FIRM: Fine-grained Item Relational Memory for LLM-based Recommender Agents

Anonymous authors
Paper under double-blind review

Abstract

LLM-based recommender agents store user preferences and item characteristics in natural-language memory, and reason over item text and interaction history to rerank candidate items. Recent work improves this reranking with two predefined item relation types, *substitute* and *complement*, which reveal user intent implicit in interaction data. However, these types cannot capture the specific comparison criterion: *regular vs. decaffeinated beans* and *small vs. bulk packs* are both substitutes, yet differ in caffeine content and package size, respectively. Moreover, the relations are re-inferred for every user rather than stored as knowledge shared across users. Simply generating fine-grained relations produces redundant types with inconsistent definitions. In this paper, we propose **F**ine-grained **I**tem **R**elational **M**emory (**FIRM**), a shared memory that organizes fine-grained item relations into a consistent type hierarchy. FIRM is built by three agents: the Explorer extracts a fine-grained relation for each item pair; the Manager organizes the types into a relation tree by merging redundant ones and splitting those that mix distinct criteria; and the Verifier checks type definitions for consistency. During inference, the relations among the user's history items are summarized into a user memory of behavioral tendencies. The recommender agent then retrieves the relations between history items and candidates from FIRM and reranks the candidates based on how well these relations match the user's tendencies. On three real-world datasets, FIRM improves NDCG@10 over the strongest baseline by up to 10.03%, and a memory built once remains effective across different LLMs.

1 Introduction

Large language models (LLMs) improve recommendations by providing semantic knowledge about items (Lin et al., 2025; Xi et al., 2024; Liao et al., 2024). A common approach is LLM-based reranking (Hou et al., 2024; Yue et al., 2023): a conventional recommender model retrieves candidates from the item catalog, and an LLM then reranks them by interpreting the user's history and the candidates. However, LLMs lack knowledge that emerges only from actual user choices, so a prompt alone often fails to convey latent user preferences and consumption contexts.

To fill this gap, prior work supplies LLMs with knowledge extracted from user interactions, beyond what item titles and descriptions provide. Some studies do so without an explicit memory: STAR (Lee et al., 2024) includes the number of users who consumed each pair of items, and ItemRAG (Kim et al., 2026) augments each item's description with summaries of related items retrieved via co-consumption. *LLM-based recommender agents* go further by maintaining a per-item memory in natural language. AgentCF (Zhang et al., 2024) and MemRec (Chen et al., 2026) record each item's characteristics and the preferences of users who favor it, and update the memory from observed interactions or by propagating it to neighboring items. Through these updates, item memory accumulates implicit characteristics revealed by users' actual choices.

Nevertheless, such item memories are kept separately for each item. Propagation across related items and co-consumption patterns reveal that two items are associated, but not how they are related. AgentDR (Yang et al., 2026) makes this relation explicit: for each user, it generates lists of *substitutes* and *complements* and adjusts the ranking based on how similar the candidates are to these lists. However, this distinction cannot capture the specific criterion by which a pair is compared. Moreover, because the relations are inferred separately for each user, they are not shared across users.

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

We aim to construct a fine-grained relational memory that serves as a shared knowledge source across users. Building such a memory raises two challenges as follows:
(C1) Fine-grained item relations. Existing relation types in AgentDR (Yang et al., 2026) are too coarse to characterize how an item pair should be compared or used together. For example, *regular vs. decaffeinated beans* and *small vs. bulk packs* are both substitutes, yet are compared by caffeine content and package size, respectively. Complements likewise play different roles: *syrup* adds flavor to coffee, whereas a *dripper* enables brewing. Because such criteria and roles vary across item pairs and domains, they cannot be enumerated in advance. Therefore, the memory should add and refine relation types incrementally from the information about actual item pairs.
(C2) Relation canonicalization. Inferring relations independently for each pair can produce different names or definitions for the same relation. For example, the same criterion may appear as *caffeine content* for one pair and *caffeine level* for another. Conversely, distinct criteria may be conflated into one type: *size* may mix package quantity with product dimensions. The memory should canonicalize relation types so that equivalent relations are represented consistently, while distinct relations are kept separate.

To address these challenges, we introduce **F**ine-grained **I**tem **R**elational **M**emory (**FIRM**), constructed by three specialized agents. The **Explorer** selects item pairs for relation inference using item text and collaborative signals, and proposes a fine-grained relation for each selected pair that captures its specific comparison criterion or usage role (C1). The **Manager** organizes the proposed relations into a hierarchical relation tree by adding, merging, and splitting relation types (C1, C2). The **Verifier** checks whether each pair satisfies the definition of its assigned type and revises the assignment or the definition when it does not (C2).

After FIRM is built, our recommender agent uses it in two steps during inference. First, **User Memory Construction** summarizes each user's history into a user memory that captures both the characteristics of preferred items and relation-based preferences, using item descriptions and the relations among history items retrieved from FIRM. Then, **Relation-aware Reranking** retrieves the relations between history items and candidates from FIRM, and evaluates each candidate by how well these relations align with the preferences in the user memory. In this way, fine-grained relations are established once and shared across users, while their relevance to each user is judged from that user's own history.

Our main contributions are summarized as follows.

- We introduce FIRM, a relational memory shared across users. Three agents derive fine-grained item relations, organize them into a consistent type hierarchy, and verify pair assignments and type definitions.
- We develop a relation-aware reranking method that summarizes each user's history into a user memory of preferred item characteristics and relation-based preferences, and evaluates candidates by how well their relations to history items align with these preferences.
- We show that FIRM achieves the highest NDCG on three real-world datasets, improving NDCG@10 over the strongest baseline by up to 10.03% on Grocery, and that a memory built once remains effective across different LLMs.

2 Preliminaries

Let $\mathcal{U}$ and $\mathcal{I}$ denote the sets of users and items, respectively, and let x_i denote the textual description of item $i \in \mathcal{I}$. For user $u \in \mathcal{U}$, the interaction sequence is defined in chronological order as $S_u = [i_1^u, i_2^u, \ldots, i_L^u]$, where $i_l^u \in \mathcal{I}$ denotes the item in the l-th interaction and i_L^u is the most recent one. Sequential recommendation aims to predict the next item i_{L+1}^u given S_u. We follow the two-stage setup widely used in recent work (Hou et al., 2024; Yue et al., 2023; Yang et al., 2026). In the first stage, a conventional sequential recommender, *e.g.*, SASRec (Kang & McAuley, 2018), retrieves a candidate set $\mathcal{C}_u \subseteq \mathcal{I}$. In the second stage, an LLM ranker receives the history items and candidates in a fixed prompt format and outputs a ranking:

$$\pi_u = \mathrm{LLM}_{\mathrm{rank}}(S_u, \mathcal{C}_u), \tag{1}$$

where π_u is a permutation of $\mathcal{C}_u$. The goal of reranking is to place i_{L+1}^u as high as possible in π_u. In this paper, we focus on improving the ranking quality in the second stage.

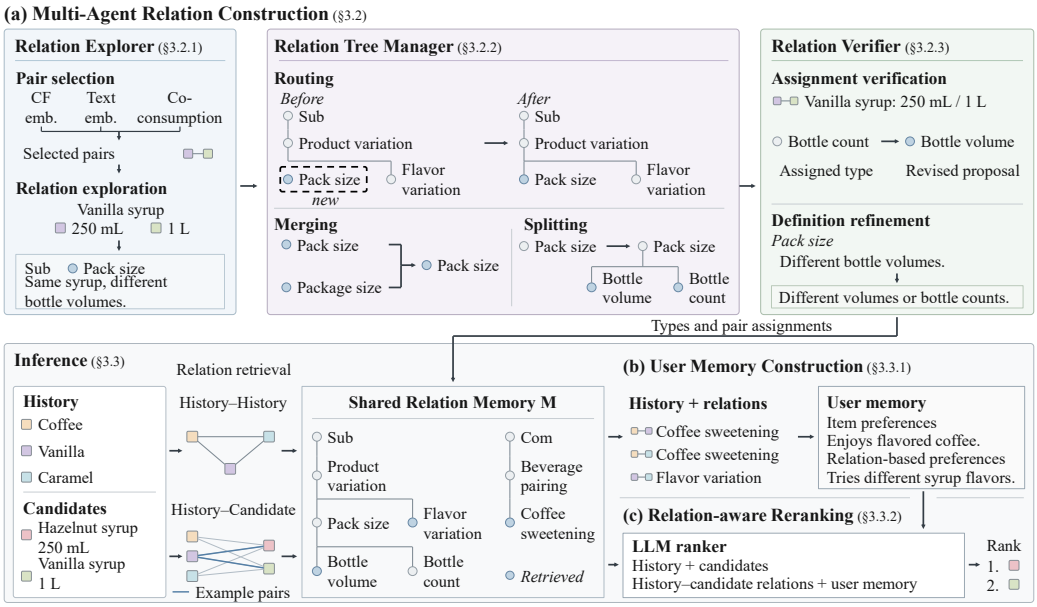


Figure 1: Overview of FIRM. (a) **Multi-Agent Relation Construction**: for selected item pairs, Explorer proposes fine-grained relations, Manager organizes them into a hierarchical relation tree through merging and splitting, and Verifier examines pair assignments and type definitions. (b) **User Memory Construction**: a user memory is built from item information and the relations among history items. (c) **Relation-aware Reranking**: relations between history items and candidates are retrieved from the relational memory to rerank the candidates.

3 Method: Fine-grained Item Relational Memory

3.1 Overview

In this section, we present **F**ine-grained **I**tem **R**elational **M**emory (**FIRM**), which organizes fine-grained relations between item pairs into consistent relation types shared across users. As illustrated in Figure 1, the overall process comprises three stages. Appendix A details the inputs of each stage and summarizes the notation in Table 2, and Appendix C lists the prompts.

- **Multi-Agent Relation Construction** (§3.2): Given item pairs selected using textual information and collaborative signals, three agents identify fine-grained relations, organize them into shared relation types, verify pair assignments, and refine type definitions.
- **User Memory Construction** (§3.3.1): We generate a user memory by interpreting item descriptions and the relations among items in the user's history, capturing both item characteristics and relation-based contexts.
- **Relation-aware Reranking** (§3.3.2): The recommender agent retrieves relations between history items and candidates from the relational memory without inferring new relations, and evaluates them in light of the user memory to rerank the candidates.

3.2 Multi-Agent Relation Construction

This stage constructs the relational memory $\mathcal{M} = (\mathcal{T}, \mathcal{A})$. The relation tree $\mathcal{T}$ is shared across users and organizes relation types hierarchically. Each relation node $r \in \mathcal{T}$ has a name and a definition specifying when the relation applies. We denote an unordered item pair by $p = (i_j, i_k)$. The assignment function $\mathcal{A} : \mathcal{I} \times \mathcal{I} \to \mathcal{T} \cup \{\emptyset\}$ maps each pair to a single relation node, denoted by $\mathcal{A}(p)$; $\mathcal{A}(p) = \emptyset$ if pair p is unassigned.

3.2.1 Relation Explorer

The Relation Explorer proposes a fine-grained relation for each item pair (C1).

Pair Selection. Inferring relations for all item pairs is costly, and most pairs have no meaningful relation. We therefore first select pairs for exploration. We use collaborative embeddings learned from interactions in the training period and text embeddings of item information. In each embedding space, we pair each item with the top-ρ fraction of items ranked by cosine similarity, forming $\mathcal{P}_{\mathrm{CF}}$ and $\mathcal{P}_{\mathrm{T}}$, respectively. We also include $\mathcal{P}_{\mathrm{co}}$, the set of pairs whose items were both consumed by at least m users during the training period. The Explorer processes the union $\mathcal{P} = \mathcal{P}_{\mathrm{CF}} \cup \mathcal{P}_{\mathrm{T}} \cup \mathcal{P}_{\mathrm{co}}$.

Relation Exploration. We prompt the LLM to describe the fine-grained relation of each pair in natural language within two categories, substitution (`Sub`) and complementarity (`Com`), without specifying fine-grained criteria in advance. Substitution refers to a relation in which two items can replace one another, whereas complementarity refers to a relation in which using the items together increases their utility. Specifically, for each selected pair $p \in \mathcal{P}$, the Explorer generates the following triplet:

$$(b_p, r_p, e_p) = \mathrm{LLM}_{\mathrm{explore}}(x_{i_j}, x_{i_k}), \tag{2}$$

where $b_p \in \{\texttt{Sub}, \texttt{Com}, \texttt{None}\}$ denotes the top-level relation category, r_p is a short name for the pair's fine-grained relation (e.g., "regular–decaf substitution"), and e_p is a one-sentence explanation of what distinguishes the pair from other pairs in the same category (e.g., "both are ground coffee of the same roast, but one is a decaffeinated alternative to the other"). Pairs with $b_p = \texttt{None}$ are excluded from subsequent stages.

The Explorer processes each pair independently so that each proposal is grounded in the pair itself. The Manager, described next, resolves the resulting inconsistencies across proposals and uses r_p and e_p as the name and initial definition of each relation node it creates.

3.2.2 Relation Tree Manager

The Relation Tree Manager performs relation canonicalization by organizing the Explorer's independent proposals into relation types shared across pairs (C1, C2). After the Explorer has proposed relations for all selected pairs, the Manager and Verifier process the proposals in batches. To process semantically similar pairs together, we cluster the selected pairs using k-means on text embeddings, order them by cluster, and divide them into batches of B pairs, with $\mathcal{P}_t \subseteq \mathcal{P}$ denoting the t-th batch (Appendix A.4). Each batch starts from the relational memory $\mathcal{M} = (\mathcal{T}, \mathcal{A})$ produced by the previous batch. The Manager organizes the proposals through Routing, Merging, and Splitting, in that order, after which the Verifier checks pair assignments and updates type names and definitions; after the last batch, the same operations run over the whole memory for a fixed number of consolidation rounds (Algorithm 1, Appendix A.2).

Relation Tree. The relation tree $\mathcal{T}$ has two root nodes $b \in \{\texttt{Sub}, \texttt{Com}\}$ corresponding to the two relation categories, with relation types organized hierarchically beneath each root. We denote the set of pairs assigned directly to node r by $\mathcal{P}(r)$, its children by $\mathrm{ch}(r)$, its parent by $\mathrm{pa}(r)$, and the nodes on the path from the root to r by $\mathrm{path}(r)$. The children $\mathrm{ch}(b)$ of each root are higher-level types that group more specific relation types. The Explorer's proposed relations are registered as relation types immediately below these higher-level types, and deeper levels are created only through Splitting. Whenever a node is provided as input to an LLM, its name and definition are supplied. The Manager organizes relation types through the following three operations.

Relation Routing. Routing registers the Explorer's proposed relations as nodes and attaches new nodes to the existing tree: a pair whose proposed relation has the same or an equivalent name as an existing node is assigned directly to that node, and otherwise we create a new node with name r_p and initial definition e_p and assign pair p to it. To determine the parents of the new nodes, we partition the corresponding pairs by relation category $b \in \{\texttt{Sub}, \texttt{Com}\}$ into $\mathcal{P}_t^b$. For each set, the LLM either selects parent nodes from the existing children $\mathrm{ch}(b)$ of root b or proposes new parent nodes that group semantically similar nodes:

$$\{\mathrm{pa}(\mathcal{A}(p))\}_{p \in \mathcal{P}_t^b} = \mathrm{LLM}_{\mathrm{route}}\big(\{(r_p, e_p)\}_{p \in \mathcal{P}_t^b},\ \mathrm{ch}(b)\big) \tag{3}$$

where $\mathrm{pa}(\mathcal{A}(p))$ is either an existing node in $\mathrm{ch}(b)$ or a new node whose name and definition are proposed by the LLM. Each newly proposed parent is added as a child of root b. Nodes representing the same relation are consolidated in the subsequent Merging operation. If no suitable parent can be determined, the node is deferred to Routing in the next batch.

Relation Merging. Merging consolidates multiple relation types that represent the same relation into a single type. Since Routing places the Explorer's proposals under semantically related parents, sibling nodes may still represent the same relation under different names or definitions. We therefore compare their names and definitions to merge semantically equivalent types, and also submit near-identical types under different parents as candidates. Starting from root b, we traverse the tree from top to bottom, visiting nodes r whose children were added or modified in the current batch. The LLM receives the children $\mathrm{ch}(r)$ as merge candidates and the siblings $\mathrm{ch}(\mathrm{pa}(r))$ of r as context, and is instructed not to merge children whose distinctions are comparable in granularity to those among the siblings. This context is omitted when r is a root node.

$$\mathrm{ch}^*(r) = \mathrm{LLM}_{\mathrm{merge}}\big(\mathrm{ch}(r),\ \mathrm{ch}(\mathrm{pa}(r)),\ r\big), \tag{4}$$

where $\mathrm{ch}^*(r)$ denotes the set of children of r remaining after merging, including nodes that were not merged. Among nodes judged to represent the same relation, we retain the one with the largest number of assigned pairs, counting pairs assigned to its descendants as well; the other nodes' children and directly assigned pairs are transferred to the retained node, and those nodes are removed.

Relation Splitting. Splitting creates subtypes by dividing the pairs assigned to a relation type according to finer distinctions. A type with many assigned pairs may encompass several more specific relations. In each batch, we therefore split nodes r whose number of directly assigned pairs $|\mathcal{P}(r)|$ exceeds a threshold τ_{split}, excluding nodes that absorbed more than a fixed fraction of their pairs through Merging in the same batch. The LLM receives the node r to be split, the explanations $\{e_p\}_{p\in\mathcal{P}(r)}$ of its directly assigned pairs (sampled when there are many), and its children $\mathrm{ch}(r)$:

$$\{(r', \mathcal{P}^*(r'))\} = \mathrm{LLM}_{\mathrm{split}}\big(r,\ \{e_p\}_{p\in\mathcal{P}(r)},\ \mathrm{ch}(r)\big) \tag{5}$$

where r' is either a newly proposed child node or the original node r, and $\mathcal{P}^*(r')$ denotes the set of pairs assigned to it. The LLM proposes names and definitions for new child nodes based on additional distinctions identified in the pair explanations, using the existing children $\mathrm{ch}(r)$ as context to avoid creating subtypes that overlap with existing ones. Each pair is either moved to a single new child or retained at r.

3.2.3 Relation Verifier

The Relation Verifier validates pair assignments to the relation types organized by the Manager, using item information to determine whether each pair satisfies the definition of its assigned type, and then updates type names and definitions to reflect the assigned pairs (C2).

Assignment Verification. The Verifier validates the assignments of pairs that were newly assigned or moved to another node in the current batch. It also revalidates pairs directly assigned to nodes whose definitions have been updated, since such updates may invalidate existing assignments. For each pair p, the LLM receives both items' information, the pair explanation, the currently assigned type $\mathcal{A}(p)$, and its path $\mathrm{path}(\mathcal{A}(p))$:

$$(b_p^*, r_p^*, e_p^*) = \mathrm{LLM}_{\mathrm{verify}}\big(x_{i_j},\ x_{i_k},\ e_p,\ \mathcal{A}(p),\ \mathrm{path}(\mathcal{A}(p))\big). \tag{6}$$

If the current assignment is valid, the Verifier retains it without proposing a revision. Otherwise, it proposes a revised relation category, fine-grained relation, and explanation (b_p^*, r_p^*, e_p^*), which is added to a pending set for Routing in the next batch. Proposals remaining after the final batch are reprocessed for up to a fixed number of additional rounds. If neither relation category applies ($b_p^* =$ `None`), the assignment is removed.

Definition Refinement. A relation type's initial definition comes from the explanation of the first pair assigned to it. As more pairs are assigned or the tree structure changes, this definition may no longer represent the assigned pairs adequately. The Verifier therefore updates the names and definitions of nodes that received pair assignments or whose subtypes changed in the current batch. The LLM receives node r, its parent $\mathrm{pa}(r)$, its children $\mathrm{ch}(r)$, and the explanations of its directly assigned pairs:

$$r^* = \mathrm{LLM}_{\mathrm{refine}}\big(r,\ \mathrm{pa}(r),\ \mathrm{ch}(r),\ \{e_p\}_{p\in\mathcal{P}(r)}\big), \tag{7}$$

where r^* denotes node r with its updated name and definition; the tree structure and pair assignments remain unchanged. The updated definition must remain within the scope of $\mathrm{pa}(r)$'s definition, encompass the child types, and reflect the relations of the directly assigned pairs. Refinement proceeds

from lower to higher levels so that updates to subtypes are reflected in their parents' definitions. Pairs directly assigned to a refined node are then revalidated as described above.

3.3 Inference using Relations

The remaining two stages use the relational memory during inference. First, the recommender agent summarizes the user's history into a user memory, and then evaluates the candidates by interpreting their relations to the history items in light of this memory.

3.3.1 User Memory Construction

The item relational memory captures item relations but not individual users' preferences. To capture these, we construct a user memory from each user's history and the relational memory, which requires identifying which relations are relevant to the user's choices. For each item pair (i_j^u, i_k^u) in user u's history S_u, we retrieve the assigned relation type $\mathcal{A}(i_j^u, i_k^u)$ from the relational memory. The LLM interprets the retrieved relation types together with the history to generate the user memory.

$$m_u = \mathrm{LLM}_{\mathrm{umem}}\Big(\mathcal{S}_u,\ \{\mathcal{A}(i_j^u, i_k^u) \mid 1 \le j < k \le L,\ \mathcal{A}(i_j^u, i_k^u) \neq \emptyset\}\Big). \tag{8}$$

User memory m_u consists of two parts. The first describes item preferences, including the item categories and attributes the user tends to choose. The second describes relation-based preferences inferred from recurring relation types and transitions between types in the history, capturing the comparison criteria and usage contexts underlying the user's choices, *e.g.*, "prefers decaffeinated coffee over regular coffee." This memory provides the basis for interpreting what the relations between history items and candidates mean for the user at recommendation time.

3.3.2 Relation-aware Reranking

The recommender agent retrieves the relations between history items and candidates from the relational memory and uses them together with the user memory to rerank the candidates. For each history item $i_j^u \in S_u$ and candidate $i_c \in \mathcal{C}_u$, it retrieves the relation type $\mathcal{A}(i_j^u, i_c)$, yielding the set of retrieved relations.

$$\mathcal{R}_u = \{\mathcal{A}(i_j^u, i_c) \mid i_j^u \in \mathcal{S}_u,\ i_c \in \mathcal{C}_u,\ \mathcal{A}(i_j^u, i_c) \neq \emptyset\} \tag{9}$$

We then add the user memory m_u and the retrieved relations $\mathcal{R}_u$ to the inputs of Eq. (1). Each retrieved relation is given as its category, type name, and definition. Note that candidates with no retrieved relations are evaluated based on item information and user memory.

$$\pi_u = \mathrm{LLM}_{\mathrm{rank}}\big(S_u,\ \mathcal{C}_u,\ m_u,\ \mathcal{R}_u\big). \tag{10}$$

4 Experiments

Our experiments address the following research questions. (RQ1) Does FIRM improve reranking over prompting-based, knowledge-augmented, and agent-based methods? (RQ2) Do the Manager and the Verifier improve reranking beyond a memory built from the Explorer's proposals alone? (RQ3) How do the history–candidate relations and the user memory contribute to reranking, and does the fine granularity of relation types matter? (RQ4) Does a relational memory built once with a 7B model transfer to reranking LLMs of other scales and families?

4.1 Experimental Setup

Datasets. We use three datasets from XMarket (Bonab et al., 2021), the source also adopted by AgentDR (Yang et al., 2026): Sports, Home, and Grocery, which contain Amazon interactions in the Sports and Outdoors, Home and Kitchen, and Grocery and Gourmet Food categories, respectively. See Appendix B.1 for detailed data statistics and preprocessing.

Evaluation protocol. We adopt leave-one-out evaluation, holding out each user's last interaction for testing (Kang & McAuley, 2018; Yue et al., 2023). For each user, SASRec (Kang & McAuley, 2018) retrieves 20 candidates excluding the history items. If the ground-truth item is missing, it replaces the

Table 1: Overall performance comparison. Best and second-best results are bold and underlined. Imp. is the relative improvement of FIRM over the best baseline, and * marks significance (two-sided paired t-test over test users, $p < 0.01$).

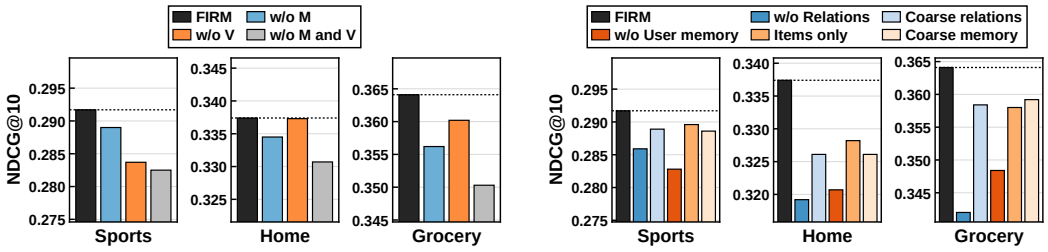
Method	Sports				Home				Grocery			
	N@1	N@3	N@5	N@10	N@1	N@3	N@5	N@10	N@1	N@3	N@5	N@10
SASRec	0.0706	0.0767	0.0792	0.0831	0.0964	0.1003	0.1011	0.1021	0.1290	0.1359	0.1389	0.1417
Prompting-based methods												
Zero-shot	0.0735	0.1589	0.1965	0.2658	0.0766	0.1746	0.2131	0.2847	0.0724	0.1869	0.2262	0.2896
CoT	<u>0.1210</u>	0.1735	0.2035	0.2582	0.0732	0.1636	0.2054	0.2758	0.0677	0.1636	0.2059	0.2720
Self-Refine	0.1069	0.1769	0.2117	0.2780	0.1036	0.1877	0.2235	0.2947	0.1236	0.2035	0.2406	0.3066
Profile	0.0846	0.1616	0.2002	0.2713	0.0853	0.1724	0.2141	0.2857	0.0957	0.1931	0.2329	0.2982
Knowledge-augmented methods												
DOKE	0.1155	0.1711	0.2046	0.2690	0.1310	0.1891	0.2204	0.2803	0.1683	0.2290	0.2563	0.3106
LLM-Rec	0.0915	0.1436	0.1762	0.2536	0.1160	0.1813	0.2188	0.2910	0.1108	0.1645	0.1965	0.2694
STAR	0.0681	0.1348	0.1806	0.2470	0.0743	0.1360	0.1785	0.2537	0.0868	0.1696	0.2219	0.2905
ItemRAG	0.0740	0.1412	0.1756	0.2460	0.0831	0.1658	0.2043	0.2771	0.0878	0.1692	0.2045	0.2704
Agent-based methods												
AgentCF	0.1162	0.1673	0.1991	0.2620	<u>0.1430</u>	0.2096	0.2492	0.3157	<u>0.1792</u>	<u>0.2424</u>	<u>0.2723</u>	<u>0.3309</u>
MemRec	0.1052	<u>0.1796</u>	<u>0.2182</u>	<u>0.2815</u>	0.1297	<u>0.2132</u>	<u>0.2535</u>	<u>0.3198</u>	0.1492	0.2271	0.2647	0.3256
AgentDR	0.1040	0.1598	0.1955	0.2674	0.1093	0.1745	0.2133	0.2858	0.1222	0.1839	0.2212	0.2896
FIRM	**0.1264**	**0.1952***	**0.2290***	**0.2917***	**0.1638***	**0.2356***	**0.2725***	**0.3374***	**0.1839***	**0.2748***	**0.3081***	**0.3641***
Imp.	4.46%	8.69%	4.95%	3.62%	14.55%	10.51%	7.50%	5.50%	2.62%	13.37%	13.15%	10.03%

lowest-ranked candidate (83.5–88.7% of test users, Appendix B.1.2). Unlike random negatives (Hou et al., 2024), the other candidates are thus hard negatives from SASRec, while the target is present for every method. The candidates are shuffled once per user and fixed, so all methods rerank them in the same order. Following Hou et al. (2024), we report NDCG@K for $K \in \{1, 3, 5, 10\}$, written as N@K, which rewards higher ranks of the target. All methods receive the $H = 5$ most recent history items, and all LLM calls use Qwen2.5-7B-Instruct unless otherwise noted.

Baselines. We compare FIRM with eleven baselines in three groups: **prompting-based methods**, namely Zero-shot (Hou et al., 2024), CoT (Wei et al., 2022), Self-Refine (Madaan et al., 2023), and Profile (Wang & Lim, 2023); **knowledge-augmented methods** that enrich the prompt with collaborative or item knowledge, namely DOKE (Yao et al., 2023), STAR (Lee et al., 2024), LLM-Rec (Lyu et al., 2024), and ItemRAG (Kim et al., 2026); and **agent-based methods** that maintain natural-language memories, namely AgentCF (Zhang et al., 2024), MemRec (Chen et al., 2026), and AgentDR (Yang et al., 2026), whose substitute-and-complement reranking we apply to the shared candidate set. Implementation details and baseline adaptations are provided in Appendix B.1.

4.2 Overall Performance Comparison (RQ1)

Table 1 compares FIRM with the eleven baselines. We highlight three key findings. (i) FIRM achieves the best results in all metrics on all three datasets, with statistically significant gains in every metric except NDCG@1 on Sports. It improves NDCG@10 over the strongest baseline by 6.39% on average and by up to 10.03% on Grocery. The relative gains grow as the cutoff decreases from 10 to 3, averaging 8.53% in NDCG@5 and 10.86% in NDCG@3, indicating that FIRM ranks the target item closer to the top. (ii) Interaction knowledge yields the most consistent gains when kept in a memory. The knowledge-augmented methods supply it without a memory: all except STAR on Sports and Home improve NDCG@1 over Zero-shot, yet their gains in NDCG@10 are inconsistent or absent, even though STAR and DOKE directly link each candidate to the history items. In contrast, the strongest baseline from NDCG@3 to NDCG@10 on every dataset maintains natural-language memories of users and items, namely MemRec on Sports and Home and AgentCF on Grocery. The further gains of FIRM indicate that specifying how items are compared or used together adds evidence beyond separate item memories and bare links. (iii) AgentDR improves NDCG@1 over Zero-shot but only matches it in NDCG@10. We attribute this to its ranking by similarity to user-specific substitute and complement lists, which pushes down candidates similar to neither list. The profile-based ranking fused with a learned weight only partly offsets this. FIRM outperforms AgentDR by 17.62% in NDCG@10 on average, consistent with retrieving fine-grained relations from a shared memory and judging their fit to user preferences.



(a) Relation construction (RQ2)

(b) Relation utilization (RQ3)

Figure 2: NDCG@10 of FIRM and its variants. (a) removes the Manager (M), the Verifier (V), or both, and rebuilds the relation memory and the user memories from the same Explorer proposals. (b) varies either the history–candidate relation inputs in blue or the user memory construction in orange while keeping the other at the full setting, where Coarse uses only the substitution or complement label. Each panel uses its own y-axis range.

4.3 Effect of Relation Construction and Utilization (RQ2 and RQ3)

Figure 2a isolates the contribution of the Manager (M) and the Verifier (V). Without M, each distinct proposed name remains a separate relation type outside any relation tree, and V verifies each pair only against the definition of its assigned type. Without V, assignments and definitions remain as M produces them. The key observations are as follows. (i) Using both agents performs best in NDCG@10 on all three datasets, and removing both performs worst, lowering NDCG@10 by 3.2%, 2.0%, and 3.8% on Sports, Home, and Grocery, respectively. (ii) Removing M lowers NDCG@10 on every dataset even with V, since shared types let the user memory capture recurring relations and the reranker interpret the history–candidate relations, whereas independent inference names the same relation differently (C2). (iii) The effect of V on NDCG@10 depends on the dataset: removing V lowers it by 2.74% on Sports, where w/o V performs close to w/o M and V, but by only 0.03% and 1.07% on Home and Grocery. NDCG@3 drops on all three datasets, by 5.02%, 2.97%, and 1.93% (Table 4), so on Home the Verifier mainly affects the top of the ranking.

Figure 2b varies the two reranking inputs with the relational memory fixed: the history–candidate relations are removed (w/o Relations) or replaced by their top-level category, `Sub` or `Com` (Coarse relations), and the user memory is removed (w/o User memory), built without relations (Items-only memory), or built with the top-level categories (Coarse memory). (i) The retrieved relations contribute beyond the user memory: removing them lowers NDCG@10 by 4.5% on average, as they specify how each candidate is compared with or used with the history items. Consistently, the gain concentrates on users for whom relations are retrieved (Appendix B.5). (ii) The user memory adds information beyond item preferences: removing it lowers NDCG@10 by 4.1% on average, and the items-only memory lies between the two, so relation-based preferences account for the difference. (iii) Fine-grained relation types add a consistent gain in both inputs (C1): relation types improve NDCG@10 over the top-level categories on all three datasets, by 2.0% on average, and the full memory outperforms the items-only memory everywhere, whereas Coarse memory falls below it on Sports and Home, so relation-based preferences help on every dataset only with fine-grained types.

4.4 Transferability of Relational Memory (RQ4

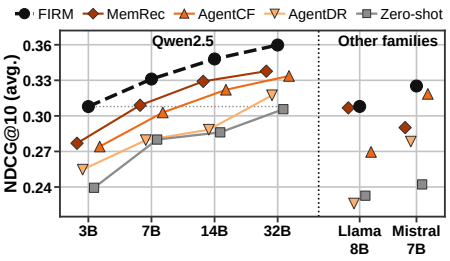


Figure 3: Average NDCG@10 over three datasets per reranking LLM, memories built with Qwen2.5-7B.

We fix all memories built with Qwen2.5-7B-Instruct, including those of the memory-based baselines, and replace only the reranking LLM with Qwen2.5 models of 3B, 14B, and 32B and with Llama-3.1-8B and Mistral-7B-Instruct from other families. Figure 3 plots the average NDCG@10 over the three datasets, and Tables 6 and 7 in Appendix B report the results per dataset. The key observations are as follows. (i) Across scales, FIRM achieves the highest NDCG@10 averaged over the three datasets at every size, and is the best in NDCG@10 on every dataset except Sports at 14B, where MemRec is marginally higher. Its NDCG@10 increases with scale on

all three datasets, by 16.8% on average from 3B to 32B. Its average margin over Zero-shot with the same reranker at 14B and 32B remains comparable to or larger than that at 7B, so larger rerankers still benefit from the memory. (ii) Across families, FIRM achieves the highest NDCG@10 averaged over the three datasets with both families and improves over Zero-shot on every dataset, though MemRec leads on Sports and Home with Llama-3.1-8B and AgentCF on Sports with Mistral-7B. These results suggest that other rerankers can interpret the relation types and definitions stored in natural language, so the memory can be built once and reused across reranking LLMs. Building the memory is a one-time cost of 3.4 to 6.5 hours, 17% to 38% of that of AgentCF (Appendix B.4).

5 Related Work

Memory-augmented Recommendation. Memory-augmented recommenders accumulate knowledge from interactions and retrieve it for the current recommendation. (i) One line stores user and item information. AgentCF (Zhang et al., 2024) updates user and item memories from the discrepancy between predicted and actual choices, and MemRec (Chen et al., 2026) synthesizes the memories of neighbors in a collaborative graph. AMEM4Rec (Nguyen et al., 2026a) and STEAM (Liao et al., 2026) share behavioral patterns or fine-grained preference units across users. (ii) Another line stores knowledge from recommendation experience, such as lessons distilled from past recommendations (Wang et al., 2025; Qin et al., 2025; Nguyen et al., 2026b), per-user recommendation policies (Tao et al., 2026), and skills for consulting other users' agents (Wang et al., 2026b). Closest to ours, CGM-Rec (Nguyen Ngoc et al., 2026) keeps a graph memory of item attributes and relations, but edits it according to recommendation outcomes. FIRM instead derives fine-grained relations between item pairs from item information and organizes them into relation types with shared definitions before recommendation, so that the same relational knowledge is reused across users.

Item Relations for Recommendation. Item relations are either learned within recommendation models or provided to LLMs as evidence. (i) Sceptre (McAuley et al., 2015) predicts substitutes and complements from review text and product features, and MMSC (Wang et al., 2026a) infers them from multimodal item representations. RCF (Xin et al., 2019) and LRD (Yang et al., 2024) learn item relations from shared attributes or LLM-based item embeddings jointly with recommendation models. FolkScope (Yu et al., 2023) and COSMO (Yu et al., 2024) generate the intentions behind purchases with LLMs, refine them through human annotation, and organize them into knowledge graphs of commonsense relations such as *UsedFor* and *IsA* for downstream models. (ii) STAR (Lee et al., 2024), ItemRAG (Kim et al., 2026), and DOKE (Yao et al., 2023) provide LLMs with co-consumption counts, summaries of related items, and knowledge graph paths, respectively, and AgentDR (Yang et al., 2026) reranks candidates by their similarity to user-specific lists of substitutes and complements. Beyond recommendation, LLMs have been used to induce label taxonomies from batches of text (Wan et al., 2024) and to extract, define, and canonicalize relations for knowledge-graph construction (Zhang & Soh, 2024). Unlike these approaches, FIRM captures fine-grained comparison criteria and usage roles within substitution and complementarity, and lets a training-free LLM reranker interpret the retrieved relations in light of the user memory.

6 Conclusion

We presented FIRM, a fine-grained item relational memory for LLM-based recommender agents. Unlike existing item memories, FIRM specifies how items are related through relation types organized in a consistent hierarchy, which the Explorer proposes, the Manager canonicalizes, and the Verifier validates. At inference, the recommender agent summarizes each user's history into a user memory of item and relation-based preferences and reranks candidates by how well their relations to history items fit this memory. On three real-world datasets, FIRM improved NDCG@10 over the strongest baseline by 6.39% on average and by up to 10.03%.

Limitations and future work. The main limitation of FIRM is that the memory is static: it is built once from training interactions, so its reach is bounded by the coverage of pair selection (Appendix B.5), and items that enter the catalog later have no relations until it is rebuilt. We therefore plan to update the memory incrementally as new items and interactions arrive, and to extend relations from item pairs to sets of items consumed together.

Reproducibility Statement

All experiments use the publicly available XMarket datasets and open-weight LLMs. Section 4.1 describes the datasets, the candidate construction, and the evaluation protocol. Appendix A.2 gives the construction procedure of the relation memory, and Appendix B.1 details the data preprocessing, the implementation settings (models, serving, decoding, and hyperparameters), and the adaptation of each baseline. The prompts for all eight LLM operations are reproduced verbatim in Appendix C. The relation memory is built only from training interactions before evaluation and kept fixed during evaluation, and each user memory is generated at request time from the user's training interactions only. We will release the code and the constructed relation memories upon acceptance.

AI Use Statement

Large language models are a core component of this research: open-weight LLMs serve as the relation-construction agents and the recommender agent of FIRM and as the rerankers of all baselines, as described in the main text. Apart from this role, we used generative AI tools for implementing the method and the baselines, for translating an earlier draft of the manuscript into English, and for obtaining feedback on the experimental design and on the interpretation of the results during manuscript revision. We did not use generative AI tools for generating synthetic datasets, developing theoretical models or the conceptual framework, proposing or refining hypotheses, or cleaning and reformatting the datasets. Formulating or proving mathematical claims and qualitative or thematic data analysis are not applicable to this work. Additionally, we used generative AI tools for drafting and editing parts of the manuscript for clarity and for identifying and summarizing related literature. We have reviewed all AI-assisted work: AI-generated code was tested and its outputs were checked against the experimental logs and data, every suggested reference was verified against the original source, and all AI-drafted or translated text was reviewed by the authors before being adopted. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

References

Hamed Bonab, Mohammad Aliannejadi, Ali Vardasbi, Evangelos Kanoulas, and James Allan. Cross-market product recommendation. In *CIKM*, 2021.

Weixin Chen, Yuhan Zhao, Jingyuan Huang, Zihe Ye, Mingxuan Ju, Tong Zhao, Neil Shah, Li Chen, and Yongfeng Zhang. MemRec: Collaborative memory-augmented agentic recommender system. In *ACL*, 2026.

Yupeng Hou, Junjie Zhang, Zihan Lin, Hongyu Lu, Ruobing Xie, Julian McAuley, and Wayne Xin Zhao. Large language models are zero-shot rankers for recommender systems. In *ECIR*, 2024.

Wang-Cheng Kang and Julian McAuley. Self-attentive sequential recommendation. In *ICDM*, 2018.

Sunwoo Kim, Geon Lee, Kyungho Kim, Jaemin Yoo, and Kijung Shin. ItemRAG: Item-based retrieval-augmented generation for LLM-based recommendation. In *SIGIR*, 2026.

Dong-Ho Lee, Adam Kraft, Long Jin, Nikhil Mehta, Taibai Xu, Lichan Hong, Ed H. Chi, and Xinyang Yi. STAR: A simple training-free approach for recommendations using large language models. *arXiv preprint arXiv:2410.16458*, 2024.

Jiayi Liao, Sihang Li, Zhengyi Yang, Jiancan Wu, Yancheng Yuan, Xiang Wang, and Xiangnan He. LLaRA: Large language-recommendation assistant. In *SIGIR*, 2024.

Yuxin Liao, Le Wu, Min Hou, Yu Wang, Han Wu, and Meng Wang. From atom to community: Structured and evolving agent memory for user behavior modeling. *arXiv preprint arXiv:2601.16872*, 2026.

Jianghao Lin, Xinyi Dai, Yunjia Xi, Weiwen Liu, Bo Chen, Hao Zhang, Yong Liu, Chuhan Wu, Xiangyang Li, Chenxu Zhu, Huifeng Guo, Yong Yu, Ruiming Tang, and Weinan Zhang. How can recommender systems benefit from large language models: A survey. *TOIS*, 2025.

Hanjia Lyu, Song Jiang, Hanqing Zeng, Yinglong Xia, Qifan Wang, Si Zhang, Ren Chen, Christopher Leung, Jiajie Tang, and Jiebo Luo. LLM-Rec: Personalized recommendation via prompting large language models. In *Findings of NAACL*, 2024.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *NeurIPS*, 2023.

Julian McAuley, Rahul Pandey, and Jure Leskovec. Inferring networks of substitutable and complementary products. In *KDD*, 2015.

Minh-Duc Nguyen, Hai-Dang Kieu, and Dung D. Le. AMEM4Rec: Leveraging cross-user similarity for memory evolution in agentic LLM recommenders. *arXiv preprint arXiv:2602.08837*, 2026a.

Minh Hoang Nguyen, Tung Le, and Huy Tien Nguyen. rEDMRec: Distilling large language model reasoning into an editable experience memory for recommendation. *arXiv preprint arXiv:2608.18952*, 2026b.

Hao Nguyen Ngoc, Tung Nguyen, Nguyen Thi Hanh, Hoang Thai Dinh, and Nguyen Xuan Tung. Continual graph memory for adaptive recommendation under intent drift. *arXiv preprint arXiv:2609.04651*, 2026.

Weicong Qin, Yi Xu, Weijie Yu, Chenglei Shen, Xiao Zhang, Ming He, Jianping Fan, and Jun Xu. MoRE: A mixture of reflectors framework for large language model-based sequential recommendation. In *RecSys*, 2025.

Zhen Tao, Riwei Lai, Chenyun Yu, Weixin Chen, Li Chen, Beibei Kong, Lei Cheng, Chengxiang Zhuo, Zang Li, and Qingqiang Sun. SAGER: Self-evolving user policy skills for recommendation agent. *arXiv preprint arXiv:2604.14972*, 2026.

Mengting Wan, Tara Safavi, Sujay Kumar Jauhar, Yujin Kim, Scott Counts, Jennifer Neville, Siddharth Suri, Chirag Shah, Ryen W. White, Longqi Yang, Reid Andersen, Georg Buscher, Dhruv Joshi, and Nagu Rangan. TnT-LLM: Text mining at scale with large language models. In *KDD*, 2024.

Junting Wang, Chenghuan Guo, Yang Jiao, Yanhui Guo, Hari Sundaram, and Yan Gao. Multi-modal relational item representation learning for inferring substitutable and complementary items. In *SIGIR*, 2026a.

Lei Wang and Ee-Peng Lim. Zero-shot next-item recommendation using large pretrained language models. *arXiv preprint arXiv:2304.03153*, 2023.

Ziyan Wang, Yingpeng Du, Zhu Sun, Haoyan Chua, Kaidong Feng, Wenya Wang, and Jie Zhang. Re2LLM: Reflective reinforcement large language model for session-based recommendation. In *AAAI*, 2025.

Zongwei Wang, Min Gao, Guangyu Hu, Xinyi Gao, and Junliang Yu. Personalized communication skills for agentic recommender systems. *arXiv preprint arXiv:2608.08417*, 2026b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

Yunjia Xi, Weiwen Liu, Jianghao Lin, Xiaoling Cai, Hong Zhu, Jieming Zhu, Bo Chen, Ruiming Tang, Weinan Zhang, and Yong Yu. Towards open-world recommendation with knowledge augmentation from large language models. In *RecSys*, 2024.

Xin Xin, Xiangnan He, Yongfeng Zhang, Yongdong Zhang, and Joemon M. Jose. Relational collaborative filtering: Modeling multiple item relations for recommendation. In *SIGIR*, 2019.

Mingdai Yang, Nurendra Choudhary, Jiangshu Du, Edward W. Huang, Philip S. Yu, Karthik Subbian, and Danai Koutra. AgentDR: Dynamic recommendation with implicit item-item relations via LLM-based agents. In *WWW*, 2026.

Shenghao Yang, Weizhi Ma, Peijie Sun, Qingyao Ai, Yiqun Liu, Mingchen Cai, and Min Zhang. Sequential recommendation with latent relations based on large language model. In *SIGIR*, 2024.

Jing Yao, Wei Xu, Jianxun Lian, Xiting Wang, Xiaoyuan Yi, and Xing Xie. Knowledge plugins: Enhancing large language models for domain-specific recommendations. *arXiv preprint arXiv:2311.10779*, 2023.

Changlong Yu, Weiqi Wang, Xin Liu, Jiaxin Bai, Yangqiu Song, Zheng Li, Yifan Gao, Tianyu Cao, and Bing Yin. FolkScope: Intention knowledge graph construction for e-commerce commonsense discovery. In *Findings of ACL*, 2023.

Changlong Yu, Xin Liu, Jefferson Maia, Tianyu Cao, Yang Li, Yifan Gao, Yangqiu Song, Rahul Goutam, Haiyang Zhang, Bing Yin, and Zheng Li. COSMO: A large-scale e-commerce common sense knowledge generation and serving system at Amazon. In *SIGMOD Companion*, 2024.

Zhenrui Yue, Sara Rabhi, Gabriel de Souza Pereira Moreira, Dong Wang, and Even Oldridge. LlamaRec: Two-stage recommendation using large language models for ranking. *arXiv preprint arXiv:2311.02089*, 2023.

Bowen Zhang and Harold Soh. Extract, define, canonicalize: An LLM-based framework for knowledge graph construction. In *EMNLP*, 2024.

Junjie Zhang, Yupeng Hou, Ruobing Xie, Wenqi Sun, Julian McAuley, Wayne Xin Zhao, Leyu Lin, and Ji-Rong Wen. AgentCF: Collaborative learning with autonomous language agents for recommender systems. In *WWW*, 2024.

A Method Details

This section details the inputs, prompt instructions, and rule-based steps of the three stages of FIRM.

A.1 Notation Table

Table 2 summarizes the notation of the main text and the input and output of each LLM operation.

Table 2: Notation used in the main text (top) and the input and output of each LLM operation in FIRM, grouped by the agent that issues it (bottom).

Symbol	Description	Symbol	Description
Sequential recommendation (§2)			
$\mathcal{U}$, $\mathcal{I}$	Sets of users and items	x_i	Textual description of item i
$\mathcal{S}_u = [i_1^u, \ldots, i_L^u]$	History of user u, oldest first	i_l^u	Item of the l-th interaction of u
$\mathcal{C}_u$	Candidates retrieved for user u	π_u	Ranking of $\mathcal{C}_u$
Relational memory (§3.2)			
$\mathcal{M} = (\mathcal{T}, \mathcal{A})$	Relational memory	$p = (i_j, i_k)$	Unordered item pair
$\mathcal{T}$	Relation tree	$\mathcal{A}(p)$	Node assigned to p, $\emptyset$ if unassigned
$b \in \{\texttt{Sub}, \texttt{Com}\}$	Relation category, a root of $\mathcal{T}$	$\mathcal{P}(r)$	Pairs assigned directly to node r
$r \in \mathcal{T}$	Relation node, with name and definition	$\mathrm{ch}(r)$, $\mathrm{pa}(r)$	Children and parent of node r
$\mathrm{ch}(b)$	Higher-level types under root b	$\mathrm{path}(r)$	Nodes from the root to r
Relation construction (§3.2.1 to §3.2.3)			
ρ	Fraction of nearest items paired per space	B	Number of pairs per batch
m	Co-consumption threshold	τ_{split}	Split threshold on $\lvert\mathcal{P}(r)\rvert$
$\mathcal{P}_{\text{CF}}$, $\mathcal{P}_{\text{T}}$, $\mathcal{P}_{\text{co}}$	Pairs from CF, text, co-consumption	(b_p, r_p, e_p)	Proposed category, name, explanation for p
$\mathcal{P}$, $\mathcal{P}_t$	Selected pairs, their t-th batch	$\mathcal{Q}$	Pending proposals not yet placed
Inference (§3.3)			
m_u	User memory of user u	$i_c \in \mathcal{C}_u$	A candidate item
H	History items shown to the reranker	$\mathcal{R}_u$	Retrieved history–candidate relations

Operation	Input	Output
Relation Explorer (§3.2.1)		
$\text{LLM}_{\text{explore}}$, Eq. (2)	x_{i_j}, x_{i_k}	Proposal (b_p, r_p, e_p) for pair p
Relation Tree Manager (§3.2.2)		
$\text{LLM}_{\text{route}}$, Eq. (3)	$\{(r_p, e_p)\}$ of new types, $\mathrm{ch}(b)$	Parent of each new type, existing or new
$\text{LLM}_{\text{merge}}$, Eq. (4)	$\mathrm{ch}(r)$, $\mathrm{ch}(\mathrm{pa}(r))$, r	$\mathrm{ch}^*(r)$, children of r after merging
$\text{LLM}_{\text{split}}$, Eq. (5)	r, $\{e_p\}_{p \in \mathcal{P}(r)}$, $\mathrm{ch}(r)$	Subtypes r' with their pairs $\mathcal{P}^*(r')$
Relation Verifier (§3.2.3)		
$\text{LLM}_{\text{verify}}$, Eq. (6)	x_{i_j}, x_{i_k}, e_p, $\mathcal{A}(p)$, $\mathrm{path}(\mathcal{A}(p))$	$\mathcal{A}(p)$ kept, or revised (b_p^*, r_p^*, e_p^*)
$\text{LLM}_{\text{refine}}$, Eq. (7)	r, $\mathrm{pa}(r)$, $\mathrm{ch}(r)$, $\{e_p\}_{p \in \mathcal{P}(r)}$	r^*, node r with updated name and definition
Recommender agent (§3.3)		
LLM_{umem}, Eq. (8)	$\mathcal{S}_u$ and the relations among its items	User memory m_u
LLM_{rank}, Eq. (10)	$\mathcal{S}_u$, $\mathcal{C}_u$, m_u, $\mathcal{R}_u$	Ranking π_u

A.2 Construction Procedure

Algorithm 1 summarizes the construction of the relation memory $\mathcal{M} = (\mathcal{T}, \mathcal{A})$, where the children of the two roots are higher-level types and every node below them is a relation type. The Explorer first proposes a relation for every selected pair, and the Manager and the Verifier then update the

memory batch by batch. We refer to each call to UPDATE as a round. A round processes one batch $\mathcal{P}_t$, while the additional rounds after the final batch process only the pending proposals. The pending set $\mathcal{Q}$ holds the latest relation proposal of each pair awaiting placement in the tree, including the revised proposals returned by the Verifier. Proposals that remain in $\mathcal{Q}$ after n_{retry} additional rounds are discarded. A relation type that still has no parent is attached to the higher-level type closest to it in embedding space when their cosine similarity is at least θ, and deleted otherwise, in which case its pairs are left unassigned. The construction then ends with n_{retry} consolidation rounds in which no new Explorer proposal enters, while revised proposals from the Verifier still enter $\mathcal{Q}$ and are routed in the following round: every sibling set is a merge candidate, every node with more than τ_{split} directly assigned pairs is split, every assigned pair is verified against the current definitions, and every type is refined. Proposals left in $\mathcal{Q}$ after the last consolidation round are discarded as well, and their pairs remain unassigned. Two rule-based steps finally rename generic types and fold subtypes equivalent to their parent (Appendix A.5). The following subsections describe, agent by agent, which nodes or pairs each operation acts on and when it is applied.

Algorithm 1 Multi-Agent Relation Construction

Require: Pair batches $\mathcal{P}_1, \mathcal{P}_2, \ldots$ with $\mathcal{P} = \bigcup_t \mathcal{P}_t$, product information $\{x_i\}_{i \in \mathcal{I}}$
Ensure: Relation memory $\mathcal{M} = (\mathcal{T}, \mathcal{A})$

$\mathcal{T} \leftarrow \{\texttt{Sub}, \texttt{Com}\},\ \mathcal{A} \leftarrow \emptyset,\ \mathcal{M} \leftarrow (\mathcal{T}, \mathcal{A}),\ \mathcal{Q} \leftarrow \emptyset$ ▷ $\mathcal{Q}$: pending proposals
for all $p = (i_j, i_k) \in \mathcal{P}$ **do** ▷ Explorer
  $(b_p, r_p, e_p) \leftarrow \text{LLM}_{\text{explore}}(x_{i_j}, x_{i_k})$
for all batches $\mathcal{P}_t$ in order **do**
  $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(p, b_p, r_p, e_p) \mid p \in \mathcal{P}_t,\ b_p \neq \texttt{None}\}$
  $\mathcal{M}, \mathcal{Q} \leftarrow \text{UPDATE}(\mathcal{M}, \mathcal{Q})$
while $\mathcal{Q} \neq \emptyset$ and fewer than n_{retry} additional rounds have run **do** ▷ retry pending proposals
  $\mathcal{M}, \mathcal{Q} \leftarrow \text{UPDATE}(\mathcal{M}, \mathcal{Q})$
$\mathcal{Q} \leftarrow \emptyset$ ▷ discard proposals still pending
attach each parentless type to its nearest higher-level type, or delete it ▷ Appendix A.2
for n_{retry} consolidation rounds **do** ▷ every node and pair eligible, no new proposals
  $\mathcal{M}, \mathcal{Q} \leftarrow \text{UPDATE}(\mathcal{M}, \mathcal{Q})$
$\mathcal{M} \leftarrow \text{RENAME}(\mathcal{M});\ \mathcal{M} \leftarrow \text{FOLD}(\mathcal{M})$ ▷ post-construction steps, Appendix A.5
return $\mathcal{M}$

procedure $\text{UPDATE}(\mathcal{M}, \mathcal{Q})$ ▷ one round
  Manager
  $\mathcal{M}, \mathcal{Q} \leftarrow \text{ROUTE}(\mathcal{M}, \mathcal{Q})$ ▷ Eq. (3), placed proposals leave $\mathcal{Q}$
  $\mathcal{M} \leftarrow \text{MERGE}(\mathcal{M})$ ▷ Eq. (4), top-down from the roots
  $\mathcal{M} \leftarrow \text{SPLIT}(\mathcal{M})$ ▷ Eq. (5), nodes with $|\mathcal{P}(r)| > \tau_{\text{split}}$
  Verifier
  $\mathcal{M}, \mathcal{Q} \leftarrow \text{VERIFY}$(pairs assigned or moved in this round) ▷ Eq. (6), unmatched misfits enter $\mathcal{Q}$
  $\mathcal{M} \leftarrow \text{REFINE}$(nodes changed in this round) ▷ Eq. (7), bottom-up
  $\mathcal{M}, \mathcal{Q} \leftarrow \text{VERIFY}$(pairs directly assigned to refined nodes) ▷ Eq. (6), once

Hyperparameters. Besides the batch size B and the split threshold τ_{split}, the construction reuses the co-consumption threshold m of Pair Selection as the minimum size of a new parent node in Routing, counted in relation types, and of a new subtype in Splitting, counted in pairs. We set the maximum number of pairs per verification call to $n_{\text{ver}} = 20$, and use the same value for the number of explanations per refinement batch, of relation types per routing call, and of parent nodes shown in a routing call. The maximum number of additional rounds is $n_{\text{retry}} = 3$, which also sets the number of consolidation rounds; in these rounds the minimum size m of a new parent node is relaxed to one. Several rules below compare relation types by the cosine similarity between embeddings of the string "*name*: *definition*", computed with the same encoder as the item text. A single threshold $\theta = 0.95$ governs these comparisons in Routing, Merging, Splitting, and the attachment of parentless types; a refined definition is discarded when it fits the explanations worse than the current one by more than $1 - \theta$, and splitting is limited to $d_{\max} = 2$ levels below a higher-level type.

A.3 Relation Explorer

Pair Selection. We take the collaborative embeddings from the item embeddings of the SASRec model used for candidate retrieval, and compute the text embeddings with bge-large-en-v1.5 from the concatenated title, description, and product category of each item.

Relation Exploration. For each selected pair $p = (i_j, i_k)$, the Explorer proposes a relation from the product information x_{i_j} and x_{i_k}. Each x_i consists of the title, description, and product category of item i, with the description truncated to 300 characters. The prompt defines substitution (`Sub`) as the case where the two items serve as alternatives for the same purpose, and complementarity (`Com`) as the case where one item enables the use of the other or increases its utility. If the product information supports neither category, the Explorer returns `None`. Otherwise, it returns the relation category b_p, a relation name r_p of at most three words that names the comparison criterion for substitution or the usage role for complementarity, and a one-sentence explanation e_p of what distinguishes the pair from other pairs in the same category.

A.4 Relation Tree Manager

Batching. We represent each pair by the average of the text embeddings of its two items and run k-means on these representations with cosine similarity, setting the number of clusters to the number of batches, $\lceil |\mathcal{P}|/B \rceil$. The pairs are then sorted by cluster and divided into consecutive batches of B pairs, so that semantically similar pairs are processed together.

Relation Routing. Routing starts with a rule-based match for each proposal in $\mathcal{Q}$. If the subtree of root b_p contains a relation type whose normalized name equals that of r_p, the pair joins that type. Otherwise, a new relation type is registered with r_p and e_p as its initial name and definition, and the pair is assigned to it. We normalize names by lowercasing them and removing whitespace, hyphens, underscores, and other punctuation. A proposal also joins an existing type when the cosine similarity between their embeddings is at least θ. The names a type carried before being renamed, and the names of the types it absorbed, remain valid keys, so that later proposals under an earlier name still reach it. Proposals whose name is a generic placeholder such as `type` or `function` are matched by the embedding rule only. The LLM is used only in the subsequent step of Eq. (3), which attaches each relation type without a parent to the tree, including the types deferred in earlier rounds. For each such type, it either selects an existing child of root b or proposes a new parent node. A new parent is added under root b only when at least m semantically close relation types of the same category are grouped under it. If no existing parent fits and this condition is not met, the type is deferred, and the proposals of its pairs stay in $\mathcal{Q}$ for the next round. New relation types are routed in batches clustered by embedding, and each call lists the n_{ver} existing parent nodes closest to the batch. Types that remain without a parent after all batches of a round are routed once more within the same round so that they can join parents created by earlier batches.

Relation Merging. Merging visits nodes top-down from the roots and consolidates sibling relation types that represent the same relation. The prompt instructs the LLM to merge relation types only when their definitions apply to the same range of item pairs, and to keep them separate when one applies to only a subset of the pairs covered by the other or when they represent different comparison criteria or roles. Among the nodes to be merged, we retain the one with the most pairs, counting both its directly assigned pairs and those of its descendants. The remaining nodes are removed after their children and directly assigned pairs are moved to the retained node. When a node has more children than fit in one call, the children are grouped by embedding similarity, and children that are nearest to each other across groups are compared in an additional call. Types that sit under different parent nodes but whose embeddings have a cosine similarity of at least θ are also submitted as merge candidates, with the root as their common parent. In every case, whether they represent the same relation is decided by the LLM. When the retained node carries a generic name, it takes over the name of the absorbed node.

Relation Splitting. Splitting applies to each node r with $|\mathcal{P}(r)| > \tau_{\text{split}}$. Nodes that absorbed more than a fraction $1 - \theta$ of their pairs during Merging in the same round are excluded, which prevents a newly merged node from being split again immediately; nodes more than $d_{\max}$ levels

below a higher-level type are not split, and a node whose previous split moved no pair is not split again until it receives new pairs. A call receives at most $3n_{\text{ver}}$ pair explanations; for nodes with more directly assigned pairs, up to three such samples are sent in parallel, and the proposals are aggregated by name before the following rules are applied. Each proposed subtype must contain at least m pairs; the prompt states this minimum as a fixed value of 2, which equals m in all our experiments, and the rule-based check enforces m regardless of the prompt. Subtypes with fewer pairs are rejected, and their pairs remain at that node. A proposed subtype is also rejected when it would take more than a fraction θ of the pairs sent, when its embedding has a cosine similarity of at least θ with that of the node or of one of its ancestors, or when its name is a single generic word. If a proposed subtype has the same name as an existing child, it is merged into that child together with its pairs.

A.5 Relation Verifier

Assignment Verification. The Verifier checks pairs assigned to the same relation type together, with at most n_{ver} pairs per call and each item description truncated to 300 characters. Since the pairs in a call share their assigned type, the input includes, as relation context, only the nodes on the path from the root to that type. For each pair, the Verifier outputs `ok` if the assignment is valid. Otherwise, it returns `misfit` with a revised proposal (b_p^*, r_p^*, e_p^*). A misfit pair is unassigned if $b_p^* = \texttt{None}$. For any other misfit pair, we apply the rule-based match of Routing to find a relation type in the subtree of root b_p^* whose normalized name equals that of r_p^*. If such a type exists, the pair is verified again against that type and its path and is moved there only if this second check returns `ok`. A pair that fails the second check keeps its original assignment. When no such type exists, the pair is removed from its current type, and the revised proposal is added to $\mathcal{Q}$ to be routed in the next round like any other proposal. A misfit judgment that arrives without a revised proposal violates the output format and is requested again in a smaller call; if the proposal is still missing, the pair keeps its original assignment.

Definition Refinement. In each round, Refinement applies to the nodes that (i) had their directly assigned pairs changed, (ii) gained a new child or had an existing child renamed or redefined, or (iii) absorbed other nodes during Merging. The prompt instructs the LLM to preserve what the existing definition already covers, and to name each substitution type by the criterion on which its items are compared and each complementarity type by the role its items play when used together. Each round sends only the explanations of pairs not yet reflected in the definition, together with a fixed sample of ten explanations already reflected, in at most five batches per node. An update is discarded when the revised name and definition fit these explanations worse than the current ones by more than $1 - \theta$, measured as the mean cosine similarity between the explanation embeddings and the embedding of the type, and a proposed name that already exists under the same root is not applied. Refinement proceeds bottom-up, and the pairs directly assigned to the refined nodes are verified once more within the same round.

Post-construction consolidation. After the last round, two rule-based steps are applied to the relation tree. Types whose name is a generic word or is not in snake case are renamed with the refinement prompt from a random sample of n_{ver} of their pair explanations, keeping the current name when the LLM returns another generic word or a name already in use under the same root. A subtype whose refined name and definition have become equivalent to its parent's, with a cosine similarity of at least θ between their embeddings, is folded into the parent, which receives its pairs and children. Neither step changes which pairs are assigned.

A.6 User Memory Construction

The LLM summarizes the history $\mathcal{S}_u$ and the relations among its items into a user memory m_u. The history is provided as a list of item titles with their category paths, together with which items were repurchased and the number of distinct items. For each pair of history items with an assigned relation type, ordered by the position of the later item, the input lists the two items, the relation category, the type name, and the first 100 characters of the type definition. The output consists of exactly two sentences, the first describing the product categories and attributes the user tends to choose, and the second describing relation-based preferences inferred from recurring relation types and transitions

between them. When no pair has an assigned relation type, the second sentence instead describes whether the user repurchases items or tries new ones.

A.7 Relation-aware Reranking

The recommender agent scores each candidate based on the user memory m_u, the recent history, and the history–candidate relations retrieved from the relation memory. The prompt includes the H most recent items of $\mathcal{S}_u$, and relations are retrieved only between these items and the candidates. History items and candidates are listed as numbered item titles, with the history ordered from oldest to most recent. Each retrieved relation is written on one line with the candidate, the history item and its recency, the relation category, the type name, and the definition, in this order. Relations involving more recent history items are listed first. If no relation is retrieved, the prompt states this explicitly. The LLM outputs an integer score from 0 to 100 for each candidate in JSON format, and the candidates are ranked by score in descending order, with ties broken by the candidate order in the prompt, which is shuffled once per user.

B Experiments

B.1 Experimental Setup

B.1.1 Datasets and Preprocessing

We use the Sports and Outdoors, Home and Kitchen, and Grocery and Gourmet Food categories of Amazon XMarket.[1] Following the preprocessing of AgentDR (Yang et al., 2026), we discard items without titles, sort the interactions of each user by timestamp to form the interaction sequence, and remove users and items with fewer than 5 interactions. Each item comes with a title, a description, and a product category. Table 3 summarizes the statistics of the three datasets.

Table 3: Statistics of the datasets.

	Sports	Home	Grocery
# Users	10,873	12,557	7,339
# Items	4,306	7,064	4,151
# Interactions	55,990	63,330	40,339
Density	0.1196%	0.0714%	0.1324%
Avg. sequence length	5.15	5.04	5.50

B.1.2 Implementation Details

Pair selection uses $\rho = 0.001$ and $m = 2$. The Manager processes batches of $B = 2000$ pairs and splits nodes with more than $\tau_{\text{split}} = 20$ directly assigned pairs.

Candidate construction. LLM rerankers are evaluated under two protocols. One ranks the target against randomly sampled negatives, which guarantees that the target is in the candidate set but leaves the negatives unrelated to the user (Hou et al., 2024). The other reranks candidates retrieved by a conventional recommender, which provides hard negatives but bounds every metric by the recall of the retriever, so LlamaRec (Yue et al., 2023) reports results separately on the users whose target is retrieved and AgentDR (Yang et al., 2026) evaluates full ranking. We combine the two: the candidates are the SASRec top-20, and the target replaces the lowest-ranked candidate when it is missing, so all rerankers are compared on the same hard candidates with the target present. The ground-truth item is in SASRec's top-20 for 11.4%, 11.3%, and 16.5% of test users on Sports, Home, and Grocery, respectively, and is inserted for the rest. The SASRec row in Table 1 ranks the same 20 candidates by SASRec's own scores.

The reranking prompt includes the $H = 5$ most recent history items. The maximum output length is 900 tokens for exploration, 3,000 tokens for the Manager and the Verifier, 240 tokens for user memory

[1] https://xmrec.github.io/

construction, and 512 tokens for reranking. Unless otherwise noted, all LLM calls use Qwen2.5-7B-Instruct, served with vLLM 0.7.3 (bfloat16, 8,192-token context) on NVIDIA RTX 3090 (24 GB) and RTX A6000 (48 GB) GPUs. Qwen2.5-3B and 7B, Llama-3.1-8B, and Mistral-7B-Instruct run on two RTX 3090s, Qwen2.5-14B on a single A6000, and Qwen2.5-32B on two A6000s; multi-GPU configurations use tensor parallelism. Every call uses greedy decoding (temperature 0), except AgentCF's memory-optimization stage, which keeps the temperature of 0.2 set in its official code. For every method, each response is parsed according to that method's output format. A response that cannot be parsed is counted as a miss, with no retry or fallback ordering, so all test users remain in the denominator. If a parsed response omits some candidates, the omitted candidates are placed below all ranked candidates in their prompt order. SASRec uses 64-dimensional embeddings and two self-attention blocks with two heads each, with a maximum sequence length of 20. It is trained with the cross-entropy loss over all items, and training stops early based on NDCG@10 on the last interaction of each user's training sequence. At inference, it ranks all items except the user's history items, and the top 20 form the candidates.

B.1.3 Baselines

All baselines rerank the same 20 SASRec candidates in the same order, except STAR as noted below, receive the $H = 5$ most recent history items, and use the same LLM, decoding, and parsing rules as FIRM (Appendix B.1.2). In the ranking prompts, items are represented by their titles as in FIRM, and additional item text is included only where a method prescribes it. All memories, statistics, and knowledge sources are built only from the training interactions.

Among the prompting-based methods, Zero-shot (Hou et al., 2024) ranks the candidates directly from the history, CoT (Wei et al., 2022) asks the LLM to reason step by step before ranking, Self-Refine (Madaan et al., 2023) has the LLM point out up to three candidates whose positions in its initial ranking look inconsistent with the history and then output a corrected ranking of all candidates, and Profile (Wang & Lim, 2023) follows the three-step prompting of the original work, which summarizes the user's preferences, selects representative history items, and then ranks the candidates.

Among the knowledge-augmented methods, DOKE (Yao et al., 2023) provides the LLM with verbalized knowledge graph paths that connect each candidate to the history items, where the graph links items co-consumed in the training interactions and items sharing a product category. STAR (Lee et al., 2024) adds, for each candidate, the number of training users who consumed it together with each history item. It ranks the candidates list-wise with a sliding window of four candidates and stride two, a variant reported in the original work, and starts from the order given by its retrieval score rather than the shuffled order, as the method prescribes ($\lambda = 0.7$, $a = 0.5$, $l = 3$). we compute the semantic similarity in this score with bge-large instead of the Gecko embeddings used originally. We use the list-wise variant because every other method, including FIRM, ranks the candidates list-wise, so that the comparison isolates the knowledge supplied to the ranker from the ranking protocol; the pair-wise variant, which the original work found strongest, would change the protocol itself rather than the knowledge. LLM-Rec (Lyu et al., 2024), originally proposed to enrich the inputs of trained recommenders, is adapted to LLM reranking by appending to each item title the description generated with its engagement-guided prompting, which summarizes the commonalities between the item's description and those of its three most co-consumed items. These neighbors replace the PPR-selected neighbors of the original work. ItemRAG (Kim et al., 2026) augments each item with summaries of 50 related items, sampled in proportion to co-consumption frequency from the co-consumed items of the item and of its five most similar items under bge-large embeddings.

Among the agent-based methods, AgentCF (Zhang et al., 2024) optimizes user and item memories from the discrepancy between predicted and actual choices, making one pass over every prefix of each user's training sequence with two update rounds per step and one negative item sampled in proportion to popularity; we optimize the memories of all users rather than a sampled subset. MemRec (Chen et al., 2026) builds user and item memories with one warm-up round that predicts each user's last training interaction, retrieving up to 16 collaborative neighbors within a 1,800-token budget and summarizing them into seven preference facets, as in the original work.[2] AgentDR (Yang et al., 2026) originally reranks the top items of an ensemble of three recommendation tools weighted by a per-

[2]MemRec updates the memories with the observed interaction after each recommendation. Since this interaction is the test item in our offline evaluation, we disable this update at test time so that no test interaction enters the memory.

user tool memory. Since our task reranks the shared candidate set, we omit the tool comparison, the ranking comparison, and the tool aggregation, and keep all other components. The LLM summarizes a user profile from the history and generates 20 substitute and 20 complement item titles from the ten most recent items. The intent weights for substitutes, complements, and general interest are learned as in the original work, with each user's last training interaction as the held-out target and a learning rate of $\gamma = 0.1$. The candidates are reranked by the LLM according to their similarity to each list and to the user profile, and the resulting ranks are fused according to these intent weights.

B.2 Experimental Results

B.2.1 Ablation Results

Tables 4 and 5 report the values plotted in Figure 2.

Table 4: Construction ablation. M and V denote the Manager and Verifier. Each variant rebuilds the relation memory and the corresponding user memories from the same Explorer proposals.

	Sports				Home				Grocery			
Variant	N@1	N@3	N@5	N@10	N@1	N@3	N@5	N@10	N@1	N@3	N@5	N@10
FIRM	0.1264	0.1952	0.2290	0.2917	0.1638	0.2356	0.2725	0.3374	0.1839	0.2748	0.3081	0.3641
w/o M	0.1228	0.1911	0.2239	0.2890	0.1619	0.2341	0.2705	0.3345	0.1800	0.2655	0.3004	0.3562
w/o V	0.1215	0.1854	0.2200	0.2837	0.1574	0.2286	0.2714	0.3373	0.1846	0.2695	0.3037	0.3602
w/o M and V	0.1196	0.1835	0.2175	0.2825	0.1556	0.2295	0.2662	0.3307	0.1819	0.2607	0.2919	0.3503

Table 5: Utilization ablation. The relation memory is kept fixed, and each block varies either the history–candidate relations or the user memory while keeping the other at the full setting. Coarse replaces each relation type with its substitution or complement label.

Variant	Sports				Home				Grocery			
	N@1	N@3	N@5	N@10	N@1	N@3	N@5	N@10	N@1	N@3	N@5	N@10
History–candidate relation inputs												
w/o Relations	0.1227	0.1898	0.2230	0.2859	0.1319	0.2156	0.2557	0.3192	0.1559	0.2408	0.2834	0.3420
Coarse relations	0.1240	0.1922	0.2261	0.2889	0.1425	0.2253	0.2632	0.3261	0.1819	0.2700	0.3031	0.3584
FIRM	0.1264	0.1952	0.2290	0.2917	0.1638	0.2356	0.2725	0.3374	0.1839	0.2748	0.3081	0.3641
User memory construction												
w/o User memory	0.1214	0.1840	0.2175	0.2828	0.1482	0.2205	0.2551	0.3207	0.1816	0.2563	0.2897	0.3484
Items-only memory	0.1256	0.1923	0.2247	0.2896	0.1484	0.2291	0.2642	0.3282	0.1765	0.2657	0.3013	0.3580
Coarse memory	0.1280	0.1922	0.2269	0.2886	0.1408	0.2244	0.2629	0.3259	0.1799	0.2713	0.3035	0.3592

B.3 Transferability

Tables 6 and 7 report the per-dataset results behind Figure 3 at all four cutoffs, for Qwen2.5 rerankers of different sizes and for rerankers from other families, respectively. FIRM uses the relation memory and the user memories built with Qwen2.5-7B-Instruct for every reranker.

B.4 Cost Analysis

Table 8 compares the offline and online costs of FIRM with those of Zero-shot and the agent-based baselines. Offline, FIRM issues 45,427, 112,260, and 38,853 calls on Sports, Home, and Grocery. The Explorer makes one call per selected pair and accounts for about 60% of them, and the Manager and the Verifier make the rest. The calls consume 42.6M, 133.8M, and 42.5M input tokens but only 4.4M, 10.5M, and 4.8M output tokens, about a tenth of the input, because every operation returns a short JSON object. The construction takes 3.4, 6.5, and 4.8 hours, which is 17% to 38% of the time of AgentCF and 25% to 67% of the time of MemRec. Among the agent-based baselines, only AgentDR takes less offline time. The cost of FIRM grows with the number of selected pairs, which

Table 6: Recommendation performance with different reranking LLMs.

Method	Sports				Home				Grocery			
	N@1	N@3	N@5	N@10	N@1	N@3	N@5	N@10	N@1	N@3	N@5	N@10
Qwen2.5-3B												
Zero-shot	0.0492	0.1127	0.1520	0.2291	0.0544	0.1263	0.1677	0.2417	0.0538	0.1292	0.1702	0.2473
AgentDR	0.0762	0.1350	0.1737	0.2518	0.0804	0.1431	0.1820	0.2590	0.0856	0.1418	0.1768	0.2531
AgentCF	0.0876	0.1438	0.1799	0.2508	0.0917	0.1567	0.1989	0.2729	0.1303	0.1960	0.2327	0.2987
MemRec	0.0971	0.1538	0.1882	0.2567	0.1156	0.1776	0.2155	0.2863	0.1221	0.1844	0.2222	0.2877
FIRM	0.0998	0.1609	0.1971	0.2651	0.1412	0.2183	0.2525	0.3185	0.1729	0.2508	0.2812	0.3400
Qwen2.5-7B												
Zero-shot	0.0735	0.1589	0.1965	0.2658	0.0766	0.1746	0.2131	0.2847	0.0724	0.1869	0.2262	0.2896
AgentDR	0.1040	0.1598	0.1955	0.2674	0.1093	0.1745	0.2133	0.2858	0.1222	0.1839	0.2212	0.2896
AgentCF	0.1162	0.1673	0.1991	0.2620	0.1430	0.2096	0.2492	0.3157	0.1792	0.2424	0.2723	0.3309
MemRec	0.1052	0.1796	0.2182	0.2815	0.1297	0.2132	0.2535	0.3198	0.1492	0.2271	0.2647	0.3256
FIRM	0.1264	0.1952	0.2290	0.2917	0.1638	0.2356	0.2725	0.3374	0.1839	0.2748	0.3081	0.3641
Qwen2.5-14B												
Zero-shot	0.0508	0.1533	0.1895	0.2525	0.0510	0.1919	0.2292	0.2927	0.0526	0.2263	0.2576	0.3131
AgentDR	0.1105	0.1659	0.2014	0.2707	0.1146	0.1819	0.2212	0.2941	0.1307	0.1928	0.2319	0.3002
AgentCF	0.1467	0.1999	0.2325	0.2931	0.1562	0.2208	0.2567	0.3242	0.1987	0.2624	0.2935	0.3488
MemRec	0.1398	0.2104	0.2387	0.2983	0.1488	0.2275	0.2664	0.3324	0.1769	0.2594	0.2961	0.3567
FIRM	0.1424	0.2090	0.2406	0.2973	0.1859	0.2600	0.2942	0.3565	0.2331	0.3084	0.3393	0.3902
Qwen2.5-32B												
Zero-shot	0.1227	0.1790	0.2147	0.2778	0.1477	0.2201	0.2533	0.3102	0.1863	0.2515	0.2779	0.3152
AgentDR	0.1373	0.1990	0.2347	0.2973	0.1445	0.2171	0.2571	0.3263	0.1571	0.2300	0.2664	0.3286
AgentCF	0.1557	0.2126	0.2441	0.3019	0.1729	0.2357	0.2715	0.3382	0.2067	0.2712	0.3033	0.3609
MemRec	0.1375	0.2085	0.2447	0.3026	0.1484	0.2349	0.2736	0.3418	0.1859	0.2688	0.3062	0.3687
FIRM	0.1541	0.2204	0.2550	0.3084	0.1883	0.2640	0.2984	0.3632	0.2459	0.3234	0.3573	0.4077

Table 7: Recommendation performance across LLM families.

Method	Sports				Home				Grocery			
	N@1	N@3	N@5	N@10	N@1	N@3	N@5	N@10	N@1	N@3	N@5	N@10
Qwen2.5-7B												
Zero-shot	0.0735	0.1589	0.1965	0.2658	0.0766	0.1746	0.2131	0.2847	0.0724	0.1869	0.2262	0.2896
AgentDR	0.1040	0.1598	0.1955	0.2674	0.1093	0.1745	0.2133	0.2858	0.1222	0.1839	0.2212	0.2896
AgentCF	0.1162	0.1673	0.1991	0.2620	0.1430	0.2096	0.2492	0.3157	0.1792	0.2424	0.2723	0.3309
MemRec	0.1052	0.1796	0.2182	0.2815	0.1297	0.2132	0.2535	0.3198	0.1492	0.2271	0.2647	0.3256
FIRM	0.1264	0.1952	0.2290	0.2917	0.1638	0.2356	0.2725	0.3374	0.1839	0.2748	0.3081	0.3641
Llama-3.1-8B												
Zero-shot	0.0436	0.1048	0.1490	0.2215	0.0458	0.1072	0.1589	0.2346	0.0458	0.1102	0.1750	0.2419
AgentDR	0.0498	0.1025	0.1424	0.2197	0.0539	0.1092	0.1497	0.2295	0.0523	0.1087	0.1485	0.2290
AgentCF	0.1000	0.1587	0.1913	0.2535	0.0844	0.1682	0.2066	0.2760	0.0872	0.1726	0.2114	0.2792
MemRec	0.1156	0.1809	0.2143	0.2728	0.1279	0.2007	0.2402	0.3184	0.1416	0.2249	0.2647	0.3291
FIRM	0.1121	0.1582	0.1945	0.2574	0.1453	0.2127	0.2478	0.3153	0.1845	0.2551	0.2882	0.3512
Mistral-7B-Instruct												
Zero-shot	0.0663	0.1299	0.1708	0.2457	0.0559	0.1204	0.1598	0.2369	0.0600	0.1336	0.1731	0.2439
AgentDR	0.1002	0.1601	0.1958	0.2673	0.0999	0.1682	0.2091	0.2817	0.1105	0.1747	0.2127	0.2860
AgentCF	0.1469	0.2010	0.2317	0.2918	0.1543	0.2222	0.2576	0.3241	0.1795	0.2464	0.2787	0.3397
MemRec	0.0917	0.1528	0.1899	0.2526	0.1211	0.1997	0.2372	0.3045	0.1309	0.2105	0.2490	0.3133
FIRM	0.1336	0.1951	0.2268	0.2895	0.1551	0.2282	0.2615	0.3271	0.1972	0.2726	0.3051	0.3590

the budget ρ and the threshold m control: the calls on Home are 2.5 times those on Sports, as is the number of selected pairs. The costs of the agent-based baselines grow with the number of users instead, at 5, about 15, and 3 calls per user for AgentDR, AgentCF, and MemRec. A relation memory is therefore built once per catalog, reused across reranking LLMs as Section 4.4 shows, and serves a new user without any offline call.

Table 8: Offline and online costs of FIRM and the baselines. Offline costs are totals for each dataset, including preparation of the baselines' user-specific memories used in evaluation. Online costs are averages per recommendation request; for FIRM, they include generating the user memory, which it builds at request time. In/Out denotes input/output token counts, summed over all calls; M and K denote millions and thousands, respectively.

		Offline			Online		
Dataset	Method	Calls	In/Out (M)	Time (h)	Calls	In/Out (K)	Latency (s)
Sports	Zero-shot	0	0 / 0	0	1	0.55 / 0.06	1.24
	AgentDR	54,365	16.0 / 10.7	1.9	5	3.25 / 1.48	12.0
	AgentCF	160,018	132.9 / 31.1	19.7	1	1.98 / 0.46	9.88
	MemRec	32,619	39.6 / 22.7	13.4	2	3.00 / 2.21	49.0
	FIRM	45,427	42.6 / 4.4	3.4	2	1.43 / 0.21	4.6
Home	Zero-shot	0	0 / 0	0	1	0.59 / 0.06	1.31
	AgentDR	62,785	20.2 / 13.7	2.9	5	3.69 / 1.63	13.1
	AgentCF	181,133	155.6 / 35.2	16.9	1	2.21 / 0.55	12.1
	MemRec	37,671	45.9 / 25.9	9.7	2	3.25 / 2.23	49.2
	FIRM	112,260	133.8 / 10.5	6.5	2	1.56 / 0.21	4.9
Grocery	Zero-shot	0	0 / 0	0	1	0.61 / 0.06	1.36
	AgentDR	36,695	11.3 / 7.1	1.6	5	3.45 / 1.67	14.7
	AgentCF	120,190	101.9 / 23.4	19.0	1	2.24 / 0.50	10.8
	MemRec	22,017	27.0 / 14.5	10.5	2	3.20 / 2.28	47.0
	FIRM	38,853	42.5 / 4.8	4.8	2	1.55 / 0.21	4.9

Online, FIRM issues two calls per request, one that writes the user memory and one that reranks, with 1.43K, 1.56K, and 1.55K input tokens and 0.21K output tokens, and answers in 4.6 to 4.9 seconds. This is under four times the latency of Zero-shot, whose single call holds only the history and the candidates, and lower than all three agent-based baselines: AgentDR and AgentCF take 9.9 to 14.7 seconds, and MemRec takes 47 to 49 seconds because its two calls generate about 2.2K output tokens. The user memory depends only on the user's training history and the fixed relation memory, so it can be written once per user and refreshed when the history changes, which leaves a single reranking call per request.

B.5 Relation Memory Statistics and Coverage

Table 9 summarizes the relation memory built on each dataset. Pair Selection yields 26,846, 68,148, and 24,203 pairs on Sports, Home, and Grocery. The collaborative and the text embeddings contribute similar numbers of pairs, co-consumption contributes far fewer under $m = 2$, and the three sources rarely coincide: their counts exceed $|\mathcal{P}|$ by at most 6.5%, so the collaborative and the textual neighbors of an item are largely different pairs. The Explorer labels 51.7%, 55.2%, and 42.6% of the selected pairs as None. Of the remaining proposals, 88.6%, 82.8%, and 86.8% end up assigned to a relation type, and the rest are unassigned by the Verifier, belong to a type that finds no parent, or are discarded after the additional rounds. Substitution accounts for 81.5% to 89.1% of the assigned pairs, so complementary relations are the minority on every dataset and the rarest on Grocery.

The Manager organizes the assigned pairs into 708, 1,141, and 329 relation types under 71, 74, and 26 higher-level types. The final tree has 196, 393, and 80 subtypes at depths 3 and 4, which hold 33.4%, 64.5%, and 21.1% of the assigned pairs, so the tree is deepest on Home. Merging is the main canonicalization step: it consolidated 3,751, 9,727, and 1,335 sibling types into 228, 505, and 94 groups, so a retained type absorbed more than ten equivalent types on average. Merges across groups are far rarer, at 93, 116, and 30, as are the subtypes folded into their parent after construction, at 34, 59, and 17. After construction, 109, 227, and 51 types still hold more than τ_{split} pairs. Splitting is not applied to types more than $d_{\max}$ levels below a higher-level type, rejects subtypes with fewer than m pairs or with an embedding within θ of the type or its ancestors, and is not repeated for a type whose previous split moved no pair, so a type keeps its pairs when their explanations support no further distinction. The largest type holds 7.4%, 4.7%, and 24.1% of the assigned pairs.

Table 9: Statistics of the relation memory. Type counts are after post-construction folding and the final refinement pass (Appendix A.5); Verifier counts are distinct pairs over all rounds. Depth counts the higher-level type as level 1.

	Sports	Home	Grocery
Selected pairs $\|\mathcal{P}\|$	26,846	68,148	24,203
from CF embeddings	12,026	31,724	11,393
from text embeddings	12,750	36,592	12,065
from co-consumption	3,663	1,243	2,311
Pairs labeled `None` (%)	51.7	55.2	42.6
Assigned pairs	11,490	25,297	12,063
% of $\|\mathcal{P}\|$ / % of non-`None`	42.8 / 88.6	37.1 / 82.8	49.8 / 86.8
`Sub` : `Com` (%)	82.4 : 17.6	81.5 : 18.5	89.1 : 10.9
Higher-level types (`Sub` / `Com`)	67 / 4	68 / 6	19 / 7
Relation types	708	1,141	329
at depth 2 / 3–4	512 / 196	748 / 393	249 / 80
pairs at depth 2 / 3 / 4 (%)	66.5 / 27.8 / 5.6	35.5 / 38.5 / 26.0	78.9 / 16.3 / 4.8
with $> \tau_{\text{split}}$ pairs	109	227	51
Pairs per type (mean)	16.2	22.2	36.7
Largest type (pairs / % of assigned)	846 / 7.4	1,198 / 4.7	2,902 / 24.1
Merged among siblings (types / groups)	3,751 / 228	9,727 / 505	1,335 / 94
Merged across groups	93	116	30
Folded after construction	34	59	17
Renamed by the final pass	11	4	5
Verifier: pairs moved	1,101	9,369	2,706
Verifier: pairs unassigned	1,072	4,738	1,289
Verifier: kept w/o revised proposal	676	2,651	1,136
Proposals discarded	70	85	47
Rounds	20	41	19

Table 10: NDCG@10 for test users split by whether their purchase history contains a stored relation (History) and whether the reranking prompt contains a relation line between a candidate and a history item (Candidate).

Dataset	History	Candidate	#	FIRM	Zero-shot	Δ vs. Zero-shot
Sports	✓	✓	2,151	0.2895	0.2679	+0.0216
	✓	×	791	0.2719	0.2402	+0.0317
	×	✓	4,565	0.2685	0.2253	+0.0432
	×	×	3,366	0.3292	0.3254	+0.0038
Home	✓	✓	2,087	0.5220	0.3205	+0.2015
	✓	×	1,870	0.2769	0.2328	+0.0441
	×	✓	3,248	0.3557	0.2896	+0.0661
	×	×	5,352	0.2754	0.2859	-0.0105
Grocery	✓	✓	2,417	0.4759	0.3223	+0.1536
	✓	×	1,218	0.2817	0.2509	+0.0308
	×	✓	2,058	0.3516	0.2813	+0.0703
	×	×	1,646	0.2765	0.2805	-0.0040

The Verifier moved 1,101, 9,369, and 2,706 pairs to another type, 9.6%, 37.0%, and 22.4% of the assigned pairs, and unassigned 1,072, 4,738, and 1,289 pairs judged to satisfy neither category. A further 676, 2,651, and 1,136 pairs received a misfit judgment without a usable revised proposal and kept their assignment, since the Verifier changes an assignment only when it names an alternative relation. The construction ran 20, 41, and 19 rounds, including the three consolidation rounds, and 70, 85, and 47 revised proposals remained unplaced after the additional rounds and were discarded.

Table 10 shows how the gain of FIRM depends on whether relations are retrieved for a user. A stored relation exists among the history items of 27.1%, 31.5%, and 49.5% of the test users, and the reranking prompt contains at least one relation line between a candidate and a history item for

61.8%, 42.5%, and 61.0% of them. Candidate-side coverage is lowest on Home, the largest and sparsest of the three catalogs. The gain concentrates on the users for whom relations are retrieved. When relations exist on both sides, NDCG@10 exceeds Zero-shot by 0.0216, 0.2015, and 0.1536. On Home and Grocery, this group holds 16.6% and 32.9% of the test users and accounts for 64% and 68% of the overall NDCG@10 gain of FIRM over Zero-shot. On Sports, most of the gain comes instead from the 42.0% of users who have relation lines in the prompt but no stored relation among their history items. This group contributes 70% of the gain on Sports, where it gains 0.0432 and 0.0661 and 0.0703 on Home and Grocery. Users with a stored relation only among their history items, whose user memory carries a relation-based sentence while the prompt lists no relation line, gain 0.0308 to 0.0441. For users with no retrieved relation, FIRM differs from Zero-shot only in the prompt format and in the items-only user memory, and its NDCG@10 stays within 0.011 of Zero-shot, slightly above on Sports and slightly below on Home and Grocery. The gain of FIRM therefore comes from the retrieved relations rather than from the prompt, and its reach is bounded by coverage, which the pair selection budget controls.

C Prompt Templates

We provide the prompts for the eight LLM operations in FIRM. Each call consists of one system message and one user message. System messages are shown verbatim except for line wrapping. The only inserted value in a system message is `<min_members>` in the Routing prompt (Appendix C.1.2), which is m in regular rounds and one in the consolidation rounds (Appendix A.2); the Splitting prompt (Appendix C.1.4) states the minimum subtype size as a fixed number, which equals m in our experiments. In the user templates, angle-bracketed fields denote inserted values, and ellipses denote repeated entries.

C.1 Multi-Agent Relation Construction

In these prompts, `substitution`, `complement`, and `none` denote `Sub`, `Com`, and `None`, respectively. An *axis* or *group* denotes a higher-level type in $\mathrm{ch}(b)$, and a *fine type* denotes a relation type below it.

C.1.1 Relation Exploration

The Explorer receives only the product records of the two items. The output fields `parent`, `name`, and `definition` correspond to b_p, r_p, and e_p, respectively.

Prompt 1: Relation Exploration

```
SYSTEM
You describe how two products relate.

Decide the parent relation. If it is substitution or complement, also
name the specific relation that holds between these two items: for
substitution, the criterion on which they are compared, and for
complement, the role they play when used together.

Do not worry about what other pairs were labelled. Describe THIS pair
as precisely as the records allow. A later step merges equivalent
descriptions, so being specific costs nothing and being vague loses
information.

The name must be short snake_case of at most three words, like
caffeine_content or package_size for substitution and flavor_addition or
brewing_enablement for complement. Never write a sentence as the name.

The definition must state what separates THIS pair from other pairs
under the same parent relation. Never restate or paraphrase the parent
relation definition.

PARENT RELATION DEFINITIONS
- substitution: A and B serve as alternatives for the same purpose,
each able to perform the same concrete product function, so a buyer
could select one instead of the other for that specific use. A shared
category, brand, shelf, occasion, or broad goal such as health,
snacking, cooking, or hydration is NOT substitution.
- complement: A and B have distinct roles in one concrete joint use,
with one enabling the use of the other or increasing its utility. Two
interchangeable products of the same kind are not complements. Sharing
a room, occasion, meal, or theme is not a complement.
- none: neither is supported by the supplied fields.

Return JSON only: {"pair_id":"...","parent":"substitution|complement|
none","relation":null or {"name":"snake_case","definition":"one
sentence"}}

USER
{
  "pair_id": <pair_id>,
  "item_A": {"title": <title_A>, "description": <description_A>,
             "categories": <category_entries_A>},
```

```
  "item_B": {"title": <title_B>, "description": <description_B>,
             "categories": <category_entries_B>}
}
```

C.1.2 Relation Routing

The prompt lists the existing children of root b in `current_axes` and the relation types without a parent in `new_types`.

Prompt 2: Relation Routing

```
SYSTEM
You place new fine relation types onto the axes of a taxonomy.
An axis is a higher-level type that groups semantically similar fine
types.
Assign each new type to an existing axis when one fits, otherwise
propose a new axis that groups it with other new types. Reusing an axis
is strongly preferred. A new axis needs at least <min_members> members
from this batch. Name an axis by what its members share, not by one
member.
If no existing axis fits a type and no new axis can be formed for it,
leave the type out of both lists. It will be placed later.
In assignments, axis_id must be one already shown. Anything new goes in
new_axes.
When the current axis list is empty, every grouping you make must
appear in new_axes, not in assignments.

Return JSON only: {"assignments":[{"type_id":"...","axis_id":"..."}],
"new_axes":[{"name":"snake_case","definition":"...",
"members":["type_id",...]}]}

USER
{
  "parent_relation": <substitution_or_complement>,
  "current_axes": [{"axis_id": <axis_id>, "name": <axis_name>,
                    "definition": <axis_definition>,
                    "n_members": <member_count>}, ...],
  "new_types": [{"type_id": <type_id>, "name": <type_name>,
                 "definition": <type_definition>}, ...]
}
```

C.1.3 Relation Merging

For a visited node r, the prompt lists $\mathrm{ch}(r)$ in `types` and the siblings $\mathrm{ch}(\mathrm{pa}(r))$ in `context_types`. When r is a root, `group` is the root category and `context_types` is `null`.

Prompt 3: Relation Merging

```
SYSTEM
You remove duplicate relation types among the children of one node in a
relation taxonomy.
You are given the node (group), the siblings of that node
(context_types, empty at the top level), and the children of that node
(types) with the number of item pairs under each, counting pairs under
their sub-types. Only the entries in "types" may be merged.
When context_types is given, use it as a granularity reference: never
merge two types whose difference is comparable to the differences among
context_types.
Compare the names and definitions of the types. Two types represent the
same relation only when their definitions apply to the same range of
item pairs. Keep two types separate when one applies to only a subset
of the pairs covered by the other, or when they represent different
comparison criteria or roles.
When two types represent the same relation, map the type with fewer
pairs onto the one with more.
```

```
Return JSON only:
{"merges":[{"type_id":"new_type_id","into":"existing_type_id"}]}

USER
{
  "group": {"name": <group_name>, "definition": <group_definition>},
  "level": <depth>,
  "context_types": <context_records_or_null>,
  "types": [{"type_id": <type_id>, "name": <type_name>,
             "definition": <type_definition>, "n_pairs": <pair_count>},
             ...]
}
```

C.1.4 Relation Splitting

For a node r with $|\mathcal{P}(r)| > \tau_{\text{split}}$, the prompt lists its children $\text{ch}(r)$ in `current_sub_types` and the explanations of the pairs in $\mathcal{P}(r)$ in `pair_descriptions`.

Prompt 4: Relation Splitting

```
SYSTEM
You split one relation type into sub-types. You are given the type and,
for each item pair directly assigned to it, the description that was
written for that specific pair. The type is right, but its pairs may
encompass several more specific relations. Group the pairs by the finer
distinctions you find in their descriptions. Name each sub-type by the
finer distinction that separates its pairs from the rest. You are also
shown the sub-types already made for this type. Do not propose a
distinction that one of them already makes. A sub-type needs at least 2
pairs. Never reuse the parent type's name. Assign each pair to at most
one sub-type. If no sub-type fits a pair, leave it on the parent type
and do not force it. Return JSON only:
{"sub_types":[{"name":"snake_case","definition":"one
sentence","members":["pair_id",...]}]}

USER
{
  "type": {"name": <type_name>, "definition": <type_definition>},
  "current_sub_types": [{"sub_type_id": <subtype_id>, "name":
  <subtype_name>,
                         "definition": <subtype_definition>,
                         "n_members": <member_count>}, ...],
  "pair_descriptions": [{"pair_id": <pair_id>, "description":
  <explanation>}, ...]
}
```

C.1.5 Assignment Verification

The prompt gives the assigned type $\mathcal{A}(p)$ as `fine_type`, its path as `path`, and up to n_{ver} pairs with both product records and e_p. For a misfit, `relation` gives r_p^* and e_p^*, and `correct_parent` gives b_p^* when `wrong_at` is `parent`. For other values of `wrong_at`, b_p^* is the current category.

Prompt 5: Assignment Verification

```
SYSTEM
You check whether item pairs really belong where they sit in a relation
taxonomy.

You are given one fine relation type, the path above it (parent relation
-> group -> the parent types it was split from), and several item pairs
currently assigned to the fine type. Each pair comes with both product
records - title, description and categories - and the stored
description that was written for that specific pair. Judge the pair from
both.
```

```
For each pair decide ok, or misfit. For a misfit, first say HOW FAR UP the
path is still correct:
  wrong_at=fine    the parent and axis hold, only this fine type is wrong
  wrong_at=axis    the parent holds but the axis is wrong
  wrong_at=parent  the pair is not substitution or complement at all,
  OR it sits under the wrong one of the two.

When wrong_at=parent, also say which parent is right in correct_parent:
substitution, complement, or none.
Judge the parent against these definitions - the same ones used when the
relation was first proposed. A pair that does not satisfy the definition
of the parent it sits under is wrong_at=parent, even when the type name
sounds right.
PARENT RELATION DEFINITIONS
- substitution: A and B serve as alternatives for the same purpose,
each able to perform the same concrete product function, so a buyer
could select one instead of the other for that specific use. A shared
category, brand, shelf, occasion, or broad goal such as health,
snacking, cooking, or hydration is NOT substitution.
- complement: A and B have distinct roles in one concrete joint use,
with one enabling the use of the other or increasing its utility. Two
interchangeable products of the same kind are not complements. Sharing
a room, occasion, meal, or theme is not a complement.
- none: neither is supported by the supplied fields.

Two cases are easy to miss. A type sitting under substitution whose
definition says the two items are used TOGETHER, with one providing what
the other lacks, is complement. A type sitting under complement whose
two items are interchangeable products of the same kind - different
brands, roasts, flavours or pack sizes of the same thing - is
substitution, however the type is named.

You are NOT shown any other types. Do not guess an identifier. For every
misfit, give the relation that does hold: a short snake_case name of at
most three words, naming the criterion on which the items are compared
for substitution or the role they play when used together for
complement, plus a one-sentence definition. The system will match that
name against the types already registered under the parent relation and
reuse one when it exists.
The definition must state what separates THIS pair from other pairs
under the same parent relation. Never restate or paraphrase the parent
relation definition.
The new relation must NOT reuse the name of the type you just rejected.
If the only name that fits is that same name, then the pair does belong
there - answer ok instead of misfit.

Being in the same category or brand is not enough to be ok.
Every pair_id supplied must appear exactly once, either in ok or in
misfit. Do not add a pair_id that was not supplied. The field
n_pairs_to_judge says how many there are.

Return JSON only: {"ok":["pair_id",...],"misfit":[{"pair_id":"...",
"wrong_at":"fine|axis|parent","correct_parent":"substitution|complement|
none","why":"...","relation":null or
{"name":"snake_case","definition":"one sentence"}}]}

USER
{
  "fine_type": {"id": <type_id>, "name": <type_name>,
                "definition": <type_definition>},
  "path": {"parent": <parent_relation>, "axis": <axis_record_or_null>,
           "ancestor_types": [{"id": <ancestor_id>, "name":
           <ancestor_name>,
                              "definition": <ancestor_definition>,
                              "n_pairs": <direct_pair_count>}, ...],
```

```
            "depth": <ancestor_count>},
  "n_pairs_to_judge": <batch_size>,
  "pairs": [{"pair_id": <pair_id>, "item_A": <product_A>,
             "item_B": <product_B>, "description": <explanation>}, ...]
}
```

C.1.6 Definition Refinement

The prompt gives node r as `current`, its children $\mathrm{ch}(r)$ as `sub_types`, and the explanations of $\mathcal{P}(r)$ as `pair_descriptions`. The parent $\mathrm{pa}(r)$ is given as `parent_type` when r is a subtype and as `group` otherwise. Explanations are sent in batches of at most 20, each starting from the name and definition returned for the previous batch, and `pairs_already_reflected` counts the explanations in earlier batches. Updates are applied regardless of the returned `confident` flag.

Prompt 6: Definition Refinement

```
SYSTEM
You repair a relation type whose definition no longer matches the pairs
under it. You are given the current name and definition, and the
descriptions that were written for the individual pairs now assigned to
the type. The current definition was written from earlier pairs of this
same type - the field pairs_already_reflected says how many. Keep what it
already covers, stay inside the parent relation and inside parent_type
when one is given, and cover the sub_types listed. Change it only as
far as the pairs listed now require. Name a substitution type by the
criterion on which its items are compared, and a complement type by the
role its items play when used together, never by a product family. Some
types have sub-types listed in sub_types. Reflect the relation in the
pairs assigned directly to this type, but keep the meaning wide enough
to still cover those sub-types. This does not mean widening the type to
admit sub-types that do not fit its parent relation or its current
scope. Write a correct snake_case name of at most three words and a
one-sentence definition. Return JSON only:
{"name":"snake_case","definition":"...","confident":true or false}

USER
{
  "current": {"name": <current_name>, "definition": <current_definition>,
              "parent": <parent_relation>, "n_pairs": <pair_count>,
              "pairs_already_reflected": <previously_reflected_count>},
  "parent_type": <parent_type_record_or_null>,
  "group": <group_record_or_null>,
  "sub_types": [{"sub_type_id": <child_id>, "name": <child_name>,
                 "definition": <child_definition>,
                 "n_members": <direct_pair_count>}, ...],
  "pair_descriptions": [<explanation>, ...]
}
```

C.2 User Memory Construction

In this prompt, a *link* denotes a relation type and an *aisle* denotes the category path of an item. The returned `memory` is used as m_u.

Prompt 7: User Memory Construction

```
SYSTEM
You write a short, reusable memory about one shopper. A ranker will
read it later, before it looks at this shopper's history and a list of
products to order.

You are given the products this shopper bought, in the order they
bought them, which of them they bought again, and every pair of their
own past products that is joined by a known link: the two products, the
link, and what it means.
```

```
Each product also carries the aisle path it sits in. Use the aisle
paths as given; do not invent one.

Write exactly two sentences.
 The first says what kind of product this shopper goes for: form and
 packaging, size, how processed or prepared it is, diet or ingredient
 claims, which part of the store it lives in.
 The second says what the links reveal about how this shopper chooses:
 which kinds of link recur, how the links change from earlier to later
 purchases, and what comparison or joint use lies behind their choices.
 If no links are given, say instead whether they buy the same products
 again or keep trying new ones.

Never name a brand or a specific product. Never quote a number. Never
mention a recommender, a model, a rank, or these instructions. Vary how
you open.

Reply with JSON and nothing else, in exactly this shape:
{"memory": "<your sentences>"}

USER
{
  "items_they_bought": [{"position": <index>, "title": <title>,
  "aisle": <aisle_path>}, ...],
  "bought_again": [<title>, ...],
  "distinct_items": <distinct_item_count>,
  "links_between_their_own_items": [
      {"pair": [<earlier_position>, <later_position>],
       "family": <parent_relation>, "link": <type_name>,
       "meaning": <definition_prefix>}, ...]
}
```

C.3 Relation-aware Reranking

In this template, braced fields denote inserted text. `relation_lines` contains one line per retrieved relation in the format shown below the prompt, or `(none stored for this shopper)` when no relation is retrieved.

Prompt 8: Relation-aware Reranking

```
SYSTEM
You rank 20 candidate products for one shopper. Each supplied relation
states how a candidate connects to one history item. For each
candidate: (1) judge its own fit to the history, (2) check whether a
supplied relation actually applies to this shopper's observed sequence,
(3) combine them. Repeated choices suggest replenishment; varied
choices suggest exploration. A relation that fits the shopper's pattern
counts more than one that does not. A candidate with no relation is not
thereby disliked. Do not invent preferences, brands, or compatibility.
A short note describes what this shopper has been doing. Treat it as
the observed pattern, not as an invented preference: weigh each
supplied relation by how well it matches that note.

USER
A shopper recently purchased (oldest to newest):
{history_titles}

Candidates:
{candidate_titles}

What this shopper has been doing:
{user_memory}

Relations between candidates and history items:
{relation_lines}
```

```
Return one JSON object with exactly the keys "1" through "20". Each
value is an integer score from 0 to 100 for that candidate: higher
means more likely to be bought next. Output the JSON object only - no
explanation, no reasoning, no extra keys.
```

Relation-line format for Prompt 8

```
- Candidate {c} - History {h} ({recency}): {parent} / {name}:
{definition}
```

Here `c` and `h` are positions in the numbered candidate and history lists, and `recency` is `most recent`, `second most recent`, or `older`. The `parent` field is `substitution` or `complement`, and `name` and `definition` are those of $\mathcal{A}(i_j^u, i_c)$.