SCALING THE INFERENCE PROGRAM: SYSTEM-LEVEL TEST-TIME SCALING FOR RECOMMENDATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Test-time scaling improves predictions without updating model weights, but it typically spends additional computation independently for each query. Recommendation exposes a distinct system-level opportunity: various requests repeatedly pass through the same frozen recommendation models, making the reusable, request-conditioned inference program a natural object on which to spend additional compute. We introduce *System-Level Test-Time Scaling* and instantiate it with the proposed framework PROGRAMREC. Starting from a blank program interface, a large language model iteratively constructs contract-valid inference programs that jointly specify feature construction F_π, a recursive execution graph G_π, and numerical parameters Θ_π. Aggregate validation feedback and the history of evaluated programs guide subsequent proposals. The selected program is then frozen and reused deterministically across requests without further LLM calls or model updates. Across three datasets, the strongest PROGRAMREC variant achieves the best overall performance, while larger discovery budgets yield gains in every setting. These results establish reusable inference-program construction as a new scaling axis for recommendation. The code is available at `https://anonymous.4open.science/r/ProgramRec-0675`.

1 INTRODUCTION

A recommender system is more than the models it contains: its predictions also depend on the inference program that governs how those models are used. Although its execution may adapt to each request, the program itself can remain fixed and reusable across the heterogeneous requests served by the system. Recent recommendation agents make this layer explicit by using large language models (LLMs) to organize the inference process for individual requests Yang et al. (2026b); Huang et al. (2025); Lao et al. (2026); Yang et al. (2026a). Yet their additional LLM computation is still performed as requests arrive, even when experience is retained across interactions. Conventional test-time scaling is likewise query-level, improving the current prediction through sampling, verification, or extended deliberation Wang et al. (2023); Lightman et al. (2024); Yao et al. (2023); Zhang et al. (2025b); Snell et al. (2025), without improving the inference procedure shared across future requests. Therefore, the recommendation highlights a distinct system-level scaling opportunity in which additional compute can be allocated to the reusable, request-conditioned inference program. We ask whether a pre-deployment discovery budget can construct such a program over frozen recommender systems and enable its reuse across subsequent requests without further LLM calls or model updates.

Constructing such a reusable inference program is challenging because discovery must search over the complete set of executable programs. Each candidate should jointly specify its feature construction, execution graph, and numerical parameters. Its quality is observed only after the entire program has been executed, with no direct supervision of its individual decisions. Moreover, the program must remain bounded and deterministic after discovery. Although frozen, it must still adapt its behavior to each request without making further LLM calls or updating the underlying models. The challenge is therefore to search this structured space with limited feedback and produce a reliable program for repeated deployment.

In this paper, we introduce PROGRAMREC, a framework for *System-Level Test-Time Scaling* in recommendation. PROGRAMREC allocates a one-time discovery budget to construct and improve

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.

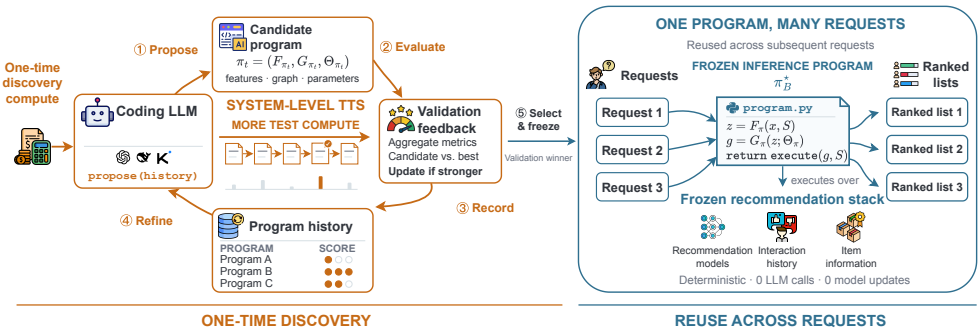


Figure 1: **Overview of our PROGRAMREC.** During one-time discovery, a coding LLM iteratively proposes complete executable programs, receives aggregate validation feedback, and uses the accumulated program history to refine subsequent proposals. The validation-selected program is then frozen and reused across requests. During deployment, it produces request-conditioned rankings using the frozen recommendation models without further LLM calls or model updates.

a shared inference program for a frozen recommender system. As illustrated in Figure 1, a coding LLM proposes complete executable programs that jointly specify feature construction, execution graphs, and numerical parameters. Each candidate is evaluated using an aggregate validation score, and its source and outcome are added to the discovery history. Subsequent proposals use this history to explore alternative structures and refine promising programs. When the budget is exhausted, the validation-selected program is frozen and reused across subsequent requests without further LLM calls or model updates. The coding LLM supplies the discovery compute, while the recommender system is the object being scaled.

Experiments across three recommendation datasets reveal three main findings. First, PROGRAMREC consistently improves the frozen recommender system, with performance increasing from $B = 1$ to the final discovery budget across every dataset–coding-LLM setting and nearly every individual trajectory. Second, discovery follows a formation-to-refinement pattern: trajectory feedback helps establish effective program structures, after which targeted parameter refinement yields further gains in a representative controlled analysis. Third, code-level transfer shows that the discovered program can remain effective across recommendation environments, while different coding LLMs produce distinct and competitive program families. Together, these results establish *System-Level Test-Time Scaling* as a new axis for improving recommender systems.

The contributions of this paper are as follows:

- We introduce System-Level Test-Time Scaling, which moves the scaling object from a request-specific execution trajectory to a reusable, request-conditioned inference program. This shift is particularly important for recommendation, where the resulting program can be reused across subsequent requests.
- We develop PROGRAMREC, a trajectory-guided discovery framework in which a coding LLM uses program history and aggregate validation feedback to construct executable, frozen inference programs. The validation-selected program is frozen and executed deterministically across subsequent requests without further LLM calls or model updates.
- We evaluate this framework across three recommendation datasets using three coding LLMs. The results demonstrate improvements with increasing discovery budgets, a representative formation-to-refinement pattern, and distinct reference program families, and cross-dataset transfer of discovered programs.

2 SYSTEM-LEVEL TEST-TIME SCALING FOR RECOMMENDATION

Recommendation setting. Let $x = (h_x, \mathcal{I}_x)$ denote a recommendation request, where $h_x = (i_1, \ldots, i_{|h_x|})$ is the observed interaction history and $\mathcal{I}_x$ is the set of candidate items. Let $i_x^+ \in \mathcal{I}_x$ denote the ground-truth next item following h_x. The system produces a top-K ranked list $\widehat{\mathbf{r}}_x =$

$(\widehat{i}_{x,1}, \ldots, \widehat{i}_{x,K})$, with the goal of ranking i_x^+ as highly as possible. We consider a frozen recommendation stack:

$$\mathcal{S} = (\mathcal{E}, \mathcal{D}), \qquad \mathcal{E} = \{e_j(\cdot; \theta_j)\}_{j=1}^{m}. \tag{1}$$

Here, $\mathcal{E}$ contains the available recommendation models, and $\mathcal{D}$ defines the user-side and item-side information accessible during inference. The model parameters and data interface remain fixed throughout discovery and deployment. A fixed execution contract $\mathcal{C}$ specifies the program interface, permitted operations, output format, and resource bounds. These rules define the space of executable inference programs

$$\Pi(\mathcal{C}) = \{\pi \mid \pi \text{ is executable under } \mathcal{C}\}. \tag{2}$$

Each $\pi \in \Pi(\mathcal{C})$ is a complete inference program containing a controller structure and its numerical parameters. For a request x, the program invokes components from $\mathcal{E}$ using information provided by $\mathcal{D}$ and produces

$$\widehat{\mathbf{r}}_x = \mathrm{Exec}_{\mathcal{C}}(\pi, x; \mathcal{S}). \tag{3}$$

The controller within π may construct candidate sets, route among recommendation models, compose their scores or ranks, and rerank the resulting items. Its decisions may vary with the current request, making the execution request-conditioned, while the complete program and its numerical parameters remain fixed and shared across requests.

From query-level to system-level scaling. Conventional test-time scaling allocates additional computation to the execution of an individual query after it arrives Zhang et al. (2025b). Query-level TTS, including algorithmically discovered allocation policies Zheng et al. (2026b), applies its inference budget within the execution trajectory of an individual query. System-Level Test-Time Scaling takes the shared inference program of a recommender system as the scaling object. It uses a pre-deployment discovery budget B to construct this program while keeping the recommendation stack fixed.

Let $\mathcal{A}_\psi$ denote the program-discovery policy implemented by a coding LLM with fixed parameters ψ. We define B as the maximum number of contract-valid programs that may be evaluated during discovery. Starting from the executable program space $\Pi(\mathcal{C})$, $\mathcal{A}_\psi$ constructs an adaptive sequence of candidate programs. Each proposal may depend on previously evaluated programs, their aggregate validation scores, and the current best program. The programs evaluated within budget B form the set

$$\mathcal{P}_{\mathcal{A}_\psi}(B) = \{\pi_1, \ldots, \pi_{T_B}\} \subseteq \Pi(\mathcal{C}), \qquad T_B \le B, \tag{4}$$

where T_B is the number of contract-valid programs evaluated before the budget is exhausted. Increasing B extends the discovery trajectory, allowing $\mathcal{A}_\psi$ to explore new program structures and refine previously discovered ones.

Let $\mathcal{V}$ denote the fixed validation environment and $\mathcal{M}_{\mathrm{val}}$ its ranking evaluator. The inference program returned under budget B is

$$\pi_B^\star \in \arg\max_{\pi \in \mathcal{P}_{\mathcal{A}_\psi}(B)} \mathcal{M}_{\mathrm{val}}(\pi; \mathcal{S}, \mathcal{V}). \tag{5}$$

The recommendation stack $\mathcal{S}$, execution contract $\mathcal{C}$, validation environment $\mathcal{V}$, evaluator $\mathcal{M}_{\mathrm{val}}$, and coding-LLM parameters ψ remain fixed as B varies. Increasing B changes the program-discovery trajectory and the resulting selection $\pi_B^\star$. System-Level Test-Time Scaling uses B to scale the construction of the shared inference program that governs subsequent requests. Appendix Table 4 summarizes the distinction between conventional query-level TTS and the system-level formulation studied here.

Scaling objective. Once selected, $\pi_B^\star$ is frozen and reused across deployment requests. Its execution remains request-conditioned, but its program structure and numerical parameters do not change, and serving requires no further coding-LLM calls or recommendation-model updates. Let $\mathbb{P}_{\mathrm{dep}}$ denote the distribution of unseen deployment requests and let m denote the target ranking metric. The deployment utility induced by discovery budget B is

$$\mathcal{J}(B) = \mathbb{E}_{(x, i_x^+) \sim \mathbb{P}_{\mathrm{dep}}}\left[m\left(\mathrm{Exec}_{\mathcal{C}}(\pi_B^\star, x; \mathcal{S}), i_x^+\right)\right]. \tag{6}$$

This objective makes the change in scaling object explicit. Discovery does not optimize the execution trace of a single request; it selects one request-conditioned program whose utility is defined over the deployment request distribution. The program remains shared across requests, even though its execution trace may vary from request to request.

3 PROGRAMREC: INFERENCE-PROGRAM DISCOVERY

Figure 1 illustrates PROGRAMREC, which constructs a trajectory of executable inference programs from a blank interface under discovery budget B. This section defines the program representation and describes the trajectory-guided discovery process.

3.1 EXECUTABLE PROGRAM SPACE

This section instantiates the executable program space $\Pi(\mathcal{C})$ by specifying how each program constructs derived features, organizes structural and compositional operators into a controller graph, and executes this graph for a recommendation request. Let $\mathbf{z}_x$ collect the frozen model outputs and the user- and item-side information exposed by $\mathcal{D}$ for request x. We represent a program as:

$$\pi = (F_\pi, G_\pi, \Theta_\pi), \qquad G_\pi = (V_\pi, E_\pi, r_\pi). \tag{7}$$

Here, F_π constructs the derived features used by the program, V_π is the set of operator nodes, $E_\pi \subseteq V_\pi \times V_\pi$ specifies their directed dependencies, and $r_\pi \in V_\pi$ is the root node whose output determines the final ranking. The collection $\Theta_\pi = \{\theta_v\}_{v \in V_\pi}$ contains the numerical parameters associated with the nodes.

The coding LLM constructs F_π as part of the program, determining which features are derived from $\mathbf{z}_x$ and where they enter the inference computation. These program-discovered features may modify item scores, control model composition, or determine request-dependent branch choices. The remaining computation is constructed from the fixed, typed operator vocabulary $\mathcal{O}_\mathcal{C}$ admitted by the execution contract:

$$\mathcal{O}_\mathcal{C} = \mathcal{O}_\mathcal{C}^{\mathrm{str}} \cup \mathcal{O}_\mathcal{C}^{\mathrm{comp}}, \qquad \mathrm{op}(v) \in \mathcal{O}_\mathcal{C} \quad \forall v \in V_\pi. \tag{8}$$

Structural operators determine candidate construction, branching, and multi-stage retrieval and reranking. Compositional operators normalize and coordinate model outputs through score or rank aggregation. These operators can be nested recursively, while features constructed by F_π can condition compatible composition or switching nodes.

For a request x, $\mathrm{Exec}_\mathcal{C}$ receives the system information $\mathbf{z}_x$ and evaluates G_π recursively from its leaves to its root. Let $\mathrm{ch}(v)$ denote the children of node v, and let $\mathbf{o}_v(x)$ denote the intermediate output produced at that node. The recursive evaluation is

$$\mathbf{o}_v(x) = g_{\mathrm{op}(v)}\left(\{\mathbf{o}_u(x) : u \in \mathrm{ch}(v)\}, \mathbf{z}_x; F_\pi, \theta_v\right), \tag{9}$$

where $g_{\mathrm{op}(v)}$ applies the fixed semantics of the selected operator using the inputs permitted by $\mathcal{C}$. Recursive evaluation continues until the root output is obtained:

$$\widehat{\mathbf{r}}_x = \mathrm{Exec}_\mathcal{C}(\pi, x; \mathcal{S}) = \mathrm{Rank}_K\left(\mathbf{o}_{r_\pi}(x)\right). \tag{10}$$

The execution trace may vary with x, while the complete program π remains fixed and shared across requests. Appendix A.1.3 specifies the program interface, operator semantics, and execution limits.

3.2 TRAJECTORY-GUIDED PROGRAM DISCOVERY

With the executable program space fixed, PROGRAMREC uses the discovery budget B to construct and evaluate a trajectory of complete inference programs. Each program simultaneously specifies its derived feature construction F_π, controller graph G_π, and operator parameters Θ_π. Request conditioning emerges when the program uses features constructed by F_π to determine a node's composition weights or branch choice. Accordingly, the complete executable program is the unit of discovery.

Discovery begins with the blank interface and an empty history $\mathcal{H}_0 = \emptyset$. At round t, the coding LLM $\mathcal{A}_\psi$ receives the fixed system interface, the execution contract $\mathcal{C}$, and the preceding history $\mathcal{H}_{t-1}$, and generates

$$\pi_t \sim \mathcal{A}_\psi(\cdot \mid \mathcal{S}, \mathcal{C}, \mathcal{H}_{t-1}), \qquad \pi_t \in \Pi(\mathcal{C}). \tag{11}$$

Here, π_t denotes the complete program constructed at round t. Its source specifies the feature construction F_{π_t}, controller graph G_{π_t}, and node parameters Θ_{π_t}. Before submission, the coding LLM may revise the source using contract-checking and target-free execution feedback. This feedback

reports executability violations and runtime diagnostics without accessing the validation evaluator. Once the source satisfies $\mathcal{C}$, it is submitted as a fixed program and evaluated once in the validation environment:

$$v_t = \mathcal{M}_{\text{val}}(\pi_t; \mathcal{S}, \mathcal{V}). \tag{12}$$

The score v_t measures the aggregate ranking performance of the complete program. It becomes available only after submission and is used to guide subsequent proposals.

The programs evaluated up to round t define a validation incumbent. Let $s_t^\star$ denote its discovery round:

$$s_t^\star \in \arg\max_{1 \le s \le t} v_s, \qquad \bar{\pi}_t = \pi_{s_t^\star}, \qquad \bar{v}_t = v_{s_t^\star}. \tag{13}$$

Thus, $\bar{\pi}_t$ is the strongest program available after spending t units of discovery budget. Individual proposals may improve or reduce validation performance, while the incumbent records the best program reached by the trajectory.

To make each evaluation interpretable in the context of the preceding search, PROGRAMREC summarizes its outcome relative to both the frozen reference and the previous incumbent. Let v_{ref} denote the validation score of the strongest frozen recommendation model. We define

$$\mathbf{f}_t = \left(v_t, \Delta_t^{\text{ref}}, \Delta_t^{\text{inc}}, u_t\right), \qquad \Delta_t^{\text{ref}} = v_t - v_{\text{ref}}, \tag{14}$$

where, for $t > 1$,

$$\Delta_t^{\text{inc}} = v_t - \bar{v}_{t-1}, \qquad u_t = \mathbb{I}\left[v_t > \bar{v}_{t-1}\right]. \tag{15}$$

For the first proposal, we set $\Delta_1^{\text{inc}} = 0$ and $u_1 = 1$. Here, Δ_t^{ref} indicates whether the candidate improves upon the strongest individual recommender, Δ_t^{inc} measures its change from the preceding best program, and u_t records whether it becomes the new incumbent.

The complete program and its feedback are then appended to the ordered discovery history:

$$\mathcal{H}_t = \mathcal{H}_{t-1} \oplus (\pi_t, \mathbf{f}_t) = \left((\pi_s, \mathbf{f}_s)\right)_{s=1}^{t}. \tag{16}$$

The updated history $\mathcal{H}_t$ is provided to the coding LLM when constructing the next proposal π_{t+1}. After the available budget has been consumed, the output of discovery is the incumbent reached at the final evaluated round:

$$\pi_B^\star = \bar{\pi}_{T_B}, \tag{17}$$

where T_B is the number of contract-valid programs evaluated within budget B. This operationally realizes the selection objective in Equation 5. Increasing B extends the source–outcome trajectory and creates additional opportunities to construct a stronger program. System-Level Test-Time Scaling, therefore, acts on the quality of the shared inference program reached by discovery.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Datasets and evaluation. We evaluate PROGRAMREC on three public recommendation datasets: Amazon Movies and TV, GoodReads, and Yelp. Following prior work Kang & McAuley (2018); Tang et al. (2026); Yang et al. (2026b), we formulate each dataset as a temporal next-item recommendation task, in which the system ranks the held-out next item among the candidate items based on the observed interaction history. For each discovery trajectory, we select the candidate inference program with the highest validation NDCG@10 as the final program $\pi_B^\star$. We report $\mathrm{NDCG@}K$ and $\mathrm{Recall@}K$ for $K \in \{5, 10\}$. Dataset sources, preprocessing, split construction, and complete statistics are provided in Appendix A.2.1.

Baselines. For each dataset, we instantiate the frozen stack $\mathcal{S} = (\mathcal{E}, \mathcal{D})$ with three pretrained recommendation models: SASRec Kang & McAuley (2018), LightGCN He et al. (2020), and SimpleX Mao et al. (2021). We compare against these three individual models, validation-selected StaticFusion, a matched implementation of RouteRec Zhou et al. (2026), RecToolAgent Yang et al. (2026b), and Random Search over the same executable program space $\Pi(\mathcal{C})$. All methods use the same processed splits, frozen recommendation models, candidate sets, and ranking evaluator. The contract exposes the same interaction histories, item information, and frozen-model outputs to PROGRAMREC and Random Search. Detailed baseline configurations are provided in Appendix A.2.2.

Table 1: Overall recommendation performance at $K \in \{5, 10\}$. Each PROGRAMREC variant reports means over three independent trajectories with the same evaluated-program budget $B = 10$. Best and second-best means are bolded and underlined, respectively.

	Amazon Movies and TV				GoodReads				Yelp			
	NDCG		Recall		NDCG		Recall		NDCG		Recall	
Method	@5	@10	@5	@10	@5	@10	@5	@10	@5	@10	@5	@10
SASRec	.0164	.0220	.0247	.0417	.0408	.0545	.0630	.1055	.0224	.0274	.0325	.0481
LightGCN	.0090	.0134	.0133	.0266	.0261	.0353	.0415	.0705	.0278	.0349	.0418	.0639
SimpleX	.0098	.0136	.0133	.0247	.0207	.0280	.0321	.0549	.0292	.0371	.0425	.0672
StaticFusion	.0210	.0278	.0313	.0531	.0467	.0621	.0712	.1189	.0374	.0462	.0539	.0816
RouteRec	.0111	.0157	.0161	.0304	.0281	.0378	.0462	.0768	.0337	.0405	.0488	.0700
RecToolAgent	.0173	.0260	.0323	.0588	.0499	.0644	.0793	.1236	.0248	.0408	.0425	.0614
Random Search	.0186	.0263	.0266	.0503	.0499	.0646	.0769	.1227	.0327	.0406	.0470	.0717
PROGRAMREC (DeepSeek-V4-Pro)	**.0369**	**.0427**	**.0493**	**.0674**	**.0987**	**.1128**	**.1376**	**.1813**	**.0395**	.0484	**.0564**	.0843
PROGRAMREC (GPT-5.6 Luna)	.0305	.0373	.0414	.0623	.0745	.0894	.1075	.1533	.0356	.0435	.0524	.0772
PROGRAMREC (Kimi K2.7 Code)	.0270	.0316	.0380	.0525	.0904	.1035	.1264	.1673	.0392	**.0489**	.0555	**.0856**

Implementation details. We implement all methods using the same frozen model scores and ranking evaluator. For PROGRAMREC, we instantiate the coding LLM $\mathcal{A}_\psi$ with DeepSeek-V4-Pro Xu et al. (2026), GPT-5.6 Luna OpenAI (2026), and Kimi K2.7 Code Team et al. (2025). We run three independent discovery trajectories for each of the nine dataset and coding-LLM pairs. Each trajectory starts from the blank interface and an empty history, evaluates up to $B = 10$ contract-valid programs, and selects its validation incumbent according to Equation 5. Each selected program is frozen before test evaluation, and we report the mean test performance over the three trajectories. Frozen execution requires no coding-LLM calls or recommendation-model updates. Baseline configurations and additional implementation details are provided in Appendix Sections A.2.2 and A.2.3, respectively.

4.2 MAIN RESULTS

Table 1 supports three conclusions. First, effective coordination of multiple frozen recommenders improves upon relying on any individual model. The strongest baseline improves NDCG@10 over the strongest individual recommender by 26.4%, 18.5%, and 24.5% on Amazon, GoodReads, and Yelp, respectively. This consistent advantage shows that recommendation quality depends not only on the underlying models, but also on how their outputs and the available information are organized during inference. Second, PROGRAMREC discovers substantially stronger programs within this executable space. It achieves the best mean performance in all reported dataset-metric settings. Compared with the strongest baseline on each dataset, NDCG@10 improves by 53.6%, 74.6%, and 5.8%, respectively. Moreover, every PROGRAMREC configuration outperforms the Random Search under the same per-trajectory budget. This result shows that trajectory-guided discovery contributes beyond the expressiveness of the program space itself. Third, the coding LLM shapes the outcome of program discovery. DeepSeek-V4-Pro achieves the strongest mean performance on Amazon and GoodReads and leads two metrics on Yelp, while Kimi K2.7 performs best on Yelp. With the program interface, evaluator, and discovery budget held fixed, these differences show that coding LLMs vary in how effectively they construct and refine inference programs from trajectory feedback.

4.3 SCALING WITH DISCOVERY COMPUTE

Figure 2 reports the frozen test utility induced by $B \mapsto \pi^\star_B$, averaged over three independent discovery trajectories per dataset–coding-LLM pair. Increasing B extends the discovery trajectory and makes additional complete programs available for validation selection. Each newly constructed program can revise the derived feature construction F_π, controller graph G_π, and operator parameters Θ_π. The selected $\pi^\star_B$ can therefore change its feature use, structural organization, and composition policy as discovery compute increases.

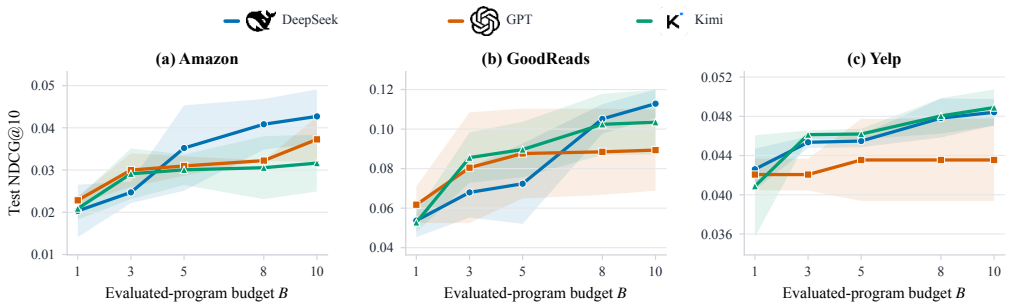


Figure 2: **Scaling the inference program with discovery compute.** At each budget B, the validation incumbent $\pi_B^\star$ is frozen and evaluated on test. Curves report the mean test NDCG@10 across three independent discovery trajectories per dataset and coding-LLM pair, and shaded bands show $\pm$ sample standard deviation.

This additional discovery compute produces stronger frozen programs. Mean test NDCG@10 increases from the initial to the final budget in every setting, and nearly every individual trajectory finishes above its starting point, with a median endpoint gain of 61.2%. The shaded bands indicate one-sample standard deviation and reveal meaningful variation in both the magnitude and timing of improvement, particularly at intermediate budgets. Nevertheless, the upward trend in the endpoint holds across datasets and coding LLMs. DeepSeek-V4-Pro continues to benefit from later discovery rounds, GPT-5.6 often obtains most of its gains earlier, and Kimi K2.7 follows a dataset-dependent progression. These results establish a budget-to-program scaling relation. The additional one-time discovery compute expands the trajectory of complete executable programs and produces a stronger reusable inference program.

4.4 Ablation Study

We study the capabilities realized in the programs constructed by coding LLMs using deterministic post-discovery transformations. For each dataset, we freeze the discovered program and rewrite the nodes or parameters associated with one capability. The feature variants suppress the use of user-profile, confidence-and-agreement, or derived context features. The structural variant removes the candidate-processing and reranking stages present in the program. The composition and control variants disable request conditioning, replace multi-model coordination with the strongest frozen recommender, or replace rank-based composition with score-based composition. No rediscovery is performed, and all variants are evaluated on the same validation split.

Table 2 identifies derived context features and model coordination as the dominant contributors across the three frozen programs. Removing the former weakens the contextual information incorporated by the realized feature construction, while replacing multi-model coordination with a single frozen recommender produces the largest overall degradation. Multi-stage execution also provides consistent gains, showing that the organization of candidate processing and reranking contributes beyond feature construction and model composition. The remaining transformations produce smaller and dataset-dependent changes. Request conditioning, user-profile features, confidence-and-agreement features, and rank consensus contribute differently within the three discovered programs

Table 2: Component ablations of the DeepSeek-V4-Pro programs. The full row reports validation NDCG@10, and the remaining rows report absolute changes. Blue, orange, and green denote feature-construction, structural, and composition-and-control ablations, respectively.

Program variant	Amazon	GoodReads	Yelp
Full ProgramRec	.0624	.1252	.0560
w/o User Profile	−.0001	+.0004	−.0005
w/o Confidence/Agreement	+.0036	−.0002	−.0032
w/o Derived Context Features	−.0233	−.0558	−.0083
w/o Multi-Stage Structure	−.0028	−.0099	−.0002
w/o Request Conditioning	+.0032	−.0002	−.0029
w/o Model Coordination	−.0259	−.0566	−.0148
w/o Rank Consensus	+.0035	+.0000	+.0000

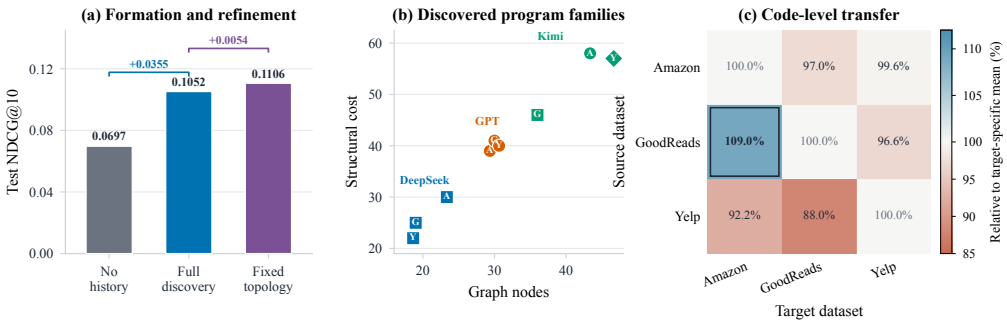


Figure 3: **Understanding program discovery and reuse.** **(a)** GoodReads test NDCG@10 for discovery without history, full discovery, and fixed-topology refinement in one analyzed DeepSeek trajectory; brackets show paired improvements. **(b)** Node count and structural cost of nine reference programs, one per dataset–coding-LLM pair. Colors denote coding LLMs; squares, circles, and diamonds denote `CASCADE`, `SWITCH`, and `CANDIDATE_UNION` roots; letters identify datasets. **(c)** For each source dataset, three DeepSeek programs from independent trajectories are frozen and executed on every target test split. Cells report mean NDCG@10 relative to the target-specific mean, and off-diagonal cells that exceed it are outlined.

because their effects depend on the surrounding graph and operator parameters. Since these transformations are applied without rediscovery, the results measure how each frozen program uses its realized components, rather than the best performance attainable after removing a capability from the program space.

4.5 Understanding Program Discovery and Reuse

The preceding results establish the effectiveness of the discovered programs. We next examine how trajectory history shapes their formation, what program families emerge from different coding LLMs, and whether the resulting program logic can be reused across datasets.

Program formation and refinement. Figure 3(a) analyzes one DeepSeek trajectory and shows how history contributes to program formation. Removing the sources and validation outcomes of earlier proposals reduces test NDCG@10 from 0.1052 to 0.0697, corresponding to a paired improvement of 0.0355 for full discovery. After a strong topology has been discovered, restricting a separate search to its numerical parameters further increases NDCG@10 to 0.1106, an improvement of 0.0054. In this analyzed trajectory, discovery therefore progresses from constructing an effective inference structure to extracting additional performance through targeted numerical refinement. The substantially larger formation gain further indicates that, in this case, history contributes mainly by establishing a useful structure before numerical refinement becomes effective.

Discovered program families. Each reference program's graph has a root node r_π, defined in Equation 7, whose output determines the final ranking. Figure 3(b) compares the root operators of reference programs, one per dataset–coding-LLM pair. Within this reference set, DeepSeek-V4-Pro produces compact `CASCADE` programs, GPT-5.6 consistently uses `SWITCH`, and Kimi K2.7 Code yields larger and more structurally diverse programs spanning `SWITCH`, `CASCADE`, and `CANDIDATE_UNION`. These patterns show that different coding LLMs compose the same executable vocabulary into distinct program families.

Code-level transfer. Figure 3(c) evaluates whether the discovered program logic remains useful across datasets. For each source dataset, we freeze programs from three independent DeepSeek-V4-Pro trajectories and execute their unmodified code across all three target test splits via the corresponding interfaces. Across the six off-diagonal transfers, the source-program means retain 88.0% to 109.0% of the target-specific mean test NDCG@10. These results demonstrate that the discovered code can capture coordination strategies whose utility extends beyond the dataset that produced them.

4.6 Efficiency Analysis

ProgramRec separates one-time program discovery from repeated deployment. We evaluate both stages using provider API calls during discovery and the contract-defined structural cost of the selected program at deployment. Table 3 reports the mean and range across three trajectories for each dataset–coding-LLM pair. Mean discovery usage ranges from 80.7 to 108.0 API calls, while mean structural cost ranges from 23.0 to 56.3. The two measures reveal different efficiency profiles. Kimi uses the fewest API calls on every dataset, yet produces the lowest-cost program on Amazon and the highest-cost programs on GoodReads and Yelp. The most compact programs on GoodReads and Yelp are instead produced by DeepSeek and GPT, respectively. Structural cost also varies across trajectories under the same coding LLM, reflecting differences among the selected program structures. Thus, discovery usage does not directly determine deployment complexity, and no coding LLM yields uniformly compact programs across datasets.

Table 3: **Discovery and deployment efficiency.** Provider API calls and frozen-program structural costs across trajectories. Values are reported as mean [minimum, maximum].

Dataset	Coding LLM	Calls	$\mathrm{Cost}_{\mathcal{C}}$
Amazon	DeepSeek	93.0 [89, 95]	50.3 [30, 63]
Amazon	GPT	108.0 [89, 130]	47.7 [39, 54]
Amazon	Kimi	81.3 [78, 86]	38.7 [20, 58]
GoodReads	DeepSeek	85.0 [81, 88]	29.0 [25, 35]
GoodReads	GPT	97.3 [93, 105]	32.7 [23, 41]
GoodReads	Kimi	80.7 [71, 94]	56.3 [46, 64]
Yelp	DeepSeek	103.7 [99, 108]	42.7 [22, 59]
Yelp	GPT	100.3 [97, 103]	23.0 [11, 40]
Yelp	Kimi	86.0 [79, 99]	53.3 [40, 63]

5 Related Work

Test-time scaling improves a fixed model by allocating additional computation after an input arrives Zhang et al. (2025b); Snell et al. (2025). Existing methods expand longer or multiple reasoning trajectories Wei et al. (2022); Wang et al. (2023); Brown et al. (2024); Muennighoff et al. (2025), verify or refine intermediate solutions Zhou et al. (2024b); Lightman et al. (2024); Madaan et al. (2023); Shinn et al. (2023), or explore structured reasoning paths Yao et al. (2023); Zhou et al. (2024a); Zheng et al. (2026a), while complementary work studies how the inference budget should be allocated Wu et al. (2025). These methods scale trajectories generated by a specified inference strategy. Agentic discovery makes the strategy itself searchable by using large language models and evaluation feedback to construct executable agent designs and code-represented workflows Hu et al. (2025); Zhang et al. (2025a). AutoTTS applies this idea to discover controllers that allocate inference compute across reasoning branches for individual queries Zheng et al. (2026b). In recommendation, large language models have been developed as unified or instruction-tuned recommenders Lin et al. (2025); Geng et al. (2022); Bao et al. (2023), rankers and rerankers Hou et al. (2024); Gao et al. (2025); Luo et al. (2025), and sources of augmented interaction graphs and recommendation features Cui et al. (2025); Wei et al. (2024). Agentic systems further use language models to plan, invoke recommendation tools, maintain user context, and coordinate specialized roles Wang et al. (2024a;b); Huang et al. (2025); Yang et al. (2026b); Lao et al. (2026); Yang et al. (2026a). Recommendation-specific test-time methods merge predictions from diverse models Lyu et al. (2026) or adapt a sequential recommender using retrieved interactions Tang et al. (2026). ProgramRec realizes System-Level TTS through pre-deployment agentic discovery of a shared inference program for a frozen recommendation stack.

6 Conclusion

We introduced *System-Level Test-Time Scaling* and instantiated it with ProgramRec, a framework that uses a coding LLM, aggregate validation feedback, and program history to construct reusable inference programs for frozen recommender systems. Each discovered program jointly specifies feature construction, an execution graph, and numerical parameters, and remains request-conditioned after discovery. Across three datasets, ProgramRec outperforms strong baselines and improves with discovery budget. Selected programs are reused deterministically without LLM calls or model updates, establishing inference-program construction as a new scaling axis for recommendation.

AI USE STATEMENT

Generative AI tools were used to improve clarity and readability and to assist with writing and debugging research code. The research ideas, methodology, experimental design, and interpretation of results were developed by the authors. All AI-assisted text was reviewed and revised by the authors, and all AI-assisted code was inspected and tested before use. The coding LLMs evaluated as components of PROGRAMREC and their roles in the experiments are documented in the main paper. The authors take full responsibility for the final content of this work.

REFERENCES

Keqin Bao, Jizhi Zhang, Yang Zhang, Wenjie Wang, Fuli Feng, and Xiangnan He. Tallrec: An effective and efficient tuning framework to align large language model with recommendation. In *Proceedings of the 17th ACM Conference on Recommender Systems*, pp. 1007–1014, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702419. doi: 10.1145/3604915.3608857. URL https://doi.org/10.1145/3604915.3608857.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Ziqiang Cui, Yunpeng Weng, Xing Tang, Xiaokun Zhang, Shiwei Li, Peiyang Liu, Bowei He, Dugang Liu, Weihong Luo, Xiuqiang He, and Chen Ma. Semantic retrieval augmented contrastive learning for sequential recommendation. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 162621–162645. Curran Associates, Inc., 2025. doi: 10.52202/085713-5429. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/ee2841db84cd09a5f6e3e313ce3d79d9-Paper-Conference.pdf.

Jingtong Gao, Bo Chen, Xiangyu Zhao, Weiwen Liu, Xiangyang Li, Yichao Wang, Wanyu Wang, Huifeng Guo, and Ruiming Tang. Llm4rerank: Llm-based auto-reranking framework for recommendations. In *Proceedings of the ACM on Web Conference 2025*, pp. 228–239, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400712746. doi: 10.1145/3696410.3714922. URL https://doi.org/10.1145/3696410.3714922.

Shijie Geng, Shuchang Liu, Zuohui Fu, Yingqiang Ge, and Yongfeng Zhang. Recommendation as language processing (rlp): A unified pretrain, personalized prompt & predict paradigm (p5). In *Proceedings of the 16th ACM Conference on Recommender Systems*, pp. 299–315, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392785. doi: 10.1145/3523227.3546767. URL https://doi.org/10.1145/3523227.3546767.

Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, YongDong Zhang, and Meng Wang. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '20, pp. 639–648, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380164. doi: 10.1145/3397271.3401063. URL https://doi.org/10.1145/3397271.3401063.

Yupeng Hou, Junjie Zhang, Zihan Lin, Hongyu Lu, Ruobing Xie, Julian McAuley, and Wayne Xin Zhao. Large language models are zero-shot rankers for recommender systems. In *Advances in Information Retrieval: 46th European Conference on Information Retrieval, ECIR 2024, Glasgow, UK, March 24–28, 2024, Proceedings, Part II*, pp. 364–381, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-56059-0. doi: 10.1007/978-3-031-56060-6_24. URL https://doi.org/10.1007/978-3-031-56060-6_24.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/36b7acf6f6010652b3f2a433774a66fe-Paper-Conference.pdf.

Xu Huang, Jianxun Lian, Yuxuan Lei, Jing Yao, Defu Lian, and Xing Xie. Recommender ai agent: Integrating large language models for interactive recommendations. *ACM Trans. Inf. Syst.*, 43(4), June 2025. ISSN 1046-8188. doi: 10.1145/3731446. URL https://doi.org/10.1145/3731446.

Wang-Cheng Kang and Julian McAuley. Self-Attentive Sequential Recommendation . In *2018 IEEE International Conference on Data Mining (ICDM)*, pp. 197–206, Los Alamitos, CA, USA, 2018. IEEE Computer Society. doi: 10.1109/ICDM.2018.00035.

Changxin Lao, Fei Pan, Guozhuang Ma, Han Li, Huihuang Lin, Jijun Shi, Kangzhi Zhao, Kun Gai, Mo Zhou, Qinqin Zhou, et al. Agentx: Towards agent-driven self-iteration of industrial recommender systems. *arXiv preprint arXiv:2606.26859*, 2026.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 39578–39601, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/aca97732e30bcf1303bc22ac3924fd16-Paper-Conference.pdf.

Jianghao Lin, Xinyi Dai, Yunjia Xi, Weiwen Liu, Bo Chen, Hao Zhang, Yong Liu, Chuhan Wu, Xiangyang Li, Chenxu Zhu, et al. How can recommender systems benefit from large language models: A survey. *ACM Transactions on Information Systems*, 43(2):1–47, 2025.

Sichun Luo, Bowei He, Haohan Zhao, Wei Shao, Yanlin Qi, Yinya Huang, Aojun Zhou, Yuxuan Yao, Zongpeng Li, Yuanzhang Xiao, Mingjie Zhan, and Linqi Song. Recranker: Instruction tuning large language model as ranker for top-k recommendation. *ACM Trans. Inf. Syst.*, 43(5), July 2025. ISSN 1046-8188. doi: 10.1145/3705728. URL https://doi.org/10.1145/3705728.

Fuyuan Lyu, Zhentai Chen, Jingyan Jiang, Lingjie Li, Xing Tang, Xiuqiang He, and Xue Liu. Exploring test-time scaling via prediction merging on large-scale recommendation. In *Proceedings of the 49th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 1301–1312, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400725999. doi: 10.1145/3805712.3809740. URL https://doi.org/10.1145/3805712.3809740.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=S37hOerQLB.

Kelong Mao, Jieming Zhu, Jinpeng Wang, Quanyu Dai, Zhenhua Dong, Xi Xiao, and Xiuqiang He. Simplex: A simple and strong baseline for collaborative filtering. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, CIKM '21, pp. 1243–1252, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384469. doi: 10.1145/3459637.3482297. URL https://doi.org/10.1145/3459637.3482297.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling, 2025. URL https://arxiv.org/abs/2501.19393.

OpenAI. GPT-5.6 System Card. Technical report, OpenAI, July 2026. URL https://deploymentsafety.openai.com/gpt-5-6/gpt-5-6.pdf.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=vAElhFcKW6.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=4FWAwZtd2n.

Xing Tang, Ziqiang Cui, Jingyang Bin, Xiaokun Zhang, Fuyuan Lyu, Jingyan Jiang, Dugang Liu, Chen Ma, and Xiuqiang He. Retrieve-then-adapt: Retrieval-augmented test-time adaptation for sequential recommendation, 2026. URL https://arxiv.org/abs/2604.05379.

Kimi Team, Yifan Bai, Yiping Bao, Y Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1PL1NIMMrw.

Yancheng Wang, Ziyan Jiang, Zheng Chen, Fan Yang, Yingxue Zhou, Eunah Cho, Xing Fan, Yanbin Lu, Xiaojiang Huang, and Yingzhen Yang. RecMind: Large language model powered agent for recommendation. In Kevin Duh, Helena Gomez, and Steven Bethard (eds.), *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 4351–4364, Mexico City, Mexico, June 2024a. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-naacl.271. URL https://aclanthology.org/2024.findings-naacl.271/.

Zhefan Wang, Yuanqing Yu, Wendi Zheng, Weizhi Ma, and Min Zhang. Macrec: A multi-agent collaboration framework for recommendation. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2760–2764, New York, NY, USA, 2024b. Association for Computing Machinery. ISBN 9798400704314. doi: 10.1145/3626772.3657669. URL https://doi.org/10.1145/3626772.3657669.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=_VjQlMeSB_J.

Wei Wei, Xubin Ren, Jiabin Tang, Qinyong Wang, Lixin Su, Suqi Cheng, Junfeng Wang, Dawei Yin, and Chao Huang. Llmrec: Large language models with graph augmentation for recommendation. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining*, pp. 806–815, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400703713. doi: 10.1145/3616855.3635853. URL https://doi.org/10.1145/3616855.3635853.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference scaling laws: An empirical analysis of compute-optimal inference for LLM problem-solving. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=VNckp7JEHn.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Hanchen Yang, Kaiwen Yang, Junpeng Zhuang, Yang He, Keting Cen, Bochao Liu, Zhongbo Sun, An Liu, Zhongteng Han, and Chenyi Lei. Sr-agent: An experience-driven agentic framework for post-ranking strategy refinement in e-commerce recommendation, 2026a. URL https://arxiv.org/abs/2607.17719.

Mingdai Yang, Zhiwei Liu, Weizhi Zhang, Yibo Wang, Hao Peng, and Philip Yu. Personalized recommendation tool learning via autonomous language agents, 2026b. URL https://arxiv.org/abs/2607.19739.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 11809–11822. Curran Associates, Inc., 2023. doi: 10.52202/075280-0517. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/271db9922b8d1f4dd7aaef84ed5ac703-Paper-Conference.pdf.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=z5uVAKwmjf.

Qiyuan Zhang, Fuyuan Lyu, Zexu Sun, Lei Wang, Weixu Zhang, Zhihan Guo, Yufei Wang, Niklas Muennighoff, Irwin King, Xue Liu, and Chen Ma. What, how, where, and how well? a survey on test-time scaling in large language models, 2025b. URL https://arxiv.org/abs/2503.24235.

Tong Zheng, Chengsong Huang, Runpeng Dai, Yun He, Rui Liu, Xin Ni, Huiwen Bao, Kaishen Wang, Hongtu Zhu, Jiaxin Huang, et al. Parallel-probe: Towards efficient parallel thinking via 2d probing. *arXiv preprint arXiv:2602.03845*, 2026a.

Tong Zheng, Haolin Liu, Chengsong Huang, Huiwen Bao, Sheng Zhang, Rui Liu, Runpeng Dai, Ruibo Chen, Chenxi Liu, Tianyi Xiong, et al. Llms improving llms: Agentic discovery for test-time scaling. *arXiv preprint arXiv:2605.08083*, 2026b.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Forty-first International Conference on Machine Learning*, 2024a. URL https://openreview.net/forum?id=njwv9BsGHF.

Aojun Zhou, Ke Wang, Zimu Lu, Weikang Shi, Sichun Luo, Zipeng Qin, Shaoqing Lu, Anya Jia, Linqi Song, Mingjie Zhan, and Hongsheng Li. Solving challenging math word problems using GPT-4 code interpreter with code-based self-verification. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=c8McWs4Av0.

Kaiji Zhou, Vladimir Kalmykov, and Yue Feng. Routerec: Strict evaluation of recommender-agent selection and aggregation, 2026. URL https://arxiv.org/abs/2607.09908.

A APPENDIX

A.1 METHOD DETAILS

This section first distinguishes query-level from system-level test-time scaling and then details the executable program interface, feature construction, operator vocabulary, proposal validation, and complete discovery procedure of PROGRAMREC.

A.1.1 QUERY-LEVEL AND SYSTEM-LEVEL TEST-TIME SCALING

Table 4 summarizes the distinction between conventional query-level test-time scaling and the system-level formulation studied in this work. The central difference lies in the object produced by additional computation: a request-specific execution trajectory or a reusable inference program.

Table 4: **Query-level and system-level test-time scaling.** Conventional query-level TTS spends additional computation within an individual request, whereas System-Level TTS constructs a reusable inference program before deployment.

Dimension	Conventional query-level TTS	System-Level TTS in PROGRAMREC
Scaling object	Execution trajectory for an individual request	Inference program shared across requests
Compute allocation	After each request arrives	During pre-deployment program discovery
Optimization target	Outcome of the current request	Expected utility over the deployment request distribution
Scaling artifact	Request-specific prediction or execution trace	Frozen validation-selected program $\pi_B^\star$
Future requests	Additional scaling computation is performed again	The selected program is reused without rediscovery
Request adaptation	A separate trajectory is constructed for each request	The shared program performs request-conditioned execution
Deployment behavior	Extra scaling computation is incurred per request	The frozen program executes without coding-LLM calls or model updates

A.1.2 EXECUTABLE PROGRAM INTERFACE

PROGRAMREC represents each candidate as a complete inference program $\pi = (F_\pi, G_\pi, \Theta_\pi)$ encoded by a Python module containing exactly one class named `RecommendationController`. The evaluator owns the frozen recommendation stack, the evaluation data, and the ranking metric. The generated program determines how inference-time information is transformed into program-specific features by F_π, how frozen recommendation models and derived features are organized into the recursive execution graph G_π, and which numerical parameters Θ_π govern its execution.

Every candidate implements the following interface:

```
class RecommendationController:
    def __init__(self, public_schema, beta, seed): ...
    def initial_state(self): ...
    def recommend(self, state, observation): ...
    def get_last_trace(self): ...
```

The constructor receives the public system schema, the structural-cost bound, and a deterministic seed. The `initial_state` method returns a bounded JSON state, while `recommend` returns a contract-valid recursive graph whose nodes encode the structure and parameters of the inference computation. Feature-construction routines and their parameters are defined by the same generated module and are executed as part of the complete program. The `get_last_trace` method returns

bounded diagnostics used during program checking. The source code and numerical constants remain fixed after discovery, although the features, intermediate node outputs, and routing decisions realized during execution may depend on the current request.

Discovery begins from the following blank source:

```
from __future__ import annotations

class RecommendationController:
    def __init__(self, public_schema, beta, seed):
        pass

    def initial_state(self):
        raise NotImplementedError("blank discovery template")

    def recommend(self, state, observation):
        raise NotImplementedError("blank discovery template")

    def get_last_trace(self):
        return []
```

The blank source provides only the required method signatures. It contains no feature construction, execution graph, numerical parameters, preferred recommendation model, or default recommendation strategy, and is therefore not itself an evaluated candidate. At each discovery round, the coding LLM replaces it with a complete program that must satisfy the execution contract before submission.

For a submitted candidate, the complete program $\pi = (F_\pi, G_\pi, \Theta_\pi)$ is fixed before the evaluator computes its aggregate validation score. The program and its validation outcome are then added to the discovery history used to construct subsequent proposals. After the discovery budget is exhausted, the validation incumbent is frozen and reused unchanged across test and deployment requests.

A.1.3 Feature Construction and Operator Vocabulary

Feature construction. Each inference program $\pi = (F_\pi, G_\pi, \Theta_\pi)$ jointly specifies how recommendation information is represented and how the resulting signals are organized during execution. The feature program F_π transforms observed interaction histories, item information, and frozen-model outputs into request- and item-side features. For the programs studied here, we summarize that F_π derives request-side features from observed histories and frozen-model outputs and constructs complementary item-side evidence features. The request representation controls request-dependent composition and routing, while the item-side features modify candidate scores at selected stages of the computation. The graph program G_π then organizes frozen recommenders and the outputs of F_π using a typed vocabulary of composition, routing, candidate construction, and reranking operators. The associated parameters Θ_π determine feature weights, mixture and routing coefficients, normalization choices, temperatures, and candidate cutoffs.

Table 5 summarizes the request-side representation generated by the discovered F_π from the observed interaction history and frozen-recommender outputs. Its dimensions capture history and profile properties, individual-model confidence, and pairwise agreement between recommenders.

In addition to the request representation, F_π generates item-side evidence features that can be introduced at different stages of the inference program. The discovered feature program includes seven evidence channels: `short_profile`, `long_profile`, `sequence`, `graph`, `novelty`, `cooccurrence`, and `metadata`. These channels capture complementary relations between a candidate item, the observed interaction history, and the frozen recommendation stack. The program determines which channels are active, how they are weighted, and where they enter G_π.

Operator vocabulary. As defined in Equation 7, the graph program is $G_\pi = (V_\pi, E_\pi, r_\pi)$, where V_π is the set of operator nodes, $E_\pi \subseteq V_\pi \times V_\pi$ specifies their directed dependencies, and $r_\pi \in V_\pi$ is the root node whose output determines the final ranking. Each node $v \in V_\pi$ selects an operator

Table 5: All request-side features generated by the discovered feature program F_π.

Feature group	Generated dimensions	Dim.
History and profile	Normalized history length, log history length, repeat ratio, long-window mean popularity, recent-minus-long popularity, short-profile norm, long-profile norm, and short–long profile cosine	8
Model confidence	Top-1 margin, top-10 margin, normalized top-20 entropy, top-20 score deviation, and candidate-score range for each frozen recommender	15
Model agreement	Top-10 Jaccard overlap and candidate-score correlation for each pair of frozen recommenders	6
Total	Request representation generated by F_π	29

$\mathrm{op}(v)$ from the typed vocabulary

$$\mathcal{O}_{\mathcal{C}} = \mathcal{O}_{\mathcal{C}}^{\mathrm{str}} \cup \mathcal{O}_{\mathcal{C}}^{\mathrm{comp}},$$

and is associated with parameters $\theta_v \in \Theta_\pi$, where $\Theta_\pi = \{\theta_v\}_{v \in V_\pi}$. Compositional operators transform or combine score and rank representations, while structural operators control candidate construction, branch selection, and multi-stage retrieval and reranking. Table 6 summarizes the operator vocabulary used in our experiments.

Table 6: Typed recursive operators available to the discovered graph program G_π. Structural costs are additive over graph nodes.

Operator	Type	Execution semantics	Cost
`EXPERT`	Compositional	Return the score vector produced by one frozen recommender	1
`NORMALIZE`	Compositional	Apply z-score, min–max, or reciprocal-rank normalization to an input score vector	1
`MIX`	Compositional	Combine two to eight score vectors using static or request-conditioned weights	2
`RANK_MIX`	Compositional	Combine two to eight reciprocal-rank vectors using static or request-conditioned weights	3
`ADD_EVIDENCE`	Compositional	Add selected item-side features generated by F_π to an input score vector	1
`SWITCH`	Structural	Select one of two to eight recursively defined branches using request-conditioned control	2
`CANDIDATE_UNION`	Structural	Pool bounded candidate sets from multiple sources and rank their union with a separate scoring subgraph	3
`CASCADE`	Structural	Retrieve a bounded candidate set and rerank it with a separate subgraph	4

Let $\mathrm{ch}(v)$ denote the children of node v and let $\mathbf{o}_v(x)$ denote its intermediate output for request x. Following Equation 9, the evaluator recursively computes $\mathbf{o}_v(x)$ by applying the fixed semantics $g_{\mathrm{op}(v)}$ to the outputs $\{\mathbf{o}_u(x) : u \in \mathrm{ch}(v)\}$ together with the system information $\mathbf{z}_x$, the features constructed by F_π, and the node parameters θ_v. The request-side representation generated by F_π can condition compatible `MIX`, `RANK_MIX`, and `SWITCH` nodes, while its item-side outputs enter the graph through `ADD_EVIDENCE`. Structural operators may contain complete recursive subgraphs as their sources, scorers, retrievers, rerankers, and fallback branches. Execution terminates at r_π, and the ranked output is obtained from $\mathbf{o}_{r_\pi}(x)$ according to Equation 10.

The structural cost of a discovered graph is

$$\mathrm{Cost}_{\mathcal{C}}(G_\pi) = \sum_{v \in V_\pi} c_{\mathcal{C}}(\mathrm{op}(v)) \leq 64, \tag{18}$$

where V_π is the set of nodes in G_π and $c_\mathcal{C}$ is the operator cost defined in Table 6. A valid graph contains at most 64 nodes, has depth at most 12, and assigns at most eight children to a multi-input operator. Temperatures are constrained to $[0.05, 10]$, and the candidate cutoffs used by `CANDIDATE_UNION` and `CASCADE` are at most 512. These bounds ensure that every discovered program remains deterministic and executable under the fixed contract.

A.1.4 Proposal Construction and Validation

Figure 4 shows how one candidate program is constructed and formally evaluated. At each discovery round, the coding LLM receives the blank program interface, the public system description, and the history of previously evaluated programs. It then writes a complete candidate $\pi = (F_\pi, G_\pi, \Theta_\pi)$, checks its executability, and may revise the source before submitting it for validation.

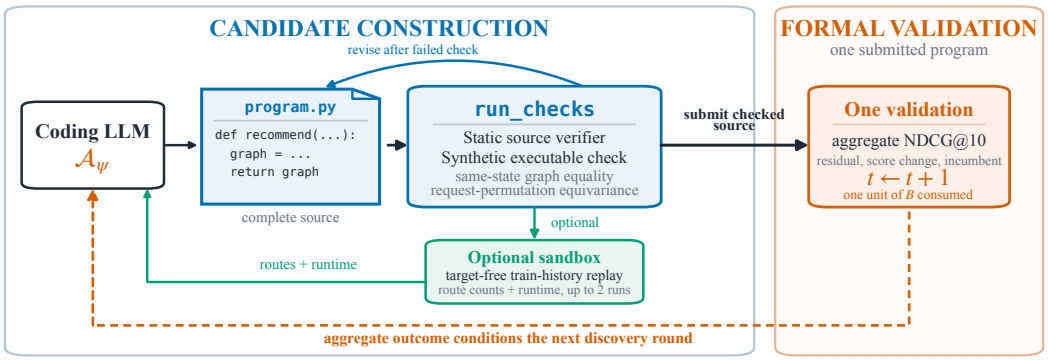


Figure 4: Construction and formal evaluation of one candidate program. The coding LLM writes and checks a complete program and may inspect target-free execution diagnostics before submission. Submitting a checked program triggers one aggregate validation evaluation, consumes one unit of the discovery budget B, and adds the resulting feedback to the discovery history.

Program construction. The coding LLM may inspect the executable interface, public system schema, and earlier program sources and outcomes before writing a candidate. Each source revision specifies a complete feature program, graph program, and parameter set. A proposal lineage permits up to five complete source writes and two target-free sandbox executions, allowing the coding LLM to repair invalid programs or refine their behavior before submission.

Executable checks. The `run_checks` procedure first applies a static source verifier and then executes the candidate on a synthetic fixture. The verifier checks the required program interface, permitted imports, bounded source and state representations, and the absence of unsupported operations. The executable check verifies that the returned graph satisfies the operator, depth, node-count, and structural-cost constraints and that repeated execution is deterministic. A source revision invalidates the previous check result, so the exact source submitted for validation must pass the checks.

Target-free diagnostics. After passing the executable checks, the current program may be executed in a target-free sandbox. The sandbox evaluates the complete feature and graph programs on observed training histories and reports structural properties, route counts, fallback behavior, and runtime. These diagnostics help the coding LLM identify execution problems and refine the candidate without adding a ranking score to the discovery history.

Formal validation. Only the currently checked source can be submitted for formal evaluation. The evaluator executes the submitted program on the complete validation split and returns an aggregate NDCG@10 score, the difference from the public reference, the change relative to the current incumbent, and whether it becomes the new incumbent. The submitted source and this feedback are appended to the ordered discovery history used in the next round. Contract checks, source revisions,

and target-free sandbox executions do not consume the evaluated-program budget; one unit of B is consumed only when a checked program is formally evaluated.

A.1.5 Complete Discovery Procedure

Algorithm 1 gives the complete discovery procedure. The ordered history $\mathcal{H}_t$ contains the source and aggregate validation feedback of every formally evaluated program, while $\mathcal{R}_{t,w}^{\text{build}}$ records the checks and target-free diagnostics produced while constructing the current proposal. Only formal validation adds an entry to $\mathcal{H}_t$ and consumes one unit of the evaluated-program budget B.

At each round, the coding LLM constructs a complete candidate from the blank interface together with the accumulated discovery history and the diagnostics generated within the current proposal lineage. Failed executable checks remain local to the proposal and lead to a source revision. A checked source can optionally be inspected in the target-free sandbox before submission. Once submitted, the complete program is evaluated on validation exactly once, and its aggregate outcome becomes available to subsequent discovery rounds. The final output is the frozen program with the highest validation score among all contract-valid programs evaluated within the available budget.

Discovery complexity. Let N_{val} be the number of validation requests, M the number of candidate items, H_x the length of the observed history for request x, $n_\pi = |V_\pi|$ the number of nodes in G_π, q_π the number of candidate-selection nodes, and k_π the largest candidate cutoff used by the program. The frozen recommendation scores are supplied by $\mathcal{S}$ and are therefore treated as inputs to program execution. Constructing the request- and item-side outputs of F_π requires

$$O(H_x + M) \tag{19}$$

time per request because the number of frozen recommenders, request dimensions, and item-side channels are fixed. Score transformation and composition nodes require $O(M)$ time each, while a candidate-selection node can be implemented in $O(M \log k_\pi)$ time. The execution time of one program on one request is therefore

$$T_{\text{exec}}(\pi, x) = O(H_x + (n_\pi + q_\pi \log k_\pi)\, M)\,. \tag{20}$$

Evaluating a discovery trajectory containing $T_B \le B$ valid programs consequently requires

$$O\left(\sum_{t=1}^{T_B} \sum_{x \in \mathcal{V}} \left[H_x + (n_{\pi_t} + q_{\pi_t} \log k_{\pi_t})\, M\right]\right) \tag{21}$$

deterministic evaluator time. Using the contract bounds $n_\pi \le n_{\max}$, $q_\pi \le n_{\max}$, and $k_\pi \le k_{\max}$, this is upper-bounded by

$$O(B N_{\text{val}} \left[H_{\max} + (n_{\max} + n_{\max} \log k_{\max})\, M\right])\,, \tag{22}$$

where $n_{\max} = 64$ and $k_{\max} = 512$ in our implementation. Thus, under the fixed execution contract, deterministic discovery evaluation scales linearly with the evaluated program budget and the number of validation requests.

Proposal construction permits at most BW complete source writes and BU target-free sandbox executions over a full trajectory. Coding-LLM generation time is provider-dependent and is therefore measured by recorded API usage rather than included in the deterministic evaluator complexity. Static source verification is linear in source length, while each executable check and sandbox run is bounded by the same graph and resource constraints used for formal execution.

Deployment complexity. After discovery, only the frozen validation incumbent $\pi_B^\star$ is executed. Its per-request time complexity is

$$T_{\text{deploy}}(x) = O\big(H_x + \big(n_{\pi_B^\star} + q_{\pi_B^\star} \log k_{\pi_B^\star}\big)\, M\big)\,, \tag{23}$$

which contains no multiplicative dependence on the discovery budget B. Deployment requires no LLM calls or updates to the recommendation model. The worst-case working memory for a batch of b requests is $O(b\, n_{\pi_B^\star} M)$ when all intermediate node outputs are retained, while the request-side feature representation contributes only $O(29b)$ additional values. The one-time discovery cost, therefore, produces a single bounded program whose serving complexity is determined by its frozen structure rather than by the amount of compute used to discover it.

Algorithm 1 Trajectory-guided program discovery in PROGRAMREC

Require: Frozen recommendation stack $\mathcal{S}$, execution contract $\mathcal{C}$, validation environment $\mathcal{V}$, evaluator $\mathcal{M}_{\text{val}}$, coding LLM $\mathcal{A}_\psi$
Require: Blank interface $\pi_\emptyset$, evaluated-program budget B, source-write limit W, sandbox limit U, public reference score v_{ref}
Ensure: Frozen validation incumbent $\pi_B^\star$

$\mathcal{H}_0 \leftarrow \emptyset$, $\bar{\pi}_0 \leftarrow \varnothing$, $\bar{v}_0 \leftarrow -\infty$, $T_B \leftarrow 0$
for $t = 1, \ldots, B$ **do**
  $\mathcal{R}_{t,0}^{\text{build}} \leftarrow \emptyset$, $u \leftarrow 0$, submitted $\leftarrow$ false
  for $w = 1, \ldots, W$ **do**
    $\widetilde{\pi}_{t,w} \sim \mathcal{A}_\psi\big(\cdot \mid \mathcal{S}, \mathcal{C}, \pi_\emptyset, \mathcal{H}_{t-1}, \mathcal{R}_{t,w-1}^{\text{build}}\big)$
    $\mathbf{d}_{t,w}^{\text{check}} \leftarrow \text{Check}(\widetilde{\pi}_{t,w}; \mathcal{C})$
    $\mathcal{R}_{t,w}^{\text{build}} \leftarrow \mathcal{R}_{t,w-1}^{\text{build}} \oplus (\widetilde{\pi}_{t,w}, \mathbf{d}_{t,w}^{\text{check}})$
    if $\mathbf{d}_{t,w}^{\text{check}}$ indicates failure **then**
      continue
    end if
    if $\mathcal{A}_\psi$ requests a sandbox execution and $u < U$ **then**
      $\mathbf{d}_{t,w}^{\text{box}} \leftarrow \text{Sandbox}(\widetilde{\pi}_{t,w}; \mathcal{S}, \mathcal{C})$
      $\mathcal{R}_{t,w}^{\text{build}} \leftarrow \mathcal{R}_{t,w}^{\text{build}} \oplus \mathbf{d}_{t,w}^{\text{box}}$
      $u \leftarrow u + 1$
    end if
    if $\mathcal{A}_\psi$ submits the current checked source **then**
      $\pi_t \leftarrow \widetilde{\pi}_{t,w}$, submitted $\leftarrow$ true
      break
    end if
  end for
  if submitted $=$ false **then**
    break
  end if
  $v_t \leftarrow \mathcal{M}_{\text{val}}(\pi_t; \mathcal{S}, \mathcal{V})$
  if $T_B = 0$ **then**
    $\delta_t^{\text{inc}} \leftarrow 0$
  else
    $\delta_t^{\text{inc}} \leftarrow v_t - \bar{v}_{t-1}$
  end if
  $\delta_t^{\text{ref}} \leftarrow v_t - v_{\text{ref}}$
  $a_t \leftarrow \mathbb{I}[T_B = 0 \ \vee \ v_t > \bar{v}_{t-1}]$
  $\mathbf{f}_t \leftarrow (v_t, \delta_t^{\text{ref}}, \delta_t^{\text{inc}}, a_t)$
  $\mathcal{H}_t \leftarrow \mathcal{H}_{t-1} \oplus (\pi_t, \mathbf{f}_t)$
  if $a_t = 1$ **then**
    $\bar{\pi}_t \leftarrow \pi_t$, $\bar{v}_t \leftarrow v_t$
  else
    $\bar{\pi}_t \leftarrow \bar{\pi}_{t-1}$, $\bar{v}_t \leftarrow \bar{v}_{t-1}$
  end if
  $T_B \leftarrow t$
end for
$\pi_B^\star \leftarrow \text{Freeze}(\bar{\pi}_{T_B})$
return $\pi_B^\star, \mathcal{H}_{T_B}$

A.2 EXPERIMENTAL DETAILS

This section provides details of the datasets, frozen recommendation stack, baseline implementations, discovery configuration, evaluation protocol, and additional analyses.

A.2.1 Datasets and Evaluation Protocol

We use the processed Amazon Movies and TV, GoodReads, and Yelp subsets released with RecToolAgent Yang et al. (2026b). These subsets are derived from the Amazon Reviews, GoodReads Book Graph, and Yelp Open Dataset, respectively. Interactions are ordered chronologically for each user. The final interaction is reserved for testing, the preceding interaction is used for validation, and all earlier interactions form the observed training history. Table 7 summarizes the resulting item catalogs and evaluation requests.

Table 7: **Dataset and split statistics.** Total interactions are counted before the temporal leave-two-out split. Each user contributes one validation and one test interaction, while the remaining interactions form the training histories. Avg. length denotes the average complete interaction-sequence length, and density is the ratio of interactions to all possible user–item pairs.

Dataset	Users	Items	Interactions	Train interactions	Avg. length	Density (%)
Amazon	1,054	1,839	35,492	33,384	33.7	1.83
GoodReads	3,204	2,167	246,034	239,626	76.8	3.54
Yelp	4,303	3,429	258,486	249,880	60.1	1.75

All methods follow full-catalog temporal next-item evaluation. For a request x with observed history h_x and held-out next item i_x^+, the candidate set is

$$\mathcal{I}_x = (\mathcal{I} \setminus \{i : i \in h_x\}) \cup \{i_x^+\}, \tag{24}$$

where $\mathcal{I}$ denotes the complete item catalog. Thus, previously observed items are excluded while the held-out next item is retained.

Let $s_x(i)$ denote the final score assigned to candidate $i \in \mathcal{I}_x$. The rank of the held-out item is

$$r_x = 1 + \sum_{i \in \mathcal{I}_x} \mathbb{I}\big[s_x(i) > s_x(i_x^+)\big] + \sum_{i \in \mathcal{I}_x} \mathbb{I}\big[s_x(i) = s_x(i_x^+) \wedge i < i_x^+\big], \tag{25}$$

where score ties are resolved by ascending item identifier. For a split containing N requests, we compute

$$\text{Recall}\,@K = \frac{1}{N}\sum_{x=1}^{N} \mathbb{I}[r_x \le K], \qquad \text{NDCG}\,@K = \frac{1}{N}\sum_{x=1}^{N} \frac{\mathbb{I}[r_x \le K]}{\log_2(r_x + 1)}. \tag{26}$$

We report both metrics for $K \in \{5, 10\}$ and use validation NDCG@10 to select the final program within each discovery trajectory.

A.2.2 Baseline Configurations

All baselines use the same candidate sets, observed-history masks, frozen recommendation stack, and ranking metrics. The stack $\mathcal{S} = (\mathcal{E}, \mathcal{D})$ contains SASRec Kang & McAuley (2018), LightGCN He et al. (2020), and SimpleX Mao et al. (2021). We load their pretrained user and item representations from the benchmark and keep all recommender parameters frozen. For each request x, the score of every expert $e_j \in \mathcal{E}$ is normalized over the eligible candidate set $\mathcal{I}_x$, with the standard deviation floored at 10^{-6}. Items in the observed history are masked throughout execution. The individual-recommender baselines directly rank candidates using these frozen scores.

StaticFusion. We include StaticFusion as a strong validation-tuned control for determining whether performance gains arise simply from combining the same frozen recommenders. It has access to the same expert scores and validation split as ProgramRec but applies a single globally selected, request-independent composition to every request. It therefore separates the benefit of conventional ensemble selection from that of feature-dependent and graph-structured program execution. StaticFusion enumerates nonnegative three-model weights that sum to one at resolution $1/20$, producing 231 weight vectors. For each vector $\mathbf{w}$, it evaluates one score composition $\sum_j w_j s_j(x, i)$ and four reciprocal-rank compositions $\sum_j w_j/(c + r_j(x, i))$ with $c \in \{0, 10, 60, 100\}$, yielding 1,155 fixed candidates. The candidate with the highest validation NDCG@10 is frozen for test

evaluation. The selected SASRec–LightGCN–SimpleX weights are $(0.55, 0.25, 0.20)$ on Amazon, $(0.60, 0.20, 0.20)$ on GoodReads, and $(0.30, 0.40, 0.30)$ on Yelp; all three selected candidates use score composition.

RouteRec. We include RouteRec as a representative learned routing baseline that combines the same frozen recommenders through a fixed, supervised model. It tests whether a conventional combination rule fitted from calibration examples is sufficient without searching over feature construction and executable graph structure. Our matched implementation follows the StackCheap variant of RouteRec Zhou et al. (2026). For each request, it forms the union of the three recommenders' top-10 lists. Every candidate is represented by four features from each recommender—top-list membership, reciprocal rank, rank percentile, and min–max-normalized score—together with item popularity computed from the training split. A class-balanced logistic stacker is fitted on the training-calibration split, and its regularization coefficient is selected from $C \in \{0.03, 0.1, 0.3\}$ using validation $\mathrm{NDCG@10} + 0.5\,\mathrm{Recall@10}$. The selected values are 0.03, 0.03, and 0.10 for Amazon, GoodReads, and Yelp, respectively. The resulting stacker is then frozen for test evaluation.

RecToolAgent. We include RecToolAgent as the closest agentic recommendation baseline based on LLM-mediated use of pretrained recommendation tools. This comparison distinguishes offline discovery of a reusable inference program from invoking an LLM to compare and rerank outputs from recommendation tools for individual requests. We run the released RecToolAgent control path Yang et al. (2026b) with its frozen benchmark assets and DeepSeek-V4-Pro. We reuse the released user-profile cache and retain the original prompts, the recommendation-tool comparison, the weight-memory updates, and the final reranking procedure. For each request, the pipeline performs three model-specific relevance judgments, one recommendation-tool comparison, and one reranking operation that maps the top-15 candidates to the final top-10 list. We use a single evaluation epoch and disable recommender weight saving. The frozen top-10 outputs are parsed as exact integer item identifiers and rescored using the common evaluation metrics.

Random Search. We include Random Search to separate the value of the executable program space from the value of coding-LLM-guided discovery. It uses the same contract and evaluated-program budget as PROGRAMREC, but constructs candidates non-adaptively rather than using validation feedback to guide subsequent proposals. Each trajectory evaluates $B = 10$ distinct contract-valid candidates: one public reference that returns the validation-best frozen recommender and nine non-adaptive random candidates. For each non-reference candidate, the sampler constructs a fresh program $\pi = (F_\pi, G_\pi, \Theta_\pi)$. Candidate-specific feature statistics are computed without target labels and bound before evaluation. Invalid or duplicate constructions are resampled without consuming the evaluated program budget, and validation scores are used only to select the incumbent. After all ten candidates have been evaluated, the program with the highest validation NDCG@10 is frozen and evaluated once on the test split. We run three trajectories per dataset, using different seeds, and report the mean test performance across their independently selected programs.

A.2.3 IMPLEMENTATION DETAILS

Coding LLMs and trajectories. We instantiate the coding LLM $\mathcal{A}_\psi$ with DeepSeek-V4-Pro Xu et al. (2026), GPT-5.6 Luna OpenAI (2026), and Kimi K2.7 Code Team et al. (2025). DeepSeek-V4-Pro is invoked with thinking enabled and maximum reasoning effort, GPT-5.6 Luna uses high reasoning effort, and Kimi K2.7 Code uses mandatory thinking with temperature 1.0 and top-p 0.95. Each model receives the same logical discovery prompt, program interface, execution contract, history format, validation-feedback interface, and tool semantics. For each dataset-coding-LLM pair, we run three separate discovery trajectories. Each trajectory begins from the blank program interface and an empty history $\mathcal{H}_0$, and evaluates up to $B = 10$ contract-valid programs. The structural-cost budget is $\beta = 64$, while program checking and validation/test execution use a batch size 128 and a deterministic evaluator.

Proposal construction and checking. At discovery round t, the coding LLM can inspect the public interface and $\mathcal{H}_{t-1}$, write a complete inference program, run executable checks, optionally inspect target-free execution diagnostics, and submit the checked source. Each proposal lineage permits at most five complete source writes and two target-free sandbox executions. The checker

verifies the source interface, allowed operations, deterministic execution, permutation equivariance, and graph-budget constraints without accessing recommendation targets. The sandbox similarly contains no targets or user identifiers and reports only execution properties such as graph topology, route counts, candidate coverage, fallback behavior, and runtime. A new write invalidates the preceding check, and only the currently checked source can be submitted. Contract-invalid revisions and byte-identical historical programs are rejected before validation and do not consume the evaluated-program budget B.

Validation feedback and selection. Every submitted contract-valid program is evaluated once on the complete validation split, yielding $v_t = \mathrm{NDCG@10}$. The next proposal receives this aggregate score, its residual relative to the public best-expert reference, its change relative to the current incumbent, and an indicator of whether it became the new incumbent. Per-request targets, ranks, metric contributions, and branch counterfactuals are never exposed. If a trajectory contains T valid submitted programs, the incumbent available at a budget of b is

$$s_b^\star = \min\left(\operatorname*{arg\,max}_{1\le t\le \min(b,T)} v_t\right), \qquad \pi_b^\star = \pi_{s_b^\star}, \tag{27}$$

So, exact validation ties are resolved in favor of the earlier submitted program. The main results use the final available incumbent $\pi_B^\star$ from each trajectory and report the mean test metrics over the three trajectories.

Frozen test execution. After discovery, the selected source, validation result, and selection record are hash-bound before test access. The test evaluator accepts only the source matching this frozen record and executes it once on the complete test split. Test execution makes no coding-LLM calls, performs no calibration or recommendation model update, and does not modify the frozen program state. Test outcomes are never added to $\mathcal{H}_t$ and therefore cannot affect any proposal or program selection.

A.3 Additional Experimental Results

A.3.1 Multi-Trajectory Results and Budget Scaling

Table 8 summarizes the final validation-selected programs for each dataset and coding-LLM pair. The round at which the final incumbent first appears, its validation NDCG@10, and its subsequent frozen test NDCG@10 are reported as the mean and sample standard deviation over three independent discovery trajectories. The selected programs frequently emerge late in discovery: the mean selection round lies between 7.7 and 10.0 in eight of the nine conditions, while GPT on Yelp stabilizes earlier. Performance variation is largest on GoodReads and smaller on Yelp, showing that trajectory sensitivity depends on both the dataset and the coding LLM.

Table 8: Final validation-selected programs over three independent discovery trajectories per dataset and coding LLM. Entries report mean $\pm$ sample standard deviation. B_{sel} is the round at which the final validation incumbent first appears.

Dataset	Coding LLM	B_{sel}	Validation NDCG@10	Test NDCG@10
Amazon	DeepSeek-V4-Pro	8.7 ± 2.3	0.0690 ± 0.0060	0.0427 ± 0.0064
Amazon	GPT-5.6 Luna	7.7 ± 4.0	0.0586 ± 0.0074	0.0373 ± 0.0054
Amazon	Kimi K2.7 Code	9.0 ± 1.0	0.0593 ± 0.0060	0.0316 ± 0.0067
GoodReads	DeepSeek-V4-Pro	10.0 ± 0.0	0.1299 ± 0.0050	0.1128 ± 0.0072
GoodReads	GPT-5.6 Luna	8.0 ± 2.6	0.1066 ± 0.0212	0.0894 ± 0.0205
GoodReads	Kimi K2.7 Code	7.7 ± 2.3	0.1229 ± 0.0102	0.1035 ± 0.0164
Yelp	DeepSeek-V4-Pro	9.3 ± 1.2	0.0539 ± 0.0021	0.0484 ± 0.0014
Yelp	GPT-5.6 Luna	2.3 ± 2.3	0.0497 ± 0.0075	0.0435 ± 0.0042
Yelp	Kimi K2.7 Code	10.0 ± 0.0	0.0547 ± 0.0024	0.0489 ± 0.0019

To measure scaling with discovery compute, at each budget $B \in \{1, 3, 5, 8, 10\}$ we recover the validation incumbent among the programs available up to that budget, freeze its complete source and

parameters, and evaluate it on test. Figure 2 visualizes the resulting mean test NDCG@10 with one-sample-standard-deviation bands, while Table 9 provides the corresponding numerical values. Test outcomes are used only for this post-hoc analysis and never affect proposal generation or validation selection.

Table 9: **Frozen test NDCG@10 across discovery budgets.** Entries report mean ± sample standard deviation over three independent trajectories. DeepSeek, GPT, and Kimi denote DeepSeek-V4-Pro, GPT-5.6 Luna, and Kimi K2.7 Code, respectively.

Coding LLM	$B = 1$	$B = 3$	$B = 5$	$B = 8$	$B = 10$
Amazon Movies and TV					
DeepSeek	$.0203 \pm .0062$	$.0247 \pm .0025$	$.0352 \pm .0101$	$.0408 \pm .0060$	$.0427 \pm .0064$
GPT	$.0228 \pm .0010$	$.0300 \pm .0040$	$.0310 \pm .0023$	$.0322 \pm .0001$	$.0373 \pm .0054$
Kimi	$.0209 \pm .0026$	$.0291 \pm .0060$	$.0301 \pm .0036$	$.0305 \pm .0075$	$.0316 \pm .0067$
GoodReads					
DeepSeek	$.0537 \pm .0084$	$.0680 \pm .0127$	$.0723 \pm .0203$	$.1052 \pm .0074$	$.1128 \pm .0072$
GPT	$.0617 \pm .0093$	$.0805 \pm .0280$	$.0876 \pm .0228$	$.0885 \pm .0218$	$.0894 \pm .0205$
Kimi	$.0528 \pm .0048$	$.0856 \pm .0127$	$.0897 \pm .0140$	$.1024 \pm .0155$	$.1035 \pm .0164$
Yelp					
DeepSeek	$.0426 \pm .0021$	$.0453 \pm .0006$	$.0455 \pm .0006$	$.0478 \pm .0020$	$.0484 \pm .0014$
GPT	$.0421 \pm .0016$	$.0421 \pm .0016$	$.0435 \pm .0042$	$.0435 \pm .0042$	$.0435 \pm .0042$
Kimi	$.0409 \pm .0052$	$.0461 \pm .0004$	$.0462 \pm .0005$	$.0480 \pm .0018$	$.0489 \pm .0019$

Across all dataset–coding-LLM conditions, mean frozen test NDCG@10 is higher at $B = 10$ than at $B = 1$. At the trajectory level, the median relative endpoint gain is 61.2%, showing that additional proposal evaluations generally produce stronger frozen programs. The scaling curves need not be monotonic because checkpoint incumbents are selected solely by validation performance; occasional test fluctuations therefore reflect trajectory-dependent selection rather than the use of test feedback.

A.3.2 Component Ablations

We examine the roles of feature construction, structural organization, and model coordination by applying deterministic transformations to one frozen DeepSeek-V4-Pro program for each dataset. Each transformation modifies only the designated use of F_π, the structure of G_π, or the corresponding parameters in Θ_π, while preserving the remaining program. The transformed source is checked against the same execution contract and replayed on the same validation requests without rediscovery or additional fitting. Table 10 defines the transformations.

A.3.3 Program Formation and Refinement

We isolate program formation through trajectory feedback from numerical refinement within an established program structure using a controlled GoodReads analysis with DeepSeek-V4-Pro. Figure 3(a) reports the corresponding results.

Full discovery. The standard discovery procedure provides each proposal lineage with the ordered history $\mathcal{H}_t$, including previously submitted program sources, their aggregate validation outcomes, and the current incumbent.

No trajectory history. Each proposal lineage starts from the blank interface without access to earlier submitted programs, aggregate validation outcomes, incumbent information, or dialogue state. Contract-checking feedback and target-free sandbox feedback remain available within the current lineage, isolating the contribution of information accumulated across proposals.

Fixed-topology refinement. This is a subsequent refinement stage initialized from the validation-selected program produced by full discovery. We preserve its root, graph connectivity, operator types, branch arities, frozen-recommender references, normalization modes, and feature set. The refinement stage evaluates an additional $B = 10$ parameter-only programs in which only numerical

Table 10: **Component-ablation definitions.** Exact deterministic transformations used in the ablation study of Section 4.4.

Program variant	Transformation applied to the frozen program
Feature construction	
w/o User Profile	Zero the request-conditioning coefficients derived from profile summaries and remove the `short_profile` and `long_profile` item-feature weights.
w/o Confidence/Agreement	Zero the request-conditioning coefficients that describe frozen-model confidence and pairwise agreement.
w/o Derived Context Features	Recursively bypass every `ADD_EVIDENCE` node while preserving its input subgraph.
Structural organization	
w/o Multi-Stage Structure	Promote the frozen score-composition subgraph to the root and remove the surrounding candidate-construction, cascade, and reranking stages.
Composition and control	
w/o Request Conditioning	Use frozen bias-only weights at composition nodes and the frozen bias-only branch choice at switching nodes.
w/o Model Coordination	Replace the complete inference program with the strongest individual frozen recommender on the corresponding validation split.
w/o Rank Consensus	Replace every `RANK_MIX` node with `MIX` while preserving its inputs and weighting parameters; this transformation is a no-op when the graph contains no `RANK_MIX` node.

weights, biases, temperatures, feature-adjustment coefficients, and bounded retrieval widths may change.

Full discovery and the no-history condition each start from the blank interface and evaluate $B = 10$ contract-valid programs. Fixed-topology refinement instead begins from the full-discovery winner and consumes an additional $B = 10$ refinement budget. All stages use the same executable interface, evaluator, and search-action limits. The validation-selected program from each stage is frozen before one test execution. Each condition selects its validation incumbent before one frozen test execution. Paired comparisons use $20{,}000$ bootstrap resamples of the aligned $3{,}204$ GoodReads test requests; the resulting intervals quantify request-level variation rather than uncertainty across discovery trajectories.

A.3.4 Discovered Program Structures

Figure 3(b) visualizes the graph size and structural cost of nine reference programs, one validation-selected program from a reference discovery trajectory for each dataset–coding-LLM pair. Tables 11 and 12 provide their exact graph properties and operator compositions. The root column records $\mathrm{op}(r_\pi)$, the node and depth columns describe G_π, and structural cost is computed using $\mathrm{Cost}_{\mathcal{C}}(G_\pi)$. These tables describe the concrete programs underlying the figure.

All nine programs use the request-dependent information generated by F_π, but organize the frozen recommenders and item-side evidence through different realizations of G_π. Within this reference set, the DeepSeek-V4-Pro programs use `CASCADE` roots and relatively compact graphs, the GPT-5.6 Luna programs use `SWITCH` roots with similar graph sizes, and the Kimi K2.7 Code programs exhibit larger and more varied structures. The operator counts further show that programs constructed from the same vocabulary can realize substantially different computation graphs. Structural-cost variation across all trajectories is reported separately in Section A.3.6.

A.3.5 Code-Level Transfer

We analyze nine DeepSeek-V4-Pro programs: the validation-selected program from each of three discovery trajectories for every source dataset. For transfer, the source-defined F_π and G_π and

Table 11: **Structures of the reference programs.** Exact graph properties of the nine programs shown in Figure 3(b). These are individual program measurements.

Coding LLM	Root operator	Nodes	Depth	Cost_C
Amazon Movies and TV				
DeepSeek-V4-Pro	CASCADE	24	5	30
GPT-5.6 Luna	SWITCH	30	6	39
Kimi K2.7 Code	SWITCH	44	8	58
GoodReads				
DeepSeek-V4-Pro	CASCADE	19	6	25
GPT-5.6 Luna	SWITCH	30	7	41
Kimi K2.7 Code	CASCADE	36	7	46
Yelp				
DeepSeek-V4-Pro	CASCADE	18	6	22
GPT-5.6 Luna	SWITCH	30	7	40
Kimi K2.7 Code	CANDIDATE_UNION	46	6	57

Table 12: **Operator composition of the reference programs.** Entries count nodes in each frozen graph. EXP, NORM, R-MIX, FEAT, SW, UNION, and CASC abbreviate EXPERT, NORMALIZE, RANK_MIX, ADD_EVIDENCE, SWITCH, CANDIDATE_UNION, and CASCADE.

Dataset	EXP	NORM	MIX	R-MIX	FEAT	SW	UNION	CASC
DeepSeek-V4-Pro								
Amazon	7	7	1	1	7	0	0	1
GoodReads	9	3	1	0	4	0	1	1
Yelp	5	5	1	0	6	0	0	1
GPT-5.6 Luna								
Amazon	16	6	2	2	2	1	1	0
GoodReads	15	3	1	2	6	1	1	1
Yelp	15	6	2	1	3	1	1	1
Kimi K2.7 Code								
Amazon	17	17	2	3	2	1	1	1
GoodReads	13	12	1	2	5	2	0	1
Yelp	19	18	5	2	1	0	1	0

all numerical constants in Θ_π remain unchanged. The target interface supplies the target catalog, frozen-model outputs, observed histories, item information, and target-specific normalization statistics required by the executable contract. No target-side discovery, validation feedback, parameter fitting, or coding-LLM call is used.

Figure 3(c) reports transfer performance relative to the target-specific program mean. Table 13 supplements that view with absolute frozen test NDCG@10 and sample standard deviation over the source trajectories. Transfer remains substantial across the off-diagonal cells, while its variability depends on the source–target pair.

Table 13: **Code-level transfer with trajectory variation.** For each source dataset, three independently discovered DeepSeek-V4-Pro programs are executed unchanged on every target test split. Entries report mean frozen test NDCG@10 $\pm$ sample standard deviation over the source trajectories. Bold marks an off-diagonal mean above the corresponding target-specific diagonal mean.

Source $\setminus$ Target	Amazon Movies and TV	GoodReads	Yelp
Amazon	$.0427 \pm .0064$	$.1094 \pm .0070$	$.0482 \pm .0027$
GoodReads	$\mathbf{.0466} \pm .0062$	$.1128 \pm .0072$	$.0468 \pm .0032$
Yelp	$.0394 \pm .0039$	$.0993 \pm .0090$	$.0484 \pm .0014$

A.3.6 DISCOVERY AND DEPLOYMENT EFFICIENCY

PROGRAMREC incurs coding-LLM cost only during discovery. For a discovery trajectory τ, we define its discovery-call count as the number of coding-LLM invocations recorded in the completed trajectory ledger up to the selection of $\pi^{\star}_{B}$. This count covers the complete model–tool interaction used to inspect resources, construct and revise source code, invoke executable checks and target-free sandboxes, and submit candidate programs. We measure deployment complexity using the contract-defined structural cost $\mathrm{Cost}_{\mathcal{C}}(G_{\pi^{\star}_{B}})$ of the selected program. Frozen test execution and subsequent deployment require no coding-LLM calls and do not update the underlying recommenders.

For each dataset and coding-LLM pair, Table 14 reports the mean, minimum, and maximum of both quantities over three discovery trajectories. The call count measures one-time discovery effort, whereas $\mathrm{Cost}_{\mathcal{C}}$ measures the structure retained by the deployed program.

Table 14: Discovery calls and frozen-program structural cost over three trajectories per dataset–coding-LLM pair. Values are reported as mean [minimum, maximum].

Dataset	Coding LLM	Calls	$\mathrm{Cost}_{\mathcal{C}}$
Amazon	DeepSeek	93.0 [89, 95]	50.3 [30, 63]
Amazon	GPT	108.0 [89, 130]	47.7 [39, 54]
Amazon	Kimi	81.3 [78, 86]	38.7 [20, 58]
GoodReads	DeepSeek	85.0 [81, 88]	29.0 [25, 35]
GoodReads	GPT	97.3 [93, 105]	32.7 [23, 41]
GoodReads	Kimi	80.7 [71, 94]	56.3 [46, 64]
Yelp	DeepSeek	103.7 [99, 108]	42.7 [22, 59]
Yelp	GPT	100.3 [97, 103]	23.0 [11, 40]
Yelp	Kimi	86.0 [79, 99]	53.3 [40, 63]

Across the individual trajectories, discovery calls range from 71 to 130, while frozen-program structural costs range from 11 to the contract limit of 64. The two quantities do not vary proportionally. For example, Kimi uses the fewest mean discovery calls on GoodReads but produces the highest mean structural cost, whereas GPT uses approximately one hundred calls on Yelp but produces the lowest mean structural cost on that dataset. Thus, a longer discovery interaction does not necessarily yield a larger deployed program, and all selected programs remain executable within the fixed structural budget.