frac{1}{0.2 \cdot m + 1};\ m = 0.5 \cdot L(qc) \tag{2}$$

Index Chunks Length Penalty: As we aimed to retrieve over all repositories, proper filtering of large index chunks became more important. We mentioned that sometimes large file- or function- level chunks are getting the highest cosine-similarity scores, extensively consuming the limited context length. This becomes more relevant for models with smaller 8k context-length. We introduced softer length-penalty for index chunks which preserves mid-size chunks better than linear penalties:

$$w_i(ic) = 1 - (L(ic)/B)^2 \tag{3}$$

Retrieval and scoring: For each query chunk, we retrieve top-$k = 3$ index chunks using FAISS kNN cosine similarity search. We apply the aforementioned score reweighting followed by threshold filtering which drops all chunks under the threshold. Empirically we estimated similarity thresholds near 0.2 to be optimal, so the final index-chunk similarity score is:

$$s(qc, ic) = \langle \text{embed}(qc), \text{embed}(qi) \rangle \cdot w_q(qc) \cdot w_i(ic) \tag{4}$$

$$s(qc, ic) = s(qc, ic) \cdot \mathbb{I}\{s(qc, ic) \geq 0.2\} \tag{5}$$

We then insert chunks by descending s from right to left until the budget B is reached. This ordering ensures that any left-side trimming removes the least relevant context first. We use the $<|$file_sep$|>$ between chunks.

III. Implementation and efficiency

Models and Latency: Public phase runs mainly used Nomic 7B embeddings; private phase used Qwen3 0.6B for the better trade-off between final quality and compute requirements. The reason for using a lighter model is no separation between offline and online stages in private phase, so index creation was done along with query answering, consuming most of the time. In scenarios where user's code-base can be sent to cloud to be pre-processed in order to update the index, costly time consumption of embedding models drops significantly.

We used Qwen2.5-Coder 1.5B for generatng descriptions, carefully selecting prompt with special tokens and generation limits (1-2 sentences). Docker images cached light models; a helper script cached larger ones when needed.

Assuming <1 minute/query on an A40-class GPU, most time was spent on query embeddings, then description generation, then FAISS search and packing. Consolidating indices into single file gave the biggest latency improvement, batched query processing (meaning across different query chunks) also proved worthwhile.

We measured time consmuption for Python practice phase data that included only 47 repositories. Nomic 7B index creation took 3 hours to complete and query processing took 30 minutes. For Qwen 0.6B it took 30 and 4 minutes correspondingly. We conclude that Qwen 0.6B suits fast code completion purposes well enough, especially when index pre-processing can be done on the side. Possible scenarios of using Nomic 7B include distillation into smaller model, index pre-processing, careful query chunk selection and faster GPUs. We highlight that our method used extensive number of query chunks created by 4 different strategies. In case of stronger time constraints for online stage, number of calls to embedding model can be easily reduced by some heuristics on chunk selection.

IV. Results

A. *Public leaderboard*

Main Python submission used Nomic 7B embedding model.
Python (1st place): Average ChrF **0.7469** (Mellum 0.6950, Codestral 0.8034, Qwen 0.7424).
Kotlin (4th place): Average ChrF **0.6590** (Mellum 0.6115, Codestral 0.7139, Qwen 0.6515). We had not submitted the latest solution to Kotlin, but saw direct correlation between Python and Kotlin results among first submissions.

B. ***Private phase***

Both submissions used Qwen 0.6B embedding model.
Python (1st place): Average ChrF **0.734** (Mellum 0.656; Codestral 0.820; Qwen 0.725).
Kotlin (2nd place): Average ChrF **0.731** (Mellum 0.684; Codestral 0.791; Qwen 0.719).

V. Ablations and observations

1) *Cross-repository retrieval* enables stronger gains than swapping embedders in single-repository settings.
2) *Short descriptions* by smaller models can improve retrieval at negligible cost compared to candidate-code generation or beam search.
3) *Score reweighting methods* balance diversity of both index and query chunk types that contribute to context; linear variants of index-chunk length penalty excessively suppress mid-size functions.
4) Conservative thresholds reduce noise without starving the budget, this is an important hyperparameter and needs to be carefully selected for the particular embedding model and language.

VI. Development timeline

1) Research stage:
   a) Search for existing solutions, common chunking strategies and strong code embedding models
   b) Formulate the specifics of RAG applications for Code Completion, e.g. different chunking for query, problems of chunk length-imbalance and completion-irrelevance
   c) Create a plan for development, manage team-work
2) Baseline solution: max-length index, single whole-query chunk, strong Nomic model
3) Full code-base refactoring, separate into online and offline stages.
4) Query-chunking: max-length, full-file and centered query chunks. Top-k for FAISS.
5) Introduce short description generation; tune prompt to keep 1–2 sentences.
6) Use description both in retrieval and always in suffix of the final context.
7) Add distance- and length- penalties for query chunks to increase retrieval from near the hole.
8) Add quadratic length penalty for index chunks in order to deal with oversize chunks.
9) Migrate to cross-repo retrieval
10) Latency: consolidate FAISS index into a single file, batched query chunks processing.
11) Private phase: switch to Qwen3 0.6B embeddings for speed/memory.

VII. Limitations and future work

A. ***Hierarchical retrieval.***

Retrieval time ultimately grows with index size. For larger indexes hierarchical retrieval may be the optimal solution: first, per-file or per-repo clusters are being formed; each cluster is getting assigned an embedding by total summary and metadata; query chunks are first matched with cluster-embeddings, and then re-matched within each cluster. Instead of per-repo approach, cluster assignment may be done on the embedding level as well - just clusterize embeddings of default chunks and select the mean of each cluster as the new embedding.

B. ***Query chunks irrelevance.***

As we have discussed, it is frequently incorrect to assume that query chunks contain sufficient information about the completion task. The next step in our plan was to deal with this problem via specific hierarchical retrieval design. Namely, we suggest a following strategy for improving context-chunks relevance:

1) All index chunks are being assigned a cluster label, e.g. filename, by which we can filter the index.
2) First, query chunks are getting matched with index with the goal to find files similar to query (regularly some files do not fit into context length of embedding model). Import-level chunks may be good at this stage.
3) After the most relevant files for the current query file were found, we perform second-stage chunk matching within them, only based on generated query text description. This should help us find ”missing parts” and control the noise from pure text-to-code matching over the whole index.

VIII. Conclusion

We presented an effective RAG pipeline for Code Completion task that has proven itself by achieving great results in the JetBrains & Mistral AI competition leaderboard. Our approach is based on the cosine-similarity top-k retrieval of relevant chunks from the index over all repositories. It divides the task into offline index-preprocessing and online query-answering stages, explores different chunking strategies, benefits from short text descriptions generation and balances chunks in context via several score reweighting mechanisms.

References

[1] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in *NeurIPS*, 2020.

[2] M. Popović, “chrF: character n-gram F-score for automatic MT evaluation,” in *WMT*, 2015.

[3] J. Johnson, M. Douze, and H. Jégou, “Billion-scale similarity search with GPUs,” *IEEE TPAMI*, 2019.

[4] S. Robertson and H. Zaragoza, “The probabilistic relevance framework: BM25 and beyond,” *Foundations and Trends in Information Retrieval*, 2009.