COMPONENT-AWARE FEEDBACK FOR SELF-EVOLVING PROGRAMS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-guided evolutionary search can discover complex programs, but existing methods mostly only save candidate programs and fitness scores while discarding which component edits produced which fitness metric changes. Existing methods force the mutator LLM to infer the effect of prior edits from cluttered histories, making program search slow and unstable. This is especially true for locally servable LLMs to evolve multi-component systems. We introduce component-aware feedback, which compares each evaluated program with its parent, identifies the components that changed, and logs them with the associated metric differences into an attribution memory that later mutations read. The memory keeps each change in two reference frames, local against the parent it came from and global against the seed program, which shows both the immediate effect of a change and the cumulative progress made since the seed. We study this on LLM reranking, a multi-objective optimization problem where a multi-stage pipeline must balance quality against serving cost. Across twelve BRIGHT datasets, our method reaches the strongest baseline's final quality after a median of one third of the search budget and ends 7.2% higher in held-out nDCG@10, and under a cost-aware objective it finds pipelines that are on average more accurate while using 11% fewer tokens per query, showing component-aware feedback to be a promising direction for more efficient self-evolving systems.

1 INTRODUCTION

Evolutionary search guided by large language models (LLMs) improves algorithms by proposing changes to executable programs, evaluating the resulting programs, and using the outcomes to guide further proposals. FunSearch and AlphaEvolve demonstrate the potential of this approach for mathematical and algorithmic discovery (Romera-Paredes et al., 2024; Novikov et al., 2025). Since then, a growing line of systems has improved the loop in many ways: which parent to mutate, which population to grow, and to break out of local minima (Lange et al., 2025; Cemri et al., 2026). The other half of the loop has received much less attention. Every iteration ends with an evaluation, and evaluations are the only signal the search learns from. As the evolved programs grow larger, one question becomes increasingly important: *What information should each evaluation retain to best guide the mutator LLM in proposing the next change?* Today the answer is mostly a program and its score. The mutation prompt shows the parent, a few high scoring or diverse programs, their fitness, and the score changes of recent children (Novikov et al., 2025; Lange et al., 2025; Cemri et al., 2026). For a program with several interacting components, the full pipeline code accompanied by an overall fitness score leaves substantial interpretation challenge to the mutator LLM. To tell which earlier edits helped, the mutator must diff full programs implicitly, decide which of the changed functions and constants mattered, and connect them to whichever metrics moved. It must do all of this from a context that grows longer and more cluttered as the search proceeds. LLMs are known to use long contexts unevenly and to lose track of accumulated multi-turn histories (Liu et al., 2024b; Laban et al., 2026). This burden matters most for the open, locally servable models that make evolutionary search affordable, whereas AlphaEvolve, ShinkaEvolve, and AdaEvolve all report their results with frontier models (Novikov et al., 2025; Lange et al., 2025; Cemri et al., 2026). Reflection-based search and semantic-delta memories show that representing earlier modifications helps (Ye et al., 2024; Jiang et al., 2026), but they ask an LLM to describe what an edit did, which is both expensive and hard to trust. The changes in component and the corresponding

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.

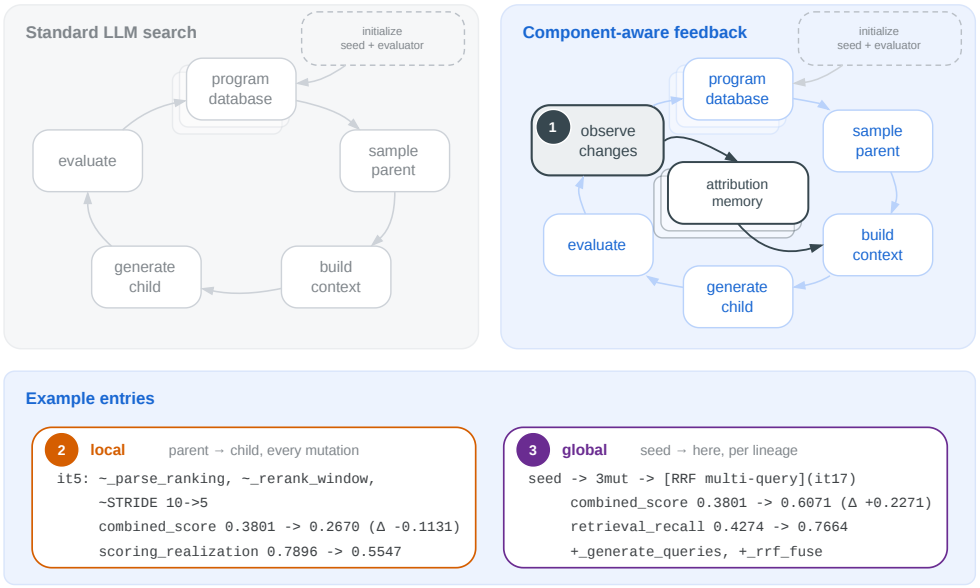


Figure 1: **Local and global component-aware feedback.** A standard LLM-guided search (top left) rebuilds its context from the database at every iteration and keeps nothing between iterations. We add an observe step (1), which diffs a child against its parent, and an attribution memory, which records each edit and is read by the context step (top right). Local and global entries (bottom) differ only in their reference point. A local entry (2) compares a child with its parent, and a global entry (3) compares it with the seed.

metric change can be programatically extracted. We argue that the mutator is best guided by these component-aware feedback.

We propose component-aware feedback for self-evolving programs, which turns each evaluation into a set of such records. After every evaluation, an *observe* step decomposes the parent and the child into named components, namely their functions and module-level constants. It identifies which components were added, removed, or modified, and associates this edit set with the vector of differences in every metric the evaluator reports. The records accumulate in an *attribution memory* that the mutation prompt reads, and each record is kept in two reference frames. When evolution is making steady progress, the *local* frame compares a child with its parent and tells the mutator whether its last step helped. When search stagnates, the *global* frame compares with seed and shows what has been attempted and changed so far. The records are built entirely from evaluations the search already performs, and we add them to AdaEvolve (Cemri et al., 2026) without changing its search controller. We study this approach on LLM reranking, which is an important problem on its own, and contains multiple components by nature. Reranking decides what a downstream model reads in search and retrieval-augmented systems. It remains an active research area, spanning prompting and log-likelihood methods (Sun et al., 2023; Qin et al., 2024; Chen et al., 2025b) and rerankers trained with reinforcement learning (Zhuang et al., 2026; Liu et al., 2026b), many of which are evaluated on the reasoning-intensive BRIGHT benchmark (SU et al., 2025). LLM reranker is a pipeline whose components constrain one another. Retrieval sets the ceiling on achievable quality, and the reranking strategy decides how much of that ceiling is achieved, and at what computation cost. Evolving LLM reranking is often a trade off problem between quality and cost, where a good edit of one component may negatively impact the other. We therefore evolve the complete pipeline, and study a quality-only objective and a cost-aware one.

We evaluate on all twelve BRIGHT datasets, with one open-weight 35B model with 3B active parameters serving as both mutator and reranker (Qwen Team, 2026). With component-aware feedback, the search is faster, more consistent, and finds better programs. It raises held-out nDCG@10 by 7.2% over AdaEvolve and leads on nine out of twelve datasets. On ten datasets, it reaches AdaE-

volve's final search score after a median of 31 out of 100 iterations, and its final scores vary less across runs. Under the cost-aware objective, it finds pipelines that are more accurate and use 11% fewer tokens per query compared to AdaEvolve. Its programs also transfer better. When run unchanged on other domains, they retain 90% of the score obtained by evolving on the target directly, against 79% for AdaEvolve. An ablation shows that the gains come from both attribution and from exposing more metrics. On circle packing and signal processing, our method is comparable to existing evolve methods, and works better on tasks where objectives compete with each other. These results suggest that when a program has many interacting components, telling the mutator which component changes moved which metrics is as important as deciding which programs to mutate.

2 METHODOLOGY

2.1 PROBLEM DEFINITION

LLM-guided evolutionary search can be fundamentally broken down into a simple loop. A user initializes the search by providing a seed program p_0 and an evaluator with a fitness function F, and the system repeats five steps until a budget of T iterations is reached:

1. **Selection:** sample a parent program p from the database D, together with a small set of inspiration programs.
2. **Context:** assemble a prompt from the parent and inspiration programs.
3. **Mutation:** generate a child program p' from a mutation model M given that prompt.
4. **Evaluation:** run the evaluator on the child program, obtaining a fitness score $f' = F(p')$.
5. **Update:** add p' into the database and update the controller that guides future selection.

The goal of this loop is to maximize the fitness function $F : \mathcal{P} \to \mathbb{R}$ over the space of executable programs $\mathcal{P}$ by repeatedly generating and evaluating new programs p_i over a fixed budget of T iterations. Most systems maintain not one database but K databases $D = \{D_1, D_2 \ldots, D_K\}$ that evolve semi-independently and periodically exchange programs, which keeps the design space from collapsing into a single lineage. Much of the effort in designing these systems goes into how to control the search to most efficiently utilize the fixed budget (e.g. which parent to sample, which database to spend the next iteration on, when to abandon a direction).

2.2 ADAPTIVE SEARCH CONTROL

Prior works controlled the search with hand-designed parameters, fixing exploration rates, database counts and stagnation criteria ahead of a run and re-tuning them per problem. AdaEvolve (Cemri et al., 2026) removes this friction by deriving those parameters from the search itself. For each database k it maintains an accumulated improvement signal, built from the normalized fitness gain $\delta_t^{(k)}$ that iteration t produced,

$$G_t^{(k)} \;=\; \rho\, G_{t-1}^{(k)} \;+\; (1-\rho)\big(\delta_t^{(k)}\big)^2 \tag{1}$$

with decay $\rho \in [0, 1)$. Because each value is built from its predecessor, $G_t^{(k)}$ summarizes how productive that database has recently been. The search parameters then become functions of this signal rather than constants.

Despite these improvements, the content of the mutation prompt is not a function of any such signal. It is a fixed template of meta-instructions containing the full source code of the parent and a few selected programs as inspiration, the fitness scores of each, and a list of score deltas for the parent's earlier children. The improvement signal is used only to decide which parent is sampled, and the inspiration programs are selected by fitness scores and by how textually different they are from that parent. Everything in the template is therefore an information dump of source code and fitness scores, and none of it indicates the state the search is in the way the improvement signal does for the search parameters. That signal describes how the search is doing rather than what it has tried, so it cannot attribute which changes produced which effects. Thus in its current state, the mutation

model is left to infer both the state of the search and the effect of every prior edit from a cluttered history.

2.3 COMPONENT-AWARE FEEDBACK

Component-aware feedback removes this inference. For every program the search evaluates, it records which components the mutation changed and how each metric responded, and carries that record forward into the prompts that follow. The mutator can then see what has already been tried and what each attempt did, so that later mutations build on those outcomes instead of rediscovering them. We add an *observe* step, which runs after each evaluation, and an *attribution memory* that the context step reads (Figure 1). The context is then no longer an undifferentiated dump of programs and scores, but rather a record of which changes were made and what each one produced.

We decompose each program into named components, its functions and its module-level constants. A program is then a map from component names to source, where $p[n]$ is the source code of a component named n and $\emptyset$ if p has no component with that name. Observe diffs the parent's map against the child's to identify which components the mutation touched, and attributes the resulting change in every metric to that set,

$$\mathcal{C}_t \;=\; \{\, n \mid p[n] \neq p'[n] \,\}, \qquad \Delta_t \;=\; \mu(p') - \mu(p), \tag{2}$$

where $\mathcal{C}_t$ is the set of names n for which the parent and child hold different source, and $\mu(\cdot)$ is the vector of metrics the evaluator returns. Three cases fall out of that single condition. A name in the child but not the parent is a component the mutation created, a name in the parent but not the child is one it removed, and a name in both with different source is one it modified. The pair $m_t = (\mathcal{C}_t, \Delta_t)$ is an *edit*, and it is the unit the attribution memory stores.

Edits accumulate in the attribution memory $\mathcal{A}$, which the context step reads alongside the database,

$$\text{prompt}_t \;=\; C\big(p,\, D_t,\, \mathcal{A}_t\big), \qquad \mathcal{A}_t \;=\; \mathcal{A}_{t-1} \cup \{m_t\}, \tag{3}$$

so the context depends on the whole trajectory of the search rather than only the programs that happened to be surfaced in D_t. The context becomes a function of the search state, as the search parameters already were in Eq. 1. As with the improvement signal, one memory is kept per database and the two are merged whenever the databases exchange programs, so an edit made in one lineage becomes available to the other at migration rather than immediately.

2.4 LOCAL AND GLOBAL REFERENCE FRAMES

An edit is only meaningful relative to a reference point. Applying Eq. 2 against the parent and against the seed gives two views that together show where the search stands,

$$m_t^{\text{loc}} \;=\; \big(\mathcal{C}_t^{\text{loc}},\, \Delta_t^{\text{loc}}\big), \qquad m_t^{\text{glob}} \;=\; \big(\mathcal{C}_t^{\text{glob}},\, \Delta_t^{\text{glob}}\big), \tag{4}$$

where $\mathcal{C}_t^{\text{loc}} = \{\, n \mid p[n] \neq p'[n] \,\}$ and $\Delta_t^{\text{loc}} = \mu(p') - \mu(p)$ measure the child against its parent, and $\mathcal{C}_t^{\text{glob}} = \{\, n \mid p_0[n] \neq p'[n] \,\}$ and $\Delta_t^{\text{glob}} = \mu(p') - \mu(p_0)$ measure it against the seed.

The local frame measures an edit against its parent and answers whether that change helped. Local entries go into the mutation prompt as a sliding window of the ten most recent edits.

The global frame replaces the parent with the seed, so it describes a whole lineage rather than a single step. A program reached by a dozen mutations is compared directly against p_0, which says which components that entire chain has introduced or discarded and what the chain bought. Global entries are built only when the search stagnates, for the prompt that proposes a new strategy, where the shared origin makes those chains comparable and gives a direction time to pay off.

The two frames can disagree, and an edit that lost ground against a strong parent is negative in Δ^{loc} while its lineage is still ahead in Δ^{glob}, so the direction is worth continuing even though the last step was not. Figure 1 shows an example entry in each frame. Appendix A shows full prompts with each frame.

2.5 APPLICATION TO LLM RERANKING

We study this on LLM reranking. A program p maps a query q to a ranking σ over a corpus $\mathcal{D}$, retrieving its own candidates from $\mathcal{D}$ and then reordering them. The fitness score is ranking quality averaged over a query set Q,

$$p : (q, \mathcal{D}) \longmapsto \sigma, \qquad F(p) \;=\; \frac{1}{|Q|} \sum_{q \in Q} \text{nDCG@10}\big(p(q, \mathcal{D})\big). \tag{5}$$

Beyond nDCG@10 the evaluator reports the recall of the retrieved set, the nDCG an oracle reranker would achieve on it, the fraction of that ceiling the program achieves, and the cost of a query in LLM calls and tokens. Together these form the metric vector μ of Eq. 2.

A fitness score of nDCG@10 alone credits a program for any gain in ranking quality and never charges it for the LLM calls that produced the gain. We therefore also optimize a cost-aware objective that trades ranking quality against efficiency. We measure efficiency in LLM tokens per query. Every input or output token is one more the model must process or generate, so token usage is closely aligned with both the serving cost and the latency of a program. It measures neither directly, but it is a good proxy for how efficient the programs our search proposes are. We do not use wall-clock time because it depends on serving conditions such as resource contention from other processes, which vary over a run and would score the same program differently at different points in it. The objective combines a quality term a and an efficiency term b, each normalized against the seed program,

$$s \;=\; 1 + \min(a, b) + \lambda\,(a + b), \qquad a = \frac{\text{nDCG} - \text{nDCG}_0}{1 - \text{nDCG}_0}, \qquad b = 1 - \frac{\tau}{\tau_0}, \tag{6}$$

where τ is tokens per query, subscript 0 denotes the seed, and $\lambda = 0.05$. The quality term a is the gain in nDCG@10 over the seed as a fraction of the largest gain possible from it, and the efficiency term b is the fraction of the seed's tokens per query the program saves. Both are zero at the seed and one at their ceiling, and the constant places the seed at a score of 1. We clip each term to $[-1, 1]$, since a program that uses far more tokens than the seed would otherwise have an unbounded efficiency term. The min term directs the search toward whichever of quality and efficiency is weaker. Without the λ term, the search could spend its whole budget fixated on either quality or efficiency, when the goal is the improve both. The λ term adds a small bonus for gains in both, so progress on either objective is rewarded.

3 EXPERIMENTS

3.1 EXPERIMENTAL SETUP

We evaluate on BRIGHT (SU et al., 2025), twelve retrieval datasets spanning StackExchange domains, coding, and theorem-based questions. Each dataset is split 60/40 into train and test. The search optimizes a fitness score evaluated on the train split, either nDCG@10 or the cost-aware score of Eq. 6. We report held-out test nDCG@10 of the final program, mean, and standard deviation over three runs per dataset.

The seed program is BM25 (Robertson & Zaragoza, 2009) over the corpus followed by a sliding-window listwise reranker (Sun et al., 2023). All methods share this seed, evaluator, budget of $T = 100$ iterations, model for both mutation and reranking, $K = 2$ databases with migration every 10 iterations, and four context programs per prompt. The mutator samples at temperature 0.7 with up to $16{,}384$ output tokens, and the reranking calls inside evolved programs decode greedily. The main results use Qwen3.6-35B-A3B (Qwen Team, 2026). We compare against BM25, the seed program, and AdaEvolve (Cemri et al., 2026). Appendix B shows the seed and examples of evolved programs under both objectives.

3.2 EXPERIMENTAL RESULTS

Ranking quality and cost. Table 1 reports held-out nDCG@10 under the quality-only objective. With component-aware feedback, the search returns programs that average 0.346 against AdaE-

Table 1: **Ranking performance on BRIGHT.** nDCG@10 on the train and held-out test splits, split 60/40 per dataset. Best is the best single run on test. $\Delta\%$ is our relative gain over AdaEvolve on test. Mean $\pm$ std over 3 runs.

	BM25		seed program		AdaEvolve			**Ours**			
dataset	train	test	train	test	train	test	best	train	test	best	$\Delta\%$
StackExchange											
bio.	0.184	0.167	0.372±.006	0.362	0.572±.027	0.581±.044	0.619	**0.602**±.005	**0.613**±.021	**0.633**	+5.6
earth.	0.238	0.363	0.404±.012	0.509	0.492±.016	0.559±.014	0.575	**0.529**±.020	**0.573**±.035	**0.606**	+2.4
econ.	0.180	0.118	0.299±.009	0.201	0.407±.024	0.253±.030	0.278	**0.412**±.018	**0.274**±.023	**0.299**	+8.1
psy.	0.089	0.228	0.307±.006	0.501	0.434±.023	**0.540**±.035	0.563	**0.448**±.008	0.533±.066	**0.604**	−1.3
rob.	0.154	0.075	0.337±.006	0.239	0.376±.014	0.254±.015	0.268	**0.404**±.025	**0.275**±.005	**0.281**	+8.4
stack.	0.199	0.142	0.322±.016	0.219	0.406±.029	0.211±.044	0.252	**0.424**±.011	**0.219**±.040	**0.260**	+3.9
sus.	0.172	0.137	0.267±.009	0.300	0.314±.029	0.302±.060	0.363	**0.353**±.026	**0.321**±.053	**0.377**	+6.3
Coding											
leet.	0.303	0.196	0.289±.004	0.190	0.360±.016	0.207±.014	0.220	**0.381**±.031	**0.247**±.017	**0.264**	+19.1
pony	0.052	0.047	0.165±.011	0.150	**0.267**±.011	**0.220**±.032	**0.254**	0.248±.003	0.204±.001	0.205	−7.0
Theorem-based											
aops	0.037	0.098	0.057±.004	0.130	**0.081**±.013	**0.119**±.028	**0.145**	0.080±.019	0.087±.025	0.115	−26.4
theoQ.	0.086	0.071	0.129±.006	0.124	0.314±.060	0.286±.048	0.325	**0.381**±.037	**0.348**±.054	**0.380**	+21.7
theoT.	0.004	0.037	0.063±.002	0.157	0.421±.131	0.344±.130	0.494	**0.536**±.022	**0.459**±.033	**0.496**	+33.5
mean	0.141	0.140	0.251	0.257	0.370	0.323	0.363	**0.400**	**0.346**	**0.377**	+7.2

Table 2: **Cost-aware search on BRIGHT.** Test nDCG@10 and LLM tokens per query of the final program. Both searches optimize the cost-aware objective rather than nDCG alone. Δ is the change in test nDCG@10 against Table 1. Cheaper is the quality-only program's tokens per query divided by this program's. Mean $\pm$ std over 3 runs.

	AdaEvolve				**Ours**			
dataset	nDCG@10	Δ	tok/q	cheaper	nDCG@10	Δ	tok/q	cheaper
StackExchange								
bio.	0.542±.012	-0.039	**10,118**	6.2x	**0.558**±.010	-0.055	12,071	15.1x
earth.	0.521±.021	-0.038	27,539	2.2x	**0.540**±.030	-0.033	**14,930**	2.6x
econ.	**0.225**±.007	-0.029	14,453	5.2x	0.217±.044	-0.056	**7,055**	5.2x
psy.	0.441±.031	-0.099	**10,422**	5.6x	**0.488**±.019	-0.045	25,736	2.8x
rob.	**0.238**±.010	-0.016	45,186	1.6x	0.226±.038	-0.049	**23,025**	5.4x
stack.	**0.202**±.057	-0.009	16,518	3.8x	0.185±.077	-0.035	**10,693**	3.4x
sus.	0.297±.048	-0.004	14,505	1.2x	**0.341**±.027	+0.020	**12,466**	2.6x
Coding								
leet.	**0.232**±.014	+0.024	7,553	3.3x	0.219±.029	-0.029	**6,920**	5.2x
pony	**0.166**±.113	-0.054	**7,635**	8.3x	0.141±.013	-0.063	12,548	7.4x
Theorem-based								
aops	0.090±.048	-0.029	**2,151**	37.5x	**0.091**±.019	+0.003	8,847	3.7x
theoQ.	0.218±.020	-0.069	6,398	4.7x	**0.264**±.007	-0.084	**5,049**	12.6x
theoT.	0.379±.067	+0.036	**8,432**	3.9x	**0.393**±.064	-0.066	12,765	9.0x
mean	0.296	-0.027	14,243	7.0x	0.305	-0.041	12,676	6.2x

volve's 0.323, a relative gain of 7.2%, and leads on nine of the twelve datasets. Both searches far exceed the seed at 0.257 and BM25 at 0.140.

Under the quality-only objective this improvement comes at a cost. These programs use 71.8k LLM tokens per query averaged over the twelve datasets against AdaEvolve's 53.7k. A component that widens the candidate pool is credited with higher ranking quality and is never penalized for adding more LLM calls or tokens per query. Thus, an efficient pipeline and an expensive one are indistinguishable under a quality-only objective. Component-aware feedback finds the better-ranking program at the expense of cost and latency because nothing in the objective promotes efficiency. Exploring the trade-off between ranking quality and efficiency therefore requires a fitness score that accounts for both.

The cost-aware objective of Eq. 6 scores ranking quality and token usage together. Under it, both searches return programs six to seven times cheaper than under quality alone (Table 2). Component-

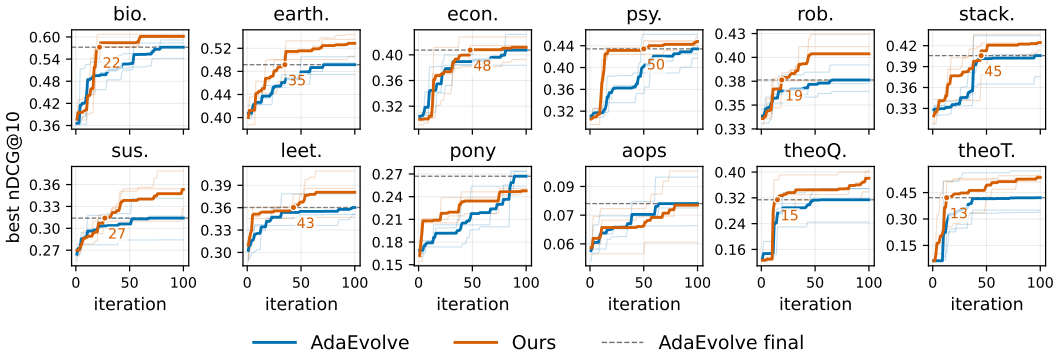


Figure 2: **Convergence on BRIGHT.** Best-so-far nDCG@10 on the train split, one panel per dataset. Solid lines are the mean over 3 runs, faint lines the individual runs. The marker gives the iteration at which our mean first reaches AdaEvolve's final score.

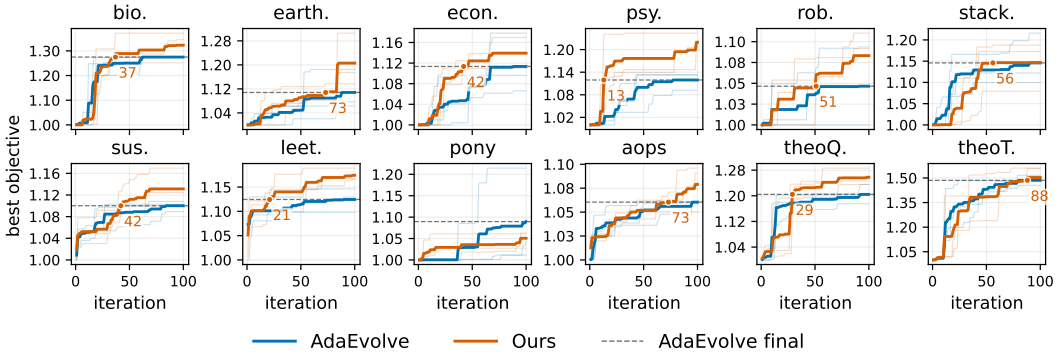


Figure 3: **Convergence under the cost-aware objective.** Best-so-far objective value on the train split, one panel per dataset. The seed program scores 1. Solid lines are the mean over 3 runs, faint lines the individual runs. The marker gives the iteration at which our mean first reaches AdaEvolve's final value.

aware feedback leads on both terms, reaching 0.305 nDCG@10 at 12,676 tokens per query against AdaEvolve's 0.296 at 14,243, an 11% saving at a higher score.

Convergence speed. If the mutator no longer has to infer the effect of each prior edit from a cluttered history, the search should reach a given score in fewer iterations. Figures 2 and 3 plot the best-so-far train score against iteration, one panel per dataset. Under the quality-only objective our mean reaches AdaEvolve's final score on ten of the twelve datasets, at a median of 31 iterations out of 100 and a mean of 32. Under the cost-aware objective it does so on eleven of twelve, at a median of 42 and a mean of 48. Our method's curve also stays above AdaEvolve's for most of the run on eleven of the twelve datasets under both objectives.

Both searches plateau well before the budget runs out, improving by a median of under 1% over the final quarter under the quality-only objective, so most of what a search will find it finds early. Each iteration pairs a mutation call with an evaluation, and in most applications of evolutionary search the evaluation dominates the cost, whether it runs a pipeline over a benchmark, trains a model, or executes a simulation. A search that reaches a given score in a third of the iterations therefore does so for roughly a third of the compute, and under a fixed compute budget spends more of that budget searching from better programs.

Cross-domain generalization. To test whether the programs component-aware feedback finds are specific to the dataset they were evolved on, we take the three programs each method optimized on biology under the quality-only objective and run them unchanged on four other datasets. The four datasets differ in how closely they resemble biology. Earth science is a neighboring natural science, robotics shares biology's StackExchange source, and leetcode and theoremqa come from the coding and theorem-based sections of the BRIGHT benchmark. Each is scored on the held-out split.

Table 3: **Cross-domain generalization on BRIGHT.** Programs optimized on biology scored on other datasets' held-out splits (nDCG@10). Seed and optimized on that dataset are the unevolved program and each method's search run on that dataset respectively, all from Table 1. Retained is the transferred program's score as a percentage of the same method's score when optimized on that dataset. Mean ± std over 3 runs.

		optimized on that dataset		optimized on biology		retained	
dataset	seed	AdaEvolve	**Ours**	AdaEvolve	**Ours**	AdaEvolve	**Ours**
earth.	0.509	0.559±.014	**0.573**±.035	0.517±.029	**0.548**±.037	92%	**96%**
rob.	0.239	0.254±.015	**0.275**±.005	0.220±.024	**0.284**±.028	87%	**103%**
leet.	0.190	0.207±.014	**0.247**±.017	0.126±.056	**0.177**±.022	61%	**71%**
theoQ.	0.124	0.286±.048	**0.348**±.054	0.222±.104	**0.311**±.108	78%	**89%**
mean	0.266	0.327	**0.361**	0.271	**0.330**	79%	**90%**

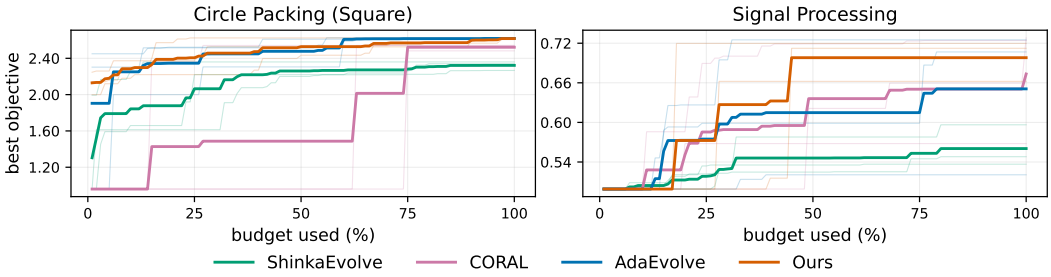


Figure 4: **Mathematical optimization.** Best-so-far objective against the fraction of the search budget spent, one panel per benchmark, higher is better. Solid lines are the mean over 3 runs, faint lines the individual runs.

Component-aware feedback finds programs that generalize better across datasets. Both methods lose ranking quality when we applied their biology programs to a new dataset, but ours lose less on every dataset (Table 3) and keep 90% of their in-domain score on average compared to 79% for AdaEvolve, even matching the in-domain programs on robotics.

Beyond retrieval. Beyond retrieval, we evaluate component-aware feedback on two mathematical optimization benchmarks (Figure 4). Circle packing places 26 circles in a unit square to maximize the sum of their radii. Signal processing evolves a filter that removes noise from a changing signal while tracking it closely, scored on a combination of several measures of smoothness, accuracy and noise removal. Alongside AdaEvolve we compare against ShinkaEvolve (Lange et al., 2025), another LLM-driven evolutionary search, and CORAL (Qu et al., 2026), a multi-agent system of autonomous coding agents. CORAL has no notion of an iteration, so we give it a budget equal to the average cost of one of our runs.

On circle packing, component-aware feedback matches AdaEvolve, 2.617 against 2.619, with the same small spread acorss runs, and both lead CORAL at 2.522 and ShinkaEvolve at 2.322. Circle packing has a single score, so attribution adds little to the score change AdaEvolve already reports. On signal processing, component-aware feedback is more consistent than AdaEvolve and CORAL. Its three runs stay between 0.662 and 0.720, while AdaEvolve and CORAL each have a run that drops to 0.521 and 0.568, which gives component-aware feedback the highest mean, 0.698 versus 0.673 for CORAL, 0.651 for AdaEvolve and 0.560 for ShinkaEvolve. Unlike circle packing, signal processing has competing goals, since a filter that removes more noise also responds more slowly to real changes. Attribution shows the mutator which edits moved which measure, which is what balancing these competing objectives requires, as with quality and cost in reranking.

Ablations. Component-aware feedback has two parts, the evaluator's decomposed metrics and their attribution to the components that changed. We isolate each part to test its value on biology, leetcode, and theoremqa, one dataset from each section of the BRIGHT benchmark. We compare against AdaEvolve and only evolving the reranking prompt. Evolving the whole program raises test nDCG@10 from 0.245 to 0.358. Decomposed metrics raise it to 0.382 and attribution to 0.403, so each part accounts for about half of our gain over AdaEvolve. Attribution adds little on train, 0.454 versus 0.448, but narrows the drop from train to test from 0.066 to 0.051.

Table 4: **Component ablation on BRIGHT.** nDCG@10 on one dataset per BRIGHT section. Prompt evolution evolves the reranking prompt, and the other rows evolve the program. Mean ± std over 3 runs.

	bio.		leet.		theoQ.		mean	
method	train	test	train	test	train	test	train	test
prompt evolution	0.408±.012	0.373±.011	0.376±.007	0.221±.019	0.146±.001	0.140±.004	0.310	0.245
AdaEvolve	0.572±.027	0.581±.044	0.360±.016	0.207±.014	0.314±.060	0.286±.048	0.416	0.358
Ours	0.602±.005	0.613±.021	0.381±.031	**0.247**±.017	**0.381**±.037	**0.348**±.054	**0.454**	**0.403**
− attribution	**0.609**±.036	**0.635**±.060	**0.392**±.023	0.208±.004	0.343±.073	0.302±.030	0.448	0.382

4 RELATED WORK

LLM-guided program evolution. FunSearch and AlphaEvolve established the basic loop of LLM mutation, program evaluation, and a program database (Romera-Paredes et al., 2024; Novikov et al., 2025). EoH and ReEvo extended the loop to heuristic design (Liu et al., 2024a; Ye et al., 2024), and OpenEvolve provides an open implementation of AlphaEvolve (Sharma, 2025). Most later work improves search control. ShinkaEvolve does so through parent sampling, novelty rejection, and model selection (Lange et al., 2025), AdaEvolve through adaptive exploration and budget allocation (Cemri et al., 2026), and EvoX by evolving the search strategy itself (Liu et al., 2026a). CORAL and SwarmResearch instead hand the loop to autonomous coding agents (Qu et al., 2026; Virk et al., 2026). Our work is orthogonal to search control as we focus on the feedback provided to the mutator.

Feedback for LLM optimizers OpenEvolve and AdaEvolve shows past solutions and their scores in the prompt. ReEvo and ShinkaEvolve summarize past attempts in LLM-written reflections (Ye et al., 2024; Lange et al., 2025). For compound AI systems, MIPRO, TextGrad, Trace, and GEPA assign credit to individual modules using surrogate models, textual gradients, execution traces, and reflection (Opsahl-Ong et al., 2024; Yuksekgonul et al., 2025; Cheng et al., 2024; Agrawal et al., 2026). DeltaEvolve describes each parent to child change in text and pairs it with a qualitative signal of improvement (Jiang et al., 2026). Component-aware feedback instead reads directly from the source code diff and the evaluator, so we do not need LLM summaries, extra evaluations, or learned models.

LLM reranking. Zero-shot LLM rerankers differ in how they elicit relevance. Pointwise methods use the likelihood of relevance labels or of the query (Nogueira et al., 2020; Zhuang et al., 2024a; Sachan et al., 2022), or attention patterns (Chen et al., 2025a). Comparative methods use pairwise, setwise, listwise, or tournament prompting (Qin et al., 2024; Zhuang et al., 2024b; Sun et al., 2023; Gangi Reddy et al., 2024; Chen et al., 2025b). Another line of work trains rerankers, through distillation (Pradeep et al., 2023; Weller et al., 2025) or reinforcement learning (Zhuang et al., 2026; Zhang et al., 2025; Liu et al., 2026b; Cai et al., 2026), and these rerankers are often evaluated on BRIGHT (SU et al., 2025). JudgeRank and DIVER combine several reasoning or retrieval stages into one pipeline (Niu et al., 2024; Sun et al., 2025). Each of these methods is a hand-designed point in a space of retrieval and reranking choices. We instead search that space with a fixed, locally served LLM. Because all methods we compare share one seed, one model, and one evaluation protocol, our results measure the search itself and are not meant to compete with published BRIGHT numbers.

5 CONCLUSION

Most work on LLM-guided evolutionary search refines how the search is controlled. We asked instead what each evaluation should retain to guide the mutator. Our answer is component-aware feedback, a record of which components each edit changed and their associated outcomes. With it, a locally served model reaches the strongest baseline's final score in a third of the iterations and evolves reranking pipelines that score higher and generalize better to new datasets. More broadly, it helps the search better navigate trade-offs between competing objectives, such as quality against cost in reranking. Component-aware feedback provides a practical way to make self-evolving systems more efficient, complementary to improving the search itself.

AI USE STATEMENT

Qwen3.6-35B-A3B is used as the mutator and the reranker in our study, as described in Section 3.2. The authors used LLM-based assistants to polish the writing and survey related works. As well as help structuring tables, figures, and prompts. The authors wrote all technical content, designed the method and experiments, and checked every result. We take full responsibility for the content presented in the paper.

ETHICS STATEMENT

This work uses only publicly available benchmarks. BRIGHT is built from public StackExchange posts, coding problems, and math resources. Circle packing and signal processing are synthetic tasks. Our method requires running code that LLMs generate, but all candidate programs are executed within a controlled environment. Evolutionary search also uses substantial compute. Our results show that it reaches a given score in fewer iterations, and the cost-aware objective finds cheaper pipelines, and both of these reduce that cost. We are not aware of other ethical concerns specific to this work.

REPRODUCIBILITY STATEMENT

Section 3.1 specifies the full experiment setup, and all experiments are carried out with open source LLMs and public benchmarks. All experiments use the open-weight Qwen3.6-35B-A3B served locally with vLLM. Appendix A gives the complete system, mutation, and strategy prompts for both AdaEvolve and our method. Appendix B.1 gives the seed program and example evolved programs. We report mean and standard deviation over three independent evolve runs for every result.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=RQm2KQTM5r.

Yuzheng Cai, Yanzhao Zhang, Dingkun Long, Mingxin Li, Pengjun Xie, and Weiguo Zheng. ERank: Fusing supervised fine-tuning and reinforcement learning for effective and efficient text reranking. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 30121–30129, 2026. doi: 10.1609/aaai.v40i36.40261. URL https://ojs.aaai.org/index.php/AAAI/article/view/40261.

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, 2026. URL https://arxiv.org/abs/2602.20133.

Shijie Chen, Bernal Jiménez Gutiérrez, and Yu Su. Attention in large language models yields efficient zero-shot re-rankers. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=yzloNYH3QN.

Yiqun Chen, Qi Liu, Yi Zhang, Weiwei Sun, Xinyu Ma, Wei Yang, Daiting Shi, Jiaxin Mao, and Dawei Yin. TourRank: Utilizing large language models for documents ranking with a tournament-inspired strategy. In *Proceedings of the ACM Web Conference 2025*. Association for Computing Machinery, 2025b. doi: 10.1145/3696410.3714863. URL https://arxiv.org/abs/2406.11678.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next AutoDiff: Generative optimization with rich feedback, execution traces, and LLMs. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/83ba7056bce2c3c3c27e17397cf3e1f0-Abstract-Conference.html.

Revanth Gangi Reddy, JaeHyeok Doo, Yifei Xu, Md Arafat Sultan, Deevya Swain, Avirup Sil, and Heng Ji. FIRST: Faster improved listwise reranking with single token decoding. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 8642–8652. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.491. URL https://aclanthology.org/2024.emnlp-main.491/.

Jiachen Jiang, Tianyu Ding, and Zhihui Zhu. DeltaEvolve: Accelerating scientific discovery through momentum-driven evolution, 2026. URL https://arxiv.org/abs/2602.02919.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. LLMs get lost in multi-turn conversation. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2505.06120.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

Fei Liu, Tong Xialiang, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 32201–32223. PMLR, 2024a. URL https://proceedings.mlr.press/v235/liu24bs.html.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024b. doi: 10.1162/tacl_a_00638.

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, Alvin Cheung, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. EvoX: Meta-evolution for automated discovery, 2026a. URL https://arxiv.org/abs/2602.23413.

Wenhan Liu, Xinyu Ma, Weiwei Sun, Yutao Zhu, Yuchen Li, Dawei Yin, and Zhicheng Dou. ReasonRank: Empowering passage ranking with strong reasoning ability. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 22007–22031. Association for Computational Linguistics, 2026b. doi: 10.18653/v1/2026.acl-long.1005. URL https://aclanthology.org/2026.acl-long.1005/.

Tong Niu, Shafiq Joty, Ye Liu, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. JudgeRank: Leveraging large language models for reasoning-intensive reranking, 2024. URL https://arxiv.org/abs/2411.00142.

Rodrigo Nogueira, Zhiying Jiang, Ronak Pradeep, and Jimmy Lin. Document ranking with a pretrained sequence-to-sequence model. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 708–718. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.findings-emnlp.63. URL https://aclanthology.org/2020.findings-emnlp.63/.

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.

Krista Opsahl-Ong, Michael J. Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 9340–9366. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.emnlp-main.525. URL https://aclanthology.org/2024.emnlp-main.525/.

Ronak Pradeep, Sahel Sharifymoghaddam, and Jimmy Lin. RankZephyr: Effective and robust zero-shot listwise reranking is a breeze!, 2023. URL https://arxiv.org/abs/2312.02724.

Zhen Qin, Rolf Jagerman, Kai Hui, Honglei Zhuang, Junru Wu, Le Yan, Jiaming Shen, Tianqi Liu, Jialu Liu, Donald Metzler, Xuanhui Wang, and Michael Bendersky. Large language models are effective text rankers with pairwise ranking prompting. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 1504–1518. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-naacl.97. URL https://aclanthology.org/2024.findings-naacl.97/.

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. In *Conference on Language Modeling (COLM)*, 2026.

Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026. URL https://qwen.ai/blog?id=qwen3.6-35b-a3b.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, April 2009. ISSN 1554-0669. doi: 10.1561/1500000019. URL https://doi.org/10.1561/1500000019.

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. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://www.nature.com/articles/s41586-023-06924-6.

Devendra Sachan, Mike Lewis, Mandar Joshi, Armen Aghajanyan, Wen-tau Yih, Joelle Pineau, and Luke Zettlemoyer. Improving passage retrieval with zero-shot question generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 3781–3797. Association for Computational Linguistics, 2022. doi: 10.18653/v1/2022.emnlp-main.249. URL https://aclanthology.org/2022.emnlp-main.249/.

Asankhaya Sharma. OpenEvolve: An open-source implementation of AlphaEvolve. https://github.com/codelion/openevolve, 2025.

Hongjin SU, Howard Yen, Mengzhou Xia, Weijia Shi, Niklas Muennighoff, Han yu Wang, Liu Haisu, Quan Shi, Zachary S Siegel, Michael Tang, Ruoxi Sun, Jinsung Yoon, Sercan O Arik, Danqi Chen, and Tao Yu. BRIGHT: A realistic and challenging benchmark for reasoning-intensive retrieval. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=ykuc5q381b.

Duolin Sun, Meixiu Long, Dan Yang, Junjie Wang, Yecheng Luo, Yue Shen, Jian Wang, Hualei Zhou, Chunxiao Guo, Peng Wei, Jiahai Wang, and Jinjie Gu. DIVER: A multi-stage approach for reasoning-intensive information retrieval, 2025. URL https://arxiv.org/abs/2508.07995.

Weiwei Sun, Lingyong Yan, Xinyu Ma, Shuaiqiang Wang, Pengjie Ren, Zhumin Chen, Dawei Yin, and Zhaochun Ren. Is ChatGPT good at search? investigating large language models as re-ranking agents. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 14918–14937, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.923. URL https://aclanthology.org/2023.emnlp-main.923/.

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.

Orion Weller, Kathryn Ricci, Eugene Yang, Andrew Yates, Dawn Lawrie, and Benjamin Van Durme. Rank1: Test-time compute for reranking in information retrieval. In *Second Conference on Language Modeling*, 2025. URL https://arxiv.org/abs/2502.18418.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/4ced59d480e07d290b6f29fc8798f195-Abstract-Conference.html.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 639(8055):609–616, 2025. doi: 10.1038/s41586-025-08661-4.

Le Zhang, Bo Wang, Xipeng Qiu, Siva Reddy, and Aishwarya Agrawal. REARANK: Reasoning re-ranking agent via reinforcement learning. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 2458–2471. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.emnlp-main.125. URL https://aclanthology.org/2025.emnlp-main.125/.

Honglei Zhuang, Zhen Qin, Kai Hui, Junru Wu, Le Yan, Xuanhui Wang, and Michael Bendersky. Beyond yes and no: Improving zero-shot LLM rankers via scoring fine-grained relevance labels. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*, pp. 358–370. Association for Computational Linguistics, 2024a. URL https://aclanthology.org/2024.naacl-short.31/.

Shengyao Zhuang, Honglei Zhuang, Bevan Koopman, and Guido Zuccon. A setwise approach for effective and highly efficient zero-shot ranking with large language models. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Association for Computing Machinery, 2024b. doi: 10.1145/3626772.3657813. URL https://arxiv.org/abs/2310.09497.

Shengyao Zhuang, Xueguang Ma, Zheng Yao, Shuai Wang, Bevan Koopman, Jimmy Lin, and Guido Zuccon. Rank-R1: Enhancing reasoning in LLM-based document rerankers via reinforcement learning. In *Proceedings of the 49th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Association for Computing Machinery, 2026. doi: 10.1145/3805712.3809961. URL https://cs.uwaterloo.ca/~jimmylin/publications/Zhuang_etal_SIGIR2026_RankR1.pdf.

A PROMPTS

The search makes two kinds of LLM calls. Every iteration, a mutation call edits a parent program. When the search stagnates, a strategy call proposes new directions for the mutation calls to implement. Both searches use the same system prompt (Prompt 1) and the same exploration and exploitation blocks (Prompts 2 and 3). All examples come from biology runs.

A.1 SYSTEM PROMPT

The system prompt states the task, the function the program must implement, and the tools the program can call. Only the dataset name changes between datasets.

Prompt 1: System prompt

```
You are an expert tasked with iteratively improving a solution.
Your goal is to maximize the COMBINED SCORE while exploring diverse approaches.
The system maintains a collection of diverse solutions — both high combined score AND diversity are
  valuable.

The task: given a biology search query, return document ids ranked best—first. Scoring is
nDCG@10 against hidden gold labels. Reading gold labels or importing the evaluator or
dataset is prohibited.

CONTRACT — the evaluator calls exactly this function (keep its name and signature):
    rank(query: str, tools) —> list[str]        # doc_ids, most —> least relevant
```

```
TOOLS (passed in as `tools`; do not import anything to get these):
    tools.bm25(query_text: str, k: int) —> list[str]   # BM25 top—k over the corpus; callable repeatedly
    tools.doc(doc_id: str) —> str                      # full document text
    tools.llm(messages: list[dict], max_tokens: int = 1024) —> str   # one chat completion; greedy, token—
  capped
```

A.2 MUTATION PROMPT

Every iteration makes one mutation call. Depending on whether the search is exploring or exploiting, the mutation prompt can include a exploration block (Prompt 2) or an exploitation block (Prompt 3).

Prompt 2: Mutation prompt, exploration block

```
## PARENT SELECTION CONTEXT
This parent was selected through diversity—driven sampling to explore different regions.

### EXPLORATION GUIDANCE
— Consider alternative algorithmic approaches
— Don't be constrained by the parent's approach
— Look for fundamentally different algorithms or novel techniques
— Balance creativity with correctness

Your goal: Discover new approaches that might outperform current solutions.
```

Prompt 3: Mutation prompt, exploitation block

```
## PARENT SELECTION CONTEXT
This parent was selected from the archive of top—performing programs.

### OPTIMIZATION GUIDANCE
— This solution works well, but meaningful improvements are still possible
— You may refine the existing approach OR introduce better algorithms
— Consider: algorithmic improvements, better data structures, efficient libraries
— Ensure correctness is maintained

Your goal: Improve upon this solution.
```

Prompts 4 and 5 are examples of full mutation prompts from AdaEvolve and component-aware feedback.

Prompt 4: AdaEvolve mutation prompt

```
# Current Solution Information
— Main Metrics:
— combined_score: 0.4569

Metrics:
  — ndcg_at_10: 0.4569
  — num_queries: 62
— Focus areas: — Consider simplifying — solution length exceeds 500 characters

# Program Generation History
## Previous Attempts

No previous attempts yet.

## Other Context Solutions
These programs represent diverse approaches and creative solutions that may be relevant to the current task:

### Program 1 (combined_score: 0.3837)
Score breakdown:  — ndcg_at_10: 0.3837  — num_queries: 62

```python
[... 36 lines of program source ...]
```

Program 2 (combined_score: 0.4389)
Score breakdown: — ndcg_at_10: 0.4389 — num_queries: 62

```python
[... 66 lines of program source ...]
```
```

```
```

Program 3 (combined_score: 0.4569)
Score breakdown: — ndcg_at_10: 0.4569 — num_queries: 62

```python
[... 84 lines of program source ...]
```

Program 4 (combined_score: 0.4569)
Score breakdown: — ndcg_at_10: 0.4569 — num_queries: 62

```python
[... 84 lines of program source ...]
```

Current Solution

PARENT SELECTION CONTEXT
This parent was selected from the archive of top—performing programs.

OPTIMIZATION GUIDANCE
— This solution works well, but meaningful improvements are still possible
— You may refine the existing approach OR introduce better algorithms
— Consider: algorithmic improvements, better data structures, efficient libraries
— Ensure correctness is maintained

Your goal: Improve upon this solution.

Program Information
combined_score: 0.4569
Score breakdown:
  — ndcg_at_10: 0.4569
  — num_queries: 62

```python
[... 84 lines of program source ...]
```

BREAKTHROUGH IDEA — IMPLEMENT THIS

The search has stagnated globally. You MUST implement this breakthrough idea:

IDEA: Explicit Relevance Scoring with LLM

HOW TO IMPLEMENT:
Replace the sliding—window listwise reranker with an explicit relevance scoring approach. Retrieve top 50 candidates via BM25 and query expansion, then prompt the LLM to assign a numeric relevance score (1—10) to each document individually. Parse these scores and use the sorted function to order candidates by score descending, returning the top 10. This directly optimizes for nDCG@10 by focusing the LLM's reasoning on absolute relevance rather than relative ordering of many items, which reduces token consumption and eliminates ordering drift in lower—ranked positions. By scoring each doc independently, you avoid context window limitations and ensure the most relevant documents bubble to the top based on explicit evidence.

TARGET METRIC: nDCG@10 precision and top—10 recall

CAUTIONS: Ensure the prompt strictly requests a structured numeric output to allow reliable parsing with regex. Handle cases where the LLM returns non—numeric values by defaulting to a low score. Keep the candidate pool small (30—50) to stay within token limits while maintaining sufficient recall.

APPROACH TYPE: sorted

CRITICAL:
— You MUST implement the breakthrough idea
— Ensure the paradigm is actually used in your implementation (not just mentioned in comments)
— Correctness is essential — your implementation must be correct and functional
— Verify output format matches evaluator requirements
— Make purposeful changes that implement the idea
— Test your implementation logic carefully

PREVIOUS ATTEMPTS ON THIS PARENT
Summary: 0 improved, 0 unchanged, 1 regressed
  1. 0.4569 —> 0.2710 (—0.1859) [REGRESSED]
Avoid repeating approaches that didn't work.

Task
Suggest improvements to the program that will improve its COMBINED_SCORE.
The system maintains diversity across these dimensions: score, complexity.
Different solutions with similar combined_score but different features are valuable.

You MUST use the exact SEARCH/REPLACE diff format shown below to indicate changes:

```
<<<<<<< SEARCH
# Original code to find and replace (must match exactly)
=======
# New replacement code
>>>>>>> REPLACE

Example of valid diff format:
<<<<<<< SEARCH
for i in range(m):
    for j in range(p):
        for k in range(n):
            C[i, j] += A[i, k] * B[k, j]
=======
# Reorder loops for better memory access pattern
for i in range(m):
    for k in range(n):
        for j in range(p):
            C[i, j] += A[i, k] * B[k, j]
>>>>>>> REPLACE

CRITICAL: You can suggest multiple changes. Each SEARCH section must EXACTLY match code in "# Current
  Solution" — copy it character—for—character, preserving all whitespace and indentation. Do NOT
  paraphrase or reformat.
Be thoughtful about your changes and explain your reasoning thoroughly.
Include a concise docstring at the start of functions describing the exact approach taken.

IMPORTANT: If an instruction header of "## IMPORTANT: ..." is given below the "# Current Solution", you MUST
   follow it. Otherwise,
focus on targeted improvements of the program.

— Time limit: Programs should complete execution within 1500 seconds; otherwise, they will timeout.
```

Prompt 5: Mutation prompt with component-aware feedback

```
# Current Solution Information
— Main Metrics:
— combined_score: 0.3942

Metrics:
  — ndcg_at_10: 0.3942
  — num_queries: 62
  — retrieval_recall: 0.4274
  — oracle_ndcg: 0.4813
  — scoring_realization: 0.8190
  — llm_calls_per_query: 18.0000
  — llm_tokens_per_query: 53770.3710
— Focus areas: — Consider simplifying — solution length exceeds 500 characters

# Program Generation History
## What this search has already tried

### Best so far (0.3903)
— made at it8: ~_parse_ranking, ~_rerank_window, ~STRIDE 10—>5, ~WINDOW 20—>15   combined_score 0.3801 →
   0.3903 (Δ +0.0103)
      llm_calls_per_query 9.0000 → 18.0000   llm_tokens_per_query 34707.4032 → 60074.0968   oracle_ndcg
  0.4813 → 0.4813   retrieval_recall 0.4274 → 0.4274   scoring_realization 0.7896 → 0.8110
  its parts:
    _parse_ranking:
        [... 52 lines of source ...]
    _rerank_window:
        [... 29 lines of source ...]
    rank:
        [... 16 lines of source ...]

### Other programs in the archive (2)
— program (0.3868)
  made at it9: +_rerank_window, ~_parse_ranking   combined_score 0.2781 → 0.3868 (Δ +0.1087) (+rewired `
  rank`)
      llm_calls_per_query 1.0000 → 18.0000   llm_tokens_per_query 4213.5968 → 53108.7742   oracle_ndcg
  0.3200 → 0.4813   retrieval_recall 0.2737 → 0.4274   scoring_realization 0.8692 → 0.8036
  its parts:
    _parse_ranking:
        [... 19 lines of source ...]
    _rerank_window:
```

```
        [... 21 lines of source ...]
    rank:
        [... 13 lines of source ...]
— program (0.3801)
  its parts:
    _parse_ranking:
        [... 11 lines of source ...]
    _rerank_window:
        [... 24 lines of source ...]
    rank:
        [... 16 lines of source ...]

### Changes, and what each one moved  (most recent 10)
[... 7 earlier edits ...]
— it10: ~_parse_ranking, ~_rerank_window, +STRIDE=5, ~WINDOW 25—>15   combined_score 0.2943 → 0.3834 (Δ
  +0.0891) (+rewired `rank`)
      llm_calls_per_query 1.0000 → 18.0000   llm_tokens_per_query 4984.4032 → 60007.6452   oracle_ndcg
  0.4813 → 0.4813   retrieval_recall 0.4274 → 0.4274   scoring_realization 0.6114 → 0.7964
    _parse_ranking:
        Parse LLM ranking output into 0—based indices.
        Tries bracket format, numbered lists, comma—separated; fills missing with original order.
    _rerank_window:
        Reorder window_ids by relevance using LLM listwise ranking.
        Uses structured prompt with reasoning instruction. Falls back to input order on failure.
— it11: —STRIDE, ~DEPTH 100—>50, ~WINDOW 15—>20   combined_score 0.3834 → 0.2847 (Δ —0.0987) (+rewired
   `rank`)
      llm_calls_per_query 18.0000 → 1.0000   llm_tokens_per_query 60007.6452 → 4470.6613   oracle_ndcg
  0.4813 → 0.3996   retrieval_recall 0.4274 → 0.3505   scoring_realization 0.7964 → 0.7124
— it15: ~_parse_ranking, ~_rerank_window   combined_score 0.3868 → 0.3779 (Δ —0.0089) (+rewired `rank`)
      llm_calls_per_query 18.0000 → 18.0000   llm_tokens_per_query 53108.7742 → 55624.3226   oracle_ndcg
  0.4813 → 0.4813   retrieval_recall 0.4274 → 0.4274   scoring_realization 0.8036 → 0.7852
    _parse_ranking:
        Parse LLM ranking output into a valid permutation of 0..k—1.
        Prioritizes explicit bracket format [a] > [b] > [c], then falls back to
        numbered lists. Always returns a valid permutation by filling gaps with
        original order.
    _rerank_window:
        Reorder window_ids by relevance using LLM listwise ranking.
        Uses a system message for better instruction following and a structured
        prompt with explicit format constraints. Falls back to input order on failure.

# Current Solution

## PARENT SELECTION CONTEXT
This parent was selected from the archive of top—performing programs.

### OPTIMIZATION GUIDANCE
— This solution works well, but meaningful improvements are still possible
— You may refine the existing approach OR introduce better algorithms
— Consider: algorithmic improvements, better data structures, efficient libraries
— Ensure correctness is maintained

Your goal: Improve upon this solution.

## Program Information
combined_score: 0.3942
Score breakdown:
  — ndcg_at_10: 0.3942
  — num_queries: 62
  — retrieval_recall: 0.4274
  — oracle_ndcg: 0.4813
  — scoring_realization: 0.8190
  — llm_calls_per_query: 18.0000
  — llm_tokens_per_query: 53770.3710

```python
[... 58 lines of program source ...]
```

BREAKTHROUGH IDEA — IMPLEMENT THIS

The search has stagnated globally. You MUST implement this breakthrough idea:

IDEA: Reciprocal Rank Fusion (RRF) for Multi—Query Retrieval

HOW TO IMPLEMENT:
Biology queries often miss relevant documents due to terminology gaps. Use the LLM to generate 3—5
  semantically related queries. Run tools.bm25 for the original and each expanded query. Merge all candidate
   pools using Reciprocal Rank Fusion (RRF), scoring each document as the sum of 1 / (k + rank) across all
```

```
  queries (with k=60). This single fusion step captures lexical and semantic diversity in one pass, directly
   boosting retrieval recall and oracle NDCG without complex pipelines.

TARGET METRIC: retrieval_recall, oracle_ndcg

CAUTIONS: Keep expanded queries concise to stay within token limits. Use a fixed RRF constant. Ensure strict
   deduplication while preserving rank positions. Handle cases where BM25 returns fewer than k results
  gracefully.

APPROACH TYPE: custom_rrf_fusion

CRITICAL:
— You MUST implement the breakthrough idea
— Ensure the paradigm is actually used in your implementation (not just mentioned in comments)
— Correctness is essential — your implementation must be correct and functional
— Verify output format matches evaluator requirements
— Make purposeful changes that implement the idea
— Test your implementation logic carefully

# Task
[... identical to the Task section of Prompt 4 ...]
```

A.3 STRATEGY PROMPT

When the search stagnates, a strategy call proposes three new strategies. Later mutation calls implement them on the best program. AdaEvolve's strategy prompt shows the best program and a list of strategies already tried. Component-aware feedback adds the seed as a reference point.

Prompt 6: AdaEvolve strategy prompt

```
## Problem Objective

You are an expert tasked with iteratively improving a solution.
Your goal is to maximize the COMBINED SCORE while exploring diverse approaches.
The system maintains a collection of diverse solutions — both high combined score AND diversity are
  valuable.

The task: given a biology search query, return document ids ranked best—first. Scoring is
nDCG@10 against hidden gold labels. Reading gold labels or importing the evaluator or
dataset is prohibited.

CONTRACT — the evaluator calls exactly this function (keep its name and signature):
    rank(query: str, tools) —> list[str]       # doc_ids, most —> least relevant

TOOLS (passed in as `tools`; do not import anything to get these):
    tools.bm25(query_text: str, k: int) —> list[str]   # BM25 top—k over the corpus; callable repeatedly
    tools.doc(doc_id: str) —> str                      # full document text
    tools.llm(messages: list[dict], max_tokens: int = 1024) —> str   # one chat completion; greedy, token—
  capped


## Optimization Targets

Optimize the primary scalar score defined by the evaluator.

## Evaluator Code (shows how solutions are scored)

```python
[... evaluator source ...]
```

Current Best Program (score: 0.483734)

```python
[... 90 lines of program source ...]
```

CRITICAL: Analyze the current program first. What algorithm does it use?
What are its strengths and weaknesses? How can you improve upon it?

CRITICAL: ANALYZE THE CURRENT PROGRAM FIRST
Before suggesting new ideas, carefully analyze the Current Program above:
— What algorithm/approach does it use? (This is what's WORKING — score 0.483734)
— What are its strengths? (Why does it achieve this score?)
— What are its weaknesses? (What limits further improvement?)
```

— How can you improve it? (How to beat it?)

IMPORTANT: The program above is the CURRENT program that needs to be improved. Start by understanding what works, then suggest breakthrough ideas that build on or improve it.

Analysis Framework — Complete Before Generating Ideas

STEP 0: Understand the TASK (MOST IMPORTANT — DO THIS FIRST)
— What is the problem asking you to do?
— What is the goal or objective? (maximize, minimize, optimize)
— What are the inputs and outputs?
— What needs to be improved? (variables/decisions that affect the goal)
— What constraints exist?

STEP 1: Analyze the Evaluator Code
— How are solutions scored?
— What metrics are computed?
— What causes failures or penalties?

STEP 2: Identify Metrics
— What is the primary metric or Pareto objective set?
— How is it calculated?
— What secondary metrics exist?
— If variance/std is penalized, the program needs consistency across scenarios

STEP 3: Identify Constraints
— What conditions must be satisfied?
— What validation happens?
— What causes score penalties?

STEP 4: Identify Problem Structure
— Is processing sequential or global?
— Are decision variables discrete or continuous?
— What dependencies exist between decisions?
— **CRITICAL:** What data does your program receive vs what the evaluator uses?
— **CRITICAL:** How are metrics computed across components? Independently then aggregated, or jointly?

STEP 5: Determine Appropriate Approach
— Match approach to problem structure
— Consider what has worked vs failed before
— Identify promising library/technique combinations

STEP 6: Identify Improvement Opportunities
— What would increase each metric?
— What would satisfy constraints better?
— What fundamentally different approaches could work?

Current best score is 0.483734. Your ideas must improve the configured optimization targets and, in multiobjective mode, explicitly reason about objective trade—offs.

Previously Tried Ideas — CHECK THIS FIRST

CRITICAL: Review what was already tried. Do NOT suggest ideas that use
the same libraries, functions, or approaches as FAILED attempts.

— FAILED: sorted — Explicit Relevance Scoring with LLM (improvement: +0.0000)
— FAILED: set.union — Multi—Query Entity Expansion for Recall (improvement: +0.0000)
— FAILED: itertools.combinations — Pairwise Knockout Tournament for Top—10 Selection (improvement: +0.0000)
— SUCCESS: scipy.ndimage.generic_filter — Multi—stage retrieval funnel with explicit top—10 optimization (improvement: +0.0268)
— SUCCESS: numpy.fft.fft — Query understanding and adaptive retrieval using LLM—generated search terms (improvement: +0.0268)
— SUCCESS: scipy.cluster.hierarchy.linkage — Iterative retrieval with gap analysis and complementary query generation (improvement: +0.0268)

STRICT PROHIBITION: Do NOT keep suggesting approaches that have already failed.
If an approach failed, understand WHY before suggesting similar techniques.
Prioritize approaches that are fundamentally different from failed attempts.

Learning from Failures — Understand Root Causes:
When a technique fails badly (score decreased significantly), understand WHY before suggesting alternatives:
— **Fundamental mismatch:** Wrong problem type (e.g., continuous optimizer on discrete problem) —> avoid that entire class of approaches
— **Structural mismatch:** Wrong approach for problem structure (e.g., linear proxy for non—linear objective) —> use approaches that match the actual structure
— **Implementation issues:** If the same library failed multiple times or very badly (>10% decrease), it likely indicates a fundamental mismatch — suggest a different class of approaches

```
## Technique Guidance

Note: Standard scientific libraries (scipy, numpy, etc.) are available. PyTorch and TensorFlow are not
  available.

For Continuous Optimization with Constraints:
— scipy.optimize.minimize with constraint handling (SLSQP, trust—constr)
— Multiple initial guesses for global optimization
— Geometric approaches (Voronoi, convex hull)

For Discrete/Combinatorial Problems:
— Greedy heuristics with good ordering
— Local search (swaps, moves)
— scipy.optimize.linear_sum_assignment for assignment problems
— scipy.optimize.linprog for linear constraints

For Graph/Network Problems:
— NetworkX algorithms (shortest path, min spanning tree, flow)
— Spectral methods (eigenvalue—based ordering)

For Repair/Reconstruction:
— Heuristic—based detection and correction
— Structural constraint exploitation
— Averaging/interpolation for consistency

For Robust Filtering/Noise Reduction:
— scipy.signal (medfilt, savgol_filter, wiener) for direct filtering
— Use methods that handle outliers better than mean—based (median, percentile)
— Do NOT use scipy.optimize.minimize to tune filter parameters
— Use filtering functions directly, not multi—stage optimization

General Principles:
— Prefer single—function library calls over multi—stage pipelines
— Match algorithm to problem structure
— Simple approaches with good heuristics often beat complex methods

## ANTI—PATTERNS — Critical rules about what NOT to do

1. Do NOT use multi—stage optimization: Do NOT call one function then optimize its output. Deterministic
  setup code followed by a single optimization call is allowed.

2. Do NOT use scipy.optimize.minimize for hyperparameter tuning: Use minimize to solve the problem directly,
   NOT to tune parameters for another function.

3. Do NOT use scipy.optimize.minimize for discrete problems: Continuous optimizers cannot handle discrete
  constraint violations properly.

4. Each idea MUST be a single—function library call: Do NOT suggest multi—stage processing (e.g., "call A
  then call B").

AVOID: DEAP, genetic algorithm libraries, domain—specific complex libraries, custom research algorithms, or
   any library requiring additional `pip install`

Learning from Success:
When an approach succeeds, think: what principle made it work? Learn and think of better ideas, don't just
  add complexity. If breakthrough patterns are known, prioritize approaches that match them.

## DIVERSITY REQUIREMENTS

Before generating ideas, explicitly think:
— Idea 1: [Type A — e.g., algorithmic refinement or library—based approach]
— Idea 2: [Type B — e.g., structural change or processing pattern — DIFFERENT from A]
— Idea 3: [Type C — e.g., different technique or optimization method — DIFFERENT from A and B]

Verify: Are these DIFFERENT types? NOT variations of the same approach.

Each idea must:
— Use DIFFERENT libraries/techniques than failed attempts
— Target DIFFERENT metrics/aspects from the evaluator
— Be independently implementable
— Prefer clear implementations (different != more complex)

### Be Specific and Actionable

Not vague: "Try optimization"
Specific: "Use scipy.optimize.minimize with SLSQP method"

— Include exact library names, function names, methods, parameters
— Provide step—by—step implementation guide
```

```
— Focus on core logic that implements the idea correctly
— Handle edge cases and avoid errors/warnings
— For optimization: use multiple initializations, appropriate iteration counts and convergence criteria (
  evaluation timeout: {self.eval_timeout}s)

## Output Format

IMPORTANT: Respond with a JSON object containing exactly 3 idea objects under the "ideas" key.
Do not include code patches or diffs — describe strategies in natural language.

Generate 3 breakthrough ideas of DIFFERENT types.

Each idea must be a JSON object with these fields:
— "idea": Clear, direct description with library/technique name
— "description": Detailed implementation guide (5—10 sentences)
— "what_to_optimize": What metrics/areas to focus on
— "cautions": Important implementation details to watch for
— "approach_type": Exact "library.function" format (e.g., "scipy.optimize.minimize")

Diversity Requirement: Each idea must use a DIFFERENT approach type.
Do not generate variations of the same technique.

Return ONLY a JSON object in this shape: {"ideas": [ ... ]} with 3 paradigm objects. No other text.

Example:
```json
{
    "ideas": [
        {
            "idea": "Use scipy.optimize.minimize with SLSQP",
            "description": "Apply scipy.optimize.minimize directly to optimize all variables together...",
            "what_to_optimize": "primary evaluator score",
            "cautions": "Ensure constraints are properly formulated, use multiple starting points",
            "approach_type": "scipy.optimize.minimize"
        }
    ]
}
```
```

Prompt 7: Strategy prompt with component-aware feedback

```
## Problem Objective

You are an expert tasked with iteratively improving a solution.
Your goal is to maximize the COMBINED SCORE while exploring diverse approaches.
The system maintains a collection of diverse solutions — both high combined score AND diversity are
  valuable.

The task: given a biology search query, return document ids ranked best—first. Scoring is
nDCG@10 against hidden gold labels. Reading gold labels or importing the evaluator or
dataset is prohibited.

CONTRACT — the evaluator calls exactly this function (keep its name and signature):
    rank(query: str, tools) —> list[str]        # doc_ids, most —> least relevant

TOOLS (passed in as `tools`; do not import anything to get these):
    tools.bm25(query_text: str, k: int) —> list[str]   # BM25 top—k over the corpus; callable repeatedly
    tools.doc(doc_id: str) —> str                      # full document text
    tools.llm(messages: list[dict], max_tokens: int = 1024) —> str   # one chat completion; greedy, token—
  capped


## Optimization Targets

Optimize the primary scalar score defined by the evaluator.

## Evaluator Code (shows how solutions are scored)

```python
[... 383 lines of program source ...]
```

The Seed Program, where the search starts (score: 0.380053)

```python
[... 67 lines of program source ...]
```
```

NOTE: This seed is the REFERENCE FRAME, not the program to improve. Every number in the search history below is stated as a delta from this program. The program you are improving is the Current Best Program shown next.

The Current Best Program (score: 0.607106, found at iteration 20)

This is the program your idea will be applied to. It is what the seed above became after the search; read it before proposing anything, because an idea it already implements is a wasted iteration.

```python
[... 120 lines of program source ...]
```

CRITICAL: ANALYZE THE CURRENT PROGRAM FIRST
Before suggesting new ideas, carefully analyze the Current Program above:
— What algorithm/approach does it use? (This is what's WORKING — score 0.607106)
— What are its strengths? (Why does it achieve this score?)
— What are its weaknesses? (What limits further improvement?)
— How can you improve it? (How to beat it?)

IMPORTANT: The program above is the CURRENT program that needs to be improved. Start by understanding what works, then suggest breakthrough ideas that build on or improve it.

Analysis Framework — Complete Before Generating Ideas

STEP 0: Understand the TASK (MOST IMPORTANT — DO THIS FIRST)
— What is the problem asking you to do?
— What is the goal or objective? (maximize, minimize, optimize)
— What are the inputs and outputs?
— What needs to be improved? (variables/decisions that affect the goal)
— What constraints exist?

STEP 1: Analyze the Evaluator Code
— How are solutions scored?
— What metrics are computed?
— What causes failures or penalties?

STEP 2: Identify Metrics
— What is the primary metric or Pareto objective set?
— How is it calculated?
— What secondary metrics exist?
— If variance/std is penalized, the program needs consistency across scenarios

STEP 3: Identify Constraints
— What conditions must be satisfied?
— What validation happens?
— What causes score penalties?

STEP 4: Identify Problem Structure
— Is processing sequential or global?
— Are decision variables discrete or continuous?
— What dependencies exist between decisions?
— **CRITICAL:** What data does your program receive vs what the evaluator uses?
— **CRITICAL:** How are metrics computed across components? Independently then aggregated, or jointly?

STEP 5: Determine Appropriate Approach
— Match approach to problem structure
— Consider what has worked vs failed before
— Identify promising library/technique combinations

STEP 6: Identify Improvement Opportunities
— What would increase each metric?
— What would satisfy constraints better?
— What fundamentally different approaches could work?

Current best score is 0.380053. Your ideas must improve the configured optimization targets and, in multiobjective mode, explicitly reason about objective trade—offs.

What the search has explored, measured from the seed

seed: combined_score 0.3801 llm_calls_per_query 9.0000 llm_tokens_per_query 34707.4032 oracle_ndcg 0.4813 retrieval_recall 0.4274 scoring_realization 0.7896

A named [Strategy] in a lineage is a paradigm; an unnamed one is ordinary mutation, which is how the search starts. Every figure is against the seed.

CURRENT BEST (combined_score 0.6071 at it20) —— a new idea gets applied to this program:

```
— seed → 3mut → [Reciprocal Rank Fusion (RRF) for Multi—Query Retrieval](it17) → [Reciprocal Rank
  Fusion (RRF) for Multi—Query Retrieval](it20)
      combined_score 0.3801 → 0.6071   (Δ +0.2271 from seed)
      llm_calls_per_query 9.0000 → 63.2742   llm_tokens_per_query 34707.4032 → 146786.0806   oracle_ndcg
  0.4813 → 0.8036   retrieval_recall 0.4274 → 0.7664   scoring_realization 0.7896 → 0.7555
      net change from seed: +_generate_queries, +_rrf_fuse, ~_parse_ranking, ~_rerank_window, —DEPTH, —
  STRIDE, —WINDOW (+rewired `rank`)
      _generate_queries:
          Generate 3 semantically related queries using LLM for multi—query retrieval.
          Uses LLM to expand the original query into related formulations that capture
          different terminology, improving retrieval recall across diverse biology concepts.
      _rrf_fuse:
          Fuse multiple BM25 result lists using Reciprocal Rank Fusion.
          Scores each document as sum of 1/(k + rank) across all queries.
          Higher scores indicate better overall ranking across queries.
      _parse_ranking:
          Extract ranking order from LLM output, prioritizing [n] brackets.
      _rerank_window:
          Rerank a window of candidates using LLM listwise comparison.

The 10 most recent lineages, oldest first:
— seed → 3mut → [Reciprocal Rank Fusion (RRF) for Multi—Query Retrieval](it17) → [Reciprocal Rank
  Fusion (RRF) for Multi—Query Retrieval](it20) → [Tournament—Style Pairwise Elimination for Top—10
  Ranking](it31)
      combined_score 0.3801 → 0.3810   (Δ +0.0010 from seed)
      llm_calls_per_query 9.0000 → 45.0000   llm_tokens_per_query 34707.4032 → 24599.2581   oracle_ndcg
  0.4813 → 0.5824   retrieval_recall 0.4274 → 0.4986   scoring_realization 0.7896 → 0.6543
      net change from seed: +_generate_queries, +_listwise_rank, +_llm_pairwise, +_pairwise_tournament, +
  _rrf_fuse, —_rerank_window, ~_parse_ranking, —DEPTH, —STRIDE, —WINDOW (+rewired `rank`)
      _listwise_rank:
          Listwise rank candidates using a single LLM comparison call.
          Takes a small set of candidates (~10) and asks the LLM to order them
          from most to least relevant using bracket notation.
      _llm_pairwise:
          Determine which of two documents is more relevant to the query.
          Uses LLM for direct pairwise comparison. Ties are broken deterministically
          by preferring Document A (which appears earlier in the pre—ranked list,
          serving as a BM25/RRF rank tiebreaker).
      _pairwise_tournament:
          Tournament—style pairwise elimination for top—10 ranking.
          Splits candidates into pairs, uses LLM to determine relevance,
          repeats knockout rounds until <=10 candidates remain, then listwise ranks them.
          This logarithmic approach minimizes LLM calls while maximizing top—10 precision.

— seed → 3mut → [Reciprocal Rank Fusion (RRF) for Multi—Query Retrieval](it17) → [Reciprocal Rank
  Fusion (RRF) for Multi—Query Retrieval](it20) → [Deterministic Query Expansion with Logarithmic RRF
  Fusion](it32)
      combined_score 0.3801 → 0.3837   (Δ +0.0036 from seed)
      llm_calls_per_query 9.0000 → 32.5161   llm_tokens_per_query 34707.4032 → 85195.2581   oracle_ndcg
  0.4813 → 0.4917   retrieval_recall 0.4274 → 0.4373   scoring_realization 0.7896 → 0.7803
      net change from seed: +_deterministic_expand, +_log_rrf_fuse, ~_parse_ranking, ~_rerank_window, —
  DEPTH, —STRIDE, —WINDOW (+rewired `rank`)
      _deterministic_expand:
          Expand query deterministically by extracting key terms and handling negations.
          Replaces LLM—based query generation with a lightweight, deterministic strategy
          that improves retrieval recall and oracle NDCG without LLM overhead.
      _log_rrf_fuse:
          Fuse multiple BM25 result lists using logarithmic RRF with original query weighting.
          Heavily weights documents from the original query while boosting expanded matches.
          Penalizes documents appearing only in expanded queries to maintain precision.

— seed → 3mut → [Reciprocal Rank Fusion (RRF) for Multi—Query Retrieval](it17) → [Reciprocal Rank
  Fusion (RRF) for Multi—Query Retrieval](it20) → [Batched Pointwise LLM Relevance Scoring with Structured
   JSON](it33)
      combined_score 0.3801 → 0.3911   (Δ +0.0110 from seed)
      llm_calls_per_query 9.0000 → 2.0000   llm_tokens_per_query 34707.4032 → 8821.9194   oracle_ndcg
  0.4813 → 0.6531   retrieval_recall 0.4274 → 0.6003   scoring_realization 0.7896 → 0.5988
      net change from seed: +_batch_score_candidates, +_generate_queries, +_rrf_fuse, —_parse_ranking, —
  _rerank_window, —DEPTH, —STRIDE, —WINDOW (+rewired `rank`)
      _batch_score_candidates:
          Score candidates by relevance using a single batched LLM call with JSON output.
          Evaluates each candidate on a 0—100 relevance scale using biology—specific
          criteria (accuracy, direct relevance, completeness, specificity). Returns
          candidates sorted by score descending. Single LLM call eliminates error
          propagation from window overlaps and reduces calls from ~60 to 1.

[... 7 more lineages ...]

## Technique Guidance
```

Note: Standard scientific libraries (scipy, numpy, etc.) are available. PyTorch and TensorFlow are not available.

For Continuous Optimization with Constraints:
— scipy.optimize.minimize with constraint handling (SLSQP, trust—constr)
— Multiple initial guesses for global optimization
— Geometric approaches (Voronoi, convex hull)

For Discrete/Combinatorial Problems:
— Greedy heuristics with good ordering
— Local search (swaps, moves)
— scipy.optimize.linear_sum_assignment for assignment problems
— scipy.optimize.linprog for linear constraints

For Graph/Network Problems:
— NetworkX algorithms (shortest path, min spanning tree, flow)
— Spectral methods (eigenvalue—based ordering)

For Repair/Reconstruction:
— Heuristic—based detection and correction
— Structural constraint exploitation
— Averaging/interpolation for consistency

For Robust Filtering/Noise Reduction:
— scipy.signal (medfilt, savgol_filter, wiener) for direct filtering
— Use methods that handle outliers better than mean—based (median, percentile)
— Do NOT use scipy.optimize.minimize to tune filter parameters
— Use filtering functions directly, not multi—stage optimization

General Principles:
— Prefer single—function library calls over multi—stage pipelines
— Match algorithm to problem structure
— Simple approaches with good heuristics often beat complex methods

ANTI—PATTERNS — Critical rules about what NOT to do

1. **Do NOT use multi—stage optimization**: Do NOT call one function then optimize its output. Deterministic setup code followed by a single optimization call is allowed.

2. **Do NOT use scipy.optimize.minimize for hyperparameter tuning**: Use minimize to solve the problem directly, NOT to tune parameters for another function.

3. **Do NOT use scipy.optimize.minimize for discrete problems**: Continuous optimizers cannot handle discrete constraint violations properly.

4. **Each idea MUST be a single—function library call**: Do NOT suggest multi—stage processing (e.g., "call A then call B").

AVOID: DEAP, genetic algorithm libraries, domain—specific complex libraries, custom research algorithms, or any library requiring additional `pip install`

Learning from Success:
When an approach succeeds, think: what principle made it work? Learn and think of better ideas, don't just add complexity. If breakthrough patterns are known, prioritize approaches that match them.

DIVERSITY REQUIREMENTS

Before generating ideas, explicitly think:
— Idea 1: [Type A — e.g., algorithmic refinement or library—based approach]
— Idea 2: [Type B — e.g., structural change or processing pattern — DIFFERENT from A]
— Idea 3: [Type C — e.g., different technique or optimization method — DIFFERENT from A and B]

Verify: Are these DIFFERENT types? NOT variations of the same approach.

Each idea must:
— Use DIFFERENT libraries/techniques than failed attempts
— Target DIFFERENT metrics/aspects from the evaluator
— Be independently implementable
— Prefer clear implementations (different != more complex)

Be Specific and Actionable

Not vague: "Try optimization"
Specific: "Use scipy.optimize.minimize with SLSQP method"

— Include exact library names, function names, methods, parameters
— Provide step—by—step implementation guide
— Focus on core logic that implements the idea correctly

```
- Handle edge cases and avoid errors/warnings
- For optimization: use multiple initializations, appropriate iteration counts and convergence criteria (
  evaluation timeout: {self.eval_timeout}s)

## Output Format

IMPORTANT: Respond with a JSON object containing exactly 3 idea objects under the "ideas" key.
Do not include code patches or diffs - describe strategies in natural language.

Generate 3 breakthrough ideas of DIFFERENT types.

Each idea must be a JSON object with these fields:
- "idea": Clear, direct description with library/technique name
- "description": Detailed implementation guide (5-10 sentences)
- "what_to_optimize": What metrics/areas to focus on
- "cautions": Important implementation details to watch for
- "approach_type": Exact "library.function" format (e.g., "scipy.optimize.minimize")

Diversity Requirement: Each idea must use a DIFFERENT approach type.
Do not generate variations of the same technique.

Return ONLY a JSON object in this shape: {"ideas": [ ... ]} with 3 paradigm objects. No other text.

Example:
```json
{
    "ideas": [
        {
            "idea": "Use scipy.optimize.minimize with SLSQP",
            "description": "Apply scipy.optimize.minimize directly to optimize all variables together...",
            "what_to_optimize": "primary evaluator score",
            "cautions": "Ensure constraints are properly formulated, use multiple starting points",
            "approach_type": "scipy.optimize.minimize"
        }
    ]
}
```
```

B PROGRAMS

B.1 SEED PROGRAM

Both searches initialize every run with this program. It retrieves the top 100 BM25 candidates and reranks them with a sliding-window listwise reranker (Sun et al., 2023), using windows of 20 with a stride of 10.

Program 1: Seed program

```python
"""Initial BRIGHT reranker - BM25 top-100 + RankGPT sliding-window baseline."""

import re

# Tunables - evolution may change these or remove them entirely.
DEPTH = 100     # BM25 candidates to rerank
WINDOW = 20     # sliding-window size
STRIDE = 10     # sliding-window step


def _parse_ranking(text, k):
    """Parse '[3] > [1] > ...' into 0-based indices: deduped, restricted to 0..k-1, with
    any omitted identifiers appended in original order (always a full permutation)."""
    order, seen = [], set()
    for m in re.findall(r"\[(\d+)\]", text or ""):
        idx = int(m) - 1
        if 0 <= idx < k and idx not in seen:
            seen.add(idx)
            order.append(idx)
    order.extend(i for i in range(k) if i not in seen)
    return order


def _rerank_window(query, window_ids, tools):
    """One listwise LLM call: reorder window_ids by relevance. Falls back to input order
```

```
    on any failure, so the window is always a valid permutation of its inputs."""
    passages = "\n\n".join(
        f"[{i + 1}] {tools.doc(cid)}" for i, cid in enumerate(window_ids)
    )
    system = "Rank the passages."
    user = (
        f"Query:\n{query}\n\n"
        f"Candidate passages:\n{passages}\n\n"
        f"Rank ALL {len(window_ids)} passages from most to least relevant to the query. "
        f"Respond ONLY with the ranking as identifiers in the format "
        f"[a] > [b] > [c] > ... using every identifier from 1 to {len(window_ids)} exactly once."
    )
    try:
        text = tools.llm(
            [{"role": "system", "content": system},
             {"role": "user", "content": user}],
            max_tokens=1024,
        )
        order = _parse_ranking(text, len(window_ids))
    except Exception:
        order = list(range(len(window_ids)))
    return [window_ids[i] for i in order]


def rank(query, tools):
    """Return doc_ids ranked most -> least relevant for `query`."""
    # First stage: BM25 candidate pool.
    ranked = tools.bm25(query, DEPTH)

    # Second stage: RankGPT listwise sliding window, bottom -> top so relevant docs
    # bubble upward through the overlap.
    window = min(WINDOW, len(ranked))
    end = len(ranked)
    while window and end > 0:
        start = max(0, end - window)
        ranked[start:end] = _rerank_window(query, ranked[start:end], tools)
        if start == 0:
            break
        end -= STRIDE
    return ranked
```

B.2 Evolved programs

For each objective, we show one of the final programs from biology with the highest train score. The docstrings and comments are written and maintained by the mutation model.

Quality-only objective. Query expansion with reciprocal rank fusion and a denser sliding window, reaching 0.633 nDCG@10 on test at 151k tokens per query.

Program 2: Evolved program, quality-only objective

```
"""BM25 retrieval with multi-query RRF fusion + sliding window LLM reranker."""

import re


def _generate_queries(query, tools):
    """Generate 3 semantically related queries using LLM for multi-query retrieval.

    Uses LLM to expand the original query into related formulations that capture
    different terminology, improving retrieval recall across diverse biology concepts.
    """
    prompt = (
        f"Original query: {query}\n\n"
        f"Generate exactly 3 related search queries for biology research.\n"
        f"Keep each query concise (3-8 words). Output one per line.\n"
        f"Format: 1. query1\n2. query2\n3. query3\n"
        f"Do not include any other text."
    )
    try:
        text = tools.llm(
            [{"role": "system", "content": "You are a helpful assistant."},
             {"role": "user", "content": prompt}],
            max_tokens=256
```

```
        )
        # Extract numbered queries
        lines = text.strip().split('\n')
        expanded = []
        for line in lines:
            line = line.strip()
            # Remove numbering
            m = re.match(r"\d+[\.\)]\s*(.+)", line)
            if m:
                expanded.append(m.group(1).strip())
            elif line and not line.startswith('#'):
                expanded.append(line)
        # Keep only first 3
        return [query] + expanded[:3]
    except Exception:
        return [query]


def _rrf_fuse(all_results, k=60):
    """Fuse multiple BM25 result lists using Reciprocal Rank Fusion.

    Scores each document as sum of 1/(k + rank) across all queries.
    Higher scores indicate better overall ranking across queries.
    """
    scores = {}
    for rank_list in all_results:
        for rank, doc_id in enumerate(rank_list, 1):
            scores[doc_id] = scores.get(doc_id, 0) + 1.0 / (k + rank)
    # Sort by score descending
    fused = sorted(scores.keys(), key=lambda x: -scores[x])
    return fused


def _parse_ranking(text, k):
    """Extract ranking order from LLM output, prioritizing [n] brackets."""
    order, seen = [], set()
    for m in re.findall(r"\[(\d+)\]", text or ""):
        idx = int(m) - 1
        if 0 <= idx < k and idx not in seen:
            seen.add(idx)
            order.append(idx)
    # Fill missing with original order to guarantee a valid permutation
    order.extend(i for i in range(k) if i not in seen)
    return order[:k]


def _rerank_window(query, window_ids, tools):
    """Rerank a window of candidates using LLM listwise comparison."""
    if not window_ids:
        return []
    passages = "\n\n".join(f"[{i+1}] {tools.doc(cid)}" for i, cid in enumerate(window_ids))
    prompt = (
        f"Query: {query}\n\n"
        f"Candidates:\n{passages}\n\n"
        f"Rank these {len(window_ids)} passages from most to least relevant.\n\n"
        f"Output ONLY the final ranking in this exact format: [a] > [b] > [c]\n"
        f"Use every identifier from 1 to {len(window_ids)} exactly once.\n"
        f"Do not include any other text."
    )
    try:
        text = tools.llm(
            [{"role": "system", "content": "You are a helpful assistant that ranks documents."},
             {"role": "user", "content": prompt}],
            max_tokens=1024
        )
        order = _parse_ranking(text, len(window_ids))
    except Exception:
        order = list(range(len(window_ids)))
    return [window_ids[i] for i in order]


def rank(query, tools):
    """Return doc_ids ranked most -> least relevant for `query`."""
    # Multi-query retrieval with RRF fusion
    queries = _generate_queries(query, tools)
    all_results = []
    for q in queries:
        results = tools.bm25(q, 100)
        all_results.append(results)
```

```
    # Fuse using RRF to boost recall and oracle NDCG
    ranked = _rrf_fuse(all_results, k=60)

    if not ranked:
        return []

    # Sliding window LLM reranker
    window, stride, end = 15, 5, len(ranked)
    while end > 0:
        start = max(0, end - window)
        ranked[start:end] = _rerank_window(query, ranked[start:end], tools)
        if start == 0:
            break
        end -= stride
    return ranked
```

Cost-aware objective. This program replaces the sliding window with a single listwise LLM call over the top 100 candidates. Before ranking, it asks the LLM which concepts the top ten BM25 results are missing and retrieves those concepts. It reaches 0.569 nDCG@10 on test with 2 LLM calls and 17.5k tokens per query, about a ninth of the tokens of the quality-only program.

Program 3: Evolved program, cost-aware objective

```
"""Single-call LLM reranker — BM25 top-100 with comprehensive ranking in one LLM call."""

import re


def _parse_ranking(text, k):
    """Parse '[3] > [1] > ...' into 0-based indices: deduped, restricted to 0..k-1, with
    any omitted identifiers appended in original order (always a full permutation)."""
    order, seen = [], set()
    for m in re.findall(r"\[(\d+)\]", text or ""):
        idx = int(m) - 1
        if 0 <= idx < k and idx not in seen:
            seen.add(idx)
            order.append(idx)
    order.extend(i for i in range(k) if i not in seen)
    return order


def rank(query, tools):
    """Coverage-Greedy Candidate Expansion with Conceptual Gap Analysis."""
    # 1. Initial BM25 retrieval
    initial = tools.bm25(query, 50)
    if not initial:
        return []

    # 2. Conceptual Gap Analysis
    # Preview top 10 candidates to identify missing biological concepts
    preview = "\n".join(f"[{i+1}] {tools.doc(d)}" for i, d in enumerate(initial[:10]))
    gap_prompt = (
        f"Query: {query}\n\n"
        f"Top retrieved docs cover:\n{preview}\n\n"
        f"Identify up to 3 important biological concepts missing from these results. "
        f"Return them as a comma-separated list of short search terms."
    )
    try:
        gap_resp = tools.llm([{"role": "user", "content": gap_prompt}], max_tokens=128)
        gap_terms = [t.strip().strip('"\'') for t in gap_resp.split(",") if t.strip()]
    except Exception:
        gap_terms = []

    # 3. Targeted retrieval for gaps
    gap_hits = []
    for term in gap_terms[:3]:
        gap_hits.extend(tools.bm25(term, 15))

    # 4. RRF Fusion
    rrf = {}
    for i, d in enumerate(initial):
        rrf[d] = rrf.get(d, 0) + 1.0 / (i + 60)
    for d in gap_hits:
        rrf[d] = rrf.get(d, 0) + 1.0 / 60.0
```

```
    fused = sorted(rrf.items(), key=lambda x: x[1], reverse=True)
    candidates = [d for d, _ in fused[:100]]

    if not candidates:
        return []

    # 5. LLM Comprehensive Ranking
    passages = "\n".join(f"[{i+1}] {tools.doc(d)}" for i, d in enumerate(candidates))
    prompt = (
        f"Query: {query}\n\n"
        f"Rank ALL {len(candidates)} passages from most to least relevant.\n"
        f"Passages:\n{passages}\n\n"
        f"Respond ONLY with the ranking in the format [1] > [2] > [3] ... "
        f"using every identifier from 1 to {len(candidates)} exactly once."
    )

    try:
        text = tools.llm(
            [{"role": "system", "content": "You are a relevance ranking assistant."},
             {"role": "user", "content": prompt}],
            max_tokens=1024,
        )
        order = _parse_ranking(text, len(candidates))
    except Exception:
        order = list(range(len(candidates)))

    return [candidates[i] for i in order]
```