Efficient Code Embeddings from Code Generation Models

Daria Kryvosheieva[1,2]* **Saba Sturua**[2] **Michael Günther**[2]
Scott Martens[2] **Han Xiao**[2]

[1]Massachusetts Institute of Technology [2]Jina AI GmbH
Prinzessinnenstraße 19, 10969, Berlin, Germany
research@jina.ai

Abstract

`jina-code-embeddings` is a novel code embedding model suite designed to retrieve code from natural language queries, perform technical question-answering, and identify semantically similar code snippets across programming languages. It makes innovative use of an autoregressive backbone pre-trained on both text and code, generating embeddings via last-token pooling. We outline the training recipe and demonstrate state-of-the-art performance despite the relatively small size of the models, validating this approach to code embedding model construction.

1 Introduction

The rapid adoption of AI-powered development environments like Cursor and Claude Code has transformed software engineering, with code embedding models serving as a critical foundation for retrieval and context engineering of these systems.

Code embedding models, starting with Gu et al. [2018], have evolved into a well-established subfield with dedicated benchmarks [Husain et al., 2019, Li et al., 2025] and leaderboards [Enevoldsen et al., 2025]. While code generation models like Codex [Zaremba and Brockman, 2021] can directly synthesize code from natural language prompts, practical code generation requires contextual understanding of existing codebases, API usage patterns, and integration requirements. This naturally positions code generation systems as retrieval-augmented generation (RAG) architectures [Lewis et al., 2020], where embedding models serve as the critical retrieval component.

Despite their importance, current code embedding models face a fundamental training data limitation. Supervised training typically relies on aligned data such as inline comments, documentation strings, and pedagogical examples from technical documentation—sources that provide insufficient semantic grounding for complex real-world development scenarios. In contrast, the abundant unaligned code and natural language documentation used to train modern LLMs remains largely underutilized for embedding model development.

We address this gap by introducing two high-quality code embedding models: `jina-code-embeddings-0.5b` and `1.5b`, with 494 million and 1.54 billion parameters, respectively. Our approach implements several key innovations: First, we leverage dedicated pre-trained code generation LLMs as backbones, adapting them specifically for embedding generation. Second, through comprehensive task analysis across functional areas of code embedding applications, we develop targeted training strategies that optimize performance for each use case. The resulting models achieve significant improvements over previous models of comparable size, with benchmark performance competitive with much larger alternatives.

*Work done during internship at Jina AI.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: DL4CODE.

2 Related Work

General-purpose semantic embedding models can function as code embedding models, and if large enough and trained on sufficient relevant data, can even perform comparably with specialized models. For example, among recent general-purpose text embedding models, Qwen3 Embedding [Zhang et al., 2025] and Gemini Embedding [Google DeepMind, 2025] perform well on code tasks. However, general-purpose models that support all kinds of texts are very large and expensive to train and deploy.

Many code embedding models are based on variants of the BERT model [Devlin et al., 2019], like CodeBERT [Feng et al., 2020] and `jina-embeddings-v2-base-code` [Günther et al., 2024]. However, this approach requires specialized datasets and struggles with adequate data sources.

An alternative approach is to take pre-trained general-purpose embedding models and adapt them to code, leveraging their existing language knowledge to compensate for shortcomings in aligned natural language and code data. Sturua et al. [2024] use a LoRA adapter [Hu et al., 2022] to specialize the general-purpose `jina-embeddings-v3` model for code retrieval.

Recent developments have demonstrated that autoregressive decoder architectures can be adapted to generate high-quality embeddings [Lee et al., 2025]. This class of model generates tokens sequentially as a function of the preceding tokens and can be trained using standard denoising objectives and copious quantities of readily available natural language text. They can then be adapted, after pre-training, to produce high-quality embeddings by pooling the token embedding outputs of the last hidden layer and fine-tuning the model. This is the approach used in the Qwen3 embedding model family [Zhang et al., 2025], and in `jina-embeddings-v4` [Günther et al., 2025].

Many different pooling methods have been explored for embedding model generation. Reimers and Gurevych [2019] found *mean pooling* to perform better than *max pooling* or *CLS pooling*[2] for encoder-based transformer models. For decoder-only models, embeddings are most commonly generated via last-token pooling Wang et al. [2024]. Recently, Lee et al. [2025] have proposed a novel pooling scheme they call *latent attention pooling*, inspired by the transformer architecture, which has trainable weights. They report significant improvements in general embedding performance using this method.

Instruction-tuning for specific domains and task types generally yields improved performance [Su et al., 2023]. Adding instructions to each text before generating an embedding trains the network to produce embeddings that reflect that instruction. The Qwen3 model [Zhang et al., 2025], which implements instruction-tuning, provides for user-generated instructions, making it difficult to optimize performance and leading to uncertainty about how the model will behave.

3 Model Architecture and Task Prefixes

`jina-code-embeddings-0.5b` and `1.5b` employ an autoregressive decoder backbone architecture, i.e., a model which generates tokens sequentially as a function of the preceding tokens. They build on the `Qwen2.5-Coder-0.5B` and `Qwen2.5-Coder-1.5B` backbones [Hui et al., 2024], which are both very compact LLMs.

The final hidden layer of the LLMs is transformed into an embedding via last-token pooling. We found, after some experimentation, that last-token pooling gave us better performance than mean pooling or latent attention pooling, as documented in Appendix B. CLS pooling was not tested, but is generally not favored for decoder-only architectures.

We analyzed downstream code embedding tasks and divided them into five categories: Natural language to code retrieval (**NL2Code**), technical question answering (**TechQA**), code-to-code retrieval (**Code2Code**), code to natural language retrieval (**Code2NL**), and code to completion retrieval (**Code2Completion**).

For each of these tasks, we created an instruction string in English that prefixes the text passed to the model, listed in Table 1. Different instruction strings are used for queries and for documents.

[2]CLS pooling uses the output embedding of a pre-pended `[CLS]` token as the final embedding vector.

Table 1: Task categories and their corresponding instruction prefixes.

Task type	Query prefix	Document prefix
NL2Code	`"Find the most relevant code snippet given the following query:\n"`	`"Candidate code snippet:\n"`
TechQA	`"Find the most relevant answer given the following question:\n"`	`"Candidate answer:\n"`
Code2Code	`"Find an equivalent code snippet given the following code snippet:\n"`	`"Candidate code snippet:\n"`
Code2NL	`"Find the most relevant comment given the following code snippet:\n"`	`"Candidate comment:\n"`
Code2Completion	`"Find the most relevant completion given the following start of code snippet:\n"`	`"Candidate completion:\n"`

4 Training

We initialized the model with weights of the pre-trained backbone `Qwen2.5-Coder-0.5B` and then applied further training with a contrastive objective using the InfoNCE loss function [van den Oord et al., 2019]. Pairs of inputs are classed as related or unrelated, and the model learns to embed related items closely together and unrelated items further apart.

We use Matryoshka representation learning [Kusupati et al., 2022] during training to produce truncatable embeddings, so users can make flexible trade-offs between precision and resource usage.

4.1 Training Data

The training data consists of query-document pairs for a variety of code retrieval tasks, largely using docstrings, comments, commit messages, problem statements, and internet forum questions as queries, alongside matching code snippets, diffs, or answers as documents. These pairs have been collected from various sources, including the training splits of MTEB code tasks, the non-MTEB code retrieval dataset CoSQA+, and public datasets created for non-retrieval purposes. We also used GPT-4o [OpenAI, 2024] to synthetically generate datasets when available data is scarce.

Details of the training datasets and synthetic data generation are outlined in Appendix A.

4.2 Procedure

In each training step, we sample a batch $B = (q_1, d_1), ..., (q_n, d_n)$ of n query-document text pairs. We generate normalized embeddings for all texts in the selected pairs. We then construct a matrix of similarity values $S_{\text{dense}}(B)$ by calculating the cosine similarity of all combinations of embeddings $\mathbf{q}_i$ and $\mathbf{d}_j$ in B. We train by taking the training embedding pairs $(\mathbf{q}_i, \mathbf{d}_i)$ as similar, and all other combinations of $(\mathbf{q}_i, \mathbf{d}_j), i \neq j$ in each batch as dissimilar.

Then, we apply the contrastive InfoNCE loss function $\mathcal{L}_{\text{NCE}}$ [van den Oord et al., 2019] on the resulting matrix of similarity scores.

$$\mathcal{L}_{\text{NCE}}(S(B), \tau) := -\sum_{i,j=0}^{n} \ln \sigma(S(B), \tau, i, j) \quad \text{where } \sigma(S, \tau, i, j) := \frac{e^{S_{i,j}/\tau}}{\sum_{k=0}^{n} e^{S_{i,k}/\tau}} \tag{1}$$

where τ is the temperature and n is the batch size, which increases the weight of small differences in similarity scores in calculating the loss. During training, we maintain constant hyperparameters: $\tau = 0.05$, $n = 512$ for the 0.5B parameter model and $n = 256$ for the 1.5B parameter one, and sequence length is 512.

Each new batch B was sampled from one of the training datasets according to pre-configured sampling rates (probabilities), which we treated as hyperparameters.

Training was for 1500 steps on four 80GB VRAM A100 GPUs. Training the 0.5B parameter model took approximately 8.3 hours, and approximately 12 hours for the 1.5B parameter one. As described in Appendix B, we repeated training under three conditions to determine the best pooling method.

5 Evaluation

To assess performance on code retrieval, we evaluate the model on the MTEB-CoIR benchmark [Li et al., 2025], which consists of 10 tasks spanning text-to-code, code-to-text, code-to-code, and hybrid code retrieval types. We also evaluate the model on code-related MTEB tasks CodeSearchNetRetrieval, CodeEditSearchRetrieval, HumanEval, MBPP, DS-1000, WikiSQL, and MLQuestions, as well as CosQA+ and our in-house benchmarks. See Appendix C for evaluation hyperparameters. Results are reported in Table 2.

Table 2: Evaluation Results on Code Retrieval Tasks

Benchmark	JCE-0.5B	JCE-1.5B	JV4	Qw3-0.6B	VC3	GE-001
CoSQA+	15.42%	16.38%	13.29%	15.63%	13.57%	**16.44%**
CoSQA*	39.25%	35.10%	29.99%	37.75%	34.11%	**51.94%**
MBPP	89.01%	90.13%	89.93%	88.29%	**94.68%**	93.46%
COIR-CSN*	85.73%	86.45%	84.03%	84.78%	**89.35%**	81.06%
CSN*	90.68%	91.38%	84.84%	90.77%	**93.92%**	91.38%
Doc2Code	95.98%	96.34%	91.46%	94.77%	**97.18%**	96.54%
SWE-Bench	83.00%	86.33%	81.00%	76.12%	87.02%	**87.40%**
CES*	83.25%	**84.43%**	72.75%	64.21%	80.30%	81.69%
CP-FT	63.00%	**65.06%**	45.93%	38.50%	59.24%	61.18%
AppsR*	84.17%	86.63%	78.32%	75.22%	93.77%	**95.70%**
LeetCode	57.86%	59.075%	**59.11%**	58.23%	58.89%	58.40%
CodeChef	94.03%	96.89%	87.98%	84.29%	99.18%	**99.55%**
SynText2SQL*	72.80%	73.91%	**76.98%**	66.91%	63.39%	59.24%
Spider	81.65%	**82.18%**	81.18%	81.45%	81.99%	81.15%
WikiSQL	**98.31%**	98.02%	96.06%	96.04%	95.71%	90.94%
CF-MT*	89.56%	89.91%	70.07%	90.79%	**93.47%**	64.95%
CF-ST*	85.73%	86.18%	85.47%	86.43%	**90.56%**	85.70%
StackOQA*	91.04%	92.37%	93.80%	89.96%	**96.90%**	96.02%
DS-1000	59.77%	62.88%	64.11%	61.19%	69.49%	**70.10%**
MLQuestions	**81.05%**	77.46%	54.71%	60.52%	66.87%	62.95%
CTOC*	90.37%	92.54%	92.23%	86.28%	**93.49%**	92.59%
CTODL*	41.69%	37.319%	**46.29%**	31.78%	38.72%	32.84%
CodeChefXLang	99.70%	99.44%	92.82%	90.94%	99.13%	**99.79%**
CSN-CC*	90.41%	91.12%	83.69%	**91.41%**	90.09%	84.69%
HumanEval	96.77%	98.41%	96.74%	94.84%	**99.77%**	98.90%
Overall AVG	78.41%	79.04%	74.11%	73.49%	**79.23%**	77.38%
MTEB Code AVG	78.72%	78.94%	74.87%	74.69%	**79.84%**	76.48%

Models: JCE: `jina-code-embeddings`; JV4: `jina-embeddings-v4`; Qw3-0.6B: `Qwen3-Embedding-0.6B`; VC3: `voyage-code-3`; GE-001: `gemini-embedding-001`
Benchmarks: COIR-CSN: COIRCodeSearchNetRretrieval; CSN: CodeSearchNetRetrieval; CES: CodeEditSearchRetrieval; CP-FT: CommitPackFT; AppsR: AppsRetrieval; SynText2SQL: SyntheticText2SQL: CF-MT: CodeFeedbackMT; CF-ST: CodeFeedbackST; StackOQA: StackOverflowQA; CTOC: CodeTransOceanContest; CTODL: CodeTransOceanDL; CSN-CC: CodeSearchNetCCRetrieval
* Benchmarks of the MTEB Code leaderboard.

Both `jina-code-embeddings-0.5b` and `1.5b` outperform similar-sized general-purpose embedding model `Qwen3-Embedding-0.6B` and the substantially larger models `jina-embeddings-v4` and `gemini-embedding-001`.

6 Conclusion

We have introduced `jina-code-embeddings`, a family of code embedding models with 0.5B and 1.5B parameters. By using an autoregressive backbone pre-trained on both text and code, along with task-specific instruction prefixes and last-token pooling, the models excel at a wide variety of tasks and domains related to code retrieval. Despite their smaller size compared to other models, the `jina-code-embeddings` suite achieves state-of-the-art performance, demonstrating the validity and effectiveness of its unique construction methodology.

References

Ethan Caballero and Ilya Sutskever. Description2Code Dataset, 2016. URL https://github.com/ethancaballero/description2code.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proc. NAACL-HLT 2019, vol. 1*, 2019. doi: 10.18653/V1/N19-1423. URL https://aclanthology.org/N19-1423/.

Kenneth Enevoldsen, Isaac Chung, et al. MMTEB: Massive Multilingual Text Embedding Benchmark. *arXiv preprint arXiv:2502.13595*, 2025. URL https://arxiv.org/abs/2502.13595.

Zhangyin Feng, Daya Guo, Duyu Tang, et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP 2020*, 2020. URL https://aclanthology.org/2020.findings-emnlp.139/.

Google DeepMind. Gemini Embedding: Generalizable Embeddings from Gemini. *arXiv preprint arXiv:2503.07891*, 2025. URL https://arxiv.org/abs/2503.07891.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep Code Search. In *Proc. ICSE '18*, 2018. URL https://dl.acm.org/doi/10.1145/3180155.3180167.

Michael Günther, Jackmin Ong, Isabelle Mohr, et al. Jina Embeddings 2: 8192-Token General-Purpose Text Embeddings for Long Documents. *arXiv preprint arXiv:2310.19923*, 2024. URL https://arxiv.org/abs/2310.19923.

Michael Günther, Saba Sturua, Mohammad Kalim Akram, et al. jina-embeddings-v4: Universal Embeddings for Multimodal Multilingual Retrieval. *arXiv preprint arXiv:2506.18902*, 2025. URL https://arxiv.org/abs/2506.18902.

Edward J. Hu, Yelong Shen, Phillip Wallis, et al. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR 2022*, 2022. URL https://iclr.cc/virtual/2022/poster/6319.

Binyuan Hui, Jian Yang, Zeyu Cui, et al. Qwen2.5-Coder Technical Report. *arXiv preprint arXiv:2409.12186*, 2024. URL https://arxiv.org/abs/2409.12186.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv preprint arXiv:1909.09436*, 2019. URL http://arxiv.org/abs/1909.09436.

Aditya Kusupati, Gantavya Bhatt, Aniket Rege, et al. Matryoshka Representation Learning. In *Proc. NeurIPS '22*, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/c32319f4868da7613d78af9993100e42-Paper-Conference.pdf.

Chankyu Lee, Rajarshi Roy, Mengyao Xu, et al. NV-Embed: Improved Techniques for Training LLMs as Generalist Embedding Models. *arXiv preprint arXiv:2405.17428*, 2025. URL https://arxiv.org/abs/2405.17428.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Proc. NeurIPS '20*, 2020. URL https://dl.acm.org/doi/abs/10.5555/3495724.3496517.

Xiangyang Li, Kuicai Dong, Yi Quan Lee, et al. CoIR: A Comprehensive Benchmark for Code Information Retrieval Models. In *Proc. ACL 2025, vol. 1*, 2025. URL https://aclanthology.org/2025.acl-long.1072/.

OpenAI. GPT-4o system card. *arXiv preprint arXiv:2410.21276*, 2024. URL https://arxiv.org/abs/2410.21276.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proc. EMNLP-IJCNLP 2019*, 2019. URL https://aclanthology.org/D19-1410/.

Saba Sturua, Isabelle Mohr, Mohammad Kalim Akram, et al. jina-embeddings-v3: Multilingual Embeddings With Task LoRA. *arXiv preprint arXiv:2409.10173*, 2024. URL https://arxiv.org/abs/2409.10173.

Hongjin Su, Weijia Shi, Jungo Kasai, et al. One Embedder, Any Task: Instruction-Finetuned Text Embeddings. In *Proc. ACL 2023*, 2023. URL https://aclanthology.org/2023.findings-acl.71/.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation Learning with Contrastive Predictive Coding. *arXiv preprint arXiv:1807.03748*, 2019. URL https://arxiv.org/abs/1807.03748.

Liang Wang, Nan Yang, Xiaolong Huang, et al. Improving Text Embeddings with Large Language Models. In *Proc. ACL 2024, vol. 1*, 2024. URL https://aclanthology.org/2024.acl-long.642/.

Wojciech Zaremba and Greg Brockman. OpenAI Codex. https://openai.com/index/openai-codex/, 2021. [Online; accessed 27-August-2025].

Yanzhao Zhang, Mingxin Li, Dingkun Long, et al. Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. *arXiv preprint arXiv:2506.05176*, 2025. URL https://arxiv.org/abs/2506.05176.

A Training Datasets

Training data for `jina-code-embeddings` draws on a variety of sources, described in Section 4.1.

- Training data splits for MTEB code tasks, and the CoSQA+ dataset.
- Other public datasets adapted to our training needs.
- Fully or partially synthetic datasets generated using GPT-4o [OpenAI, 2024].

The **SyntheticDLTrans** dataset consists of generated deep learning code translations between frameworks. Since very little non-synthetic data of this kind is available, we generated all query-document pairs from scratch by repeatedly prompting GPT-4o to output a pair of equivalent deep learning code snippets in two specified distinct frameworks (e.g., PyTorch and TensorFlow). We instructed GPT-4o that the code snippets can define model architectures (Transformers, CNNs, RNNs, simple deep networks, etc.), training loops, evaluations, and mathematical operations on tensors.

We also synthesized a multilingual extension of the CodeChef dataset [Caballero and Sutskever, 2016], using the original programming solutions in C++ and Python to generate solutions in eight more programming languages (C, C#, Go, Java, JavaScript, Ruby, Rust, TypeScript). We prompted GPT-4o to produce an equivalent solution in a specified new language given one C++ solution and one Python solution as references (but we did not use problem statements for generation). The resulting dataset has been adapted for three tasks: **CodeChefP2S** (problem-to-solution), **CodeChefS2S** (monolingual solution-to-solution), and **CodeChefXLang** (crosslingual solution-to-solution).

Synthetic examples were validated by manual inspection of samples.

Table 3 provides details about the provenance of all training datasets. Table 4a shows the task categories of each training dataset, and Table 4b does the same for the evaluation datasets.

Table 3: Datasets used to train `jina-code-embeddings`

Dataset	Type	Source
AppsRetrieval	MTEB Code	https://huggingface.co/datasets/CoIR-Retrieval/apps
CodeFeedbackMT		https://huggingface.co/datasets/CoIR-Retrieval/codefeedback-mt
CodeFeedbackST		https://huggingface.co/datasets/CoIR-Retrieval/codefeedback-st
CodeTransOceanContest		https://huggingface.co/datasets/CoIR-Retrieval/codetrans-contest
CodeTransOceanDL		https://huggingface.co/datasets/CoIR-Retrieval/codetrans-dl
CodeSearchNetCCRetrieval		https://huggingface.co/datasets/CoIR-Retrieval/CodeSearchNet-ccr
COIR-CodeSearchNet		https://huggingface.co/datasets/CoIR-Retrieval/CodeSearchNet
CoSQA		https://huggingface.co/datasets/CoIR-Retrieval/cosqa
StackOverflowQA		https://huggingface.co/datasets/CoIR-Retrieval/stackoverflow-qa
SyntheticText2SQL		https://huggingface.co/datasets/CoIR-Retrieval/synthetic-text2sql
CodeForcesP2S	Adapted	https://huggingface.co/datasets/MatrixStudio/Codeforces-Python-Submissions
CodeForcesS2S		https://github.com/ethancaballero/description2code
CodeSearchNet		https://github.com/github/CodeSearchNet
CommitPackFT		https://huggingface.co/datasets/bigcode/commitpackft
CoSQA+		https://github.com/DeepSoftwareAnalytics/CoSQA_Plus
DataScience		https://kaggle.com/datasets/stackoverflow/stacksample
Doc2Code		https://github.com/EdinburghNLP/code-docstring-corpus
GlaiveCodeAssistantV2		https://huggingface.co/datasets/glaiveai/glaive-code-assistant-v2
HackerEarth		https://github.com/ethancaballero/description2code
LeetCodeP2S LeetCodeXLang		https://huggingface.co/datasets/greengerong/leetcode
MBPP		https://huggingface.co/datasets/google-research-datasets/mbpp
MLQuestions		https://huggingface.co/datasets/McGill-NLP/mlquestions
Spider		https://huggingface.co/datasets/xlangai/spider
StackExchangeBody StackExchangePost StackExchangeTitle		https://github.com/EleutherAI/stackexchange_dataset/
SWE-Bench		https://huggingface.co/datasets/princeton-nlp/SWE-bench
WikiSQL		https://huggingface.co/datasets/Salesforce/wikisql
CodeChefP2S CodeChefS2S CodeChefXLang SyntheticDLTrans	Synthetic	

Table 4: Breakdown of the training (a) and evaluation (b) datasets by task type.

(a) Training datasets

Dataset	Task type
AppsRetrieval	NL2Code
CodeChefP2S	
CodeForcesP2S	
CodeSearchNet	
CommitPackFT	
CoSQA	
CoSQA+	
Doc2Code	
LeetCodeP2S	
MBPP	
Spider	
SWE-Bench	
SyntheticText2SQL	
WikiSQL	
CodeFeedbackMT	TechQA
CodeFeedbackST	
DataScience	
GlaiveCodeAssistantV2	
MLQuestions	
StackExchangeBody	
StackExchangePost	
StackExchangeTitle	
StackOverflowQA	
CodeChefS2S	Code2Code
CodeChefXLang	
CodeForcesS2S	
CodeTransOceanContest	
CodeTransOceanDL	
HackerEarth	
LeetCodeXLang	
SyntheticDLTrans	
COIRCodeSearchNetRetrieval	Code2NL
CodeSearchNetCCRetrieval	Code2Completion

(b) Evaluation datasets

Dataset	Task type
AppsRetrieval	NL2Code
CodeChef	
CodeEditSearchRetrieval	
CodeSearchNetRetrieval	
CommitPackFT	
CoSQA	
CoSQA+	
Doc2Code	
HumanEval	
LeetCode	
MBPP	
Spider	
SWE-Bench	
SyntheticText2SQL	
WikiSQL	
CodeFeedbackMT	TechQA
CodeFeedbackST	
DS-1000	
MLQuestions	
StackOverflowQA	
CodeChefXLang	Code2Code
CodeTransOceanContest	
CodeTransOceanDL	
COIRCodeSearchNetRetrieval	Code2NL
CodeSearchNetCCRetrieval	Code2Completion

B Ablation

We trained three versions of the 0.5B model with the same training data, hyperparameters, and number of steps (1500), but with three different pooling methods: last-token, mean, and latent-attention. We found that last-token pooling results in the highest average performance (see Table 5).

Table 5: Results of the pooling ablation experiments.

Benchmark	Last-token	Mean	Latent attention
CoSQA+	15.42%	15.36%	**15.55%**
CoSQA*	**39.25%**	37.13%	38.58%
MBPP	**89.01%**	87.01%	88.57%
COIR-CSN*	**85.73%**	85.01%	85.50%
CSN*	**90.68%**	90.48%	90.65%
Doc2Code	**95.98%**	95.91%	95.94%
SWE-Bench	83.00%	**83.88%**	83.31%
CES*	**83.25%**	82.94%	83.09%
CP-FT	63.00%	62.32%	**63.10%**
AppsR*	84.17%	83.26%	**84.43%**
LeetCode	57.86%	58.08%	**58.17%**
CodeChef	94.03%	92.08%	**95.03%**
SynText2SQL*	72.80%	72.60%	**72.93%**
Spider	81.65%	**81.99%**	81.57%
WikiSQL	**98.31%**	93.50%	97.85%
CF-MT*	**89.56%**	86.09%	88.95%
CF-ST*	**85.73%**	84.55%	85.23%
StackOQA*	**91.04%**	90.46%	90.58%
DS-1000	59.77%	58.91%	**60.20%**
MLQuestions	81.05%	79.78%	**81.07%**
CTOC*	90.37%	86.85%	**90.70%**
CTODL*	**41.69%**	38.17%	40.58%
CodeChefXLang	**99.70%**	99.31%	99.21%
CSN-CC*	**90.41%**	88.65%	89.72%
HumanEval*	**96.77%**	95.78%	96.35%
Overall AVG	**78.41%**	77.20%	78.27%
MTEB Code AVG	**78.72%**	77.18%	78.41%

C Evaluation Hyperparameters

`jina-code-embeddings` and `Qwen3-Embedding-0.6B` were evaluated in FP16 with a batch size of 8 and a sequence length of 8192; `jina-embeddings-v4` was evaluated in BF16 with a batch size of 8 and a sequence length of 8192. The Voyage and Gemini models were evaluated via the respective APIs with a batch size of 8, except the tasks COIRCodeSearchNetRetrieval and CodeSearchNetCCRetrieval, which we did not evaluate due to the large size of the benchmarks and the resulting cost in time and money, so we took public scores from the MTEB GitHub.