EVOLVING SEMANTIC INDEXES FOR GENERATIVE RETRIEVAL

Anonymous authors
Paper under double-blind review

ABSTRACT

Generative retrieval (GR) casts search and recommendation as autoregressive generation of discrete semantic identifiers (SID). SIDs play one of the most important roles to determine the quality of GR results. For example, different SID models, collision handling mechanisms, decoding frameworks, and training recipes, can significantly vary the retrieval performance. However, these discrete and interdependent design choices cannot be optimized end-to-end using gradients from downstream retrieval and therefore remain largely designed by hand. In this paper, we introduce a framework for automatically improving these components through LLM-driven evolutionary search on the SID framework, which to the best of our knowledge is the first application of LLM-driven evolutionary program search to GR. A locally servable, open-weight 31B language model is used to mutate executable GR components, so no proprietary LLM API is used at any point, and a 1M-parameter proxy decoder provides fast downstream feedback in place of repeatedly training the 1B target generator, which would otherwise make evolutionary search computationally infeasible. Across 21 SID programs, proxy and target Recall@100 exhibit strong rank correlation ($\rho = 0.90$). Guided by this proxy, evolution consistently improves the seed GR on NQ320K, MSMARCO300K, and a multilingual e-commerce dataset, under a training recipe held fixed across conditions. On NQ320K, jointly evolving SID and collision handling improves Recall@1 from 55.8% to 59.3% and Recall@100 from 85.8% to 90.6%. Our component study reveals that proxy reliability depends on what component is evolved. Improvements transfer reliably when evolving the semantic index, but transfer less reliably when evolving the generator's training objective or the full GR pipeline. These results suggest that LLM-driven evolutionary search is a practical method for improving GR design and show that carefully assessing proxy reliability is essential for successful GR pipeline optimization.

1 INTRODUCTION

Generative retrieval (GR) casts document retrieval as an autoregressive generation task. Given a query, the model is trained to generate the document identifier which maps back to a document (Tay et al., 2022; Wang et al., 2022). In GR systems that are based on semantic identifiers (SIDs), documents are represented by short sequences of discrete tokens, often constructed from their semantic embeddings (Rajput et al., 2023; Ju et al., 2025). These identifiers define the prediction problem for the generator. During autoregressive decoding, documents that share a SID prefix share early generation decisions, which means a relevant document will become unreachable if a SID token in its prefix is discarded during beam search. SID design therefore determines both what the GR model must learn, and how it can reach relevant documents during decoding (Zeng et al., 2023). Improving a generative retriever thus requires careful design of SID generation. This design could involve several interdependent procedures: (1) an index builder assigns SIDs to documents; (2) collision handling distinguishes documents that were assigned the same SID; (3) a training objective teaches the generator to predict these identifiers; (4) generative decoding searches over valid SID sequences. Each procedure affects how the others operate. For example, changing the SID structure reforms the prediction targets, while resolving collisions may induce longer SID sequences for the generator to learn. Existing methods have shown that SID and generative retrieval models can be learned jointly within predefined architectures (Sun et al., 2023; Yang et al., 2023). How-

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.

ever, ordinary backpropagation does not directly choose among different implementations of SID construction, collision handling, training, and decoding. Optimizing these executable procedures presents a broader, discrete search problem. To the best of our knowledge no prior work has applied LLM-driven evolutionary program search to generative retrieval, and this paper is the first to show that doing so improves a GR pipeline.

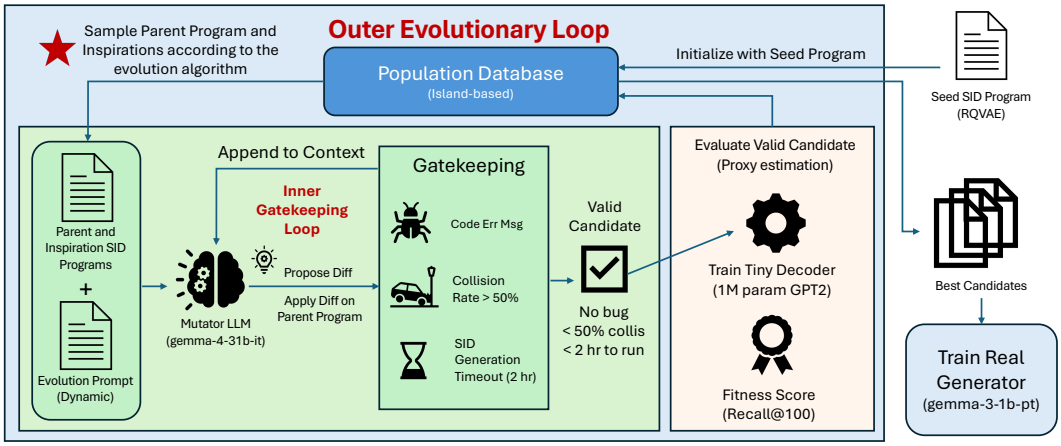


Figure 1: Our two-tiered evolutionary framework. Outer loop manages the population database, selects parent programs, and constructs feedback for the mutator LLM. Inner loop gatekeeps and refines candidate programs, approximating fitness with a tiny decoder.

LLM-driven evolutionary search offers a way to explore this search space by proposing code mutations and retaining programs that perform well on an execution-based evaluation (Romera-Paredes et al., 2024; Novikov et al., 2025). Directly applying this approach to generative retrieval faces a substantial evaluation bottleneck. Evaluating each candidate index requires training the downstream generator and measuring its retrieval performance. In our setting, a target generator training run takes five hours on an H200 GPU, whereas training a proxy takes 20 minutes. Using the proxy therefore reduces the per-candidate evaluation time by 15×, making it practical to evaluate many more candidate programs within a fixed computational budget. The central challenge is therefore to gain inexpensive feedback that remains informative for guiding evolutionary search. Besides this, recent discovery systems often use frontier models as the mutator (Qu et al., 2026; Virk et al., 2026; Cemri et al., 2026), which is expensive and less transparent. We run the entire search with an open-weight model that fits on a single GPU. We do not claim that a 31B mutator LLM is sufficient for program search in general, only that it was sufficient for evolving GR to outperform the seed.

As shown in Figure 1, in this paper, we introduce a framework for evolving executable generative retrieval components using a small retrieval proxy and a locally served, open-weight mutator. Specifically, we use `gemma-4-31b-it` (Gemma Team, 2026), which can be served locally on a single H200 via vLLM (Kwon et al., 2023). Building on the AdaEvolve search algorithm (Cemri et al., 2026), the mutator proposes `SEARCH/REPLACE` edits to the program under fixed input and output constraints. For SID generation, the program receives document and query embeddings together with training relevance labels and returns an SID for each document. Evolution can change the procedure implementing this mapping. A tiny autoregressive decoder with approximately 1M non-embedding parameters is trained and tested on the training dataset only, to provide proxy retrieval performance feedback for candidate program evaluation. Programs are selected and evolved using proxy feedback alone, and in the end the best candidate is fully evaluated by training the `google/gemma-3-1b-pt` (Gemma Team, 2025) target generator on the training dataset, and measuring the retrieval performance on the reserved test set from its final checkpoint. Across NQ320K (Kwiatkowski et al., 2019), MSMARCO300K (Bajaj et al., 2016) and a multilingual e-commerce dataset, the selected evolved SID programs improve their corresponding seed programs when the SIDs they produce are used to train the target 1B generator. On NQ320K, jointly evolving SID construction and collision handling improves Recall@1 from 55.8% to 59.3% and Recall@100 from 85.8% to 90.6%. Our component study also identifies a boundary of proxy-guided search. Improvements transfer more consistently for the index-construction programs, whereas changes to

the generator's training objective or the full GR pipeline, can improve proxy performance while reducing target performance. This distinction suggests that the usefulness of a small proxy evaluator depends on the artifacts being optimized. A shared SID structure can benefit both proxy and target generators, while a training objective that the proxy benefits from may not be suitable for the much larger target decoder. We state the scope of our claims plainly. We do not claim state-of-the-art GR, nor that GR dominates lexical or dense retrieval. Our contributions are as follows:

- We introduce a framework for evolving executable generative retrieval procedures with a tiny proxy decoder and a locally served 31B language model mutator, the first such framework for generative retrieval and one that uses no proprietary LLM API.
- We demonstrate retrieval improvements over matched seed systems across three retrieval datasets, observing substantial gains from jointly evolving SID construction and collision handling.
- We show that proxy-to-target transfer depends on the component being evolved, with semantic index improvements transferring more consistently than decoder training objective changes.

2 Generative Retrieval

Generative retrieval maps a query to a document by generating its identifier (Tay et al., 2022; Wang et al., 2022). Let $\mathbb{D}$ denote the document corpus. Each document $d \in \mathbb{D}$ is assigned a semantic identifier $\boldsymbol{z}(d) = (z_1(d), \ldots, z_{L(d)}(d))$, where $L(d)$ is the number of tokens in its identifier. Given query text q, an autoregressive generator with parameters $\boldsymbol{\theta}$ models

$$P_{\boldsymbol{\theta}}(\boldsymbol{z}(d) \mid q) = \prod_{\ell=1}^{L(d)} P_{\boldsymbol{\theta}}\big(z_\ell(d) \mid q, \boldsymbol{z}_{<\ell}(d)\big), \tag{1}$$

where $\boldsymbol{z}_{<\ell}(d)$ denotes the preceding identifier tokens. We train the generator on relevant query-SID pairs using teacher-forced next-token cross-entropy loss.

Inference traces a prefix trie of the valid SID sequences. Constrained beam search only permits children trie nodes as next tokens. In the end, the completed SIDs are mapped back to documents. Documents sharing the same SID prefix will share early decoding decisions, which means early pruning will remove every document beneath it from the remaining search. The SID index therefore determines both the prediction targets during training and the branching structure during retrieval.

2.1 Semantic Indexing

Our seed program combines RQVAE-based SID construction with TIGER-style collision handling (Lee et al., 2022; Rajput et al., 2023). We first encode document text using a fixed Contriever or mContriever encoder Izacard et al. (2022). These embeddings provide the input to the index builder. The generative retriever will only receive query text.

SID construction. The seed constructs SIDs using residual quantization. Let $\boldsymbol{x}(d)$ be the fixed embedding of document d and $\boldsymbol{r}_0(d)$ its latent representation produced by the RQVAE encoder. At each level ℓ, the quantizer selects a codebook vector and subtracts it from the current residual:

$$z_\ell(d) = \arg\min_k \|\boldsymbol{r}_{\ell-1}(d) - \boldsymbol{e}_{\ell,k}\|_2^2, \boldsymbol{r}_\ell(d) = \boldsymbol{r}_{\ell-1}(d) - \boldsymbol{e}_{\ell,z_\ell(d)}, \tag{2}$$

where $\boldsymbol{e}_{\ell,k}$ is the k-th vector in the codebook at level ℓ. After M levels, the selected indices $(z_1(d), \ldots, z_M(d))$ form the semantic part of the identifier. The RQVAE is trained with reconstruction and quantization losses to preserve information from the document embeddings following (Lee et al., 2022). These losses do not directly optimize downstream retrieval performance.

Collision handling. It is common for SID construction to assign the same SID sequence to multiple documents. Following TIGER (Rajput et al., 2023), the seed program adds a unique disambiguation token $z_{M+1}(d)$ within each group of documents that were assigned the same SID sequence:

$$\boldsymbol{z}(d) = \big(z_1(d), \ldots, z_M(d), z_{M+1}(d)\big), \qquad L(d) = M + 1. \tag{3}$$

SID construction and collision handling define the generator prediction targets. Semantic tokens determine which documents share a prefix, while the final token distinguishes documents with identical semantic codes. Because changing the quantizer also changes the collision groups, we study both evolving SID construction with fixed collision handling and evolving the two components jointly.

3 PROXY-GUIDED EVOLUTIONARY SEARCH

We mainly evolve the program that constructs SIDs and use a small generative retriever to evaluate the SIDs it produces. Each candidate program is evaluated by executing the program to obtain document to SID mappings, training the small generator on the relevant query to SID pairs. and measuring its retrieval performance. This feedback serves as a proxy that guides evolutionary program search without training the full 1B target generator at every step.

Executable search space. When evolving semantic indexes, the candidate program receives fixed corpus document embeddings, training query embeddings, and training relevance pairs. The program returns an integer array containing one SID per document, aligned to the input documents. The input and output interface is fixed, while the procedure implementing this mapping is mutable. For example, evolution can change the quantization process, codebook initialization, or even switching to entirely different methods. The input embeddings, and relevance labels are fixed, and are from the training data. The mutator LLM only has access to the SID generation code and performance feedbacks. The mutation prompts (Appendix B) encourage creativity but also include suggested algorithm families, such as alternative quantization and clustering methods. Testing data is entirely unreachable to the mutator LLM.

For SID-only search, collision handling is held fixed. For joint SID and collision handling search, the mutable program determines how documents with identical SIDs are distinguished. We also study broader search spaces for changes to the training objective of generative retrievers, generator decoding logic, and collision-handling-only changes. Full pipeline search is also studied, where SID construction, collision handling, generator training, and decoding can all be changed. These scopes allow us to examine how the choice of mutable component affects proxy-guided search.

Proxy generative retriever. While SID collision rate and per-codebook perplexity can describe desired properties of SIDs, they do not directly measure the SIDs' usefulness for retrieval. Across 21 evaluated NQ320K SID programs, the collision rate logged during evolution is negatively associated with final target Recall@100 with a Spearman $\rho = -0.855$. Although this makes collision rate a usable signal, it cannot serve as the fitness function as it is too easy to hack. For example, random assignments can produce few collisions and balance code usage without providing meaningful semantic structure for the downstream generator. We therefore evaluate candidate programs through downstream retrieval, using a small generator to assess how well their identifiers can be learned and retrieved. Our proxy is a two-layer GPT-2 Radford et al. (2019) model with approximately 1M non-embedding parameters (7M if including token embeddings). Like the target 1B generator, it receives query text and predicts relevant document's SIDs using constrained decoding.

To keep search blind to test data, we randomly split each dataset's original training relevance pairs into $\mathcal{R}_{\text{proxy}}$ (two thirds) and $\mathcal{R}_{\text{feedback}}$ (one third). The split is over query document pairs, so a query with several relevant documents can contribute pairs to both sides. During search, both the candidate program and the proxy retriever $g_{\text{P}}(c)$ see only $\mathcal{R}_{\text{proxy}}$, and the candidate's fitness is

$$F_{\text{P}}(c) = \text{Recall@100}\big(g_{\text{P}}(c); \mathcal{R}_{\text{feedback}}\big). \tag{4}$$

The proxy measures how well a small generator can learn and retrieve the candidate's produced SIDs. Its scores do not need to match the target generator, but must provide useful preferences among programs in the search space. Whether these preferences transfer to the target generator is an empirical question which we further explore in Section 5.2.

Program mutation and execution. As shown in Figure 1, we use AdaEvolve (Cemri et al., 2026) to maintain an island-based population of programs and control exploration and exploitation. At each outer step, the search controller selects a parent program and additional programs to provide context for mutation. We use `gemma-4-31b-it` (Gemma Team, 2026), served locally with vLLM (Kwon et al., 2023) on a single H200, to propose `SEARCH/REPLACE` edits within the marked mutable regions of the parent program. The proposed edits are applied to the parent, and the resulting candidate program is executed. We check that the candidate executes successfully and passes the collision gate before training the proxy. For SID-only search, we reject candidates with a collision rate above 50%, and if a candidate is accepted, we apply TIGER-style collision handling. Candidate execution is limited to two hours. When a proposal fails, the mutator receives the execution error, collision diagnostics, or timeout message and is asked to revise its edits to the parent. We allow up to 20 retries before abandoning the proposal and moving to the next outer step.

4 EXPERIMENTAL SETUP

Our experiments examine whether using a smaller, cheaper model to find good programs can boost the performance of a final 1B-parameter model. We measured how well this shortcut works across different components, compared the new setups against a baseline, and evaluated both the program quality and the compute costs involved.

Datasets and Baselines. We evaluate on NQ320K (Kwiatkowski et al., 2019), MSMARCO300K (Bajaj et al., 2016), which are both reduced collections of the original datasets, following prior work (Tay et al., 2022; Zhou et al., 2024; 2023; Mekonnen et al., 2025). We also evaluate on a proprietary multi-lingual dataset from a popular e-commerce company. The NQ320K collection contains 109,739 documents and 7,830 evaluation queries. The MSMARCO300K subset contains 319,927 documents and 808 evaluation queries. The e-commerce corpus contains millions of distinct purchased items collected over one month. We use two random seeds to downsample the purchase queries into two equal-sized training sets. Both are tested on the 25,000 purchase queries from the following day after the month. The queries are stratified sampled by purchase frequency. Items are represented by title, description and brand, with purchases as query-item relevance labels.

We use fixed Contriever (Izacard et al., 2022) embeddings for NQ and MSMARCO and mContriever embeddings for e-commerce because the majority of it is not in English. Keeping the encoder fixed makes the seed and evolved SID programs operate on the same input representations, which isolates the effect of SID structure. In Table 1, the only difference between Seed GR and SID evolution rows is the semantic index. The SID construction program receives document and query embeddings together with training relevance pairs, and the generative retriever receives query text, the resulting SIDs, and relevance pairs. Search uses only the original training relevance pairs, split as in Section 3, with no synthetic or test queries. For final training, the best program by proxy fitness is rerun on all original training pairs to produce SIDs, and the target generator is trained on the same pairs. For NQ and MSMARCO only, both steps also include one synthetic query per corpus document generated by `gemma-4-31b-it` (Gemma Team, 2026), following prior work (Zhou et al., 2022; 2023; Zhang et al., 2024; Mekonnen et al., 2025) that typically generates ten per document with a weaker generator. E-commerce uses no synthetic queries because purchase data is plentiful. All reported numbers are on the held-out test queries, which neither stage ever sees. Our primary comparison is against the generative retrieval pipeline using the seed program (Seed GR), which uses RQVAE and fixed collision handling described in Section 2.1. We also report BM25 and Contriever performance as reference rather than competitors.

Program Search and Evaluation Configuration. The seed program uses three residual codebooks with 256 entries each and a 256-dimensional latent representation. We train the seed RQVAE (Lee et al., 2022; Rajput et al., 2023) for 100 epochs using AdamW with learning rate 10^{-4}, batch size 2,048, weight decay 0.01, and commitment coefficient 0.01. The proxy is a two-layer GPT-2 decoder with approximately 1M non-embedding parameters, or 7M if including token embeddings. AdaEvolve (Cemri et al., 2026) maintains the program population with 3 islands, and `gemma-4-31b-it` proposes mutations through vLLM (Kwon et al., 2023) on one H200. Candidate execution is limited to two hours, with up to 20 inner-loop retries. SID-only candidate must pass a 50% collision rate gate before fixed collision is applied. Joint SID and collision handling candidates must pass a 5% gate on their returned SIDs. After each evolution step, the proxy GPT-2 is trained with cross-entropy loss on SID tokens and EOS for 10 epochs using AdamW with learning rate 5×10^{-4}, batch size 512, weight decay 0.01.

We study several evolution regimes: *SID-only* evolves SID construction while retaining TIGER collision handling, whereas *joint SID-collision* updates both simultaneously. On NQ, we additionally evaluate *collision evolution* (fixing the seed SID program), *sequential SID-collision* (evolving collision handling after fixing an evolved SID constructor), *objective evolution* (modifying the downstream generator's loss, distinct from the SID constructor's loss), and *full-pipeline evolution* (spanning SID construction, collision handling, generator training, and decoding).

We evaluate the best programs from each evolution run with `gemma-3-1b-pt` (Gemma Team, 2025), which we refer to as the "target" generator. We use the pretrained rather than the instruction-tuned checkpoint because it carries general pretraining knowledge without having been adapted with any downstream task, which makes it a clean base for learning the GR objective. It is always trained with AdamW optimizer for 3 epochs with learning rate 5×10^{-5}, batch size 32, 0.01 weight decay,

Table 1: **Overall results.** "SID evolution" reports the target's performance when trained on SIDs generated by the best SID program found through evolutionary search, where the best program is selected using the proxy's Recall@100. Values with ± are means and standard deviations across three independent training runs. Bold indicates the highest mean for each metric within a dataset.

Dataset	Method	R@1 (%)	R@5 (%)	R@10 (%)	R@100 (%)	MRR@10 (%)
NQ320K	BM25	32.08	54.51	62.35	83.69	41.62
	Contriever	36.18	62.45	71.37	89.41	47.36
	Seed GR	55.82 ± 0.04	71.12 ± 0.53	75.27 ± 0.44	85.82 ± 0.22	62.50 ± 0.19
	SID evolution	$\mathbf{59.28 \pm 0.06}$	$\mathbf{76.44 \pm 0.18}$	$\mathbf{81.05 \pm 0.09}$	$\mathbf{90.57 \pm 0.20}$	$\mathbf{66.74 \pm 0.08}$
MSMARCO300K	BM25	18.94	**47.52**	**60.64**	**86.63**	**31.32**
	Contriever	12.38	30.82	41.58	73.64	20.56
	Seed GR	19.43 ± 0.18	40.97 ± 0.35	51.49 ± 0.88	79.70 ± 0.18	28.61 ± 0.11
	SID evolution	$\mathbf{20.48 \pm 0.44}$	41.96 ± 1.05	55.01 ± 0.26	82.55 ± 1.40	30.09 ± 0.06
E-commerce	BM25	2.73 ± 0.00	6.43 ± 0.01	8.50 ± 0.01	17.82 ± 0.10	4.33 ± 0.02
	mContriever	9.34 ± 0.18	20.49 ± 0.12	25.89 ± 0.31	46.95 ± 0.14	14.06 ± 0.06
	Seed GR	10.79 ± 0.60	26.79 ± 1.00	35.68 ± 0.84	66.83 ± 0.45	17.70 ± 0.73
	SID evolution	$\mathbf{11.29 \pm 0.68}$	$\mathbf{27.83 \pm 0.85}$	$\mathbf{36.97 \pm 0.74}$	$\mathbf{68.48 \pm 0.47}$	$\mathbf{18.42 \pm 0.75}$

and 0.05 warmup fraction. On an H200 GPU, a proxy run takes around 20 minutes, compared with five hours for the target generator. For 100 candidate evaluations, these runtimes correspond to 33 hours of proxy computation versus 400 to 500 hours of true target evaluation. LLM mutation, SID construction, and failed candidate executions incur additional costs. We report target generator's Recall@1, Recall@10, Recall@100, and MRR@10, all as percentages.

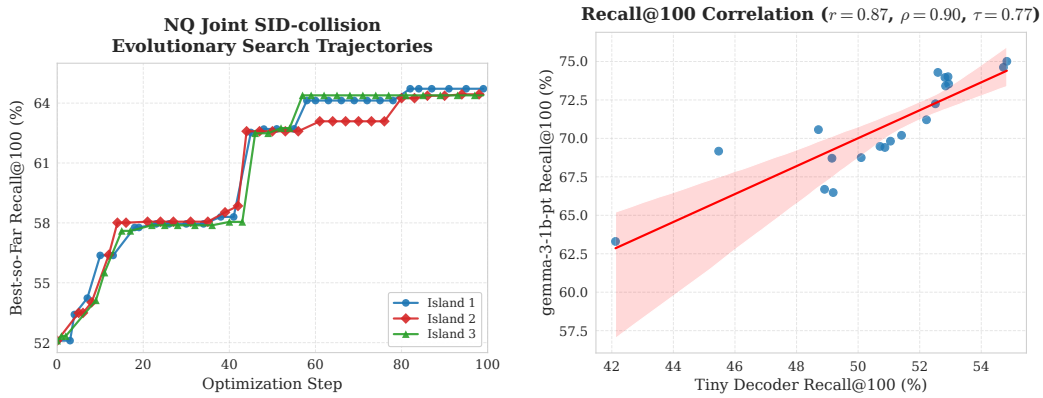
5 Results

5.1 Evolved Semantic Indexes

Table 1 shows improvements over Seed GR on all three datasets. SID evolution increases Recall@100 from 85.82% to 90.57% on NQ, from 79.70% to 82.55% on MSMARCO, and from 66.83% to 68.48% on e-commerce. Recall@1 and MRR@10 also improve notably in each comparison, showing that although evolutionary search optimizes Recall@100, its benefits also extend to early ranking quality, rather than only retrieving additional relevant documents near the end of the list. For NQ and MSMARCO, the best SID evolution program comes from jointly evolve SID and collision handling. For e-commerce, it comes from evolving SID only. For NQ, while the seed underperforms Contriever on Recall@100, a different SID structure is sufficient to improve the final generator to outperform Contriever, while everything else from data to the target generator and training recipe remains the same. On MSMARCO, evolved GR exceeds BM25 at rank one, but BM25 remains stronger at the larger recall ranks and in MRR@10. We do not claim that the evolved GR pipeline achieves state-of-the-art, nor that GR dominates lexical or dense retrieval, but instead we argue evolutionary search on the semantic index can reliably improve a well-established GR pipeline whoes every other component is held fixed. While improving the semantic index improves retrieval performance, it does not eliminate the limitations of the general GR pipeline. On e-commerce dataset where relevance is defined as purchase instead of whether the document contains answers to the query like in NQ and MSMARCO, we see GR outperforms BM25 significantly. This is expected because the purchased product does not necessarily contain the query terms in their title, brand, or item description. Also search queries are often ambiguous and short, making lexical retrieval hard to compete with dense retrieval and generative retrieval.

5.2 Proxy Transfer Depends on What is Evolved

Figure 2(a) shows the optimization trajectory of the run that produced the best Joint SID-collision program for NQ. The LLM-driven evolutionary search is able to drive the defined fitness score. If the evaluation performed after each step is exactly faithful to the real task, then the optimized solution would have been the actual best solution. However, often due to computation limitations, it would be too expensive or even computationally infeasible for the evaluator to be exactly what the real task is. For example in our case, the true evaluation is training the target `gemma3-1b-pt` on the SIDs constructed by the newly mutated SID program, but that would take five hours and an



(a) Three-island search progression

(b) Proxy–target correlation

Figure 2: (a) Joint SID-collision search on NQ with three islands: island-based evolution is a main feature of AdaEvolve. Each curve connects the recorded best-so-far proxy Recall@100 values for one island. The x-axis is the global evolution step. (b) A separate correlation experiment over 21 joint SID-collision programs pairs proxy and target Recall@100, with Spearman $\rho = 0.90$. The red shaded regions represent 95% confidence interval.

Table 2: **Transfer across search scopes.** Selected NQ programs, grouped by whether the generator objective can change. Scores are percentages; Δ is target R@100 relative to Seed GR, in percentage points. Proxy values are single search evaluations; target values summarize three runs, except full-pipeline evolution (two). Objective and full-pipeline searches start at proxy R@100 of 52.10% Searches have different initializations and completed budgets, so this compares observed outcomes rather than isolated component effects.

Search scope	Proxy R@100 (%)	Target R@1 (%)	Target R@100 (%)	Δ
Seed GR	52.31	55.82 ± 0.04	85.82 ± 0.22	—
Evolving the generator training objective				
Objective evolution	53.80	53.51 ± 0.38	84.46 ± 0.33	-1.36
Full-pipeline evolution	62.85	53.85 ± 0.50	76.12 ± 0.21	-9.69
Evolving the semantic index (generator objective unchanged)				
SID-only evolution	61.47	56.33 ± 0.40	87.40 ± 0.20	$+1.58$
Collision evolution	52.69	55.79 ± 0.10	85.72 ± 0.37	-0.10
Sequential SID–collision	61.30	55.96 ± 0.10	87.25 ± 0.21	$+1.43$
Joint SID–collision	**64.39**	$\mathbf{59.28 \pm 0.06}$	$\mathbf{90.57 \pm 0.20}$	$\mathbf{+4.75}$

evolutionary search that spans 100 steps would cost nearly 500 hours, which translates to about 21 days. Not to mention that this is on the dataset size that we presented in this study, in e-commerce production settings where we train the generator on one year worth of purchase data, training the generator once can take weeks. This is also the main motivation for proxy evaluator. Because the need for a light weight proxy, the effectiveness of the evolved program is only as reliable as the correlation between the proxy evaluation and the real downstream task. Figure 2(b) examines 21 programs from a separate joint SID-collision calibration experiment, where we randomly sample 20 different candidates programs throughout the optimization and with the seed, makes it 21. We train both the 1M parameter proxy and the 1B parameter target generator on the SIDs constructed from these 21 candidates on NQ without the synthetic queries. Proxy and target Recall@100 have strong positive rank correlation (Spearman $\rho = 0.90$). The proxy need not reproduce the target's absolute score to be useful. As long as it favors semantic index changes that also help the target generator, the evolutionary search approach can be effective. Selection under the proxy is also near-optimal in hindsight. Among the fully evaluated NQ programs, the highest-proxy program attains a target Recall@100 of 90.43% against a best available 90.87%.

The contrast in Table 2 shows why this evidence cannot be extended to every mutable component. Once the generator's training objective becomes mutable, the proxy does not only become less informative, it is infact negatively correlated with the target performance. Objective evolution in-

creases proxy Recall@100 from its initialization of 52.10% to 53.80%, but target Recall@100 falls to 84.46%. Full-pipeline evolution also mutates the generator's objective. It reaches a much higher proxy score of 62.85% while reducing target Recall@100 to 76.12%. This transfer failure extends beyond the selected examples in Table 2. Across five distinct evolved programs (three objective only and two full pipeline programs), proxy Recall@100 and final target Recall@100 have a negative rank correlation (Spearman $\rho = -0.80$, $n = 5$). All five programs improve proxy fitness over initialization while regressing on target recall compared to Seed GR. The main distinction is what the generators learn versus how they learn it. Semantic index evolution changes the SID structures shared by both generators. Objective evolution changes the optimization signal, whose effect can depend on model architecture, pretraining, and training recipe. A highly optimized objective for the 1M parameter proxy does not necessarily transfer to the 1B parameter target. The full-pipeline evolution is an attempt to balance evolving objective and semantic index, but has shown no positive results. We analyze the potential cause, but our presented results do not explain the cause. Our finding is that proxy transfer depends on the allowed search region.

As shown in Table 2, SID-only evolution improves target retrieval, while collision-only evolution leaves it close to the seed at 85.72% versus 85.82% Recall@100. Joint evolution provides the largest gain. Evolving collision handling after fixing an evolved SID constructor reaches 87.25%, with no additional mean improvement over SID-only evolution at 87.40%. Joint evolution reaches 90.57%, exceeding the sequential result by 3.32 points. This pattern is consistent with a dependence between the two procedures. SID construction determines which documents share an identifier and thus the groups within which collision handling must distinguish documents. Holding that partition fixed restricts the collision procedure's choices, while joint search can change both the groups and the suffix targets. The NQ results suggest value in this coordinated design, although different search budgets, gates, and selected programs prevent attributing the gain to coordination alone.

6 The Evolved SID program

We examine the joint SID and collision handling program evolved on NQ with proxy Recall@100 of 64.39% and target Recall@100 increases from 85.82% to 90.57%. The program retains the classic RQVAE structure and four-token SID in Equation 3, with $M = 3$ semantic tokens and one collision suffix. Its changes mainly revolve around how the semantic codes are assigned and trained, and how the suffix is constructed.

Direction-based residual quantization. The evolved program selects the code with maximum cosine similarity to the current residual, replacing squared-Euclidean assignment in Equation 2. It still subtracts the selected, unnormalized code vector. Codebook vector lengths therefore affect the residual passed to the next level, even though selection depends on direction. This is used during both index training and final SID construction. The program also doubles each codebook from 256 to 512 entries while retaining the 256-dimensional latent representation.

Training the index with query information. The seed program learns the index using only document reconstruction. The evolved program adds query-based supervision by encouraging each reconstructed document embedding to align with the relevant training queries. Specifically, for each document d, the embeddings of its associated training queries are averaged and ℓ_2-normalize to form a query target $\boldsymbol{t}(d)$. If d has no associated training query then $\boldsymbol{t}(d) = \mathbf{0}$. Let $\hat{\boldsymbol{x}}(d)$ denote the reconstructed document embedding obtained by summing its selected code vectors, and let $\mathcal{N}$ denote ℓ_2 normalization. For a document minibatch $\mathcal{B}$, the evolved program adds the alignment loss

$$\mathcal{L}_{\text{align}} = \frac{1}{|\mathcal{B}|} \sum_{d \in \mathcal{B}} \left[1 - \langle \mathcal{N}(\hat{\boldsymbol{x}}(d)), \boldsymbol{t}(d) \rangle\right]. \tag{5}$$

This loss encourages each reconstructed document embedding to point in the same direction as the average query embedding associated with that document. It is weighted by 0.8 and optimized together with the reconstruction, quantization, and codebook regularization losses. This idea resonates with existing work Zhang et al. (2025); Zeng et al. (2024) that semantic indexing should be aware of the downstream task, but the formulations are different.

Geometry-based collision handling. The seed assigns disambiguation tokens to documents with identical SID by document ID order. The evolved program instead averages the normalized document embeddings within each collision group and ranks members by decreasing cosine similarity to

that centroid. Small suffix values now represent relatively central documents across different groups, giving the suffix a shared interpretation that the original TIGER-style collision handling lacks.

6.1 Other Evolved Programs

The best NQ SID program introduces relevance supervision to the reconstruction. Other evolved programs also leverage the query and relevance, but in very different formulations. Some MS-MARCO SID programs train document and query projections on relevance pairs, then freezes the document encoder and fits residual K-means. Some NQ programs remove neural index training entirely. One example mixes normalized document embeddings with the mean of normalized relevant query embeddings, with weights 0.7 for the document and 0.3 for the associated query, then applies residual K-means. Related programs from the e-commerce datasets incorporate query embeddings into centroid updates. Relevance information can therefore shape the vectors or the clustering procedure without a reconstruction decoder. Every program above was proposed by a locally servable 31B model, which establishes that evolutionary search over GR components can be carried out without a proprietary LLM API. We leave the use of state-of-the-art proprietary LLMs as future work.

7 Related Work

Generative Retrieval. DSI and NCI were the first to formulate retrieval as document identifier generation (Tay et al., 2022; Wang et al., 2022), since then the generative retrieval community has tackled and improved many aspects of the pipeline. TIGER (Rajput et al., 2023) constructs SIDs through residual quantization. GenRet and ASI (Sun et al., 2023; Yang et al., 2023) jointly learn indexing and retrieval. RIPOR and MERGE (Zeng et al., 2024; Zhang et al., 2025) incorporate relevance supervision into identifier construction. DDRO directly optimizes document relevance (Mekonnen et al., 2025). GLEN (Lee et al., 2023) addresses SID collisions, and TSGen (Zhang et al., 2024) introduces term-set identifiers with permutation-invariant decoding. Our work focuses on making automatic semantic index evolution feasible and shows that index-related improvements transfer more consistently than changes to the generator's training objective.

Evolutionary program search and proxy evaluation. FunSearch and AlphaEvolve use a coding LLM to propose code edits to candidate programs, managed by an island-based evolutionary search algorithm Romera-Paredes et al. (2024); Novikov et al. (2025). Subsequent work improves search through adaptive population management in AdaEvolve (Cemri et al., 2026), evolving search strategies in EvoX (Liu et al., 2026), and coordinated multi-agent discovery in CORAL (Qu et al., 2026) and SwarmResearch (Virk et al., 2026). Proxy evaluation also has established precedents such as AutoML-Zero (Real et al., 2020), which searches learning algorithms on proxy tasks. Lion (Chen et al., 2023) addresses generalization from small search tasks to larger workloads. ShinkaEvolve (Lange et al., 2025) demonstrates transfer of evolved training losses to larger models.

8 Conclusion

In this paper, we presented a framework for improving generative retrieval through evolutionary search over executable components, using a locally served 31B LLM as mutator and a tiny 1M parameter retrieval proxy for candidate evaluation, in order to make the evolution practical. Across three retrieval datasets, proxy-selected semantic index programs improve the overall generative retrieval pipeline on the target 1B generator, demonstrating the value of automatically refining SID construction and collision handling. Our component study reveals that evolve-time improvements transfer more consistently for semantic indexes than changes to the generator training objective. The effectiveness of proxy-guided evolutionary search therefore depends on what is being optimized, and correlation with the target system must be established for the chosen search space. These findings support semantic index evolution as a practical direction for automated generative retrieval system design and motivate further study of evaluators that remain informative for different model scales and generative retrieval components.

AI USE STATEMENT

We used `gemma-4-31b-it` to propose program mutations and generate synthetic training queries, as described in the paper. Codex assisted with literature review, interpretation and presentation of results, and manuscript polishing. Candidate programs underwent execution checks and empirical evaluation. The authors take responsibility for the final manuscript, citations, methods, and reported results.

ETHICS STATEMENT

This study uses public retrieval benchmarks and proprietary, anonymous e-commerce purchase data. Purchase-based relevance may reflect existing exposure and popularity biases. Improvements in retrieval metrics do not establish fairness guarantees.

REPRODUCIBILITY STATEMENT

Sections 2.1 and 3 describe the seed index, search procedure, and proxy evaluation. Section 4 documents datasets, model checkpoints, training configurations, and evaluation protocols, while Section 6 details the selected evolved program. Independent reproduction of the e-commerce experiments requires access to the proprietary dataset, which will not be publicly available.

REFERENCES

Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, Mir Rosenberg, Xia Song, Alina Stoica, Saurabh Tiwary, and Tong Wang. MS MARCO: A human generated MAchine Reading COmprehension dataset. *arXiv preprint arXiv:1611.09268*, 2016. doi: 10.48550/arXiv.1611.09268. URL https://arxiv.org/abs/1611.09268.

Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, Alex Dimakis, and Ion Stoica. Adaevolve: Adaptive LLM driven zeroth-order optimization. *CoRR*, abs/2602.20133, 2026. doi: 10.48550/ARXIV.2602.20133. URL https://doi.org/10.48550/arXiv.2602.20133.

Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. Symbolic discovery of optimization algorithms. In *Advances in Neural Information Processing Systems*, volume 36, pp. 49205–49233, 2023. doi: 10.52202/075280-2140. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/9a39b4925e35cf447ccba8757137d84f-Abstract.html.

Gemma Team. Gemma 3 technical report, 2025. URL https://arxiv.org/abs/2503.19786.

Gemma Team. Gemma 4 technical report, 2026. URL https://arxiv.org/abs/2607.02770.

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. Unsupervised dense information retrieval with contrastive learning. *Transactions on Machine Learning Research*, 2022. ISSN 2835-8856. URL https://openreview.net/forum?id=jKN1pXi7b0.

Clark Mingxuan Ju, Liam Collins, Leonardo Neves, Bhuvesh Kumar, Louis Yufeng Wang, Tong Zhao, and Neil Shah. Generative recommendation with semantic ids: A practitioner's handbook. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management (CIKM)*, 2025.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the*

Association for Computational Linguistics, 7:452–466, 2019. doi: 10.1162/tacl_a_00276. URL https://aclanthology.org/Q19-1026/.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pp. 611–626, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution, 2025. URL https://arxiv.org/abs/2509.19349.

Doyup Lee, Chiheon Kim, Saehoon Kim, Minsu Cho, and Wook-Shin Han. Autoregressive image generation using residual quantization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11523–11532, 2022. URL https://openaccess.thecvf.com/content/CVPR2022/html/Lee_Autoregressive_Image_Generation_Using_Residual_Quantization_CVPR_2022_paper.html.

Sunkyung Lee, Minjin Choi, and Jongwuk Lee. GLEN: Generative retrieval via lexical index learning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7693–7704. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.477. URL https://aclanthology.org/2023.emnlp-main.477/.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z. Pan, Alexander Du, Kurt Keutzer, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. Evox: Meta-evolution for automated discovery. *CoRR*, abs/2602.23413, 2026. doi: 10.48550/ARXIV.2602.23413. URL https://doi.org/10.48550/arXiv.2602.23413.

Kidist Amde Mekonnen, Yubao Tang, and Maarten de Rijke. Lightweight and direct document relevance optimization for generative information retrieval. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '25, pp. 1327–1338, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400715921. doi: 10.1145/3726302.3730023. URL https://doi.org/10.1145/3726302.3730023.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Ao Qu, Han Zheng, Zijian Zhou, Yihao Yan, Yihong Tang, Shao Yong Ong, Fenglu Hong, Kaichen Zhou, Chonghe Jiang, Minwei Kong, Jiacheng Zhu, Xuan Jiang, Sirui Li, Cathy Wu, Bryan Kian Hsiang Low, Jinhua Zhao, and Paul Pu Liang. CORAL: Towards autonomous multi-agent evolution for open-ended discovery, 2026. URL https://arxiv.org/abs/2604.01658.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

Shashank Rajput, Nikhil Mehta, Anima Singh, Raghunandan Hulikal Keshavan, Trung Vu, Lukasz Heldt, Lichan Hong, Yi Tay, Vinh Q. Tran, Jonah Samost, Maciej Kula, Ed H. Chi, and Maheswaran Sathiamoorthy. Recommender systems with generative retrieval. In *Advances in Neural Information Processing Systems*, volume 36, pp. 10299–10315, 2023. doi: 10.52202/075280-0452. URL https://proceedings.neurips.cc/paper/2023/hash/20dcab0f14046a5c6b02b61da9f13229-Abstract.html.

Esteban Real, Chen Liang, David So, and Quoc Le. AutoML-Zero: Evolving machine learning algorithms from scratch. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 8007–8019. PMLR, 2020. URL https://proceedings.mlr.press/v119/real20a.html.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nat.*, 625(7995):468–475, 2024. doi: 10.1038/S41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

Weiwei Sun, Lingyong Yan, Zheng Chen, Shuaiqiang Wang, Haichao Zhu, Pengjie Ren, Zhumin Chen, Dawei Yin, Maarten de Rijke, and Zhaochun Ren. Learning to tokenize for generative retrieval. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-2010. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91228b942a4528cdae031c1b68b127e8-Abstract-Conference.html.

Yi Tay, Vinh Tran, Mostafa Dehghani, Jianmo Ni, Dara Bahri, Harsh Mehta, Zhen Qin, Kai Hui, Zhe Zhao, Jai Gupta, Tal Schuster, William W Cohen, and Donald Metzler. Transformer memory as a differentiable search index. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 21831–21843. Curran Associates, Inc., 2022. doi: 10.52202/068431-1587. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/892840a6123b5ec99ebaab8be1530fba-Paper-Conference.pdf.

Yuvraj Virk, Zack Edds, Chunqiu Steven Xia, and Lingming Zhang. SwarmResearch: Orchestrating coding agents for open-ended discovery, 2026. URL https://arxiv.org/abs/2607.02807.

Yujing Wang, Yingyan Hou, Haonan Wang, Ziming Miao, Shibin Wu, Qi Chen, Yuqing Xia, Chengmin Chi, Guoshuai Zhao, Zheng Liu, Xing Xie, Hao Sun, Weiwei Deng, Qi Zhang, and Mao Yang. A neural corpus indexer for document retrieval. In *Advances in Neural Information Processing Systems*, volume 35, pp. 25600–25614, 2022. doi: 10.52202/068431-1856. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/a46156bd3579c3b268108ea6aca71d13-Abstract-Conference.html.

Tianchi Yang, Minghui Song, Zihan Zhang, Haizhen Huang, Weiwei Deng, Feng Sun, and Qi Zhang. Auto search indexer for end-to-end document retrieval. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 6955–6970. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.findings-emnlp.464. URL https://aclanthology.org/2023.findings-emnlp.464/.

Hansi Zeng, Chen Luo, Bowen Jin, Sheikh Muhammad Sarwar, Tianxin Wei, and Hamed Zamani. Scalable and effective generative information retrieval, 2023. URL https://arxiv.org/abs/2311.09134.

Hansi Zeng, Chen Luo, Bowen Jin, Sheikh Muhammad Sarwar, Tianxin Wei, and Hamed Zamani. Scalable and effective generative information retrieval. In *Proceedings of the ACM Web Conference 2024*, WWW '24, pp. 1441–1452, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400701719. doi: 10.1145/3589334.3645477. URL https://doi.org/10.1145/3589334.3645477.

Fuwei Zhang, Xiaoyu Liu, Xinyu Jia, Yingfei Zhang, Shuai Zhang, Xiang Li, Fuzhen Zhuang, Wei Lin, and Zhao Zhang. Multi-level relevance document identifier learning for generative retrieval. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10066–10080, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.497. URL https://aclanthology.org/2025.acl-long.497/.

Peitian Zhang, Zheng Liu, Yujia Zhou, Zhicheng Dou, Fangchao Liu, and Zhao Cao. Generative retrieval via term set generation. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 458–468. Association for Computing Machinery, 2024. doi: 10.1145/3626772.3657797. URL https://doi.org/10.1145/3626772.3657797.

Yujia Zhou, Jing Yao, Zhicheng Dou, Ledell Wu, Peitian Zhang, and Ji-Rong Wen. Ultron: An ultimate retriever on corpus with a model-based indexer, 2022. URL https://arxiv.org/abs/2208.09257.

Yujia Zhou, Zhicheng Dou, and Ji-Rong Wen. Enhancing generative retrieval with reinforcement learning from relevance feedback. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 12481–12490, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.768. URL https://aclanthology.org/2023.emnlp-main.768/.

Yujia Zhou, Jing Yao, Zhicheng Dou, Yiteng Tu, Ledell Wu, Tat-Seng Chua, and Ji-Rong Wen. Roger: Ranking-oriented generative retrieval. *ACM Trans. Inf. Syst.*, 42(6), October 2024. ISSN 1046-8188. doi: 10.1145/3603167. URL https://doi.org/10.1145/3603167.

A Seed and Evolved SID Programs

A.1 Seed RQVAE with Fixed Collision Handling

Program 1. Seed RQVAE with fixed collision handling

```
"""Evolve SID generation with de-collision blended IN-BAND.

The mutator owns ONE function: `generate_sids(query_emb, doc_emb, qrel_pairs,
   doc_ids, *, device)`.
Its return is `(n_docs, L) int32` with **every row unique** -- no post-hoc de-
   collision runs.

The scorer rejects any candidate whose row-uniqueness is not
essentially perfect (< 5% collision) -- the mutator must produce good SIDs AND
resolve collisions in the same pass.
"""
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
torch.manual_seed(0)


def generate_sids(query_emb, doc_emb, qrel_pairs, doc_ids, *, device):
    """
    Args:
        query_emb (torch.Tensor): Shape (n_queries, 768), float32.
        doc_emb (torch.Tensor): Shape (n_docs, 768), float32.
        qrel_pairs (torch.Tensor): (N, 2) long -- (q_row, d_row).
        doc_ids (Sequence[str]): length n_docs -- corpus doc ids in row order.
            Passed in so that decollision policies can use stable per-doc ties
            (see the TIGER-style append at the tail of the seed body).
        device (str): "cuda" or "cpu".

    Returns:
        np.ndarray: Shape (n_docs, L) with dtype=int32. **Every row must be
   unique.**
            The scorer rejects candidates whose collision rate is > 5%.
    """
# EVOLVE-BLOCK-START
    """
    Seed baseline:
      1. Train an RQ-VAE (3 layers, codebook 256) on doc embeddings.
      2. Extract raw codes -> (n_docs, 3) int32 (may collide).
```

Program 1. Seed RQVAE with fixed collision handling (continued)

```
      3. Append ONE disambiguation token per collision group (TIGER-style)
         -> (n_docs, 4) int32, all rows unique.
    """
    NUM_LAYERS = 3
    CODEBOOK_SIZE = 256
    LATENT_DIM = 256
    HIDDEN_MULT = 4
    COMMITMENT_COST = 0.01
    LR = 1e-4
    WEIGHT_DECAY = 0.01
    CODEBOOK_INIT_STD = 0.02
    EPOCHS = 100
    BATCH_SIZE = 2048
    RECON_WEIGHT = 1.0
    VQ_WEIGHT = 1.0

    dev = torch.device(device if torch.cuda.is_available() else "cpu")

    doc_tensor = torch.as_tensor(doc_emb, device=dev, dtype=torch.float32)
    x_all = F.normalize(doc_tensor, p=2, dim=-1)
    n_docs, in_dim = x_all.shape
    hidden_dim = HIDDEN_MULT * in_dim

    class _Encoder(nn.Module):
        def __init__(self):
            super().__init__()
            self.net = nn.Sequential(
                nn.Linear(in_dim, hidden_dim), nn.GELU(),
                nn.Linear(hidden_dim, hidden_dim), nn.GELU(),
                nn.Linear(hidden_dim, LATENT_DIM)
            )
        def forward(self, x): return self.net(x)

    class _Decoder(nn.Module):
        def __init__(self):
            super().__init__()
            self.net = nn.Sequential(
                nn.Linear(LATENT_DIM, hidden_dim), nn.GELU(),
                nn.Linear(hidden_dim, hidden_dim), nn.GELU(),
                nn.Linear(hidden_dim, in_dim)
            )
        def forward(self, z): return self.net(z)

    class _ResidualVQ(nn.Module):
        def __init__(self):
            super().__init__()
            self.codebooks = nn.ModuleList([
                nn.Embedding(CODEBOOK_SIZE, LATENT_DIM) for _ in range(
NUM_LAYERS)
            ])
            for emb in self.codebooks:
                emb.weight.data.normal_(mean=0.0, std=CODEBOOK_INIT_STD)

        def forward(self, z_e):
            residual = z_e
            quantized_sum = torch.zeros_like(z_e)
            total_vq_loss = z_e.new_zeros(())

            for codebook in self.codebooks:
                flat_r = residual.view(-1, residual.size(-1))
```

Program 1. Seed RQVAE with fixed collision handling (continued)

```
                distances = (
                    flat_r.pow(2).sum(dim=1, keepdim=True)
                    + codebook.weight.pow(2).sum(dim=1)
                    - 2 * torch.matmul(flat_r, codebook.weight.t())
                )
                idx = distances.argmin(dim=1).unsqueeze(1)
                encodings = torch.zeros(idx.size(0), codebook.num_embeddings,
device=z_e.device)
                encodings.scatter_(1, idx, 1)
                quantized = torch.matmul(encodings, codebook.weight).view(z_e.
shape)
                quantized_sum = quantized_sum + quantized

                e_loss = F.mse_loss(quantized.detach(), residual)
                q_loss = F.mse_loss(quantized, residual.detach())
                total_vq_loss = total_vq_loss + q_loss + COMMITMENT_COST *
e_loss

                residual = residual - quantized

            z_q = z_e + (quantized_sum - z_e).detach()
            return z_q, total_vq_loss

        @torch.no_grad()
        def encode(self, z_e):
            residual = z_e
            codes = []
            for codebook in self.codebooks:
                distances = (
                    residual.pow(2).sum(dim=1, keepdim=True)
                    + codebook.weight.pow(2).sum(dim=1)
                    - 2 * torch.matmul(residual, codebook.weight.t())
                )
                idx = torch.argmin(distances, dim=1)
                codes.append(idx)
                residual = residual - codebook(idx)
            return torch.stack(codes, dim=1)

    class _VQVAE(nn.Module):
        def __init__(self):
            super().__init__()
            self.encoder = _Encoder()
            self.vq = _ResidualVQ()
            self.decoder = _Decoder()

        def forward(self, x):
            z_e = F.normalize(self.encoder(x), dim=-1)
            z_q, vq_loss = self.vq(z_e)
            x_recon = self.decoder(z_q)
            return x_recon, vq_loss

        @torch.no_grad()
        def codes(self, x):
            z_e = F.normalize(self.encoder(x), dim=-1)
            return self.vq.encode(z_e)

    model = _VQVAE().to(dev)
    no_decay_keys = ("bias", "LayerNorm.weight")
    all_params = list(model.named_parameters())
    optim = torch.optim.AdamW([
```

Program 1. Seed RQVAE with fixed collision handling (continued)

```
        {"params": [p for n, p in all_params if not any(k in n for k in
    no_decay_keys)], "weight_decay": WEIGHT_DECAY},
        {"params": [p for n, p in all_params if any(k in n for k in
    no_decay_keys)], "weight_decay": 0.0},
      ], lr=LR)

      # Training
      for epoch in range(EPOCHS):
          perm = torch.randperm(n_docs)
          model.train()
          for start in range(0, n_docs, BATCH_SIZE):
              idx = perm[start:start + BATCH_SIZE]
              batch = x_all[idx].to(dev, non_blocking=True)
              x_recon, vq_loss = model(batch)
              recon_loss = F.mse_loss(x_recon, batch)
              loss = RECON_WEIGHT * recon_loss + VQ_WEIGHT * vq_loss
              optim.zero_grad()
              loss.backward()
              optim.step()

      model.eval()
      raw_codes = np.zeros((n_docs, NUM_LAYERS), dtype=np.int32)
      with torch.no_grad():
          for start in range(0, n_docs, BATCH_SIZE):
              end = min(start + BATCH_SIZE, n_docs)
              batch = x_all[start:end].to(dev, non_blocking=True)
              raw_codes[start:end] = model.codes(batch).cpu().numpy().astype(np.
    int32)

      # ----- In-band de-collision (TIGER-style disambiguation token) -----
      # Append ONE extra token per collision group, ranked by doc_id string.
      # This is the SAME logic as `decollision.decollide_sids` -- pasted here so
      # the mutator can rewrite it (or replace it with a fundamentally different
      # scheme: content-hash tail, LSH suffix, adaptive-depth, query-conditioned
      # ordering, etc.). The scorer no longer runs any post-hoc de-collision;
      # the returned array must already be row-unique.
      from collections import defaultdict
      max_code = int(raw_codes.max())
      groups = defaultdict(list)
      for i in range(n_docs):
          groups[tuple(int(x) for x in raw_codes[i])].append(i)
      all_codes = np.empty((n_docs, NUM_LAYERS + 1), dtype=np.int32)
      all_codes[:, :NUM_LAYERS] = raw_codes
      for _sid, members in groups.items():
          members_sorted = sorted(members, key=lambda idx: str(doc_ids[idx]))
          for rank, idx in enumerate(members_sorted):
              all_codes[idx, NUM_LAYERS] = max_code + 1 + rank
# EVOLVE-BLOCK-END

      assert isinstance(all_codes, np.ndarray), f"Expected np.ndarray output,
    got {type(all_codes)}"
      assert len(all_codes) == len(doc_emb), f"Doc count mismatch! Expected {len
    (doc_emb)} rows, got {len(all_codes)}"
      assert not np.isnan(all_codes).any(), "Output contains NaN values!"
      assert not np.isinf(all_codes).any(), "Output contains Infinite values!"

      return all_codes
```

A.2 NQ JOINT SID–COLLISION PROGRAM

Program 2. NQ joint SID–collision program

```
"""Evolve SID generation with de-collision blended IN-BAND.

The mutator owns ONE function: `generate_sids(query_emb, doc_emb, qrel_pairs,
  doc_ids, *, device)`.
Its return is `(n_docs, L) int32` with **every row unique** -- no post-hoc de-
  collision runs.

The scorer rejects any candidate whose row-uniqueness is not
essentially perfect (< 5% collision) -- the mutator must produce good SIDs AND
resolve collisions in the same pass.
"""
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
torch.manual_seed(0)


def generate_sids(query_emb, doc_emb, qrel_pairs, doc_ids, *, device):
    """
    Args:
        query_emb (torch.Tensor): Shape (n_queries, 768), float32.
        doc_emb (torch.Tensor): Shape (n_docs, 768), float32.
        qrel_pairs (torch.Tensor): (N, 2) long -- (q_row, d_row).
        doc_ids (Sequence[str]): length n_docs -- corpus doc ids in row order.
            Passed in so that decollision policies can use stable per-doc ties
            (see the TIGER-style append at the tail of the seed body).
        device (str): "cuda" or "cpu".

    Returns:
        np.ndarray: Shape (n_docs, L) with dtype=int32. **Every row must be
  unique.**
            The scorer rejects candidates whose collision rate is > 5%.
    """
# EVOLVE-BLOCK-START
    """
    Seed baseline:
      1. Train an RQ-VAE (3 layers, codebook 256) on doc embeddings.
      2. Extract raw codes -> (n_docs, 3) int32 (may collide).
      3. Append ONE disambiguation token per collision group (TIGER-style)
         -> (n_docs, 4) int32, all rows unique.
    """
    NUM_LAYERS = 3
    CODEBOOK_SIZE = 512
    LATENT_DIM = 256
    HIDDEN_MULT = 4
    COMMITMENT_COST = 0.01
    LR = 1e-4
    WEIGHT_DECAY = 0.01
    CODEBOOK_INIT_STD = 0.02
    EPOCHS = 100
    BATCH_SIZE = 2048
    RECON_WEIGHT = 1.0
    VQ_WEIGHT = 1.0
    CODEBOOK_ORTHO_WEIGHT = 1e-4

    dev = torch.device(device if torch.cuda.is_available() else "cpu")
```

Program 2. NQ joint SID–collision program (continued)

```
    doc_tensor = torch.as_tensor(doc_emb, device=dev, dtype=torch.float32)
    x_all = F.normalize(doc_tensor, p=2, dim=-1)
    n_docs, in_dim = x_all.shape
    hidden_dim = HIDDEN_MULT * in_dim

    class _Encoder(nn.Module):
        def __init__(self):
            super().__init__()
            self.net = nn.Sequential(
                nn.Linear(in_dim, hidden_dim), nn.GELU(),
                nn.Linear(hidden_dim, hidden_dim), nn.GELU(),
                nn.Linear(hidden_dim, LATENT_DIM)
            )
        def forward(self, x): return self.net(x)

    class _Decoder(nn.Module):
        def __init__(self):
            super().__init__()
            self.net = nn.Sequential(
                nn.Linear(LATENT_DIM, hidden_dim), nn.GELU(),
                nn.Linear(hidden_dim, hidden_dim), nn.GELU(),
                nn.Linear(hidden_dim, in_dim)
            )
        def forward(self, z): return self.net(z)

    class _ResidualVQ(nn.Module):
        def __init__(self):
            super().__init__()
            self.codebooks = nn.ModuleList([
                nn.Embedding(CODEBOOK_SIZE, LATENT_DIM) for _ in range(
NUM_LAYERS)
            ])
            for emb in self.codebooks:
                emb.weight.data.normal_(mean=0.0, std=CODEBOOK_INIT_STD)

        def forward(self, z_e):
            residual = z_e
            quantized_sum = torch.zeros_like(z_e)
            total_vq_loss = z_e.new_zeros(())

            for codebook in self.codebooks:
                flat_r = residual.view(-1, residual.size(-1))
                # Use cosine similarity for assignment to improve semantic
clustering
                # flat_r: (B, D), codebook.weight: (C, D)
                norm_r = F.normalize(flat_r, p=2, dim=-1)
                norm_w = F.normalize(codebook.weight, p=2, dim=-1)
                sims = torch.matmul(norm_r, norm_w.t()) # (B, C)

                idx = sims.argmax(dim=1).unsqueeze(1)
                encodings = torch.zeros(idx.size(0), codebook.num_embeddings,
device=z_e.device)
                encodings.scatter_(1, idx, 1)
                quantized = torch.matmul(encodings, codebook.weight).view(z_e.
shape)
                quantized_sum = quantized_sum + quantized

                e_loss = F.mse_loss(quantized.detach(), residual)
                q_loss = F.mse_loss(quantized, residual.detach())
```

Program 2. NQ joint SID–collision program (continued)

```
            # Entropy regularizer with temperature to encourage better
codebook distribution
            # sims: (B, CODEBOOK_SIZE) on dev
            probs = F.softmax(sims / 0.5, dim=-1)
            # Corrected entropy: sum over C, then mean over B
            entropy_loss = -torch.mean(torch.sum(probs * torch.log(probs +
 1e-6), dim=1))

            # Orthogonality loss to prevent codebook collapse: sum of
squared dot products of normalized weights
            # norm_w: (CODEBOOK_SIZE, LATENT_DIM) on dev
            ortho_loss = torch.sum(torch.matmul(norm_w, norm_w.t())**2) -
CODEBOOK_SIZE

            total_vq_loss = total_vq_loss + q_loss + (COMMITMENT_COST *
2.0) * e_loss - 0.05 * entropy_loss + CODEBOOK_ORTHO_WEIGHT * ortho_loss

            residual = residual - quantized

        z_q = z_e + (quantized_sum - z_e).detach()
        return z_q, total_vq_loss

    @torch.no_grad()
    def encode(self, z_e):
        residual = z_e
        codes = []
        for codebook in self.codebooks:
            # Consistent cosine assignment for inference
            norm_r = F.normalize(residual, p=2, dim=-1)
            norm_w = F.normalize(codebook.weight, p=2, dim=-1)
            idx = torch.matmul(norm_r, norm_w.t()).argmax(dim=1)
            codes.append(idx)
            residual = residual - codebook(idx)
        return torch.stack(codes, dim=1)

class _VQVAE(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = _Encoder()
        self.vq = _ResidualVQ()
        self.decoder = _Decoder()

    def forward(self, x):
        z_e = F.normalize(self.encoder(x), dim=-1)
        z_q, vq_loss = self.vq(z_e)
        x_recon = self.decoder(z_q)
        return x_recon, vq_loss

    @torch.no_grad()
    def codes(self, x):
        z_e = F.normalize(self.encoder(x), dim=-1)
        return self.vq.encode(z_e)

model = _VQVAE().to(dev)
no_decay_keys = ("bias", "LayerNorm.weight")
all_params = list(model.named_parameters())
optim = torch.optim.AdamW([
    {"params": [p for n, p in all_params if not any(k in n for k in
no_decay_keys)], "weight_decay": WEIGHT_DECAY},
```

Program 2. NQ joint SID–collision program (continued)

```
      {"params": [p for n, p in all_params if any(k in n for k in
no_decay_keys)], "weight_decay": 0.0},
  ], lr=LR)

  # Refined K-Means++ style initialization for the first codebook
  with torch.no_grad():
      subset_idx = torch.randperm(n_docs)[:CODEBOOK_SIZE]
      init_weights = x_all[subset_idx].to(dev)
      if in_dim != LATENT_DIM:
          proj = nn.Linear(in_dim, LATENT_DIM, bias=False).to(dev)
          nn.init.orthogonal_(proj.weight)
          init_weights = proj(init_weights)
      model.vq.codebooks[0].weight.copy_(F.normalize(init_weights, p=2, dim
=-1))

  # Task-aware signal: Compute a "target" vector for each doc based on
queries that retrieve it
  # doc_targets: (n_docs, in_dim) on dev
  doc_targets = torch.zeros((n_docs, in_dim), device=dev)
  counts = torch.zeros((n_docs, 1), device=dev)
  if qrel_pairs.shape[0] > 0:
      q_idx = qrel_pairs[:, 0]
      d_idx = qrel_pairs[:, 1]
      q_embs = torch.as_tensor(query_emb, device=dev, dtype=torch.float32)
      doc_targets.index_add_(0, d_idx, q_embs[q_idx])
      counts.index_add_(0, d_idx, torch.ones((qrel_pairs.shape[0], 1),
device=dev))
  doc_targets = F.normalize(doc_targets / (counts + 1e-6), p=2, dim=-1)

  WARMUP_EPOCHS = 10
  for epoch in range(EPOCHS):
      lr_scale = min(1.0, float(epoch + 1) / WARMUP_EPOCHS) if WARMUP_EPOCHS
 > 0 else 1.0
      for param_group in optim.param_groups:
          param_group['lr'] = LR * lr_scale

      # Dead-code revival every 10 epochs to maximize codebook utilization
      if epoch > 0 and epoch % 10 == 0:
          model.eval()
          with torch.no_grad():
              sample_idx = torch.randperm(n_docs)[:min(n_docs, BATCH_SIZE *
4)]
              z_e_sample = F.normalize(model.encoder(x_all[sample_idx].to(
dev)), dim=-1)
              residual = z_e_sample
              for cb in model.vq.codebooks:
                  norm_r = F.normalize(residual, p=2, dim=-1)
                  norm_w = F.normalize(cb.weight, p=2, dim=-1)
                  idx = torch.matmul(norm_r, norm_w.t()).argmax(dim=1)
                  counts = torch.bincount(idx, minlength=CODEBOOK_SIZE)
                  dead_mask = (counts == 0)
                  if dead_mask.any():
                      num_dead = dead_mask.sum().item()
                      replace_idx = torch.randperm(residual.size(0), device=
dev)[:num_dead]
                      cb.weight[dead_mask] = residual[replace_idx]
                  # Update residual for the next layer's check
                  residual = residual - cb(idx)
              model.train()
```

Program 2. NQ joint SID–collision program (continued)

```
        perm = torch.randperm(n_docs)
        model.train()
        for start in range(0, n_docs, BATCH_SIZE):
            idx = perm[start:start + BATCH_SIZE]
            batch = x_all[idx].to(dev, non_blocking=True) # (batch, in_dim)
            targets = doc_targets[idx] # (batch, in_dim)

            x_recon, vq_loss = model(batch) # x_recon: (batch, in_dim), batch:
 (batch, in_dim)
            recon_loss = F.mse_loss(x_recon, batch)
            # Use cosine distance for better semantic alignment alignment
            # x_recon: (batch, in_dim), targets: (batch, in_dim)
            align_loss = 1.0 - F.cosine_similarity(x_recon, targets).mean()

            loss = RECON_WEIGHT * recon_loss + VQ_WEIGHT * vq_loss + 0.8 *
align_loss
            optim.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optim.step()

            # EMA-style codebook update for stability:
            # Update codebook embeddings based on the mean of the vectors they
 quantized
            with torch.no_grad():
                z_e_batch = F.normalize(model.encoder(batch), dim=-1)
                residual_ema = z_e_batch
                for cb in model.vq.codebooks:
                    norm_r = F.normalize(residual_ema, p=2, dim=-1)
                    norm_w = F.normalize(cb.weight, p=2, dim=-1)
                    idx = torch.matmul(norm_r, norm_w.t()).argmax(dim=1)

                    # Compute sum of encoded vectors for each code: (
CODEBOOK_SIZE, LATENT_DIM)
                    sums = torch.zeros_like(cb.weight)
                    sums.index_add_(0, idx, residual_ema)
                    counts = torch.bincount(idx, minlength=CODEBOOK_SIZE).
float().unsqueeze(1)

                    # Dynamic EMA momentum: slower updates early on to
stabilize codebook
                    momentum = 0.01 if epoch < WARMUP_EPOCHS else 0.05
                    ema_update = sums / (counts + 1e-6)
                    cb.weight.data.mul_(1.0 - momentum).add_(ema_update *
momentum)

                    residual_ema = residual_ema - cb(idx)

    model.eval()
    raw_codes = np.zeros((n_docs, NUM_LAYERS), dtype=np.int32)
    with torch.no_grad():
        for start in range(0, n_docs, BATCH_SIZE):
            end = min(start + BATCH_SIZE, n_docs)
            batch = x_all[start:end].to(dev, non_blocking=True)
            raw_codes[start:end] = model.codes(batch).cpu().numpy().astype(np.
int32)

    # ----- In-band de-collision (Semantic-Aware Ranking) -----
    # We resolve collisions by appending a tail token based on proximity to
    # the group centroid in the original embedding space.
```

Program 2. NQ joint SID–collision program (continued)

```
    from collections import defaultdict

    # Move embeddings to CPU for NumPy processing to avoid device errors
    doc_emb_cpu = torch.as_tensor(doc_emb).cpu().numpy().astype(np.float32)

    max_code = int(raw_codes.max())
    groups = defaultdict(list)
    for i in range(n_docs):
        groups[tuple(int(x) for x in raw_codes[i])].append(i)

    all_codes = np.empty((n_docs, NUM_LAYERS + 1), dtype=np.int32)
    all_codes[:, :NUM_LAYERS] = raw_codes

    # Pre-normalize for cosine similarity
    norms = np.linalg.norm(doc_emb_cpu, axis=1, keepdims=True) + 1e-9
    norm_embs = doc_emb_cpu / norms

    for _sid, members in groups.items():
        if len(members) == 1:
            # No collision: assign a default sentinel 0
            all_codes[members[0], NUM_LAYERS] = 0
            continue

        # Rank by similarity to group centroid to preserve semantic local
  structure in the tail
        group_embs = norm_embs[members]
        centroid = np.mean(group_embs, axis=0)
        centroid /= (np.linalg.norm(centroid) + 1e-9)

        # Cosine similarity (dot product of normalized vectors)
        sims = group_embs @ centroid

        # Stable sort: primary key is similarity (desc), secondary key is
  member index
        # Use members list directly for tie-breaking to ensure stability
        sorted_idx_in_members = np.lexsort((members, -sims))

        for rank, local_idx in enumerate(sorted_idx_in_members):
            global_idx = members[local_idx]
            # Use rank + 1 so that collided docs are distinguished from non-
  collided docs (0)
            all_codes[global_idx, NUM_LAYERS] = rank + 1
# EVOLVE-BLOCK-END

    assert isinstance(all_codes, np.ndarray), f"Expected np.ndarray output,
  got {type(all_codes)}"
    assert len(all_codes) == len(doc_emb), f"Doc count mismatch! Expected {len
  (doc_emb)} rows, got {len(all_codes)}"
    assert not np.isnan(all_codes).any(), "Output contains NaN values!"
    assert not np.isinf(all_codes).any(), "Output contains Infinite values!"

    return all_codes
```

A.3 NQ Residual K-Means Program

The fixed collision-handling procedure is applied after this constructor.

Program 3. NQ residual K-means program

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
torch.manual_seed(0)

def generate_sids(query_emb, doc_emb, qrel_pairs, *, device):
    """
    Args:
        query_emb (np.ndarray): Shape (307358, 768), dtype=float32.
            Dense embeddings for unique queries.
        doc_emb (np.ndarray): Shape (109739, 768), dtype=float32.
            Dense embeddings for all documents in the corpus.
        qrel_pairs (np.ndarray): Shape (N_pairs, 2), dtype=int32.
            Index mapping pairs connecting queries to relevant documents:
            - `qrel_pairs[:, 0]`: Row indices into `query_emb`
            - `qrel_pairs[:, 1]`: Row indices into `doc_emb`
            Example usage for contrastive/alignment loss:
                `q_vecs = torch.from_numpy(query_emb[qrel_pairs[:, 0]])`
                `d_vecs = torch.from_numpy(doc_emb[qrel_pairs[:, 1]])`
        device (str): Execution device ("cuda" or "cpu").

    Returns:
        np.ndarray: Shape (109739, NUM_LAYERS) with dtype=int32,
            containing discrete codes in range [0, CODEBOOK_SIZE) for each
  document.
    """
# EVOLVE-BLOCK-START
    """
    Residual K-Means (RQ-KMeans) Paradigm.
    Successively quantizes the residue of the previous layer.
    This creates a coarse-to-fine hierarchical SID that significantly
    reduces collision rates by utilizing the full codebook capacity.
    """
    NUM_LAYERS = 3
    CODEBOOK_SIZE = 256
    KMEANS_ITERS = 15
    dev = torch.device(device if torch.cuda.is_available() else "cpu")

    # doc_tensor shape: (n_docs, dim), device: dev
    doc_tensor = torch.as_tensor(doc_emb, device=dev, dtype=torch.float32)
    x_all = F.normalize(doc_tensor, p=2, dim=-1)
    n_docs, dim = x_all.shape

    # Task-Aware Alignment for Layer 1
    x_first_layer = x_all.clone()
    if qrel_pairs.size(0) > 0:
        q_tensor = torch.as_tensor(query_emb, device=dev, dtype=torch.float32)
        q_tensor = F.normalize(q_tensor, p=2, dim=-1)
        q_idx = torch.as_tensor(qrel_pairs[:, 0], device=dev)
        d_idx = torch.as_tensor(qrel_pairs[:, 1], device=dev)

        q_sum = torch.zeros((n_docs, dim), device=dev)
        q_cnt = torch.zeros((n_docs, 1), device=dev)
        q_sum.index_add_(0, d_idx, q_tensor[q_idx])
        q_cnt.index_add_(0, d_idx, torch.ones((qrel_pairs.size(0), 1), device=
  dev))

        mask = (q_cnt > 0).squeeze()
        avg_q = q_sum / torch.clamp(q_cnt, min=1.0)
```

Program 3. NQ residual K-means program (continued)

```
        x_first_layer[mask] = 0.7 * x_all[mask] + 0.3 * avg_q[mask]
        x_first_layer = F.normalize(x_first_layer, p=2, dim=-1)

    def run_kmeans_residual(data, k, iters=15):
        n_samples = data.size(0)
        if n_samples <= k:
            return torch.arange(n_samples, device=dev), data

        indices = torch.randperm(n_samples, device=dev)[:k]
        centroids = data[indices].clone()

        labels = torch.zeros(n_samples, dtype=torch.long, device=dev)
        for _ in range(iters):
            dist = torch.cdist(data, centroids)
            labels = dist.argmin(dim=1)
            new_centroids = torch.zeros_like(centroids)
            for i in range(k):
                mask = (labels == i)
                if mask.any():
                    new_centroids[i] = data[mask].mean(dim=0)
                else:
                    new_centroids[i] = data[torch.randint(0, n_samples, (1,),
  device=dev)]
            centroids = new_centroids
        return labels, centroids

    all_codes = torch.zeros((n_docs, NUM_LAYERS), dtype=torch.long, device=dev
  )
    current_residual = x_first_layer

    for layer in range(NUM_LAYERS):
        labels, centroids = run_kmeans_residual(current_residual,
  CODEBOOK_SIZE, iters=KMEANS_ITERS)
        all_codes[:, layer] = labels

        # Residual: R_{l+1} = Normalize(R_l - Centroid_{label})
        assigned_centroids = centroids[labels]
        current_residual = current_residual - assigned_centroids
        current_residual = F.normalize(current_residual, p=2, dim=-1)

    # Status tracking placeholder
    print(f"Residual K-Means Complete | Layers: {NUM_LAYERS} | Codebook: {
  CODEBOOK_SIZE}^3")
    all_codes = all_codes.cpu().numpy().astype(np.int32)
# EVOLVE-BLOCK-END

    assert isinstance(all_codes, np.ndarray), f"Expected np.ndarray output,
  got {type(all_codes)}"
    assert len(all_codes) == len(doc_emb), f"Doc count mismatch! Expected {len
  (doc_emb)} rows, got {len(all_codes)}"
    assert not np.isnan(all_codes).any(), "Output contains NaN values!"
    assert not np.isinf(all_codes).any(), "Output contains Infinite values!"

    return all_codes
```

B MUTATOR PROMPT TEMPLATES

Prompt 1. Exploration

```
## Task
You are an expert AI researcher discovering a better Semantic ID (SID)
generation algorithm for generative retrieval. Unlike the classic setup,
**de-collision is IN-BAND** -- your `generate_sids` returns SIDs that are
already unique row-by-row. Your goal is to propose code modifications WITHIN
the `# EVOLVE-BLOCK-START` and `# EVOLVE-BLOCK-END` enclosed regions to
minimize SID collision rate at the source, maximize codebook perplexity, and
most importantly, maximize downstream generator Recall@10 on a held-out test
split when trained on query-to-SID supervision.

## You must use clear and fundamental reasoning. What makes the SID good for
generative retrieval?
- **Hierarchical Structure:** The SID should preserve coarse-to-fine
hierarchy as the generator (a language model) excel at coarse-to-fine
reasoning.
- **Similar yet Distinguishable:** A good SID must balance clustering
(generalization) with uniqueness (discrimination). Similar items should share
early tokens. But their SID should also be meaningfully distinguishable.
- **Task Alignment:** A good SID reflects the retrieval task dynamics, not
just raw semantic content. Content semantics is not the same as query
relevance.
- **Balanced Codebook:** At inference time, the generator uses
trie-constrained decoding with beam search. Good SID should produce a
well-balanced prefix tree.

## De-collision is now part of your job
The returned array is used AS-IS for training and retrieval -- the scorer runs
no post-hoc `decollide_sids`. **Every row must be unique.** The scorer rejects
candidates with > 15% collision, but that ceiling is a soft safety net -- an
E3 candidate should aim for a collision rate near zero. In-band de-collision
is the whole point.

Two axes to think about:
- **When**: resolve collisions in-band during quantization (avoid producing
  collisions in the first place -- balanced clustering, split-on-collision,
  Gumbel jitter on tied assignments) OR after quantization (append tail
  tokens to disambiguate residual collisions). Both are valid; hybrid is often
  best -- reduce the raw collision rate first, then patch what's left.
- **How the disambiguating suffix is ORDERED**: a good suffix places
  semantically-similar collided docs near each other in the tail token space,
  so the generator learns a smooth extension from the coarse prefix. Recent
  work explores: rank by doc_emb similarity within group, LSH bits of the
  residual vector as the tail, content-hash bits, adaptive depth (only extend
  the sequence for docs that actually collided -- TIGER's baseline appends to
  everyone; BONSAI's variable-length trie only appends where needed).

Beam search still uses the trie: keep the tail short and structured. A tail
of 1 token per doc (as in the seed) is the cheapest option; longer tails
are fine if they aid disambiguation but must stay bounded.

## Code Structure & Formatting Rules
- The code contains isolated zones enclosed by `# EVOLVE-BLOCK-START` and `#
EVOLVE-BLOCK-END`. All your edits must come from there. Leave all code
outside of these markers completely untouched.
- **Use SEARCH/REPLACE diffs**: SEARCH must exactly match the current code;
REPLACE is your edit.
```

Prompt 1. Exploration (continued)

```
- **Helper & Supporting Functions:** You are encouraged to write helper
functions. However, they **MUST** be defined *inside* the region enclosed by
`# EVOLVE-BLOCK-START` and `# EVOLVE-BLOCK-END`.

## Memory scale (hard constraint)
The corpus can hold up to ~320 K documents and up to ~370 K training queries
(MSMARCO scale; NQ is ~110 K docs / ~310 K queries). **NEVER allocate an
`(n_queries, n_docs)` or `(n_docs, n_docs)` dense tensor** -- that is
hundreds of GB and OOMs immediately. Safe patterns for supervised /
contrastive / task-aligned quantization:
- iterate `qrel_pairs` in mini-batches (a few thousand pairs at a time),
- sample K negatives per positive instead of scoring the full corpus,
- compute distance-to-codebook (`n_docs × codebook_size`, safe: `320K × 256
≈ 320MB`).

Anything you allocate that exceeds a few GB is a red flag. If unsure, chunk
it.

## Existing known ideas to use, combine, and evolve
You can explore entirely different paradigms:

- **Task-Aware / Supervised Quantization:** Use `qrels` and `query_emb` in
your loss function (e.g., contrastive loss), to make the SID aligned with the
downstream task.
- **Better Initialization:** Initialize codebook weights based on existing
document embeddings can prevent codebook collapse. For example, KMeans or
KMeans++ initialization.
- **Product Quantization (PQ) & Optimized PQ (OPQ):** Instead of residual
quantization on the full vector, split the vector into sub-vectors and apply
KMeans/VQ to each slice independently.
- **Non-Neural Clustering:** Residual Quantized K-Means (RQ-KMeans),
Hierarchical K-Means,
- **Add or Tweak Loss:** Prevent codebook collapse; Coarse-to-fine Hierarchy
tweaks; Retrieval & Beam Search Decoding-Aware Alignment; Collision
Avoidance; ...
- **Structured de-collision suffix:** rank-based (TIGER), residual-LSH bits,
content-hash bits, similarity-ordered within-group ranking
(semantically-close collided docs get near-tail codes), or adaptive-depth
extension (only collided groups grow the sequence).
- **Balanced / split-on-collision quantization:** enforce capacity limits per
code at each layer; when a code exceeds its capacity, split it (add jitter,
re-quantize residual with a fresh sub-codebook).

## Guidelines
- Prefer one or a few coherent ideas per edit rather than many unrelated
tweaks.
- Explain in comments or structure *why* a formulation is reasonable.
- If you add new parameters or signals, give them meaningful names and clear
roles.
- Novel formulations and ideas are encouraged.
- You should mainly propose multiple piece-wise edits. Rewriting the entire
EVOLVE-BLOCK is possible, but do so paringly.
- Whatever strategy you pick, the FINAL returned array MUST have unique rows.
The scorer rejects at > 15% collision, but a good candidate lands near zero --
 in-band de-collision is the whole point.
```

Prompt 2. Exploitation

```
## Task
You are an expert AI researcher **refining** a strong Semantic ID (SID)
generation algorithm for generative retrieval. Unlike the classic setup,
**de-collision is IN-BAND** -- the parent's `generate_sids` returns unique
rows and the scorer runs no post-hoc de-collision. The parent below has
already been evaluated and is one of the best-performing solutions in the
archive. Your job is to squeeze out **marginal gains** with careful, minimal
edits -- NOT to overhaul the algorithm.

Your goal remains: minimize SID collision rate at the source, maximize
codebook perplexity, and most importantly, maximize downstream generator
Recall@10 on a held-out test split.

## Refinement mindset -- what makes a good exploitation edit?
- **Hierarchical Structure:** Tighten the coarse-to-fine signal without
changing the paradigm. E.g., adjust `NUM_LAYERS`, add small residual-scale
factors, or tune commitment cost per layer to sharpen hierarchy.
- **Similar yet Distinguishable:** Small tweaks that improve codebook balance
-- EMA on codebook updates, dead-code re-initialization, temperature
annealing on softmax assignments, minor codebook size changes.
- **Task Alignment:** Add a small task-aligned loss term (e.g., a weak
contrastive regularizer using `qrels`) on top of the existing objective -- do
not replace the existing loss.
- **Balanced Codebook:** Add a small entropy/load-balancing regularizer,
gentle noise on the assignment logits, or a warmup phase for the codebook.
- **De-collision refinement:** If the parent uses rank-based tail tokens, try
similarity-ordered ranking (docs nearest by `doc_emb` in a group get adjacent
tail codes) -- this smooths the beam-search extension without changing
anything else. Keep it a MINIMAL change to the tail logic.

## Concrete kinds of edits that fit exploitation
- Tune hyperparameters: `LR`, `EPOCHS`, `BATCH_SIZE`, `COMMITMENT_COST`,
`CODEBOOK_INIT_STD`, loss weights.
- Change optimizer / schedule: AdamW -> Lion, add cosine warmup, add gradient
clipping.
- Add small stabilization: EMA codebook, dead-code revive, small dropout on
latents.
- Regularization tweaks: add small orthogonality or entropy term, adjust
`WEIGHT_DECAY` groups.
- Init changes: switch codebook init from Normal to KMeans/KMeans++ on the
first batch of encoded features.
- Tail-ordering refinements: `sorted(members, key=lambda i: str(doc_ids[i]))`
-> similarity-based ordering within-group.

## Guidelines
- Prefer **1-3 small, focused edits** per attempt. Do NOT rewrite the whole
EVOLVE-BLOCK.
- Preserve the parent's core architecture (RQ-VAE, PQ, hierarchical KMeans,
whatever it is) AND its de-collision family (rank-tail, hash-tail, in-band
split, etc.).
- Keep changes minimal and reason about why each one should help WITHOUT
hurting collision rate or training stability.
- Explain your reasoning briefly before each SEARCH/REPLACE block, especially
why the change is safe.
- If a change adds a new hyperparameter, set it to a conservative value and
add a brief comment.
- The FINAL returned array MUST still have unique rows. The scorer's ceiling
is 15% collision but the goal is near zero.

## Code Structure & Formatting Rules
```

Prompt 2. Exploitation (continued)

```
- The code contains isolated zones enclosed by `# EVOLVE-BLOCK-START` and `#
EVOLVE-BLOCK-END`. All your edits must come from there. Leave all code
outside of these markers completely untouched.
- **Use SEARCH/REPLACE diffs**: SEARCH must exactly match the current code;
REPLACE is your edit.
- **Helper & Supporting Functions:** You are encouraged to write helper
functions. However, they **MUST** be defined *inside* the region enclosed by
`# EVOLVE-BLOCK-START` and `# EVOLVE-BLOCK-END`.

## Memory scale (hard constraint)
The corpus can hold up to ~320 K documents and up to ~370 K training queries
(MSMARCO scale; NQ is ~110 K docs / ~310 K queries). **NEVER allocate an
`(n_queries, n_docs)` or `(n_docs, n_docs)` dense tensor** -- that is
hundreds of GB and OOMs immediately. Safe patterns for supervised /
contrastive / task-aligned quantization:
- iterate `qrel_pairs` in mini-batches (a few thousand pairs at a time),
- sample K negatives per positive instead of scoring the full corpus,
- compute distance-to-codebook (`n_docs × codebook_size`, safe: `320K × 256
≈ 320MB`).

Anything you allocate that exceeds a few GB is a red flag. If unsure, chunk
it.
```

Prompt 3. Paradigm shift

```
## Task
You are an expert AI researcher and the current line of attack for Semantic
ID (SID) generation has **stagnated**. Recent attempts have failed to produce
a globally better solution. Your job is to **abandon the parent's paradigm
entirely** and switch to a fundamentally different family of methods.
Remember: **de-collision is IN-BAND** -- the returned array must already be
row-unique.

Your goal remains: minimize SID collision rate at the source, maximize
codebook perplexity, and most importantly, maximize downstream generator
Recall@10 on a held-out test split.

## What "paradigm shift" means here
Look at the parent program's core method AND its de-collision scheme. Then
pick a method from the list below that is **furthest from the parent's
paradigm**. Do NOT tweak the parent -- replace it.

## Candidate paradigms (pick ONE that is far from the parent)
- **Product Quantization (PQ):** Split the doc embedding into M sub-vectors
of dim D/M; run KMeans on each slice independently; SID is the concatenation
of M cluster indices. No residual, no neural encoder.
- **Optimized PQ (OPQ):** Learn a rotation R before PQ so the sub-vector
splits are optimally decorrelated.
- **Hierarchical K-Means:** Recursively cluster doc embeddings. First-level
cluster -> partition -> recursively cluster within each partition. Each level
contributes one SID token.
- **Residual K-Means (RQ-KMeans, non-neural):** Like RQ-VAE but pure KMeans
on residuals. No autoencoder, no gradients -- just KMeans passes on residual
vectors.
- **Task-Aware Supervised Quantization:** Train a small model with a
contrastive loss using `qrels` and `query_emb` so the SID is
```

Prompt 3. Paradigm shift (continued)

```
retrieval-aligned rather than reconstruction-aligned. Contrastive loss on
`query_emb[q_idx]` vs quantized `doc_emb[d_idx]` from qrel pairs.
- **LSH-Style Random Projections + Quantization:** Random projection into
low-dim space then binary/ternary quantization; can be layered for hierarchy.
- **Balanced Clustering (constrained KMeans):** Explicitly enforce roughly
equal cluster sizes at each layer to maximize codebook perplexity. Naturally
reduces collision rate.
- **DiffQ / Gumbel-Softmax VQ:** Replace argmin quantization with
differentiable Gumbel-Softmax assignments and temperature annealing.
- **Adaptive-depth / variable-length SIDs (BONSAI-style):** Grow the SID only
where collisions occur. Easy docs get short SIDs; hard docs get longer ones.
Trie stays balanced without a uniform tail token.
- **In-band split-on-collision:** After each quantization layer, docs sharing
a code that exceeds a capacity limit are re-assigned to a fresh sub-codebook
(or perturbed and re-quantized). Zero residual collisions by construction.
- **Content-hash / LSH suffix:** Replace the rank-based tail with a hash of
the residual vector or of `doc_text[i]`. Semantically-similar colliders end
up in adjacent tail codes without needing a separate ordering step.

## Guidelines
- Pick ONE paradigm and commit to it. Do not mix families.
- Rewriting the entire EVOLVE-BLOCK is not only allowed here, it is
**expected**. Big SEARCH/REPLACE blocks are fine.
- Keep the function signature `generate_sids(query_emb, doc_emb, qrel_pairs,
doc_ids, *, device)` intact and continue returning a `(n_docs, L)` array of
int codes with **row-unique guarantee**. The scorer's ceiling is 15%
collision but a good candidate lands near zero.
- Preserve the `Loss / Collision / Ppx` status print (it is used by outer
logging).
- Reason briefly about **why** the new paradigm should beat the parent: what
weakness of the parent does it address? Which of the 4 SID quality principles
(Hierarchical, Similar-yet-Distinguishable, Task-Aligned, Balanced) does it
strengthen? How does it handle collisions in-band?
- If the paradigm has no gradient training (KMeans, LSH), remove the training
loop cleanly and leave a placeholder print so status tracking still runs.

## Code Structure & Formatting Rules
- The code contains isolated zones enclosed by `# EVOLVE-BLOCK-START` and `#
EVOLVE-BLOCK-END`. All your edits must come from there. Leave all code
outside of these markers completely untouched.
- **Use SEARCH/REPLACE diffs**: SEARCH must exactly match the current code;
REPLACE is your edit.
- **Helper & Supporting Functions:** You are encouraged to write helper
functions. However, they **MUST** be defined *inside* the region enclosed by
`# EVOLVE-BLOCK-START` and `# EVOLVE-BLOCK-END`.

## Memory scale (hard constraint)
The corpus can hold up to ~320 K documents and up to ~370 K training queries
(MSMARCO scale; NQ is ~110 K docs / ~310 K queries). **NEVER allocate an
`(n_queries, n_docs)` or `(n_docs, n_docs)` dense tensor** -- that is
hundreds of GB and OOMs immediately. Safe patterns for supervised /
contrastive / task-aligned quantization:
- iterate `qrel_pairs` in mini-batches (a few thousand pairs at a time),
- sample K negatives per positive instead of scoring the full corpus,
- compute distance-to-codebook (`n_docs × codebook_size`, safe: `320K × 256
≈ 320MB`).

Anything you allocate that exceeds a few GB is a red flag. If unsure, chunk
it.
```