Adapting Dense Retrievers with Synthetic Multi-Positive Chunk Supervision for COBOL Code Retrieval

Anonymous ACL submission

Abstract

Semantic retrieval of COBOL code is important for search, maintenance and program understanding in legacy systems but remains challenging due to domain-specific vocabulary, repetitive structures and limited training data. We study encoder-based COBOL retrieval using a pipeline based on synthetic query generation, multi-passage data construction, sampled/all-positive training objectives and positive-aware hard negative mining. Synthetic queries are generated using Codestral-22B and GPT-OSS-120B and their impact on retrieval quality is evaluated. Experiments on a corpus of 27K passages and 64 human-verified queries[1] show that synthetic query–code fine-tuning improves retrieval quality, with granite-125m improving Recall@20 from 71.69 to 78.85. Codestral-generated queries outperform GPT-OSS-generated queries and hard negative mining improves most Granite models. These results demonstrate that domain-adapted synthetic supervision is an effective and scalable strategy for COBOL code retrieval.

1 Introduction

COBOL systems often contain large files, repetitive implementation patterns and business logic distributed across multiple interconnected programs, making traditional keyword-based retrieval insufficient for capturing semantic intent (Dau et al., 2024; Maharaj et al., 2026). While large decoder-based language models (Chen et al., 2024b) (Guo et al., 2024) (Rozière et al., 2023) can support code understanding tasks, applying them directly for retrieval and question answering over large enterprise repositories can be computationally expensive, difficult to scale and sensitive to context window limitations. Embedding-based retrieval models(Karpukhin et al., 2020; Reimers and Gurevych, 2019; Xiong et al., 2020; Lin et al., 2022; Zhang et al., 2024) provide a more efficient alternative by enabling semantic matching between natural language queries and code artifacts through dense vector representations. However, effective retrieval in COBOL remains challenging due to domain-specific terminology, duplicated logic and limited availability of high-quality training data. To address these challenges, this work focuses on improving retrieval quality through synthetic query generation, stronger training objectives and hard negative selection strategies.

COBOL retrieval often requires aggregating information across multiple related code fragments rather than a single passage. For example, Figure 1 shows a query whose answer is distributed across several programs, where each passage captures part of the logic. This highlights the inherently multi-positive nature of COBOL retrieval and motivates training approaches that leverage multiple relevant passages.

Due to the limited availability of supervised query–code retrieval datasets for COBOL, synthetic queries were generated using Codestral-22B (Mistral AI, 2024) and GPT-OSS-120B (Agarwal et al., 2025) to compare code-specialized and general-purpose query generators. The training workflow compares Joint Likelihood (JointLH) with Random One-Positive Likelihood (Rand1LH), where Rand1LH acts as a sampled single-positive baseline from a multi-positive set (Wang et al., 2026). This is useful for COBOL retrieval because related logic may appear across multiple code sections or programs. In addition, positive-aware hard negative mining is used to select more meaningful negative examples by considering the score of known positive passages, reducing false negatives during training (de Souza Pereira Moreira et al., 2024).

Overall, the work aims to improve COBOL code retrieval by combining synthetic query gener-

[1]The code, trained models, and datasets used in this work will be released publicly upon publication.

ation, passage grouping, comparison of all-positive & sampled single-positive training objectives and positive-aware hard negative mining.

Our contributions are:

1. Construction of novel COBOL retrieval dataset with human-verified multi-passage relevance annotations for realistic program understanding tasks.

2. Systematic study of synthetic query generation using both code-focused and general-purpose language models, highlighting differences in their impact on retrieval performance.

3. Analysis of positive-aware hard negative mining and its impact on improving retrieval robustness in the presence of structurally similar COBOL code.

2 Methodology

The methodology focuses on improving COBOL code retrieval through a pipeline consisting of passage grouping, synthetic query generation, loss-function comparison and positive-aware hard negative mining. The main objective is to create realistic multi-passage query-code training examples and train a retriever that can rank relevant COBOL passages higher than similar but incorrect passages.

2.1 Passage Grouping

COBOL programs sourced from one private enterprise application (797) and X-COBOL dataset (1255) (Ali et al., 2023) are segmented into passage-level units using code-aware boundaries resulting in 27K & 10K passages respectively. We construct candidate passage groups using embedding-based similarity: each passage is encoded with a Granite embedding model, ranked by cosine similarity and grouped greedily using a maximum group size of 5. Training instances are derived from these candidate groups by selecting one or more passages as positives during query generation. This produces both focused single-passage examples and broader multi-passage examples, enabling the retriever to learn relevance signals for individual COBOL fragments as well as related code fragments spanning multiple program locations (Karpukhin et al., 2020; Su et al., 2024).

2.2 Synthetic Query Generation

Synthetic queries were generated from grouped COBOL passages using large language models, including Codestral-22B (Mistral AI, 2024) and GPT-OSS-120B (Agarwal et al., 2025). For each group, we construct prompts that require generating a single natural language query that depends on multiple passages, emphasizing relationships such as control flow, data flow and cross-section dependencies. We use few-shot prompting and sample 1–5 passages per group to create diverse training instances, treating all sampled passages as positives (Wang et al., 2026). This resulted in overall 7550 training instances.

2.3 Loss Function Comparison

We compare Joint Likelihood (JointLH) with Random One-Positive Likelihood (Rand1LH) from (Wang et al., 2026). JointLH optimizes over all positive passages for a query, while Rand1LH randomly samples one positive passage at each training iteration and applies the standard InfoNCE loss to that sampled query–positive pair. This comparison evaluates whether using all available positives is more effective than random single-positive sampling for COBOL retrieval, where relevant logic may span multiple code fragments.

2.4 Positive-Aware Hard Negative Mining

Hard negatives are irrelevant passages that are highly similar to the query and therefore more informative than random negatives. During contrastive training, hard negatives help the model learn finer decision boundaries by pushing similar but irrelevant passages away from the query representation. However, in COBOL, similar code may share variables, file operations, or control-flow patterns with gold passages, increasing the risk of false negatives. We mine one hard negative per query by retrieving candidates with embedding similarity and filtering them relative to the known positive score. This follows NV-Retriever's positive-aware mining approach (de Souza Pereira Moreira et al., 2024). We also exclude candidates from the same program or document as any gold positive to avoid treating related COBOL logic as negative evidence.

2.5 Training and Comparison

The final training data was prepared using grouped COBOL passages, generated multi-passage queries, positive passages and mined hard negatives. Models were trained using JointLH and Rand1LH objectives to compare all-positive training against random single-positive sampling (Wang et al., 2026). The generated queries from Codestral-22B (Mistral AI, 2024) and GPT-OSS-120B (Agarwal et al.,

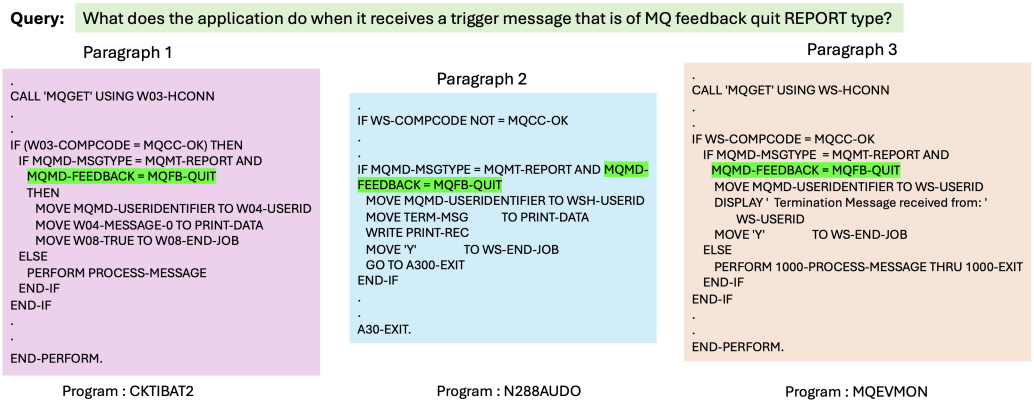

Figure 1: Example of multi-positive retrieval demonstrating several correct matches for a single query.

2025) were also compared to understand which model produced more useful and grounded multi-passage retrieval queries. The overall evaluation focused on whether the retriever could retrieve relevant COBOL code passages more accurately for complex queries involving program logic, field flow, file handling and validation behavior.

3 Dataset

3.1 Test Set

The evaluation benchmark was designed to assess retrieval performance on realistic COBOL understanding tasks requiring both single-file and multi-file reasoning across one or more relevant passages. Therefore we only considered 27K passages extracted from private enterprise repository (2.1). This corpus covers a broad range of business logic patterns, including file handling, validation routines, control-flow processing, table operations, status management and business-rule implementation.

Relevant annotations of query and the result passage(s) were constructed using a hybrid annotation workflow combining LLM-assisted candidate generation with human verification. Candidate positive passages were initially identified through passage-grouping strategies and retrieval-assisted generation pipelines. These candidate annotations were subsequently reviewed and validated by two SMEs to ensure that the selected passages were semantically relevant and necessary for accurately answering the corresponding query. The final test set came to 64 retrieval queries.

4 Experiments

4.1 Experimental Setup

We fine-tune five encoder-based embedding models on synthetic COBOL query–code pairs, including two Granite RoBERTa-based models, two Granite ModernBERT-based models and one GTE ModernBERT-based model. Training uses both in-batch negatives and one mined hard negative per query (NUM_HN=1). All models are trained for 1500 steps with batch size 64, learning rate 2×10^{-6}, the default Hugging Face `AdamW` optimizer and `[CLS]` pooling. Evaluation is performed using Recall@20 on the held-out test set. Experiments are run on a single NVIDIA A100 80GB GPU with 64GB system memory.

Model	Recall@20
bge-code-v1	32.16
granite-small-r2	65.96
granite-r2	67.58
granite-30m	69.35
granite-125m	71.69
nomic-embed-code	73.26
Qwen3-Embedding-0.6B	74.22
gte-modernbert	74.24
Qwen3-Embedding-4B	76.04
CodeRankEmbed	76.25
Qwen3-Embedding-8B	**76.35**

Table 1: Recall@20 performance of embedding models on COBOL code retrieval task

Model	GPT-OSS R@20	Codestral R@20
granite-30m	68.57	70.13
granite-small-r2	68.57	70.65
granite-r2	72.01	74.09
gte-modernbert	**74.68**	75.05
granite-125m	74.30	**78.85**

Table 2: Recall@20 comparison of models fine-tuned using synthetic queries generated by GPT-OSS and Codestral on the test set.

Model	JointLH R@20	Rand1LH R@20
granite-30m	67.06	70.13
granite-small-r2	67.79	70.65
granite-r2	**73.18**	74.09
gte-modernbert	72.24	75.05
granite-125m	73.10	**78.85**

Table 3: Recall@20 comparison between JointLH and Rand1LH loss functions on the test set.

Model	Base R@20	Best FT R@20
granite-30m	69.35	70.13
granite-small-r2	65.96	70.65
granite-r2	67.58	74.09
gte-modernbert	**74.24**	77.21
granite-125m	71.69	**78.85**

Table 4: Recall@20 comparison between base embedding models and their best fine-tuned variants on the test set.

4.2 Models and Baselines

We compare encoder-based embedding retrievers with decoder-based embedding retrievers. The encoder baselines include Granite embedding models (Awasthy et al., 2025), gte-modernbert-base (Li et al., 2023) and CodeRankEmbed (Suresh et al., 2024). We also evaluate decoder-based embedding retrievers including bge-code-v1 (Chen et al., 2024a), nomic-embed-code (Suresh et al., 2024), and Qwen3-Embedding models (Zhang et al., 2025).

Table 1 reports zero-shot Recall@20. Qwen3-Embedding-8B gives the best result with 76.35, followed by CodeRankEmbed at 76.25 and Qwen3-Embedding-4B at 76.04. Smaller encoder models remain competitive with granite-125m reaching 71.69 Recall@20. These results suggest that decoder-based embedding retrievers provide strong zero-shot accuracy, while encoder-based retrievers offer a better efficiency trade-off for large-scale COBOL retrieval.

4.3 Effect of Synthetic Query–Code Fine-tuning

Table 4 shows that synthetic COBOL query–code fine-tuning improves Recall@20 across all evaluated encoders. The largest gain is achieved by `granite-125m`, improving from 71.69 to 78.85 (+7.16), followed by `granite-r2` from 67.58 to 74.09 (+6.51) and `granite-small-r2` from 65.96 to 70.65 (+4.69). In contrast, `granite-30m` shows only a modest gain (+0.78), suggesting that mid-sized encoders benefit more from domain-specific synthetic supervision while remaining efficient for COBOL retrieval.

Table 2 compares the effect of synthetic query generator choice. Codestral-generated queries improve Recall@20 over GPT-OSS-generated queries for all evaluated models, with the best result obtained by `granite-125m` at 78.85. This suggests that code-focused synthetic query generation provides stronger supervision for COBOL retrieval fine-tuning.

Model	w/o HN R@20	with HN R@20
granite-30m	69.92	70.13
granite-small-r2	70.65	70.65
granite-r2	71.74	74.09
gte-modernbert	77.21	75.05
granite-125m	**77.34**	**78.85**

Table 5: Effect of positive-aware hard negative (HN) mining on COBOL retrieval performance.

4.4 Hard Negative Mining Analysis

Table 5 shows that positive-aware hard negative mining improves Recall@20 for most evaluated models. The best result is achieved by `granite-125m`, reaching 78.85 Recall@20 with hard negatives, while `granite-r2` shows the largest gain of +2.35. These results suggest that hard negatives help the retriever distinguish similar-looking COBOL passages, such as repeated file-read routines, status checks, table operations and error-handling blocks. Overall, positive-aware hard negative mining improves granite-family retrieval performance and provides the strongest fine-tuned result.

5 Conclusion

We studied COBOL code retrieval using synthetic query–code fine-tuning and positive-aware hard negative mining. Fine-tuning improves retrieval performance, with `granite-125m` reaching 78.85 Recall@20. Codestral-generated queries outperform GPT-OSS-generated queries, and hard negative mining improves most Granite models. Overall, compact encoder retrievers are effective for scalable COBOL semantic search.

Limitations

The evaluation is limited to a 64-query test set built from COBOL programs in enterprise repositories. Since the experiments were not conducted on public code repositories, the generalization of these results to broader code retrieval settings remains future work.

References

OpenAI Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Hai-Biao Bao, Boaz Barak, Ally Bennett, Tyler Bertao, N. Archer Brett, Eugene Brevdo, Greg Brockman, Sébastien Bubeck, Cheng Chang, Kai Chen, and 105 others. 2025. gpt-oss-120b&gpt-oss-20b model card.

Mir Sameed Ali, Nikhil Manjunath, and Sridhar Chimalakonda. 2023. X-cobol: A dataset of cobol repositories. *ArXiv*, abs/2306.04892.

Parul Awasthy, Aashka Trivedi, Yulong Li, Mihaela A. Bornea, David Cox, Abraham Daniels, Martin Franz, Gabe Goodhart, Bhavani Iyer, Vishwajeet Kumar, Luis A. Lastras, Scott McCarley, Rudra Murthy, P Vignesh, Sara Rosenthal, Salim Roukos, Jaydeep Sen, Sukriti Sharma, Avi Sil, and 3 others. 2025. Granite embedding models. *ArXiv*, abs/2502.20204.

Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024a. M3-embedding: Multilinguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In *Annual Meeting of the Association for Computational Linguistics*.

Yuxuan Chen, Mingwei Liu, Guangsheng Ou, Anji Li, Dekun Dai, Yanlin Wang, and Zibin Zheng. 2024b. Are decoder-only large language models the silver bullet for code search? *IEEE Transactions on Software Engineering*, 52:1215–1233.

Anh TV Dau, Hieu Trung Dao, Anh Tuan Nguyen, Hieu Trung Tran, Phong X Nguyen, and Nghi DQ Bui. 2024. Xmainframe: A large language model for mainframe modernization. *arXiv preprint arXiv:2408.04660*.

Gabriel de Souza Pereira Moreira, Radek Osmulski, Mengyao Xu, Ronay Ak, Benedikt Schifferer, and Even Oldridge. 2024. Nv-retriever: Improving text embedding models with effective hard-negative mining. *ArXiv*, abs/2407.15831.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *ArXiv*, abs/2401.14196.

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Yu Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. 2020. Dense passage retrieval for open-domain question answering. *ArXiv*, abs/2004.04906.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards general text embeddings with multi-stage contrastive learning. *ArXiv*, abs/2308.03281.

Sheng-Chieh Lin, Minghan Li, and Jimmy J. Lin. 2022. Aggretriever: A simple approach to aggregate textual representations for robust dense passage retrieval. *Transactions of the Association for Computational Linguistics*, 11:436–452.

Kishan Maharaj, Nandakishore Menon, Ashita Saxena, and Srikanth Tamilselvam. 2026. Robustness and reasoning fidelity of large language models in long-context code question answering. *ArXiv*, abs/2602.17183.

Mistral AI. 2024. Codestral-22b-v0.1. `https://huggingface.co/mistralai/Codestral-22B-v0.1`. Accessed: 2026-05-26.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *ArXiv*, abs/1908.10084.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, I. Evtimov, Joanna Bitton, Manish P Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre D'efossez, Jade Copet, and 6 others. 2023. Code llama: Open foundation models for code. *ArXiv*, abs/2308.12950.

Hongjin Su, Howard Yen, Mengzhou Xia, Weijia Shi, Niklas Muennighoff, Han Wang, H. Liu, Quan Shi, Zachary S. Siegel, Michael Tang, Ruoxi Sun, Jinsung Yoon, Sercan Ö. Arik, Danqi Chen, and Tao Yu. 2024. Bright: A realistic and challenging benchmark for reasoning-intensive retrieval. *ArXiv*, abs/2407.12883.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. 2024. Cornstack: High-quality contrastive data for better code retrieval and reranking. In *International Conference on Learning Representations*.

Benben Wang, Minghao Tang, Hengran Zhang, Jiafeng Guo, and Keping Bi. 2026. Training dense retrievers with multiple positive passages. *ArXiv*, abs/2602.12727.

Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul N. Bennett, Junaid Ahmed, and Arnold Overwijk. 2020. Approximate nearest neighbor negative contrastive learning for dense text retrieval. *ArXiv*, abs/2007.00808.

Han Zhang, Yunjing Jiang, Mingming Li, Haowei Yuan, and Wen-Yun Yang. 2024. pebr: A probabilistic approach to embedding based retrieval. *ArXiv*, abs/2410.19349.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. 2025. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *ArXiv*, abs/2506.05176.

A Appendix

This appendix provides additional experimental results and supplementary analyses supporting the findings presented in the main paper. Specifically, we include comparisons between single-passage and multi-passage synthetic query generation strategies for COBOL retrieval, GPU compute usage across fine-tuned embedding models, and additional details about the training dataset construction and augmentation process.

A.1 Additional Results

Model	Single Passage Generation R@20	Multi Passage Generation R@20
granite-30m	69.35	70.13
granite-small-r2	66.22	70.65
granite-r2	71.82	74.09
gte-modernbert	75.52	75.05
granite-125m	**76.12**	**78.85**

Table 6: Recall@20 comparison of models fine-tuned using synthetic queries generated with single-passage and multi-passage generation on the test set.

Model	Params	GPU Hrs
granite-30m	30M	0.31
granite-small-r2	47M	0.20
granite-125m	125M	0.79
granite-r2	149M	0.63
gte-modernbert	149M	0.63

Table 7: GPU compute usage for fine-tuned embedding models.

A.2 Train Dataset

To further improve diversity and coverage, we augment this process using the X-COBOL dataset (Ali et al., 2023), a large-scale collection of COBOL repositories. Passages sampled from X-COBOL are incorporated into the same multi-passage query generation pipeline, exposing the model to a broader range of programming patterns, business logic variations, and repository-level structures. Queries generated from these external samples follow the same prompting strategy, ensuring consistency in supervision while expanding the distribution of training data beyond the primary corpus. Using the strategy, we generated 2713 queries on the private enterprise COBOL corpus and 4837 queries on X-COBOL corpus.

B Appendix

B.1 Query Generation Prompt

You are a question generation specialist for COBOL mainframe systems. Given multiple COBOL passages, generate **one natural language query** that can only be answered by combining information from **most or all of the passages**.

Requirements

- The query must require comparison, combination, or reasoning across passages.
- It must not be answerable from a single passage alone.
- Focus on aspects such as:
  - control flow or data flow
  - validation logic
  - file handling and status codes
  - table search and lookup behavior
  - error handling
  - dependencies across routines

Specificity and grounding

- Use concrete technical elements when appropriate:
  - paragraph names
  - file names
  - field names
  - variables or tables
- Avoid generic or vague queries.

Query style

- The query must begin with one of the following: *How, What, Which, When, Where*
- The query should:
  - read like a realistic developer query

  - ask a single integrated question

- The query should not:
  - ask separately about each passage
  - be a yes/no question
  - refer to the passages or provided context explicitly

Forbidden phrases

- Do not use phrases such as:
  - "according to the passages"
  - "based on the passages"
  - "from the provided code"
  - "in the above snippets"

- The query must read as a **standalone retrieval query**.

Input Format

- Input is provided as a field named `document`

- The document contains multiple passages labeled:
  - Passage 1, Passage 2, Passage 3, . . .

Output Format The model must return only valid JSON with the following structure:

```
{
  "query": "..."
}
```

- No additional text or explanation should be included.

- No markdown formatting should be used.

Generation Settings

- Temperature: 0.8

- Top-p: 0.9

- Max tokens: 4096

Note The generated query is designed to simulate real-world COBOL retrieval scenarios, where answering requires synthesizing information across multiple program fragments rather than relying on a single passage.

Model	Retrieval@5	Rerank@5
granite-30m	0.5341	0.6273
granite-small-r2	0.5753	0.6122
granite-r2	0.5695	0.5979
gte-modernbert	**0.5935**	0.6305
granite-125m	0.5383	**0.6539**

Table 8: Comparison of Recall@5 between retrieval and reranking across the best finetuned models

Table 8 compares Recall@5 for first-stage retrieval and second-stage reranking across models. Reranking improves Recall@5 for all models, showing its ability to refine the ranking of initially retrieved candidates. The gains are more pronounced for smaller models such as `granite-30m` and `granite-125m`, suggesting that reranking compensates for weaker initial representations. In contrast, models with stronger baseline retrieval performance, such as `gte-modernbert`, exhibit smaller gains, indicating limited headroom for further ranking improvements. These results highlight the importance of a two-stage retrieval pipeline, particularly for compact embedding models.