A BIT AT A TIME: SPARSE DELTA UPDATES FOR INDEXING FAST-EVOLVING CODEBASES

Anonymous authors
Paper under double-blind review

ABSTRACT

Modern code retrieval systems increasingly operate over codebases that evolve continuously: functions are reformatted, locally edited, refactored, or rewritten while preserving their underlying intent. Standard retrieval pipelines usually treat each version as an independent target, optimizing ranking quality but ignoring whether related versions can share stable index-facing representational structure. We study this problem as **efficient continual code retrieval**, where a model should retrieve a behaviorally similar version of a code object while enabling representation and index reuse across code evolution. To evaluate this setting, we construct CODECONTINUUM-R, an evolution-aware benchmark built by expanding standard code-search examples into version families that span surface-level rewrites, local structural edits, and higher-level refactorings. Surprisingly, we find that sparse representations already provide a strong foundation for this setting: their active latent supports form discrete routing signatures that remain substantially shared across evolved code variants while maintaining competitive retrieval quality. Building on this observation, we propose **continuality-preserved sparse coding**, which augments sparse autoencoder training with two lightweight evolution-aware regularizers: a neuron-overlap regularizer that encourages behaviorally similar variants to activate shared Top-K latents, and an activation-order regularizer that preserves the relative priority of shared active neurons. Across Jina-Code-Embedding and Qwen3-Embedding backbones, our method improves sparse-support overlap and activation-order consistency while largely preserving retrieval accuracy. These improvements enable more compact neuron-reuse storage and more reliable small-budget routing in multi-stage sparse retrieval for evolving codebases.

1 INTRODUCTION

Modern representation learning is increasingly deployed in settings where targets are not static. A document may be revised, a product description may be updated, and a program may be reformatted, locally edited, refactored, or rewritten by developers and AI coding assistants. In many cases, the observable form of an object changes while its underlying intent remains largely stable. Code retrieval is a particularly important example: repositories evolve through commits, automated formatting, dependency updates, and refactorings, yet developers still expect search systems to retrieve the relevant functionality regardless of the specific version currently stored in the corpus. Existing code retrieval benchmarks (Husain et al., 2019; Hasan et al., 2021; Yao et al., 2018; Lu et al., 2021; Huang et al., 2021; Li et al., 2025) have substantially advanced evaluation along dimensions such as robustness, multilinguality, repository context, and cross-modal retrieval, but they mostly assume a static target corpus. As a result, they do not directly evaluate whether intent-preserving versions of the same code object can share stable, memory-efficient representation and indexing structure. This motivates the problem of **efficient continual code retrieval**: retrieval systems should not only rank behaviorally similar code variants above unrelated snippets, but also expose index-facing states that can be reused as code evolves.

To study this problem, we introduce **CODECONTINUUM-R** (Code Continuum for Retrieval), an evolution-aware code retrieval benchmark derived from standard text-to-code retrieval data. Specifically, we expand each original code snippet into a version family with evolved variants produced at three levels of code change: surface-preserving rewrites that alter concrete syntax, local structural

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

edits that perturb the implementation details slightly, and higher-level refactorings that reorganize the code structure. This construction turns standard code retrieval into an evolution-aware setting: a model must retrieve not only the original code, but also its behaviorally related evolved variants. More importantly, it enables us to study whether retrieval representations remain structurally stable and reusable as code evolves, rather than requiring each new snapshot to be represented and indexed from scratch.

Sparse representations provide a natural starting point for studying such reuse because they expose explicit index-facing structure. Unlike dense embeddings, which do not explicitly expose reusable representation-level sub-components, sparse representations decompose each code embedding into a small set of active latent features whose indices form discrete routing signatures over inverted lists, candidate pools, or other index units. Empirically, we find that even without evolution-aware supervision, vanilla sparse coding preserves substantial Top-K support overlap across evolved variants while maintaining strong retrieval quality, revealing considerable reusable structure across code evolution. However, this reuse remains incomplete: support drift can cause related variants to touch different sparse index pools, while priority drift can reorder shared active features and destabilize small-budget routing that relies on only the strongest activations.

Building on these observations, we propose **continuality-preserved sparse coding**, a lightweight extension of sparse autoencoder training for evolving code. Our method introduces two evolution-aware regularizers over version families: one encourages original snippets and their evolved variants to retain overlapping Top-K latent supports, promoting reuse of sparse index structure, while the other preserves the relative order of shared active neurons, stabilizing feature priorities for prefix-based and multi-stage retrieval. Across Jina-Code-Embeddings and Qwen3-Embedding backbones, these regularizers substantially improve support overlap and activation-order consistency while largely preserving retrieval quality, leading to lower index memory through neuron reuse and more reliable small-budget routing in multi-stage sparse retrieval.

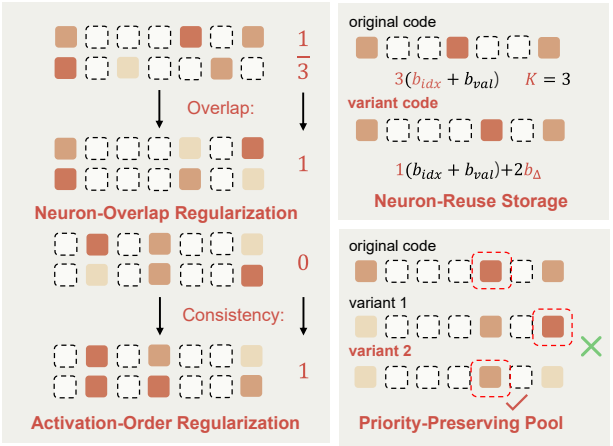


Figure 1: Overview of our method. We propose two regularization terms **(Left)**, illustrate neuron-reuse storage **(Upper Right)** and explore priority-preserving pools for multi-stage retrieval **(Lower Right)**.

We discuss in detail the evolution of code representation learning, code retrieval benchmarks and sparse coding in Appendix A, and the potential positive and negative societal impacts in Appendix F. Our contributions are summarized as follows:

- We identify **continuous representation learning** as an underexplored setting where representation targets evolve through continuous, relatively small changes, and instantiate it in evolution-aware code retrieval.
- We introduce an evolution-aware benchmark built from CodeSearchNet-Python (Husain et al., 2019) with three levels of controlled code evolution: LibCST codemods, Cosmic-Ray-assisted edits, and language-model rewrites. Utilizing this benchmark, we show that sparse representations are strong and efficient baselines for evolving code, exposing stable sparse supports that enable index sharing and consistent coarse retrieval.
- We propose **continuality-preserved sparse coding** with neuron-overlap and activation-order regularization, improving sparse-support stability, compact-index sharing, and multi-stage routing consistency while largely preserving retrieval accuracy.

2 CONTINUAL CODE RETRIEVAL: SETUP AND BENCHMARK

2.1 PROBLEM FORMULATION

We study retrieval over evolving codebases, where multiple revisions of the same code object may coexist. For each code object i, let

$$V_i = \{d_i^{(0)}, d_i^{(1)}, \ldots, d_i^{(L_i)}\} \tag{1}$$

denote its version family, where $d_i^{(0)}$ is the original snippet and $d_i^{(\ell)}$ denotes an evolved revision. Each family V_i is paired with a natural-language query q_i, and the retrieval corpus is $\mathcal{C} = \bigcup_{i=1}^{n} V_i$.

A retriever scores query–code pairs with $s_\theta(q, d)$ and aims to rank versions associated with q_i above unrelated snippets:

$$s_\theta(q_i, d_i^{(\ell)}) > s_\theta(q_i, d_j^{(m)}), \qquad j \neq i. \tag{2}$$

Beyond ranking quality, continual retrieval should also allow related revisions to reuse representation and indexing structure, rather than requiring every snapshot to be stored and routed independently. The goal is therefore to preserve retrieval accuracy while retaining reusable structure across versions without removing revision-specific information.

2.2 CODECONTINUUM-R: CONTROLLED CODE EVOLUTION

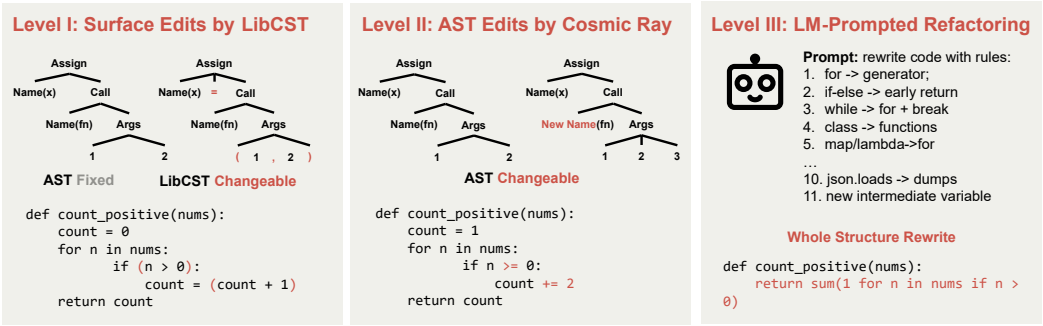


Figure 2: **Overview of CODECONTINUUM-R.** Version families span surface edits, local AST mutations, and LM-generated refactorings.

We construct CODECONTINUUM-R from CodeSearchNet-Python (Husain et al., 2019) to study representation and index reuse under controlled code evolution. Shown in Figure 2, each query-code pair $(q_i, d_i^{(0)})$ is expanded into a version family through three transformation levels.

Level I: Surface edits. We use LibCST to modify concrete syntax while preserving the abstract syntax tree (AST). LibCST retains whitespace and comments in a concrete syntax tree, enabling precise source-to-source rewriting. Edits change whitespace, redundant parentheses, delimiter formatting, or non-semantic comments; for example, `x*x` becomes `(x * x)`. We retain a variant only when its parsed AST matches the original. This level isolates sensitivity to surface form, without introducing structural program changes.

Level II: Local mutations. We apply mutation operators from Cosmic Ray to modify, delete, or replace local AST nodes. Unlike surface edits, these transformations can introduce localized behavioral changes. For example, replacing > with >= changes a comparison's boundary condition. This level therefore evaluates retrieval robustness under small structural and semantic perturbations, without assuming exact equivalence during evolution.

Level III: LM refactorings. We prompt Qwen2.5-Coder-32B-Instruct (Hui et al., 2024) to reorganize implementations while preserving their intended functionality. The prompt requires preservation of public interfaces, evaluation order, and side effects. Rewrites simplify control flow, introduce helper variables, replace suitable loops with library calls, or extract repeated logic. Unlike the local operators above, these transformations can affect several parts of an implementation at once.

All candidates undergo parsing, interface, and lightweight static checks to remove invalid or unreliable code. For each transformation level, retained variants are grouped with their original snippet and unchanged query. This level-specific organization separates surface variation from stronger structural changes. We additionally sample 10000 original snippets and their variants for test-based validation, generating five executable tests per original and applying the same suite to its variants. The generated tests provide an additional check on behavioral consistency across versions. Construction, filtering, and validation details are given in Appendix B.

2.3 Evaluation Metrics

We evaluate retrieval quality using ranking metrics such as nDCG, MRR, or Recall@K, treating all versions in V_i as relevant to q_i. Beyond accuracy, we consider two properties: continuity of the representation signal and reuse of the index resources it accesses. Both criteria are independent of the representation type and indexing backend.

Representation continuity. A useful representation for evolving code should remain stable across related revisions rather than change arbitrarily with each edit. Let $r_\theta(d)$ denote the representation of code snippet d. We measure continuity by comparing each evolved variant with its original version:

$$\mathrm{Cont}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{L_i}\sum_{\ell=1}^{L_i} s\Big(r_\theta(d_i^{(0)}), r_\theta(d_i^{(\ell)})\Big). \tag{3}$$

where the similarity function $s(\cdot,\cdot)$ is chosen according to the representation type, with higher continuity indicating less drift under code evolution. For sparse representations in CodeContinuum-R, we instantiate this notion using support overlap and activation-order consistency.

Index reuse. Related code revisions are easier to maintain when they can share the same indexing structure rather than being indexed independently. Let $I_\theta(d)$ denote the set of index units used by snippet d, such as inverted lists, routing buckets, or storage blocks. We measure reuse by the average overlap between the index units used by an original snippet and its evolved variants:

$$\mathrm{Reuse}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{L_i}\sum_{\ell=1}^{L_i}\frac{\Big|I_\theta(d_i^{(0)}) \cap I_\theta(d_i^{(\ell)})\Big|}{\Big|I_\theta(d_i^{(0)}) \cup I_\theta(d_i^{(\ell)})\Big|}. \tag{4}$$

Higher overlap indicates greater potential to share storage and routing structure across revisions. In our sparse setting, these index units correspond naturally to active latent features used for indexing and routing.

3 Sparse Representations under Code Evolution

3.1 Motivation: From Dense Embeddings to Reusable Sparse Structure

Dense retrievers provide strong semantic representations of code, but expose little structure that can be reused across revisions. Each snapshot d is represented as a complete embedding $\mathbf{h}(d) = f_\theta(d) \in \mathbb{R}^p$ and is typically indexed together with revision-specific ANN metadata. Thus, even when two revisions are semantically close, their storage and index-update costs remain tied to complete vectors rather than to the amount of change between them.

Sparse representations expose finer-grained index-facing structure. Given a frozen dense embedding $\mathbf{h}(d)$, we train a sparse autoencoder

$$\mathbf{z}(d) = \mathrm{TopK}\left(\sigma_+(\mathbf{W}_{\mathrm{enc}}(\mathbf{h}(d) - \mathbf{b}_{\mathrm{pre}}) + \mathbf{b}_{\mathrm{enc}})\right), \quad \hat{\mathbf{h}}(d) = \mathbf{W}_{\mathrm{dec}}\mathbf{z}(d) + \mathbf{b}_{\mathrm{pre}}. \tag{5}$$

We denote the active support by

$$S_K(d) = \mathrm{supp}(\mathbf{z}(d)). \tag{6}$$

Unlike a dense vector, $S_K(d)$ provides a discrete routing signature whose active dimensions can serve directly as inverted-list keys or routing buckets. Consequently, two revisions can reuse index-facing structure whenever their supports overlap.

The base sparse-coding objective is

$$\mathcal{L}_{\text{sc}} = \sum_{K' \in \mathcal{K}} \alpha_{K'} \|\mathbf{h}(d) - \hat{\mathbf{h}}_{K'}(d)\|_2^2 + \beta \mathcal{L}_{\text{aux}} + \gamma \mathcal{L}_{\text{ucl}}, \tag{7}$$

which combines Multi-TopK reconstruction loss on a series of sparsity levels $\mathcal{K}$, auxiliary latent-utilization loss $\mathcal{L}_{\text{aux}}$ and contrastive loss $\mathcal{L}_{\text{ucl}}$.

3.2 Vanilla Sparse Coding Is Already Revision-Consistent

We first ask whether reusable sparse structure emerges without any evolution-aware training. Table 1 shows that vanilla sparse coding already preserves high Top-K support overlap while maintaining strong retrieval quality. For example, at $K = 32$, Jina-Code-1.5B and Qwen3-4B achieve 89.78% and 88.96% neuron overlap, respectively. Controlled analyses further show that this phenomenon cannot be explained solely by lexical or AST similarity (Appendix D.2), and persists on naturally occurring Git revisions and independently collected cross-framework semantic pairs (Appendix D.3, Appendix D.4).

Backbone	K	Neuron Overlap	Activation Order Consistency	nDCG@10
Jina-Code-1.5B	32	89.78%	75.06%	90.07
	16	91.30%	77.36%	89.62
Qwen3-4B	32	88.96%	76.30%	92.12
	16	92.79%	76.80%	90.04

Table 1: **Unregularized sparse coding on CodeContinuum-R.** Sparse representations already provide high support overlap across evolved code variants while maintaining strong retrieval quality.

Despite this strong baseline, two failure modes remain. First, support drift causes evolved revisions to activate different sparse dimensions, reducing reuse of inverted lists or candidate pools. Second, priority drift changes the relative order of otherwise shared neurons: activation-order consistency remains only 75–77%, which can destabilize small-prefix routing in multi-stage retrieval. These two failure modes motivate the regularizers introduced next.

4 Continuality-Preserved Sparse Coding

Vanilla sparse coding already exposes substantial reusable structure across code revisions, but still suffers from support drift and priority drift. We therefore augment the base sparse-coding objective with two evolution-aware regularizers that target these two failure modes.

4.1 Neuron-Overlap Regularization

Sparse retrieval systems use active latent neurons as index-facing signals. Related code revisions often retain substantial retrieval-relevant structure even when localized edits introduce small behavioral changes. We therefore encourage, rather than enforce, stability in their high-priority sparse supports, so that reusable structure can continue to map to overlapping inverted lists or candidate pools. Importantly, this objective does not assume full semantic equivalence between all variants or require their representations to be identical; it only biases behaviorally similar revisions toward reusing part of their Top-K support, while the base reconstruction and retrieval objectives preserve revision-specific information. The neuron-overlap regularizer is designed to reduce unnecessary support drift under such code evolution.

For a code snapshot d, let $S_K(d)$ be the indices of the K largest entries in $\mathbf{z}(d)$. Consider an original code snapshot $d_i^{(0)}$ and one of its evolved variants $d_i^{(\ell)}$. We compare their latent memberships only on the union of their selected supports, $U_i^{(\ell)} = S_K(d_i^{(0)}) \cup S_K(d_i^{(\ell)})$, since neurons outside this union are irrelevant to Top-K retrieval. To make the hard Top-K membership differentiable, we use a soft mask:

$$\mathbf{m}_r(d) = \sigma\left(\frac{\mathbf{z}_r(d) - \mathbf{z}_K(d)}{\tau_m}\right), \quad r \in S_K(d_i^{(0)}) \cup S_K(d_i^{(\ell)}), \tag{8}$$

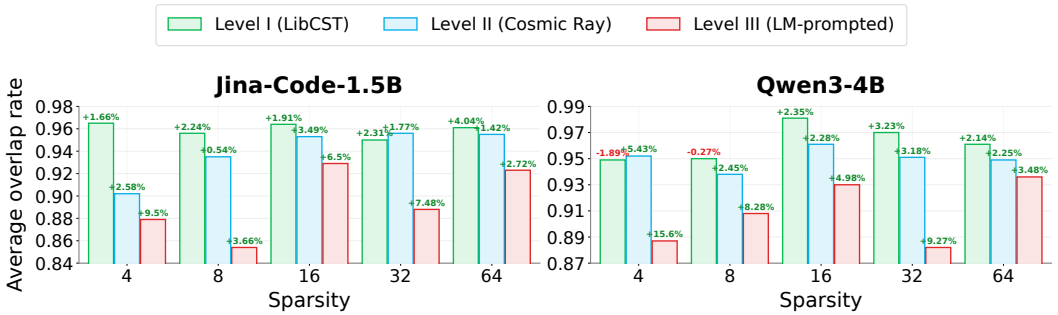


Figure 3: **Average overlap rate with neuron-overlap regularization across different sparsity and evolution levels.** Overlap improvement relative to unsupervised sparse coding is marked green, while degradation is marked in red.

where $\mathbf{z}_K(d)$ keeps the K-th largest value in $\mathbf{z}(d)$, τ_m is a temperature and the r-th element in $\mathbf{m}_r(d)$ approximates whether neuron r belongs to the Top-K support of d.

The neuron-overlap regularizer then penalizes differences between the soft membership masks of the original snapshot and its evolutionary variants:

$$\mathcal{L}_{\text{ov}} = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{1}{L_i} \sum_{\ell=1}^{L_i} \frac{1}{|U_i^{(\ell)}|} \sum_{r \in U_i^{(\ell)}} \left(\mathbf{m}_r(d_i^{(0)}) - \mathbf{m}_r(d_i^{(\ell)}) \right)^2. \tag{9}$$

This loss encourages evolved variants to keep activating the same high-priority latent neurons as the original code. Importantly, it does not force all latent values to be identical; it only encourages stable Top-K membership, which is the part of the sparse representation most directly used by index construction and routing. Figure 3 shows that it improves Top-K overlap across different settings, especially under stronger evolution where unsupervised sparse coding exhibits larger neuron drift.

4.2 Activation-Order Consistency Regularization

High support overlap alone is not sufficient for efficient multi-stage retrieval. In practice, sparse retrievers often consume active neurons progressively. A small prefix of the strongest neurons may be used for coarse routing, while a larger Top-K set is used for final retrieval or reranking. In this case, two evolved snapshots can share nearly the same Top-K support but still behave differently if the strongest neurons are permuted. The activation-order regularizer addresses this issue by preserving the relative priority of shared active neurons.

For the original snapshot $d_i^{(0)}$ and a variant $d_i^{(\ell)}$, let $C_i^{(\ell)} = S_K(d_i^{(0)}) \cap S_K(d_i^{(\ell)})$ be their common active neurons. We only compare pairs of neurons inside this common set. For a pair of neurons (r, s), the original snapshot provides the reference order through the sign of $z_r(d_i^{(0)}) - z_s(d_i^{(0)})$. If the same pair is reversed in the evolved snapshot, the following pairwise loss becomes large:

$$\ell_{r,s}^{(\ell)} = \log\left(1 + \exp\left(-\frac{\operatorname{sgn}\left(\mathbf{z}_r(d_i^{(0)}) - \mathbf{z}_s(d_i^{(0)})\right)\left(\mathbf{z}_r(d_i^{(\ell)}) - \mathbf{z}_s(d_i^{(\ell)})\right)}{\tau_o}\right)\right). \tag{10}$$

where τ_o controls the sharpness of the order penalty. To avoid unstable supervision from nearly tied activations, we exclude pairs whose activation gap in the original snapshot is smaller than a threshold ϵ. Let $\Omega_i^{(\ell)}$ denote the remaining comparable pairs. The activation-order regularizer averages the pairwise loss over all related variants:

$$\mathcal{L}_{\text{ord}} = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{1}{L_i} \sum_{\ell=1}^{L_i} \frac{1}{|\Omega_i^{(\ell)}|} \sum_{(r,s) \in \Omega_i^{(\ell)}} \ell_{r,s}^{(\ell)}. \tag{11}$$

This regularizer is complementary to neuron-overlap regularization. The overlap term encourages the same neurons to be selected, while the order term encourages the shared neurons to appear in a

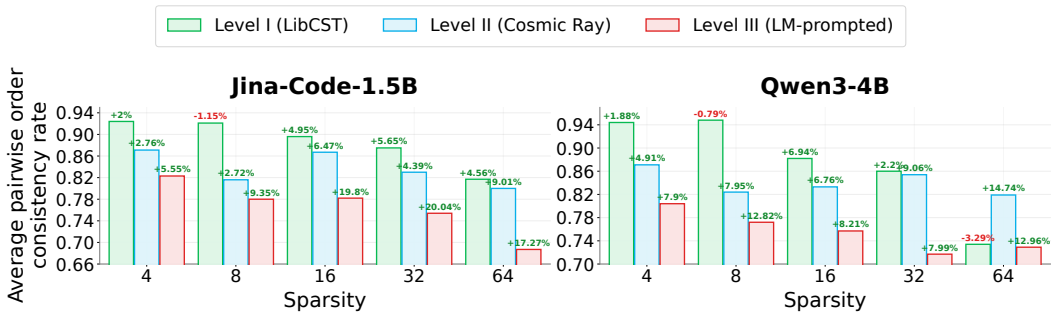


Figure 4: **Average pairwise order consistency rate with activation-order consistency regularization across different sparsity and evolution levels.** Improvement on activation-order consistency is marked green, while degradation is marked red.

Backbone	K	R1	R2	Neuron Overlap	Activation Order Consistency	Performance (nDCG@10)
Jina-1.5B	32	✗	✗	89.78%	75.06%	<u>90.07</u>
		✓	✗	<u>93.11%</u>	77.83%	**91.3**
		✗	✓	90.40%	<u>81.97%</u>	89.1
		✓	✓	**95.83%**	**83.76%**	89.85
	16	✗	✗	91.30%	77.36%	89.62
		✓	✗	<u>94.86%</u>	73.20%	**90.57**
		✗	✓	90.09%	**84.83%**	88.93
		✓	✓	**98.35%**	<u>82.35%</u>	<u>90.34</u>
Qwen3-4B	32	✗	✗	88.96%	76.30%	**92.12**
		✓	✗	<u>93.44%</u>	74.82%	90.97
		✗	✓	92.87%	<u>81.05%</u>	<u>91.46</u>
		✓	✓	**96.03%**	**85.37%**	90.48
	16	✗	✗	92.79%	76.80%	**90.04**
		✓	✗	<u>95.72%</u>	78.41%	87.65
		✗	✓	91.18%	<u>82.37%</u>	<u>88.4</u>
		✓	✓	**97.05%**	**86.29%**	87.31

Table 2: **Performance on Evolution-Aware Data.** We evaluate multi-dimensional performance across different backbones, sparsity levels and regularization settings. The best/second-best values in each metric are **bold** / <u>underlined</u>.

similar priority order. As a result, evolved variants are more likely to enter the same coarse candidate pools under small routing budgets. Figure 4 verifies that activation-order regularization improves the consistency of sparse priorities across code revisions. Compared with sparse coding trained without $\mathcal{L}_{\text{ord}}$, our full objective reduces pairwise order inversions and improves top-prefix agreement across different sparsity budgets and evolution levels.

4.3 Overall Training Objective

The final objective combines the unsupervised sparse-coding loss with the two evolution-aware regularizers:

$$\mathcal{L} = \mathcal{L}_{\text{sc}} + \lambda_{\text{ov}}\mathcal{L}_{\text{ov}} + \lambda_{\text{ord}}\mathcal{L}_{\text{ord}}. \tag{12}$$

where $\mathcal{L}_{\text{sc}}$ preserves the reconstruction and retrieval utility of the sparse code, $\mathcal{L}_{\text{ov}}$ improves support reuse across evolutionary code versions, and $\mathcal{L}_{\text{ord}}$ stabilizes the priority order of shared active neurons for multi-stage sparse retrieval.

Backbone	Sparse	Neuron-Reuse Storage	R1	R2	Sparsity 4	8	16	32
Jina-1.5B	✗	✗	✗	✗	124.9			
	✓	✗	✗	✗	0.4216	0.8431	1.6862	3.3724
	✓	✓	✗	✗	0.2825	0.5397	1.0329	1.757
	✓	✓	✓	✗	<u>0.2643</u>	<u>0.5164</u>	<u>0.9816</u>	<u>1.6861</u>
	✓	✓	✓	✓	**0.2379**	**0.492**	**0.9677**	**1.645**
Qwen3-4B	✗	✗	✗	✗	207.3			
	✓	✗	✗	✗	0.4216	0.8431	1.6862	3.3724
	✓	✓	✗	✗	0.2691	0.5137	0.8345	1.7841
	✓	✓	✓	✗	<u>0.2613</u>	<u>0.5056</u>	<u>0.8197</u>	<u>1.7559</u>
	✓	✓	✓	✓	**0.2485**	**0.4992**	**0.7863**	**1.7382**

Table 3: **Index Memory Analysis.** We evaluate the index memory cost of five settings under two backbones. Evaluation is conducted on the evolution-aware data in Section 2.2, measured in GB. The smallest and second-smallest index sizes for each backbone and sparsity setting are **bold** / <u>underlined</u>.

5 Experiments

5.1 Benchmark Performance

Performance on Evolution-Aware Data. We evaluate the model's performance on the evolution-aware dataset proposed in Section 2.2 from three perspectives: neuron overlap, activation order consistency and retrieval quality. Table 2 shows that across both backbones and sparsity budgets, the neuron-overlap regularizer **R1** substantially increases neuron sharing, while the activation-order regularizer **R2** generally improves pairwise order consistency. The combination of these two regularizers yields the strongest sparse-indexing behavior: best overlap rate in all backbone-sparsity settings and the best or second-best pairwise order rate in most cases without requiring a large sacrifice in retrieval quality. Appendix C shows more detailed setups.

Evaluation on Retrieval Quality Preservation. To evaluate whether the regularizers harm retrieval quality, we train models on the evolution-aware dataset in Section 2.2 and evaluate on nine tasks from the COIR (Li et al., 2025) and MTEB (Muennighoff et al., 2023) benchmarks under two sparsity settings. We find that adding the neuron-overlap regularizer (**R1**), the activation-order regularizer (**R2**), or both only introduces a modest performance degradation compared with unregularized sparse coding, with retrieval quality in some datasets remaining comparable or even improving under regularization. These results indicate that the proposed regularizers primarily reshape the sparse representations to improve evolution consistency and index sharing, while largely preserving the semantic retrieval performance. Detailed results are available in Appendix C.3.

5.2 Empirical Analysis

Index Memory Analysis. Building on the dense–sparse comparison in Section 3.1, we further evaluate whether evolution-aware regularization improves neuron-reuse storage. Table 3 shows that the neuron-overlap and activation-order regularizers further reduce memory usage beyond vanilla sparse coding with neuron-reuse storage. We additionally compare against conventional vector-compression baselines, including Product Quantization (PQ) and Semantic Hashing, under approximately matched index-memory budgets. Our approach achieves higher retrieval quality at both evaluated budgets, suggesting that evolution-aware sparse reuse provides benefits beyond compressing each representation independently; detailed settings and results are reported in Appendix C.6.

Multi-Stage Retrieval Analysis. Table 4 evaluates whether activation-order consistency improves priority-preserving multi-stage retrieval, where a small prefix of active neurons is used for first-stage candidate generation before second-stage reranking. **R2** consistently improves retrieval under small routing prefixes, with gains diminishing as N approaches the final budget M. Combining **R1** and **R2** further improves most small- and medium-prefix settings. We further verify that evolution-aware

Backbone	R2	R1	M/N						
			16			32			
			4	8	16	4	8	16	32
Jina-1.5B	✗	✗	73.48	81.7	89.62	75.93	83.46	87.13	**90.07**
	✓	✗	83.51	86.52	89.93	79.04	84.35	86.79	89.1
	✓	✓	**87.16**	**89.68**	**90.34**	**82.2**	**85.87**	**87.27**	89.85
Qwen3-4B	✗	✗	76.25	82.93	**90.04**	79.46	84.29	88.91	**92.12**
	✓	✗	82.67	85.1	88.4	83.65	87.38	**89.85**	91.46
	✓	✓	**85.19**	**86.72**	87.31	**85.02**	**87.95**	89.3	90.48

Table 4: **Multi-Stage Retrieval Analysis.** We evaluate the effects of neuron-overlap (**R1**) and activation-order consistency (**R2**) regularizers on multi-stage retrieval on the LM-prompted subset of evolution-aware dataset. We consider seven top-M/N settings and nDCG@10 is the evaluation metric. The best/second-best results are **bold** / underlined.

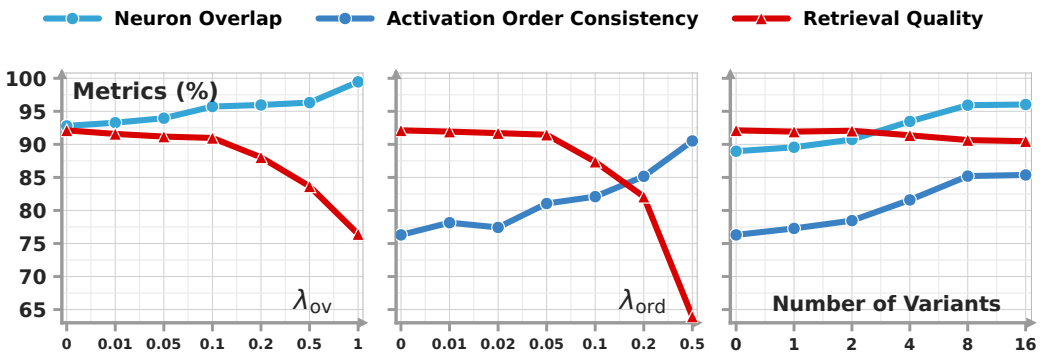


Figure 5: **Sensitivity analysis on regularization weights λ_{ov}, λ_{ord}, and number of evolutionary variants.** We focus on three metrics: Neuron Overlap, Activation Order Consistency and retrieval performance (nDCG@10).

index reuse composes with the sparse ANN engine Seismic (Bruch et al., 2025), with system-level results reported in Appendix E.

Sensitivity Analysis. We evaluate the sensitivity to three factors: neuron-overlap regularization's weight λ_{ov}, activation-order consistency regularization's weight λ_{ord}, and the number of evolved variants. As shown in Figure 5, with other parameters set to their default values, we find that $\lambda_{ov} = 0.1$ and $\lambda_{ord} = 0.05$ achieve a balance between retrieval quality and neuron overlap/activation order consistency, while eight evolved variants give the best trade-off. We further examine robustness to sparser evolution frequencies in Appendix C.5, where the index-sharing benefit remains substantial even when evolved revisions are much less frequent.

6 Conclusion

We introduce continuous representation learning as a setting where representations must remain useful and structurally reusable as objects undergo continuous evolutionary changes. Using evolution-aware code retrieval as a concrete testbed, we built a benchmark with three levels of code evolution and showed that sparse representations naturally provide stable active supports across evolved variants. Building on this observation, we proposed continuity-preserved sparse coding, which improves both Top-K neuron overlap and activation-order consistency through two lightweight regularizers. Experiments across multiple backbones demonstrate that our method strengthens sparse-index reuse, reduces memory cost, and improves small-budget multi-stage retrieval while largely preserving retrieval quality. These results suggest that representation learning for evolving corpora should consider not only semantic accuracy, but also the stability and reusability of the underlying representation structures.

AI USE STATEMENT

In this work, we used generative AI tools to assist with method implementation, synthetic dataset generation, dataset cleaning and reformatting, and translation. We did not use generative AI tools to write proofs, develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, propose or refine hypotheses, design or provide feedback on research methodology or experiments, support qualitative or thematic data analysis, or interpret results. Additionally, we used generative AI tools to modify scientific figures and summarize existing literature. We reviewed all AI-assisted work. For example, all AI-assisted figure modifications were independently reviewed and verified by the authors, and language edits were reviewed to ensure that they preserved the intended technical meaning and claims. We take responsibility for the final content of this work, including any text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work adheres to the ICLR Code of Ethics. Although our proposed methods are broadly applicable, their deployment in real-world scenarios may carry societal considerations, particularly regarding bias, fairness, and privacy. We advocate for responsible application of our techniques and disclose that we have no conflicts of interest.

REPRODUCIBILITY STATEMENT

We provide comprehensive details of our methodology, datasets, model architectures, and evaluation protocols in both the main text and Appendix. Full mathematical derivations and additional experimental results are included in the Appendix. Our code and scripts will be released upon acceptance.

REFERENCES

Michal Aharon, Michael Elad, and Alfred Bruckstein. K-svd: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Transactions on signal processing*, 54(11): 4311–4322, 2006.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Sebastian Bruch, Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. Efficient sketching and nearest neighbor search algorithms for sparse vector sets. *arXiv preprint arXiv:2509.24815*, 2025.

Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. Bge m3-embedding: Multi-lingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. *arXiv preprint arXiv:2402.03216*, 4(5), 2024.

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

Hoagy Cunningham, Aidan Ewart, Logan Riggs, Robert Huben, and Lee Sharkey. Sparse autoencoders find highly interpretable features in language models. *arXiv preprint arXiv:2309.08600*, 2023.

Zhuyun Dai and Jamie Callan. Context-aware term weighting for first stage passage retrieval. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pp. 1533–1536, 2020.

Andor Diera, Abdelhalim Dahou, Lukas Galke Poech, Fabian Karl, Florian Sihler, and Ansgar Scherp. Gencodesearchnet: A benchmark test suite for evaluating generalization in programming language understanding. In *Proceedings of the 1st GenBench Workshop on (Benchmarking) Generalisation in NLP*, pp. 12–24, 2023.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://arxiv.org/pdf/2310.11248.pdf.

David L Donoho. Compressed sensing. *IEEE Transactions on information theory*, 52(4):1289–1306, 2006.

Michael Elad and Michal Aharon. Image denoising via sparse and redundant representations over learned dictionaries. *IEEE Transactions on Image processing*, 15(12):3736–3745, 2006.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*, pp. 1536–1547, 2020.

Thibault Formal, Carlos Lassance, Benjamin Piwowarski, and Stéphane Clinchant. Splade v2: Sparse lexical and expansion model for information retrieval. *arXiv preprint arXiv:2109.10086*, 2021a.

Thibault Formal, Benjamin Piwowarski, and Stéphane Clinchant. Splade: Sparse lexical and expansion model for first stage ranking. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2288–2292, 2021b.

Leo Gao, Tom Dupré la Tour, Henk Tillman, Gabriel Goh, Rajan Troll, Alec Radford, Ilya Sutskever, Jan Leike, and Jeffrey Wu. Scaling and evaluating sparse autoencoders. *arXiv preprint arXiv:2406.04093*, 2024.

Luyu Gao, Zhuyun Dai, and Jamie Callan. Coil: Revisit exact lexical match in information retrieval with contextualized inverted list. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 3030–3042, 2021.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7212–7225, 2022.

Lixuan Guo, Yifei Wang, Tiansheng Wen, Yifan Wang, Aosong Feng, Bo Chen, Stefanie Jegelka, and Chenyu You. Csrv2: Unlocking ultra-sparse embeddings. *arXiv preprint arXiv:2602.05735*, 2026.

Masum Hasan, Tanveer Muttaqueen, Abdullah Al Ishtiaq, Kazi Sajeed Mehrab, Md Mahim Anjum Haque, Tahmid Hasan, Wasi Ahmad, Anindya Iqbal, and Rifat Shahriyar. Codesc: A large code–description parallel dataset. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 210–218, 2021.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. Cosqa: 20,000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 5690–5700, 2021.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

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.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004*, 2023.

Daria Kryvosheieva, Saba Sturua, Michael Günther, Scott Martens, and Han Xiao. Efficient code embeddings from code generation models. *arXiv preprint arXiv:2508.21290*, 2025.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pp. 18319–18345. PMLR, 2023.

Carlos Lassance, Hervé Déjean, Thibault Formal, and Stéphane Clinchant. Splade-v3: New baselines for splade. *arXiv preprint arXiv:2403.06789*, 2024.

Chankyu Lee, Rajarshi Roy, Mengyao Xu, Jonathan Raiman, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. Nv-embed: Improved techniques for training llms as generalist embedding models. *arXiv preprint arXiv:2405.17428*, 2024.

Honglak Lee, Alexis Battle, Rajat Raina, and Andrew Ng. Efficient sparse coding algorithms. *Advances in neural information processing systems*, 19, 2006.

Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Hao Zhang, Xinyi Dai, Yasheng Wang, and Ruiming Tang. Coir: A comprehensive benchmark for code information retrieval models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 22074–22091, 2025.

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

Tom Lieberum, Senthooran Rajamanoharan, Arthur Conmy, Lewis Smith, Nicolas Sonnerat, Vikrant Varma, János Kramár, Anca Dragan, Rohin Shah, and Neel Nanda. Gemma scope: Open sparse autoencoders everywhere all at once on gemma 2. In *Proceedings of the 7th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pp. 278–300, 2024.

Jimmy Lin and Xueguang Ma. A few brief notes on deepimpact, coil, and a conceptual framework for information retrieval techniques. *arXiv preprint arXiv:2106.14807*, 2021.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*, 2023.

Ye Liu, Rui Meng, Shafiq Joty, Silvio Savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. Codexembed: A generalist embedding model family for multiligual and multi-task code retrieval. *arXiv preprint arXiv:2411.12644*, 2024.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Alireza Makhzani and Brendan Frey. K-sparse autoencoders. *arXiv preprint arXiv:1312.5663*, 2013.

Antonio Mallia, Omar Khattab, Torsten Suel, and Nicola Tonellotto. Learning passage impacts for inverted indexes. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 1723–1727, 2021.

Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. Mteb: Massive text embedding benchmark. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 2014–2037, 2023.

Bruno A Olshausen and David J Field. Sparse coding with an overcomplete basis set: A strategy employed by v1? *Vision research*, 37(23):3311–3325, 1997.

Senthooran Rajamanoharan, Arthur Conmy, Lewis Smith, Tom Lieberum, Vikrant Varma, János Kramár, Rohin Shah, and Neel Nanda. Improving dictionary learning with gated sparse autoencoders. *arXiv preprint arXiv:2404.16014*, 2024a.

Senthooran Rajamanoharan, Tom Lieberum, Nicolas Sonnerat, Arthur Conmy, Vikrant Varma, János Kramár, and Neel Nanda. Jumping ahead: Improving reconstruction fidelity with jumprelu sparse autoencoders. *arXiv preprint arXiv:2407.14435*, 2024b.

Ruslan Salakhutdinov and Geoffrey Hinton. Semantic hashing. *International Journal of Approximate Reasoning*, 50(7):969–978, 2009.

Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. Cocosoda: Effective contrastive learning for code search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 2198–2210. IEEE, 2023.

Yewei Song, Cedric Lothritz, Xunzhu Tang, Tegawendé Bissyandé, and Jacques Klein. Revisiting code similarity evaluation with abstract syntax tree edit distance. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 38–46, 2024.

Saba Sturua, Isabelle Mohr, Mohammad Kalim Akram, Michael Günther, Bo Wang, Markus Krimmel, Feng Wang, Georgios Mastrapas, Andreas Koukounas, Nan Wang, et al. jina-embeddings-v3: Multilingual embeddings with task lora. *arXiv preprint arXiv:2409.10173*, 2024.

Nandan Thakur, Jimmy Lin, Samuel Havens, Michael Carbin, Omar Khattab, and Andrew Drozdov. Freshstack: Building realistic benchmarks for evaluating retrieval on technical documents. *Advances in Neural Information Processing Systems*, 38, 2026.

Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 58(1):267–288, 1996.

Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, Pierre-Antoine Manzagol, and Léon Bottou. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of machine learning research*, 11(12), 2010.

Voyage AI. voyage-3-large: the new state-of-the-art general-purpose embedding model. https://blog.voyageai.com/2025/01/07/voyage-3-large/, January 2025. Voyage AI Blog.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 conference on empirical methods in natural language processing*, pp. 8696–8708, 2021.

Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F Xu, and Graham Neubig. Mconala: A benchmark for code generation from multiple natural languages. In *Findings of the association for computational linguistics: Eacl 2023*, pp. 265–273, 2023.

Jônatas Wehrmann, Otávio Parraga, and Rodrigo C Barros. Cobe: A natural language code search robustness benchmark. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, 2022.

Tiansheng Wen, Yifei Wang, Zequn Zeng, Zhong Peng, Yudi Su, Xinyang Liu, Bo Chen, Hongwei Liu, Stefanie Jegelka, and Chenyu You. Beyond matryoshka: Revisiting sparse coding for adaptive representation. *arXiv preprint arXiv:2503.01776*, 2025.

Weixiang Yan, Yuchen Tian, Yunzhe Li, Qian Chen, and Wen Wang. Codetransocean: A comprehensive multilingual benchmark for code translation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 5067–5089, 2023.

Ziyu Yao, Daniel S Weld, Wei-Peng Chen, and Huan Sun. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference*, pp. 1693–1703, 2018.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th international conference on mining software repositories (MSR)*, pp. 476–486. IEEE, 2018.

Dejiao Zhang, Wasi Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. Code representation learning at scale. *arXiv preprint arXiv:2402.01935*, 2024.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, et al. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

A Related Work

Code Retrieval and Code Representation Learning. Code retrieval and code representation learning aim to encode natural-language queries, source code, and program context into semantic embedding spaces for code search. Early work established large-scale neural code search and bimodal pre-training paradigms, including CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), CodeT5 (Wang et al., 2021), and UniXcoder (Guo et al., 2022), which respectively introduced natural-language/code pre-training, data-flow-aware representations, identifier-aware encoder-decoder learning, and unified cross-modal code representations. More recent studies have increasingly focused on scalable and retrieval-oriented embedding models. CoCoSoDa (Shi et al., 2023) improves code search through multimodal contrastive learning and hard negative mining, while CodeSage (Zhang et al., 2024) studies code representation learning at scale with encoder-based code embeddings for source-code understanding tasks. Beyond code-specific encoders, strong general-purpose embedding models such as BGE-M3 (Chen et al., 2024), GTE-Qwen2 (Li et al., 2023), NV-Embed (Lee et al., 2024), Jina Embeddings v3 (Sturua et al., 2024), and Qwen3-Embedding (Zhang et al., 2025) have pushed dense retrieval toward multilingual, long-context, multi-granularity, and instruction-aware embedding learning. Code-specialized embedding models have also emerged as a distinct line of work: Voyage-Code-3 (Voyage AI, 2025) is optimized for code retrieval with long-context support, multiple embedding dimensions, and quantized representations, while CodeXEmbed (Liu et al., 2024) unifies multilingual and multi-task code retrieval into a generalist code embedding framework and reports state-of-the-art results on code retrieval benchmarks; Jina Code Embeddings (Kryvosheieva et al., 2025) further explores deriving efficient code embeddings from code generation models for natural-language-to-code retrieval, technical QA, and cross-language code similarity. The evolution of code representation models reflects a shift from function-level semantic matching to scalable, context-aware, and multilingual embeddings that capture increasingly complex program semantics, with a growing emphasis on long-context, multi-task, and retrieval-optimized representations for real-world code search and understanding tasks.

Benchmarks in Code Retrieval. Benchmarks for code retrieval have evolved from documentation-based function search toward more realistic, robust, and context-aware retrieval settings. Early benchmarks such as CodeSearchNet (Husain et al., 2019) and the code search tasks in CodeXGLUE (Lu et al., 2021) established the dominant text-to-code retrieval protocol, where natural-language descriptions or docstrings are used to retrieve function-level code snippets across multiple programming languages. Subsequent datasets, including CoDesc (Hasan et al., 2021), StaQC (Yao et al., 2018), CoNaLa (Yin et al., 2018), and CoSQA (Huang et al., 2021), move closer to practical developer search scenarios by constructing query-code pairs from code descriptions, Stack Overflow questions, programming intents, or web queries. This line of benchmarks reveals an important mismatch between documentation-style queries and real user intents: the latter are often underspecified, noisy, ambiguous, and only partially aligned with the lexical surface form of the target code. More recent benchmarks further shift the focus from in-domain matching to semantic generalization and robustness. CodeSearchNet AdvTest (Lu et al., 2021) reduces reliance on identifier and function-name overlap, COBE (Wehrmann et al., 2022) evaluates robustness under natural-language and code transformations, and GenCodeSearchNet (Diera et al., 2023) systematically probes generalization in natural-language code search. In parallel, multilingual and multi-task benchmarks such as MCoNaLa (Wang et al., 2023), xCodeEval (Khan et al., 2023), and CoIR (Li et al., 2025) broaden code retrieval beyond English text-to-code search, covering multilingual queries, code-to-code retrieval, code-to-text retrieval, and hybrid retrieval settings. Finally, repository-level benchmarks such as RepoBench-R (Liu et al., 2023) and CrossCodeEval (Ding et al., 2023) suggest that practical code retrieval often requires locating relevant cross-file context rather than retrieving isolated snippets. Code retrieval benchmarks have experienced a shift from clean, function-level semantic matching toward evaluation settings that better capture real developer intents, robustness to superficial lexical cues, multilingual and cross-modal alignment, and context-sensitive retrieval over larger software artifacts.

Sparse Coding for Retrieval. Classical sparse coding (Olshausen & Field, 1997; Aharon et al., 2006; Tibshirani, 1996) originates from efficient coding and dictionary learning, where signals are reconstructed from a small number of learned atoms, with influential applications in image denoising (Elad & Aharon, 2006) and compressed sensing (Donoho, 2006). This general principle has been instantiated through several families of methods: optimization-based sparse coding and dictionary learning (Lee et al., 2006), which explicitly impose ℓ_1 or ℓ_0 constraints; sparse and denoising

autoencoders (Vincent et al., 2010), which learn nonlinear latent codes through reconstruction objectives with sparsity-inducing regularization; k-sparse and top-k mechanisms (Makhzani & Frey, 2013), which directly control the number of active latent features; and binary or hashing-based representations (Salakhutdinov & Hinton, 2009), which trade continuous similarity for compact and efficient retrieval codes. In information retrieval, this idea appears first in lexical sparse models such as TF-IDF and BM25, and more recently in learned sparse retrieval methods that combine neural semantic modeling with sparse index structures. DeepCT (Dai & Callan, 2020) learns contextualized term weights that can be stored in standard inverted indexes, DeepImpact (Mallia et al., 2021) and uniCOIL (Lin & Ma, 2021) improve impact-based sparse retrieval, COIL (Gao et al., 2021) introduces contextualized lexical matching over inverted lists, and SPLADE series (Formal et al., 2021b;a; Lassance et al., 2024) learn sparse lexical expansion representations that preserve exact-match efficiency while capturing semantic expansion beyond surface-form overlap. A recent and increasingly important line of work revisits sparse autoencoders (SAEs) as a mechanism for decomposing dense neural activations into overcomplete sparse feature dictionaries. SAEs trained on language-model activations have been shown to recover more monosemantic and causally meaningful features than raw neurons (Cunningham et al., 2023), while scaling studies, Top-K/Gated variants (Gao et al., 2024; Rajamanoharan et al., 2024a), JumpReLU SAEs (Rajamanoharan et al., 2024b), CSR (Wen et al., 2025; Guo et al., 2026), and open SAE suites such as Gemma Scope (Lieberum et al., 2024) improve the reconstruction-sparsity frontier and make such decompositions more scalable. For retrieval, these developments suggest a promising direction beyond purely dense embeddings: sparse codes can retain neural semantic abstraction while offering explicit feature-level evidence, controllable sparsity, reduced representational entanglement, and compatibility with efficient sparse indexing.

B Details in the Construction of Evolution-Aware Retrieval Data

This section describes the full construction pipeline of our evolution-aware retrieval data, from selecting anchor functions to generating and filtering evolved variants at different transformation levels. We also report the final dataset statistics after all validity checks, including the number of retained anchors and the number of valid variants in each evolution level.

Source and anchor preprocessing. We build our evolution-aware retrieval data from the Python subset of CodeSearchNet (Husain et al., 2019), which provides paired natural-language descriptions and function-level code snippets. Each original code snippet is treated as an anchor document $d_i^{(0)}$, and its associated natural-language description is used as the retrieval query q_i. For each anchor, we keep the original query fixed and construct multiple transformed snapshots of the corresponding code. Before applying any evolution operator, we apply a conservative preprocessing step to ensure that anchors are suitable for controlled rewriting. We discard snippets that cannot be parsed by the Python parser, snippets without a complete function definition and snippets with unsupported top-level structures. We also remove examples that are extremely short (less than 50 characters), or excessively long (more than 5000 characters). After preprocessing, the remaining anchors form the base set from which all three levels of evolution-aware variants are generated.

Level I: LibCST surface edits. The first evolution level is designed to isolate surface-form variation while preserving the underlying Python AST. To this end, we implement five deterministic transformations over the LibCST tree. First, we perturb whitespace and layout fields, including spaces around operators, blank lines, and line breaks in multi-line constructs. Second, we insert or remove redundant parentheses around expressions, conditions, return values, and assignment right-hand sides whenever the parenthesized form is syntactically equivalent. Third, we rewrite delimiter formatting in calls and container literals, such as adding or removing trailing commas and switching between single-line and multi-line argument layouts. Fourth, we normalize string literal surface forms when the evaluated string value is unchanged, for example by changing quote styles or escaping choices under a strict equivalence check. Fifth, we edit non-semantic comments by inserting, removing, or relocating non-docstring comments without touching executable statements. These operations are applied at the concrete-syntax level rather than the AST level, which allows us to create code variants that differ in tokenization and formatting while preserving the program structure. After each transformation, we parse both the original and transformed snippets with the Python AST parser and keep the variant only if the two ASTs are identical.

Level II: Cosmic-Ray AST-level edits. The second evolution level introduces stronger local code changes by applying mutation operators adapted from Cosmic Ray. Level II allows the transformed code to differ at the AST level, thereby testing whether retrieval representations remain stable under small but structural program edits. For each anchor snippet, we first check whether the code can be cleanly parsed; snippets containing Python error nodes are excluded from Cosmic-Ray mutation and handled separately. We then enumerate Cosmic-Ray operators that can be instantiated without additional arguments and apply them to valid mutation occurrences in the snippet. For each original code sample, we collect a pool of unique mutated candidates, with at most 256 candidates per sample and at most 96 mutation occurrences attempted for each operator. From this pool, we keep up to eight variants for each anchor. In our main construction, we use the deterministic first selection strategy with random seed 42, and process the CodeSearchNet-Python corpus splits independently. This level therefore creates controlled local structural changes that are stronger than formatting-only edits, while still keeping the transformation process deterministic, traceable, and aligned with the original retrieval corpus.

Level III: LM-prompted refactoring. The third evolution level is designed to introduce higher-level and more realistic code rewrites than the deterministic transformations used in Level I and Level II. We generate these variants with Qwen2.5-Coder-32B-Instruct (Hui et al., 2024), running inference on NVIDIA A100-SXM4-40GB GPUs with 4-bit quantization. For each anchor code snippet, we use a manually designed Python refactoring checklist that contains a broad set of behaviorally similar rewrite rules, such as replacing suitable loops with generator expressions or comprehensions.

To obtain diverse but controlled rewrites, we generate three LM-prompted variants for each anchor. For the i-th code snippet, we initialize a pseudo-random generator with seed $42+i$, randomly shuffle the full refactoring checklist, and split it into three disjoint rule subsets. Each subset is then inserted into the same prompt template, so that the model produces three variants of the same original code under different refactoring focuses. This design avoids using a single overly broad instruction set for every generation, while ensuring that all variants are generated from a fixed and reproducible pool of transformation rules. The prompt template used for generation is shown below.

System message. You are an expert Python developer. Your task is to refactor the given Python code while preserving its behavior exactly.

User message.

You will be given a Python code snippet and a subset of refactoring rules. Rewrite the code according to the rules whenever they are applicable.

Strict requirements:

1. Preserve the external behavior of the original code exactly.
2. Do not change the public function name, function signature, argument names, return type, or expected side effects.
3. Do not change the order of side effects, exception behavior, evaluation order, file-system effects, network effects, logging effects, mutation of inputs, or global-state updates.
4. Do not introduce new third-party dependencies.
5. Do not add explanations, comments about the rewrite, Markdown fences, or any text outside the rewritten code.
6. If a rule is not applicable to the given code, ignore that rule rather than forcing an unsafe transformation.
7. Prefer small, idiomatic, behavior-preserving refactorings over aggressive rewrites.
8. The output must be valid Python code.

Refactoring rules to consider:

1. {rule_1}
2. {rule_2}
3. {rule_3}
4. ...

Original code:

```
{code}
```

Return only the refactored Python code.

In the actual prompt, the placeholder {`code`} is replaced by the anchor code snippet, and the placeholders {`rule_1`}, {`rule_2`}, ... are replaced by one of the three shuffled checklist partitions. The complete checklist includes the following behaviorally similar refactoring rules:

- converting result-building `for` loops into generators, generator expressions, or lazy iterators when evaluation timing and side-effect order are unchanged
- rewriting nested `if-else` statements as early returns or guard clauses
- merging semantically related conditions and introducing intermediate boolean variables when necessary
- converting bounded `while` loops into `for` loops with `break`
- abstracting structurally similar branches with strategy functions, mapping tables, configuration objects, or polymorphic dispatch
- replacing stateless utility classes or namespace-only classes with functions
- normalizing private or internal function argument order while updating all call sites
- introducing intermediate variables for complex expressions

- rewriting complex `map` or `lambda` expressions as explicit loops when they involve complicated logic, side effects, exception handling, or debugging needs
- migrating path manipulation from `os.path` to `pathlib.Path` while using `str(path)` for APIs that require string paths
- replacing simple accumulation loops with list, set, or dictionary comprehensions
- extracting repeated blocks into intention-revealing helper functions
- replacing repeated magic numbers or strings with constants, enums, or configuration entries
- splitting long functions into smaller functions for parsing, validation, core logic, and formatting
- replacing large string- or type-based `if-elif` chains with dispatch tables, strategy functions, `match-case`, or `functools.singledispatch`
- replacing manual resource management with context managers
- fixing mutable default arguments using `None`-based initialization
- narrowing overly broad exception handlers while preserving exception context
- replacing manual index loops with `enumerate`, `zip`, or direct iteration
- simplifying redundant boolean-return branches into direct condition returns
- refactoring complex nested dictionary access with helper functions, `.get()` chains, dataclasses, or Pydantic-style models without suppressing originally expected exceptions
- converting simple data-carrier classes into `@dataclass` classes when they do not rely on complex inheritance or custom initialization
- replacing hand-written caches with `functools.lru_cache`, `cached_property`, or explicit cache objects when arguments are hashable and cache lifetime is correct
- replacing manual JSON string construction with `json.dumps` and encapsulating scattered JSON serialization or deserialization behind clear boundary functions
- extracting repeated validation logic into validation functions, decorators, or type constraints without hiding important caller-visible error handling

Filtering and validation. After generating candidate variants from the three transformation sources, we apply a shared conservative filtering pipeline to remove invalid or unreliable rewrites. We first require every candidate to be valid Python code: variants that cannot be parsed by the Python parser are discarded. We then check that the transformed snippet still exposes the same primary function as the anchor, including the same function name and compatible function signature. Variants that rename the entry function, delete the entry function, change the required arguments, or introduce unsupported top-level structures are removed, since such changes would break the alignment between the original CodeSearchNet query and its target code document. We further apply lightweight static checks to detect obvious scope and binding errors, such as references to newly introduced undefined variables, invalid control-flow statements, or helper functions that are called but not defined in the rewritten snippet. For LM-prompted variants, we additionally remove generations that include Markdown fences, natural-language explanations, incomplete code fragments, or multiple unrelated alternative solutions. To avoid degenerate examples, we also discard variants that are identical to the original snippet after normalization. Level-specific constraints are applied on top of this shared pipeline. For Level I variants, we require the Python AST of the transformed code to be identical to the AST of the original code, ensuring that this level captures only concrete-syntax and formatting-level changes. For Level II variants, we keep the mutation metadata produced by Cosmic Ray and record whether each candidate is generated by an actual mutation operator or by the deterministic text fallback. For Level III variants, we keep only generations that satisfy the prompt constraints. This filtering strategy does not claim to prove full semantic equivalence, which is difficult for arbitrary Python programs, but it substantially reduces invalid, interface-breaking, and clearly behavior-changing variants while keeping the construction scalable. These static and structural checks substantially reduce syntactically invalid, interface-breaking, and clearly unreliable variants, but they do not by themselves establish semantic equivalence, which is difficult to verify for arbitrary Python programs. We therefore complement them with the test-based semantic validation described below.

Table 5: Test-based semantic validation across different transformation levels. For each sampled original snippet, an LLM generates five execution tests, which are then applied to its corresponding variants. We report the average test-suite pass rate.

Variant Type	Test Suite Pass Rate (%)
Level I: LibCST surface edits	100.00
Level II: Cosmic-Ray AST edits	88.96
Level III: LM-prompted refactoring	93.43

Test-based semantic validation.

The filtering pipeline above removes syntactically invalid, interface-breaking, and obviously unreliable variants, but these checks do not directly verify behavioral consistency between an original snippet and its transformed versions. We therefore perform an additional test-based semantic validation on a sampled subset of the data.

Specifically, we sample 10000 original code snippets together with their corresponding variants. For each original snippet, we use an LLM to generate five executable test cases that approximate its observable behavior. We then run the same generated test suite on the corresponding Level I, Level II, and Level III variants and report the average test-suite pass rate for each transformation type.

As shown in Table 5, most generated variants preserve the behaviors exercised by the generated tests. Level I achieves a 100% pass rate, which is consistent with its construction: LibCST transformations modify only concrete syntax while preserving the Python AST. Level II obtains a lower pass rate of 88.96%, as Cosmic-Ray mutation operators introduce localized AST-level changes that may alter program behavior. Level III achieves a 93.43% pass rate; although the generation prompt explicitly encourages behavior-preserving rewrites and the resulting variants undergo additional filtering, some rewrites may still change behaviors not fully constrained by the generation procedure.

We emphasize that this evaluation provides only an empirical underapproximation of behavioral consistency rather than a proof of full semantic equivalence. The generated test suites cover only a limited subset of possible inputs and execution paths. We therefore use this analysis as a complementary validation of the semantic quality of the generated variants, rather than as a formal equivalence guarantee.

Positive-set organization and dataset statistics. We organize the resulting data by evolution level rather than merging all transformed variants into a single positive pool. For each level, the original anchor document $d_i^{(0)}$ and its valid transformed snapshots $\{d_i^{(1)}, \ldots, d_i^{(L)}\}$ are associated with the same retrieval query q_i, forming a level-specific positive set P_i^{level}. This organization preserves the distinction between surface-level concrete-syntax edits, AST-level local mutations, and LM-prompted higher-level refactorings. It therefore allows us to evaluate not only whether retrieval quality is preserved under evolution, but also how retrieval degradation, sparse-support overlap, and routing consistency vary as the transformation strength increases. We report the final number of retained examples for each evolution level in Table 6.

Evolution level	Number of variants
Level I: LibCST surface edits	304173
Level II: Cosmic-Ray AST-level edits	453188
Level III: LM-prompted refactoring	190256

Table 6: Statistics of the evolution-aware retrieval data after filtering and validation.

C EXPERIMENT DETAILS

C.1 DATASETS

Apart from the introduced evolution-aware dataset, we also include nine public code-to-code or text-to-code retrieval tasks from COIR (Li et al., 2025) and MTEB (Muennighoff et al., 2023).

- **DS1000Retrieval (DS1000)** (Lai et al., 2023): DS-1000 is a natural data-science code benchmark comprising 1,000 StackOverflow-derived Python problems across seven libraries, with reliable execution-based evaluation and perturbations designed to reduce memorization.
- **FreshStackRetrieval (FS)** (Thakur et al., 2026): FreshStack is a realistic code-retrieval benchmark built from community-asked Stack Overflow questions on five recent, niche programming topics and GitHub technical-document corpora, with nugget-level relevance judgments for evaluating retrieval and RAG systems in challenging developer-assistance settings.
- **HumanEvalRetrieval (HE)** (Chen et al., 2021): HumanEvalRetrieval is based on HumanEval, a benchmark of 164 hand-written Python programming problems with function signatures, docstrings, reference solutions, and unit tests, designed to evaluate functional correctness in synthesizing code from natural-language specifications.
- **MBPPRetrieval (MBPP)** (Austin et al., 2021): MBPPRetrieval is built upon the Mostly Basic Programming Problems (MBPP) benchmark, which contains 974 short, crowd-sourced Python programming tasks with natural-language descriptions, reference solutions, and test cases designed for entry-level programmers.
- **AppsRetrieval (APPs)** (Hendrycks et al., 2021): AppsRetrieval is built on APPS, a benchmark of 10,000 natural-language programming problems spanning introductory, interview, and competition levels, with human-written solutions and test cases for evaluating code-generation correctness.
- **CodeSearchNetRetrieval (CodeS)** (Husain et al., 2019): CodeSearchNetRetrieval is a code retrieval dataset derived from the CodeSearchNet Corpus, where natural-language documentation comments are used as queries to retrieve their corresponding functions from a large multilingual corpus of open-source code.
- **CosQA** (Huang et al., 2021): CoSQA is a human-annotated dataset of 20,604 real-world web query–Python function pairs, designed for evaluating and training models on code search and code question answering.
- **CodeSearchNetCCRetrieval (CodeC)** (Li et al., 2025): CodeSearchNetCCRetrieval is a code-to-code retrieval dataset derived from CodeSearchNet for code context retrieval, where each GitHub function is split into an initial code segment as the query and the remaining segment as the retrieval target, covering six programming languages with a 1M-document corpus.
- **CodeTransOceanDL(CodeDL)** (Yan et al., 2023): CodeTransOcean-DL is a Python deep-learning similar-code retrieval dataset derived from CodeTransOcean, where each query code snippet is matched to semantically equivalent code implemented in a different deep learning framework (e.g., TensorFlow, PyTorch, PaddlePaddle, or MXNet), with 564/72/180 train/dev/test queries and 816 corpus instances.

C.2 EVALUATION METRICS

We evaluate evolution-aware retrieval from three complementary perspectives: retrieval quality, stability of sparse-neuron priorities, and overlap of active sparse neurons. All metrics are averaged over the evaluation set. Let $d_i^{(0)}$ denote the original code snippet, and let $d_i^{(1)}, \ldots, d_i^{(L)}$ denote its evolved variants. Following the notation in Section 2, the positive set for query q_i is $P_i = \{d_i^{(0)}, d_i^{(1)}, \ldots, d_i^{(L)}\}$. For sparse representations, let $\mathbf{z}_i^{(\ell)} \in \mathbb{R}^h$ be the sparse code of snapshot $d_i^{(\ell)}$, and let $S_K(d_i^{(\ell)})$ denote the indices of its Top-K active neurons.

- **Retrieval Quality (nDCG@10).** nDCG@10 measures whether the retriever ranks behaviorally similar positive snapshots above unrelated code snippets. For each query q_i, we rank all candidate documents in descending order according to $\mathrm{sim}(\mathbf{E}_{\mathcal{Q}}(q_i), \mathbf{E}_{\mathcal{D}}(d))$, and denote the document at rank r by $\pi_i(r)$. Let $\mathrm{rel}_i(d)$ be the relevance score of document d with respect to q_i. In our evolution-aware setting, unless otherwise specified, we use binary relevance:

$$\mathrm{rel}_i(d) = \mathbb{1}[d \in P_i].$$

The discounted cumulative gain at cutoff 10 is

$$\mathrm{DCG@10}(q_i) = \sum_{r=1}^{10} \frac{2^{\mathrm{rel}_i(\pi_i(r))} - 1}{\log_2(r+1)}.$$

Let $\mathrm{IDCG@10}(q_i)$ be the maximum possible DCG@10 obtained by an ideal ranking for q_i. The final retrieval score is

$$\mathrm{nDCG@10} = \frac{1}{|\mathcal{Q}|} \sum_{q_i \in \mathcal{Q}} \frac{\mathrm{DCG@10}(q_i)}{\mathrm{IDCG@10}(q_i)}.$$

- **Activation Order Consistency.** Activation Order Consistency measures whether an original snippet and its evolved variants preserve the relative priority order of their commonly activated neurons.

For the original snippet $d_i^{(0)}$ and an evolved variant $d_i^{(\ell)}$, define their shared Top-K active neurons as

$$C_i^{(\ell)} = S_K(d_i^{(0)}) \cap S_K(d_i^{(\ell)}).$$

We compare the relative order of neuron pairs inside $C_i^{(\ell)}$. To avoid unstable comparisons between nearly tied neurons, we use the same near-tie exclusion as in the order regularizer:

$$\Omega_i^{(\ell)} = \left\{ (r,s) : r,s \in C_i^{(\ell)},\ r < s,\ \left| z_{i,r}^{(0)} - z_{i,s}^{(0)} \right| > \epsilon \right\}.$$

A pair (r,s) is order-consistent if the original and evolved snapshots assign the same relative priority to the two neurons:

$$\left(z_{i,r}^{(0)} - z_{i,s}^{(0)} \right) \left(z_{i,r}^{(\ell)} - z_{i,s}^{(\ell)} \right) > 0.$$

Therefore, Activation Order Consistency at sparsity K is defined as

$$\mathrm{AOC@}K = \frac{1}{|\mathcal{B}|L} \sum_{d_i \in \mathcal{B}} \sum_{\ell=1}^{L} \frac{1}{|\Omega_i^{(\ell)}|} \sum_{(r,s) \in \Omega_i^{(\ell)}} \mathbb{1}\left[\left(z_{i,r}^{(0)} - z_{i,s}^{(0)} \right) \left(z_{i,r}^{(\ell)} - z_{i,s}^{(\ell)} \right) > 0 \right],$$

where $\mathcal{B}$ denotes the evaluated set of original snippets.

- **Neuron Overlap.** Neuron overlap measures the extent to which an original snippet and its evolved variants activate the same Top-K sparse neurons.

For each original-evolved pair $(d_i^{(0)}, d_i^{(\ell)})$, we compute the fraction of Top-K neurons shared by the two sparse codes:

$$\mathrm{Overlap@}K(d_i^{(0)}, d_i^{(\ell)}) = \frac{\left| S_K(d_i^{(0)}) \cap S_K(d_i^{(\ell)}) \right|}{K}.$$

The final Neuron Overlap score is the average overlap across all snippets and all evolutionary variants:

$$\mathrm{NO@}K = \frac{1}{|\mathcal{B}|L} \sum_{d_i \in \mathcal{B}} \sum_{\ell=1}^{L} \mathrm{Overlap@K}(\mathrm{d_i^{(0)}}, \mathrm{d_i^{(\ell)}}).$$

C.3 Detailed Results of Evaluation on Benchmark Retrieval Preservation.

Table 7 evaluates whether regularizations compromise standard benchmark performance.

Backbone	K	R1	R2	DS1000	FS	HE	MBPP	APPs	CodeS	CosQA	CodeC	CodeDL	Avg.
Jina-1.5B	1536	-		62.88	38.93	98.41	90.13	86.63	91.38	35.10	91.12	37.28	70.21
	32	✗	✗	59.14 ± 1.62	31.75 ± 0.39	97.15 ± 0.78	87.79 ± 0.91	83.19 ± 1.46	90.07 ± 0.71	32.08 ± 0.65	89.36 ± 0.52	31.64 ± 1.13	66.91
		✓	✗	58.60 ± 0.72	32.45 ± 0.63	96.82 ± 0.89	86.32 ± 0.72	81.08 ± 0.53	91.30 ± 0.64	30.72 ± 1.27	87.20 ± 1.04	29.35 ± 0.87	66.00
		✗	✓	57.93 ± 2.06	30.90 ± 1.30	96.55 ± 0.85	85.56 ± 1.53	81.47 ± 0.96	89.10 ± 0.72	31.19 ± 1.34	87.90 ± 0.82	28.97 ± 0.57	65.51
		✓	✓	58.12 ± 1.48	32.76 ± 1.23	97.60 ± 0.39	86.42 ± 0.76	82.22 ± 1.24	89.85 ± 0.89	30.53 ± 1.84	87.73 ± 0.97	29.18 ± 1.56	66.05
	16	✗	✗	51.20 ± 1.42	25.40 ± 0.58	92.86 ± 0.45	82.25 ± 0.71	79.52 ± 0.83	89.62 ± 1.28	27.21 ± 0.96	88.26 ± 1.42	22.45 ± 0.23	62.09
		✓	✗	49.24 ± 0.76	24.64 ± 1.29	92.17 ± 0.94	81.63 ± 1.41	78.18 ± 0.64	90.57 ± 0.58	24.86 ± 1.03	86.85 ± 1.14	20.10 ± 0.79	60.92
		✗	✓	49.75 ± 1.03	24.19 ± 0.89	91.93 ± 1.24	80.79 ± 0.72	78.43 ± 0.13	88.93 ± 0.76	25.13 ± 0.84	87.04 ± 1.13	18.76 ± 1.35	60.55
		✓	✓	50.09 ± 1.08	25.53 ± 0.96	93.04 ± 1.42	81.34 ± 0.65	78.69 ± 1.26	90.34 ± 1.53	25.52 ± 0.96	87.43 ± 0.63	19.58 ± 0.92	61.28
Qwen3-4B	2560	-		65.43	40.46	98.75	91.09	82.18	93.20	40.38	93.26	32.19	70.78
	32	✗	✗	60.28 ± 0.82	34.57 ± 0.76	96.14 ± 1.43	88.46 ± 1.70	80.13 ± 0.58	92.12 ± 0.81	37.41 ± 0.92	91.92 ± 1.43	27.63 ± 1.18	67.63
		✓	✗	58.35 ± 0.00	33.19 ± 0.74	95.02 ± 0.82	87.31 ± 0.47	79.90 ± 1.38	90.97 ± 1.03	38.06 ± 0.85	92.79 ± 1.54	28.34 ± 0.75	67.10
		✗	✓	59.13 ± 2.03	32.88 ± 0.40	95.15 ± 1.16	86.94 ± 0.37	80.19 ± 0.62	91.46 ± 1.15	36.52 ± 0.76	91.56 ± 1.74	27.29 ± 0.72	66.79
		✓	✓	60.05 ± 0.81	33.04 ± 1.26	94.86 ± 0.63	87.12 ± 1.05	79.77 ± 0.54	90.48 ± 1.59	36.97 ± 1.83	91.71 ± 0.72	27.63 ± 0.28	66.85
	16	✗	✗	53.05 ± 1.43	28.81 ± 0.95	93.59 ± 1.83	83.52 ± 0.75	77.96 ± 0.82	90.04 ± 1.30	31.09 ± 1.28	88.64 ± 0.63	15.48 ± 1.47	62.46
		✓	✗	51.24 ± 0.63	27.17 ± 0.52	92.12 ± 1.28	81.76 ± 1.46	76.05 ± 0.85	87.65 ± 0.45	29.16 ± 0.90	87.07 ± 1.22	12.72 ± 0.94	60.55
		✗	✓	52.79 ± 0.76	26.89 ± 1.21	92.37 ± 0.84	82.47 ± 0.69	75.73 ± 0.92	88.40 ± 1.27	28.34 ± 1.46	87.60 ± 0.71	11.03 ± 1.76	60.63
		✓	✓	52.08 ± 0.63	27.03 ± 1.74	90.98 ± 1.52	80.91 ± 1.18	75.84 ± 0.86	87.31 ± 1.32	28.49 ± 0.90	88.30 ± 2.27	11.59 ± 1.69	60.28

Table 7: **Evaluation on benchmark preservation under regularization.** We evaluate whether the proposed regularizations in Section 4.1 (**R1**) and Section 4.2 (**R2**) preserve original out-of-domain retrieval quality. The evaluation metric is nDCG@10, and evaluation is performed on five independent runs with different random seeds, with mean and standard deviation reported.

C.4 Implementation Details

All experiments in this paper were conducted on a cluster with 8 NVIDIA A100-SXM4-40GB GPUs and an Intel(R) Xeon(R) Platinum 8275CL CPU @ 3.00GHz.

Backbone Embedding Extraction. We extract and cache dense embeddings from frozen code embedding backbones before training sparse representations. The same extraction protocol is used for original code snippets, evolved variants, and natural-language queries unless otherwise specified. Detailed setups are provided in Table 8.

Backbone	Dimension	Precision	Maximum query length	Maximum code length
Jina-Code-1.5B	1536	BF16	32768	32768
Qwen3-4B	2560	BF16	32768	32768

Table 8: Backbone Embedding Extraction Settings.

Sparse Autoencoder Training. We train a lightweight sparse autoencoder on top of cached dense embeddings. The encoder maps each dense embedding into an overcomplete non-negative latent space, keeps the Top-K activations, and reconstructs the original dense representation. Detailed setups are provided in Table 9.

Evolution-Consistency Training. We further train sparse representations with evolutionary groups. Each group contains one original code snippet and its evolved variants, and the training objective combines the base sparse-coding loss with neuron-overlap and activation-order consistency regularization. We use the same Top-K values as the sparsity budget in the regularizers. Detailed setups are provided in Table 10.

Backbone	d	h	K	β	γ	τ	Optimizer	lr	Batch size	Epoch
Jina-Code-1.5B	1536	32768	$\{4, ..., 64\}$	1/32	0.1	0.1	AdamW	4e-5	256	10
Qwen3-4B	2560	32768	$\{4, ..., 64\}$	1/32	0.1	0.1	AdamW	1e-5	256	10

Table 9: Sparse Autoencoder Training Settings.

Variant Number per Anchor	λ_{ov}	λ_{ord}	τ_m
8	0.1	0.05	0.05

Table 10: Evolution-Consistency Training Settings.

C.5 Robustness to Evolution Frequency

Our main benchmark construction intentionally includes multiple evolved revisions per anchor to stress-test representation and index reuse. In real codebases, however, many functions may remain unchanged or be modified only infrequently. We therefore subsample evolved variants to vary the average number of variants per anchor from 0 to 8, while keeping the sparsity budget fixed at $K = 32$ and using Qwen3-Embedding-4B as the backbone.

Average variants per anchor	**Index Size (GB)**
0	3.37
0.5	2.79
1	2.45
2	2.08
4	1.90
8	1.74

Table 11: Index size under different evolution frequencies.

As shown in Table 11, the index-sharing benefit decreases smoothly as fewer evolved revisions are present, but remains substantial even in low-frequency settings. At an average of 0.5 variants per anchor, the index size decreases from 3.37 to 2.79, corresponding to a 17.2% reduction. The benefit grows consistently as more related revisions coexist. These results indicate that the observed memory savings are not solely a consequence of densely populating every anchor with generated variants. Rather, they persist when only a subset of the corpus contains reusable, behaviorally similar revisions.

C.6 Comparison with Conventional Vector Compression under Matched Memory Budgets.

Experiment Setup. To distinguish the benefit of evolution-aware index reuse from generic representation compression, we compare our method with two conventional vector-compression baselines: Product Quantization (PQ) and Semantic Hashing (SH). All methods use Qwen3-Embedding-4B as the underlying backbone, and we configure the baselines to approximately match the index-memory budgets of our sparse representations. Specifically, for the Top-$K = 16$ setting, PQ uses 16 subspaces with 256 centroids per subspace, while Semantic Hashing uses a 128-bit binary code (SH-128). For Top-$K = 32$, PQ uses 32 subspaces with 256 centroids per subspace and Semantic Hashing uses a 256-bit code (SH-256). These configurations result in approximately matched index sizes.

Results. As shown in Table 12, under the memory footprint of our $K = 16$ sparse setting, PQ and SH require approximately 0.84 GB and achieve 57.52 and 60.97 nDCG@10, respectively, while our method uses 0.79 GB and reaches 62.46. Similarly, when matching the memory footprint of our $K = 32$ setting, PQ and SH use 1.69 GB and obtain 61.58 and 63.31 nDCG@10, whereas our method uses 1.74 GB and achieves 66.85. These results suggest that the advantage of our approach

Reference Sparse Budget	Method	Configuration	Index Size (GB)	nDCG@10 (%)
Ours, $K = 16$	PQ	16 subspaces, 256 centroids	0.84	57.52
	SH	128-bit code	0.84	60.97
	Ours	$K = 16$	**0.79**	**62.46**
Ours, $K = 32$	PQ	32 subspaces, 256 centroids	1.69	61.58
	SH	256-bit code	1.69	63.31
	Ours	$K = 32$	1.74	**66.85**

Table 12: Retrieval performance under approximately matched index-memory budgets. For each sparse operating point of our method ($K = 16$ and $K = 32$), we configure Product Quantization (PQ) and Semantic Hashing (SH) to obtain a comparable index size.

is not solely due to reducing the storage cost of individual representations: exploiting shared sparse structure across behaviorally similar revisions provides an additional source of memory efficiency.

D Additional Validation of Revision-Consistent Sparse Structure

Our main experiments use CODECONTINUUM-R as a controlled environment for studying whether behaviorally related code revisions expose reusable sparse index-facing structure. A natural question is whether the observed sparse-support consistency is specific to our generated revisions, or whether it can be explained primarily by lexical or structural similarity between code snippets.

We therefore perform two complementary analyses. First, we explicitly control lexical and AST-level structural similarity within the LM-generated revision pairs and compare them with unrelated high-similarity pairs. Second, we evaluate the same sparse-reuse phenomenon on naturally occurring revision pairs mined from the Git histories of widely used Python repositories. We additionally use CodeTransOcean-DL (Yan et al., 2023) as an independently collected semantic-positive control with cross-framework implementations. Together, these analyses test whether the revision-consistent sparse structure observed in our benchmark persists beyond the specific transformation pipeline used to construct CODECONTINUUM-R.

D.1 Evaluation Metrics

We use lexical similarity and AST similarity to separately characterize surface-form and structural similarity between code snippets.

Lexical similarity. We tokenize each code snippet and compute the Jaccard similarity between the resulting token sets. For two code snippets d_i and d_j with token sets $\mathcal{T}(d_i)$ and $\mathcal{T}(d_j)$, we define

$$\mathrm{LexSim}(d_i, d_j) = \frac{|\mathcal{T}(d_i) \cap \mathcal{T}(d_j)|}{|\mathcal{T}(d_i) \cup \mathcal{T}(d_j)|}.$$

A value close to one indicates strong token-level surface overlap, whereas a lower value indicates larger lexical divergence.

AST similarity. We follow the Tree Similarity of Edit Distance (TSED) formulation (Song et al., 2024). Each snippet is parsed into a syntax tree, and the structural distance between two trees G_i and G_j is computed using the APTED tree-edit-distance algorithm. Let $\Delta(G_i, G_j)$ denote the minimum accumulated cost of node insertion, deletion, and relabeling operations required to transform one tree into the other. We convert this distance into a normalized similarity score:

$$\mathrm{ASTSim}(d_i, d_j) = \max\left(1 - \frac{\Delta(G_i, G_j)}{\max(|G_i|, |G_j|)}, 0\right),$$

where $|G|$ denotes the number of nodes in the corresponding syntax tree. Unlike lexical similarity, this metric explicitly measures similarity in program structure.

Sparse-support and routing consistency. Sparse-support consistency is measured using Overlap@K, as defined in Appendix C.2. We report Overlap@16 and Overlap@32 for the controlled pair analysis, and Overlap@32 for the natural-revision experiments. For the latter, we additionally report **Route@4 preserved**, which measures the fraction of consecutive revision pairs that preserve the small-budget routing structure used when only the strongest four sparse activations are retained for coarse routing.

Memory saving. For natural revision pairs, we also report the relative storage reduction obtained by reusing the sparse structure shared across consecutive snapshots, compared with storing the two snapshots independently. This metric directly quantifies the system-level benefit associated with the observed support reuse.

D.2 Controlling for Lexical and Structural Similarity

A possible alternative explanation for high sparse-support overlap is that behaviorally related revisions simply remain lexically or structurally similar. This concern is particularly relevant because stronger code evolutions generally induce larger surface-form changes and also tend to exhibit lower support overlap. We therefore construct several controls that explicitly vary structural similarity while preserving or removing the revision relationship.

We consider the following pair types:

- **All LM revisions.** Each pair consists of an original code snippet and one of its LM-generated behaviorally similar revisions. This is the natural Level-III revision setting used in CODECONTINUUM-R.
- **LM revisions with low AST similarity.** We rank LM-revision pairs by AST similarity and separately evaluate the bottom-25% and bottom-10% subsets. These subsets contain the most structurally divergent LM revisions and test whether sparse-support consistency remains high after substantial structural change.
- **Unrelated high-similarity pairs.** As a counterfactual control, we evaluate unrelated functions with high lexical and AST similarity. These pairs do not share a revision relationship despite having strong surface-form and structural similarity. If sparse overlap were driven mainly by these two factors, such unrelated pairs should exhibit support overlap comparable to the behaviorally related revisions.

Pair Type	Lexical Sim.	AST Sim.	Overlap@16	Overlap@32
LM revisions (all)	0.8393	0.7581	92.79	88.96
LM revisions (bottom-25%)	0.7107	0.4656	85.86	87.98
LM revisions (bottom-10%)	0.6543	0.3604	84.70	86.42
Unrelated high-similarity pairs	0.7967	0.9624	54.13	53.94

Table 13: Sparse-support overlap under different lexical and structural similarity conditions. The bottom-25% and bottom-10% rows contain the most structurally dissimilar LM-generated revisions. The unrelated control contains function pairs with high lexical and AST similarity but without a revision relationship.

Table 13 shows that reducing structural similarity only moderately decreases sparse-support overlap for behaviorally related revisions. Across all LM-generated revisions, the average overlap is 92.79% at $K = 16$ and 88.96% at $K = 32$. Even among the bottom-10% most structurally dissimilar LM revisions, where AST similarity drops from 0.7581 to 0.3604, overlap remains high at 84.70% and 86.42%, respectively.

The unrelated control provides a complementary contrast. Although these pairs have high lexical and AST similarity, with average similarities of 0.7967 and 0.9624, their sparse-support overlap is only 54.13% at $K = 16$ and 53.94% at $K = 32$. Thus, strong lexical or structural similarity alone is insufficient to reproduce the support consistency observed between behaviorally related revisions.

D.3 Validation on Natural Repository Revisions

The preceding analysis controls for surface and structural similarity within our generated revision data, but it does not by itself rule out the possibility that the phenomenon arises from the synthetic transformation pipeline. We therefore perform a second evaluation using naturally occurring revisions mined from the Git histories of widely used Python repositories.

Revision-pair construction. We collect consecutive code revisions from seven popular open-source projects, including Django, FastAPI, pandas, pytest, PyTorch, scikit-learn, and Transformers. We focus on localized commits and filter out changes that are either too small to constitute a meaningful revision, such as trivial edits, or too broad to provide a localized evolutionary relationship, such as sweeping repository-wide changes. For each retained code object, we pair consecutive snapshots and evaluate whether their sparse index-facing structures remain reusable. For repositories with sufficiently many valid pairs, we sample up to 5000 revision pairs. We also construct a mixed set by sampling pairs across all repositories. Unlike CODECONTINUUM-R, none of these revision pairs are produced by our LibCST, Cosmic-Ray, or LM-based transformation pipeline.

Results. Table 14 shows that revision-consistent sparse structure also appears in naturally evolving repositories. Across all evaluated projects, consecutive revisions retain high Overlap@32, ranging from 0.9456 for FastAPI to 0.9888 for Transformers. The corresponding Route@4 preservation rates are above 0.997 in every repository, indicating that the strongest sparse routing signals are highly

Repository	Pair Num.	Lexical Sim.	Overlap@32	Route@4 Preserved	Memory Saving
Mixed	5,000	0.9899	0.9809	0.9992	73.57%
Django	5,000	0.9849	0.9720	0.9996	72.90%
FastAPI	1,779	0.9772	0.9456	0.9970	70.92%
pandas	5,000	0.9892	0.9787	0.9996	73.40%
pytest	2,621	0.9816	0.9643	0.9978	72.32%
PyTorch	5,000	0.9911	0.9839	0.9998	73.79%
scikit-learn	5,000	0.9919	0.9822	0.9994	73.67%
Transformers	5,000	0.9936	0.9888	0.9998	74.16%

Table 14: Sparse-structure consistency on naturally occurring code revisions mined from Git histories. **Pair Num.** denotes the number of consecutive revision pairs. We report lexical similarity, sparse-support overlap at $K = 32$, preservation of the top-4 routing structure, and the memory saving obtained from revision-level sparse reuse.

stable across these naturally occurring changes. This consistency translates directly into storage reuse. Revision-level sparse sharing yields memory savings between 70.92% and 74.16% across the evaluated repositories, with the mixed set achieving a 73.57% reduction. These results show that the index-reuse opportunity observed in CODECONTINUUM-R is also present in real repository histories and is not created solely by our synthetic revision-generation procedure.

D.4 External Cross-Framework Semantic Pairs

Finally, we evaluate independently collected semantic-positive pairs from CodeTransOcean-DL (Yan et al., 2023). These pairs implement the same or closely related functionality in different deep-learning frameworks and are not generated by our evolution pipeline. They therefore provide an external setting in which implementations can differ while remaining semantically related.

Pair Type	Lexical Sim.	AST Sim.	Overlap@16	Overlap@32
CodeTransOcean-DL positive pairs	0.8285	0.8367	84.20	81.46

Table 15: Sparse-support consistency on CodeTransOcean-DL positive cross-framework pairs. These pairs are independently collected and are not produced by our revision-generation pipeline.

As shown in Table 15, despite implementation and framework differences, CodeTransOcean-DL positive pairs retain 84.20% overlap at $K = 16$ and 81.46% at $K = 32$. This provides additional evidence that high sparse-support consistency is not restricted to the specific transformations used in CODECONTINUUM-R.

D.5 Discussion

The three analyses above address complementary alternative explanations for the main empirical observation of this work. First, structurally distant LM revisions retain high sparse-support overlap, showing that substantial AST-level change does not eliminate support consistency. Second, unrelated functions with high lexical and structural similarity exhibit much lower overlap, indicating that surface-form and structural similarity alone are insufficient to explain the phenomenon. Third, both naturally occurring Git revisions and independently collected CodeTransOcean-DL semantic pairs exhibit strong sparse-support consistency outside our synthetic revision pipeline.

We therefore interpret revision-consistent sparse structure as a naturally occurring property of the learned sparse representations that can be exploited for index reuse. We do not claim that individual sparse neurons correspond exclusively to program semantics, nor that all forms of software evolution preserve the same degree of sparse structure. Rather, the evidence shows that behaviorally related code revisions retain substantially more reusable latent structure than can be explained solely by measured surface or structural similarity.

Importantly, this observation precedes the evolution-aware regularizers proposed in Section 4. The regularizers are introduced to further strengthen an already emerging property, particularly support stability and activation priority under memory-constrained routing—rather than to create revision-level reuse from scratch. This distinction motivates our view of CODECONTINUUM-R as a controlled testbed for studying revision-aware sparse indexing, while the natural-revision and external-pair experiments demonstrate that the underlying reuse opportunity extends beyond the benchmark construction itself.

E Composition with Sparse ANN Retrieval Systems

Our main experiments study how sparse index-facing representations can be shared across behaviorally related code revisions. This problem is complementary to sparse approximate nearest-neighbor (ANN) retrieval, which asks how to efficiently search a given collection of sparse vectors. To make this distinction concrete, we instantiate our revision-aware index construction on top of Seismic (Bruch et al., 2025), a representative sparse ANN engine, and evaluate the resulting end-to-end retrieval system.

E.1 Seismic as the sparse retrieval backend.

Seismic is designed for approximate maximum-inner-product search over learned sparse representations. Given a fixed collection of sparse document vectors, it builds an inverted index over active dimensions. To reduce query-time work, Seismic prunes long inverted lists, partitions their postings into geometrically cohesive blocks, and associates each block with a compact summary of the vectors it contains. At query time, these block summaries are used to identify promising regions of the inverted lists and skip blocks that are unlikely to contribute high-scoring documents. Candidate documents from the remaining blocks are subsequently scored using the sparse vectors stored in the forward index. Thus, Seismic improves the efficiency of searching a sparse vector collection, while ordinarily treating every indexed vector as an independent document.

Our revision-aware indexing operates at a different layer. Rather than replacing the sparse ANN algorithm, it changes which revision-level information must be stored independently by the retrieval engine. This makes the two techniques composable: an existing sparse ANN engine can search representatives of revision families, while the shared and variant-specific sparse information maintained by our index construction is used to recover individual revisions afterwards.

E.2 Revision-aware composition.

We retain the retrieval task used throughout the paper. Queries are the natural-language descriptions from CodeSearchNet-Python, and the corpus is constructed from Level-III revision families. Each family contains one original anchor and three LM-generated revisions, and all four snapshots are treated as relevant documents for the corresponding query.

We compare three retrieval configurations:

- **Exhaustive.** We store every sparse snapshot independently and compute sparse similarity exhaustively against the full snapshot collection. This provides an exact sparse-retrieval reference without ANN pruning.
- **Seismic-Full-NL.** We directly index every original and revised snapshot as an independent sparse vector in Seismic. This is the standard way of applying a sparse ANN engine to our retrieval corpus and does not exploit relationships among revisions of the same anchor.
- **Seismic-Group-NL.** We combine Seismic with revision-aware index reuse. Instead of inserting every revision independently into the Seismic engine, we index one representative sparse vector for each revision family. Information specific to the remaining revisions is stored separately as sparse residuals relative to the shared family representation. At retrieval time, Seismic first retrieves candidate revision families through their representatives. We then expand the corresponding revisions and rerank them using exact sparse similarity. Consequently, Seismic remains responsible for efficient first-stage sparse ANN search, whereas our method reduces redundant revision-level storage before that search is performed.

This composition does not modify the retrieval objective or the sparse similarity function. It only changes the organization of the indexed corpus: the **Full** configuration exposes all snapshots independently to Seismic, whereas the **Group** configuration exposes family representatives to Seismic and retains revision-specific information outside the ANN engine.

E.3 EVALUATION.

We evaluate corpora containing 10K and 50K anchor functions, corresponding to 40K and 200K total snapshots, respectively. We report nDCG@10 and Recall@100 to measure retrieval quality. We separately measure **Engine Bytes**, the storage consumed by the searchable sparse index maintained by the retrieval engine, and **Residual Bytes**, the additional storage used for revision-specific residual information in the grouped configuration. **Total Bytes** is their sum. We additionally report p50 query latency to measure the median retrieval time of each configuration.

Anchors	Method	nDCG@10	Recall@100	Engine Bytes	Residual Bytes	Total Bytes	p50 Latency
10K	Exhaustive	0.9618	0.9920	5,489,844	–	5,489,844	16.2859 ms
	Seismic-Full-NL	0.9618	0.9920	21,521,009	–	21,521,009	3.8077 ms
	Seismic-Group-NL	0.9619	0.9922	8,553,609	3,457,643	12,011,252	2.3968 ms
50K	Exhaustive	0.9224	0.9821	27,409,844	–	27,409,844	77.6055 ms
	Seismic-Full-NL	0.9224	0.9821	70,141,614	–	70,141,614	6.4656 ms
	Seismic-Group-NL	0.9222	0.9824	29,060,151	17,281,664	46,341,815	6.5308 ms

Table 16: Composition of revision-aware sparse indexing with Seismic. SEISMIC-FULL-NL indexes every revision independently, whereas SEISMIC-GROUP-NL indexes one representative per revision family and stores revision-specific sparse residuals separately. Retrieval quality, storage, and median query latency are reported at two corpus scales.

Results. Table 16 shows that revision-aware grouping can be combined with sparse ANN retrieval without materially changing retrieval quality. At 10K anchors, SEISMIC-GROUP-NL obtains 0.9619 nDCG@10 and 0.9922 Recall@100, compared with 0.9618 and 0.9920 for SEISMIC-FULL-NL. At 50K anchors, the corresponding results are 0.9222/0.9824 and 0.9224/0.9821, respectively. The small differences indicate that retrieving representative families and subsequently expanding their revisions preserves the relevant candidate set in this setting.

The main difference appears in index storage. When all 10K revision families are indexed snapshot-by-snapshot, Seismic consumes 21.52M bytes. Grouping the revisions reduces the Seismic engine itself to 8.55M bytes; even after adding 3.46M bytes of revision-specific residuals, total storage is 12.01M bytes, a 44.2% reduction relative to SEISMIC-FULL-NL. At 50K anchors, total storage decreases from 70.14M to 46.34M bytes, corresponding to a 33.9% reduction. The reduction in *Engine Bytes* is even larger because only family representatives participate directly in the ANN index.

Importantly, this storage reduction does not eliminate the latency benefit of the sparse ANN backend. At 10K anchors, the grouped configuration achieves a median latency of 2.40 ms, compared with 3.81 ms for indexing every revision independently and 16.29 ms for exhaustive sparse retrieval. At 50K anchors, the grouped and full Seismic configurations have comparable median latency (6.53 ms versus 6.47 ms), while both are substantially faster than exhaustive retrieval.

These results clarify the role of our method relative to modern sparse retrieval systems. Our contribution is not an alternative sparse ANN search algorithm. Instead, revision-aware sparse indexing reduces redundancy in the collection presented to such an algorithm. Seismic provides one concrete instantiation: it searches compact family representatives efficiently, while revision-specific information is recovered during candidate expansion and exact reranking. The same separation is applicable more generally to sparse retrieval engines that accept sparse document representations and return a candidate set for subsequent scoring. In this sense, revision-aware index reuse and sparse ANN retrieval address orthogonal parts of the retrieval stack and can be applied together.

F BROADER IMPACTS

Positive societal impacts. This work studies evolution-aware code retrieval, with the goal of making code search systems more stable and resource-efficient over evolving codebases. By encouraging behaviorally similar code variants to share sparse supports and routing structures, the proposed method can reduce redundant index storage and unnecessary re-indexing when software is renamed, reformatted, or refactored. This may lower the computational, memory, and energy cost of maintaining large-scale code retrieval systems, especially in rapidly changing repositories used by developer tools and AI coding assistants. More stable retrieval under evolution may also improve developer productivity by helping users locate relevant code even when its surface form has changed, and may make high-quality code search more accessible to organizations with limited compute or storage resources.

Negative societal impacts and mitigation strategies. Although this paper is primarily foundational and is not tied to a particular deployment, improved code retrieval can have potential negative uses when integrated into real software-engineering systems. More efficient and robust code search could help unauthorized users locate sensitive implementation details, security-critical code, license-restricted code, or vulnerable functions if applied to repositories without proper access control. Incorrect retrieval or overly aggressive sharing of index structures could also cause harm even in intended use cases: for example, a developer may miss a relevant updated implementation, retrieve an obsolete or unsafe variant, or rely on misleading candidates in security- or reliability-critical settings. In addition, the current benchmark is restricted to Python and controlled evolution sources, so systems trained or evaluated in this setting may underperform for other languages, coding styles, or developer communities. Possible mitigations include deploying such retrieval systems only within authorized codebases, enforcing repository-level access control and audit logging, avoiding exposure of raw proprietary code through indexes, evaluating retrieval behavior across languages and project types before deployment, and using confidence checks or fallback re-indexing when evolution consistency is uncertain. For high-stakes software contexts, retrieved results should remain assistive rather than automatically acted upon, with human review and security testing before changes are adopted.