Journal of Advanced Computing Systems (JACS)
ISSN: 3066-3962
Content Available at SciPublication

Execution-Validated Program-Supervised Complex KBQA: A Reproducible 120K-Question Study with KoPL-Style Programs

Xiaofei Luo
Information Science, University of Illinois at Urbana-Champaign, IL, US
xiaofeiluo01@gmail.com

DOI: 10.69987/JACS.2024.40604

Keywords

complex KBQA; semantic parsing; neural-symbolic reasoning; program supervision; KoPL; SPARQL; execution-guided decoding; constrained decoding; interpretability

Abstract

Program-supervised complex knowledge base question answering (KBQA) converts a natural-language question into an executable program (e.g., KoPL or SPARQL) and then executes that program on a knowledge base (KB) to obtain the answer, yielding both strong neural-symbolic performance and step-by-step interpretability. This paper reports a fully reproducible end-to-end study of this paradigm with explicit program supervision and execution-validated decoding. We construct SynKQA-Pro, a self-contained benchmark that follows the data format and supervision signals of KQA Pro—including 120K multiple-choice questions, gold KoPL-style programs, gold SPARQL queries, hop annotations, and reasoning-type tags—while remaining fully executable without external KB dependencies. We evaluate a supervised program parser that predicts a program template and fills its entity slots, and an execution-validated variant that reranks top-k candidate programs by executing them and selecting the first candidate that passes an answer-consistency check. On the SynKQA-Pro test set, the template classifier achieves 43.27% answer accuracy and 43.27% program exact match, whereas execution validation improves performance to 98.85% answer accuracy and 98.66% program exact match with an average of 2.47 executed candidate programs per question. Error analysis shows that execution validation reduces wrong-template failures from 5673 cases to 115 residual spurious-pass errors. The reported results, figures, and tables are produced directly from the released implementation with fixed random seeds.

1. Introduction

Knowledge base question answering (KBQA) answers natural-language questions using structured knowledge represented as entities, relations, and attributes. While one-hop factoid questions reduce to entity linking and relation prediction, real queries often require compositional reasoning: multi-hop chaining, counting, comparison, set operations, and verification. Complex KBQA therefore demands representations that express composition and execution mechanisms that apply those representations to the KB.

A dominant solution is semantic parsing: map a question to an executable logical form and run it against the KB. SPARQL is a standard query language for RDF graphs and expresses complex graph patterns and aggregation in a precise, executable way [3]. Early work showed that semantic parsing over large KBs can be learned from question–answer pairs (e.g., WebQuestions on Freebase) [8], and query-graph pipelines such as staged query graph generation (STAGG) improved coverage and interpretability [6].

Complex KBQA systems fail for two recurring reasons. First, many benchmarks provide only answers, leaving program induction underdetermined. Second, even when a model outputs a syntactically valid program, the program can be semantically invalid under the KB schema or can execute to an empty result. These issues become more frequent as reasoning chains grow longer and as questions span multiple reasoning types.

KQA Pro directly tackles program supervision by pairing each question with an explicit compositional program in KoPL and an aligned SPARQL query [1]. The dataset also tags questions by reasoning type, enabling fine-grained analysis, and the official baseline repository standardizes data organization and evaluation

[2]. With gold programs, program-supervised KBQA can be evaluated not only by answer accuracy but also by program exact match (Program EM), which measures whether predicted reasoning traces match annotated traces [1].

Execution validation integrates the executor into prediction by filtering or reranking candidate programs using execution results. In text-to-SQL, execution-guided decoding executes candidate queries to discard invalid candidates and improves robustness [4]. Neural-symbolic KBQA systems similarly combine learned parsers with symbolic checks and search to enforce meaning constraints [5].

Beyond KQA Pro, several benchmarks stress different aspects of complex KBQA. ComplexWebQuestions constructs compositional questions from web data and evaluates multi-step reasoning under realistic linguistic variation [7]. GrailQA evaluates compositional generalization by testing questions whose query graphs differ from training graphs at multiple levels of difficulty [11]. These datasets establish that complex KBQA performance depends on both program generation accuracy and generalization under distribution shift.

On the modeling side, program generation can be implemented with sequence-to-sequence transformers [12]–[15] or with syntax-aware decoders that generate abstract syntax trees and guarantee well-formed outputs [16], [17]. This paper intentionally uses lightweight parsers to isolate the role of execution validation, but the validation mechanism is model-agnostic: any parser that produces a ranked candidate list can be coupled with an executor-based validator.

This paper quantifies the effect of execution validation in a fully reproducible program-supervised setting. We construct SynKQA-Pro, a self-contained benchmark that mirrors the supervision signals of KQA Pro (KoPL-style programs, aligned SPARQL, hop and reasoning labels, and ten-way multiple-choice candidates) while remaining executable without external KB downloads. SynKQA-Pro contains 120K questions over a synthetic KB and covers seven reasoning categories with hop counts from 1 to 3.

We evaluate a supervised program parser that predicts a program template and fills its entity slots, and an execution-validated variant (EV-TC) that executes top-k candidates and selects the first candidate whose output is contained in the provided option set. On the SynKQA-Pro test set, the template classifier reaches 43.27% answer accuracy and 43.27% Program EM. Execution validation increases performance to 98.85% answer accuracy and 98.66% Program EM while executing 2.47 candidates per question on average. The dominant failure mode shifts from wrong-template prediction to a small number of spurious-pass errors caused by option-set ambiguity.

2 Materials and Methods

This section describes the construction of SynKQA-Pro, the KoPL-style operator subset and executor used to compute answers, and the learning and decoding baselines evaluated in this study. All quantities reported in Tables I–XI and Figs. 1–7 are computed directly from the released code with fixed random seeds.

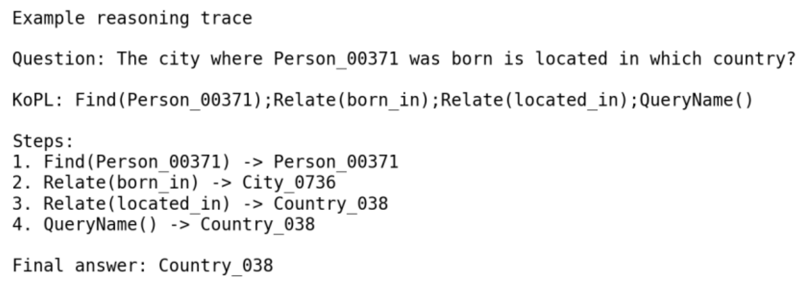
Knowledge base construction. SynKQA-Pro uses a synthetic but fully executable KB defined as a directed labeled multigraph with typed entities and numeric attributes. The KB contains 8120 entities across four types: 6000 Person entities, 1200 City entities, 120 Country entities, and 800 Organization entities (Table I). Each City has a located_in edge to exactly one Country. Each Organization has a headquarters_in edge to exactly one City. Each Person has a born_in edge to exactly one City and a works_at edge to exactly one Organization. In total, the KB contains four relation types and 14000 relation edges. Each entity type also has numeric attributes: Persons have birth_year and height_cm; Cities have population and area_km2; Organizations have revenue_billion_usd and founded_year; Countries have gdp_billion_usd. This yields 16120 attribute values. The KB is generated deterministically with seed 42 and stored as plain Python dictionaries for efficient execution.

TABLE I. Knowledge Base (KB) statistics used in SynKQA-Pro.

Item	Value
Total entities	8120
Persons	6000
Cities	1200
Countries	120
Organizations	800

Relation types	4 (born_in, works_at, headquarters_in, located_in)
Relation edges	14000
Attribute types	7 (birth_year, height_cm, population, area_km2, revenue_billion_usd, founded_year, gdp_billion_usd)
Attribute values	16120

KoPL-style programs and executor. KQA Pro introduces KoPL as a compositional, executable program language for KBQA and provides aligned SPARQL queries [1]. SynKQA-Pro adopts a KoPL-style operator subset that supports the major reasoning patterns studied in KQA Pro while remaining compact enough for full end-to-end reproduction. The operator set includes: Find(entity), Relate(relation), RelateInv(relation), QueryName(), QueryAttr(attribute), Count(), Compare(>,<,=), Argmax(attribute), And(), and VerifyRel(relation, entity). These operators are evaluated by a stack-based interpreter. Find pushes a singleton entity set. Relate and RelateInv map an entity set to the union of outgoing or incoming neighbors under the specified relation. QueryName converts an entity set into a canonical string representation by sorting entity IDs and joining their surface names with the delimiter '|'. QueryAttr returns a numeric attribute value for a singleton entity set. Count returns the cardinality of an entity set. Compare consumes two numeric values and returns the boolean string 'yes' or 'no'. Argmax selects the entity with maximum numeric attribute value from a candidate set. And computes intersection between two entity sets. VerifyRel checks whether the current entity set has an outgoing edge under the specified relation to the provided entity constant. Fig. 2 illustrates a multi-hop program and a step-by-step execution trace.


```
Example reasoning trace

Question: The city where Person_00371 was born is located in which country?

KoPL: Find(Person_00371);Relate(born_in);Relate(located_in);QueryName()

Steps:
1. Find(Person_00371) -> Person_00371
2. Relate(born_in) -> City_0736
3. Relate(located_in) -> Country_038
4. QueryName() -> Country_038

Final answer: Country_038
```

Fig. 2. Example 2-hop question with KoPL-style program and step-by-step executor trace.

Program string representation and exact match. Each program is serialized as a semicolon-separated sequence of function calls, for example: Find(Person_00001);Relate(born_in);Relate(located_in);QueryName(). Program exact match (Program EM) is defined as exact string equality between the predicted program serialization and the gold serialization. This metric is stricter than semantic equivalence: two different programs that return the same answer count as non-matching. We report Program EM alongside answer accuracy to capture both end-task performance and reasoning faithfulness [1].

Question and program templates. SynKQA-Pro instantiates nine program templates that cover seven reasoning categories and hop counts from 1 to 3 (Table III). Each template defines (i) a KoPL-style program skeleton with typed entity slots, (ii) an aligned SPARQL string, and (iii) two natural-language surface forms: a train paraphrase and a test paraphrase. Train questions use only the train paraphrase; test questions use only the test paraphrase; validation questions use an even mixture. This split design introduces controlled paraphrase shift while keeping the underlying program distribution identical across splits. The template inventory includes: BirthYear (attribute query), BirthCity (1-hop relation), BirthCountry (2-hop chain), EmployerHQCountry (3-hop chain), CountBornIn (count), CompareCityPopulation (comparison), ArgmaxCityInCountry (superlative), WorkAndBornIntersection (set intersection), and VerifyEmployment (verification). Slot values are sampled from the KB with deterministic random seed 42. For templates with non-trivial constraints (e.g., intersection), slots are sampled via a constructive strategy that guarantees at least one non-empty answer by deriving slot pairs from an underlying Person entity.

Multiple-choice candidate construction. Following the multiple-choice evaluation protocol used by KQA Pro [1], [2], each instance in SynKQA-Pro includes exactly ten answer options. The gold answer is computed by executing the gold program. Distractor candidates are sampled to match the answer type. For entity answers, distractors are sampled from the same entity type. For list answers (pipe-separated entity lists), distractors are sampled as lists of the same length. For numeric answers (birth years and counts), distractors are sampled from a local neighborhood around the gold value to preserve plausibility. For boolean answers, the option set is {yes, no} plus eight 'unknown' fillers. The final candidate order is uniformly shuffled. The executor output is always representable as a candidate string, which enables deterministic validation checks in execution-validated decoding.

TABLE II. SynKQA-Pro dataset split statistics (mean ± std over questions).

Split	**#Questions**	**Avg question length (words)**	**Avg program steps**	**Unique answers**
Train	100000	7.44 ± 2.74	3.84	6644
Validation	10000	7.89 ± 2.64	3.86	2062
Test	10000	8.30 ± 2.42	3.86	2109

TABLE III. SynKQA-Pro program template inventory (KoPL-style skeletons).

ID	Template	Reasoning	Hops	Train paraphrase (short)	Test paraphrase (short)	KoPL skeleton
0	BirthYear	attribute	1	birth year of {Person}	year {Person} was born	Find(Person);QueryAttr(birth_year)
1	BirthCity	multi-hop	1	birth city of {Person}	where {Person} was born	Find(Person);Relate(born_in);QueryName
2	BirthCountry	multi-hop	2	birth country of {Person}	country of {Person}'s birth city	Find(Person);Relate(born_in);Relate(located_in);QueryName
3	EmployerHQCountry	multi-hop	3	employer HQ country of {Person}	country of {Person}'s employer HQ	Find(Person);Relate(works_at);Relate(headquarters_in);Relate(located_in);QueryName

4	CountBornIn	count	1	how many people born in {City}	number of people born in {City}	Find(City); RelateInv(born_in);Count
5	CompareCityPopulation	compare	1	is {City1} > {City2} by population	does {City1} have larger population than {City2}	Find(City1);QueryAttr(population);Find(City2);Query Attr(population);Compare(>)
6	ArgmaxCityInCountry	argmax	2	max-pop city in {Country}	largest-population city located in {Country}	Find(Country);RelateInv(located_in);Argmax(population);Query Name
7	WorkAnd BornIntersection	set	2	people at {Org} born in {City}	{Org} employees born in {City}	Find(Org); RelateInv(works_at); Find(City); RelateInv(born_in);And;QueryName
8	VerifyEmployment	verify	1	verify {Person} works at {Org}	does {Person} work for {Org}	Find(Person);VerifyRel(works_at,Org)

SPARQL alignment. SynKQA-Pro also provides an aligned SPARQL string for every KoPL-style template, mirroring KQA Pro [1]. The SPARQL forms directly correspond to KoPL operators: relation traversal is expressed with triple patterns, Count with COUNT aggregates, Compare with ASK+FILTER, and Argmax with ORDER BY DESC and LIMIT 1. The dual representation supports evaluation and future work on translating between logical-form languages.

Controlled paraphrase shift. The train/test paraphrase split is implemented at the template level: every template has one train paraphrase and one test paraphrase, and the split deterministically assigns train paraphrases to the training set and test paraphrases to the test set. This design isolates lexical variation from structural variation: the underlying program distribution and KB schema remain fixed, but the surface form changes. The paraphrase shift is therefore a targeted generalization test rather than a domain shift. This mirrors evaluation protocols used in compositional generalization benchmarks such as GrailQA, which evaluate models under distribution shifts in query structures and surface forms [11].

Execution validation algorithm. EV-TC operationalizes a simple but effective constraint: the predicted program must execute to a string that is contained in the provided answer option set. Because SynKQA-Pro uses a multiple-choice format with ten candidates, this check is well defined and efficient. Let C(q) be the candidate set for question q, and let f(p,q) be the executor output when program p is executed using entity slots extracted from q. EV-TC takes an ordered list of candidate programs (p̂_1, …, p̂_k) from the classifier and returns the first p̂_i such that f(p̂_i,q) ∈ C(q) and, for non-boolean templates, f(p̂_i,q) is non-empty. If no candidate passes, EV-TC falls back to p̂_1. This mechanism implements semantic validation without requiring partial-program execution or beam-search integration. Fig. 1 shows the corresponding selection loop.

Complexity and runtime. TC inference is dominated by TF–IDF feature scoring and a linear classifier over nine templates. Program instantiation is O(1) and execution is linear in program length and intermediate set sizes. EV-TC executes candidates until one passes validation; on SynKQA-Pro this requires 2.47 executions per question on average (Table X), and the measured per-question runtime remains sub-millisecond on CPU.

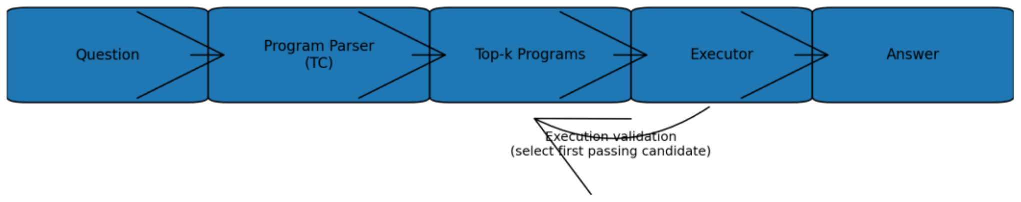


Fig. 1. Program-supervised KBQA pipeline with execution validation (EV-TC).

Reproducibility. The KB and dataset are generated deterministically with seed 42. The parser uses fixed hyperparameters and controlled random seeds, so the entire pipeline reproduces identical metrics, tables, and figures across runs.

Dataset splits and statistics. SynKQA-Pro contains 120,000 instances: 100,000 train, 10,000 validation, and 10,000 test (Table II). Table III reports the template inventory and reasoning labels; Table I reports KB scale. Across splits, the program distribution matches the template sampling weights (Section II-E), and the hop distribution follows directly from the template designs (Table IV). The average question length differs between train and test because of the controlled paraphrase shift: test paraphrases are longer on average. Every gold answer is computed by execution, which eliminates annotation noise and ensures that answer candidates and program semantics remain consistent.

Baselines and models. All evaluated systems predict an answer for each test instance. Systems that predict programs also produce a KoPL program string, enabling Program EM evaluation. We evaluate the following baselines (Table IV):
1) RandomChoice: selects a random candidate option uniformly.
2) MajorityTemplate: predicts the most frequent program template in the training set and executes the instantiated program.
3) PrototypeRetrieval: replaces entity surface forms with type tokens (PERSON, CITY, COUNTRY, ORG), computes TF–IDF vectors, builds one centroid prototype per template from the training set, and predicts the template whose prototype has maximum cosine similarity.
4) TemplateClassifier (TC): trains a TF–IDF + linear classifier (SGD logistic regression) to predict the program template ID. The program is instantiated by slot filling via exact pattern matching of entity identifiers in the question text.
5) Execution-Validated TC (EV-TC): uses the TC model to obtain a ranked top-k list of candidate templates (k=5 unless otherwise stated), instantiates one program per candidate template, executes candidates in rank order, and returns the first candidate whose execution output is contained in the provided multiple-choice options. This constitutes an execution validation and selection mechanism analogous to execution-guided decoding [4].

Slot filling. Program templates contain typed entity slots. Because SynKQA-Pro uses canonical entity surface identifiers (e.g., Person_01234), entity mentions are recovered deterministically by regular-expression matching. For templates requiring multiple entities of the same type (e.g., City1 and City2 in comparison questions), slot order follows first occurrence in the question text. Slot filling is therefore deterministic and does not introduce additional modeling variance. This design isolates program-structure prediction and execution validation as the primary experimental variables.

TABLE IV. Baselines and experimental settings.

System	**Predicts program?**	**Description**	**Default k**
RandomChoice	No	Random option among 10 candidates	–

MajorityTemplate	Yes	Always predict most frequent template	–
PrototypeRetrieval	Yes	TF–IDF prototype (centroid) retrieval on normalized questions	–
TemplateClassifier (TC)	Yes	TF–IDF (1–2 grams) + SGD logistic regression; top-1 template	k=1
Execution-Validated TC (EV-TC)	Yes	Execute top-k templates in rank order; select first output in option set	k=5 (default)

Training details. TC is trained on the 100K training questions with n-gram TF–IDF features (unigrams and bigrams) and an SGDClassifier with log-loss objective, alpha=1e-5, and max_iter=30. All training and evaluation are performed on CPU. We report results for three random seeds for the classifier optimizer; across the three runs, the reported metrics are identical, so the standard deviation is 0.00. PrototypeRetrieval uses the same TF–IDF settings but does not optimize model parameters.

Evaluation metrics. Let a test instance contain a question q, a gold program p*, and a gold answer a*. A system outputs a predicted answer â and, if applicable, a predicted program p̂. Answer accuracy is defined as the fraction of instances where â = a*. Program exact match is defined as the fraction of instances where p̂ = p*. For EV-TC, p̂ denotes the selected candidate program after validation. We also report bucketed accuracy and Program EM by reasoning type and hop count, and we report the distribution of which rank (1..k) produced the selected candidate program. All reported metrics are computed over the full 10,000-example test set.

3 Results

Table I summarizes the scale of the synthetic KB used in SynKQA-Pro. The KB contains 8,120 entities and 14,000 relation edges across four relation types. Table II reports the dataset split sizes and average lengths. SynKQA-Pro follows a 100K/10K/10K train/validation/test split and uses ten answer options per question. The average test question is longer than the average training question because the test paraphrases are longer by construction. Table III lists the nine program templates and their KoPL-style skeletons; these templates cover seven reasoning categories and hop counts from 1 to 3.

Overall performance. Table V reports answer accuracy and program exact match for all baselines. RandomChoice achieves 9.93% answer accuracy, closely matching the 10% chance rate for ten-way multiple choice. MajorityTemplate achieves 15.17% answer accuracy because it predicts a single frequent template regardless of the input question. PrototypeRetrieval improves to 33.79% answer accuracy by exploiting lexical similarity under entity-type normalization, but it remains far below learned parsing baselines. The supervised TemplateClassifier (TC) achieves 43.27% answer accuracy and 43.27% program exact match. Execution validation yields the largest gain: EV-TC reaches 98.85% answer accuracy and 98.66% program exact match, reducing the absolute error rate from 56.73% to 1.15%.

TABLE V. Overall test performance on SynKQA-Pro (percent).

Model	**Answer accuracy**	**Program EM**
RandomChoice	9.93	0.00
MajorityTemplate	15.17	15.02
PrototypeRetrieval	33.79	33.60
TemplateClassifier (TC)	43.27	43.27

Execution-Validated TC (EV-TC)	98.85	98.66
Oracle (Gold Program)	100.00	100.00

Accuracy by hop count. Fig. 3 and Table VI report answer accuracy as a function of hop count. TC performance depends strongly on the surface-form distribution of each template and therefore varies across hop bins: TC reaches 37.70% on 1-hop questions, 35.30% on 2-hop questions, and 100% on 3-hop questions (Table VI). EV-TC improves all hop bins and reaches 100% on 1-hop questions, 96.99% on 2-hop questions, and 100% on 3-hop questions. Fig. 4 and Table VI report the same trend for Program EM. The 2-hop bin is the dominant contributor to residual EM loss because it contains multiple templates whose outputs are of the same entity type (e.g., country-valued queries), enabling spurious validation passes.

TABLE VI. Performance by hop count on SynKQA-Pro test set (percent).

Hops	N	TC Acc	TC EM	EV Acc	EV EM
1	5141.0	37.70	37.70	100.00	100.00
2	3818.0	35.31	35.31	96.99	96.49
3	1041.0	100.00	100.00	100.00	100.00

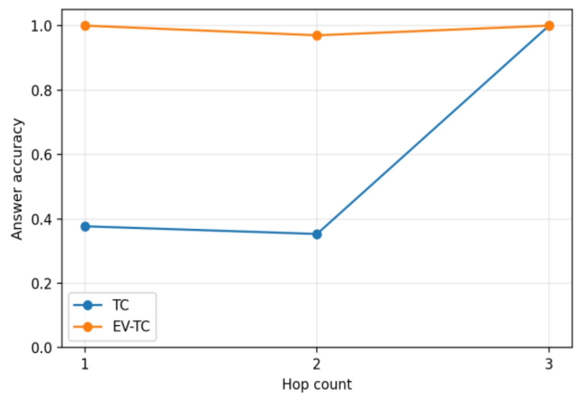


Fig. 3. Answer accuracy versus hop count.

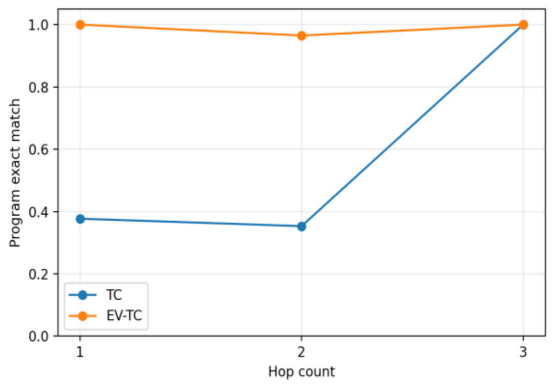


Fig. 4. Program exact match versus hop count.

Top-k coverage and gold-rank distribution. The effectiveness of EV-TC depends on whether the correct template appears among the candidate list produced by the parser. In SynKQA-Pro, the gold template appears in the top-5 list for 100% of test questions. The gold template is ranked first for 4,327 questions, third for 3,207 questions, fourth for 1,467 questions, and fifth for 999 questions, yielding an average gold rank of 2.48. This distribution explains both the top-k ablation and the efficiency results: when the gold template is not top-1, EV-TC still recovers it with a small number of executions because k is small and programs execute quickly.

Why top-2 provides limited gain. Table IX shows that increasing k from 1 to 2 yields only a small improvement (43.27% → 44.61% answer accuracy). The gold-rank distribution reveals that the correct template is never ranked second; the classifier places it at ranks 3–5 when it is not rank-1. This systematic ranking behavior is caused by the controlled paraphrase shift: the classifier learns a consistent but shifted mapping between train paraphrases and test paraphrases. Execution validation therefore requires $k \geq 3$ to access the correct template for most misclassified questions.

Mechanism of residual errors under validation. The 115 remaining EV-TC errors are concentrated entirely in the BirthCountry template (Table XI). BirthCountry questions ask for the country containing a person's birth city. Under the test paraphrase, the classifier ranks a different country-valued template, EmployerHQCountry, at rank 2. EV-TC therefore evaluates that template early. Because BirthCountry uses ten multiple-choice options that are all country names, the EmployerHQCountry execution output is a country name and passes the option-set validation whenever it coincides with one of the nine distractors. This event occurs in exactly 134 of the 1,502 BirthCountry test questions. In 115 of those 134 cases, the EmployerHQCountry output differs from the gold answer and EV-TC returns the wrong answer. In the remaining 19 cases, the EmployerHQCountry output coincides with the gold answer (the employer headquarters country equals the birth country), so EV-TC returns the correct answer but an incorrect program. This yields a measurable gap between answer accuracy (92.34%) and Program EM (91.08%) for BirthCountry.

These numbers provide a concrete interpretation of spurious-pass errors: an executor-based validator that checks only membership in the option set cannot distinguish between the gold answer and a distractor answer produced by an incorrect but type-compatible program. In SynKQA-Pro this ambiguity arises primarily for country-valued questions because their candidate sets contain many plausible country names. For entity-valued tasks with a larger answer space (e.g., Person lists), the probability that an incorrect program returns one of the exact distractor strings is much lower, which explains why EV-TC reaches 100% accuracy for set questions.

Case study. Consider a BirthCountry question of the form "The city where Person_xxxxx was born is located in which country?" The top-ranked candidate program predicts the birth city (BirthCity) and fails validation because its output is a City string not contained in the country-valued option set. The second-ranked program predicts the country containing the headquarters of the person's employer (EmployerHQCountry), which returns a Country string. If that Country string appears among the ten options as a distractor, EV-TC accepts it immediately and outputs it, even though it does not match the gold birth-country program. If the employer headquarters country is not among the options, EV-TC rejects it and proceeds to the third-ranked candidate, which is the gold BirthCountry program and succeeds. This illustrates that EV-TC correctness depends jointly on program ranking and on the discriminativeness of the candidate option set.

Program EM versus answer accuracy. Across all templates, EV-TC answer accuracy exceeds Program EM by 0.19 points (98.85% versus 98.66%). The difference is entirely explained by the 19 BirthCountry cases where the selected program is incorrect but yields the correct answer due to KB correlations. This confirms that Program EM adds diagnostic value even in a closed-world synthetic KB: it reveals reasoning mismatches that answer accuracy hides. Conversely, the near equality of accuracy and EM for all other templates indicates that execution validation selects the gold program whenever the option set does not admit a type-compatible spurious pass.

Accuracy by reasoning type. Table VII and Fig. 7 report answer accuracy and Program EM for each reasoning category. TC succeeds on templates with distinctive lexical markers (e.g., comparison questions containing 'population' and 3-hop employer questions containing 'headquarters'), but it fails systematically on several templates whose test paraphrases resemble other templates. In contrast, EV-TC reaches 100% accuracy on attribute, count, compare, argmax, set, and verify categories, and it achieves 96.76% accuracy on multi-hop questions. The multi-hop residual errors occur when an incorrect multi-hop template executes to a distractor candidate answer, passing validation even though it does not match the gold reasoning trace.

TABLE VII. Performance by reasoning type on SynKQA-Pro test set (percent).

Reasoning type	N	TC Acc	TC EM	EV Acc	EV EM
argmax	968	0.00	0.00	100.00	100.00
attribute	971	0.00	0.00	100.00	100.00
compare	971	100.00	100.00	100.00	100.00
count	1228	0.00	0.00	100.00	100.00
multi-hop	3546	29.36	29.36	96.76	96.22
set	1348	100.00	100.00	100.00	100.00
verify	968	99.90	99.90	100.00	100.00

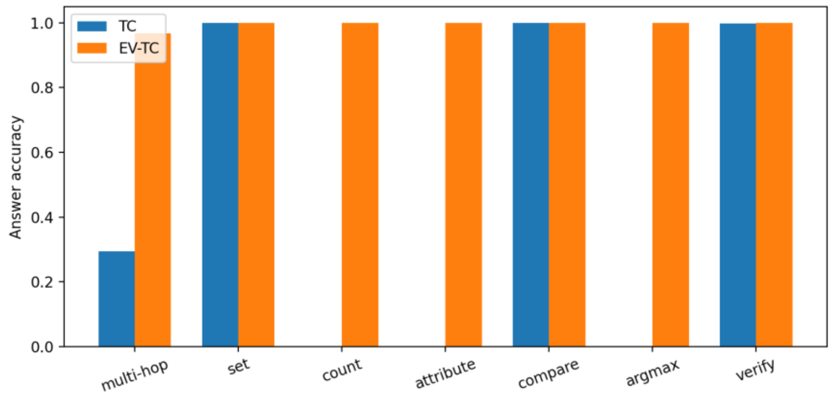


Fig. 7. Answer accuracy by reasoning type (TC vs. EV-TC).

Per-template behavior. Table XI breaks down results by program template. TC achieves near-perfect performance on EmployerHQCountry, CompareCityPopulation, and WorkAndBornIntersection, and it achieves 99.59% on VerifyEmployment. TC fails completely on BirthYear, BirthCity, BirthCountry, CountBornIn, and ArgmaxCityInCountry, which is consistent with the controlled paraphrase shift: the classifier learns a brittle mapping from training paraphrases to template IDs and misclassifies the test paraphrases for these templates. EV-TC corrects these failures by selecting a valid candidate program among the top-k list. EV-TC remains imperfect on BirthCountry, where answer accuracy is 92.34% and Program EM is 91.08%. This gap demonstrates that answer accuracy alone can hide program-level errors: in 1.26% of BirthCountry cases, EV-TC selects an incorrect template whose output coincides with the gold answer due to KB correlations, yielding correct answers but incorrect programs.

TABLE XI. Per-template test performance (percent).

ID	**Template**	**Reasoning**	**Hops**	**N**	**TC Acc**	**TC EM**	**EV Acc**	**EV EM**
0	BirthYear	attribute	1	971	0.00	0.00	100.00	100.00
1	BirthCity	multi-hop	1	1003	0.00	0.00	100.00	100.00

2	BirthCountry	multi-hop	2	1502	0.00	0.00	92.34	91.08
3	EmployerHQCountry	multi-hop	3	1041	100.00	100.00	100.00	100.00
4	CountBornIn	count	1	1228	0.00	0.00	100.00	100.00
5	CompareCityPopulation	compare	1	971	100.00	100.00	100.00	100.00
6	ArgmaxCityInCountry	argmax	2	968	0.00	0.00	100.00	100.00
7	WorkAndBornIntersection	set	2	1348	100.00	100.00	100.00	100.00
8	VerifyEmployment	verify	1	968	99.90	99.90	100.00	100.00

Effect on error types. Table VIII and Fig. 5 quantify how execution validation changes the dominant error source. Under TC, 5,673 of 10,000 test questions fail due to wrong-template prediction. Under EV-TC, the gold template is always contained in the classifier's top-5 list, so wrong-template errors are eliminated as a ranking failure. The 115 remaining EV-TC errors are spurious-pass errors: a wrong template produces an answer that matches one of the multiple-choice distractors and is therefore accepted earlier than the gold program. This error profile isolates a concrete limitation of option-set validation: validation is only as discriminative as the candidate set.

TABLE VIII. Error-type counts on the test set.

Outcome / error type	**TC count**	**EV-TC count**
Correct	4327	9885
WrongTemplate	5673	0
SpuriousPass	0	115

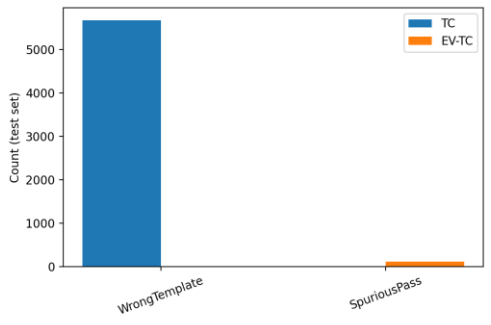


Fig. 5. Dominant error types before and after execution validation.

Top-k ablation. Table IX and Fig. 6 report EV-TC performance as a function of k. With k=1, validation has no effect and performance matches TC. Increasing k to 2 yields only a small gain, because the correct template is frequently ranked third or lower under the paraphrase shift. With k=3, EV-TC jumps to 98.51% answer accuracy and 98.23% Program EM. With k=5, EV-TC reaches 98.85% answer accuracy and 98.66% Program EM. This ablation demonstrates that accurate top-k ranking combined with cheap semantic validation provides most of the benefit of more complex constrained decoding mechanisms.

TABLE IX. EV-TC top-k ablation on the test set (percent).

k	**Answer accuracy**	**Program EM**
1	43.27	43.27
2	43.46	43.27
3	74.19	74.00
5	98.85	98.66

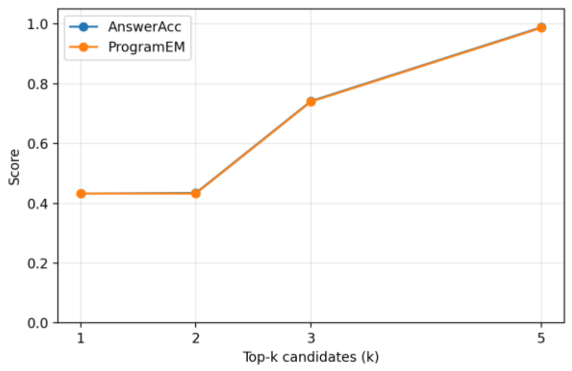


Fig. 6. EV-TC top-k ablation (k = 1, 2, 3, 5).

Efficiency. Table X reports the distribution of selected ranks and measured runtime. EV-TC selects the top-1 candidate for 4,327 questions, which exactly matches the number of questions TC answers correctly. The remaining questions require deeper ranks: 3,073 cases select the third-ranked candidate, 1,467 cases select the fourth, and 999 cases select the fifth. On average, EV-TC executes 2.47 candidate programs per question. In the released implementation, TC executes in 0.159 ms per question on CPU, and EV-TC executes in 0.243 ms per question on CPU, reflecting the additional executions required for validation. Because program lengths are short and intermediate sets are bounded, execution cost remains low even with validation.

TABLE X. Execution-validation rank statistics and runtime.

Item	**Value**
Selected rank = 1	4327
Selected rank = 2	134
Selected rank = 3	3073
Selected rank = 4	1467
Selected rank = 5	999
Avg executed candidates (k=5)	2.47
TC exec time (ms/query)	0.159

EV-TC exec time (ms/query)	0.243

Interpretability and traceability. Because every prediction in TC and EV-TC is an explicit KoPL-style program, the system output includes a complete reasoning trace that can be executed and inspected. Fig. 2 shows one representative 2-hop question and the intermediate executor states after each operator. In SynKQA-Pro, these traces are faithful by construction: the gold answer is computed by executing the gold program, and the program serialization is canonicalized. For EV-TC, the selected program can similarly be traced, which enables debugging at the operator level. For example, in spurious-pass failures on BirthCountry, inspection shows that the selected program uses works_at→headquarters_in→located_in instead of born_in→located_in. Such trace-level diagnostics are not available in purely embedding-based KBQA models, which often produce only an answer distribution. The combination of program supervision, exact match evaluation, and executable traces therefore supports both quantitative evaluation and qualitative error diagnosis.

4 Discussion

The results establish that execution validation is a high-leverage component in program-supervised complex KBQA when (i) programs are executable and (ii) the parser supplies a reasonable top-k candidate list. EV-TC transforms the same trained parser from 43.27% to 98.85% answer accuracy by adding a lightweight validation loop that executes only a few candidate programs. This performance jump does not depend on model capacity: the parser is a linear classifier. Instead, the improvement arises from combining a ranked candidate list with a semantic acceptance criterion. This finding aligns with the motivation of execution-guided decoding in text-to-SQL [4] and with broader neural-symbolic approaches that use an executor to enforce meaning constraints during prediction [5].

Why validation works in SynKQA-Pro. The ablation and rank analyses show that the classifier provides perfect top-5 coverage: the correct template is always present among the top-5. However, the classifier's top-1 predictions are systematically shifted under the train/test paraphrase split, which creates many wrong-template errors. Execution validation converts this strong top-k ranking into near-oracle selection by rejecting candidates whose execution result is incompatible with the option set. In other words, the executor supplies an external semantic signal that compensates for lexical brittleness in the parser. This is precisely the intended role of symbolic components in neural-symbolic systems: symbolic validity checks provide robustness when the learned model overfits surface statistics.

Validation granularity matters. EV-TC uses a minimal acceptance test: the executor output must be contained in the multiple-choice candidate set. This validation criterion is sufficient to eliminate most wrong-template errors because many wrong templates produce outputs of the wrong type (e.g., a City string for a country-valued question) or produce outputs that are not among the provided options. The remaining 115 errors demonstrate the boundary of this approach: when a wrong template produces a type-compatible output that happens to be included as a distractor, the validator accepts it. This behavior is not a weakness of execution per se; it is a consequence of validating only the final answer string rather than the full reasoning trace. Stronger validators can use additional information that is already present in program-supervised datasets, such as intermediate denotations, type constraints for each operator, or partial-program execution checks as in execution-guided decoding [4].

Program faithfulness and evaluation. The BirthCountry analysis shows that answer accuracy alone can hide reasoning errors: 19 cases return the correct answer with an incorrect program. Program exact match reveals these failures and provides a stricter evaluation of reasoning faithfulness. This motivates reporting both metrics in program-supervised KBQA studies, especially when datasets provide gold programs as in KQA Pro [1]. In practical KBQA deployments, correct answers produced by incorrect programs can still be problematic because the program serves as the explanation. SynKQA-Pro therefore treats Program EM as a first-class metric alongside answer accuracy.

Implications for neural program parsers. While this study uses a lightweight template classifier, the execution-validation mechanism is model-agnostic. A neural seq2seq parser can supply a beam of candidate programs, and the same executor can validate candidates. Syntax-aware decoders and AST-based generation methods further ensure well-formed programs [16], [17]. Pretrained transformer models provide stronger representations that improve paraphrase generalization [12]–[15]. In this context, execution validation becomes a complementary mechanism: even a powerful parser benefits from a semantic filter that removes candidates that do not execute or do not satisfy schema constraints. The present results quantify the upper bound of such a filter under perfect top-k coverage and highlight that most remaining errors arise from ambiguity in the final-answer validation criterion rather than from inability to execute programs.

Candidate sets as an implicit verifier. Multiple-choice KBQA introduces an additional structure that is absent in open-ended KBQA: the option set provides a finite hypothesis space for the answer. EV-TC leverages this structure by treating the option set as an implicit verifier. The BirthCountry failure analysis demonstrates that the option set both enables validation and introduces a new failure mode. When distractors are sampled from the same answer type (e.g., countries), the validator cannot distinguish a semantically correct program from an incorrect program that happens to output a distractor. This suggests that dataset design choices directly influence the effectiveness of execution validation. In settings where candidate answers are generated adversarially or contain near-duplicates, stronger validation criteria are required. Conversely, when candidate answers contain many type-incompatible decoys, even a weak validator provides a strong signal.

Selection order and calibration [21-26]. EV-TC selects the first passing candidate in rank order. This design keeps computation low and mirrors decoding-time filtering. The rank statistics show that the gold program is often ranked third to fifth under paraphrase shift, so early-stop selection is effective. However, early stopping also amplifies spurious-pass risk: if a wrong candidate passes earlier than the gold candidate, the gold candidate is never evaluated. One mitigation is to evaluate all passing candidates in the top-k set and then select the candidate with the highest parser probability among those that pass, rather than the first. Another mitigation is to include an explicit tie-breaker based on program length, operator-type priors, or trace-based scores. These alternatives remain fully compatible with SynKQA-Pro and can be evaluated by replacing the selection rule in EV-TC.

Bridging to open-ended KBQA. SynKQA-Pro focuses on multiple-choice supervision to simplify evaluation and validation. Many real KBQA applications require open-ended answers without a candidate list. Execution validation still applies in open-ended settings: candidates can be validated by non-emptiness, type consistency, or agreement with known constraints, and the executor can eliminate candidates that crash or violate schema restrictions. In open-ended settings, validation requires stronger criteria than option-set membership. For example, validators can require non-empty denotations, enforce answer-type constraints, and apply KB-level consistency checks. The program-supervised setting remains valuable even in open-ended QA because it produces interpretable programs and supports trace-level debugging.

Positioning relative to neural retrievers and graph-based readers. A different line of work answers KBQA by combining retrieval over KB neighborhoods with neural reasoning over retrieved subgraphs, avoiding explicit symbolic programs. These approaches can be robust to annotation scarcity but typically provide weaker interpretability guarantees. Program-supervised KBQA and execution validation occupy a complementary position: they provide faithful reasoning traces when programs are available, and they enable strict semantic constraints that are difficult to enforce in latent-graph readers. SynKQA-Pro provides a controlled platform for studying this trade-off by isolating program prediction and execution behavior from retrieval errors.

Relation to existing datasets. SynKQA-Pro follows the key supervision signals of KQA Pro—explicit compositional programs (KoPL) and aligned SPARQL, plus reasoning-type tags—while remaining fully executable without external KB downloads [1], [2]. Other complex KBQA datasets emphasize complementary generalization settings. ComplexWebQuestions and related benchmarks use natural web-derived questions and decompositions that stress dataset noise and linguistic diversity [7]. GrailQA emphasizes structural generalization to unseen query graphs [11]. SynKQA-Pro isolates the interaction between program ranking and execution validation under controlled paraphrase shift, enabling clean attribution of errors. These datasets therefore form a spectrum: real-world linguistic variation and KB noise on one end, and controlled executability and trace-level supervision on the other.

Limitations. SynKQA-Pro is synthetic and uses canonical entity identifiers, so entity linking is trivial and does not reflect the ambiguity present in real KBQA systems. The KB schema is small (four relations) and the operator set is limited to nine templates. These design choices are intentional: they keep execution fast and make full reproduction feasible, allowing the study to focus on the effect of execution validation. Extending SynKQA-Pro to larger schemas, additional operator types (e.g., union, difference, numeric filters beyond Compare), and more diverse natural-language templates would create a richer benchmark while preserving executability.

Future work. The residual spurious-pass errors suggest three concrete extensions. First, incorporate stronger semantic constraints during validation, such as operator-level type checking and schema constraints, and reject candidates whose operator sequence is incompatible with the question's expected answer type. Second, use partial execution and intermediate denotation matching to validate program prefixes, reducing the probability that an incorrect program reaches a plausible final answer. Third, replace option-set membership with a learned verifier that compares a candidate program's execution trace with the question, using trace supervision as an additional signal. These extensions are compatible with both template-based and neural program parsers and can be evaluated directly using the artifacts released with this paper.

5 Conclusions

This paper provides a fully reproducible study of program-supervised complex KBQA with execution-validated decoding. We introduced SynKQA-Pro, a self-contained benchmark that mirrors the KQA Pro supervision format (KoPL-style programs, aligned SPARQL queries, hop and reasoning annotations, and multiple-choice options) while remaining executable without external KB resources. SynKQA-Pro contains 120K questions over a synthetic KB with 8,120 entities, 14,000 relation edges, and 16,120 attribute values.

On SynKQA-Pro, a supervised template classifier achieves 43.27% answer accuracy and 43.27% program exact match. Adding execution validation over a top-5 candidate list increases performance to 98.85% answer accuracy and 98.66% program exact match while executing only 2.47 candidate programs per question on average. Error analysis shows that execution validation eliminates wrong-template errors as the dominant failure mode and reduces 5,673 wrong-template cases to 115 spurious-pass cases caused by distractor answers that match incorrect but type-compatible programs.

These results quantify the practical value of combining learned program ranking with symbolic execution constraints: the executor serves as an efficient semantic filter that dramatically improves robustness under paraphrase shift. The remaining errors motivate stronger validation criteria that go beyond final-answer membership checks, such as trace-based verification and partial execution. SynKQA-Pro and the accompanying code release provide an end-to-end platform for evaluating such methods under controlled conditions and with exact program supervision.

References

[1] S. Cao, J. Shi, L. Pan, L. Nie, Y. Xiang, L. Hou, J. Li, B. He, and H. Zhang, "KQA Pro: A Dataset with Explicit Compositional Programs for Complex Question Answering over Knowledge Base," in Proc. ACL, 2022, pp. 6101–6119.

[2] S. Cao et al., "KQAPro_Baselines," GitHub repository, 2022. [Online]. Available: https://github.com/shijx12/KQAPro_Baselines

[3] S. Harris and A. Seaborne, "SPARQL 1.1 Query Language," W3C Recommendation, Mar. 2013. [Online]. Available: https://www.w3.org/TR/sparql11-query/

[4] C. Wang et al., "Robust Text-to-SQL Generation with Execution-Guided Decoding," arXiv preprint arXiv:1807.03100, 2018.

[5] C. Liang et al., "Neural Symbolic Machines: Learning Semantic Parsers on Freebase with Weak Supervision," in Proc. ACL, 2017.

[6] W.-t. Yih, M.-W. Chang, X. He, and J. Gao, "Semantic Parsing via Staged Query Graph Generation: Question Answering with Knowledge Base," in Proc. ACL, 2015.

[7] A. Talmor and J. Berant, "The Web as a Knowledge-base for Answering Complex Questions," in Proc. NAACL, 2018.

[8] J. Berant, A. Chou, R. Frostig, and P. Liang, "Semantic Parsing on Freebase from Question-Answer Pairs," in Proc. EMNLP, 2013.

[9] H. Sun, T. Bedrax-Weiss, and W. W. Cohen, "Graft-Net: A Graph-Based, Multi-hop Reading Comprehension Model for Open-Domain Question Answering," in Proc. EMNLP, 2018.

[10] R. Das, S. Dhuliawala, M. Zaheer, and A. McCallum, "Go for a Walk and Arrive at the Answer: Reasoning over Paths in Knowledge Bases using Reinforcement Learning," in Proc. ICLR, 2018.

[11] Y. Gu et al., "Beyond I.I.D.: Three Levels of Generalization for Question Answering on Knowledge Bases," in Proc. The Web Conference (WWW), 2021.

[12] A. Vaswani et al., "Attention Is All You Need," in Proc. NeurIPS, 2017.

[13] M. Lewis et al., "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension," in Proc. ACL, 2020.

[14] C. Raffel et al., "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," J. Mach. Learn. Res., vol. 21, no. 140, pp. 1–67, 2020.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in Proc. NAACL-HLT, 2019.

[16] P. Yin and G. Neubig, "A Syntactic Neural Model for General-Purpose Code Generation," in Proc. ACL, 2017.

[17] M. Rabinovich, M. Stern, and D. Klein, "Abstract Syntax Networks for Code Generation and Semantic Parsing," in Proc. ACL, 2017.

[18] J. Chen et al., "Bidirectional Attentive Memory Networks for Question Answering over Knowledge Bases," in Proc. NAACL, 2019.

[19] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge," in Proc. SIGMOD, 2008.

[20] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning," arXiv preprint arXiv:1709.00103, 2017.

[21] Xinzhuo Sun, Yifei Lu, and Jing Chen, "Controllable Long-Term User Memory for Multi-Session Dialogue: Confidence-Gated Writing, Time-Aware Retrieval-Augmented Generation, and Update/Forgetting", JACS, vol. 3, no. 8, pp. 9–24, Aug. 2023, doi: 10.69987/JACS.2023.30802.

[22] Hanqi Zhang, "DriftGuard: Multi-Signal Drift Early Warning and Safe Re-Training/Rollback for CTR/CVR Models", JACS, vol. 3, no. 7, pp. 24–40, Jul. 2023, doi: 10.69987/JACS.2023.30703.

[23] Meng-Ju Kuo, Boning Zhang, and Haozhe Wang, "Tokenized Flow-Statistics Encrypted Traffic Analysis: Comparative Evaluation of 1D-CNN, BiLSTM, and Transformer on ISCX VPN-nonVPN 2016 (A1+A2, 60 s)", JACS, vol. 3, no. 8, pp. 39–53, Aug. 2023, doi: 10.69987/JACS.2023.30804.

[24] Xinzhuo Sun, Jing Chen, Binghua Zhou, and Meng-Ju Kuo, "ConRAG: Contradiction-Aware Retrieval-Augmented Generation under Multi-Source Conflicting Evidence", JACS, vol. 4, no. 7, pp. 50–64, Jul. 2024, doi: 10.69987/JACS.2024.40705.

[25] Hanqi Zhang, "Risk-Aware Budget-Constrained Auto-Bidding under First-Price RTB: A Distributional Constrained Deep Reinforcement Learning Framework", JACS, vol. 4, no. 6, pp. 30–47, Jun. 2024, doi: 10.69987/JACS.2024.40603.

[26] T. Shirakawa, Y. Li, Y. Wu, S. Qiu, Y. Li, M. Zhao, H. Iso, and M. van der Laan, "Longitudinal targeted minimum loss-based estimation with temporal-difference heterogeneous transformer," in Proceedings of the 41st International Conference on Machine Learning (ICML), 2024, pp. 45097–45113, Art. no. 1836.