SELF-EVOLVEREC: SELF-EVOLVING RECOMMENDER SYSTEMS WITH LLM-BASED DIRECTIONAL FEEDBACK

Sein Kim[1,✉], **Sangwu Park**[1,✉], **Hongseok Kang**[1,✉], **Wonjoong Kim**[1,✉], **Jimin Seo**[1,✉], **Yeonjun In**[1,✉], **Kanghoon Yoon**[1,✉], **Chanyoung Park**[1,✉*]
[1]KAIST

ABSTRACT

Traditional methods for automating recommender system design, such as Neural Architecture Search (NAS), are often constrained by a fixed search space defined by human priors, limiting innovation to pre-defined operators. While recent LLM-driven code evolution frameworks shift fixed search space target to open-ended program spaces, they primarily rely on scalar metrics (e.g., NDCG) that fail to provide qualitative insights into model failures or directional guidance for improvement. To address this, we propose Self-EvolveRec, a novel framework that establishes a directional feedback loop by integrating a User Simulator for qualitative critiques and a Model Diagnosis Tool for quantitative internal verification. Furthermore, we introduce a Diagnosis Tool - Model Co-Evolution strategy to ensure that evaluation criteria dynamically adapt as the recommendation architecture evolves. Extensive experiments demonstrate that Self-EvolveRec significantly outperforms state-of-the-art NAS and LLM-driven code evolution baselines in both recommendation performance and user satisfaction. Our code is available at `https://github.com/Sein-Kim/self_evolverec`.

1 INTRODUCTION

Driven by the growth of online data Covington et al. (2016); Gomez-Uribe & Hunt (2016), recommender systems have evolved from MF He et al. (2017); Salakhutdinov & Mnih (2007); Kim et al. (2016) to Transformers Kang & McAuley (2018); Sun et al. (2019). However, optimal performance relies heavily on the entire recommendation pipeline (e.g., loss functions, negative sampling) rather than model architecture alone Naumov et al. (2019); Zou & Sun (2025), making manual refinement of this pipeline by human expertise inefficient and costly Zheng et al. (2023b). To mitigate manual design inefficiencies, Automated Machine Learning (AutoML) techniques, such as Neural Architecture Search (NAS) Zoph & Le (2017); Elsken et al. (2019) have emerged as a prominent approach to automating the discovery of optimal recommendation architectures Liu et al. (2020); Song et al. (2020); Krishna et al. (2021); Zhang et al. (2023); Li et al. (2022).

However, these methods are inherently constrained by a fixed search space bounded by human priors Ci et al. (2021); Real et al. (2020), limiting optimization to symbolic combinations within a closed pool of pre-defined operators. Due to this lack of generative expressivity, existing NAS methods struggle to address non-architectural components—such as loss functions and data processing—and fail to jointly optimize the entire pipeline. Consequently, achieving comprehensive system optimization requires shifting from a closed pool to an open-ended program space Real et al. (2020).

To realize open-ended optimization, a new paradigm known as LLM-driven code evolution has emerged. Pioneering frameworks such as FunSearch Romera-Paredes et al. (2024) and Eureka Ma et al. (2023) validated LLM-driven code evolution approach by leveraging LLMs to optimize isolated functions. Substantially expanding this scope, AlphaEvolve Novikov et al. (2025) targets entire codebases rather than single functions, by orchestrating an autonomous evolutionary pipeline through direct code modifications. DeepEvolve Liu et al. (2025b) extends the evolution loop by incorporating Retrieval-Augmented Generation (RAG) Lewis et al. (2020); Gao et al. (2023). By retrieving academic papers from arXiv, it leverages external scientific knowledge to facilitate systematic idea generation and codebase refinement. Unlike NAS confined to combining existing modules, these LLM-driven approaches enable open-ended optimization akin to human researchers, allowing the invention of novel components beyond pre-defined design space.

*Corresponding author

Although existing LLM-driven methodologies have introduced a novel open-ended paradigm for code-level evolution, they suffer from a critical limitation: the evolution process is guided primarily by scalar metrics (e.g., accuracy) that lack diagnostic insights. Consequently, without a qualitative analysis of model behaviors or root causes of failures, these frameworks are restricted to an undirected trial-and-error search. This limitation is particularly critical in the recommendation domain. Unlike mathematical problems where correctness is defined by a deterministic ground-truth Novikov et al. (2025), a recommendation failure is multifaceted and stems from diverse root causes. For instance, a drop in NDCG does not inherently reveal whether the system suffers from excessive popularity bias or a lack of category diversity Steck (2018); Ziegler et al. (2005). Because scalar metrics condense these complex failure modes into a single numerical value, the LLM-agent cannot discern whether to mitigate bias or enhance diversity to resolve the deficiency. Thus, we argue that effective code evolution in recommender systems requires directional feedback that analyzes the root causes of failure and encapsulates user experience to effectively guide the evolution process.

To address this challenge, we propose Self-EvolveRec, a novel LLM-driven code evolution framework that orchestrates a directional feedback loop by integrating a **User Simulator** and a **Model Diagnosis Tool** to guide the evolution of recommendation systems. The User Simulator evaluates recommendation lists using diverse user personas, providing qualitative natural language feedback on potential improvements. For instance, the simulator critiques: "I seek low-cost accessories, not expensive electronics." This provides a diagnostic explanation for the failure, whereas scalar metrics only reflect a performance drop without revealing the cause. While the simulator captures user experience, relying solely on simulated feedback risks subjective bias and fails to detect underlying structural or behavioral deficiencies of the model, such as embedding collapse. To mitigate this problem, our proposed Model Diagnosis Tool serves as a deterministic verification mechanism. Unlike the simulator and standard scalar metrics (e.g., NDCG), the model diagnosis tool directly probes the model's underlying mechanisms and structural properties to quantitatively substantiate issues. For instance, the model diagnosis tool detects structural failures such as embedding collapse by analyzing the geometric distribution of item representations, which simulator feedback alone cannot verify. By corroborating qualitative user critiques with quantitative diagnostic signals, the framework accurately pinpoints critical structural deficiencies within the current recommendation pipeline.

Furthermore, as the recommendation pipeline undergoes structural evolution, static diagnostic criteria become obsolete due to the model structural mismatches, where the fixed diagnosis tool can no longer interpret the mechanisms of newly evolved components. This inadequacy arises not only from architectural incompatibility but also from the diagnosis tool's inability to quantitatively instantiate new qualitative insights provided by the simulator. To address this, we design the "Diagnosis Tool - Model Co-Evolution", ensuring that verification logic dynamically aligns with both the shifting architecture and the emerging scope of qualitative feedback. For instance, if the simulator raises a new complaint about short-term bias, the co-evolution mechanism dynamically generates a corresponding probe, such as testing how recommendations shift when the most recent interaction is removed, to verify this specific claim. Our main contributions are summarized as follows:

- We propose Self-EvolveRec, a LLM-driven code evolution framework for recommender systems, establishing a directional feedback loop. By coupling a User Simulator with a Model Diagnosis Tool, our framework resolves structural and behavioral failures that are often undetectable to scalar metrics.
- We introduce a "Diagnosis Tool - Model Co-Evolution strategy" to ensure verification reliability. By dynamically synchronizing diagnostic logic with both architectural shifts and emerging user feedback, our framework prevents evaluation criteria from becoming obsolete as the recommendation pipeline evolves.
- Our extensive experiments demonstrate that Self-EvolveRec outperforms existing NAS and LLM-driven Code Evolution baselines. Furthermore, the results validate that directional feedback leads to deterministic improvements in recommendation performance, user satisfaction, and the technical quality of the evolved algorithmic logic.

2 Problem Definition

Dataset. Let $\mathcal{D} = (\mathcal{U}, \mathcal{V}, \mathcal{E}, \mathcal{X})$ denote the dataset with user set $\mathcal{U}$ and item set $\mathcal{V}$. The interaction set $(u, v, t_{u,v}, r_{u,v}, \text{rev}_{u,v}) \in \mathcal{E}$ indicates that user $u \in \mathcal{U}$ interacted with item $v \in \mathcal{V}$ at timestamp $t_{u,v}$, providing a rating $r_{u,v}$ and a textual review $\text{rev}_{u,v}$. Based on $\mathcal{E}$, we define the interaction history for user u as $\mathcal{H}_u = \{v \mid (u, v, t_{u,v}) \in \mathcal{E}\}$, chronologically ordered by $t_{u,v}$. Additionally, $x_v \in \mathcal{X}$ denotes the set of side information, where x_v represents the raw attribute set for item v (e.g., category, title, and price).

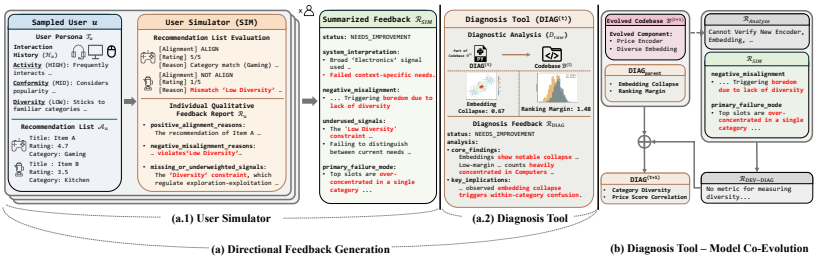

Figure 1: Two core mechanism of Self-EvolveRec. (a) is the overview of the Directional Feedback Generation: (a.1) is the user simulator, (a.2) is the model diagnosis tool. (b) is the Diagnosis Tool - Model Co-evolution.

Optimization Goal. Let $\mathcal{B} \in \mathbb{S}$ denote the entire codebase governing the *recommendation pipeline* within the **open-ended program space** $\mathbb{S}$. The evolutionary process starts with a *seed codebase* $\mathcal{B}^{(0)}$ and proceeds through T iterations, where $\mathcal{B}^{(t)}$ represents the evolved codebase at iteration t. Specifically, $\mathcal{B}^{(0)}$ constitutes a fully functional seed recommendation pipeline, encapsulating a seed recommender architecture $f_{\mathcal{B}^{(0)}}(\cdot; \theta_{\mathcal{B}^{(0)}})$ (e.g., NCF He et al. (2017)), data processing logic (e.g., basic loaders for interactions $\mathcal{E}$ and raw attributes $\mathcal{X}$), and a standard optimization loop (e.g., loss computation, and parameter updates). Our goal is to evolve this seed codebase $\mathcal{B}^{(0)}$ into an optimal codebase $\mathcal{B}^*$ that maximizes a standard recommendation metric $\mathcal{M}$ (e.g., Hit Ratio, NDCG) within T iterations. We formulate this code evolution task as a bi-level optimization problem:

$$\mathcal{B}^* = \underset{\mathcal{B} \in \mathbb{S}}{\operatorname{argmax}} \underbrace{\mathcal{M}\left(f_{\mathcal{B}}(\mathcal{E}_{\text{val}}, \mathcal{X}; \theta^*_{\mathcal{B}})\right)}_{\text{score}(\mathcal{B})} \quad \text{s.t. } \theta^*_{\mathcal{B}} = \underset{\theta}{\operatorname{argmin}} \mathcal{L}_{\mathcal{B}}\left(f_{\mathcal{B}}(\mathcal{E}_{\text{train}}, \mathcal{X}; \theta)\right) \tag{1}$$

where $\mathcal{L}_{\mathcal{B}}$ represents the loss function defined within $\mathcal{B}$. Here, $\theta^*_{\mathcal{B}}$ indicates the optimal model parameters for the architecture defined by $\mathcal{B}$, learned on the training set $\mathcal{E}_{\text{train}}$ by minimizing $\mathcal{L}_{\mathcal{B}}$.

3 Proposed Framework: Self-EvolveRec

In this section, we propose Self-EvolveRec, a novel framework designed to enable LLMs to perform directional feedback-based evolutionary code optimization for recommender systems. As illustrated in Figure 1, Self-EvolveRec operates through two core mechanisms. **First**, we establish *Directional Feedback Generation* (Sec. 3.1, Figure 1 (a)), which integrates a *User Simulator* (Sec. 3.1.1) and a *Model Diagnosis Tool* (Sec. 3.1.2) to provide qualitative and quantitative guidance. By directional feedback, Self-EvolveRec enables the LLM agent to identify the root causes of failures in the recommendation pipeline beyond numerical performance alone. This feedback then guides the *Evolution Pipeline* (Sec. 3.2, Figure 2) to perform precise, deterministic code modifications. **Second**, we introduce *Diagnosis Tool - Model Co-evolution* (Sec. 3.3, Figure 1 (b)), a strategy that ensures the diagnosis tool to dynamically adapt to the structural changes of the pipeline, maintaining the validity of the feedback loop throughout the evolution process.

3.1 Directional Feedback Generation

While scalar metrics (e.g., NDCG) quantify how well a recommendation pipeline performs, they lack the semantic depth to explain why it fails or how to resolve it. To address this, we introduce **Directional Feedback**, which integrates qualitative critiques from User Simulator and quantitative verifications from Model Diagnosis Tool. This mechanism translates non-interpretable numerical metrics into actionable insights, enabling LLMs to pinpoint and resolve deficiencies within pipeline.

3.1.1 User Simulator: Qualitative Critique

We employ a User Simulator (SIM) to complement standard metrics with qualitative directional feedback. While scalar metrics (e.g., NDCG) quantify *how well* a model performs, they fail to reveal the root causes of failure McNee et al. (2006) such as insufficient category diversity or an inability to capture short-term interests. By adopting the agentic paradigm Zhang et al. (2024); Ma et al. (2025), as depicted in Figure 1 (a.1), our simulator acts as a diverse set of virtual users, offering explicit natural language critiques that pinpoint specific deficiencies in the recommendation pipeline.

To ensure behavioral realism, we characterize each simulated user u through a structured persona $\mathcal{T}_u$. This persona is constructed by combining the user's interaction history $\mathcal{H}_u$ with sociopsychological traits Zhang et al. (2024); Mezghani et al. (2012). Following previous studies Zhang et al. (2024), we primarily utilize three key traits: (1) **Activity**, (2) **Conformity**, and (3) **Diversity**. Based on these definitions, we compute numerical scores for each trait and categorize them into three levels (i.e., LOW, MID, HIGH) using quantile-based thresholds. Further details on calculating these traits are described in App. B. These traits condition the LLM to exhibit distinct behavioral patterns,

ranging from passive users to highly active users with diverse interests. We conduct experiments on alternative traits like the Big Five Ma et al. (2025); Goldberg (1992), as explored in App. F.4.

The feedback generation process proceeds in two steps. First, SIM conducts an individual assessment for a sampled user u by evaluating the recommendation list $\mathcal{A}_u$ from a trained recommender $f_{\mathcal{B}}$ (e.g., NCF) based on their persona $\mathcal{T}_u$ and interaction history $\mathcal{H}_u$. This assessment identifies specific behavioral or semantic misalignment, which is formalized into an individual qualitative feedback report $R_u = \text{LLM}(\mathcal{I}_{\text{SIM}}, \mathcal{T}_u, \mathcal{H}_u, \mathcal{A}_u)$, where $\mathcal{I}_{\text{SIM}}$ is instruction for user simulator[1]. Second, to mitigate individual user bias and capture common failure patterns, we summarize reports from a set of sampled users $\mathcal{U}_{\text{sample}}$ into a comprehensive summary:

$$\mathcal{R}_{\text{SIM}} = \text{LLM}(\mathcal{I}_{\text{SUMMARIZE}}, \{R_u \mid u \in \mathcal{U}_{\text{sample}}\}) \tag{2}$$

where $\mathcal{I}_{\text{SUMMARIZE}}$ guides the LLM to abstract common failure patterns from individual critiques.

3.1.2 Model Diagnosis Tool: Quantitative Verification

While the SIM generates qualitative feedback from user experience, relying solely on this feedback fails to detect hidden structural issues such as embedding collapse. To address this, we introduce the Model Diagnosis Tool (DIAG; denoted as $\text{DIAG}^{(t)}$ at iteration t), which is a computational probing module designed to verify structural or behavioral deficiencies of the recommender. As depicted in Figure 1 (a.2), unlike the SIM, the $\text{DIAG}^{(t)}$ directly accesses the model parameters θ and data loaders within the codebase $\mathcal{B}^{(t)}$ to conduct a systematic validity check. As the seed diagnosis tool $\text{DIAG}^{(0)} \in \mathcal{B}^{(0)}$, we implement two foundational probes to detect common structural failures:

1) Embedding Collapse: To detect a state where representations degenerate into a narrow subspace losing discriminative power, DIAG computes the mean pairwise cosine similarity across sampled item embeddings. A high similarity score serves as a proxy for representation degeneration.

2) Ranking Margin: DIAG analyzes the ranking margin $\Delta_{u,v} = s(u, v) - s(u, v')$ for all users $u \in \mathcal{U}$ and their observed interactions $v \in \mathcal{H}_u$. Here, $s(u, v)$ and $s(u, v')$ denote the predicted logits for the ground-truth item v and a randomly sampled negative item $v' \notin \mathcal{H}_u$, respectively. To assess overall discriminative power, DIAG computes the global average margin $\mathbb{E}_{u \in \mathcal{U}, v \in \mathcal{H}_u}[\Delta_{u,v}]$. Specifically, a high margin indicates robust discrimination between ground-truth item and negative item, whereas a low or negative margin indicates a failure to distinguish ground-truth.

These probes generate a set of raw numerical diagnostics analysis $D_{\text{raw}} = \text{DIAG}^{(t)}(\mathcal{B}^{(t)})$. Subsequently, to bridge the gap between numerical diagnostics and algorithmic solutions, an LLM acts as a senior researcher to interpret these signals, converting them into a structured diagnosis report:

$$\mathcal{R}_{\text{DIAG}} = \text{LLM}(\mathcal{I}_{\text{DIAG}}, D_{\text{raw}}) \tag{3}$$

Furthermore, DIAG serves as a verification mechanism to check whether the qualitative deficiencies pointed out by SIM have actually resolved (detailed in Sec. 3.3).

3.2 Evolution Pipeline

In this section, we detail the iterative execution workflow of Self-EvolveRec, designed to autonomously refine the codebase $\mathcal{B}$ through cycles of evaluation, reasoning, and evolution. To support these cycles, Self-EvolveRec adopts a population-based exploration strategy Tanese et al. (1989); Liu et al. (2025b); Romera-Paredes et al. (2024); Novikov et al. (2025) by maintaining an Evolutionary Archive $\mathcal{P}^{(t)}$ that stores the comprehensive history of codebases and feedback. At each iteration, a target codebase is selected from this archive to serve as the parent, denoted as $\mathcal{B}_{\text{parent}} \sim \text{Sample}(\mathcal{P}^{(t)})$. Unlike existing studies Novikov et al. (2025); Liu et al. (2025b) that rely solely on scalar metrics, Self-EvolveRec actively integrates the **Directional Feedback** mechanisms (Sec. 3.1). By integrating qualitative insights from the *user feedback report* ($\mathcal{R}_{\text{SIM}}$) and quantitative findings from the *diagnosis report* ($\mathcal{R}_{\text{DIAG}}$), Self-EvolveRec shifts the focus from mere scalar metrics to pinpoint the root causes of failure. As illustrated

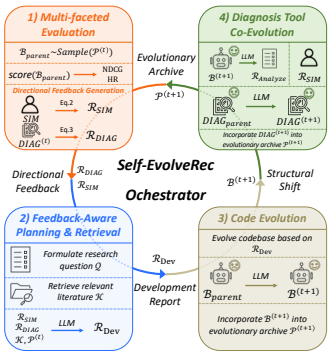

Figure 2: Overall evolutionary pipeline of Self-EvolveRec.

[1] Throughout the paper, we denote the task-specific instructions guiding the LLM-agent as $\mathcal{I}_{\text{task}}$ (e.g., $\mathcal{I}_{\text{SIM}}$, $\mathcal{I}_{\text{PLAN}}$, $\mathcal{I}_{\text{CODE}}$). While we describe the high-level objective of each instruction within the main text, the exact prompt templates are provided in App. G.1.

in Figure 2, Self-EvolveRec follows a four-phase workflow to address these failures and evolve the codebase. We first detail the three-phase workflow dedicated to refinement of the recommendation codebase, i.e., 1), 2), and 3) in Figure 2 ($\mathcal{B}_{\text{parent}} \rightarrow \mathcal{B}^{(t+1)}$)[2], followed by the Diagnosis Tool Co-Evolution stage, i.e., 4) in Figure 2 (Sec. 3.3).

1) Multi-faceted Evaluation. First, Self-EvolveRec evaluates the currently selected codebase $\mathcal{B}_{\text{parent}}$ to obtain the scalar score shown in Equation 1 (i.e., $\mathsf{score}(\mathcal{B}_{\text{parent}})$), while simultaneously generating the user feedback $\mathcal{R}_{\text{SIM}}$ and diagnosis report $\mathcal{R}_{\text{DIAG}}$[3]. These outputs serve as the directional context for the subsequent planning.

2) Feedback-Aware Planning & Retrieval. Self-EvolveRec introduces a *feedback-aware planning & retrieval* that shifts the RAG methods in previous work Liu et al. (2025b) from general-purpose knowledge gathering to targeted failure resolution. By conditioning on directional feedback from $\mathcal{R}_{\text{SIM}}$ and $\mathcal{R}_{\text{DIAG}}$, the LLM agent acts as a planner to formulate precise feedback targeted research queries $\mathcal{Q} = \text{LLM}(\mathcal{I}_{\text{PLAN}}, \mathcal{R}_{\text{SIM}}, \mathcal{R}_{\text{DIAG}}, \mathcal{P}^{(t)})$. Conditioned on specific failure modes and the evolutionary archive $\mathcal{P}^{(t)}$, the agent generates targeted queries $\mathcal{Q}$ (e.g., "Retrieve methods to mitigate category mismatch identified in user reports") to retrieve relevant academic literature $\mathcal{K}$ from online sources, including arXiv. These targeted queries ensure a retrieval focusing on resolving the identified structural deficiencies, rather than broad algorithmic exploration. These insights are then integrated into a structured *Development Report*:

$$\mathcal{R}_{\text{Dev}} = \text{LLM}(\mathcal{I}_{\text{REPORT}}, \mathcal{R}_{\text{SIM}}, \mathcal{R}_{\text{DIAG}}, \mathcal{P}^{(t)}, \mathcal{K}) \tag{4}$$

where $\mathcal{R}_{\text{Dev}}$ outlines the algorithmic modifications required to address the identified issues.

3) Code Evolution. Guided by $\mathcal{R}_{\text{Dev}}$, the agent implements code-level modifications to instantiate the updated codebase $\mathcal{B}^{(t+1)} = \text{LLM}(\mathcal{I}_{\text{CODE}}, \mathcal{R}_{\text{Dev}}, \mathcal{B}_{\text{parent}}, \mathcal{P}^{(t)})$. Upon successful execution, the new codebase $\mathcal{B}^{(t+1)}$ is incorporated into the population history $\mathcal{P}^{(t+1)}$, following standard evolutionary protocols Novikov et al. (2025); Liu et al. (2025b). This cycle recursively refines $\mathcal{B}$ to optimize the objective defined in Equation 1.

3.3 Diagnosis Tool - Model Co-Evolution

The evolution of the DIAG (i.e., 4) in Figure 2) is driven by two objectives: **First**, as $\mathcal{B}$ undergoes structural transformations, such as the introduction of new loss functions or architectural layers, a static DIAG inevitably becomes obsolete. DIAG must continuously adapt to analyze these new components effectively. **Second**, DIAG serves to quantitatively verify the qualitative insights provided by the SIM. While the SIM offers rich, human-like critiques that traditional metrics fail to capture, these critiques must be translated into measurable metrics to confirm their validity and assess their impact for precise code optimization. Therefore, DIAG evolves to stay compatible with the evolved codebase while formulating specific metrics that mathematically capture the essence of the SIM's feedback. As depicted in Figure 1 (b), if the SIM reports "boredom due to lack of diversity", DIAG autonomously implements a tailored metric (e.g., measuring diversity among top-k items) to quantify this feedback precisely, ensuring that the subsequent evolution is guided by concrete objectives.

To address this, we implement a co-evolution mechanism that begins with understanding the structural shifts. Since the codebase is continuously updated ($\mathcal{B}_{\text{parent}} \rightarrow \mathcal{B}^{(t+1)}$), the agent first scans the new codebase $\mathcal{B}^{(t+1)}$ and original $\text{DIAG}_{\text{parent}}$[4] to generate a structural analysis report:

$$\mathcal{R}_{\text{Analyze}} = \text{LLM}(\mathcal{I}_{\text{Analyze}}, \mathcal{B}^{(t+1)}, \text{DIAG}_{\text{parent}}) \tag{5}$$

The $\mathcal{R}_{\text{Analyze}}$ summarizes key information such as the updated execution flow, newly added modules, and modified loss functions. This blueprint enables the agent to design diagnostic criteria that are structurally compatible with the new architecture.

Subsequently, to synchronize the diagnosis tool with both the structural changes and the user feedback $\mathcal{R}_{\text{SIM}}$, the agent executes the evolution cycle. By integrating the qualitative $\mathcal{R}_{\text{SIM}}$ with the structural blueprint $\mathcal{R}_{\text{Analyze}}$, the agent identifies evaluation gaps (e.g., cannot verify new encoder, and embedding). Similar to the main pipeline (Sec. 3.2), it retrieves relevant methodologies $\mathcal{K}_{\text{DIAG}}$ using research queries $\mathcal{Q}_{\text{DIAG}} = \text{LLM}\left(\mathcal{I}_{\text{PLAN-DIAG}}, \mathcal{R}_{\text{SIM}}, \mathcal{R}_{\text{Analyze}}, \mathcal{P}^{(t)}\right)$ and then updates the DIAG:

[2] The Model Diagnosis Tool is also part of the codebase $\mathcal{B}$. However, its evolution is conducted separately as described in Sec. 3.3. Thus, although Sec. 3.2 focuses on evolving $\mathcal{B}$ excluding the diagnosis tool, we do not explicitly distinguish them for notational convenience.

[3] To maximize efficiency, if the selected codebase $\mathcal{B}_{\text{parent}}$ is already recorded in the archive $\mathcal{P}^{(t)}$ with complete evaluation logs, Self-EvolveRec retrieves the cached results instead of re-executing the evaluation process.

[4] The agent retrieves the corresponding model diagnosis tool $\text{DIAG}_{\text{parent}}$ that was originally paired with $\mathcal{B}_{\text{parent}}$ from the $\mathcal{P}^{(t)}$.

Table 1: Overall model performance (S: Seed Model, A: AlphaEvolve, D: DeepEvolve).

Dataset	Metric	NAS		Seed Recommender: NCF				Seed Recommender: NGCF				Seed Recommender: SASRec				Seed Recommender: MoRec			
		AutoFIS	NASRec	S	A	D	**Ours**	S	A	D	**Ours**	S	A	D	**Ours**	S	A	D	**Ours**
CDs	NDCG@5	0.2077	0.2026	0.2312	0.2556	0.2421	**0.2723**	0.3446	0.3449	0.3280	**0.3721**	0.3559	0.3528	0.3610	**0.3865**	0.2558	0.2414	0.3701	**0.3977**
	HR@5	0.3040	0.2989	0.2979	0.3477	0.3497	**0.3799**	0.4924	0.4896	0.4465	**0.4974**	0.4676	0.4623	0.4870	**0.5274**	0.3779	0.3607	0.4864	**0.5340**
Electronics	NDCG@5	0.1753	0.1706	0.1078	0.1183	0.1817	**0.1907**	0.1531	0.1808	0.1726	**0.1925**	0.2325	0.2063	0.2508	**0.2600**	0.1883	0.1912	0.1938	**0.2056**
	HR@5	0.2456	0.2444	0.1610	0.1733	0.2600	**0.2714**	0.2247	0.2590	0.2385	**0.2759**	0.3208	0.2891	0.3427	**0.3591**	0.2724	0.2675	0.2745	**0.2921**
Office	NDCG@5	0.1714	0.1377	0.1620	0.1751	0.1705	**0.1759**	0.1711	0.1799	0.1805	**0.1930**	0.1799	0.1939	0.1816	**0.2329**	0.1851	0.1848	0.1697	**0.1884**
	HR@5	0.2490	0.2070	0.2343	0.2523	0.2548	**0.2659**	0.2494	0.2634	0.2591	**0.2743**	0.2526	0.2750	0.2631	**0.3218**	0.2689	0.2633	0.2447	**0.2703**
MovieLens	NDCG@5	0.1369	0.2916	0.3413	0.3465	0.1908	**0.3764**	0.1824	0.2355	0.2010	**0.3588**	0.5667	0.5583	0.5722	**0.5765**	0.3796	0.4993	0.5281	**0.5460**
	HR@5	0.2099	0.4346	0.4970	0.5091	0.2735	**0.5475**	0.2876	0.3672	0.3162	**0.5220**	0.7283	0.7199	0.7344	**0.7366**	0.5414	0.6743	0.6967	**0.7131**

$$\mathcal{R}_{\text{Dev-DIAG}} = \text{LLM}\left(\mathcal{I}_{\text{REPORT-DIAG}}, \mathcal{R}_{\text{SIM}}, \mathcal{R}_{\text{Analyze}}, \mathcal{K}_{\text{DIAG}}\right), \tag{6}$$
$$\text{DIAG}^{(t+1)} = \text{LLM}\left(\mathcal{I}_{\text{CODE-DIAG}}, \mathcal{R}_{\text{Dev-DIAG}}, \mathcal{B}^{(t+1)}, \text{DIAG}_{\text{parent}}, \mathcal{P}^{(t)}\right)$$

The new model diagnosis tool $\text{DIAG}^{(t+1)}$ is incorporated into the population history $\mathcal{P}^{(t+1)}$. This adaptive process guarantees that $\text{DIAG}^{(t+1)}$ is equipped with both the logic to inspect new architectures and the specific metrics required to transform the simulator's qualitative feedback into actionable, quantitative signals. We further investigate the additional evolutionary pipeline of the User Simulator in experiments in App. F.3.

4 EXPERIMENTS

Datasets. For evaluations, we used three Amzaon datasets Hou et al. (2024) (CDs, Electronics, and Office) and the MovieLens dataset Harper & Konstan (2015). Following prior works Kang & McAuley (2018); Sun et al. (2019), we use five-core datasets, ensuring that each user and item has at least five interactions. Detailed statistics for each dataset are provided in Table 5 in App. D.

Baselines. We evaluate Self-EvolveRec against representative NAS-based recommender architecture search methods, including AutoFIS Liu et al. (2020) and NASRec Zhang et al. (2023), as well as recent LLM-driven evolutionary frameworks such as AlphaEvolve Novikov et al. (2025) and DeepEvolve Liu et al. (2025b). Details regarding the baseline methods are provided in App. E.1.

Evaluation Protocol. We use the leave-last-out strategy Kang & McAuley (2018); Sun et al. (2019) for evaluation or recommender models, where we use the most recent item and second most item for testing and validation, respectively, and the remaining history for training. Each test item is paired with 99 random sampled non-interacted items. We report performance using two standard metrics: Normalized Discounted Cumulative Gain (NDCG@5) and Hit Ratio (HR@5). To ensure fair comparisons, we employ GPT-5-mini for planning and retrieval, while utilizing GPT-5 as the coding agent for all frameworks. Please refer to the App. C for more details regarding the implementation.

4.1 PERFORMANCE COMPARISON

To demonstrate the effectiveness of our framework, we evaluate Self-EvolveRec through three distinct perspectives. First, we compare Self-EvolveRec against various baselines using traditional metrics (NDCG and HR), to measure numerical ranking performance (Sec. 4.1.1). Second, we conduct a user satisfaction analysis using an agentic user simulator to evaluate how effectively the evolved recommender satisfies complex user preferences, providing a complementary perspective to ranking accuracy (Sec. 4.1.2). Finally, we analyze the creativity, explicitness, insight, and personalization of the evolved codebases using LLM-as-a-Judge Zheng et al. (2023a) to evaluate the quality of algorithmic improvements (Sec. 4.1.3).

4.1.1 OVERALL PERFORMANCE.

The results of the recommendation task on four datasets are given in Table 1. From the results, we have the following observations: **1)** Self-EvolveRec consistently outperforms AlphaEvolve and DeepEvolve across all datasets, regardless of the seed model. These results demonstrate that integrating qualitative critiques with quantitative diagnostics enables effective algorithmic evolution across diverse recommender architectures. **2)** AlphaEvolve and DeepEvolve exhibit inconsistency across datasets, with both baselines occasionally underperforming the initial seeds. This inconsistency indicates that without directional feedback, these baselines are limited to trial-and-error searches. Conversely, Self-EvolveRec leverages semantic critiques to identify and resolve specific structural deficiencies, leading to more informed and effective improvements. **3)** Self-EvolveRec significantly surpasses NAS-based baselines, such as AutoFIS and NASRec, in all datasets. This confirms that shifting from a closed, human-defined operator pool to an open-ended program space allows for more expressive logic and structural algorithmic refinement.

4.1.2 USER SATISFACTION ANALYSIS.

To bridge the gap between static metrics and actual user satisfaction, we adopt the agentic simulation environment from Agent4Rec Zhang et al. (2024) and PUB Ma et al. (2025) as a scalable proxy for A/B testing. In environment, generative agents driven by distinct traits (e.g., Big Five) are presented with pages of four items and decide whether to continue or terminate the session based on relevance and diversity. Following Agent4Rec, we quantify this simulation using three key metrics: **View** (item view ratio), **Satisfy** (a comprehensive score about recommender, 1 to 10), and **Depth** (the number of pages explored before termination). Notably, in Agent4Rec, higher **Depth** reflects superior user retention and relevance, as agents terminate sessions upon encountering unsatisfactory items. The results of the user satisfaction are summarized in Table 2. From the results, we have the following observations: **1)** Self-EvolveRec consistently outperforms all baselines across all satisfaction-oriented metrics (i.e., View, Satisfy, and Depth) regardless of the seed model. These results demonstrate that the directional feedback from the user simulator (i.e., $\mathcal{R}_{\text{SIM}}$) enables models to evolve beyond mere accuracy, substantially improving the perceived quality of recommendations from a user-centric perspective, and sustaining user engagement for longer durations. **2)** In contrast, baselines show suboptimal generalization to user-centric metrics and, in some cases, even underperform initial models. This suggests that scalar metric-only optimization is insufficient to capture complex dynamics of the user experience. Without guidance, evolved models tend to overfit to narrow numerical targets, which degrades overall user satisfaction.

Table 2: Multi-facet satisfaction metrics on SASRec and NCF under two agentic evaluators (Agent4Rec and PUB).

Seed Recommender	Agentic Evaluator	Metric	CDs				Electronics			
			S	A	D	**Ours**	S	A	D	**Ours**
SASRec	Agent4Rec	View	0.372	0.378	0.379	**0.381**	0.335	0.342	0.351	**0.353**
		Satisfy	4.606	4.384	4.710	**5.046**	4.173	4.308	4.487	**4.502**
		Depth	1.926	1.830	1.952	**2.048**	1.754	1.778	1.782	**1.797**
	PUB	View	0.128	0.122	0.130	**0.136**	0.133	0.134	0.141	**0.144**
		Satisfy	4.630	4.506	4.810	**4.906**	3.928	3.972	4.082	**4.134**
		Depth	1.912	1.910	1.922	**2.018**	1.856	1.888	1.904	**1.928**
NCF	Agent4Rec	View	0.365	0.354	0.369	**0.372**	0.339	0.337	0.307	**0.342**
		Satisfy	4.206	4.461	4.392	**4.650**	4.320	4.270	4.132	**4.354**
		Depth	1.776	1.882	1.840	**1.934**	1.786	1.744	1.720	**1.798**
	PUB	View	0.125	0.127	0.119	**0.134**	0.135	0.125	0.124	**0.139**
		Satisfy	4.488	4.650	4.422	**4.770**	3.842	3.826	3.564	**3.952**
		Depth	1.914	1.918	1.940	**1.952**	1.874	1.768	1.754	**1.916**

4.1.3 CODEBASE QUALITY EVALUATION.

To verify substantive algorithmic improvements beyond numerical gains, we employ an LLM-as-a-judge (GPT-5) to evaluate the code of the recommender model in all evolved codebases presented in Table 1 on a 1-10 scale against the recommender model in the seed codebase. The evaluation covers four dimensions: **Creativity** (novel mechanisms beyond simple parameter tuning), **Explicitness** (interpretability of logic flows), **Insight** (logical intention to resolve specific failures, e.g., popularity bias), and **Personalization** (user-context awareness). Detailed prompts for these criteria are provided in App. G.2. From the results in Figure 3, we have the following observations: **1)** Self-EvolveRec consistently achieves the highest scores across all criteria. Specifically, the high scores in Creativity and Insight validate effectiveness of our directional feedback loop, which integrating SIM and DIAG, enables targeted logic refinement, whereas scalar-driven baselines rely on inefficient trial-and-error without understanding underlying problems in codebase. **2)** Self-EvolveRec shows a substantial gain in Personalization. This superiority is driven by SIM, which forces evolution to reflect specific user needs directly into algorithmic logic. Conversely, baselines merely evolve models based on numerical metrics, failing to address specific user needs. **3)** AlphaEvolve exhibits inferior performance across all criteria due to the absence of external knowledge. Unlike RAG-based methods (DeepEvolve and Self-EvolveRec), its reliance on internal knowledge restricts the discovery of novel mechanisms, underscoring the necessity of retrieving external insights for open-ended development.

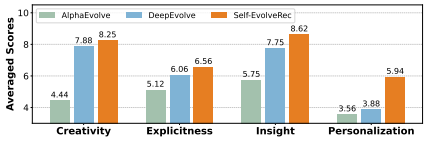

Figure 3: LLM-as-a-Judge evaluation of the evolved models.

4.2 ABLATION STUDIES

To comprehensively evaluate the impact of each component in Self-EvolveRec, we conduct ablation studies in Table 3. Note that Row (6) represents the complete framework of Self-EvolveRec. We have the following observations: Across all datasets, **1)** Introducing either the SIM (Row (2)) or the DIAG (Row (3)) consistently yields performance gains over the scalar metric only method (Row (1)). Notably, the integration of the SIM results in a more substantial increase, underscoring its critical role in providing directional feedback by pinpointing specific recommendation failures, which scalar metrics fail to capture. **2)** Comparing Row (3) and Row (5) demonstrates that static diagnostic criteria limit the potential of code evolution. Since the model architecture continuously shifts, a fixed DIAG inevitably becomes obsolete. The performance gain in Row (5) implies that dynamically synchronizing the diagnostic logic of DIAG via co-evolution is essential for maintaining a reliable feedback loop. **3)** Integrating both the SIM and the DIAG (Row (4)), achieves superior performance among all ablated variants, notably outperforming even the co-evolved DIAG without user feedback (Row (5)). This underscores the importance of combining qualitative user critiques and quantitative diagnostics, which provide recommendation failures and internal structural issues.

Table 3: Ablation studies on the components of Self-EvolveRec (Seed Recommender: SASRec).

Row	Component			CDs		Electronics	
	SIM	DIAG	Co-Evolve	NDCG@5	HR@5	NDCG@5	HR@5
(1)	✗	✗	✗	0.3610	0.4870	0.2508	0.3427
(2)	✓	✗	✗	0.3751	0.5102	0.2573	0.3551
(3)	✗	✓	✗	0.3676	0.4791	0.2515	0.3449
(4)	✓	✓	✗	0.3789	0.5164	0.2584	0.3566
(5)	✗	✓	✓	0.3727	0.5014	0.2532	0.3520
(6)	✓	✓	✓	**0.3865**	**0.5274**	**0.2600**	**0.3591**

Table 4: Performance comparison under extreme initialization scenarios (Random and Ensemble). Peak indicates the iteration number of the best performance.

Dataset	Seed Recommender		AlphaEvolve			DeepEvolve			**Ours**		
	NDCG@5	HR@5	NDCG@5	HR@5	Peak	NDCG@5	HR@5	Peak	NDCG@5	HR@5	Peak
Seed Recommender: Random											
CDs	0.0312	0.0525	0.3430	0.4549	15	0.3766	0.4963	13	**0.3883**	**0.5165**	**8**
Electronics	0.0310	0.0531	0.2037	0.2878	19	0.1972	0.2761	18	**0.2109**	**0.2952**	**11**
Seed Recommender: NCF+NGCF+SASRec											
CDs	0.3946	0.5179	0.3864	0.5075	FAIL	0.3695	0.5002	17	**0.4105**	**0.5409**	**9**
Electronics	0.2385	0.3246	0.2496	0.3386	14	0.2353	0.3240	FAIL	**0.2524**	**0.3426**	**6**

Furthermore, the complete framework of Self-EvolveRec (Row (6)) achieves the best performance, validating that dynamically aligning the DIAG with both the shifting architecture and the qualitative feedback is necessary to ensure the efficacy of the feedback loop and the evolution of the model.

4.3 Model Analysis

4.3.1 Adaptability to extreme initialization scenarios.

We evaluate Self-EvolveRec's adaptability under extreme initializations: a Random recommender (starting from scratch), and a Ensemble (NCF + NGCF + SASRec) reflecting a sophisticated industrial deployment. We have the following observations in Table 4: **1)** Even starting from random, Self-EvolveRec successfully evolves a fully functional and highly competitive recommender pipeline. Self-EvolveRec reaches peak performance in 8 to 11 iterations, significantly faster than baselines (13 to 19 iterations). This confirms that directional feedback systematically constructs valid pipelines by resolving structural deficiencies, avoiding the aimless trial-and-error of scalar-driven methods. **2)** In ensemble setting, achieving further gains is difficult due to the high initial performance and structural complexity. We observe that scalar-only baselines, struggle to navigate structural complexity of the ensemble. In contrast, Self-EvolveRec consistently identifies and resolves latent bottlenecks within ensemble, demonstrating that directional feedback is effective even in structurally complex system. These results demonstrate Self-EvolveRec as a comprehensive solution for industrial lifecycle, which involves building new services and refining complex ensemble models for established services Cheng et al. (2016); Gomez-Uribe & Hunt (2016); Wang et al. (2022).

4.3.2 Case Study: Validating Reliability of Co-evolved Diagnostic Tool

To verify the functional reliability and interpretability of the co-evolved DIAG, we conduct a case study through an induced deficiency experiment. Specifically, we evaluate whether the autonomously generated diagnostic metrics can accurately pinpoint logic-level deficiencies intentionally injected in the evolved models. As shown in Figure 4, we injected deficiencies into an evolved SASRec to force order-insensitivity by: (i) removing positional embeddings, (ii) bypassing the exponential decay module. The co-evolved DIAG successfully identifies these deficiencies via newly generated metrics, **Swap Sensitivity** (ranking shifts after swapping last two items in the sequence) and **Logit Δ Swap** (deviation in predicted logits for the target item after the swap), which were not present in the initial seed DIAG. While the normal model is order-sensitive, the injected version shows near-zero sensitivity, correctly diagnosed as "Order-insensitive" in $\mathcal{R}_{\text{DIAG}}$. Notably, this diagnostic signal directly correlates with the sharp performance decline, demonstrating that co-evolved DIAG can explain the root causes of low performance through structural verification, highlighting the pivotal role of Diagnosis Tool - Model Co-Evolution in maintaining a grounded feedback loop. Additional case studies are provided in App. F.5.

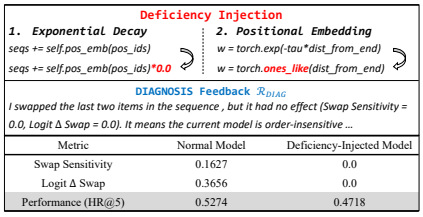

Figure 4: Case study on Diagnosis Tool - Model Co-Evolution on CDs dataset (Seed Recommender: SASRec).

5 Conclusion

In this paper, we propose a novel LLM-driven code evolution framework, named Self-EvolveRec. The main idea is to overcome the limitations of existing scalar metric-based code optimization by establishing a directional feedback loop that integrates qualitative critiques from a User Simulator with quantitative verification from a Model Diagnosis Tool. By doing so, Self-EvolveRec significantly outperforms evolutionary baselines based on NAS and LLM in both recommendation accuracy and user satisfaction. Moreover, we demonstrate the indispensability of our Diagnosis Tool - Model Co-Evolution strategy, which ensures that diagnostic criteria dynamically adapt to structural shifts, maintaining a grounded and reliable feedback loop throughout the process. In future work, we plan to address computational overhead associated with iterative training and evaluation of evolved models. We aim to explore more efficient evaluation protocols, such as predicting performance directly from architectural descriptions using LLMs Jawahar et al. (2024), to accelerate evolutionary cycle.

REFERENCES

Mehdi Ben Ayed, Fei Feng, Jay Adams, Vishwakarma Singh, Kritarth Anand, and Jiajing Xu. Recomind: A reinforcement learning framework for optimizing in-session user satisfaction in recommendation systems. *arXiv preprint arXiv:2508.00201*, 2025.

Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pp. 7–10, 2016.

Yuanzheng Ci, Chen Lin, Ming Sun, Boyu Chen, Hongwen Zhang, and Wanli Ouyang. Evolving search space for neural architecture search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 6659–6669, 2021.

Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. RecSys '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340359. doi: 10.1145/2959100.2959190. URL https://doi.org/10.1145/2959100.2959190.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

Francesco Fabbri, Gustavo Penha, Edoardo D'Amico, Alice Wang, Marco De Nadai, Jackie Doremus, Paul Gigioli, Andreas Damianou, Oskar Stål, and Mounia Lalmas. Evaluating podcast recommendations with profile-aware llm-as-a-judge. In *Proceedings of the Nineteenth ACM Conference on Recommender Systems*, pp. 1181–1186, 2025.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2(1), 2023.

Lewis R. Goldberg. The development of markers for the big-five factor structure. *Psychological Assessment*, 4:26–42, 1992. URL https://api.semanticscholar.org/CorpusID:144709415.

Carlos A. Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. 6(4), 2016. ISSN 2158-656X. doi: 10.1145/2843948. URL https://doi.org/10.1145/2843948.

F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), December 2015. ISSN 2160-6455. doi: 10.1145/2827872. URL https://doi.org/10.1145/2827872.

Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pp. 173–182, 2017.

Yupeng Hou, Jiacheng Li, Zhankui He, An Yan, Xiusi Chen, and Julian McAuley. Bridging language and items for retrieval and recommendation. *arXiv preprint arXiv:2403.03952*, 2024.

Ganesh Jawahar, Muhammad Abdul-Mageed, Laks Lakshmanan, and Dujian Ding. Llm performance predictors are good initializers for architecture search. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 10540–10560, 2024.

Wang-Cheng Kang and Julian McAuley. Self-attentive sequential recommendation. In *2018 IEEE international conference on data mining (ICDM)*, pp. 197–206. IEEE, 2018.

Donghyun Kim, Chanyoung Park, Jinoh Oh, Sungyoung Lee, and Hwanjo Yu. Convolutional matrix factorization for document context-aware recommendation. RecSys '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340359. doi: 10.1145/2959100.2959165. URL https://doi.org/10.1145/2959100.2959165.

Ravi Krishna, Aravind Kalaiah, Bichen Wu, Maxim Naumov, Dheevatsa Mudigere, Misha Smelyanskiy, and Kurt Keutzer. Differentiable nas framework and application to ads ctr prediction. *arXiv preprint arXiv:2110.14812*, 2021.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33: 9459–9474, 2020.

Zelong Li, Jianchao Ji, Yingqiang Ge, and Yongfeng Zhang. Autolossgen: Automatic loss function generation for recommender systems. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 1304–1315, 2022.

Bin Liu, Chenxu Zhu, Guilin Li, Weinan Zhang, Jincai Lai, Ruiming Tang, Xiuqiang He, Zhenguo Li, and Yong Yu. Autofis: Automatic feature interaction selection in factorization models for click-through rate prediction. In *proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2636–2645, 2020.

Fei Liu, Xinyu Lin, Hanchao Yu, Mingyuan Wu, Jianyu Wang, Qiang Zhang, Zhuokai Zhao, Yinglong Xia, Yao Zhang, Weiwei Li, et al. Recoworld: Building simulated environments for agentic recommender systems. *arXiv preprint arXiv:2509.10397*, 2025a.

Gang Liu, Yihan Zhu, Jie Chen, and Meng Jiang. Scientific algorithm discovery by augmenting alphaevolve with deep research. *arXiv preprint arXiv:2510.06056*, 2025b.

Chenglong Ma, Ziqi Xu, Yongli Ren, Danula Hettiachchi, and Jeffrey Chan. Pub: an llm-enhanced personality-driven user behaviour simulator for recommender system evaluation. In *Proceedings of the 48th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2690–2694, 2025.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.

Sean M. McNee, John Riedl, and Joseph A. Konstan. Being accurate is not enough: how accuracy metrics have hurt recommender systems. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '06, pp. 1097–1101, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595932984. doi: 10.1145/1125451.1125659. URL https://doi.org/10.1145/1125451.1125659.

Manel Mezghani, Corinne Amel Zayani, Ikram Amous, and Faiez Gargouri. A user profile modelling using social annotations: a survey. WWW '12 Companion, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312301. doi: 10.1145/2187980.2188230. URL https://doi.org/10.1145/2187980.2188230.

Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Esteban Real, Chen Liang, David So, and Quoc Le. Automl-zero: Evolving machine learning algorithms from scratch. In *International conference on machine learning*, pp. 8007–8019. PMLR, 2020.

Sonia Roccas, Lilach Sagiv, Shalom H. Schwartz, and Ariel Knafo. The big five personality factors and personal values. *Personality and Social Psychology Bulletin*, 28:789 – 801, 2002. URL https://api.semanticscholar.org/CorpusID:144611052.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Ruslan Salakhutdinov and Andriy Mnih. Probabilistic matrix factorization. NIPS'07, Red Hook, NY, USA, 2007. Curran Associates Inc. ISBN 9781605603520.

Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL https://github.com/algorithmicsuperintelligence/openevolve.

Qingquan Song, Dehua Cheng, Hanning Zhou, Jiyan Yang, Yuandong Tian, and Xia Hu. Towards automated neural interaction discovery for click-through rate prediction. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 945–955, 2020.

Harald Steck. Calibrated recommendations. In *Proceedings of the 12th ACM Conference on Recommender Systems*, RecSys '18, pp. 154–162, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359016. doi: 10.1145/3240323.3240372. URL https://doi.org/10.1145/3240323.3240372.

Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. Bert4rec: Sequential recommendation with bidirectional encoder representations from transformer. In *Proceedings of the 28th ACM international conference on information and knowledge management*, pp. 1441–1450, 2019.

Reiko Tanese, John H. Holland, and Quentin F. Stout. *Distributed genetic algorithms for function optimization*. PhD thesis, USA, 1989. AAI9001722.

Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. Neural graph collaborative filtering. In *Proceedings of the 42nd international ACM SIGIR conference on Research and development in Information Retrieval*, pp. 165–174, 2019.

Xuesi Wang, Guangda Huzhang, Qianying Lin, and Qing Da. Learning-to-ensemble by contextual rank aggregation in e-commerce. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*, pp. 1036–1044, 2022.

Zheng Yuan, Fajie Yuan, Yu Song, Youhua Li, Junchen Fu, Fei Yang, Yunzhu Pan, and Yongxin Ni. Where to go next for recommender systems? id- vs. modality-based recommender models revisited. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2639–2649, 2023.

An Zhang, Yuxin Chen, Leheng Sheng, Xiang Wang, and Tat-Seng Chua. On generative agents in recommendation. In *Proceedings of the 47th international ACM SIGIR conference on research and development in Information Retrieval*, pp. 1807–1817, 2024.

Tunhou Zhang, Dehua Cheng, Yuchen He, Zhengxing Chen, Xiaoliang Dai, Liang Xiong, Feng Yan, Hai Li, Yiran Chen, and Wei Wen. Nasrec: weight sharing neural architecture search for recommender systems. In *Proceedings of the ACM Web Conference 2023*, pp. 1199–1207, 2023.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023a.

Ruiqi Zheng, Liang Qu, Bin Cui, Yuhui Shi, and Hongzhi Yin. Automl for deep recommender systems: A survey. *ACM Transactions on Information Systems*, 41(4):1–38, 2023b.

Cai-Nicolas Ziegler, Sean M. McNee, Joseph A. Konstan, and Georg Lausen. Improving recommendation lists through topic diversification. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pp. 22–32, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930469. doi: 10.1145/1060745.1060754. URL https://doi.org/10.1145/1060745.1060754.

Barret Zoph and Quoc Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=r1Ue8Hcxg.

Kuan Zou and Aixin Sun. A survey of real-world recommender systems: Challenges, constraints, and industrial perspectives. *arXiv preprint arXiv:2509.06002*, 2025.

A Related Work

Neural Architecture Search for Recommender Systems. Neural Architecture Search (NAS) has been increasingly explored in recommender systems to automate the design of feature interactions and model architectures. Early studies like AutoFIS Liu et al. (2020) replace discrete interaction feature choices with learnable gating to filter redundant interactions. Later methods extend NAS to backbone design. AutoCTR Song et al. (2020) uses evolutionary search to assemble operator blocks into a DAG, whereas DNAS Krishna et al. (2021) and NASRec Zhang et al. (2023) leverage weight-sharing supernets for scalability. Automation has also expanded to non-architectural components. AutoLossGen Li et al. (2022) casts loss function design as an automated search problem, using reinforcement learning (RL) to explore loss formulations composed of basic mathematical operators (e.g., addition, log, multiplication). Nevertheless, these approaches remain bounded by pre-defined operator sets and wiring rules, restricting innovation to selection/parameterization within a fixed search space rather than open-ended synthesis of new procedural algorithms.

LLM-driven Code Evolution. To overcome the limitations of fixed search spaces, recent work has explored LLM-driven code evolution, which shifts the optimization target from parameters to open-ended programs. FunSearch Romera-Paredes et al. (2024) utilizes an LLM-evaluator loop to discover interpretable algorithms for mathematical tasks, while Eureka Ma et al. (2023) automates RL reward engineering by iteratively refining code based on execution feedback. Building on this line, AlphaEvolve Novikov et al. (2025) generalizes the evolutionary loop into an autonomous coding pipeline that iteratively edits and tests code using evaluator feedback, scaling to more complex algorithmic optimization tasks. More recently, DeepEvolve Liu et al. (2025b) integrates retrieval-augmented generation, leveraging external knowledge to systematically inform hypothesis generation and implementation for scientific discovery tasks. However, most existing approaches are guided primarily by a single scalar metric (e.g., accuracy or success rate), which provides limited diagnostic insight into user-centric failure modes such as bias, off-topic, or lack of diversity, which are key considerations for holistic recommender-system optimization.

LLM-based User Simulation. LLM-based user simulation has emerged as a promising alternative to static recommendation metrics and costly online A/B testing. Initial approaches focused on realistic persona construction. Agent4Rec Zhang et al. (2024) models social traits like conformity from real-world data, while Profile-aware simulators Fabbri et al. (2025) utilize natural language summaries of user history to align with human judgment. Enhancing psychological fidelity, PUB Ma et al. (2025) further integrates Big Five personality traits Goldberg (1992); Roccas et al. (2002) to replicate diverse interaction patterns. Recent research shifts focus to utilizing user simulators for system optimization. RecoMind Ayed et al. (2025) employs the simulator as a virtual training environment to refine RL policies for long-term engagement. Meanwhile, RecoWorld Liu et al. (2025a) establishes a proactive feedback loop, where the simulator explicitly signals user states (e.g., boredom) to guide the recommender's adaptation.

B Traits

Activity (Engagement Level). Activity quantifies the degree of a user's engagement with the recommender system. Since user-item interactions are typically sparse, we define activity as the cardinality of the user's interaction history:

$$T_{\text{act}}(u) = |\mathcal{H}_u|. \tag{7}$$

Users with lower $T_{\text{act}}(u)$ values correspond to passive users who interact infrequently with recommended items, whereas higher values indicate highly engaged users with rich interaction histories.

Conformity (Mainstream Adherence). Conformity measures the extent to which a user's preferences align with global public consensus. This trait captures whether a user follows mainstream tastes or exhibits individualized preferences. We define conformity as the mean squared deviation between the user's rating $r_{u,v}$ and the global average rating $\bar{r}_v$ of item v:

$$T_{\text{conf}}(u) = \frac{1}{|\mathcal{H}_u|} \sum_{v \in \mathcal{H}_u} (r_{u,v} - \bar{r}_v)^2, \tag{8}$$

where $\bar{r}_v$ denotes the average rating of item v across all users. A lower conformity value implies that the user's preferences closely align with popular sentiment, while a higher value reflects more distinctive and personalized tastes.

Diversity (Interest Breadth). Diversity characterizes the breadth of a user's interests across item categories. We define this trait as the number of unique categories associated with the items in the user's interaction history:

$$T_{\text{div}}(u) = \left|\{c_v \mid v \in \mathcal{H}_u\}\right|. \tag{9}$$

Users with lower $T_{\text{div}}(u)$ values tend to focus on a narrow set of categories, whereas higher values indicate a preference for exploring a broader and more diverse range of categories.

We categorized each user trait into three distinct levels, defined as follows:

- **Activity**:
  - HIGH: Frequently interacts with the system and maintains a high volume of engagement with recommendations.
  - MID: Interacts moderately, primarily when items strictly align with personal preferences.
  - LOW: Rarely interacts with the system and does not interact if recommendations are not relevant to their interests.
- **Conformity**:
  - HIGH: Heavily influenced by popularity and public ratings; tends to follow mainstream trends.
  - MID: Considers both popularity and personal taste, balancing trends with individual preferences.
  - LOW: Ignores popularity and trends, evaluating items purely based on intrinsic personal preference.
- **Diversity**:
  - HIGH: Seeks high variety and novelty, enjoying the exploration of diverse categories and new styles.
  - MID: Mostly consumes preferred categories but occasionally explores similar alternatives.
  - LOW: Sticks strictly to a narrow set of familiar categories and avoids exploration.

C Implementation Details

To ensure fair comparisons, all evolution-based methods are initialized with four distinct seed recommenders: NCF He et al. (2017) (MF-based), NGCF Wang et al. (2019) (graph-based), SASRec Kang & McAuley (2018) (sequential), and MoRec Yuan et al. (2023) (multi-modal). Regarding the baseline implementation of AlphaEvolve, due to the unavailability of the official code, we utilized OpenEvolve Sharma (2025), an open-source implementation, following prior work Liu et al. (2025b). In our evolutionary framework, we set the maximum evolution steps to 21 across all LLM-driven evolutionary frameworks. For all LLM-driven evolutionary frameworks, we set the maximum evolution iterations to 21. For NAS baselines, we configured the search epochs to 5 for AutoFIS and 1 for NASRec, following the hyper-parameter setting in NASRec Zhang et al. (2023). We employ GPT-5-mini for the User Simulator (SIM) and set the number of sampled users to $|\mathcal{U}_{\text{sample}}| = 20$ (refer to App. F.1.1 for an analysis of the number of sampled users). To ensure a fair comparison, we uniformly configured all recommender models—including the retraining phase of NAS models—with a user/item embedding dimension of 50, a batch size of 128, and a learning rate of 0.001. The maximum number of epochs was set to 300 for both standard training and the NAS retraining stage. All experiments were conducted on a single NVIDIA GeForce A6000 (48GB) GPU. We provide a comprehensive efficiency analysis, including time cost per iteration and user sampling impact, in App. F.1. We include more details regarding the seed recommender in App. E.2.

D Datasets

Table 5 shows the statistics of the dataset after preprocessing.

E Baselines and Seed Recommender

E.1 Baselines

(1) Neural Architecture Search Baselines

Table 5: Statistics of datasets after preprocessing.

Dataset	CDs	Electronics	Office	MovieLens
# Users	14,335	32,232	20,147	6,040
# Items	11,436	17,695	10,470	3,952
# Interactions	126,225	257,850	145,927	1,000,209

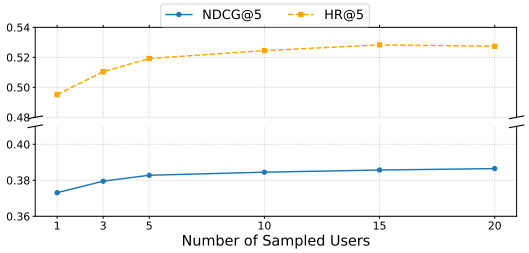

Figure 5: Recommendation performance over number of sampled user $\mathcal{U}_{\text{sample}}$ on CDs dataset (Seed Recommender: SASRec).

- AutoFIS Liu et al. (2020) automatically identifies essential feature interactions by employing learnable gates to prune redundant feature combinations.
- NASRec Zhang et al. (2023) leverages a weight-sharing supernet to efficiently search for optimal full architectures.

(2) LLM-driven Code Evolution Baselines

- AlphaEvolve Novikov et al. (2025) orchestrates an autonomous LLM-driven evolutionary pipeline to iteratively evolve and optimize algorithmic codebases.
- DeepEvolve Liu et al. (2025b) integrates retrieval-augmented generation (RAG) into the evolutionary loop to guide code optimization with external scientific knowledge.

E.2 Seed Recommender

- NCF He et al. (2017) is a pioneering neural collaborative filtering framework which combines multi-layer perceptron (MLP) to learn user-item interactions.
- NGCF Wang et al. (2019) is a graph-based model that explicitly encodes collaborative signals in the embedding space by propagating embeddings on the user-item bipartite graph
- SASRec Kang & McAuley (2018) is a sequential recommendation model leveraging self-attention mechanisms to dynamically capture user interests from interaction sequences.
- MoRec Yuan et al. (2023) is a multi-modal framework that utilizes pre-trained encoders (e.g., SBERT) to initialize item embeddings with textual features. In our experiments, we employ SASRec as the backbone architecture for MoRec

F Additional Experiments

F.1 Efficiency Analysis

F.1.1 Impact of user sampling size.

To investigate the efficiency and robustness of Self-EvolveRec on $\mathcal{U}_{\text{sample}}$, we conducted experiments by varying the user sample size of the User Simulator. As shown in Figure 5, we observe that Self-EvolveRec achieves robust performance even with a small number of sampled users. Notably, performance improves significantly as the sample size increases from 1 to 3, eventually stabilizing around a sample size of 5. This robustness stems from the **Diagnosis Tool - Model Co-Evolution** mechanism (Sec. 3.3). Although the SIM operates on a small subset of users to generate qualitative feedback (e.g., "lack of diversity"), the DIAG translates these critiques into deterministic

Table 6: Comparison of average execution time per iteration on CDs dataset (Seed Recommender: SASRec).

Method	RAG	Coding	Co-Evolve	SIM	DIAG	Total Time
AlphaEvolve	-	6m 23s	-	-	-	6m 23s
DeepEvolve	6m 16s	7m 43s	-	-	-	13m 59s
Ours	6m 24s	7m 15s	7m 06s	4m 31s*	12s	≈ 25m 28s

* The User Simulation time is measured with 20 sampled users, processed in parallel batches of 4 (5 batches $\times$ 54.24s $\approx$ 4m 31s). Diagnosis Co-Evolve includes both research (3m 54s) and coding (3m 12s).

Table 7: Performance when Feedback-aware Planning & Retrieval is removed (Seed Recommender: SASRec). C: Creativity, E: Explicitness, I: Insight, P: Personalization.

Method	CDs		Electronics		Codebase Quality			
	NDCG@5	HR@5	NDCG@5	HR@5	C	E	I	P
AlphaEvolve	0.3528	0.4623	0.2063	0.2891	4.5	5.0	6.5	3.0
DeepEvolve	0.3610	0.4870	0.2508	0.3427	7.5	6.0	7.5	4.0
Self-EvolveRec	0.3865	**0.5274**	**0.2600**	**0.3591**	**8.0**	**7.5**	**8.0**	**6.5**
w.o. Planning	**0.3988**	0.5134	0.2597	0.3568	6.5	6.0	**8.0**	4.5

numerical metrics (e.g., measuring category entropy). Consequently, even with limited user samples, the evolved DIAG effectively verifies and quantifies structural deficiencies across the global data distribution, ensuring reliable evolutionary guidance without the need for extensive user sampling.

F.1.2 Time Efficiency Analysis

We compare the execution time per iteration of evolution in Table 6. The reported times are averaged over the evolution of SASRec on the CDs dataset. We have the following observations: **1)** Although Self-EvolveRec requires approximately 25 minutes per iteration—higher than AlphaEvolve (6m) and DeepEvolve (14m)—it significantly reduces the *total number of iterations* required to reach peak performance. As illustrated in Table 4, Self-EvolveRec reaches its peak in just around 8 to 11 iterations, whereas baselines relying on aimless trial-and-error require 13 to 19 iterations or fail to outperform the initial model. **2)** While integrating RAG processes in DeepEvolve and Self-EvolveRec increases the runtime per iteration compared to AlphaEvolve, it is critical for ensuring the quality of the evolved logic. As discussed in Sec. 4.1.3, the absence of external knowledge limits AlphaEvolve's ability to discover novel mechanisms, resulting in lower *Creativity* and *Insight*. Conversely, Self-EvolveRec leverages this RAG latency to incorporate external algorithmic knowledge and diagnostic signals, enabling the discovery of novel mechanisms that scalar-driven baselines fail to achieve. **3)** The User Simulator introduces an additional time cost (4m 31s), yet it is crucial for sustaining high performance. Furthermore, as detailed in App. F.1.1, the simulator remains stable and effective even with a small number of sampled users, ensuring efficiency without compromising robustness.

F.2 Removing Feedback-aware Planning & Retrieval

To evaluate the contribution of feedback-aware planning & retrieval introduced in Sec. 3.2, we compare Self-EvolveRec against a variant that excludes Planning & Retrieval shown in Equation 4, denoted as **w.o. Planning** in Table 7. In this variant, the agent generates code directly based on the directional feedback through the code evolution step as $\mathcal{B}^{(t+1)} = \text{LLM}(\mathcal{I}_{\text{CODE}}, \mathcal{R}_{\text{SIM}}, \mathcal{R}_{\text{DIAG}}, \mathcal{B}_{\text{parent}}, \mathcal{P}^{(t)})$. From the results in Table 7, we have the following observations: **1)** w.o. Planning achieves comparable recommendation performance to the original Self-EvolveRec. This implies that the primary driver of performance is the precise identification of failure modes via directional feedback, rather than the incorporation of external research ideas. **2)** However, the absence of planning significantly degrades codebase quality.[5] While Insight remains high (effectively identify what is wrong), w.o. Planning exhibits substantial drops in Creativity and Explicitness, along with a notable decline in Personalization. This suggests that without planning, the agent relies on local heuristic patches rather than systematic, modular designs grounded in external knowledge. **3)** While DeepEvolve achieves high Creativity via external knowledge (RAG), w.o. Planning achieves higher Personalization. This confirms that the directional feedback from SIM, which captures specific user needs, is more crucial for personalization than generic knowledge retrievals.

[5] We use the evaluation criteria from Sec. 4.1.3 to evaluate the evolved models in Table 7.

Table 8: Big Five Personality

Method	CDs		Electronics	
	NDCG@5	HR@5	NDCG@5	HR@5
Self-EvolveRec	0.3865	**0.5274**	**0.2600**	**0.3591**
Big 5 Personality	**0.3915**	0.5244	0.2551	0.3561

Table 9: Evolving User Simulator. Accuracy denotes the accuracy of identifying the target item from 20 candidates.

Dataset	Evolution Strategy	Recommendation Performance (Performance of Evolved NCF) NDCG@5	HR@5	Simulator Reliability (Performance of SIM) SIM Type	Accuracy
CDs	-	-	-	NCF (Baseline)	0.2882
	Fixed Simulator	0.2723	**0.3799**	Initial SIM	0.3473
	Evolved Simulator	**0.2745**	0.3791	Evolved SIM	**0.3610**
Electronics	-	-	-	NCF (Baseline)	0.2191
	Fixed Simulator	**0.1907**	0.2714	Initial SIM	0.2238
	Evolved Simulator	0.1891	**0.2744**	Evolved SIM	**0.2341**

F.3 Evolving User Simulator

Although our framework employs a fixed user simulator for efficiency, we further investigate the impact of evolving the user simulator alongside the codebase. Analogous to Sec. 3.3, the co-evolution of the user simulator follows the same structured workflow of analysis, retrieval, and planning. Specifically, the LLM analyzes the current simulator and synthesizes a development plan $\mathcal{R}_{\text{Dev-SIM}}$ using retrieved methodologies $\mathcal{K}_{\text{SIM}}$, yielding the evolved simulator: $\text{SIM}^{(t+1)} = \text{LLM}(\mathcal{I}_{\text{Code-SIM}}, \mathcal{R}_{\text{Dev-SIM}}, \text{SIM}_{\text{parent}}, \mathcal{P}^{(t)})$. In Table 9, we evaluate simulator reliability by measuring the accuracy of identifying the target item from 20 candidates, utilizing a simple recommender model, NCF, as a baseline. While the recommendation performance remains comparable, the SIM's accuracy reveals the following observation: **1)** The comparable recommendation performance indicates that the initial SIM is already effective for modeling complex user preferences, achieving an accuracy superior to the NCF (see Simulator Reliability). This result indicates that the SIM provides reliable, high-quality directional feedback that captures complex user intent in decision making processes on given recommendation list. Consequently, even without evolving the simulator, the feedback signals from $\mathcal{R}_{\text{SIM}}$ are already robust enough to guide the Self-EvolveRec toward an optimal codebase. **2)** Nevertheless, evolution on SIM further elevates the simulator's reliability. The evolved SIM achieves an even higher accuracy compared to its initial state. This confirms that while the resulting recommendation score are similar on evolved codebase, the evolutionary process constructs a statistically more trustworthy feedback, ensuring that the directional feedback is grounded in realistic user behavior patterns.

F.4 Other User Simulator

To demonstrate the simulator agnostic toward specific user traits, we replaced the traits with the Big Five personality traits (Openness, Conscientiousness, Extraversion, Agreebleness, and Neuroticism), following prior works Ma et al. (2025). As shown in Table 8, Self-EvolveRec achieves comparable recommendation performance regardless of the trait definition. This results aligns with our observations in App. F.3, indicating that the SIM is already effective at modeling complex user preferences and generating informative feedback irrespective of the specific traits schema. Consequently, this results shows that the key factor in evolution is the precise and qualitative nature of $\mathcal{R}_{\text{SIM}}$ itself, which effectively identifies recommendation failures and provides valid guidance for codebase improvements, proving that Self-EvolveRec is robust to variations in user characterization.

F.5 Additional Case Studies

F.5.1 Addtional Case Study: Reliability of Co-evolved Diagnostic Tool via Deficiencies Injection.

In case Figure 6, we injected deficiencies into an evolved MoRec pipeline by: (i) inverting content signals, and (ii) inflating popularity scores. The co-evolved DIAG effectively detects these deficiencies through newly formulated metrics such as the **Off-Category Rate** (Mismatch rate between the categories of the Top-K recommended items and the categories present in user's recent interaction history) and **Popularity correlation** (Correlation between the model's predicted recommendation logits and the item popularity distribution). The $\mathcal{R}_{DIAG}$ accurately reports that the system has begun to "surface off-category or popular items rather than reliably relevant ones" providing clear directional feedback for subsequent correction. Notably, the strong association between these detected deficiencies and the performance

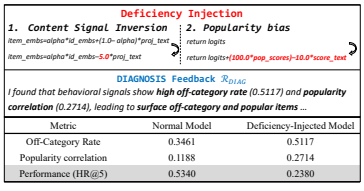

Figure 6: Case study on Diagnosis Tool - Model Co-Evolution on CDs dataset (Seed Recommender: MoRec).

drop confirms that the co-evolved metrics can effectively bridge the gap between internal behavioral shifts and external recommendation accuracy.

F.5.2 Additional Case Study: Evolutionary Trajectory.

To validate the effectiveness of the directional feedback loop, we conducted a case study comparing the evolutionary trajectories of Self-EvolveRec against baselines. Figure 7 (a) illustrates the step-by-step evolution of codebases, while Figure 7 (b) tracks the performance progress over all iterations. We have the following observations: **1)** Self-EvolveRec shows a structured evolutionary path, where algorithmic improvements are causally linked to identified failures. For instance, in Iteration ($0 \rightarrow 1$), $\mathcal{R}_{\text{SIM}}$ explicitly flagged "Ignores Subgenre", while $\mathcal{R}_{\text{DIAG}}$ detected "High Embedding Collapse." Guided by this directional feedback, the agent introduced "Category-Aware Hard Negatives" and "Popularity-annealed Weighting," resulting in an immediate performance increment (HR: $0.4676 \rightarrow 0.5138$). **2)** In contrast, baselines exhibit unstable or delayed progress due to their reliance on scalar metrics. AlphaEvolve attempts an erroneous combination of loss functions (BCE + BPR) at Iteration 3, causing a significant performance drop, which is only rectified by removing the module at Iteration 7. Consequently, it does not exceed its initial performance state throughout the evolution. DeepEvolve suffers from prolonged stagnation in low-performance regions, as evident in Figure 7 (b), and only manages a gain at Iteration 15 by retrieving the 'Refined-LogQ' module. Due to such inefficient exploration, as observed in Figure 7 (b), both baselines remain trapped in low-performance regions ($HR@5 : 0.05 - 0.25$), failing to escape suboptimal states throughout the evolution process. Conversely, Self-EvolveRec leverages directional feedback to maintain a robust evolutionary trajectory.

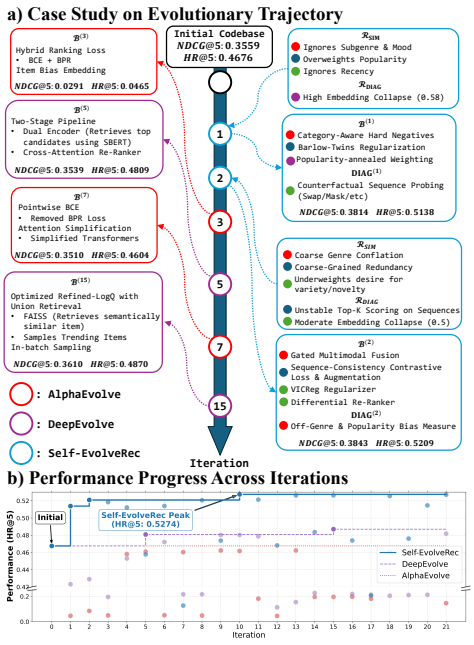

Figure 7: Case study on evolutionary trajectory on CDs dataset (Seed Recommender: SASRec). (a) is comparison of evolutionary paths. Color-coded markers (e.g., Red) illustrate causal alignment between directional feedback and evolved codebase. (b) is performance comparison across iterations.

We also investigate the Self-EvolveRec's behavior on extreme initial recommender setting, we examined Self-EvolveRec's behavior starting from a Random Recommender (Sec. 4.3.1). Figure 8(a) illustrates the step-by-step code evolution, while Figure 8(b) tracks the performance progress. We have the following observations: **1)** Self-EvolveRec demonstrates a structured evolutionary path where algorithmic improvements are causally linked to identified failures. For instance, in the transition from Iteration $0 \rightarrow 1$, $\mathcal{R}_{\text{DIAG}}$ explicitly flagged the "Random Ranking" behavior, while $\mathcal{R}_{\text{SIM}}$ highlighted the neglect of item categories ("Ignores genre"). Guided by this directional feedback, Self-EvolveRec introduced an "LM-Augmented Bi-Encoder" to embed "Genre Tags" and utilized the InfoNCE loss with an MF module. This effectively resolved the random recommendation issue, yielding a significant performance leap (HR: $0.0525 \rightarrow 0.3737$). Similarly, at Iteration 5, Self-EvolveRec detected "Recency Insensitivity" and addressed it by integrating "Time-RoPE Sequence Encoding," further boosting performance to HR: 0.5070. **2)** Consistent with the findings in App. F.5.2, baselines exhibit unstable or delayed progress due to their reliance on scalar metrics without diagnostic guidance. At Iteration 4, AlphaEvolve attempted to add "Popularity Bias and Global Bias" to the model, but this update degraded performance (HR:2405 $\rightarrow$ 0.1969), as depicted in Figure 8 (b). Consequently, AlphaEvolve removes these changes and add "User Bias" at iteration 8, illustraing the inefficient tiral-and-error process. DeepEvolve shows a successful evolution with a "Two-Stage Pipeline" at iteration 6, but failed to improve at iteration 9 due to an incompatible curriculum learning strategy for LM negatives. Also in Figure 8 (b) confirms that while baselines suffer from performance fluctuations, Self-EvolveRec maintains a robust evolutionary trajectory enabled by directional feedback.

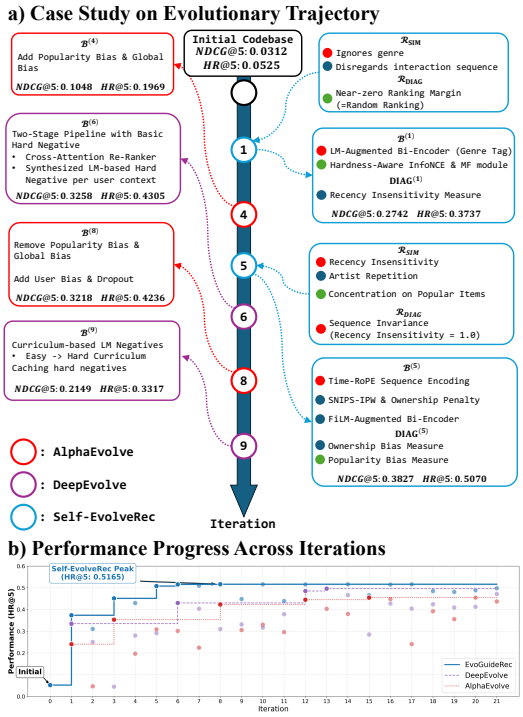

Figure 8: Case study on evolutionary trajectory on CDs dataset (Seed Recommender: Random). (a) is comparison of evolutionary paths. Color-coded markers (e.g., Red) illustrate causal alignment between directional feedback and evolved codebase. (b) is performance comparison across iterations.

G PROMPTS

G.1 TASK-SPECIFIC INSTRUCTION PROMPTS

The instruction prompts for the code evolution process were formulated by drawing upon existing methodologies Novikov et al. (2025); Liu et al. (2025b), ensuring consistency with established benchmarks.

G.2 LLM-AS-A-JUDGE

Figure 20 illustrates the LLM-as-a-Judge prompt utilized in Sec. 4.1.3 to evaluate the quality of the generated code.

System Prompt

You excel at role-playing. Picture yourself as a single user exploring a recommendation page like Amazon. Your goal is to provide realistic behavior and explain the causal reasons behind your choice to help improve the recommendation model. The goal of this task is NOT UI feedback. It is to help improve the recommendation model itself by explaining WHY certain items felt right or wrong.

User Prompt

Your fixed traits
{user's traits}
Your Recent context (Recent Interacted Items)
- {user's interaction history}
Current recommendation page
{Recommended Items}

PART 1 — BEHAVIOR (Act as a real user)

For EACH item on the recommendation page, decide:
1) Whether it aligns with your taste RIGHT NOW (yes or no).
2) If it aligns and you would choose it this time, give it a rating from 1 to 5 based on how much you think you would like it after trying.
3) Briefly explain why.

Use EXACTLY this format for each item (one line per item):
ItemID: <item_id>; Title: <title>; ALIGN: <yes|no>; RATING: <1-5 or 'NA' if not chosen>; REASON: <short reason>

PART 2 — MODEL_FEEDBACK_JSON (Causal Analysis)

Analyze the recommendation list from the perspective of decision reasons. Focus on:
- Why the items you liked felt like strong matches.
- Why the items you disliked felt off, even if they looked similar on the surface.
- What signals seem to be overused (e.g., too much focus on popularity) or underused (e.g., missing specific category nuances) by the system.
- Where the system generalized correctly vs incorrectly from your recent behavior.

IMPORTANT RULES:
- Do NOT talk about UI, layout, or presentation.
- Do NOT suggest business rules or exposure tricks.
- Do NOT mention model internals or technical terms.
- Speak as a user, but explain your choices clearly and causally.

Output ONLY the following JSON structure immediately after Part 1:

MODEL_FEEDBACK_JSON:
```
{{
 "positive_alignment_reasons": [
   "Concrete reason why several recommended items felt like natural next choices",
   "Another clear reason why the system understood part of your intent"
 ],
 "negative_misalignment_reasons": [
   "Concrete reason why some items missed your intent despite appearing related",
   "Another reason explaining why certain recommendations felt off"
 ],
 …
 "missing_or_underweighted_signals": [
   "A type of information or nuance (e.g., specific sub-category) the system missed",
   "Another missing nuance"
 ],
 "summary_diagnosis": "One short sentence summarizing how the system interpreted your intent, correctly or incorrectly."
}}
```

Figure 9: Example prompt of $\mathcal{I}_{\text{SIM}}$.

System Prompt

You are a Lead Recommender Systems Analyst synthesizing user simulation feedback
to diagnose how the recommendation model is behaving.

User Prompt

Input:
- A list of User Feedback Reports (Json objects) from multiple simulated users:

Each USER_DIAGNOSIS_JSON reflects:
- what users expected based on recent behavior,
- why certain recommendations felt right or wrong,
- which patterns consistently helped or hurt their willingness to choose items near the top.

Goal:
Produce a concise, system-level diagnosis of:
- what decision patterns the model appears to rely on,
- where those patterns align or misalign with actual user choice logic,
- which misalignments most directly reduce the quality of top-ranked items.

Do NOT discuss UI, layout, exposure strategy, or business rules.
Do NOT propose specific model architectures or losses.
Stay at the level of behavioral decision logic inferred from user feedback.

OUTPUT FORMAT (JSON ONLY — all keys required)

{{"status": "<CRITICAL | NEEDS_IMPROVEMENT | STABLE>",

"SYSTEM_INTERPRETATION_OF_USER_INTENT":
"One short sentence describing how the system seems to interpret user intent overall, based on aggregated feedback.",

…

"DIAGNOSTIC_SIGNALS_TO_TRACK": [
"A measurable signal that would reveal whether the system is ranking genuinely choice-worthy items near the top.",
"Another signal that would expose overgeneralization or missed intent."]}}

Rules:
- No numeric scores or internal error codes.
- No UI or exposure language.
- No raw user quotes; always generalize and compress.
- Every statement should describe a repeatable system behavior or decision pattern.
- Focus on issues that, if corrected, would materially improve how many top-ranked items users would realistically choose.

Figure 10: Example prompt of $\mathcal{I}_{\text{SUMMARIZE}}$.

System Prompt

You are a Senior AI Researcher analyzing the mathematical health of a Recommendation System.

User Prompt

Here are the raw mathematical metrics measured from the current model:
{D_{raw}}

Here are the definitions of what these metrics mean:
{Definition of metrics in D_{raw}}

Your task: produce a concise diagnosis summary that captures ONLY the core findings and implications.
Do NOT propose web searches or action plans. Do NOT list "what to look up".

Strictly follow the JSON format below.

Output Format (JSON):

```
{{
"status": <CRITICAL | NEEDS_IMPROVEMENT | STABLE>,
"core_findings": [
   "<1-2 sentences: the most important interpretation of the metrics>",
   "<1-2 sentences: the second most important interpretation (only if truly necessary)>"
 ],
"key_implications": [
   "<1 sentence: what this implies about model behavior>",
   "<1 sentence: what this implies about training dynamics or representation>",
   "<optional 1 sentence: risk/trade-off if relevant>"
],
"evidence": {{
   "headline_metrics": {{
   "<metric_name>": <value>,
   "<metric_name>": <value>
   }},
   "brief_metric_read": {{
   "<metric_name>": "<very short interpretation tied to its definition>",
   "<metric_name>": "<very short interpretation tied to its definition>"
   }}
}}
}}
```

Rules:
- Keep it short: core_findings <= 3 bullets, key_implications <= 3 bullets.
- Use technical terms only when they are essential (e.g., 'gradient', 'loss landscape', 'inductive bias'), and keep them brief.
- Every statement must be grounded in the provided metric values/definitions.
- If metrics look healthy, status should remain STABLE and implications should focus on "what is already working" plus one potential risk to monitor.

Figure 11: Example prompt of $\mathcal{I}_{\text{DIAG}}$.

System Prompt
You are a professor responsible for planning deep and effective research strategies.
User Prompt
You will be provided with the context of: - a research problem based on an initial research question - a starting research idea, possibly with a history showing how idea evolves through previous attempt - inspirations from earlier attempts - qualitative user diagnosis from user - quantitative mathematical diagnosis Your task is to develop search queries that identify directions for researchers to advance the idea in a transformative way. Rather than combining existing inspirations in small increments, the queries should guide researchers toward substantial evolutions. Because other researchers will rely on this plan, it must emphasize major, novel approaches instead of minor refinements. You will also be told whether the research progress is early or mature: - If the progress is early, focus on ideas that are feasible and practical, and can grow later and have great future potential. - If the progress is mature, focus on bold, high-impact shifts that challenge the current approach. Your plan should follow two steps: 1. Formulate 1 to 3 precise and diverse search queries. Make sure the queries are diverse—cover different perspectives, challenge untested assumptions, and explore alternative methods. 2. For each query, include a short note explaining why you chose it and what you hope it will reveal.

Figure 12: Example prompt of $\mathcal{I}_{\text{PLAN}}$.

System Prompt

You are a senior researcher responsible for proposing new ideas to address a defined research problem.

User Prompt

You will receive:
- The research problem, including its evaluation metric, qualitative user diagnosis feedback, quantitative mathematical diagnosis feedback, and available data
- A starting research idea, possibly with its evolution history
- Inspirations from earlier attempts
- A list of related online search results
- A research progress score (0-100%) indicating how far the idea has advanced

Your goal is to identify future research directions that address the target problem, using the starting point, prior attempts, and related works. You should analyze existing methods, identify connections, and propose practical algorithms that can be implemented with the available data.

Follow this structure to think and write:

1. Extract insights
Identify 3-5 scientific insights from the starting point and 3-5 from related works. For each insight, explain in 2-3 sentences how it relates to the target problem.

2. Organize research directions
Group the insights into 3-5 coherent directions (for example, learning objectives, model classes, or optimization methods).

3. Build a structured framework
Create a conceptual map (such as a taxonomy, grid, or matrix) that unifies existing methods, reveals patterns, and highlights gaps.

4. Generate and evaluate ideas
- For each idea, critically assess as a senior researcher with one positive and one negative reason:
- Originality (0-10): Is the idea new? Is the idea a novel combination of well-known techniques? Is it clearly different from previous contributions?
- Future Potential (0-10): Will others build on these ideas? Does this idea solve a hard problem more effectively than prior work? Does it point to a new research direction?
- User Diagnosis Alignment (0-10): How well the idea addresses the key user-observed failure patterns and improves what users actually want at the top of the list?
- Model Diagnosis Alignment (0-10): How well the idea addresses the key quantitatively observed failure patterns and improves the model behaviors measured by the diagnostics?

…

5. Write the report in Markdown
For the selected idea, include:
- A synthesis of insights and proposed directions
- The structured framework of existing methods and the new algorithm
- A list of new ideas with their assessment scores
- Detailed description of the chosen/best idea, including rationale, pseudocode, and implementation notes

The report must be focused, technically accurate. Being concise with 200-500 words without trivial and redundant information.
Support all claims with evidence, references and remain tightly aligned with the target problem.

Figure 13: Example prompt of $\mathcal{I}_{\text{REPORT}}$.

System Prompt

You are a researcher with strong software engineering skills, improving algorithmic code through iterative, performance-driven modifications in multiple rounds.

User Prompt

You will receive a research question, a proposed idea, and an existing implementation with performance metrics. Your goal is to analyze the current code and apply precise changes that enhance the specified metrics, based on the research idea and prior feedback.

You MUST use the exact SEARCH/REPLACE diff format. Do NOT use Git diff format. Do NOT use line prefixes like `+`, `-`, or `@@`.
Use this structure exactly:
```
<<<<<<< SEARCH
# Original code (must match exactly)
=======
### >>> Self-EvolveRec-BLOCK-START: <research idea>
# New code here
### <<< Self-EvolveRec-BLOCK-END
>>>>>>> REPLACE
```
...

Task Guidelines:
1. Think before coding, understand the research idea and current performance bottlenecks.
2. Propose specific, actionable changes that are aligned with the target metrics.
3. You may suggest multiple improvements beyond the research idea based on your understanding of optimization and machine learning.
4. When you are updating the code, please check the following:
- When a NEW parameter or behavior is added, verify it is invoked in all call sites or in the overall workflow.
- If a NEW parameter has a default value of None, confirm that passing a non-None value triggers the intended code path.
- Walk through or simulate function calls to confirm that each new branch or change will be executed. Avoid unreachable modifications.

Code Format Guidelines:
1. All `SEARCH` blocks must match the original code exactly.
2. When you need to modify code that is not already inside a `Self-EvolveRec` block, wrap your changes with `### >>> Self-EvolveRec-BLOCK-START: <research idea>` and `### <<< Self-EvolveRec-BLOCK-END` markers.
3. If you are updating code that is already marked by a `Self-EvolveRec` block, edit only the lines within that block and adjust the existing modification comment to reflect your new change.
4. Do NOT nest one `Self-EvolveRec` block inside another. Each region you modify should have exactly one pair of start/end markers.
…
5. Limit your changes to what is strictly necessary. Do not rewrite the entire file.
6. Ensure that all modified code remains correct and consistent, including any function signatures, parameter lists, and calls.
7. Preserve the original code's indentation and formatting. Place the lines of `### >>> Self-EvolveRec-BLOCK-START: <research idea>` and `### <<< Self-EvolveRec-BLOCK-END` at the same indentation level as the code they annotate.

Figure 14: Example prompt of $\mathcal{I}_{\text{CODE}}$.

System Prompt
You are a senior ML engineer and code architect.
User Prompt
You will be given the current implementation of: - the recommendation model (e.g., models.py) - the model diagnosis tool module (e.g., diagnosis_tools.py) - any small utilities they depend on. Your job is to extract a clear, high-level WORKFLOW SUMMARY of how MODEL DIAGNOSIS is produced from the model. Focus on: - Model (for diagnosis): - Which parts of the model are relevant for diagnosis. - Where and how the final user / sequence representation is computed. - Diagnosis inputs & hooks: - How the diagnosis module gets access to the model and data. - Which tensors or functions it calls (embeddings, features, logits, histories). - Probes & metrics: - What each metric is intended to measure. - How each metric is computed at a high level, and what “good” vs “bad” qualitatively means. - Perturbation & tests: - If and how inputs or sequences are perturbed to test sensitivity. - What behavior these tests are meant to reveal. - Output schema & limitations: - How MODEL_DIAGNOSIS is structured (keys, metric names, brief meanings). - Key assumptions and blind spots (what the current diagnosis does NOT check). Output a short, structured summary in Markdown with headings such as: # Model Overview (for Diagnosis) # Diagnosis Inputs & Hooks # Probes & Metrics # Perturbation & Tests # MODEL_DIAGNOSIS Schema & Limitations Be concise but precise (around 200-400 words) so another agent can understand how the diagnosis module reads the model and which behaviors it monitors.

Figure 15: Example prompt of $\mathcal{I}_{\text{Analyze}}$.

System Prompt

You are a professor responsible for planning deep and effective research strategies.

User Prompt

You will receive:
- The research problem, based on an initial research question
- A starting research idea, possibly with its evolution history
- Inspirations from earlier attempts
- The user's current qualitative feedbacks
- The analysis report on models and diagnosis tools
- A list of related online search results
- A research progress score (0-100%) indicating how far the idea has advanced

Your task is to develop search queries that guide researchers toward substantial improvements to the diagnosis toolkit, not just minor metric tweaks. Focus on directions such as:
- New probes/metrics that better capture real failure modes
- New probes/metrics that capture user's feedbacks
- Better ways to connect probes to specific model components and training signals
- Methods to validate, stress-test, or calibrate these metrics so they are trustworthyRather than combining existing inspirations in small increments, the queries should guide researchers toward substantial evolutions.
Because other researchers will rely on this plan, it must emphasize major, novel approaches instead of minor refinements.

You will also be told whether the research progress is early or mature:
- If the progress is early, focus on ideas that are feasible and practical, and can grow later and have great future potential.
- If the progress is mature, focus on bold, high-impact shifts that challenge the current approach.

Your plan should follow two steps:
1. Formulate 1 to 3 precise and diverse search queries. Make sure the queries are diverse—cover different perspectives, challenge untested assumptions, and explore alternative methods.
2. For each query, include a short note explaining why you chose it and what you hope it will reveal.

Figure 16: Example prompt of $\mathcal{I}_{\text{PLAN-DIAG}}$.

System Prompt

You are a senior researcher responsible for proposing new ideas to address a defined research problem.

User Prompt

You will receive:
- The research problem, including its qualitative user diagnosis feedback, code analysis of models and diagnosis tool, and available data
- A starting research idea, possibly with its evolution history
- Inspirations from earlier attempts
- A list of related online search results
- A research progress score (0-100%) indicating how far the idea has advanced

Your goal is to identify future research directions that address the target problem, using the starting point, prior attempts, and related works. You should analyze existing methods, identify connections, and propose practical algorithms that can be implemented with the available data.

Follow this structure to think and write:

1. Extract insights
Identify 3-5 scientific insights from the starting point and 3-5 from related works. For each insight, explain in 2-3 sentences how it relates to the target problem.

2. Organize research directions
Group the insights into 3-5 coherent directions (for example, learning objectives, model classes, or optimization methods).

3. Build a structured framework
Create a conceptual map (such as a taxonomy, grid, or matrix) that unifies existing methods, reveals patterns, and highlights gaps.

4. Generate and evaluate ideas
- For each idea, critically assess as a senior researcher with one positive and one negative reason:
- Originality (0-10): Is the idea new? Is the idea a novel combination of well-known techniques? Is it clearly different from previous contributions?
- Future Potential (0-10): Will others build on these ideas? Does this idea solve a hard problem more effectively than prior work? Does it point to a new research direction?

…

5. Write the report in Markdown
For the selected idea, include:
- A synthesis of insights and proposed directions
- The structured framework of existing methods and the new algorithm
- A list of new ideas with their assessment scores
- Detailed description of the chosen/best idea, including rationale, pseudocode, and implementation notes

The report must be focused, technically accurate. Being concise with 200-500 words without trivial and redundant information.
Support all claims with evidence, references and remain tightly aligned with the target problem.

Figure 17: Example prompt of $\mathcal{I}_{\text{REPORT-DIAG}}$.

System Prompt

You are a researcher with strong software engineering skills, improving algorithmic code through iterative, performance-driven modifications in multiple rounds.

User Prompt

You will receive a research question, a proposed idea, and an existing implementation of a recommendation model and diagnostic tool. Your goal is to analyze the current code and apply precise changes that enhance the specified metrics, based on the research idea and prior feedback.

You MUST use the exact SEARCH/REPLACE diff format. Do NOT use Git diff format. Do NOT use line prefixes like `+`, `-`, or `@@`.
Use this structure exactly:

```
<<<<<<< SEARCH
# Original code (must match exactly)
=======
### >>> Self-EvolveRec-BLOCK-START: <research idea>
# New code here
### <<< Self-EvolveRec-BLOCK-END
>>>>>>> REPLACE
```

...

Task Guidelines:
1. Think before coding, understand the research idea and current performance bottlenecks.
2. Propose specific, actionable changes that are aligned with the target metrics.
3. You may suggest multiple improvements beyond the research idea based on your understanding of optimization and machine learning.
4. When you are updating the code, please check the following:
- When a NEW parameter or behavior is added, verify it is invoked in all call sites or in the overall workflow.
- If a NEW parameter has a default value of None, confirm that passing a non-None value triggers the intended code path.
- Walk through or simulate function calls to confirm that each new branch or change will be executed. Avoid unreachable modifications.

Code Format Guidelines:
1. All `SEARCH` blocks must match the original code exactly.
2. When you need to modify code that is not already inside a `Self-EvolveRec` block, wrap your changes with `### >>> Self-EvolveRec-BLOCK-START: <research idea>` and `### <<< Self-EvolveRec-BLOCK-END` markers.
3. If you are updating code that is already marked by a `Self-EvolveRec` block, edit only the lines within that block and adjust the existing modification comment to reflect your new change.
4. Do NOT nest one `Self-EvolveRec` block inside another. Each region you modify should have exactly one pair of start/end markers.
…
5. Limit your changes to what is strictly necessary. Do not rewrite the entire file.
6. Ensure that all modified code remains correct and consistent, including any function signatures, parameter lists, and calls.
7. Preserve the original code's indentation and formatting. Place the lines of `### >>> Self-EvolveRec-BLOCK-START: <research idea>` and `### <<< Self-EvolveRec-BLOCK-END` at the same indentation level as the code they annotate.

Figure 18: Example prompt of $\mathcal{I}_{\text{CODE-DIAG}}$.

System Prompt

You are a researcher with strong software engineering skills, improving algorithmic code through iterative, performance-driven modifications in multiple rounds.

User Prompt

You will receive a research question, a proposed idea, and an existing implementation of a user simulator. Your goal is to analyze the current code and apply precise changes that enhance the specified metrics, based on the research idea and prior feedback.

You MUST use the exact SEARCH/REPLACE diff format. Do NOT use Git diff format. Do NOT use line prefixes like `+`, `-`, or `@@`.
Use this structure exactly:

```
<<<<<<< SEARCH
# Original code (must match exactly)
=======
### >>> Self-EvolveRec-BLOCK-START: <research idea>
# New code here
### <<< Self-EvolveRec-BLOCK-END
>>>>>>> REPLACE
```

...

Task Guidelines:
1. Think before coding, understand the research idea and current performance bottlenecks.
2. Propose specific, actionable changes that are aligned with the target metrics.
3. You may suggest multiple improvements beyond the research idea based on your understanding of optimization and machine learning.
4. When you are updating the code, please check the following:
- When a NEW parameter or behavior is added, verify it is invoked in all call sites or in the overall workflow.
- If a NEW parameter has a default value of None, confirm that passing a non-None value triggers the intended code path.
- Walk through or simulate function calls to confirm that each new branch or change will be executed. Avoid unreachable modifications.

Code Format Guidelines:
1. All `SEARCH` blocks must match the original code exactly.
2. When you need to modify code that is not already inside a `Self-EvolveRec` block, wrap your changes with `### >>> Self-EvolveRec-BLOCK-START: <research idea>` and `### <<< Self-EvolveRec-BLOCK-END` markers.
3. If you are updating code that is already marked by a `Self-EvolveRec` block, edit only the lines within that block and adjust the existing modification comment to reflect your new change.
4. Do NOT nest one `Self-EvolveRec` block inside another. Each region you modify should have exactly one pair of start/end markers.
…
5. Limit your changes to what is strictly necessary. Do not rewrite the entire file.
6. Ensure that all modified code remains correct and consistent, including any function signatures, parameter lists, and calls.
7. Preserve the original code's indentation and formatting. Place the lines of `### >>> Self-EvolveRec-BLOCK-START: <research idea>` and `### <<< Self-EvolveRec-BLOCK-END` at the same indentation level as the code they annotate.

Figure 19: Example prompt of $\mathcal{I}_{\text{CODE-SIM}}$.

System Prompt

You are a Senior Fellow at a top AI Research Lab, reviewing the code evolution of a Recommender System.
You are comparing one evolved codebases (Model A) against a Seed Code.

You must evaluate which model represents a technically superior and more meaningful evolution based solely on static code analysis.
Do NOT assume any specific downstream task performance. Judge based on the quality of the code changes.

Evaluation Criteria (Score 1-10):

1. Algorithmic Novelty
- Definition: Does the code introduce genuinely new mechanisms rather than just tuning existing ones?
- Checkpoints:
Low (1-3): Simple hyperparameter tuning, variable renaming, or mere code refactoring without functional changes.
High (8-10): Designing completely new loss functions, implementing dynamic sampling strategies, or creating novel attention/interaction mechanisms.

2. Logic Explicitness (Interpretability)
- Definition: By reading the `forward` function, how easy is it to understand the decision mechanism?
- Checkpoints:
Low (1-3): "Black Box" logic. Data flows through deep, opaque layers making it hard to trace why a specific item was recommended.
High (8-10): "Transparent" logic. The code includes explicit, readable control flows that clearly show how the output distribution is shaped by specific factors.

3. Problem-Solving Insight
- Definition: Does the modification reflect a clear, logical intention to solve a specific recommendation problem (e.g., sparsity, bias, diversity)?
- Checkpoints:
Low (1-3): Random or generic changes. Adding standard layers or changing parameters without a clear purpose or connection to a specific problem.
High (8-10): Targeted solutions. The logic directly addresses a specific deficiency with clear intent.

4. User-Centric Adaptation (Personalization Depth)
- Definition: Does the code treat the user merely as a static index (ID), or does it actively adapt to the user's specific context and constraints (e.g., recent history, price sensitivity, category affinity)?
- Checkpoints:
Low (1-3): Static Personalization. The model relies solely on `User_Embedding (dot product) Item_Embedding`. It treats personalization as a black-box latent matching problem without considering specific user attributes or immediate context.
High (8-10): Dynamic Context-Awareness. The code explicitly incorporates user-specific signals into the scoring logic. …

User Prompt

Here are the code snippets. Scoring Model A based on their changes from the Seed Code.
=== [SEED CODE (Original)] ===
{seed_code}
=== [MODEL A (Evolved)] ===
{code_a}
Return ONLY the JSON.

Figure 20: Example prompt of LLM-as-a-Judge for evolved models evaluation.