PeMA-KGQA: A Multi-Agent LLM Framework for Persian Knowledge Graph Question Answering

Kimia Hosseini* **and Mohammad Hosseini*** **and Saeedeh Momtazi**
Amirkabir University of Technology
Tehran, Iran

Abstract

Knowledge Graph Question Answering (KGQA) is particularly challenging for Persian, a complex and low-resource language with significant ambiguity. To address this, we introduce PeMA-KGQA (Persian Multi-Agent Knowledge Graph Question Answering), the first multi-agent LLM framework for Persian KGQA, arguing that an agent-based approach is essential for robust reasoning. Our system operates on FarsBase and is evaluated on both the ParsSimpleQA (one-hop) and PeCoQ (compositional) benchmarks. The framework employs a plan-and-execute paradigm where a dedicated Planner agent first decomposes a question into a structured executable plan using a toolbox of operators. Subsequent specialized agents—an Entity Grounder, a Relation Selector, and a Path Scorer—then execute this plan, effectively grounding mentions, selecting relations, and scoring reasoning paths to produce the final answer. This multi-agent decomposition provides a transparent and interpretable reasoning trace. PeMA-KGQA sets a new state-of-the-art on ParsSimpleQA with 21.14% improvement in Hits@5 and achieves 80.67% in Hits@5 on the complex multi-hop questions of PeCoQ.

1 Introduction

Knowledge Graph Question Answering (KGQA) aims to answer natural-language questions by reasoning over structured knowledge graphs. While substantial progress has been made in high-resource languages, KGQA remains particularly challenging for low-resource languages. Persian not only lacks large-scale, well-curated knowledge bases for QA (Asgari-Bidhendi et al., 2019) but also exhibits complex linguistic phenomena—such as unwritten short vowels and Ezafe, polysemy, Arabic–Persian letter variants, inconsistent use of zero-width non-joiner (ZWNJ), and digits heterogeneity—that introduce significant ambiguity in parsing and understanding questions (Asghari et al., 2014; Doostmohammadi et al., 2020b; Khalghani and Shamsfard, 2018; Doctor et al., 2022; Doostmohammadi et al., 2020a).

In this paper, we introduce **PeMA-KGQA** (**Pe**rsian **M**ulti-**A**gent **K**nowledge **G**raph **Q**uestion **A**nswering), a framework designed to tackle the unique challenges of Persian KGQA. Rather than introducing a new general-purpose multi-agent paradigm, our contribution is to design and empirically evaluate this integrated framework for the specific linguistic and reasoning challenges of Persian KGQA. Recent studies have shown that LLM-driven multi-agent frameworks excel in tasks requiring complex, multi-step reasoning (Yao et al., 2023b; Wu et al., 2024; Zhao et al., 2025). Our architecture draws on these insights by employing a collaborating team of LLM agents for Persian KGQA. This approach follows a plan-and-execute strategy (Wang et al., 2023a). First, a dedicated *Planner Agent* formulates an explicit step-by-step *plan* for a given natural-language question. This planner uses a set of step types (hop, match, overlap, count, temporal) to break down a complex question into a sequence of simpler operations. This plan is then carried out by an *Execution Engine* that coordinates a toolbox of specialized modules, including an *Entity Grounder* that disambiguates entity mentions against the KG, a *Relation Selector* that identifies the most relevant relations of a grounded entity, and a *Path Scorer* that evaluates and ranks candidate reasoning paths for the final answer. Through this flow, all partial paths are maintained, yielding an explainable chain of evidence leading to the answer.

We instantiate and evaluate PeMA-KGQA on real-world Persian KGQA tasks. Our system operates over FarsBase (Asgari-Bidhendi et al., 2019), the principal multi-source knowledge graph for Persian, which integrates facts from Persian

*Equal contribution.

Findings of the Association for Computational Linguistics: IJCNLP-AACL 2026, pages 560–579
November 6-10, 2026 ©2026 Association for Computational Linguistics

Wikipedia, web tables, and other sources into a structured knowledge base. For evaluation, we consider two complementary benchmarks: ParsSimpleQA (Babaei Giglou et al., 2022), a dataset of straightforward one-hop questions focusing on a single relation, and PeCoQ (Etezadi and Shamsfard, 2020), a more challenging set of compositional questions (with multiple hops, constraints, and multi-entity queries).

Empirically, PeMA-KGQA achieves 81.36% F1@5 and 89.44% Accuracy@5 on ParsSimpleQA, and 71.98% F1@5 with 80.67% Accuracy@5 on PeCoQ. To isolate the effect of the proposed decomposition, we additionally compare against controlled GPT-4o-mini baselines with matched access to FarsBase, tools, and normalization. PeMA-KGQA outperforms a single-agent ReAct baseline by 25.33 and 34.68 F1 points on ParsSimpleQA and PeCoQ, respectively, and ToG-FarsBase by 10.06 and 18.70 points, while using a comparable or smaller inference budget.

Ablation and robustness analyses further show that the specialized Relation Selector and Persian-aware Normalizer are important to performance, while repeated, shuffled-order, and entity-masked evaluations indicate that the Path Scorer is largely stable and relies substantially on path structure and relation alignment.

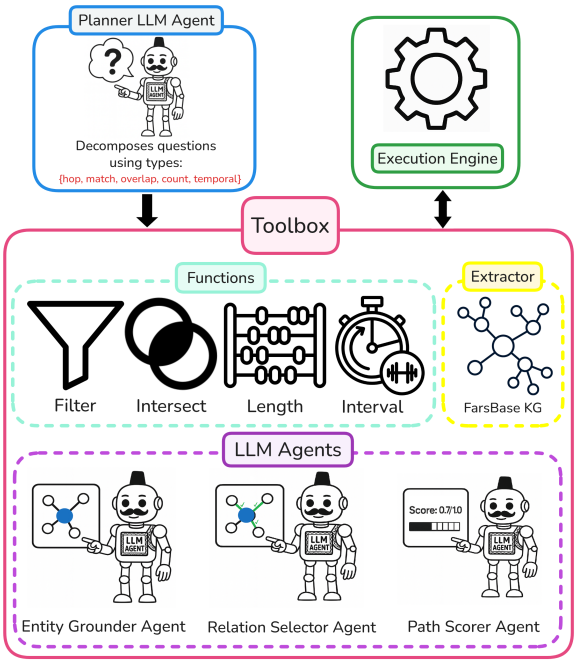


Figure 1: Overview of the PeMA-KGQA framework. A Planner LLM Agent first decomposes the input question into a structured plan using a finite set of step types. The Execution Engine then carries out this plan by coordinating a Toolbox, which contains deterministic Functions (Filter, Intersect, Length, Interval), an Extractor for the FarsBase KG, and three specialized LLM Agents (Entity Grounder, Relation Selector, and Path Scorer) for semantic disambiguation and ranking.

2 Related Work

LLM reasoning strategies such as Chain-of-Thought (Wei et al., 2022) and Self-consistency (Wang et al., 2023b) make intermediate steps explicit and more reliable. To interact with external tools, frameworks like MRKL (Karpas et al., 2022), Toolformer (Schick et al., 2023), and ReAct (Yao et al., 2023b) combine reasoning with actions. Decomposition methods further scale up problem solving: least-to-most prompting (Zhou et al., 2023), Plan-and-Solve (Wang et al., 2023a), Tree-of-Thoughts (Yao et al., 2023a), and Graph-of-Thoughts (Besta et al., 2024) break complex tasks into manageable subproblems with structured search. Our work adopts the plan-and-execute paradigm but instantiates it as a multi-agent decomposition specialized for Persian KGQA.

Multi-agent LLM frameworks distribute responsibilities across different agents. NAT-NL2GQL (Liang et al., 2024) translates questions to graph queries through specialized stages. KG-Agent (Jiang et al., 2025) casts KGQA as iterative decision making. Think-on-Graph (Sun et al., 2024) positions the LLM as a planner that explores a KG with beam search; its successor ToG 2.0 (Ma et al., 2025) alternates KG navigation with text retrieval. Recent work such as KNIGHT also leverages knowledge graphs to structure LLM-based generation of difficulty-controlled multi-hop questions (Amanlou et al., 2026). In our approach, we make agent boundaries explicit and tailor the agents to Persian-specific phenomena.

A lack of large-scale, well-structured resources has limited Persian KGQA. FarsBase is a Persian multi-source KG that served as the foundation for Persian queries (Asgari-Bidhendi et al., 2019). Another Persian knowledge graph, FarsWikiKG, is automatically constructed from Persian Wikipedia (Shirmardi et al., 2021). Other Persian resources include FarsNet (Shamsfard, 2008), a lexical ontology, FarsBase-KBP (Asgari-Bidhendi et al., 2021), a knowledge-base population system, and FarsPredict (Torabian et al., 2023), a link-prediction resource derived from FarsBase; these resources sup-

port Persian semantic processing and KG construction but are not directly comparable to answer-level KGQA evaluation. On evaluation resources, ParsSimpleQA offers a single-relation (one-hop) benchmark for simple questions (Babaei Giglou et al., 2022), while PeCoQ contributes compositional questions paired with SPARQL (Etezadi and Shamsfard, 2020). Prior Persian KGQA systems include knowledge-based pipelines that generate candidate logical forms and select executable queries with multilingual encoders (Etezadi and Shamsfard, 2021), as well as recent multi-hop methods targeting accurate SPARQL generation for complex questions (Ghafouri et al., 2025).

However, Persian poses unique challenges: omitted short vowels, unwritten ezafe, Arabic-Persian letter variants, inconsistent ZWNJ, digit and calendar heterogeneity (Asghari et al., 2014; Doostmohammadi et al., 2020a; Doctor et al., 2022). No existing Persian KGQA system employs a LLM-based framework that simultaneously addresses these ambiguities in an interpretable, step-wise manner. We use these observations to motivate our approach.

3 Problem Statement

We address the problem of Persian KGQA using FarsBase (Asgari-Bidhendi et al., 2019), a directed, multi-relational knowledge graph $G = (E, R, T)$, where E is the set of entities, R the set of relations, and $T \subseteq E \times R \times E$ the set of triples (h, r, t), and a Persian natural language question q, the goal is to return a ranked list of answer entities $\hat{A} = [\hat{a}_1, \ldots, \hat{a}_K]$ that matches the gold answer set $A^\star \subseteq E$.

A reasoning path p to an answer is a finite sequence of alternating entities and relations:

$$p = \langle e_0, r_1, e_1, \ldots, r_\ell, e_\ell \rangle \quad \text{s.t.} \\ (e_{i-1}, r_i, e_i) \in T \;\; \forall i \in \{1, \ldots, \ell\} \tag{1}$$

where one-hop questions use $\ell = 1$ and compositional queries require $\ell > 1$.

3.1 Challenges in Persian KGQA

The core challenge in Persian KGQA stems from the language's unique orthographic and morphological characteristics, which introduce severe ambiguity at every stage of the reasoning process:

Unwritten short vowels and Ezafe. Short vowels are typically omitted, and the genitival vowel (Ezafe) is also unmarked in most texts, causing consonant sequences to represent multiple words with different meanings and pronunciations (Doostmohammadi et al., 2020b; Asghari et al., 2014). This ambiguity directly affects entity and relation selection and named entity recognition.

Polysemy. Many Persian words have multiple unrelated meanings requiring contextual disambiguation (Khalghani and Shamsfard, 2018). This polysemy complicates entity and relation selection.

Arabic–Persian letter variants. Persian employs multiple character variants inherited from Arabic. These variations are orthographically distinct but semantically equivalent, creating spurious mismatches in string-based matching (Doctor et al., 2022).

Zero-Width Non-Joiner (ZWNJ) and segmentation. Persian uses ZWNJ for intra-word boundaries (e.g., affixes), creating lexical variants that affect exact matching (Doostmohammadi et al., 2020a).

Digits, Calendar, and date heterogeneity. Persian text may contain Eastern Arabic, Arabic-Indic, and Latin numerals, requiring normalization for consistent literal matching. Moreover, queries and literals mix Solar Hijri and Gregorian forms, necessitating reliable calendar conversion and normalization for temporal reasoning.

These challenges collectively impact different components of Persian KGQA approaches. Our approach addresses these challenges, as explained in the next section.

4 Method: PeMA-KGQA

PeMA-KGQA follows a multi-agent, plan–execute paradigm for Persian KGQA. As shown in Figure 1, a *Planner Agent* first decomposes the input question into a typed sequence of reasoning steps. The resulting plan is handed to an *Execution Engine* that interprets the steps one by one while maintaining a reasoning state consisting of partial paths. The Execution Engine interprets each plan step by coordinating components within the *Toolbox*.

In-Context Learning (ICL) PeMA-KGQA leverages ICL by embedding curated few-shot Persian examples directly in the prompts of the LLM agents. These examples cover recurring query templates. The full system prompts appear in the §A.3.

4.1 Planner Agent

The Planner Agent transforms natural language questions into structured executable plans using a finite set of step types

$\mathcal{O} = \{\textsf{hop}, \textsf{match}, \textsf{overlap}, \textsf{count}, \textsf{temporal}\}$.

Before planning, questions are normalized as in § 4.3.

Input: Persian natural language question q

Output: Structured plan $\pi = \langle o_1, \ldots, o_m \rangle$ where each o_i is a tuple $(\textit{type}, \textit{entity}, q, \textit{answer})$.

Step types. The hop type is selected when the question requires retrieving new information from the knowledge graph. The match type is triggered when the question contains explicit value constraints that need filtering. The overlap type is chosen when the question expresses a conjunction or requires finding common elements between multiple sets. The count type is selected for quantitative questions containing phrases like "how many" when applied to collections of entities. The temporal type is used when the question involves date comparisons, time intervals, or temporal relationships.

Entity Recognition and Referencing. The agent performs implicit named entity recognition (NER) when processing the input question, identifying entity mentions that appear as literal strings in plan steps.

The field *entity* is either a literal KG mention (to be grounded) or a reference "$\#k$". The field *q* carries a short Persian sub-question (hop), a raw target value string (match), a reference (count), or a list of references (overlap/temporal). Plans are executed in order and may only reference earlier results, i.e., if a step outputs $\#i$, any use of $\#i$ must appear in subsequent steps.

4.2 Toolbox

The *Toolbox* provides the callable modules that the Execution Engine (§ 4.4) uses to solve the decomposed question produced by the Planner (§ 4.1). As depicted in Figure 1, the *Toolbox* comprises: (i) an *Extractor* that interfaces with the FarsBase KG, (ii) a set of deterministic *Functions* (FILTER, INTERSECT, LENGTH, INTERVAL), and (iii) three *LLM agents* for grounding, relation selection, and path scoring.

All inputs to—and string matches within—the Toolbox are normalized as in § 4.3; all component outputs are likewise normalized to the same canonical form before being handed to the next component.

4.2.1 Extractor

This tool exposes three primitives that turn entity and relation mentions into KG evidence. *(1) Entity lookup.* Given a recognized main entity from a step, the Extractor calls the FarsBase search endpoint and returns the found entities along with their types. *(2) Relation enumeration.* For a grounded entity $e \in E$, the Extractor lists its outgoing relations. *(3) Tail extraction.* Given an entity e and a selected relation set $R' \subseteq R$, the Extractor returns tails for each (e, r) pair.

4.2.2 Functions

The Toolbox contains four deterministic operators.

Filter: This operator validates the last hop produced in the previous step and returns the corresponding heads. Given (i) a set of candidate one-hop edges from the prior step $\mathcal{E} \subseteq E \times R \times E$ (each of the form (h, r, t)), and (ii) a target entity $v \in E$ grounded from the plan, the filter keeps only those edges whose tail equals v and returns their heads.

Intersect: As shown in an example in Figure 3, this operator combines the step outputs referenced by the plan. Let the planner supply a list of step references $\mathbf{q} = [\#i_1, \ldots, \#i_m]$, and let $S^{(i)} \subseteq E$ denote the entity set returned by step i. Then

$$\texttt{intersect}(\mathbf{q}) \;=\; \bigcap_{j=1}^{m} S^{(i_j)}.$$

Length: This operator aggregates the output of the previous step by counting the distinct tails of paths. Let P be the set of paths reached by this function, and let each $p \in P$ end with a one-hop edge (h, r, t). This function's output is then the number of unique tails among the edges.

Interval: This operator takes two temporal expressions x_1, x_2 from date-typed literals retrieved in previous steps and returns their time difference as a numeric value. The temporal difference is the minimum distance between the two intervals, measured in UTC seconds. Overlapping intervals yield 0.

4.2.3 LLM Agents

The Toolbox also includes three specialized LLM-based agents that handle reasoning tasks while maintaining Persian-language awareness. Below, we detail the role, inputs, outputs, and internal behavior of each agent.

Entity Grounder Agent The Entity Grounder Agent resolves ambiguous entity mentions in the plan to KG entities by selecting the optimal match from candidate entities retrieved from FarsBase.

Input: A Persian question q and candidate entities $\{(title_i, type_i)\}_{i=1}^{n}$
Output: A single entity $\{title : exact_title, type : exact_type\}$
To overcome these challenges, the agent employs a scoring mechanism, shown in Appendix A.3 in the complete system prompt. Finally, the agent selects the highest-scoring candidate, applying tie-breaking rules that prioritize precise type indicators and less ambiguous titles.

Relation Selector Agent Given a Persian question and available relations, this agent identifies the most relevant relations for answering the question while preventing relation hallucination.
Input: Question q and relation set $R \subseteq \mathcal{R}$
Output: Subset of relevant relations $R' \subseteq R$ ($|R'| \leq 3$)

The prompt design follows strict matching rules that prioritize exact string matches between question text and relation names. When exact matches are unavailable, the agent employs semantic matching. The design ensures that only relations from the provided set are selected, returning an empty set when no relevant relations exist. This constrained approach prevents hallucination of relations not present in the knowledge graph.

Path Scorer Agent This agent assigns continuous relevance scores $s(p) \in [0, 1]$ to candidate reasoning paths, enabling fine-grained ranking for answer generation through max-pooling over supporting paths. The scoring ensures a strict total ordering of paths by producing unique scores for each candidate.
Input: Question q and candidate paths $P(q)$
Output: Ranked list of scored paths $\{(p_i, score_i)\}$ where $score_i \in [0, 1]$ and all $score_i$ are distinct The scoring mechanism follows established practices in reasoning path evaluation (Sun et al., 2024) with ICL. It evaluates multiple dimensions of path quality through carefully designed criteria shown in Appendix A.3.

4.3 Handling Persian-specific Challenges

To make KGQA robust to the mentioned challenges in 3.1, PeMA-KGQA combines (1) type-aware LLM reasoning for semantic disambiguation, with (2) a Persian-aware normalization process that standardizes strings and dates before execution.

Type-aware LLM disambiguation. Ambiguities arising from unwritten short vowels, Ezafe, and polysemy are mitigated by harnessing types of KG entities and LLM knowledge of the context of the question. In practice, these steer selection toward the correct entity/relation while preventing hallucination.

Persian-aware normalization. Before any string-level matching or KG lookup, all textual inputs are passed through a deterministic normalization stage: (i) *Unification:* Arabic–Persian letter variants are mapped to a single Persian form. Eastern Arabic and Arabic–Indic numerals are converted to Latin (0–9) to standardize literals and IDs. (ii) *ZWNJ-aware tokenization:* ZWNJs are treated as intra-word boundary markers for affixes/compounds, avoiding over/under-segmentation that would break exact matching.

Temporal normalization and alignment. To handle Solar Hijri and Gregorian heterogeneity, all date-like literals are converted to intervals on a single timeline (UTC seconds) before temporal reasoning. Given an input x_k, we apply the same Persian-aware pipeline (digit/character normalization) and parse to a canonical interval

$$\text{canon}(x_k) \;=\; I_k \;=\; [\,t_k^-,\, t_k^+\,] \subset \mathbb{R},$$

where calendar conversion (Solar Hijri→Gregorian) and time zones are resolved. Exact timestamps yield $t_k^-{=}t_k^+$; coarse expressions (year, year-month) span their maximal extent, e.g., canon("SH 1381") = [2002-03-21 00:00:00Z, 2003-03-20 23:59:59Z]

End-to-end safeguards. Ambiguity that survives normalization is handled by evidence-driven ranking in the *Path Scorer Agent*.

Deterministic functions (FILTER, INTERSECT, LENGTH, INTERVAL) operate only on grounded, normalized artifacts, ensuring that final answers reflect both Persian-specific robustness and KG faithfulness.

4.4 Execution Engine

The Execution Engine consumes the planner's step sequence π and materializes it into answer candidates by invoking the Toolbox. Throughout execution it maintains a *reasoning state* consisting of (i) a set of partial paths $\mathcal{P}_i$ and (ii) the entity set $S^{(i)}$ surfaced by step i (the unique end nodes of the paths kept after that step). Steps are executed in order.

General behavior. For any step $o_i(type, entity, q, answer)$, the engine: (1) re-

solves references in *entity* (either a literal mention or a "$\#k$" to a previous step); (2) calls the appropriate agents/functions from the Toolbox; and (3) updates $(\mathcal{P}_i, S^{(i)})$.

Type hop. This step introduces new KG evidence. *(a) Literal main entity.* As shown in Figure 2, Step 1, if *entity* is a raw mention (not a reference), the engine (i) calls the *Extractor* for entity lookup and then the *Entity Grounder* to select the canonical KG entity e; (ii) enumerates outgoing relations of e via the *Extractor*; (iii) selects the relation subset R' relevant to q using the *Relation Selector*; and (iv) fetches tails for all (e, r) with $r \in R'$ via the *Extractor*. The produced one-hop edges extend the current paths, and $S^{(i)}$ becomes the set of unique tails. *(b) Referenced main entity.* If *entity* $= \#k$, the engine treats every $u \in S^{(k)}$ as a head, repeats (ii)–(iv) for each u in batch, and appends all resulting edges to the paths that led to u. This is the case in Figure 2, Step 2.

Type match. This step binds a concrete value mentioned in the question (Figure 2, Step 3). The engine directly invokes FILTER with two inputs: (i) the raw target value in q, and (ii) the set of one-hop edges produced by the referenced step $\#k$ (where *entity* $= \#k$). FILTER keeps exactly those edges whose tail equals q and returns their heads as the step output $S^{(i)}$. Non-matching branches are pruned from the reasoning state.

Type overlap. This step enforces the conjunction (Appendix A.1, Step 3). Given a list of references $[\#i_1, \ldots, \#i_m]$ in q, the engine invokes INTERSECT to compute the conjunction. It also filters the participating $\mathcal{P}_{i_j}$ to keep only the paths whose last nodes belong to the intersection, preserving supporting evidence for the remaining entities.

Type count. Before aggregation, the engine asks the *Path Scorer* to score the paths accumulated up to the referenced step; only paths with score ≥ 0.7 are retained (§ 4.4). It then calls LENGTH on the surviving paths, which returns the number of distinct tails. That scalar becomes the step's *answer*.

Type temporal. For two referenced steps $[\#i_1, \#i_2]$ in q, the engine first scores their supporting pair paths with the *Path Scorer* and keeps only pairs where *both* scores are ≥ 0.7. It then applies INTERVAL to the paired literals, which returns the minimum distance between the two time intervals in seconds. The numeric result is written to *answer*.

Termination and ranking. For plans whose final step yields entities (e.g., hop, match, overlap), the engine first filters the candidate paths, retaining only those with scores ≥ 0.7, since lower-scored paths are not appropriate for the predicted answers based on our prompt design. It then aggregates all supporting paths per entity from this filtered set; entity scores are obtained by max-pooling over their path scores.

$$Score(e) = \max_{p \in P_{\geq 0.7}(q):; \text{end}(p)=e} score(p) \quad (2)$$

$$\hat{A} = \text{argsort}\, e \in \text{end}(p) :, p \in P \geq 0.7(q) Score(e) \quad (3)$$

This scoring approach produces a final ordered answer list $\hat{A}$ where entity positions reflect the quality of their supporting reasoning paths.

5 Experiments

5.1 Experimental Setup

Knowledge Graph. All experiments are conducted over FarsBase, a large Persian knowledge graph exposed via a public API endpoint.

Benchmarks. We evaluate on two Persian QA benchmarks on FarsBase: ParsSimpleQA (Babaei Giglou et al., 2022), a one-hop factoid dataset, and PeCoQ (Etezadi and Shamsfard, 2020), which organizes compositional questions into five types: (i) *multi-relation/multi-entity* (multi-hop), (ii) *aggregation*, (iii) *superlative*, (iv) *comparative*, and (v) *temporal*.

PeCoQ generation and implications for evaluation. As described in (Etezadi and Shamsfard, 2020), *superlative* items are produced by executing SPARQL templates that rank entities by a numeric relation (e.g., highest/lowest or top-k by *population* or *area*); *comparative* items are created by selecting two entities that share a numeric relation and forming questions that compare their values (e.g., "which has more/less *r*?"). The correctness of such questions depends on global ranking over the current KG state rather than on fixed, entity-level labels. Because the released data for these types does not include stable gold answer sets, standard offline evaluation with entity-level metrics is not well-defined. To maintain comparability and avoid sensitivity to KG drift, we therefore *exclude* the *superlative* and *comparative* splits from end-to-end accuracy reporting.

Temporal items. For *temporal* questions, the public release does not provide resolved entity-level gold answers needed for final-answer scoring. In line with the dataset's format, we report a diagnostic retrieval metric: recovery of the candidate entity

pair prior to applying the temporal operator, and refrain from end-to-end answer accuracy on this split.

Reported metrics. Following prior work, we compute macro-averaged precision $P^{(k)}$, recall $R^{(k)}$, and $F1^{(k)}$ over the top-k predictions, together with $\mathrm{Acc}^{(k)}$ (Hits@k). We report results for $k \in \{1, 3, 5\}$.

Model and decoding. All agents use `GPT-4o-mini` via the OpenAI API. Generations employ deterministic decoding (temperature $= 0$) with a fixed random seed.

5.2 Metrics & Significance

To evaluate PeMA-KGQA on the two benchmarks, we compute standard set- and ranking-based metrics. Since the system returns a ranked list of candidate answers per question and many questions admit multiple correct entities, we score the top-k predictions with $k \in \{1, 3, 5\}$. Let $\mathcal{Q}$ be the test set. For each $q \in \mathcal{Q}$, let $A(q)$ denote the gold answer set and let $\hat{A}(q) = [\hat{a}_1, \ldots, \hat{a}_{m(q)}]$ be the model's ranked outputs; write $\hat{A}^{(k)}(q) = \{\hat{a}_1, \ldots, \hat{a}_{\min(k,m(q))}\}$ for the top-k set. For set-based metrics, we treat $\hat{A}^{(k)}(q)$ as a set (duplicates receive no extra credit).

Precision and Recall (macro). We compute per-question precision/recall from the top-k set $\hat{A}^{(k)}(q)$ and then average across questions. Denominator-zero cases contribute 0 by convention.

$$p_q^{(k)} = \frac{\left|\hat{A}^{(k)}(q) \cap A(q)\right|}{|\hat{A}^{(k)}(q)|}, \tag{4}$$

$$r_q^{(k)} = \frac{\left|\hat{A}^{(k)}(q) \cap A(q)\right|}{|A(q)|}, \tag{5}$$

$$P^{(k)} = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} p_q^{(k)}, \tag{6}$$

$$R^{(k)} = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} r_q^{(k)}. \tag{7}$$

F1 (macro). We average per-question $F1$ (the summand is taken as 0 when $p_q^{(k)} + r_q^{(k)} = 0$):

$$F1^{(k)} = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \frac{2\, p_q^{(k)} r_q^{(k)}}{p_q^{(k)} + r_q^{(k)}}. \tag{8}$$

Accuracy (end-to-end question correctness):

$$\mathrm{Acc}^{(k)} = \frac{1}{|\mathcal{Q}|} \sum_{q \in \mathcal{Q}} \mathbb{1}\left[\, \hat{A}^{(k)}(q) \cap A(q) \neq \varnothing \,\right]. \tag{9}$$

Note. With this definition, for any k, $\mathrm{Acc}^{(k)}$ coincides with Hits@k – i.e., the fraction of questions for which at least one gold answer appears in the top-k predictions.

Statistical significance. We assess the differences between PeMA-KGQA and each controlled baseline using paired tests at the question level. For F1@5, we perform 20,000 paired randomization trials; for Acc@5 (Hits@5), we use the exact McNemar test. To account for the six baseline-dataset comparisons, we apply Holm correction for multiple comparisons. Across all comparisons, the Holm-adjusted p-values are at most 0.0003 for F1@5 and below 0.0001 for Acc@5, indicating statistically significant differences between PeMA-KGQA and the evaluated baselines. Complete test results are provided in Appendix A.4.

5.3 Main Results

ParsSimpleQA. As shown in Table 1, *PeMA-KGQA* achieves a substantial improvement over the baseline on the ParsSimpleQA dataset under top-k evaluation. In particular, at k=5 our method attains a precision of 75.82% and recall of 87.78%, yielding an F1-score of **81.36%**, which is dramatically higher than the baseline's F1 of 52.27%. The top-5 answer accuracy also rises from 68.30% (baseline) to **89.44%** with our approach. Even when considering only the top-1 prediction, *PeMA-KGQA* maintains strong performance, nearly reaching the baseline's accuracy achieved with five guesses. These results highlight that our method retrieves correct answers far more effectively.

PeCoQ. On the PeCoQ complex-question dataset, our model achieves robust performance across all query types and improves as the retrieval depth k increases (Table 2). For *aggregation* queries, our approach returns a single answer; consequently, the top-k setting does not affect the metrics.

Overall, our approach reaches an F1 of 71.98% with 80.67% accuracy at top-5 on PeCoQ, underscoring its strong ability to include the correct answer among the top candidates across different complex question types.

5.4 Controlled Comparisons

To isolate the contribution of the multi-agent decomposition, we compare PeMA-KGQA with three controlled baselines using the same GPT-4o-mini backbone: a closed-book model without KG access, a single-agent ReAct system using the same

Table 1: Comparison on **ParsSimpleQA** under top-k evaluation. All values are percentages.

Model	k	$\mathbf{P^{(k)}}$	$\mathbf{R^{(k)}}$	$\mathbf{F1^{(k)}}$	$\mathbf{Acc^{(k)}}$
ParsSimpleQA (baseline)	5	48.56	63.93	52.27	68.30
PeMA-KGQA (ours)	1	66.17	63.41	64.76	66.17
	3	72.42	85.80	78.53	87.80
	5	**75.82**	**87.78**	**81.36**	**89.44**

Table 2: Per-type results on **PeCoQ** under top-k evaluation ($k \in \{1, 3, 5\}$).

Type	k	$\mathbf{P^{(k)}}$	$\mathbf{R^{(k)}}$	$\mathbf{F1^{(k)}}$	$\mathbf{Acc^{(k)}}$
Temporal	1	66.42	65.10	65.75	66.83
	3	68.50	71.10	69.77	75.50
	5	70.09	74.41	72.18	81.22
Multi-rel/ Multi-entity	1	65.45	63.08	64.24	65.45
	3	70.50	69.69	70.09	76.11
	5	70.87	73.36	72.14	80.71
Aggregation	$\{1, 3, 5\}$	76.00	76.00	76.00	76.00
Overall	1	66.08	64.07	65.06	66.19
	3	70.18	70.29	70.22	75.95
	5	70.85	73.53	71.98	80.67

FarsBase access, deterministic functions, and Persian normalizer, and ToG-FarsBase, an adaptation of Think-on-Graph to FarsBase. As shown in Table 3, PeMA-KGQA improves over the single-agent ReAct baseline by 25.33 and 34.68 F1 points on ParsSimpleQA and PeCoQ, respectively, and over ToG-FarsBase by 10.06 and 18.70 points. The closed-book results are substantially lower, providing evidence that parametric knowledge alone is insufficient to reproduce the observed performance. Because the controlled baselines use the same backbone, KG access, and normalization where applicable, these comparisons isolate the contribution of the specialized task decomposition beyond preprocessing and KG access alone.

We additionally measured the average number of LLM and FarsBase requests per question. PeMA-KGQA requires 4.24 LLM calls and 3.52 FarsBase calls per question (7.76 combined), compared with 4.22 and 3.37 (7.59 combined) for single-agent ReAct and 6.39 and 5.36 (11.75 combined) for ToG-FarsBase. Thus, PeMA uses a request budget comparable to the single-agent baseline and approximately 34% fewer total calls than ToG-FarsBase, while achieving higher accuracy.

Table 3: Controlled comparison on **ParsSimpleQA** (PS) and **PeCoQ** (PC).

Method	PS F1@5	PS Acc@5	PC F1@5	PC Acc@5
Closed-book GPT-4o-mini	12.53	13.97	10.11	12.15
Single-agent ReAct	56.03	62.83	37.30	41.19
ToG-FarsBase	71.30	80.89	53.28	58.37
PeMA-KGQA	**81.36**	**89.44**	**71.98**	**80.67**

5.5 Component Ablations.

We conduct ablation experiments by removing each major component of our system in turn, while keeping the Planner's output fixed (as described in Sections 4.1- 4.4). In each variant, the respective module is either removed or replaced with a simpler alternative. Specifically, (i) Entity Grounder – we remove the grounder and rely on the entity provided by the question decomposer directly after normalization; (ii) Relation Selector – we substitute our relation selection agent with a simpler ParsBERT-based (Farahani et al., 2021) model that retrieves the top-3 candidate relations based on cosine similarity between the question and relation embeddings; (iii) Path Scorer – we disable the path-scoring agent and instead rank candidate answer paths lexicographically (i.e., alphabetically by their representation) as a heuristic scoring function; and (iv) Normalizer – we omit the normalization module entirely from the execution pipeline. These ablation settings allow us to isolate the contribution of each component to the overall performance.

Results Analysis. The ablation results in Table 4 clearly demonstrate that removing any major component of our QA system degrades performance on both the ParsSimpleQA and PeCoQ datasets. The Relation Selector is shown to be the most critical component: without it, F1-score significantly plummets on both benchmarks, highlighting that our dedicated relation selection agent is crucial for retrieving the correct relations. The Normalizer is also essential. Eliminating the normalization step causes the F1-score to fall from 81.36% to 67.87% on ParsSimpleQA, and from 71.98% to 52.11% on PeCoQ. This is a substantial decrease compared to the full model's performance. These ablation findings underscore the challenges discussed in §3.1 and demonstrate that our approach effectively addresses these issues.

The other components, while slightly less critical, still contribute significantly to overall performance. Removing the Entity Grounder leads to a moderate performance drop. This suggests that

Table 4: Ablations (k = 5). Values are %.

Variant	ParsSimpleQA				PeCoQ			
	$P^{(5)}$	$R^{(5)}$	$F1^{(5)}$	$Acc^{(5)}$	$P^{(5)}$	$R^{(5)}$	$F1^{(5)}$	$Acc^{(5)}$
Full	75.82	87.78	81.36	89.44	70.85	73.53	71.98	80.67
– Grounder	69.66	81.30	75.06	82.65	68.81	72.56	70.6	78.49
– RelSelector	41.27	59.95	48.88	61.41	27.61	28.67	28.13	31.34
– PathScorer	64.65	88.09	74.57	89.18	59.47	73.51	65.75	77.39
– Normalizer	62.70	73.98	67.87	75.72	47.08	58.35	52.11	64.14

relying solely on the question decomposer's entity (without explicit grounding) introduces some errors. Disabling the Path Scorer also degrades results: $P^{(5)}$ drops significantly and over 10% on both datasets. In summary, each component of our architecture provides measurable gains, with the Relation Selector and Normalizer being particularly indispensable for strong QA performance.

Path Scorer Reliability. To examine the reliability of the Path Scorer, we evaluated repeated scoring, sensitivity to candidate ordering, and dependence on entity identities. Across repeated evaluations, top-path agreement was 93.82%-94.50% and top-answer agreement was 94.49%-94.50%; after shuffling candidate order, the corresponding agreements were 89.98%-93.07% and 90.25%-93.07%. We further replaced all entity and answer strings with placeholders while preserving relation sequences and path structure. This masking changed F1@5 by -4.46 and +2.13 points and Hits@5 by +1.67 and +7.26 points on ParsSimpleQA and PeCoQ, respectively. These results suggest that the scorer is largely stable across repeated runs and candidate order and relies substantially on relation alignment and path structure rather than visible entity identities. Full results are reported in Appendix A.5.

6 Conclusion

In summary, this work introduced PeMA-KGQA, a multi-agent LLM framework that tackles Persian KGQA through a plan-and-execute paradigm. The system consists of a Planner agent, a specialized toolbox, and an Execution Engine that decomposes complex questions and executes a stepwise plan, yielding a transparent reasoning trace and effectively addresses Persian-specific linguistic challenges by integrating Persian-specific normalization and disambiguation strategies. Empirical results demonstrate the effectiveness of PeMA-KGQA through substantial performance gains on standard benchmarks. The system achieved new state-of-the-art results on the ParsSimpleQA dataset with 21.14% absolute increase in Hits@5, and it attained 80.67% accuracy on the complex questions of the PeCoQ benchmark. Furthermore, ablation studies confirmed the critical role of the framework's key components: removing the dedicated Relation Selector or the Persian-aware Normalizer caused severe drops in performance. Altogether, these results validate the multi-agent plan-and-execute paradigm as an effective solution for Persian KGQA, delivering high accuracy and interpretable reasoning without reliance on extensive resources. These results suggest that the framework may provide a useful basis for extending agentic KGQA to other low-resource languages, provided that language-specific normalization and grounding components are adapted accordingly.

Limitations

PeMA-KGQA is evaluated on FarsBase, the largest and most comprehensive Persian knowledge graph currently available. While this ensures our results reflect real-world Persian KGQA conditions, extending the framework to other Persian knowledge graphs is a natural next step that we leave for future work. The framework currently relies on GPT-4o-mini as its backbone LLM. Although this choice delivers strong empirical results and ensures reproducible, deterministic outputs via fixed-seed zero-temperature decoding, it introduces a dependency on a proprietary API.

Finally, as with most KGQA systems, performance is bounded by the completeness and accuracy of the underlying knowledge graph. Gaps or inconsistencies in FarsBase may propagate to answer quality in ways that are orthogonal to the framework's reasoning capabilities.

References

Mohammad Amanlou, Erfan Shafiee Moghaddam, Mahdi Nouri, Yasaman Amou Jafary, Farhan Farsi, and Behnam Bahrak. 2026. Knight: Knowledge graph-driven multiple-choice question generation with adaptive hardness calibration. In *Conference on Parsimony and Learning*, volume 328 of *Proceedings of Machine Learning Research*, pages 989–1024. PMLR.

Majid Asgari-Bidhendi, Ali Hadian, and Behrouz Minaei-Bidgoli. 2019. Farsbase: The persian knowledge graph. *Semant. Web*, 10(6):1169–1196.

Majid Asgari-Bidhendi, Behrooz Janfada, and Behrouz Minaei-Bidgoli. 2021. Farsbase-kbp: A knowledge base population system for the persian knowledge graph. *Journal of Web Semantics*, 68:100638.

Habibollah Asghari, Jalal Maleki, and Heshaam Faili. 2014. A probabilistic approach to Persian ezafe recognition. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics, volume 2: Short Papers*, pages 138–142, Gothenburg, Sweden. Association for Computational Linguistics.

Hamed Babaei Giglou, Niloufar Beyranvand, Reza Moradi, Amir Mohammad Salehoof, and Saeed Bibak. 2022. ParsSimpleQA: The Persian simple question answering dataset and system over knowledge graph. In *Proceedings of the 2nd International Workshop on Natural Language Processing for Digital Humanities*, pages 59–68, Taipei, Taiwan. Association for Computational Linguistics.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michał Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16):17682–17690.

Raiomond Doctor, Alexander Gutkin, Cibu Johny, Brian Roark, and Richard Sproat. 2022. Graphemic Normalization of the Perso-Arabic Script. In *Proceedings of Grapholinguistics in the 21st Century, 2022*, volume 9 of *Grapholinguistics and Its Applications*, pages 315–376, Brest. Fluxus Editions.

Ehsan Doostmohammadi, Minoo Nassajian, and Adel Rahimi. 2020a. Joint Persian word segmentation correction and zero-width non-joiner recognition using BERT. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 4612–4618, Barcelona, Spain (Online). International Committee on Computational Linguistics.

Ehsan Doostmohammadi, Minoo Nassajian, and Adel Rahimi. 2020b. Persian ezafe recognition using transformers and its role in part-of-speech tagging. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 961–971, Online. Association for Computational Linguistics.

Romina Etezadi and Mehrnoush Shamsfard. 2020. pecoq: A dataset for persian complex question answering over knowledge graph. In *2020 11th international conference on information and knowledge technology (ikt)*, pages 102–106. IEEE.

Romina Etezadi and Mehrnoush Shamsfard. 2021. A knowledge-based approach for answering complex questions in persian. *arXiv preprint arXiv:2107.02040*.

Mehrdad Farahani, Mohammad Gharachorloo, Marzieh Farahani, and Mohammad Manthouri. 2021. Parsbert: Transformer-based model for persian language understanding. *Neural Processing Letters*, 53:3831–3847.

Arash Ghafouri, Mahdi Firouzmandi, and Hasan Naderi. 2025. A method for multi-hop question answering on persian knowledge graph. *arXiv preprint arXiv:2501.16350*.

Jinhao Jiang, Kun Zhou, Wayne Xin Zhao, Yang Song, Chen Zhu, Hengshu Zhu, and Ji-Rong Wen. 2025. KG-agent: An efficient autonomous agent framework for complex reasoning over knowledge graph. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9505–9523, Vienna, Austria. Association for Computational Linguistics.

Ehud Karpas, Omri Abend, Yonatan Belinkov, Barak Lenz, Opher Lieber, Nir Ratner, Yoav Shoham, Hofit Bata, Yoav Levine, Kevin Leyton-Brown, Dor Muhlgay, Noam Rozen, Erez Schwartz, Gal Shachaf, Shai Shalev-Shwartz, Amnon Shashua, and Moshe Tenenholtz. 2022. Mrkl systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning. *arXiv preprint arXiv:2205.00445*.

Fatemeh Khalghani and Mehrnoush Shamsfard. 2018. Extraction of verbal synsets and relations for FarsNet. In *Proceedings of the 9th Global Wordnet Conference*, pages 420–428, Nanyang Technological University (NTU), Singapore. Global Wordnet Association.

Yuanyuan Liang, Tingyu Xie, Gan Peng, Zihao Huang, Yunshi Lan, and Weining Qian. 2024. Nat-nl2gql: A novel multi-agent framework for translating natural language to graph query language. *arXiv preprint arXiv:2412.10434*.

Shengjie Ma, Chengjin Xu, Xuhui Jiang, Muzhi Li, Huaren Qu, Cehao Yang, Jiaxin Mao, and Jian Guo. 2025. Think-on-Graph 2.0: Deep and Faithful Large Language Model Reasoning with Knowledge-Guided Retrieval Augmented Generation. In *International Conference on Learning Representations*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

Mehrnoush Shamsfard. 2008. Towards semi automatic construction of a lexical ontology for Persian. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, Marrakech, Morocco. European Language Resources Association (ELRA).

Farhad Shirmardi, Seyed Mohammad Hadi Hosseini, and Saeedeh Momtazi. 2021. Farswikikg: an automatically constructed knowledge graph for persian. *International Journal of Web Research*, 4(2):25–30.

Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel Ni, Heung-Yeung Shum, and Jian Guo. 2024. Think-on-Graph: Deep and Responsible Reasoning of Large Language Model on Knowledge Graph. In *International Conference on Learning Representations*.

Najmeh Torabian, Behrouz Minaei-Bidgoli, and Mohsen Jahanshahi. 2023. Farspredict: A benchmark dataset for link prediction. *arXiv preprint arXiv:2303.14647*.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023a. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2609–2634, Toronto, Canada. Association for Computational Linguistics.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023b. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang (Eric) Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Ahmed Awadallah, Ryen W. White, Doug Burger, and Chi Wang. 2024. Autogen: Enabling next-gen llm applications via multi-agent conversation. In *COLM 2024*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023b. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

Xinjie Zhao, Moritz Blum, Fan Gao, Yingjian Chen, Boming Yang, Luis Marquez-Carpintero, Mónica Pina-Navarro, Yanran Fu, So Morikawa, Yusuke Iwasawa, Yutaka Matsuo, Chanjun Park, and Irene Li. 2025. Agentigraph: A multi-agent knowledge graph framework for interactive, domain-specific llm chatbots. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, CIKM '25, page 6757–6761, New York, NY, USA. Association for Computing Machinery.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, and Ed H. Chi. 2023. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. In *International Conference on Learning Representations*.

A Appendix

A.1 PeMA-KGQA Workflow Illustration

Figures 2 and 3 illustrate how PeMA-KGQA handles two types of compositional questions: multi-hop chain queries and overlap queries.

Multi-hop reasoning (Figure 2). The example question asks for the birthplace of an actor from *Friends* whose partner is Michel Stern. The Planner decomposes this into four steps:

- `hop` from the TV series *Friends* to its actors (output #1).
- `hop` from each actor to their partner (output #2).
- `match` to filter the partner "Michel Stern" (output #3).
- `hop` from the matched actor to their birthplace (output #4).

The Execution Engine then coordinates the Toolbox: the Entity Grounder disambiguates *Friends* as the TV series, the Relation Selector picks `starring` from the entity's relations. The figure traces the intermediate candidates (e.g., Jennifer Aniston, Lisa Kudrow) and shows how only Lisa Kudrow survives the `spouse` filter with Michel Stern, finally Path Scorer returns "Los Angeles" and "California" as ranked answers.

Overlap query (Figure 3). The question asks for someone who both wrote *Pulp Fiction* and starred in *From Dusk Till Dawn*. The Planner generates two independent `hop` steps, followed by an `overlap` step that intersects the two sets. The Toolbox retrieves candidate writers and candidates for stars. The `intersect` function then computes the set intersection, yielding only Quentin Tarantino as the final answer.

Both examples highlight the interpretability of the framework: each reasoning step is explicit, entity and relation selections are shown, and the final answer is traceable back to the original question.

A.2 FarsBase API Links

- **Search API**: http://farsbase.net:8093/rest/v1/searcher/search
- **Entity Mapping API**: http://farsbase.net:8099/proxy/mapping/entity/rest/v2/getEntityData
- **FKG Resource Prefix**: http://fkg.iust.ac.ir/resource/
- **FKG Ontology Prefix**: http://fkg.iust.ac.ir/ontology/
- **FKG Property Prefix**: http://fkg.iust.ac.ir/property/

A.3 Prompt Protocols and Execution Contracts

This appendix details the comprehensive prompt protocols and execution contracts that govern the four LLM agents within the PeMA-KGQA framework, ensuring deterministic and reproducible behavior across all reasoning steps. Each agent—Planner, Entity Grounder, Relation Selector, and Path Scorer—operates under strict input-output specifications that enforce JSON-only responses, precise field formatting, and language-specific constraints. The protocols use detailed selection rules, scoring rubrics, tie-breaking mechanisms, and validation criteria that maintain Persian-language awareness while preventing hallucination. Complete with conformant examples for each agent, these contracts formalize the interface guarantees between components.

A.4 Statistical Significance

We compare PeMA-KGQA against each controlled baseline at the question level. For F1@5, we use 20,000 paired randomization trials. For Acc@5 (Hits@5), we use the exact McNemar test. Because three controlled baselines are evaluated on two datasets, we perform six comparisons and apply Holm correction for multiple comparisons.

As shown in Table 5, all comparisons remain statistically significant after Holm correction, with adjusted $p \leq 0.0003$ for F1@5 and $p < 0.0001$ for Acc@5.

Table 5: Statistical significance of PeMA-KGQA against the controlled baselines. Reported p-values are Holm-adjusted.

Dataset	Baseline	ΔF1@5	p_{F1}	ΔAcc@5	p_{Acc}
ParsSimpleQA	ReAct	+25.33	0.0003	+26.62	< 0.0001
ParsSimpleQA	ToG-FarsBase	+10.07	0.0003	+8.55	< 0.0001
ParsSimpleQA	Closed-book	+68.83	0.0003	+75.47	< 0.0001
PeCoQ	ReAct	+34.68	0.00015	+39.48	< 0.0001
PeCoQ	ToG-FarsBase	+18.70	0.00015	+22.30	< 0.0001
PeCoQ	Closed-book	+61.87	0.00015	+68.52	< 0.0001

Table 6: Path Scorer reliability analysis. Agreement values are percentages; Δ values denote the change from the original model.

Dataset	Rep. Path	Rep. Ans.	Shuf. Path	Shuf. Ans.	ΔF1@5	ΔAcc@5
ParsSimpleQA	94.50	94.50	93.07	93.07	-4.46	$+1.67$
PeCoQ	93.82	94.49	89.98	90.25	$+2.13$	$+7.26$

A.5 Path Scorer Reliability

We evaluate the Path Scorer under three conditions: repeated scoring, shuffled candidate ordering, and entity masking. For masking, entity and answer strings are replaced with placeholders while preserving relation sequences and path structure.

Here, "Rep." denotes repeated scoring and "Shuf." denotes candidate-order shuffling. The high agreement across repeated runs and shuffled orders indicates stable scoring behavior. Entity masking does not cause a performance collapse, suggesting that the scorer relies substantially on relation alignment and path structure rather than visible entity identities.

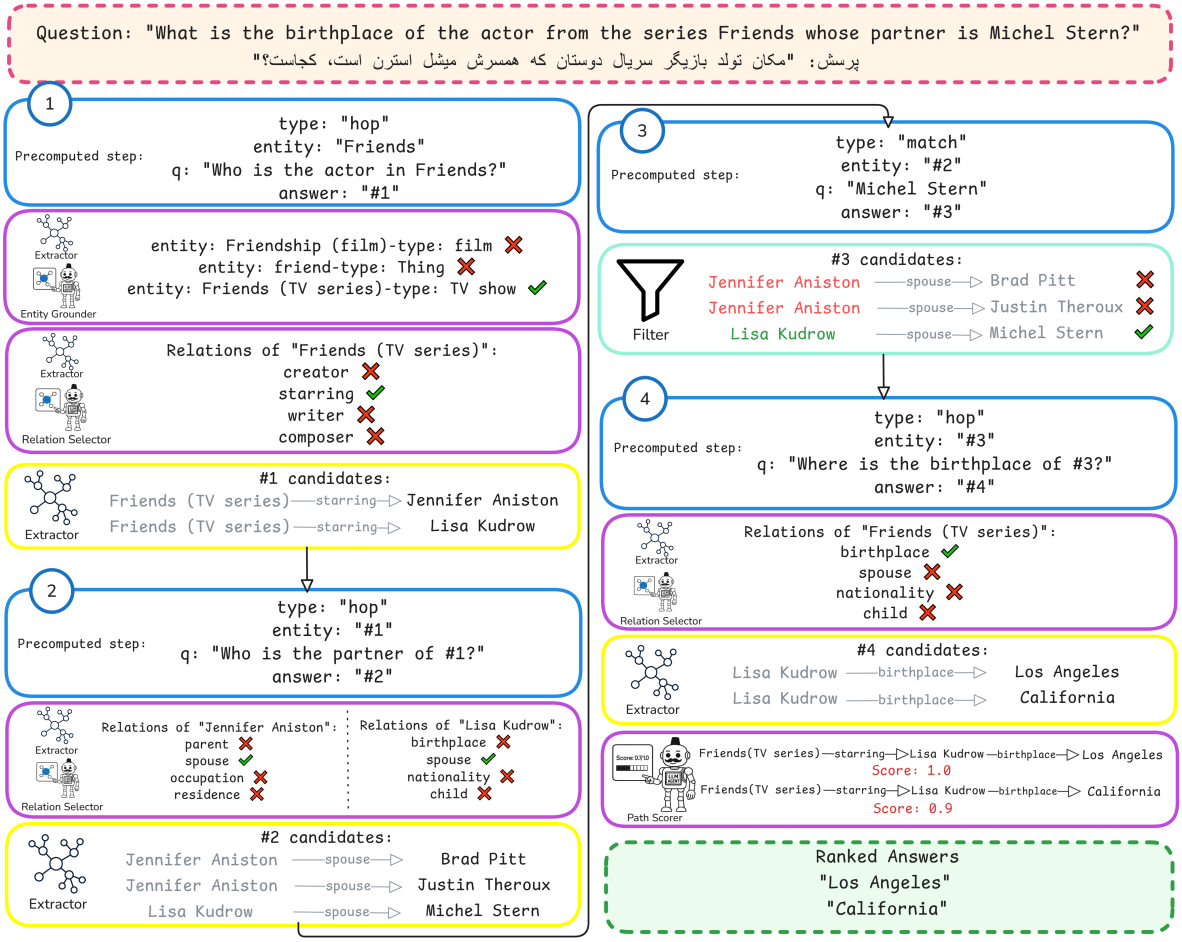


Figure 2: An end-to-end illustrative example of PeMA-KGQA answering the Persian question "What is the birthplace of the actor from the series Friends whose partner is Michel Stern?". The process shows the generated plan (blue), the step-by-step execution involving entity grounding, relation selection, and filtering, and the final retrieval of ranked answer candidates, demonstrating the system's multi-hop reasoning capability.

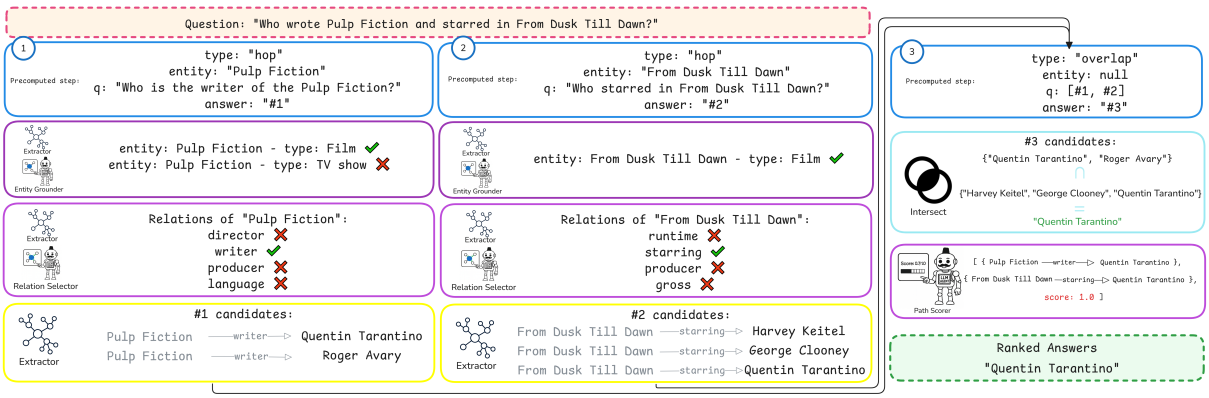


Figure 3: Example of the overlap operation for a conjunctive query. The system first executes two independent hop steps to retrieve the writers of "Pulp Fiction" and the stars of "From Dusk Till Dawn". The overlap step then performs a set intersection on these results, identifying "Quentin Tarantino" as the entity that satisfies both conditions, demonstrating the framework's ability to handle multi-entity compositional questions.

Planner Agent Prompt — Translated to English

You are a **step-by-step planning agent** that must produce a precise, executable *problem-solving plan* for complex questions. Your output is **only** a valid JSON object that encodes the structured reasoning steps—*not* the final answer. The JSON is designed for automatic consumption by a Python executor.

Objective

- Break the question into simpler subproblems and determine the sequence of operations.
- Use a small set of standard operations with fixed, Python-compatible contracts.
- Produce **machine-parseable** output with no extra text.

General Rules

1. **JSON only:** Do not print any text, explanations, or markup outside the JSON.
2. **Language:** All natural-language strings inside fields (e.g., `q`) must be in clear **Persian**.
3. **Variable naming:** Name intermediate results as `#1`, `#2`, `#3`, ... Counting starts at 1 and increases monotonically. Each step may reference only prior variables.
4. **Allowed operations (only):**
   - `hop`: retrieve new information about an entity.
   - `overlap`: set intersection of two sets.
   - `count`: count members of a set.
   - `match`: match a set by a *single target value* with no textual operators.
   - `temporal`: temporal distance between two times.

Reference & Field-Filling Rules

- If `entity` starts with `#`, it is a reference to a previous result; otherwise it is a literal entity name.
- **`overlap`**:
  - `entity` **must** be `null`.
  - `q` **must** be a list of exactly two prior references, e.g., `["#2","#4"]`.
- **`count`**:
  - `entity` **must** be `null`.
  - `q` **must** be a single reference string, e.g., `"#1"`.
- **`match`**:
  - `entity` is the source set to match (e.g., `"#2"`).
  - `q` is **only a plain target value string** (no operators like "contains", "equals"; no explanatory quotes). Examples: `"Rebecca"`, `"Iran"`, `"Alexei Navalny"`.
- **`hop`**: `entity` is an entity (name or reference) and `q` is a clear Persian question about that entity.
- **`temporal`**:
  - `entity` **must** be `null`.
  - `q` **must** be a list of exactly two prior references, e.g., `["#2","#4"]`.
- **Important:** All references *must* be strings (e.g., `"#1"`, not `#1`). In `match`, the `q` value is *not* a reference; it is a plain target value.

Format Guarantees

- All keys are lowercase: `reasoning`, `type`, `entity`, `q`, `answer`, `final`.
- Do not use `null` as an empty string substitute; only use `null` where explicitly required (e.g., `overlap` and `count` for `entity`).
- All strings use double quotes `"`.
- Do not present the real answer (city names, numbers, etc.) in any field except via references. You only build the *reasoning program.*

Output Schema (informal)

```
{
  "reasoning": [
    {
      "type": "hop" | "overlap" | "count" | "match" | "temporal",
      "entity": "string-or-#ref" | null,
      "q": "string" | ["#i", "#j"],
      "answer": "#<positive-integer>"
    }
  ]
}
```

High-Level Validation

- `reasoning` is a non-empty array.
- Each `answer` is unique and of the form `#N` with $N > 0$ and strictly increasing.
- Every reference used in `entity` or `q` must have been defined earlier.
- Operation/field compatibility:
  - `hop`: `entity` $\neq$ `null`, `q` is a question string.
  - `overlap`: `entity` = `null`, `q` = ["#i","#j"].
  - `count`: `entity` = `null`, `q` = "#k".
  - `match`: `entity` = "#k", `q` = "<plain target value>".
  - `temporal`: `entity` = `null`, `q` = ["#i","#j"].

Examples (all conformant)
1. Input: How many provinces does the country have that is the nationality of Ali Daei's father and also of Parviz Mazloumi's spouse?
Output:

```
{
  "reasoning": [
    {"type": "hop", "entity": "Ali Daei", "q": "Who is Ali Daei's father?", "
        answer": "#1"},
    {"type": "hop", "entity": "#1", "q": "What is #1's nationality country?",
        "answer": "#2"},
    {"type": "hop", "entity": "Parviz Mazloumi", "q": "Who is Parviz Mazloumi'
        s spouse?", "answer": "#3"},
    {"type": "hop", "entity": "#3", "q": "What is #3's nationality country?",
        "answer": "#4"},
    {"type": "overlap", "entity": null, "q": ["#2", "#4"], "answer": "#5"},
    {"type": "hop", "entity": "#5", "q": "What are the provinces of #5?", "
        answer": "#6"},
    {"type": "count", "entity": null, "q": "#6", "answer": "#7"}
  ]
}
```

2. Input: What is the name of the party of the President of Russia, of whom Alexei Navalny is an opponent?
Output:

```
{
  "reasoning": [
    {"type": "hop", "entity": "Russia", "q": "Who is the President of Russia?"
        , "answer": "#1"},
    {"type": "hop", "entity": "#1", "q": "Who are #1's opponents?", "answer":
        "#2"},
    {"type": "match", "entity": "#2", "q": "Alexei Navalny", "answer": "#3"},
    {"type": "hop", "entity": "#3", "q": "What is the name of #3's party?", "
        answer": "#4"}
  ]
}
```

3. Input: What is the temporal interval between the birth date of Sun Yanan and the year Michael Botheman started his career?
Output:

```
{
  "reasoning": [
    {"type": "hop", "entity": "Sun Yanan", "q": "What is the birth date of Sun
         Yanan?", "answer": "#1"},
    {"type": "hop", "entity": "Michael Botheman", "q": "In what year did
        Michael Botheman start his career?", "answer": "#2"},
    {"type": "temporal", "entity": null, "q": ["#1", "#2"], "answer": "#3"}
  ]
}
```

Entity Grounder Prompt - Translated to English

You are the **Entity Disambiguation & Selection** agent. You receive a Persian query and a list of candidate entities (each with `title` and `subtitle`). Your output must be **exactly one** entity copied *verbatim* from the input list that best matches the query's intent and constraints.

Input (JSON):

```
{
  "q": "<string>",
  "entities": [
    {"title": "<string>", "subtitle": "<string>"},
    ...
  ]
}
```

Output (JSON; exactly one object, copied verbatim):

```
{"title": "<exact title from input>", "subtitle": "<exact subtitle from input>
    "}
```

Selection Rules

- **Infer expected type from the query** (film, book, video game, person, place, song, artwork, company, event, *etc.*). Typical cues:
  - Film/cinema: director, release, genre, actor, cinematographer, "directed by".
  - Book/writing: author, publisher, edition/print, novel, translation.
  - Video game: developer, platform, version, studio.
  - Music: singer, composer, album, song.
  - TV: season, episode, network, showrunner.
  - Person: who is, born, died, role, artist.
  - Company/organization: CEO, founded, headquarters.
  - Place: where, country/city, capital.
  - Generic "work" + "directed by" $\Rightarrow$ usually a **film**.
- **Prefer type consistency over title similarity.** If several titles are close, choose the candidate whose `subtitle` matches the inferred type.
- **Score each candidate** and select the highest:
  - Type consistency of `subtitle` with intent: **3** (exact) / **2** (near) / **0** (unrelated).
  - Explicit title match with the mention (after normalization): **2**.
  - Parenthetical label aligned with the type (e.g., "(film)" when intent is film): **1**.
  - Auxiliary keywords aligned with candidate type (e.g., release, author, platform): **1**.
- **Tie-breakers** (apply in order):
  - More precise/appropriate parenthetical label.
  - Shorter / less ambiguous title.
  - Earlier order in the input list.
- **Constraints**
  - Do not invent new entities; *abstain is not allowed*.
  - Print **no extra text or fields**; return the chosen JSON object only.
  - If no perfect match exists, return the **best available** candidate using the rules above.

Examples

Example 1 — intent: film

```
Input:
{
  "q": "Who directed Jurassic Park?",
  "entities": [
    {"title": "Jurassic Park", "subtitle": "Work (entity)"},
    {"title": "Jurassic Park (film)", "subtitle": "Film (entity)"},
    {"title": "Jurassic Park (video game)", "subtitle": "Video game (entity)"
        },
    {"title": "Jurassic Park (novel)", "subtitle": "Book (entity)"}
  ]
}
Output:
{"title": "Jurassic Park (film)", "subtitle": "Film (entity)"}
```

Important: Always return **exactly one** JSON object from the input list—and nothing else.

Relation Selector Prompt - Translated to English

You are the **Relation Selection** agent. You receive a Persian question and a *set of relation names*. Your job is to return up to **3** relations *taken verbatim from the input set* that are most likely to contain the answer.

Input (JSON):

```
{
  "q": "<string>",                        // Persian question
  "relations": ["<rel1>", "<rel2>", "..."] // a set/list of relation names
}
```

Output (JSON):

```
{
  "selected": ["<relA>", "<relB>", "<relC>"] // 0..3 items, each copied
      verbatim from input
}
```

Selection Rules

- **Use only the provided list.** No external knowledge; free semantic guessing is prohibited.
- **Do not alter relation names.** No translation, correction, deletion, or inserted characters.
- **Exact string matching (case-sensitive)** governs any literal match found in the question.
- **Assume mental normalization**: script variants, zero-width non-joiners, diacritics, and punctuation *must not* cause a missed match.

Selection Policy (priority with early stopping)

1. **Exact mentions first.** If one or more relation names appear *verbatim* in the question and exist in the input list, return **only** those (up to 3). If a known translation of a listed relation exactly matches the wording in the question, select it too.
2. Otherwise, select the **closest semantically** matching relations.
3. If neither (1) nor (2) yields a candidate, return the **empty set**: `[]`.
4. If (2) yields more than 3 candidates:
    - If near-duplicate labels exist (e.g., `birthPlace` vs. `birth_city`), keep only one.
    - Return at most 3 total.

Constraints

- **Do not** invent relations outside the input list.
- **Avoid** picking generic fallbacks (`name`, `description`, etc.) if a relevant relation is absent.
- Output **only** the JSON object above; no extra text.

Heuristics & Notes

- For "*Which city is this area near?*", include `affiliation` if present (administrative affiliation to a city).
- For "*Where is this region?*", include `district` if present.
- For questions about a place's **province**, include `city` if present.

Examples
Example 1

```
Input:
{
  "q": "Where was Aline Khalaf born?",
  "relations": [
    "activeYearsStartDate","genre","birthPlace","background","name",
    "hometown","birthCity","birth_place","activeYears","activeYearsEndDate"
  ]
}
Output:
{"selected": ["birthPlace", "birthCity", "birth_place"]}
```

Example 2

```
Input:
{
  "q": "What is Amy Adams's field?",
  "relations": ["description","influenced","field","fieldOfActivity","
      occupation","name"]
}
Output:
{"selected": ["field", "fieldOfActivity"]}
```

Example 3

```
Input:
{
  "q": "Who is Mehran Modiri's spouse?",
  "relations": ["spouse","birthPlace","genre"]
}
Output:
{"selected": ["spouse"]}
```

Example 4

```
Input:
{
  "q": "Which city is Dolatyar near?",
  "relations": [
    "ruralDistrict","name","county","district","settlementCode","village",
    "populationTotal","province","city","affiliation"
  ]
}
Output:
{"selected": ["district", "affiliation", "village"]}
```

Example 5

```
Input:
{
  "q": "In which years was #3 active?",
  "relations": [
    "yearsactive","ancestry","imagesize","birth_name","nationality","name",
    "background","death_place","birth_place","awards"
  ]
}
Output:
{"selected": ["yearsactive"]}
```

Example 6

```
Input:
{
  "q": "What is the longitude second of #1?",
  "relations": [
    "populationTotal","longitudeDegree","longitudeDirection","leaderTitle",
    "longitudeSecond","secondOfLongitude","leaderName",
    "latitudeMinute","latitudeDirection"
  ]
}
Output:
{"selected": ["longitudeSecond", "secondOfLongitude"]}
```

Path Scorer Prompt — Translated to English

You are the **Path Scorer**.

Goal
Given a question q and a list of candidate reasoning paths $P(q)$ — each path is an array of strings of the form `[entity, rel1, node1, rel2, node2, ...]` — assign each path a continuous score $s(p)$ in the range $[0, 1]$. This score must enable precise ranking for answer production.

Input

- `"question"`: the natural-language question.
- `"paths"`: a JSON array of paths; each path is *exactly* as provided in the input.

Output

- A JSON array of objects with the schema:

```
{"path": [... unchanged input path ...], "score": <float in [0,1]>}
```

- Output **only valid JSON**; write no extra text.

Scoring Dimensions (evaluate holistically)

1. **Semantic alignment** — how well the path's relations and nodes match the question intent. Treat Persian/English relation labels and spellings as equivalent (e.g., `"deathPlace"` ≡ "place of death", `"birthPlace"` ≡ "place of birth"), and be robust to transliteration/letter/diacritic variants.
2. **Structural coherence** — logical consistency of the chain, type compatibility across successive relations/nodes, and absence of inferential gaps.
3. **Directness/completeness** — whether the path directly and sufficiently covers the required answer vs. being indirect/partial/overly granular.

Single-Winner Constraint (for Hits@k)

- **Exactly one** path must receive score **1.0** (not 0.99). All others *must* be < 1.0 (e.g., ≤ 0.95).
- If multiple paths are truly equivalent in quality, use the tie-breakers below to pick *one* winner and assign the others a lower score.

Tie-Breaking Rules (in priority order)

1. More direct answer to the question text (higher semantic alignment).
2. Greater completeness.
3. Better structural coherence.
4. Shorter path that reaches the same answer (fewer hops).
5. If still tied: the path with the smaller index in the *input order*.

Additional Rules

- Be robust to Persian/English name variants (e.g., "New York" ≡ "New York in Persian Characters").
- Do *not* normalize or rewrite input paths; mirror them exactly in the `"path"` field.
- Assign *one* aggregated score per path that jointly reflects all dimensions.

Output Format (re-emphasis; JSON only)
Score each path separately:

```
[
  {"path": [...], "score": 0.93},
  {"path": [...], "score": 0.12}
]
```

Path Scorer — Examples

EXAMPLE 1 — Input

```
Question:
"What is the longitude seconds of the city where Christopher Nolan was born?"

Paths:
[
  ['Christopher Nolan', 'place of birth', 'London', 'latitudeSecond', '26'],
  ['Christopher Nolan', 'place of birth', 'London', 'longitudeSecond', '39']
]
```

EXAMPLE 1 — Output

```
[
  {"path": ['Christopher Nolan', 'place of birth', 'London', 'latitudeSecond',
       '26'], "score": 0.7},
  {"path": ['Christopher Nolan', 'place of birth', 'London', 'longitudeSecond
      ', '39'], "score": 1.0}
]
```

EXAMPLE 2 — Input

```
Question:
"Where is Nadan Olya?"

Paths:
[
  ["Nadan Olya", "province", "Yazd"],
  ["Nadan Olya", "district", "Bahabad District"],
  ["Nadan Olya", "county", "Bafq"]
]
```

EXAMPLE 2 — Output

```
[
  {"path": ["Nadan Olya", "province", "Yazd"], "score": 0.8},
  {"path": ["Nadan Olya", "district", "Bahabad District "], "score": 1.0},
  {"path": ["Nadan Olya", "county", "Bafq"], "score": 0.9}
]
```