STEPREL: EFFICIENT RELATIONAL FEATURE SEARCH BEYOND FULL INTERACTION HISTORY

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents can construct relational features by exploring databases, generating SQL programs, and revising them through validation feedback. As search progresses, they accumulate useful knowledge alongside a growing history of candidate programs and intermediate results. Retaining this history repeatedly processes information that could instead be reused selectively. We introduce STEPREL, which maintains database knowledge, the current SQL program, and search evidence outside the conversation. It updates programs through local edits and periodically rebuilds the context from the retained knowledge, program, and selected evidence. This separation preserves an exact program for further revision while carrying forward useful outcomes from earlier attempts. Across 18 RelBench tasks and four LLM backbones, STEPREL maintains competitive predictive performance under a shared evaluation protocol while reducing the context needed for continued search. These results demonstrate that effective relational feature search can continue without carrying its complete interaction history.

1 INTRODUCTION

Relational prediction requires constructing features from linked tables. Useful features may involve multi-hop joins, temporal windows, and conditional aggregations that are difficult to predefine. Classical feature-synthesis methods enumerate predefined operators (Kanter & Veeramachaneni, 2015; Lam et al., 2017), while relational learning methods learn representations from database structure (Fey et al., 2024; Robinson et al., 2024). LLM agents offer another approach: they inspect the database, generate executable SQL feature programs, and revise them using validation feedback. Systems such as ReFuGe and RelAgent organize this process through structured workflows or open-ended search (Kim et al., 2026; Huang et al., 2026). This process also accumulates database knowledge, useful feature constructions, and unsuccessful attempts. Later decisions depend on how this information is retained and exposed to the agent. For example, a failed query may reveal a constraint that remains relevant after the candidate itself has been abandoned.

A natural implementation retains database observations, candidate programs, errors, and validation feedback in a growing conversation. Each request carries useful information alongside superseded programs and intermediate diagnostics. Yet these contents serve different roles: database knowledge can be reused across trials. The executable program must be preserved exactly to support reliable subsequent changes. Earlier outcomes can guide decisions even when candidates fail to improve the current program. Keeping the complete transcript repeatedly exposes the agent to the process that produced these results. Truncating or summarizing it without distinguishing their roles can instead discard information needed to continue search. This motivates explicit search state from which subsequent context can be assembled. The challenge is to retain enough information for continued search without repeatedly presenting every intermediate step to the model.

We introduce STEPREL, which maintains persistent search state outside the conversation. It stores reusable database knowledge and the current best SQL program, while maintaining structured search evidence from evaluated candidates. This evidence tracks program changes, validation outcomes, and recurring failures. The agent modifies individual SQL blocks, while the system assembles each candidate using the unchanged parts of the stored program. Periodically, STEPREL rebuilds the conversation around the database knowledge, the current program, and selected search evidence. The stored program supplies the exact basis for the next edit, while the evidence records alternatives

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

and failures beyond that program. This context supports further revisions informed by previous attempts, without replaying the full interaction history.

We evaluate STEPREL on 18 RelBench tasks with four LLM backbones. Under a shared evaluation protocol, STEPREL maintains competitive predictive performance with reduced context requirements. Targeted comparisons further examine database knowledge reuse and local program updates. These results show that search can preserve predictive quality while relying less on the full interaction history.

2 RELATED WORK

2.1 RELATIONAL LEARNING AND FEATURE ENGINEERING

Classical relational feature-engineering systems construct explicit predictors by composing joins, aggregations, and transformations along relational paths. Deep Feature Synthesis and OneBM enumerate features from predefined operators, while later systems automate source selection or search richer feature spaces (Kanter & Veeramachaneni, 2015; Lam et al., 2017; Chepurko et al., 2020). FeatAug searches predicate-aware aggregation queries over one-to-many relations rather than materializing the full candidate space (Qi et al., 2024). These methods produce executable and interpretable features, but generally operate within predefined operators or query templates.

Relational models learn representations from database structure. RelBench and 4DBInfer standardize relational prediction tasks by representing rows and key relations as heterogeneous or temporal graphs (Robinson et al., 2024; Wang et al., 2024). Architectures such as ContextGNN and RelGT improve task-specific relational modeling, while relational foundation models including Griffin, the Relational Transformer, KumoRFM-2, and RDB-PFN pursue transfer across databases through pretraining, fine-tuning, or in-context learning (Yuan et al., 2025; Dwivedi et al., 2025; Wang et al., 2025; Ranjan et al., 2025; Hudovernik et al., 2026; Wang et al., 2026). We instead study iterative LLM search over explicit relational feature programs.

2.2 LLM-BASED FEATURE ENGINEERING AND RELATIONAL AGENTS

LLMs generate executable features from task descriptions, column semantics, and data examples. FeatLLM and CAAFE generate rules or transformations for single-table prediction (Han et al., 2024; Hollmann et al., 2023). Subsequent methods incorporate downstream evaluation into the generation loop: decision-tree-based reasoning provides feedback for refining features, LLM-FE combines LLM proposals with evolutionary program search, and memory-augmented multi-agent systems retain useful transformations and evaluation outcomes across rounds (Nam et al., 2024; Abhyankar et al., 2025; Dong et al., 2026). Other studies report overproduction of simple transformations and challenges in controllability, cost, and usability (Küken et al., 2024; Schäfer et al., 2025).

In relational settings, some methods collect linked records and serialize them for direct LLM prediction, while Rel-LLM combines structured prompting with learned relational representations (Wydmuch et al., 2024; Wu et al., 2025). ReFuGe organizes schema selection, candidate generation, and validation-based feature filtering into a structured multi-agent pipeline (Kim et al., 2026). RelAgent explores databases, constructs SQL feature programs, and selects downstream models and hyperparameters through open-ended search with validation feedback (Huang et al., 2026). These works study how relational features are proposed, evaluated, and refined. STEPREL focuses on organizing the accumulated state needed to continue this search.

2.3 AGENT STATE AND CONTEXT MANAGEMENT

Long-horizon agents retain information outside the conversation through external memory, trajectory compression, and structured workspaces. MemGPT introduces hierarchical external memory, while ACON, CAT, and SAM respectively study compressed trajectories, agent-controlled context operations, and compact cues with access to retained history (Packer et al., 2023; Kang et al., 2025; Liu et al., 2026; Hu et al., 2026). These methods primarily determine which past information should remain available to the agent.

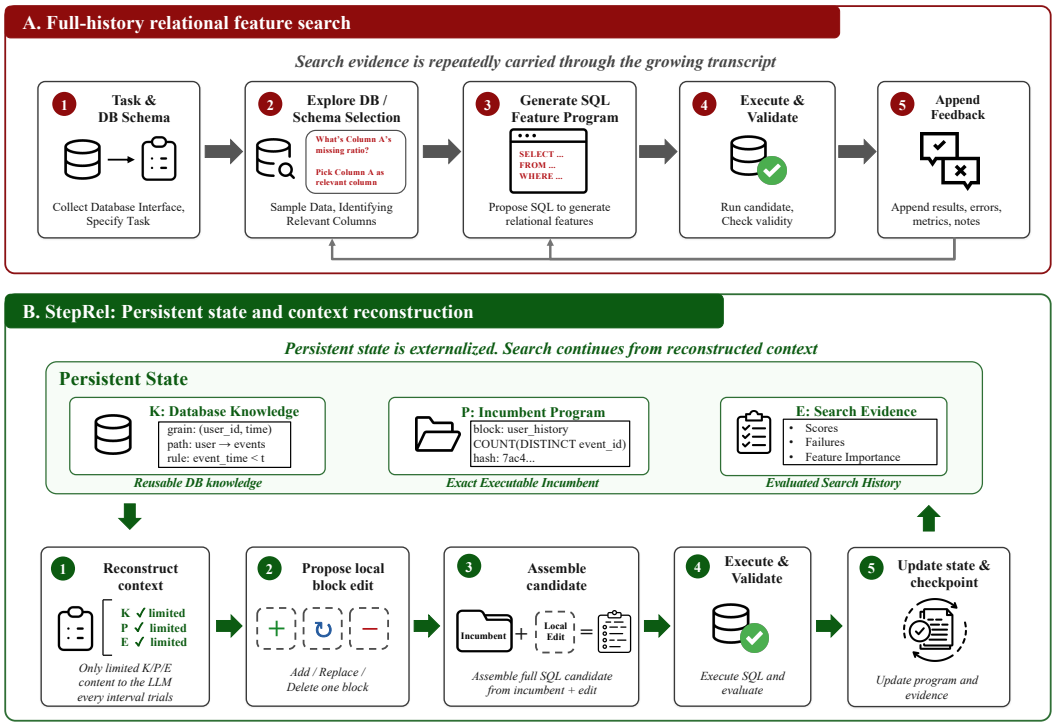


Figure 1: Overview of StepRel. (a) Full-history search retains earlier interactions in the conversation. (b) StepRel maintains database knowledge, the incumbent program, and search evidence outside the conversation, applies local program edits, and periodically reconstructs context from the retained state.

Other work represents evolving tasks as explicit execution state. AgentProg organizes long-horizon GUI interaction as a semantic program, while MAGE maintains a hierarchical execution-state structure supporting compression, validation, and revision (Tian et al., 2026; Chen et al., 2026). StepRel applies this perspective to relational feature search by maintaining executable program state and selecting accumulated search evidence for context reconstruction.

3 Agentic Relational Feature Search

3.1 Relational Prediction and Feature Programs

A relational database is a collection of tables $\mathcal{D} = \{R_1, \ldots, R_m\}$ linked through primary and foreign keys. For each split $s \in \{\mathrm{tr}, \mathrm{va}, \mathrm{te}\}$, a prediction task provides an anchor table A^s and labels Y^s. Each row of A^s identifies a prediction instance through entity keys and, where applicable, a prediction timestamp. The anchor table excludes the labels. Let G denote the columns that uniquely identify each prediction instance and align its features with its label.

A *feature block* b_j is a named SQL query that produces one or more feature columns for each prediction instance. Its output retains G and must contain at most one row per anchor. A complete feature program is an ordered collection of blocks $P = (b_1, \ldots, b_J)$. Executing these blocks against the database available for split s, denoted by $\mathcal{D}^s$, and aligning their outputs with A^s produces the feature matrix

$$X_P^s = \mathrm{Execute}(P; \mathcal{D}^s, A^s). \tag{1}$$

The matrix contains one row per prediction instance and is paired with Y^s for downstream training or evaluation. The alignment columns G are excluded from the predictive feature set.

3.2 Evaluation-Guided Program Search

Relational feature search iteratively proposes and evaluates SQL programs. At trial t, let Γ_t denote the context available to the agent. The agent proposes a candidate program $\widetilde{P}_t$, and an evaluator

returns its outcome:

$$\widetilde{P}_t \sim \pi_\theta(\cdot \mid \Gamma_t), \qquad z_t = \mathrm{Evaluate}(\widetilde{P}_t). \tag{2}$$

For an executable candidate, the evaluator constructs training and validation features, fits a downstream predictor, and measures validation performance. The outcome z_t includes execution status, the validation score when available, and diagnostic feedback. These observations guide subsequent proposals. When search terminates, the program selected by validation performance is frozen and evaluated on the held-out test split without further revision.

This process accommodates both structured pipelines and open-ended agents. A structured pipeline may prescribe separate stages for schema selection, feature generation, and filtering. An open-ended agent may decide what to inspect and how to revise a program. Both use information accumulated through database exploration and candidate evaluation. Let Σ_t denote this retained search information. The model accesses it through the context and tool responses supplied during search. A central design choice is how Σ_t is maintained and which parts enter subsequent context.

A full-history implementation retains database observations, candidate programs, errors, and feedback in the conversation. Later requests therefore process earlier interactions repeatedly. To illustrate this accumulation, let C_n be the logical input length of model request n, let B be the fixed prompt length, and let Δ_i be the tokens added to the retained history after request i. Under an append-only history, with each interaction adding approximately Δ tokens, individual and cumulative input satisfy

$$C_n = B + \sum_{i=1}^{n-1} \Delta_i, \qquad \sum_{n=1}^{N} C_n \approx NB + \frac{\Delta N(N-1)}{2}. \tag{3}$$

This idealized calculation describes repeated logical input, rather than provider-specific processing cost or billing. Requests are counted separately from trials because one candidate evaluation may involve multiple model requests.

Beyond its token cost, the transcript mixes reusable database knowledge, current and superseded programs, and outcomes from earlier attempts. These contents serve different purposes in continued search. The current program provides the starting point for further revisions, while previous outcomes record which changes succeeded, failed, or did not improve validation performance. This motivates an explicit representation of retained search information and a mechanism for selecting what enters subsequent context.

4 STEPREL

STEPREL separates information needed across search iterations from the conversation used to propose the next candidate. It represents the retained search state as

$$\Sigma_t = (K, P_t, E_t), \tag{4}$$

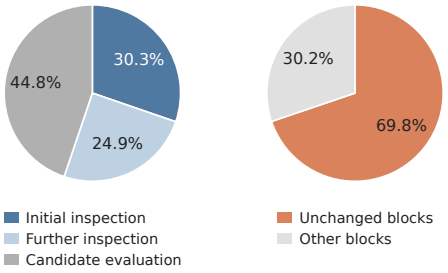
where K contains reusable database knowledge, P_t is the incumbent SQL program, and E_t records evidence from evaluated candidates. The agent revises the stored program through local edits. Periodically, the conversation is rebuilt from these components so that search can continue without retaining the complete interaction history.

A *scored trial* is a candidate evaluation that returns a finite validation score. A checkpoint is constructed after every k scored trials.

4.1 DATABASE KNOWLEDGE AND PROGRAM UPDATES

STEPREL retains reusable database knowledge in K so that later proposals can refer to it without carrying forward the full exploration history. Database exploration occurs throughout RelAgent search: among recorded inspection and candidate-evaluation calls in the RelAgent trajectories, 30.3% are inspection before the first candidate evaluation and 24.9% are inspection afterward (Figure 2a). STEPREL constructs K before search from the task definition, database schema, and a limited read-only exploration stage. The resulting database brief is reused when the conversation is rebuilt.

Successive programs also contain substantial repetition. An average of 69.8% of candidate blocks are unchanged from the preceding incumbent (Figure 2b). STEPREL therefore maintains the incumbent P_t in an external Program Store as an ordered collection of named SQL blocks and downstream model metadata. After bootstrap, the agent proposes an edit δ_t that adds, replaces, or removes one named block. The store carries forward all unchanged blocks without requiring the agent to generate them again, constructing the complete candidate as $\widetilde{P}_t = \mathrm{ApplyEdit}(P_t, \delta_t)$. Content hashes identify program versions and the individual blocks referenced by edits.



(a) Database inspection (b) SQL-block reuse

Figure 2: Database exploration and program reuse in RelAgent. (a) Shares of inspection and candidate-evaluation calls. (b) Shares of candidate SQL blocks unchanged from the preceding incumbent versus other blocks.

4.2 MAINTAINING SEARCH EVIDENCE

The incumbent records the best program found so far, but not the alternatives tested along the way. Non-improving candidates and execution failures can inform later proposals even though they do not change the incumbent. STEPREL retains these outcomes in E_t, linking each candidate to its parent, proposed revision, and execution and validation feedback.

The incumbent and evidence are updated as

$$P_{t+1} = \begin{cases} \widetilde{P}_t, & \text{if valid and improving,} \\ P_t, & \text{otherwise,} \end{cases} \qquad E_{t+1} = \mathrm{Record}\left(E_t; P_t, \widetilde{P}_t, z_t\right). \tag{5}$$

Thus, only an improving candidate becomes the starting point for the next edit, whereas every evaluated candidate contributes evidence.

Persistent storage and context selection serve different purposes. The stored records preserve candidate relationships and outcomes, while a deterministic selection produces the evidence shown to the agent. This view includes evidence associated with incumbent blocks, recent search transitions, and high-scoring trials. It also retains representative non-improving alternatives and groups execution failures, so that the view covers unsuccessful attempts as well as improvements. Additional block records are prioritized by feature importance, attempt count, and best validation score, and included as space permits. Records outside the rendered view remain in persistent storage, with omissions recorded explicitly. The resulting view carries forward search experience without reproducing every tool response.

4.3 RECONSTRUCTING THE INTERACTION CONTEXT

Between checkpoints, the live conversation accumulates proposals and feedback for local reasoning. After every k scored trials, STEPREL constructs a checkpoint $Q_t = \mathrm{RenderCheckpoint}(K, P_t, E_t)$ from the current persistent state. The checkpoint contains the database brief, the complete incumbent, and the evidence view described in Section 4.2. It initializes a fresh conversation that replaces the preceding interaction history. The stored program and accumulated evidence remain available, and the next edit is applied to the same incumbent. Checkpoint construction uses the stored records and deterministic rendering rules without an additional LLM call.

The interval k determines how many scored trials occur between reconstructions. Checkpoint size can grow with the incumbent, which is retained in full, but does not accumulate the complete sequence of earlier programs, diagnostic responses, and reasoning steps. This separates growth in the retained program from growth in the interaction history.

Algorithm 1 brings together program updates, evidence maintenance, and context reconstruction. The counter n advances only when evaluation returns a finite validation score. The selected program is frozen before test evaluation.

Algorithm 1 STEPREL Search with Context Reconstruction

Require: Database $\mathcal{D}$, search/test protocols $\Omega_{\mathrm{s}}, \Omega_{\mathrm{te}}$, checkpoint interval k
$K \leftarrow \mathrm{BuildDBContext}(\mathcal{D}, \Omega_{\mathrm{s}})$; $\Gamma \leftarrow \mathrm{InitialContext}(K)$
$P \leftarrow \mathrm{BootstrapProgram}(\Gamma)$; $E \leftarrow \varnothing$; $n \leftarrow 0$
while search budget remains **do**
  $\delta \leftarrow \mathrm{ProposeEdit}(\Gamma, P)$; $\widetilde{P} \leftarrow \mathrm{ApplyEdit}(P, \delta)$
  $z \leftarrow \mathrm{Evaluate}(\widetilde{P}; \Omega_{\mathrm{s}})$
  Update (P, E) using Eq. equation 5
  $\Gamma \leftarrow \mathrm{Append}(\Gamma, \delta, z)$
  $n \leftarrow n + \mathbf{1}\{z \text{ has a finite validation score}\}$
  if $n = k$ **then**
    $Q \leftarrow \mathrm{RenderCheckpoint}(K, P, E)$
    $\Gamma \leftarrow \mathrm{ResetWith}(Q)$; $n \leftarrow 0$
  end if
end while
$P^{\star} \leftarrow \mathrm{Freeze}(P)$
return $P^{\star}, \mathrm{Evaluate}(P^{\star}; \Omega_{\mathrm{te}})$

5 EXPERIMENTS

We evaluate whether STEPREL's explicit search state supports predictive relational feature search with less context processing. Our primary comparison holds the backbone, evaluator, data-access protocol, and search stopping policy fixed between STEPREL and RelAgent. We compare predictive utility and rollout variability (**RQ1**), measure context and token costs at matched search depth (**RQ2**), examine database knowledge reuse and local program updates (**RQ3**), and inspect representative generated SQL programs (**RQ4**).

5.1 EXPERIMENTAL SETUP

Tasks and comparison methods. We use ten binary-classification and eight regression tasks from RelBench v1 (Robinson et al., 2024), evaluated by test ROC–AUC and MAE, respectively. We exclude `rel-f1` due to temporal-validity concerns reported in prior work (Xu et al., 2026). Our primary baseline, RelAgent, performs open-ended SQL feature search while retaining the full interaction history (Huang et al., 2026). ReFuGe organizes schema selection, feature generation, and validation-based filtering into a structured multi-agent pipeline (Kim et al., 2026). It provides an additional reference under its native, lower-budget search procedure. We evaluate DeepSeek-V4-Pro, Qwen3.6-35B-A3B, Qwen3-Coder-30B-A3B-Instruct, and GPT-5.6 Luna (DeepSeek-AI, 2026; Qwen Team, 2026; Team, 2025; OpenAI, 2026).

Shared execution and search protocol. The agents use the official data splits and the same CatBoost evaluator with 200 trees, learning rate 0.05, depth 6, and L_2 regularization 3. During search, SQL accesses a validation-stage database snapshot and target-free anchors. Labels remain private to the evaluator, and the selected program is frozen before test execution. STEPREL and RelAgent share a stopping policy that checks validation improvement every six scored trials starting at trial 18, with a maximum of 42 scored trials. STEPREL reconstructs its conversation every six scored trials. Shared data-access and evaluator settings provide a common basis for comparing STEPREL, RelAgent, and ReFuGe.

Runs and analysis. We conduct three stochastic rollouts per task, backbone, and method. Each rollout selects its program by validation performance. We summarize predictive utility and variability across rollouts. RQ1 and RQ2 compare complete systems; RQ3 examines specific interfaces using the available matched contrasts.

Additional benchmark references. For broader benchmark context, we include target-table predictors such as LightGBM and TabPFN-2.5 (Ke et al., 2017; Grinsztajn et al., 2025); feature-synthesis pipelines such as DS+LightGBM and DS+AutoGluon (Kanter & Veeramachaneni, 2015; Erickson et al., 2020); relational models such as RelGNN and RelGT (Chen et al., 2025; Dwivedi et al., 2025); and relational foundation or transfer models such as KumoRFM, RDBLearn, and RDB-PFN (Fey et al., 2025; Hudovernik et al., 2026; Xu et al., 2026; Wang et al., 2026). These

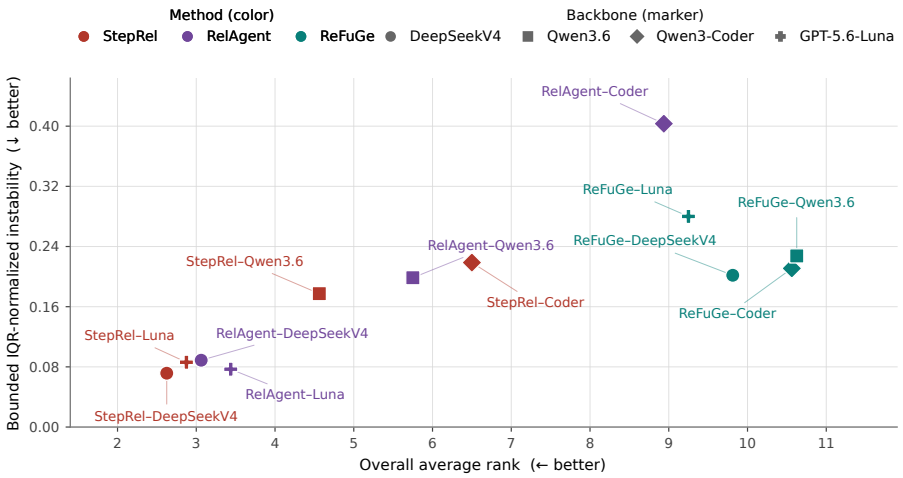


Figure 3: Overall predictive utility and rollout instability under the shared protocol. Lower is better on both axes; classification and regression receive equal weight. Ranks use common tasks with observed results, while variability uses cells with at least two finite rollouts. ReFuGe retains its lower search budget.

published results retain their original protocols and serve as supplementary benchmark references. The complete list and task-level results are provided in the supplementary comparison.

5.2 RQ1: Predictive Utility and Stability

We assess both the predictive quality of repeated searches and the variation in their outcomes. Figure 3 combines mean-performance ranks with empirical rollout instability, giving classification and regression equal weight. Instability measures task-normalized pairwise differences between rollout test scores, so lower values indicate more consistent predictive outcomes. The detailed normalization and task-level results are provided in the supplementary analysis.

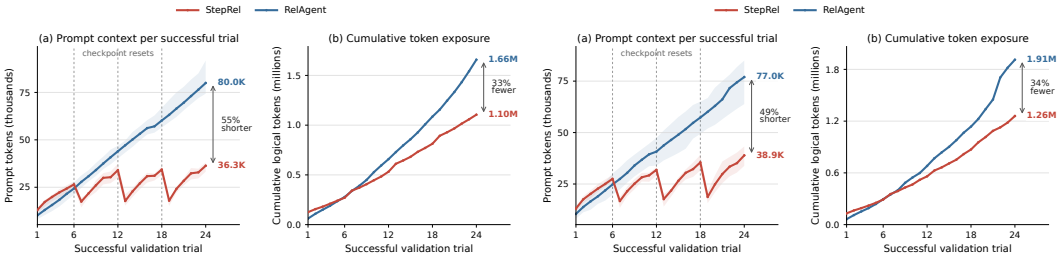
StepRel achieves a better overall mean-performance rank than same-backbone RelAgent for all four backbones, together with lower empirical instability for three of them. Its configurations generally occupy the region of stronger predictive performance and lower variation. This pattern indicates that reconstructing the conversation can preserve effective search while producing more consistent outcomes across repeated runs. The stability improvement is not uniform across backbones, and these comparisons assess the integrated system rather than isolate the effect of checkpointing alone.

Qwen3-Coder as a search stress case. RelAgent with Qwen3-Coder exhibits the largest instability among the evaluated configurations and substantially weaker predictive ranking than StepRel with the same backbone. The overall ranks are 8.94 and 6.50, respectively. It also produces a finite endpoint on only 16 of 18 tasks, compared with all 18 for StepRel. The unsuccessful trajectories repeatedly encounter feature-construction and execution errors. This case highlights the importance of maintaining usable search progress under a challenging backbone. Endpoint coverage is reported separately from empirical variability, which is measured only where repeated finite outcomes are available.

Broader benchmark context. The supplementary task-level comparison includes published tabular, relational, and agentic results under their respective protocols. It provides context for the predictive quality of the generated features. Differences in tuning budgets, evaluators, and model selection prevent that comparison from isolating the effect of our search design.

5.3 RQ2: Context Efficiency

We track prompt length and cumulative input and output tokens as search progresses. Figure 4 compares StepRel and RelAgent with DeepSeek, pairing runs by task and replicate and aligning them by scored trial. The trial-24 comparison includes pairs in which both runs reach that boundary. StepRel's cumulative cost includes one third of the reusable database-brief construction cost per rollout.



(a) Classification (b) Regression

Figure 4: Context growth and cumulative model-token processing with DeepSeek. Prompt length drops after context reconstruction, reducing cumulative processing at matched search depth. STEPREL totals include amortized database-brief construction.

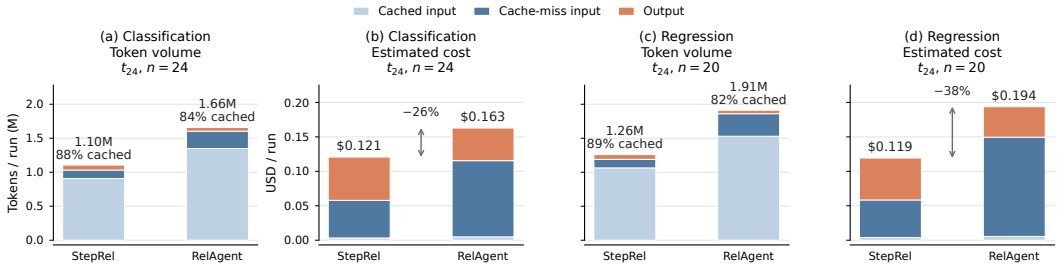


Figure 5: DeepSeek token volumes and estimated API costs through 24 scored trials for matched classification and regression runs. Stacks separate cached input, cache-miss input, and output. Costs use the evaluation price schedule, and STEPREL includes amortized database-brief construction.

Context growth and cumulative processing. The prompt curves show a recurring pattern: STEPREL's context grows within each search interval and drops after checkpoint reconstruction, while RelAgent's context continues to accumulate. The cumulative curves show that this reduction also lowers the processing required to reach the same search depth. By trial 24, STEPREL reduces prompt length by 54.6% on classification and 49.5% on regression, with cumulative token reductions of 33.3% and 34.1%, respectively. Thus, the system approximately halves late-stage context and reduces cumulative processing by one third, including the amortized cost of database exploration. Results for the other backbones appear in Appendix D. These results compare the complete systems; RQ3 examines database knowledge reuse and local program updates.

Cost with input caching. Provider caching discounts repeated input, so token reductions do not translate directly into proportional monetary savings. Figure 5 shows both token volumes and estimated charges, separating cache-hit input, cache-miss input, and output at the same trial-24 boundary. Although most input tokens are cached, cache-miss input and output account for most of the estimated cost. STEPREL reduces cache-miss input from 254.2K to 125.5K tokens on classification and from 331.0K to 125.5K on regression. Under the evaluation price schedule, estimated API cost falls by 25.9% and 38.4%, respectively, including amortized database-brief construction. Context reduction therefore yields lower estimated fees even after accounting for input caching. Pricing details and accounting at earlier search depths appear in Appendix D.

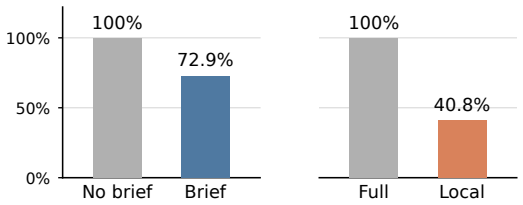
5.4 RQ3: DATABASE KNOWLEDGE AND PROGRAM UPDATES

Our motivation identifies repeated database exploration and repeated SQL generation as two sources of search overhead. We revisit these observations through targeted comparisons of the database brief and local program updates in Qwen3.6 classification runs. K uses 5 tasks and 10 seed pairs, and P uses 4 tasks and 7 seed pairs. The paired subsets differ between contrasts. Runs are paired by task and seed, with averages computed within tasks and then across tasks with equal weight. Figure 6 shows cumulative main-agent input at each run's stopping point, excluding separate brief construction. We report final validation-selected test AUC alongside these totals.

Database brief. We compare agents with and without the database brief, both using full-program proposals and structured context reconstruction. Both retain the required database interface specification and query tools. At 12 scored trials, the brief reduces recorded database calls by 39.8%,

consistent with less on-demand exploration. Over the complete runs, cumulative input is 27.1% lower and mean final test AUC changes from 0.8184 to 0.8190. The runs average 14.0 rather than 17.4 scored trials, so the final input difference also reflects shorter searches.

Local program updates. With structured context reconstruction enabled and no database brief, we compare local updates against full-program proposals. Both retain the exact incumbent. Local updates add, replace, or remove one block per proposal, while full-program proposals can change multiple blocks. Local updates use 59.2% less input over the complete runs, with mean final test AUC changing from 0.8046 to 0.8199. AUC improves on two of the four tasks. The runs average 14.25 rather than 20.25 scored trials. At the common depth of 12 trials, input remains 31.1% lower, consistent with the motivation to avoid repeatedly transmitting unchanged program content. These comparisons evaluate independently generated search trajectories. Detailed comparison settings and results appear in Appendix E.



(a) K: DB brief (b) P: Local updates

Figure 6: Each panel normalizes final cumulative input to its own baseline (100%). Input is measured at each run's stopping point.

5.5 RQ4: Generated SQL Programs

Two examples show how inspecting generated SQL reveals both the relational signals a feature captures and the information it requires at prediction time.

Counting distinct entities. One selected program counts visits and distinct advertisements within the preceding 12 days:

```
COUNT(CASE WHEN v.ViewDate >= e.timestamp - INTERVAL '12 days'
           THEN 1 END) AS visits_12d,
COUNT(DISTINCT CASE
  WHEN v.ViewDate >= e.timestamp - INTERVAL '12 days'
  THEN v.AdID END) AS ads_12d
```

The join also requires `v.ViewDate < e.timestamp`, so both counts use the same historical window. Six visits to one advertisement and six visits to different advertisements yield equal visit counts but distinct counts of one and six, separating frequency from browsing breadth. Retaining both features lets a predictor distinguish concentrated repeat activity from activity spread across multiple entities. Here, `ads_12d` ranks first in the saved surrogate-model SHAP diagnostic. The SQL makes the counted entity and time window explicit and inspectable.

Counting future events. An earlier run outside the controlled comparison counted future invitations:

```
SUM(CASE WHEN ea.start_time >= et.timestamp
          AND ea.start_time < et.timestamp + INTERVAL '7 days'
         THEN 1 ELSE 0 END) AS future_7d_invited
```

This window begins at the prediction timestamp and extends seven days into the future. In an audit, the count reconstructed all 19,239 cached training labels. These events can be present retrospectively for past training anchors while remaining unavailable when predicting future outcomes. A strong training signal can therefore disappear under validation or test access restrictions. The example motivates explicit control of the data exposed at these stages and shows why feature meaning must be considered together with information availability.

6 Conclusion

We introduced StepRel, which maintains database knowledge, SQL programs, and search evidence outside the conversation. Through local program updates and selective context reconstruction, it supports relational feature search without retaining the full interaction history. Experiments show competitive predictive performance with substantially lower context and token costs, highlighting the value of maintaining reusable search state.

AI USE STATEMENT

We used generative AI tools to assist with manuscript drafting and editing, literature search and summarization, implementation and debugging, analysis and interpretation of experimental results, and the preparation of code for figures and tables. We did not use these tools to generate synthetic datasets or prove mathematical claims. The authors reviewed and revised AI-assisted outputs, checked relevant references, and verified the implementation and reported results. The authors take full responsibility for the final content of this work, including all text, claims, analyses, and artifacts produced with AI assistance.

ETHICS STATEMENT

This work uses existing publicly available benchmark datasets whose sensitive identifying information was anonymized by the original providers. We did not collect additional personal data, recruit human participants, or attempt to re-identify individuals. Our experiments evaluate relational prediction methods within established benchmark settings and do not involve deployment in real-world decision-making. The execution controls described in the paper restrict generated SQL to the data permitted by the evaluation protocol; they should not be interpreted as a general guarantee of privacy or fairness in downstream applications.

REPRODUCIBILITY STATEMENT

The main text presents the STEPREL framework and search algorithm, while Appendices A and B provide further details on state management, execution protocols, and experimental configurations. These descriptions specify the data-access boundaries, search budgets, and evaluation choices underlying the experiments. Repeating an LLM-based search may produce different trajectories because of stochastic generation and changes to hosted model services. We will release the implementation and supporting analysis code upon publication to facilitate reproduction and further research.

REFERENCES

Nikhil Abhyankar, Parshin Shojaee, and Chandan K Reddy. Llm-fe: Automated feature engineering for tabular data with llms as evolutionary optimizers. *arXiv preprint arXiv:2503.14434*, 2025.

Jinze Bai, Jialin Wang, Zhao Li, Donghui Ding, Ji Zhang, and Jun Gao. Atj-net: Auto-table-join network for automatic learning on relational databases. In *Proceedings of the web conference 2021*, pp. 1540–1551, 2021.

Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. Creating embeddings of heterogeneous relational datasets for data integration tasks. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pp. 1335–1349, 2020.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. Mle-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*, volume 2025, pp. 50466–50494, 2025.

Tianlang Chen, Charilaos Kanatsoulis, and Jure Leskovec. Relgnn: Composite message passing for relational deep learning. *arXiv preprint arXiv:2502.06784*, 2025.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pp. 8746–8825, 2024.

Yaoqi Chen, Haibin Lai, Yuru Feng, Chuyu Han, Qianxi Zhang, Baotong Lu, Menghao Li, Xinjiang Wang, Zhirui Wang, Shusen Xu, et al. Beyond semantic organization: Memory as execution state management for long-horizon agents. *arXiv preprint arXiv:2606.06090*, 2026.

Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska, and David Karger. Arda: automatic relational data augmentation for machine learning. *arXiv preprint arXiv:2003.09758*, 2020.

Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Veličković. Principal neighbourhood aggregation for graph nets. *Advances in neural information processing systems*, 33:13260–13271, 2020.

Alexis Cvetkov-Iliev, Alexandre Allauzen, and Gaël Varoquaux. Relational data embeddings for feature enrichment with background information. *Machine Learning*, 112(2):687–720, 2023.

Milan Cvitkovic. Supervised learning on relational databases with graph neural networks. *arXiv preprint arXiv:2002.02046*, 2020.

Edoardo Debenedetti, Ilia Shumailov, Tianqi Fan, Jamie Hayes, Nicholas Carlini, Daniel Fabian, Christoph Kern, Chongyang Shi, Andreas Terzis, and Florian Tramèr. Defeating prompt injections by design. *arXiv preprint arXiv:2503.18813*, 2025.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Fengxian Dong, Zhi Zheng, Xiao Han, Wei Chen, Jingqing Ruan, Tong Xu, Yong Chen, and Enhong Chen. Memory-augmented llm-based multi-agent system for automated feature generation on tabular data. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10988–11007, 2026.

Vijay Prakash Dwivedi, Sri Jaladi, Yangyi Shen, Federico López, Charilaos I Kanatsoulis, Rishi Puri, Matthias Fey, and Jure Leskovec. Relational graph transformer. *arXiv preprint arXiv:2505.10960*, 2025.

Sašo Džeroski. Relational data mining. In *Data mining and knowledge discovery handbook*, pp. 887–911. Springer, 2010.

Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505*, 2020.

Yang Fei, Yangfan Jiang, Yin Yang, and Xiaokui Xiao. Benchmarking text-to-sql under role-based access control, 2026. URL https://arxiv.org/abs/2607.22115.

Matthias Fey, Weihua Hu, Kexin Huang, Jan Eric Lenssen, Rishabh Ranjan, Joshua Robinson, Rex Ying, Jiaxuan You, and Jure Leskovec. Position: Relational deep learning-graph representation learning on relational databases. In *Forty-first International Conference on Machine Learning*, 2024.

Matthias Fey, Vid Kocijan, Federico Lopez, J Lenssen, and Jure Leskovec. Kumorfm: A foundation model for in-context learning on relational data, 2025.

Léo Grinsztajn, Klemens Flöge, Oscar Key, Felix Birkel, Philipp Jund, Brendan Roof, Benjamin Jäger, Dominik Safaric, Simone Alessi, Adrian Hayler, et al. Tabpfn-2.5: Advancing the state of the art in tabular foundation models. *arXiv preprint arXiv:2511.08667*, 2025.

William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

Sungwon Han, Jinsung Yoon, Sercan O Arik, and Tomas Pfister. Large language models can automatically engineer features for few-shot tabular learning. *arXiv preprint arXiv:2404.09491*, 2024.

Stefan Hegselmann, Alejandro Buendia, Hunter Lang, Monica Agrawal, Xiaoyi Jiang, and David Sontag. Tabllm: Few-shot classification of tabular data with large language models. *arXiv preprint arXiv:2210.10723*, 2022.

Noah Hollmann, Samuel Müller, and Frank Hutter. Large language models for automated data science: Introducing caafe for context-aware automated feature engineering. *Advances in Neural Information Processing Systems*, 36:44753–44775, 2023.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, et al. Data interpreter: An llm agent for data science. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 19796–19821, 2025.

Weihua Hu, Yiwen Yuan, Zecheng Zhang, Akihiro Nitta, Kaidi Cao, Vid Kocijan, Jinu Sunil, Jure Leskovec, and Matthias Fey. Pytorch frame: A modular framework for multi-modal tabular learning. *arXiv preprint arXiv:2404.00776*, 2024.

Yuyang Hu, Hongjin Qian, Shuting Wang, Jiongnan Liu, Ziliang Zhao, Jiejun Tan, Zheng Liu, and Zhicheng Dou. Sam: State-adaptive memory for long-horizon reasoning agent. *arXiv preprint arXiv:2605.24468*, 2026.

Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous graph transformer. In *Proceedings of the web conference 2020*, pp. 2704–2710, 2020.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302*, 2023.

Xingyue Huang, Louis Tichelman, Jinwoo Kim, Krzysztof Olejniczak, and İsmail İlkan Ceylan. Relagent: Llm agents as data scientists for relational learning. *arXiv preprint arXiv:2605.07840*, 2026.

Valter Hudovernik, Federico López, Vid Kocijan, Akihiro Nitta, Jan Eric Lenssen, Jure Leskovec, and Matthias Fey. Kumorfm-2: Scaling foundation models for relational learning. *arXiv preprint arXiv:2604.12596*, 2026.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 13358–13376, 2023.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *2015 IEEE international conference on data science and advanced analytics (DSAA)*, pp. 1–10. IEEE, 2015.

Sayash Kapoor and Arvind Narayanan. Leakage and the reproducibility crisis in machine-learning-based science. *Patterns*, 4(9), 2023.

Shachar Kaufman, Saharon Rosset, and Claudia Perlich. Leakage in data mining: Formulation, detection, and avoidance. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 556–563, 2011.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30, 2017.

Kyungho Kim, Geon Lee, Juyeon Kim, Dongwon Choi, Shinhwan Kang, and Kijung Shin. Refuge: Feature generation for prediction tasks on relational databases with llm agents. In *Proceedings of the ACM Web Conference 2026*, pp. 8501–8504, 2026.

Jaris Küken, Lennart Purucker, and Frank Hutter. Large language models engineer too many simple features for tabular data. *arXiv preprint arXiv:2410.17787*, 2024.

Hoang Thanh Lam, Johann-Michael Thiebaut, Mathieu Sinn, Bei Chen, Tiep Mai, and Oznur Alkan. One button machine for automating feature engineering in relational databases. *arXiv preprint arXiv:1706.00327*, 2017.

Shukai Liu, Bo Jiang, Jian Yang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 20604–20617, 2026.

Joe Meyer, Tom Palczewski, Afreen Shaikh, Mahmoud Mohammadi, Dinesh Katupputhur Ramprasath, Karan Paresh, Roshan Reddy Upendra, and Mark Li. Relational in-context learning on structured data via neighborhood aggregation and structural information. In *Proceedings of the AAAI Symposium Series*, volume 9, pp. 219–220, 2026.

Jaehyun Nam, Kyuyoung Kim, Seunghyuk Oh, Jihoon Tack, Jaehyung Kim, and Jinwoo Shin. Optimized feature generation for tabular data via llms with decision tree reasoning. *Advances in neural information processing systems*, 37:92352–92380, 2024.

Jennifer Neville and David Jensen. Relational dependency networks. *Journal of Machine Learning Research*, 8(3), 2007.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pp. 26106–26128. PMLR, 2023.

OpenAI. GPT-5.6 Luna. OpenAI API Documentation, 2026. URL https://developers.openai.com/api/docs/models/gpt-5.6-luna. Accessed: 2026-07-29.

Charles Packer, Vivian Fang, Shishir_G Patil, Kevin Lin, Sarah Wooders, and Joseph_E Gonzalez. Memgpt: towards llms as operating systems. 2023.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pp. 1–22, 2023.

Danrui Qi, Weiling Zheng, and Jiannan Wang. Feataug: automatic feature augmentation from one-to-many relationship tables. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pp. 1805–1818. IEEE, 2024.

Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026. URL https://qwen.ai/blog?id=qwen3.6-35b-a3b.

Rishabh Ranjan, Valter Hudovernik, Mark Znidar, Charilaos Kanatsoulis, Roshan Upendra, Mahmoud Mohammadi, Joe Meyer, Tom Palczewski, Carlos Guestrin, and Jure Leskovec. Relational transformer: Toward zero-shot foundation models for relational data. *arXiv preprint arXiv:2510.06377*, 2025.

Petar Ristoski and Heiko Paulheim. Rdf2vec: Rdf graph embeddings for data mining. In *International semantic web conference*, pp. 498–514. Springer, 2016.

Joshua Robinson, Rishabh Ranjan, Weihua Hu, Kexin Huang, Jiaqi Han, Alejandro Dobles, Matthias Fey, Jan E Lenssen, Yiwen Yuan, Zecheng Zhang, et al. Relbench: A benchmark for deep learning on relational databases. *Advances in Neural Information Processing Systems*, 37:21330–21341, 2024.

Bastian Schäfer, Lennart Purucker, Maciej Janowski, and Frank Hutter. How usable is automated feature engineering for tabular data? *arXiv preprint arXiv:2508.13932*, 2025.

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

Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pp. 593–607. Springer, 2018.

Tianneng Shi, Jingxuan He, Zhun Wang, Linyu Wu, Hongwei Li, Wenbo Guo, and Dawn Song. Progent: Programmable privilege control for llm agents. *arXiv e-prints*, pp. arXiv–2504, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Shizuo Tian, Hao Wen, Yuxuan Chen, Jiacheng Liu, Shanhui Zhao, Guohong Liu, Ju Ren, Yunxin Liu, and Yuanchun Li. Agentprog: Empowering long-horizon gui agents with program-guided context management. In *Proceedings of the 24th Annual International Conference on Mobile Systems, Applications and Services*, pp. 883–910, 2026.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *stat*, 1050(20):10–48550, 2017.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023a.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 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)*, pp. 2609–2634, 2023b.

Minjie Wang, Quan Gan, David Wipf, Zhenkun Cai, Ning Li, Jianheng Tang, Yanlin Zhang, Zizhao Zhang, Zunyao Mao, Yakun Song, et al. 4dbinfer: A 4d benchmarking toolbox for graph-centric predictive modeling on relational dbs. *arXiv preprint arXiv:2404.18209*, 2024.

Yanbo Wang, Xiyuan Wang, Quan Gan, Minjie Wang, Qibin Yang, David Wipf, and Muhan Zhang. Griffin: Towards a graph-centric relational database foundation model. *arXiv preprint arXiv:2505.05568*, 2025.

Yanbo Wang, Jiaxuan You, Chuan Shi, and Muhan Zhang. Relational in-context learning via synthetic pre-training with structural prior. *arXiv preprint arXiv:2603.03805*, 2026.

Fang Wu, Vijay Prakash Dwivedi, and Jure Leskovec. Large language models are good relational learners. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7835–7854, 2025.

Marek Wydmuch, Łukasz Borchmann, and Filip Graliński. Tackling prediction tasks in relational databases with llms. *arXiv preprint arXiv:2411.11829*, 2024.

Linjie Xu, Yanlin Zhang, Quan Gan, Minjie Wang, and David Wipf. No need to train your rdb foundation model. *arXiv preprint arXiv:2602.13697*, 2026.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.

Yiwen Yuan, Zecheng Zhang, Xinwei He, Akihiro Nitta, Weihua Hu, Manan Shah, Blaz Stojanovic, Shenyang Andy Huang, Jan E Lenssen, Jure Leskovec, et al. Contextgnn: Beyond two-tower recommendation systems. In *International Conference on Learning Representations*, volume 2025, pp. 9822–9834, 2025.

Han Zhang, Quan Gan, David Wipf, and Weinan Zhang. Gfs: Graph-based feature synthesis for prediction over relational databases. *arXiv preprint arXiv:2312.02037*, 2023.

Jinhao Zhu, Kevin Tseng, Gil Vernik, Xiao Huang, Shishir G Patil, Vivian Fang, and Raluca Ada Popa. Miniscope: A least privilege framework for authorizing tool calling agents. *arXiv preprint arXiv:2512.11147*, 2025.

A Implementation Details

This section expands the implementation of the retained search state $\Sigma_t = (K, P_t, E_t)$ in Eq. equation 4. The database brief and current program persist across conversation resets. Candidate outcomes and diagnostics are organized into search evidence, from which selected records are rendered into the next checkpoint. Full tool responses, trial artifacts, and model interactions are saved separately for inspection; the selected evidence view is not a lossless copy of these logs.

A.1 Database Knowledge Construction

Before feature search, a read-only explorer receives the task definition and a database card containing schema, row counts, prediction keys, and candidate relation links. It uses targeted queries to examine relation coverage and fanout, temporal availability, column profiles, and unresolved questions. Name-based relation hints are distinguished from links supported by query results. The explorer produces a database brief describing the task contract, eligible schema, relation paths, temporal rules, column profiles, candidate feature families, cost warnings, and unresolved questions.

The main controlled runs use the checklist-v3 exploration protocol with a 32-tool-call budget and a target brief length of 6,000 tokens. The length is a generation target, not a bound on the subsequent search context. The explorer constructs database knowledge rather than evaluating candidate feature programs. Supporting SQL and the explorer's outputs are saved with the brief; the diagnostic Supporting SQL section is omitted from the search agent's view. Database identifiers and supporting queries are checked during brief preparation, and the corresponding records are retained with the artifact.

The brief is frozen before the main search and reused across checkpoints and repeated rollouts for the same task and exploration configuration. Task and database fingerprints, the database card, and the explorer protocol identify the reusable artifact; the construction budget and generation settings are recorded in its manifest. Rebuilding a conversation does not rerun the explorer or ask an LLM to summarize the brief again. Construction costs and per-rollout amortization are reported in Appendix D.

Knowledge acquired by the main agent after initialization is recorded in E_t, rather than modifying the frozen brief K. In particular, the cross-checkpoint database-query ledger described below is distinct from the pre-search exploration artifact.

A.2 Program Storage and Local Updates

Bootstrap proposes a complete SQL feature program. Once a finite validation result establishes an incumbent, later proposals specify one named block to add, replace, or remove, as described in Section 4.1. The Program Store represents the program as ordered named SQL blocks together with model and categorical-output metadata. Canonical serialization defines a program reference, and each block has a content hash over its name and SQL text. Replacement and removal include the expected identity of the old block, so that an edit referring to an outdated version can be detected.

For example, replacing a block named `recent_activity` supplies that block's new SQL and its expected old identity. The store loads the other blocks from P_t and assembles the complete candidate $\widetilde{P}_t$. The agent does not regenerate the unchanged blocks. The evaluator checks execution and output grain, constructs the feature matrix, and fits the fixed downstream model. The update in Eq. equation 5 accepts only a valid improvement in validation utility, using ROC–AUC for classification and negative MAE for regression. A failure or non-improving result leaves the incumbent unchanged while contributing a search record.

Program reuse and execution reuse are separate. The former carries SQL text forward when constructing a candidate. The latter uses a feature-query cache to reuse a complete block's materialized output when its SQL and execution inputs match. Cache identities include the database and evaluation-anchor fingerprints, entity restrictions, and cache version. A changed block is executed again, while unchanged blocks may reuse their outputs. This cache operates at whole-query granularity, without reusing individual expressions or subqueries. Program identity checks and cache-hit statistics are therefore different measurements.

A.3 Search Evidence Maintenance and Selection

Each candidate is associated with its parent program, proposed change, execution status, validation result, and available diagnostics. The evidence reducer organizes these records by program, block, and expression. Repeated programs accumulate attempt counts, related execution failures are grouped, and feature importance and missingness are attached to the relevant records. Non-improving successful alternatives remain available as historical attempts; a low importance value alone is not a rule for deleting a feature from the incumbent.

After evaluation, the agent receives a compact receipt with status, score, change from the incumbent, program identity, and selected diagnostics. Full evaluator outputs remain in the saved artifacts. Evidence maintenance and checkpoint selection are deterministic operations over these records.

For block evidence, the intermediate structured view retains incumbent blocks, their current feature records, and up to two alternative expression records per incumbent block. Optional historical blocks are considered in descending order of maximum importance share, attempt count, and best validation score, with stable identifiers breaking ties. Each optional block retains up to three alternative expression records. Alternative expressions are ordered by attribution type, attempt count, and importance. Optional blocks are added while the rendered intermediate state fits its configured 48,000-character budget. Required state is checked separately rather than silently dropping incumbent SQL to meet that budget.

The final checkpoint uses a shorter rendering of this structured view. Table 1 distinguishes the retained records from what is actually shown to the agent. Omission counts accompany the selected block records, so an omitted record is not presented as an unexplored feature.

Table 1: Evidence maintenance and final checkpoint selection in the default implementation. The complete incumbent is carried separately from these summaries. Limits apply to the indicated views, not to all persistent records.

Evidence	Retained information	Checkpoint selection
Block records	Incumbent membership, expression outcomes, attempt counts, importance, and references to trials	Incumbent-related records and selected historical blocks, with omission counts
Recent attempts	Trial outcomes and changes relative to the incumbent; the intermediate view keeps the latest five	The latest three transitions, including failures
Repeated programs and failures	Duplicate counts, hard-constraint records, and grouped errors	Up to eight duplicate or hard-constraint rules and three failure groups
Evaluation diagnostics	Trial digests, including five recent and three highest-scoring successful trials, plus on-demand analysis	Best/latest summaries, up to five important blocks, six block-missingness entries, and four recent on-demand records
Database queries	A deduplicated query ledger with returned feedback and attempt counts	First four and newest four entries, with limited SQL and result previews

The evaluation digest has a separate 12,000-character rendering budget. When necessary, it first drops on-demand result previews, then detailed recent trial diagnostics, and finally reduces entries to identities and scores. The final checkpoint further shortens these diagnostics as summarized in Table 1. These stages do not remove the underlying trial outputs from storage.

Database-query evidence uses its own selector. Let $D_t = (d_1, \ldots, d_n)$ be the ordered ledger of unique targeted `execute_query` records from the main search. SQL is normalized for deduplication, and a repeated query updates its existing feedback and call count without changing its original position. The default selection is

$$\mathcal{R}_{4+4}(D_t) = \begin{cases} (d_1, \ldots, d_n), & n \leq 8, \\ (d_1, \ldots, d_4, d_{n-3}, \ldots, d_n), & n > 8. \end{cases} \tag{6}$$

Each selected entry displays at most 600 characters of SQL and 1,000 characters of result text. Middle entries remain in the ledger and selection is recomputed from that ledger at the next checkpoint. The recorder also retains returned error feedback where present; ledger membership does not by itself certify successful query execution. Schema-tool responses are available in the live conversation but are not entries in this cross-checkpoint query ledger. The first-four/newest-four rule does not apply to general candidate evidence or evaluation diagnostics.

A.4 Context Reconstruction and Continued Search

After every six scored trials, the runner prepares the checkpoint Q_t described in Section 4.3. It retains the task contract and frozen database brief as system context and renders the full incumbent, search-budget status, selected block and trial evidence, evaluation summaries, and database-query

evidence into the continuation message. The previous non-system interaction is replaced by this new context.

Before applying the checkpoint, the runner checks that its incumbent agrees with the Program Store and that the required state passes reconstruction checks. While a checkpoint is pending, wrapped tools stop further actions until the outer loop performs the reset. Search then resumes from the same incumbent, with the next local edit applied to that stored program. Failed and non-finite evaluations contribute evidence but do not advance the six-scored-trial counter. Resetting the conversation does not reset the overall search budget or delete the persistent records.

Checkpoint construction uses deterministic rendering without an additional LLM call. Its text contributes input tokens when used in a subsequent model request. The evidence views have selection and length limits, but the full incumbent can grow as blocks are added. Consequently, the complete checkpoint is not assumed to have a fixed size. What is removed is the accumulated sequence of previous conversations and candidate programs, while the current program and selected search experience remain available.

The execution boundaries shared by the compared systems are specified in Appendix B, rather than being part of the context reconstruction mechanism.

B Experimental Setup

B.1 Tasks and Model Configurations

We use the official splits of ten binary-classification and eight regression RelBench v1 tasks (Robinson et al., 2024). Test ROC–AUC measures classification performance, and test MAE measures regression performance. We exclude `rel-f1` following concerns reported in prior work (Xu et al., 2026). Each agentic method is evaluated with DeepSeek-V4-Pro, Qwen3.6-35B-A3B, Qwen3-Coder-30B-A3B-Instruct, and GPT-5.6 Luna (DeepSeek-AI, 2026; Qwen Team, 2026; Team, 2025; OpenAI, 2026). The planned replication is three stochastic rollouts per task, method, and backbone. Results use the available finite endpoints, with endpoint coverage reported separately.

Table 2: Evaluation settings. The blockwise search budgets apply to StepRel and controlled RelAgent; ReFuGe retains its native staged budget. Minimum counts specify the stopping policy rather than guaranteeing completion of every run.

Setting	Configuration
Tasks and metrics	10 classification tasks (ROC–AUC), 8 regression tasks (MAE)
Agentic methods	StepRel, controlled RelAgent, ReFuGe
Replication	Three requested rollouts per task, method, and backbone
Downstream evaluator	CatBoost: 200 trees, learning rate 0.05, depth 6, $L_2 = 3$
Main blockwise search	First stopping-policy check at 18 scored trials, maximum 42; checks every six trials
New component runs	Minimum 12 scored trials, maximum 42; decisions every six trials
StepRel checkpoint interval	Six scored trials

B.2 Comparison Methods and Shared Settings

The primary system comparison uses the same backbone, task splits, data-access boundaries, and fixed CatBoost evaluator. StepRel reuses a database brief, proposes local block edits, and reconstructs its conversation from retained state. Controlled RelAgent explores the database on demand, proposes complete programs, and retains its interaction history. They use the same six-trial stopping framework in the main comparison. Implementation details for StepRel appear in Appendix A.

ReFuGe uses the shared data-access and evaluator settings while retaining its native stages for schema selection, planning, feature generation, and filtering. Its lower validation budget is a separate operating point, so it is included in predictive comparisons but not treated as a matched-depth alternative in RQ2.

Published benchmark results retain their original training, tuning, and selection protocols. In that supplementary comparison, reported RelAgent results are distinct from our controlled RelAgent runs. These results provide broader predictive context; the shared-protocol agentic comparison is the basis for assessing the complete search systems. The targeted component experiments use their own task–seed intersections and the configuration contrasts specified in Appendix E.

B.3 Validation and Test Execution

Search separates three information domains: $\mathcal{V}$, the validation-as-of feature snapshot; $\mathcal{T}$, the test-as-of feature snapshot; and $\mathcal{Y}$, private task labels. Database exploration, SQL generation, and validation-driven selection can access only $\mathcal{V}$. Candidate queries are driven by temporary target-free anchors containing the entity and time keys needed for alignment, while labels in $\mathcal{Y}$ are joined only inside the evaluator.

After search terminates, the selected program is canonicalized and persisted. The runner requires the in-memory and persisted identities to agree, writes a freeze receipt containing the program identity and temporal policy, and only then opens $\mathcal{T}$. The exact frozen program is executed without another proposal, model response, or validation-based decision. The test record carries the same program identity, making the freeze-to-test chain checkable.

This contract establishes target isolation, split-level temporal censoring, validation-only selection, and freeze-before-test ordering. Row-local point-in-time semantics additionally depend on the task's timestamp granularity and on how stored event times represent real information availability.

Saved execution checks. For the controlled `rel-avito/user-visits` example in Appendix F, the frozen-program receipt, frozen-feature manifest, and run manifest record the same canonical program identity. Both snapshot manifests hide the target table, disable external access, and record source timestamps no later than their split cutoffs. The freeze receipt precedes test-snapshot verification, and the test record contains exactly the saved selected program. The associated checkpoint reconstruction receipt passes all 15 recorded checks. These records verify the execution sequence for this audited run.

Physical snapshots also constrain nested queries. In a separate legacy RelAgent run on `rel-amazon/item-churn`, an outer review join obeyed the anchor-time bound, but a nested per-customer review count scanned the full relation. All 30 sampled anchor outputs changed when replayed against an as-of snapshot. This case illustrates why checking an outer time predicate alone is insufficient and why the split cutoff applies to the underlying tables.

B.4 Search Budgets and Stopping

A scored trial is a candidate evaluation that returns a finite validation metric without an execution error. We use ROC–AUC as the higher-is-better validation utility for classification and negative MAE for regression. Failed or non-finite evaluations are recorded but do not advance the scored trial counter.

The main system and matched-depth efficiency runs use the recorded `chunked-6-patience-v2` profile. Starting at 18 scored trials, continuation is checked at $n \in \{18, 24, 30, 36\}$. Let b_n be the best validation utility after n scored trials. The latest six-trial block makes a meaningful improvement when

$$b_n - b_{n-6} > 0.001. \tag{7}$$

The threshold is an absolute change in ROC–AUC or negative MAE, not a relative percentage. At the first three decision points, search stops after two consecutive checked blocks without a meaningful improvement; an improving block resets this counter. At the soft boundary of 36 trials, continuation requires improvement in the latest block. Search stops at the hard maximum of 42 scored trials.

The new component runs use `chunked-6-v1`, with checks at $n \in \{12, 18, 24, 30, 36\}$ and the same improvement threshold and hard maximum. They stop at the first checked block without a meaningful improvement. Thus, the main and component experiments share the six-trial decision unit but have different minimum budgets and patience rules. Comparisons against historical system runs are identified as such in the component analysis.

The configured failed-evaluation limit is six. Runtime or service failures can also leave a run incomplete, so scheduled budgets are distinguished from observed search lengths. The stopping decisions above depend on validation progress, not elapsed time or the number of model requests. STEPREL reconstructs context every six scored trials without resetting these counters.

B.5 PROGRAM SELECTION AND RESULT AGGREGATION

A trial evaluates one candidate; a rollout is one independent search; and a result cell groups rollouts for a task, method, and backbone. Each rollout selects its program by validation utility and freezes it before test execution. The primary predictive summary averages the finite test results of these selected programs within each cell. It does not choose the best rollout using test performance. A supplementary sensitivity analysis instead selects among available rollouts by validation utility, with deterministic tie-breaking, and reports that rollout's test result.

Predictive ranks. Figure 3 ranks the twelve agent–backbone configurations on the common eight classification and eight regression tasks with observed results for every configuration. Within a task, ranks use decreasing mean ROC–AUC or increasing mean MAE, with average ranks for ties. We average task ranks within each task type, then give classification and regression equal weight. Any supplementary ranking of only the three agents within a fixed backbone uses a different comparison pool and is identified separately.

Rollout variability and coverage. Instability uses task-normalized pairwise differences between finite rollout test results, transformed as $D/(1 + D)$ before aggregation. Cells require at least two finite rollouts, so their eligible task sets can differ from the common-task ranking cohort. Task averages are computed within classification and regression and then combined with equal weight. Coverage separately records tasks and rollouts with finite endpoints. Missing results are not filled using another method's score, and missing rollouts do not receive an artificial variability penalty. Appendix C gives the normalization and task-level results.

Cost and component comparisons. RQ2 pairs runs by task and replicate within each backbone and requires both runs to reach the reported scored-trial boundary. Cumulative model-token counts include input, output, and one third of the task-level database-brief construction cost for STEPREL. RQ3 instead reports main-search input for its paired component contrasts, excluding separate brief construction. Within each contrast, seeds are averaged within tasks and tasks receive equal weight. Final-run totals reflect realized stopping points, while trial-12 measurements compare a common search depth. Detailed accounting appears in Appendices D and E.

C COMPLETE PREDICTIVE RESULTS (RQ1)

This section expands the controlled comparison in RQ1 with task-level rollout means, predictive ranks, empirical variability, and endpoint coverage. We then examine sensitivity to rollout selection and provide the broader benchmark tables as supplementary context.

C.1 CONTROLLED PREDICTIVE PERFORMANCE AND STABILITY

Predictive performance. Within each rollout, the program is selected by validation score. Our primary summary averages the available finite test results across rollouts for each task and agent–backbone configuration. We rank the twelve configurations within each task, using decreasing ROC–AUC for classification and increasing MAE for regression, with average ranks for ties. The common comparison set contains eight classification and eight regression tasks with a finite result for every configuration. Classification and regression ranks are averaged separately, and their unweighted mean gives the overall rank used in the main paper.

Table 3 provides the numerical breakdown. STEPREL has a better overall mean rank than RelAgent with each of the four backbones, while the size of the difference varies across backbones. The Qwen3-Coder gap is particularly pronounced, with overall ranks of 6.50 and 8.94 for STEPREL and RelAgent, respectively. These ranks describe performance on the common task set. Coverage over all eighteen tasks is examined separately below.

Table 3: Controlled rollout-mean ranks across twelve agent–backbone configurations on eight common tasks per task type. Overall gives equal weight to classification and regression. Lower is better.

Configuration	Cls. rank	n_{cls}	Reg. rank	n_{reg}	Overall
StepRel–DeepSeek	2.375	8	2.875	8	2.625
StepRel–GPT-5.6-Luna	3.000	8	2.750	8	2.875
RelAgent–DeepSeek	2.375	8	3.750	8	3.062
RelAgent–GPT-5.6-Luna	3.625	8	3.250	8	3.438
StepRel–Qwen3.6	4.625	8	4.500	8	4.562
RelAgent–Qwen3.6	5.750	8	5.750	8	5.750
StepRel–Qwen3-Coder	6.625	8	6.375	8	6.500
RelAgent–Qwen3-Coder	10.500	8	7.375	8	8.938
ReFuGe–GPT-5.6-Luna	9.125	8	9.375	8	9.250
ReFuGe–DeepSeek	9.500	8	10.125	8	9.812
ReFuGe–Qwen3-Coder	10.000	8	11.125	8	10.562
ReFuGe–Qwen3.6	10.500	8	10.750	8	10.625

Empirical stability. To describe variation across repeated searches, we orient each task metric so that larger is better and normalize pairwise rollout differences by a robust task scale:

$$z_{jtr} = \frac{s_t x_{jtr} - c_t}{q_t}, \quad D_{jt} = \frac{1}{\binom{n_{jt}}{2}} \sum_{r<s} |z_{jtr} - z_{jts}|, \quad B_{jt} = \frac{D_{jt}}{1 + D_{jt}}, \tag{8}$$

where x_{jtr} is the test result of configuration j on task t in rollout r, and n_{jt} counts its finite rollout results. We set $s_t = 1$ for classification and -1 for regression. The center c_t and scale q_t are the median and interquartile range of the oriented configuration-level rollout means for task t. Lower B_{jt} denotes less rollout variation. The monotone transformation bounds the score and limits the influence of extreme task cells.

Stability is computed only for task cells with at least two finite rollouts. We average these scores within each task type and give the two task types equal weight in the overall summary. Unlike the common task set used for ranking, the stability task set can vary by configuration. Missing rollouts receive no variability penalty, so stability and endpoint coverage should be read together.

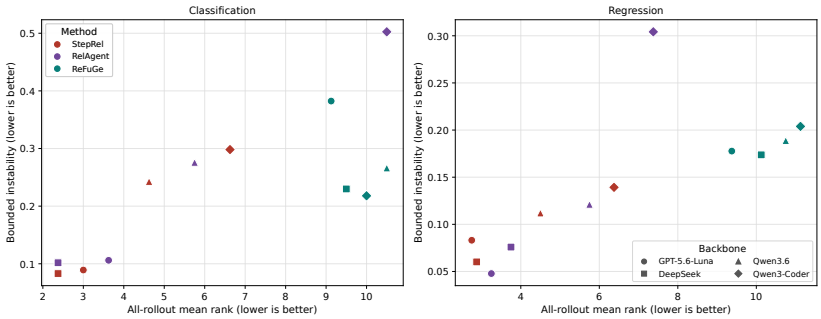


Figure 7: Rollout-mean performance and empirical instability by task type. Ranks use the common task set; instability uses task cells with at least two finite rollouts. Lower is better on both axes.

Figure 7 separates the two task types behind the main paper's overall summary. StepRel has lower overall instability than RelAgent with three of the four backbones. GPT-5.6-Luna is the exception, so improved mean rank does not imply uniformly lower variability. These are descriptive summaries of the observed rollouts, not statistical significance tests.

C.2 Task-level Rollout Means

Tables 4 and 5 report test ROC–AUC, while Tables 6 and 7 report test MAE. Each entry gives the mean and sample standard deviation of the available finite rollout results, with the program selected

by validation within each rollout. The common-task subset of these means produces the ranks in Figure 3 and Table 3.

The tables retain all evaluated tasks, marking those outside the common ranking set. Standard deviations are in the original task metric, complementing the normalized instability analysis above. These tables use rollout means, whereas the supplementary benchmark comparison in Section C.5 uses validation-selected rollouts.

Table 4: Controlled rollout-mean classification results (test ROC–AUC, higher is better) with DeepSeek-V4-Pro and GPT-5.6 Luna.

Task	STEPREL	RelAgent	ReFuGe
DeepSeek-V4-Pro			
Amazon / item-churn	**0.8205** ± 0.0073	0.8201 ± 0.0054	0.7914 ± 0.0053
Amazon / user-churn	0.7000 ± 0.0012	**0.7015** ± 0.0009	0.6619 ± 0.0017
Avito / user-clicks	**0.6860** ± 0.0042	0.6843 ± 0.0022	0.5922 ± 0.0108
Avito / user-visits	0.6743 ± 0.0013	**0.6757** ± 0.0020	0.6661 ± 0.0008
Event / user-ignore†	**0.8407** ± 0.0161	0.8337 ± 0.0187	0.7297 ± 0.0515
Event / user-repeat†	0.7777 ± 0.0028	**0.7900** ± 0.0160	0.7028 ± 0.0270
H&M / user-churn	**0.7057** ± 0.0007	0.7056 ± 0.0007	0.6816 ± 0.0033
Stack / user-badge	**0.8797** ± 0.0041	0.8790 ± 0.0022	0.8039 ± 0.0171
Stack / user-engagement	0.9008 ± 0.0007	**0.9032** ± 0.0023	0.8616 ± 0.0465
Trial / study-outcome	**0.7226** ± 0.0005	0.7208 ± 0.0043	0.6386 ± 0.0119
GPT-5.6 Luna			
Amazon / item-churn	**0.8249** ± 0.0002	0.8240 ± 0.0005	0.7935 ± 0.0089
Amazon / user-churn	**0.7061** ± 0.0006	0.7013 ± 0.0022	0.6167 ± 0.0466
Avito / user-clicks	**0.6778** ± 0.0023	0.6642 ± 0.0128	0.6059 ± 0.0175
Avito / user-visits	0.6743 ± 0.0017	**0.6755** ± 0.0008	0.6539 ± 0.0119
Event / user-ignore†	0.8172 ± 0.0256	**0.8557** ± 0.0206	0.7754 ± 0.1191
Event / user-repeat†	0.8040 ± 0.0107	**0.8052** ± 0.0174	0.7297 ± 0.0534
H&M / user-churn	0.7055 ± 0.0012	**0.7055** ± 0.0006	0.6990 ± 0.0020
Stack / user-badge	**0.8750** ± 0.0020	0.8723 ± 0.0033	0.7942 ± 0.0255
Stack / user-engagement	**0.9012** ± 0.0012	0.8989 ± 0.0009	0.8352 ± 0.0615
Trial / study-outcome	0.7199 ± 0.0077	**0.7221** ± 0.0064	0.6446 ± 0.0089

Entries are mean ± sample standard deviation across finite rollouts, with program selection by validation within each rollout. A dash denotes no finite result, and a singleton has no reported standard deviation. Bold marks the best unrounded mean within each task–backbone comparison. RelAgent uses the controlled setting; ReFuGe retains its lower search budget. † marks tasks excluded from the common-task ranks because RelAgent with Qwen3-Coder has no finite result.

Table 5: Controlled rollout-mean classification results (test ROC–AUC, higher is better) with Qwen3.6 and Qwen3-Coder-30B.

Task	StepRel	RelAgent	ReFuGe
Qwen3.6			
Amazon / item-churn	**0.8199** ± 0.0070	0.8153 ± 0.0058	0.7737 ± 0.0228
Amazon / user-churn	**0.7040** ± 0.0014	0.6917 ± 0.0019	0.6505 ± 0.0136
Avito / user-clicks	**0.6851** ± 0.0057	0.6750 ± 0.0145	0.5734 ± 0.0098
Avito / user-visits	0.6693 ± 0.0078	**0.6756** ± 0.0012	0.6470 ± 0.0173
Event / user-ignore†	**0.8492** ± 0.0428	0.8375 ± 0.0349	0.7420 ± 0.0048
Event / user-repeat†	0.7378 ± 0.0167	**0.7664** ± 0.0337	0.6544 ± 0.0120
H&M / user-churn	0.6920 ± 0.0222	**0.6947** ± 0.0088	0.6775 ± 0.0031
Stack / user-badge	**0.8735** ± 0.0009	0.8460 ± 0.0220	0.8053 ± 0.0051
Stack / user-engagement	**0.8823** ± 0.0177	0.8738 ± 0.0207	0.8022 ± 0.0498
Trial / study-outcome	**0.7115** ± 0.0145	0.6774 ± 0.0688	0.6436 ± 0.0115
Qwen3-Coder-30B			
Amazon / item-churn	**0.8115** ± 0.0033	0.6086 ± 0.1716	0.7649 ± 0.0292
Amazon / user-churn	**0.6757** ± 0.0215	0.5523 ± 0.0811	0.6224 ± 0.0000
Avito / user-clicks	**0.6520** ± 0.0365	0.6507 ± 0.0206	0.6340 ± 0.0050
Avito / user-visits	**0.6691** ± 0.0019	0.5928 ± 0.0804	0.6658 ± 0.0006
Event / user-ignore†	**0.7714** ± 0.0467	–	0.6840 ± 0.0474
Event / user-repeat†	0.6510 ± 0.0426	–	**0.6885** ± 0.0395
H&M / user-churn	**0.6980** ± 0.0073	0.6839 ± 0.0033	0.6880 ± 0.0110
Stack / user-badge	**0.8344** ± 0.0099	0.8233 ± 0.0042	0.8003 ± 0.0029
Stack / user-engagement	**0.8787** ± 0.0357	0.8341 ± 0.0383	0.8118 ± 0.0064
Trial / study-outcome	**0.6938** ± 0.0232	0.5621 ± 0.1058	0.6433 ± 0.0082

Entries are mean ± sample standard deviation across finite rollouts, with program selection by validation within each rollout. A dash denotes no finite result, and a singleton has no reported standard deviation. Bold marks the best unrounded mean within each task–backbone comparison. RelAgent uses the controlled setting; ReFuGe retains its lower search budget. † marks tasks excluded from the common-task ranks because RelAgent with Qwen3-Coder has no finite result.

Table 6: Controlled rollout-mean regression results (test MAE, lower is better) with DeepSeek-V4-Pro and GPT-5.6 Luna.

Task	STEPREL	RelAgent	ReFuGe
DeepSeek-V4-Pro			
Amazon / item-ltv	**43.650** ± 0.620	44.343 ± 0.649	52.534 ± 0.652
Amazon / user-ltv	**13.935** ± 0.044	14.031 ± 0.054	18.598 ± 0.214
Avito / ad-ctr	**0.0342** ± 0.0004	0.0344 ± 0.0006	0.0410 ± 0.0003
Event / user-attendance	**0.2470** ± 0.0008	0.2478 ± 0.0042	0.3838 ± 0.0635
H&M / item-sales	**0.0373** ± 0.0003	0.0378 ± 0.0005	0.1029 ± 0.0069
Stack / post-votes	0.0680 ± 0.0003	**0.0653** ± 0.0014	0.1344 ± 0.0004
Trial / site-success	0.3900 ± 0.0101	**0.3732** ± 0.0058	0.4480 ± 0.0158
Trial / study-adverse	**44.053** ± 0.938	44.581 ± 1.728	60.312 ± 4.984
GPT-5.6 Luna			
Amazon / item-ltv	43.734 ± 0.362	**43.676** ± 0.143	54.763 ± 1.261
Amazon / user-ltv	**13.841** ± 0.009	13.930 ± 0.121	19.912 ± 2.025
Avito / ad-ctr	**0.0342** ± 0.0002	0.0345 ± 0.0006	0.0361 ± 0.0007
Event / user-attendance	0.2602 ± 0.0118	**0.2482** ± 0.0015	0.3445 ± 0.0058
H&M / item-sales	**0.0372** ± 0.0005	0.0377 ± 0.0007	0.0975 ± 0.0005
Stack / post-votes	**0.0633** ± 0.0001	0.0634 ± 0.0000	0.1184 ± 0.0119
Trial / site-success	0.4037 ± 0.0230	**0.3818** ± 0.0060	0.4529 ± 0.0135
Trial / study-adverse	**44.504** ± 0.806	46.424 ± 0.448	55.109 ± 2.518

Entries are mean ± sample standard deviation across finite rollouts, with program selection by validation within each rollout. A dash denotes no finite result, and a singleton has no reported standard deviation. Bold marks the best unrounded mean within each task–backbone comparison. RelAgent uses the controlled setting; ReFuGe retains its lower search budget.

Table 7: Controlled rollout-mean regression results (test MAE, lower is better) with Qwen3.6 and Qwen3-Coder-30B.

Task	STEPREL	RelAgent	ReFuGe
Qwen3.6			
Amazon / item-ltv	**44.216** ± 0.247	44.433 ± 0.536	59.419 ± 6.282
Amazon / user-ltv	**14.089** ± 0.079	14.350 ± 0.292	19.988 ± 0.610
Avito / ad-ctr	**0.0349** ± 0.0011	0.0365 ± 0.0028	0.0420 ± 0.0012
Event / user-attendance	**0.2486** ± 0.0034	0.2508 ± 0.0056	0.4491 ± 0.0123
H&M / item-sales	**0.0376** ± 0.0007	0.0385 ± 0.0010	0.0988 ± 0.0011
Stack / post-votes	**0.0645** ± 0.0002	0.0658 ± 0.0005	0.1096 ± 0.0003
Trial / site-success	0.4236 ± 0.0296	**0.3678** ± 0.0203	0.4403 ± 0.0121
Trial / study-adverse	**46.253** ± 1.559	49.442 ± 0.388	60.562 ± 3.808
Qwen3-Coder-30B			
Amazon / item-ltv	**45.370** ± 1.202	55.578 ± 17.354	65.146 ± 0.420
Amazon / user-ltv	**14.273** ± 0.123	16.636 ± 1.735	19.137 ± 1.125
Avito / ad-ctr	**0.0350** ± 0.0006	0.0390 ± 0.0022	0.0432 ± 0.0049
Event / user-attendance	0.2519 ± 0.0074	**0.2407**	0.4560 ± 0.0066
H&M / item-sales	**0.0393** ± 0.0004	0.0505 ± 0.0001	0.1075 ± 0.0088
Stack / post-votes	**0.0638** ± 0.0002	0.0676 ± 0.0006	0.1092 ± 0.0002
Trial / site-success	0.4380 ± 0.0299	**0.4311** ± 0.0542	0.4556 ± 0.0152
Trial / study-adverse	**52.687** ± 3.478	56.163 ± 2.649	57.311 ± 4.829

Entries are mean ± sample standard deviation across finite rollouts, with program selection by validation within each rollout. A dash denotes no finite result, and a singleton has no reported standard deviation. Bold marks the best unrounded mean within each task–backbone comparison. RelAgent uses the controlled setting; ReFuGe retains its lower search budget.

C.3 Endpoint coverage and the Qwen3-Coder case

Predictive rank measures the quality of finite results, while endpoint coverage records whether a search returned such a result. This distinction is particularly relevant for Qwen3-Coder.

Table 8: Observed Qwen3-Coder coverage. "Task cells" counts tasks with at least one finite validation-selected endpoint; "Endpoints" counts finite seed endpoints out of the three planned per task. Each rollout selects its program by validation score.

	Classification		Regression	
Method	Task cells	Endpoints	Task cells	Endpoints
StepRel	10/10	30/30	8/8	24/24
RelAgent	8/10	23/30	8/8	22/24
ReFuGe	10/10	30/30	8/8	24/24

RelAgent produced no finite endpoint on `rel-event/user-ignore` or `rel-event/user-repeat`. Across the associated unsuccessful searches, 154 trials ended with $-\infty$: 104 because all features were constant or ignored, 44 because no feature column remained after merging, and 6 because merge keys had incompatible types. In 21 of these trials, the proposed SQL also contained a self-comparison such as `timestamp < timestamp`. This is an overlapping SQL diagnostic, not an additional failure category.

StepRel and ReFuGe returned finite endpoints for every Qwen3-Coder rollout, whereas RelAgent returned 45 of 54. Together with the rank gap, these observations show that RelAgent's difficulty with this backbone involves both predictive quality and search completion. The failure analysis does not isolate which component of StepRel accounts for the coverage difference.

C.4 Sensitivity to rollout aggregation

As a secondary summary, we select the rollout with the best validation score and report that rollout's test result. This adds selection across rollouts to the validation-based program selection already performed within each rollout. Figure 8 compares this summary with our primary rollout mean on the same common task set and with the same twelve-configuration ranking pool.

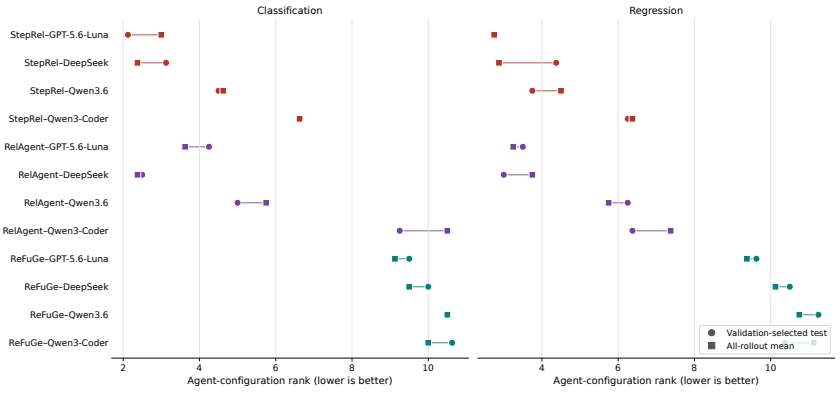


Figure 8: Sensitivity to rollout aggregation. Squares show the primary rollout-mean ranks; circles show ranks after selecting a rollout by validation score. Lower is better.

The ordering depends on aggregation. For example, RelAgent–DeepSeek has a better overall rank than StepRel–DeepSeek under rollout selection, while StepRel leads under the rollout mean. We therefore base the main predictive comparison on mean performance and report selection sensitivity separately.

C.5 SUPPLEMENTARY BENCHMARK COMPARISON

Tables 9 and 10 provide broader predictive context in two separate blocks. *Published results* retain the source methods' training, tuning, and selection protocols. *Results under our experimental setting* contain STEPREL, controlled RelAgent, and ReFuGe, using the settings in Appendix B. For this supplementary comparison, these rows report the test result of the rollout selected by validation score among the three requested rollouts. The primary RQ1 analysis above instead uses rollout means.

Reported and controlled RelAgent. The *RelAgent (reported)* row reproduces the corresponding task values from Tables 2 and 5 of Huang et al. (2026). That configuration uses GPT-5.2, selects among five independent rollouts by validation score, and searches over downstream model families and hyperparameters. Its final predictor is refitted on training and validation data. Our controlled RelAgent runs use the four backbones and fixed CatBoost evaluator described in Appendix B, with three requested rollouts and shared data-access boundaries. The two sets of results therefore represent different experimental protocols, and their difference cannot be attributed to any one of these choices.

Bold marks the best displayed value within each of the two blocks, and a dash denotes a missing finite endpoint. We report task-level values without a pooled rank across protocols. The published results situate the generated features among a broader range of relational learning methods, while the controlled comparison supports the main conclusions of RQ1.

Table 9: Supplementary classification comparison (ROC–AUC; higher is better). Published results and results under our setting are grouped separately. Bold marks the best value within each block. Our rows select among three requested rollouts by validation score; reported RelAgent selects among five. Dashes denote missing finite endpoints.

Model / configuration	Amazon item churn	Amazon user churn	Avito user clicks	Avito user visits	Event user ignore	Event user repeat	H&M user churn	Stack user badge	Stack user engage.	Trial study outcome
Published results										
Hybrid tabular-relational pipelines										
LightGBM	0.6254	0.5222	0.5360	0.5305	0.7993	0.6804	0.5521	0.6343	0.6339	0.7009
DS+LightGBM	0.8180	0.6760	0.6427	0.6446	0.8423	0.7542	0.6900	0.8620	0.9030	0.7200
DS+AutoGluon	0.8045	0.6812	0.6566	0.6587	0.8231	0.7485	0.6871	0.8538	0.8994	0.7051
DS+TabPFN-2.5	0.7983	0.6628	0.6421	0.6453	0.8570	0.7643	0.6831	0.8508	0.8907	0.7075
GNN+TabPFN	0.7789	0.6430	0.6068	0.6535	0.8413	0.7358	0.6434	0.8534	0.8748	0.5668
RDBLearn	0.8207	0.6757	**0.6904**	0.6549	0.8252	0.7504	0.6805	0.8526	0.8939	0.7158
Supervised relational deep learning										
GraphSAGE	0.8281	0.7042	0.6590	0.6620	0.8162	0.7689	0.6988	0.8886	0.9059	0.6860
HGT	0.7797	0.6643	0.6376	0.6432	0.8247	0.6496	0.6695	0.8608	0.8847	0.5837
HGTPE	0.7803	0.6619	0.6457	0.6495	0.8161	0.6536	0.6569	0.8566	0.8817	0.5921
RelGNN	0.8264	**0.7099**	0.6823	0.6618	0.8618	0.7961	0.7093	**0.8898**	**0.9075**	0.7124
RelGT	0.8255	0.7039	0.6830	0.6678	0.8157	0.7609	0.6927	0.8632	0.9053	0.6861
LM relational predictors										
LLM1	0.6210	0.5810	0.5980	0.6270	0.6930	0.7140	0.5980	0.8000	0.7800	0.5740
LLM2	0.7341	0.6255	0.5336	0.5407	0.6865	0.7011	0.6381	0.7999	0.8123	0.5917
Relational foundation model										
Rel-Zero	0.6410	0.6007	0.6228	0.5617	0.6132	0.6812	0.5595	0.6212	0.6946	0.5902
TabPFN-2.5	0.5930	0.5714	0.5444	0.5145	0.7091	0.6910	0.5499	0.5226	0.5966	0.7174
Griffin	0.6900	0.6230	0.4590	0.6070	0.8327	0.7188	0.6020	0.7350	0.7750	0.5100
RTzero	0.7090	0.6400	0.5950	0.6180	0.7747	0.7322	0.6280	0.8010	0.7570	0.5180
KumoRFM-v1	0.7993	0.6729	0.6485	0.6411	0.8920	0.7608	0.6771	0.8000	0.8709	0.7079
KumoRFM-v2	0.8217	0.6910	0.6742	**0.6941**	**0.9083**	**0.8166**	0.6927	0.8715	0.8940	**0.7203**
RDB-PFN	0.7959	0.6592	0.6446	0.6582	0.8343	0.7471	0.6721	0.8452	0.8804	0.6450
Agentic search										
RelAgent (reported)	**0.8284**	0.7078	0.6836	0.6779	0.8725	0.7820	**0.7107**	0.8842	0.9041	0.7186
Results under our experimental setting										
StepRel										
StepRel–DeepSeek	**0.8249**	0.7012	0.6828	0.6752	0.8450	0.7793	0.7062	**0.8841**	0.9016	0.7232
StepRel–Qwen3.6	0.8246	0.7057	0.6788	0.6731	0.8253	0.7232	0.7045	0.8744	0.8999	0.7203
StepRel–Qwen3-Coder	0.8137	0.6936	0.6745	0.6694	0.7994	0.6037	0.7027	0.8403	0.8994	0.7146
StepRel–GPT-5.6-Luna	0.8238	**0.7068**	0.6800	0.6759	0.8282	**0.8152**	**0.7065**	0.8769	0.9017	0.7243
RelAgent (controlled)										
RelAgent–DeepSeek	0.8234	0.7019	0.6860	**0.6775**	0.8286	0.7999	0.7052	0.8807	**0.9039**	0.7250
RelAgent–Qwen3.6	0.8205	0.6937	**0.6901**	0.6768	0.7943	0.7993	0.6962	0.8612	0.8961	0.7073
RelAgent–Qwen3-Coder	0.8066	0.6460	0.6740	0.6410	–	–	0.6871	0.8262	0.8765	0.6843
RelAgent–GPT-5.6-Luna	0.8246	0.7029	0.6535	0.6757	0.8336	0.8051	0.7062	0.8761	0.9000	**0.7327**
ReFuGe										
ReFuGe–DeepSeek	0.7975	0.6639	0.6043	0.6668	0.7870	0.6977	0.6850	0.8207	0.8899	0.6286
ReFuGe–Qwen3.6	0.8000	0.6635	0.5682	0.6670	0.7370	0.6422	0.6791	0.8017	0.8428	0.6518
ReFuGe–Qwen3-Coder	0.7935	0.6224	0.6291	0.6661	0.7386	0.6788	0.7006	0.8037	0.8175	0.6427
ReFuGe–GPT-5.6-Luna	0.8038	0.6705	0.5906	0.6652	**0.9010**	0.7262	0.6997	0.8109	0.8923	0.6450

Table 10: Supplementary regression comparison (MAE; lower is better). Published results and results under our setting are grouped separately. Bold marks the best value within each block. Our rows select among three requested rollouts by validation score; reported RelAgent selects among five.

Model / configuration	Amazon item LTV	Amazon user LTV	Avito ad CTR	Event user attend.	H&M item sales	Stack post votes	Trial site success	Trial study adverse
Published results								
Simple reference predictors								
Global Median	64.234	16.783	0.0430	0.2640	0.0760	0.0680	0.4620	57.533
Entity Median	66.436	17.423	0.0460	0.2690	0.0780	0.0690	0.4410	57.930
Hybrid tabular-relational pipelines								
LightGBM	60.569	16.783	0.0410	0.2640	0.0760	0.0680	0.4250	44.011
DS+LightGBM	**41.122**	13.928	0.0440	0.2840	0.0360	0.0650	0.4070	40.581
DS+AutoGluon	45.390	14.399	0.0450	0.2560	0.0430	0.0680	0.4300	44.706
DS+TabPFN-2.5	47.908	15.631	0.0440	0.3180	0.0590	0.0800	0.4320	47.168
RDBLearn	48.559	14.540	0.0340	0.2370	0.0640	0.0680	0.4240	43.913
Supervised relational deep learning								
GraphSAGE	50.053	14.313	0.0410	0.2580	0.0560	0.0650	0.4000	44.473
HGT	55.868	15.412	0.0460	0.2640	0.0640	0.0680	0.4430	45.169
HGTPE	55.849	15.864	0.0480	0.2610	0.0640	0.0680	0.4400	42.648
RelGNN	48.767	14.230	0.0370	0.2380	0.0540	0.0650	**0.3010**	44.461
RelGT	48.922	14.267	0.0350	0.2500	0.0540	0.0650	0.3260	43.992
Relational foundation model								
TabPFN-2.5	60.996	17.513	0.0440	0.5270	0.0960	0.1040	0.4690	55.187
Griffin	53.214	35.590	0.0500	0.4610	0.1510	0.0920	0.4630	78.232
RTzero	57.996	18.802	0.0580	0.3790	0.0890	0.1100	0.4550	73.999
KumoRFM-v1	55.254	16.161	0.0350	0.2640	0.0400	0.0650	0.4170	58.231
KumoRFM-v2	46.992	**13.921**	0.0340	**0.2350**	**0.0340**	**0.0640**	0.4330	43.293
Agentic search								
RelAgent (reported)	41.765	13.949	**0.0330**	0.2410	0.0350	**0.0640**	0.3860	**37.194**
Results under our experimental setting								
STEPREL								
STEPREL–DeepSeek	**43.325**	13.892	0.0339	0.2479	0.0374	0.0676	0.4014	**42.971**
STEPREL–Qwen3.6	43.965	14.003	**0.0336**	0.2450	0.0371	0.0642	0.3896	45.984
STEPREL–Qwen3-Coder	44.232	14.159	0.0348	0.2476	0.0391	0.0635	0.4113	49.229
STEPREL–GPT-5.6-Luna	43.527	**13.832**	0.0340	0.2466	**0.0367**	**0.0632**	0.4011	44.195
RelAgent (controlled)								
RelAgent–DeepSeek	43.681	13.969	0.0339	0.2456	0.0373	0.0636	0.3730	43.337
RelAgent–Qwen3.6	43.814	14.062	0.0351	0.2485	0.0375	0.0655	0.3912	47.152
RelAgent–Qwen3-Coder	45.979	14.832	0.0367	**0.2407**	0.0504	0.0669	**0.3685**	53.109
RelAgent–GPT-5.6-Luna	43.809	13.862	0.0341	0.2472	0.0371	0.0634	0.3884	45.806
ReFuGe								
ReFuGe–DeepSeek	51.912	18.446	0.0412	0.3473	0.0980	0.1348	0.4298	55.047
ReFuGe–Qwen3.6	53.447	19.286	0.0410	0.4367	0.0976	0.1094	0.4426	57.432
ReFuGe–Qwen3-Coder	64.887	18.480	0.0404	0.4627	0.0973	0.1091	0.4385	54.159
ReFuGe–GPT-5.6-Luna	53.360	18.803	0.0361	0.3387	0.0973	0.1105	0.4401	52.257

D CONTEXT AND TOKEN COSTS (RQ2)

This section specifies the token accounting behind RQ2, extends the comparison across backbones, and separates main-search processing from brief construction and cache-aware API charges.

D.1 ACCOUNTING AND MATCHED COMPARISONS

We pair STEPREL and controlled RelAgent runs by task and replicate within each backbone. A pair contributes at a reported search depth only when both runs reach that many scored trials and the corresponding usage records are available. These comparisons describe the cost of reaching a common search depth, rather than the average cost of every launched run. ReFuGe retains a different, lower search budget and is not included in these matched-depth comparisons.

Let p_i and o_i be the provider-reported input and output tokens of model response i. Cumulative main-search processing through scored trial t is

$$C_{\text{main}}(t) = \sum_{i \in \mathcal{I}(t)} (p_i + o_i), \tag{9}$$

where $\mathcal{I}(t)$ contains the distinct model responses through that trial. A response associated with multiple tool evaluations is counted only once. Reported reasoning tokens are part of output and are not added a second time. Prompt length at trial t is the input of the model response associated with that evaluation, not the cumulative input.

For STEPREL, the total additionally includes one third of the reusable task-level brief construction cost C_K:

$$C_{\text{total}}(t) = C_{\text{main}}(t) + C_K/3. \tag{10}$$

RelAgent has no separate brief-construction stage. Cached input still counts in full toward token volume; its billing discount is applied only in the API-cost calculation below. Token counts are compared within a backbone, since tokenization differs across models. Checkpoint rendering makes no LLM call, but its text is counted when processed by a later request.

Trial-24 results use means over paired runs. The Qwen3-Coder diagnostic uses medians over five trial-12 pairs for each task type. Pair counts and summary statistics are stated with each result. DeepSeek classification retains the 24-pair cohort used by the main-text figures. The Qwen3.6 plotted cohorts remove one identified tool-loop pair per task type; the all-pair sensitivity analysis below makes the effect of this choice explicit.

D.2 RESULTS ACROSS BACKBONES

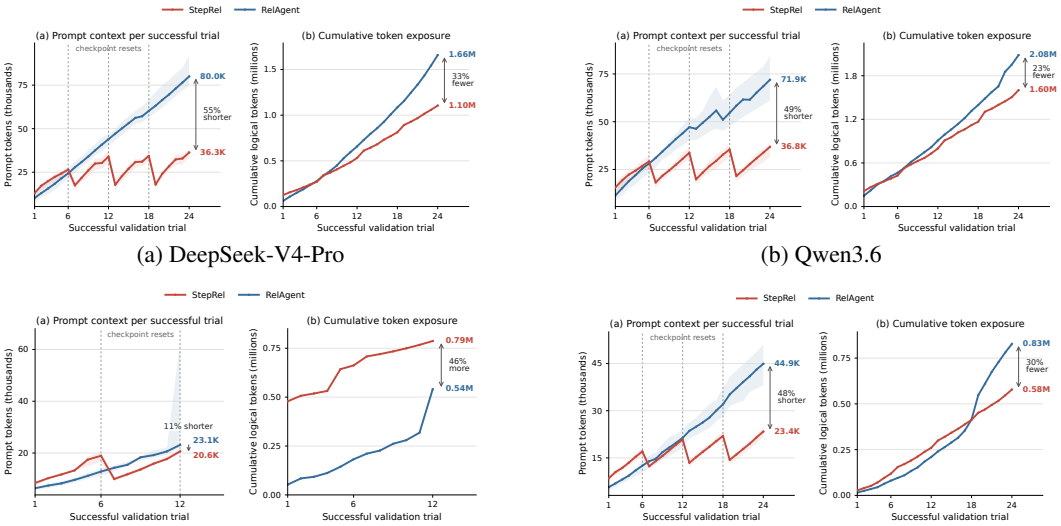
Table 11 gives the trial-24 values for DeepSeek, Qwen3.6, and Luna. Across these displayed cohorts, prompt length decreases by 39.1–55.1%, while cumulative processing decreases by 9.0–38.1%. The cumulative benefit therefore varies more across backbones than the late-stage prompt reduction. The Qwen3.6 mean-cost result is conditional on the stated tool-loop exclusions.

Table 11: Paired means at 24 scored trials. S denotes STEPREL and R RelAgent. Total tokens include input and output, with brief construction amortized over three rollouts for STEPREL. Qwen3.6 rows exclude one tool-loop pair per task type, as detailed below.

Backbone	Task type	Pairs	Prompt (K) S / R	Total (M) S / R	Reduction (%) Prompt / Total
DeepSeek	Classification	24	36.3 / 80.0	1.105 / 1.658	54.6 / 33.3
DeepSeek	Regression	20	38.9 / 77.0	1.259 / 1.912	49.5 / 34.1
Qwen3.6	Classification	20	36.8 / 71.9	1.601 / 2.084	48.8 / 23.2
Qwen3.6	Regression	18	41.1 / 91.4	2.117 / 3.418	55.1 / 38.1
Luna	Classification	26	23.4 / 44.9	0.578 / 0.827	48.0 / 30.1
Luna	Regression	22	22.8 / 37.4	0.561 / 0.616	39.1 / 9.0

Figures 9 and 10 retain the same plotting layout across backbones. The left plot in each panel shows prompt length and its interquartile band; the right plot shows cumulative processing. Dashed boundaries mark the six-scored-trial checkpoint schedule. Prompt length can drop after reconstruction,

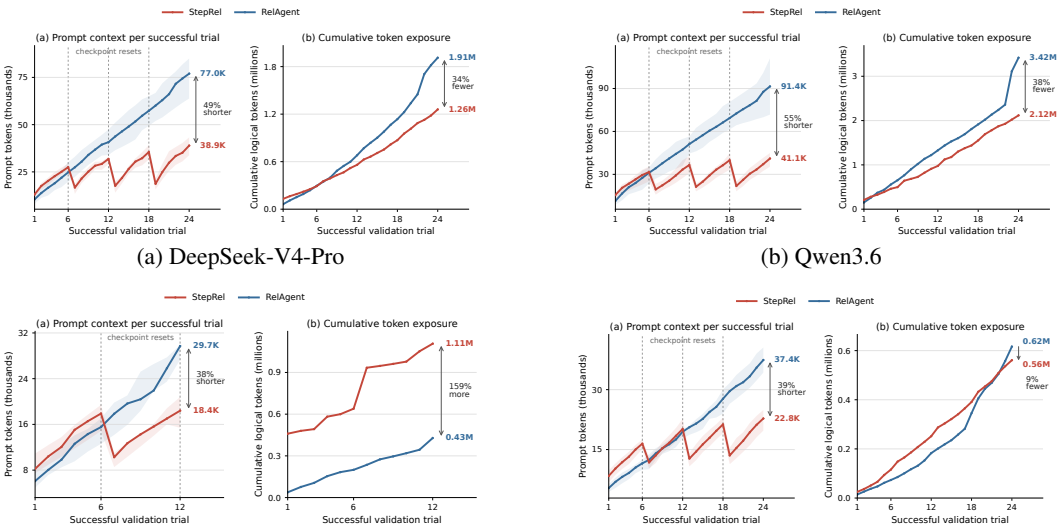
whereas each run's cumulative count continues to grow. A shorter prompt therefore need not imply a lower cumulative total, especially when initialization is included.



(a) DeepSeek-V4-Pro (b) Qwen3.6

(c) Qwen3-Coder-30B (trial 12) (d) GPT-5.6-Luna

Figure 9: Classification prompt length and cumulative input plus output across backbones. Cumulative totals include amortized brief construction. DeepSeek, Qwen3.6, and Luna use trial-24 paired means; Qwen3-Coder uses trial-12 paired medians. Qwen3.6 omits the identified tool-loop pair in this plot; Table 12 includes it.



(a) DeepSeek-V4-Pro (b) Qwen3.6

(c) Qwen3-Coder-30B (trial 12) (d) GPT-5.6-Luna

Figure 10: Regression prompt length and cumulative input plus output, with the same accounting as Figure 9. Qwen3-Coder uses trial-12 medians; the other backbones use trial-24 means. Qwen3.6 omits the identified tool-loop pair in this plot; its inclusion is reported in Table 12.

Sensitivity to tool-loop runs. The Qwen3.6 classification plot excludes the `rel-trial/study-outcome` replicate-1 pair, whose StepRel run contains 2,274 empty-table `get_table_info` calls. The regression plot excludes the `rel-stack/post-votes` replicate-2 pair, whose RelAgent run contains a 596-query exploration loop. These are zero-based replicate identifiers. In each case, both methods' runs are removed to preserve pairing. The events are identified from tool traces; they are still part of the observed operational cost and are restored in the all-pair rows of Table 12.

Table 12: Qwen3.6 cumulative tokens at trial 24, including amortized brief construction. Each exclusion removes both methods' runs for the same task and replicate. S/R is the ratio of the StepRel and RelAgent means.

			Mean tokens (M)		Ratio
Task type	Cohort	Pairs	StepRel	RelAgent	S/R
Classification	All pairs	21	17.558	2.104	8.35
Classification	Loop pair excluded	20	1.601	2.084	0.77
Regression	All pairs	19	2.127	6.993	0.30
Regression	Loop pair excluded	18	2.117	3.418	0.62

Including the classification loop changes StepRel's mean cumulative cost from 1.601M to 17.558M tokens, compared with 2.104M for RelAgent across all 21 pairs. Thus, the all-pair classification mean does not show a cost saving. For regression, including the RelAgent loop increases its mean to 6.993M, compared with 2.127M for StepRel across 19 pairs. The all-pair medians are 1.422M versus 1.813M for classification and 1.988M versus 2.303M for regression, showing that a small number of long interactions strongly affect the means. All-pair mean prompt reductions remain 48.6% and 54.7%, respectively. Prompt reduction is consequently more stable than the mean cumulative-cost comparison in these Qwen3.6 runs.

D.3 Brief Construction and Early Search Costs

Appendix A describes how the database brief is constructed and reused. Table 13 reports its full construction cost and the one-third allocation used for each rollout in the DeepSeek and Qwen3.6 cohorts. These are input-plus-output totals, rather than the length of the final brief. The full task-level cost ranges from 209.5K to 233.3K tokens on average across those four cohorts.

The three-rollout allocation reflects reuse of a frozen brief. A single standalone search on a new task would instead bear the full C_K, while a later search reusing an existing brief would incur no new construction call. These accounting choices differ from provider input caching, which changes the charge for repeated prompt tokens rather than whether exploration is rerun.

Qwen3-Coder at trial 12. Table 14 separates the main search from brief construction for the exact pairs used in the Coder plots. All these runs reach 12 scored trials, so runs that terminate before that boundary do not explain the displayed cost difference.

The main-search cumulative-token medians are lower for StepRel (365.9K versus 540.3K on classification and 385.3K versus 428.1K on regression). Including the allocated brief cost reverses the comparison: 786.7K versus 540.3K and 1.107M versus 428.1K. Prompt length is nevertheless 10.8% and 38.0% lower. These observations distinguish a smaller live context from a lower total cost at an early search boundary. They do not establish that the same Coder runs would recover the initialization cost at a later depth. Each table entry is a separately computed median, so component medians must not be added to reconstruct the total median.

The full-campaign endpoint coverage and predictive stability are discussed in Appendix C. They complement the matched-depth cost analysis but do not change the costs measured for these five pairs per task type.

D.4 Cache-Aware API Cost

Token volume does not depend on a price schedule. Monetary cost additionally distinguishes cache-hit input, cache-miss input, and output. We reweight the recorded DeepSeek usage using the evaluation price snapshot accessed on 2026-07-29:

$$\widehat{\text{cost}} = \frac{0.003625 I_{\text{cached}} + 0.435 I_{\text{uncached}} + 0.87 O}{10^6} \quad \text{USD.} \tag{11}$$

The same one-third brief allocation is included in StepRel's input and output components. These are offline estimates under the recorded prices, not provider invoices or current-price quotations.

Table 13: Canonical Database Brief construction cost averaged over tasks. The per-rollout column assigns one third of a task-level brief to each of three rollouts.

Backbone	Task type	Tasks	Full tokens	Per rollout	Model/SQL calls
DeepSeek-V4-Pro	classification	10	227.0K	75.7K	19.6/33.2
DeepSeek-V4-Pro	regression	8	214.1K	71.4K	17.5/32.2
Qwen3.6	classification	10	209.5K	69.8K	24.1/30.7
Qwen3.6	regression	8	233.3K	77.8K	26.6/33.4

Table 14: Qwen3-Coder at trial 12, using five exact pairs per task type. Each entry is a separately computed median. The total is computed per run before taking its median, so the component medians need not sum to the total median.

	Classification		Regression	
Tokens (K)	StepRel	RelAgent	StepRel	RelAgent
Prompt at trial 12	20.6	23.1	18.4	29.7
Cumulative main search	365.9	540.3	385.3	428.1
Allocated brief construction	471.6	0.0	436.8	0.0
Cumulative including brief	786.7	540.3	1,106.9	428.1

At trial 24, the estimated cost is 25.9% lower for classification ($0.1207 versus $0.1629) and 38.4% lower for regression ($0.1193 versus $0.1937). Although cached input accounts for most input tokens, cache-miss input and output dominate the charges under this price schedule. The monetary reductions therefore differ from the prompt and cumulative-token reductions. The four-panel main-text figure presents the same token and cost decomposition.

E Database Knowledge and Program Update Comparisons (RQ3)

Comparison settings. We examine two interface changes in Qwen3.6 classification searches. For K, we compare agents with and without a database brief, both using full-program proposals and structured context reconstruction. Both retain the database interface specification and query tools. For P, we compare full-program proposals with local updates, both without a database brief and with structured context reconstruction. Both retain the exact incumbent. Local updates add, replace, or remove one block per proposal, while full-program proposals can change multiple blocks. The baseline in these contrasts reconstructs its context and therefore differs from the full-history RelAgent baseline in the main system comparison.

Completed runs are paired by task and seed within each contrast. We average paired results within tasks and then give tasks equal weight. Each contrast uses its own available matched subset, so its baseline average can differ. The agents generate independent search trajectories under each setting.

Measurement and interpretation. Input counts include cached input and measure token processing rather than billed cost. DB calls count recorded tool executions, including schema inspection and SQL queries, rather than successful SQL queries alone. Test AUC is evaluated using the program selected by validation within each run. Final input totals also reflect differences in realized search length.

The database brief reduces on-demand exploration, but its fixed-depth input is higher despite fewer database calls. Its lower final input therefore does not imply a saving at every search depth. Local updates retain an input reduction at the common depth as well as at completion. Mean final AUC is similar in the K contrast and higher in the P contrast, with effects varying across tasks. These observations support the two targeted interface choices without isolating the effect of context reconstruction itself.

Table 15: Offline DeepSeek API-cost estimate at trial 24. Prices are from the snapshot accessed on 2026-07-29; these values are not provider invoices.

Task type	Method	Cached input	Uncached input	Output	Estimated USD
classification	StepRel	907.2K	125.5K	72.2K	$0.1207
classification	RelAgent	1.349M	254.2K	54.5K	$0.1629
regression	StepRel	1.064M	125.5K	69.9K	$0.1193
regression	RelAgent	1.530M	331.0K	50.8K	$0.1937

Table 16: Paired interface comparisons supporting RQ3. Input is cumulative main-agent input in tokens, excluding separate brief construction. Final values use each run's stopping point; trial 12 aligns scored search depth. AUC changes are absolute differences.

Metric	Baseline	Modified	Change
K: without brief → with brief			
Final cumulative input	1,844,352	1,344,327	−27.1%
Final test ROC–AUC	0.8184	0.8190	+0.0006
DB calls at trial 12	40.57	24.43	−39.8%
Cumulative input at trial 12	1,036,390	1,169,357	+12.8%
P: full program → local update			
Final cumulative input	2,125,856	868,208	−59.2%
Final test ROC–AUC	0.8046	0.8199	+0.0153
Cumulative input at trial 12	894,645	616,524	−31.1%

F Generated SQL Programs and Case Details (RQ4)

This section expands the two examples in RQ4 with their task context, complete relevant SQL blocks, and saved diagnostics. The examples illustrate what generated features measure and which information they require.

F.1 Activity Frequency and Entity Breadth

The first example is a selected StepRel program generated with GPT-5.6 Luna for `rel-avito/user-visits` under the controlled protocol. Each prediction is keyed by a user and prediction timestamp. The `visit_long_windows` block below computes visit counts and distinct advertisement counts over several historical windows. Only whitespace has been changed from the saved SQL.

```
SELECT e.UserID, e.timestamp,
  COUNT(CASE
    WHEN v.ViewDate >= e.timestamp - INTERVAL '8 days'
    THEN 1 END) AS visits_8d,
  COUNT(DISTINCT CASE
    WHEN v.ViewDate >= e.timestamp - INTERVAL '8 days'
    THEN v.AdID END) AS ads_8d,
  COUNT(CASE
    WHEN v.ViewDate >= e.timestamp - INTERVAL '12 days'
    THEN 1 END) AS visits_12d,
  COUNT(DISTINCT CASE
    WHEN v.ViewDate >= e.timestamp - INTERVAL '12 days'
    THEN v.AdID END) AS ads_12d,
  COUNT(CASE
    WHEN v.ViewDate >= e.timestamp - INTERVAL '8 days'
    AND v.ViewDate < e.timestamp - INTERVAL '4 days'
    THEN 1 END) AS visits_prior_4d,
  COUNT(DISTINCT CASE
    WHEN v.ViewDate >= e.timestamp - INTERVAL '8 days'
    AND v.ViewDate < e.timestamp - INTERVAL '4 days'
```

```
    THEN v.AdID END) AS ads_prior_4d
FROM eval_table e
LEFT JOIN VisitStream v
    ON v.UserID=e.UserID
    AND v.ViewDate < e.timestamp
GROUP BY e.UserID, e.timestamp
```

The join supplies the upper time bound for every count. In the 12-day window, `visits_12d` counts visit events while `ads_12d` counts distinct non-null advertisement identifiers. This separates repeated activity on the same advertisement from activity across different advertisements, even when the total number of visits is identical.

The saved `test_results.json` contains the same program as `best_program.json` and records 36,129 predictions without an execution error. In its importance diagnostic, `ads_12d` ranks first with mean absolute SHAP value 0.035389, while `visits_12d` ranks seventeenth with value 0.000400. This diagnostic fits a tree-based surrogate to the downstream model's predictions and computes SHAP values for that surrogate. It therefore describes how features explain the saved predictions, rather than measuring the performance gain from adding or removing either feature. The example makes the distinction between event frequency and entity breadth visible in an executable program.

F.2 Future Events and Information Availability

The second example comes from an earlier Qwen3.6 StepRel run on `rel-event/user-ignore`, outside the controlled comparison. This task predicts whether a user will ignore more than two event invitations in the next seven days. The saved `invitation_windows` block combines past activity with a count that starts at the prediction timestamp. The complete block is reproduced below with line breaks added for readability.

```
SELECT et.user, et.timestamp,
  COALESCE(w0.past_30d_invited, 0) as past_30d_invited,
  COALESCE(w0.past_90d_invited, 0) as past_90d_invited,
  COALESCE(w0.past_30d_yes, 0) as past_30d_yes,
  COALESCE(w0.past_30d_no, 0) as past_30d_no,
  COALESCE(w0.past_30d_maybe, 0) as past_30d_maybe,
  COALESCE(w0.future_7d_invited, 0) as future_7d_invited
FROM eval_table et
LEFT JOIN (
  SELECT ea.user_id,
  SUM(CASE
    WHEN ea.start_time >= et.timestamp - INTERVAL '30 days'
    AND ea.start_time < et.timestamp
    THEN 1 ELSE 0 END) as past_30d_invited,
  SUM(CASE
    WHEN ea.start_time >= et.timestamp - INTERVAL '90 days'
    AND ea.start_time < et.timestamp
    THEN 1 ELSE 0 END) as past_90d_invited,
  SUM(CASE
    WHEN ea.start_time >= et.timestamp - INTERVAL '30 days'
    AND ea.start_time < et.timestamp
    AND ea.status = 'yes'
    THEN 1 ELSE 0 END) as past_30d_yes,
  SUM(CASE
    WHEN ea.start_time >= et.timestamp - INTERVAL '30 days'
    AND ea.start_time < et.timestamp
    AND ea.status = 'no'
    THEN 1 ELSE 0 END) as past_30d_no,
  SUM(CASE
    WHEN ea.start_time >= et.timestamp - INTERVAL '30 days'
    AND ea.start_time < et.timestamp
    AND ea.status = 'maybe'
    THEN 1 ELSE 0 END) as past_30d_maybe,
  SUM(CASE
```

```
    WHEN ea.start_time >= et.timestamp
    AND ea.start_time < et.timestamp + INTERVAL '7 days'
    THEN 1 ELSE 0 END) as future_7d_invited
FROM event_attendees ea
WHERE ea.user_id IS NOT NULL
    AND ea.status = 'invited'
    AND ea.start_time > '1950-01-01'
GROUP BY ea.user_id) w0
    ON et.user = w0.user_id
```

The future-window count reads the same event type and horizon used to construct the task target. The recorded audit found that thresholding this count at greater than two reconstructed all 19,239 cached training labels, with 14,368 nonzero counts. This is an observed agreement on those records. The generated interval includes its lower endpoint and excludes its upper endpoint, whereas the task definition uses the opposite endpoint convention. The block also retains redundant expressions: its outer status filter makes `past_30d_yes`, `past_30d_no`, and `past_30d_maybe` zero.

For past training anchors, events in this window can already be present in the database available during search. The same events are unavailable for future anchors when validation and test snapshots stop at their respective cutoffs. The resulting feature can therefore have a strong training signal that disappears at prediction time. This example illustrates a distribution shift in feature availability and motivates the validation and test access controls in Appendix B.3. It does not establish access to held-out outcomes in the controlled evaluation.

G LIMITATIONS

Experimental scope. Our findings concern the relational prediction tasks and LLM configurations evaluated in this study. Their transfer to other database schemas, task types, and search budgets remains to be established. Stochastic generation and changes to hosted model services can also affect the programs discovered and their predictive performance.

Component attribution. The primary experiments evaluate STEPREL as a complete search system. The targeted comparisons in Appendix E examine the database brief and program-update interface. They do not independently establish the contributions of evidence selection and context reconstruction, which remain shared settings in these comparisons.

Cost interpretation. Token counts and estimated API fees characterize model interaction costs. They do not directly measure end-to-end runtime, which also depends on SQL execution, downstream model training, database size, and serving latency. Monetary savings further depend on the provider's pricing and caching policy.

H EXTENDED RELATED WORK

This section expands the main related work around relational prediction, feature-program search, and agent state and context management. It also summarizes prior work relevant to the shared data-access protocol used in our experiments.

H.1 LEARNED REPRESENTATIONS FOR RELATIONAL PREDICTION

Early statistical relational learning extended logical and probabilistic models from independent examples to linked objects, supporting collective prediction through shared relational templates (Džeroski, 2010; Neville & Jensen, 2007). Closely related work on knowledge graphs and multi-relational graphs learned entity–relation representations or propagated information over typed edges (Ristoski & Paulheim, 2016; Schlichtkrull et al., 2018). These formalisms established the general principle that an entity's prediction may depend on attributes and relations beyond its own row, but they were not designed around modern database schemas, temporal prediction tables, or heterogeneous feature types.

A subsequent line of work developed representations and learning pipelines specifically for relational databases. RDB2Graph maps database rows to graph nodes and primary–foreign key relations

to edges, while EmbDI constructs embeddings over heterogeneous relational records for downstream integration and prediction (Cvitkovic, 2020; Cappuzzo et al., 2020). Other systems operate on alternative relational intermediates. ATJ-Net learns over automatically joined tables, KEN enriches target examples with representations derived from background relations, and GFS combines graph structure with differentiable single-table models (Bai et al., 2021; Cvetkov-Iliev et al., 2023; Zhang et al., 2023). These methods illustrate several possible interfaces between database structure and a predictor: explicit joins, relational embeddings, hypergraph-like representations, or learned message passing.

RelBench and 4DBInfer made graph-centric relational prediction more systematic by providing database schemas, temporal target tables, and standardized conversion and evaluation pipelines (Robinson et al., 2024; Wang et al., 2024). PyTorch Frame provides complementary infrastructure for encoding heterogeneous tabular attributes within such models (Hu et al., 2024). Standard graph architectures such as GraphSAGE, GAT, PNA, and HGT can be applied after database-to-graph conversion (Hamilton et al., 2017; Velickovic et al., 2017; Corso et al., 2020; Hu et al., 2020). More recent RDB-specific architectures adapt the access or aggregation mechanism to relational prediction. ContextGNN constructs task-conditioned relational context, RelGNN composes multiple message-passing mechanisms, and RelGT applies graph-transformer layers and structural encodings to database-derived graphs (Yuan et al., 2025; Chen et al., 2025; Dwivedi et al., 2025). These models learn features jointly with the predictor, but the neighborhood depth, message-passing mechanism, and structural encoding are determined by the architecture and preprocessing pipeline.

Relational foundation models seek to transfer such learned mechanisms across schemas, databases, or tasks. Griffin pretrains a graph-centric relational model and adapts it to downstream databases, while the Relational Transformer targets zero-shot or transfer-oriented relational prediction through a transformer operating over relational context (Wang et al., 2025; Ranjan et al., 2025). KumoRFM and KumoRFM-2 scale in-context relational prediction over collections of databases, whereas RDB-PFN learns relational in-context inference from synthetically generated databases (Fey et al., 2025; Hudovernik et al., 2026; Wang et al., 2026). These approaches differ substantially in their pretraining data, adaptation protocol, and relation-access mechanism, but all internalize relational structure into learned parameters or activations.

Hybrid methods retain a relational preprocessing stage while delegating the final prediction to a tabular model. GNN+TabPFN first summarizes a relational neighborhood before applying a tabular foundation model, and RDBLearn constructs a relationally augmented target table for TabPFN-based inference (Meyer et al., 2026; Xu et al., 2026). Direct LLM predictors provide another learned alternative: linked records can be collected and serialized into text, or combined with learned relational representations before prediction (Wydmuch et al., 2024; Wu et al., 2025). StepRel studies iterative search over executable SQL feature programs.

H.2 Explicit relational feature synthesis

Explicit relational feature engineering materializes a predictor input through declared joins, transformations, and aggregations. Deep Feature Synthesis recursively composes transformation and aggregation primitives along relational paths, while OneBM automates a related process for one-to-many database structures (Kanter & Veeramachaneni, 2015; Lam et al., 2017). Their main advantage is provenance: each generated feature can be traced to a relational path and a sequence of predefined operators. Their main restriction is that the searchable space is fixed by the operator library and traversal strategy.

Later systems make this process more selective. ARDA augments a base table with signals from related data sources and selects useful relational features, rather than exhaustively retaining all possible augmentations (Chepurko et al., 2020). FeatAug searches predicate-conditioned aggregation queries over one-to-many relations, allowing task-specific filters and aggregation conditions to be selected without materializing the entire candidate space (Qi et al., 2024). These methods move from exhaustive enumeration toward guided relational feature search, but the query language remains defined by a fixed collection of templates or operators.

Generic relational-to-tabular pipelines occupy an intermediate point between explicit synthesis and learned relational modeling. DFS-based pipelines construct a broad, task-agnostic feature table and then train LightGBM, AutoGluon, or a tabular foundation model. RDBLearn similarly combines

generic relational aggregation with a strong tabular predictor (Xu et al., 2026). Such pipelines can be effective because the downstream model selects among many materialized statistics, but the relational mapping itself is usually fixed before task-specific validation begins. The selected model therefore adapts to the task more strongly than the relational feature generator does.

Executable SQL programs provide a larger task-dependent hypothesis space. In addition to standard column-wise aggregates, SQL can express conditional co-occurrences, interactions among multiple rows, temporal windows, multi-stage common-table expressions, and self-joins over relational motifs. RelAgent contrasts this query-defined feature space with fixed DFS operators and demonstrates cases in which SQL preserves joint row conditions that independent column-wise aggregations discard (Huang et al., 2026). SQL thus offers a direct way to specify and materialize a desired relational property. Whether a learned architecture can recover the same signal depends on its representation and training.

STEPREL belongs to this explicit-program family and focuses on the persistent search state needed to continue revising SQL feature programs.

H.3 LLMS FOR TABULAR DATA SCIENCE AND RELATIONAL SEARCH

LLMs have been used with structured data in several distinct roles. As direct predictors, they serialize rows, tables, or relational neighborhoods into text and produce labels in-context. Such approaches can exploit column semantics and pretrained world knowledge, but remain sensitive to serialization, ordering, context selection, and prompt design (Hegselmann et al., 2022; Wydmuch et al., 2024; Wu et al., 2025). Structure-aware prompting and graph tokenization provide richer interfaces, but still internalize the selected relational context inside the model input.

A second line of work uses LLMs as feature engineers rather than final predictors. CAAFE proposes executable feature transformations using column descriptions and examples, while FeatLLM generates semantically meaningful rules or features for few-shot tabular learning (Hollmann et al., 2023; Han et al., 2024). OCTree incorporates decision-tree feedback to refine individual rules, and LLM-FE treats feature construction as evolutionary program search guided by downstream evaluation (Nam et al., 2024; Abhyankar et al., 2025). Memory-augmented multi-agent systems further retain useful transformations, failures, and evaluation outcomes across generation rounds (Dong et al., 2026). These systems show that feature engineering benefits from an iterative proposal–execution–feedback loop, but mostly operate on single tables or restricted transformation languages.

General data-science agents broaden the action space from feature construction to data inspection, code generation, model training, and reporting. Data Interpreter coordinates executable analysis steps, MLAgentBench evaluates autonomous experimentation over machine-learning tasks, and MLE-bench studies end-to-end machine-learning engineering under repository- and competition-style environments (Hong et al., 2025; Huang et al., 2023; Chan et al., 2025). These systems expose similar difficulties in long-horizon planning, execution repair, and state retention. Their task units, tools, budgets, and success criteria, however, differ from the repeated evaluation of SQL feature programs over a fixed relational prediction task.

ReFuGe and RelAgent are the closest LLM-based systems in relational feature engineering. ReFuGe decomposes the workflow into a staged multi-agent pipeline for schema selection, candidate generation, validation, and feature filtering (Kim et al., 2026). This structure constrains how information and candidate features flow between stages. RelAgent instead exposes a more open-ended search environment with database-exploration queries, complete SQL feature programs, downstream validation, and an evaluation workspace containing previous trial artifacts (Huang et al., 2026). It can jointly revise SQL features, model families, hyperparameters, target transformations, and evaluation strategies.

STEPREL builds on this evaluation-guided search setting. It maintains reusable database knowledge, the incumbent SQL program, and structured search evidence outside the conversation, and selects what to carry forward when reconstructing context.

H.4 AGENT STATE AND CONTEXT MANAGEMENT

Agent-memory systems differ in both what they retain and how retained information influences later actions. Generative Agents store and retrieve natural-language memories of past observations. Reflexion retains verbal feedback derived from failures, and Voyager accumulates executable skills that can be reused in later environments (Park et al., 2023; Shinn et al., 2023; Wang et al., 2023a). These systems retain different forms of experience to guide later actions, including observations, feedback, and reusable code.

A separate line of work manages the model-visible context itself. MemGPT organizes working and external memory as a hierarchy, while ACON learns or optimizes compressed representations of long trajectories (Packer et al., 2023; Kang et al., 2025). Context-as-a-Tool (CAT) lets an agent explicitly inspect and modify its context, and SAM constructs compact state-adaptive cues backed by retained information (Liu et al., 2026; Hu et al., 2026). Prompt-compression methods such as LLMLingua reduce the number of tokens needed to present a given body of text (Jiang et al., 2023). These methods primarily ask which parts of a history should remain model-visible and how they can be represented compactly.

Other systems represent task progress as explicit execution state. AgentProg represents a long-horizon GUI task as a semantic program whose execution progress can be resumed, while MAGE organizes agent memory as structured execution state that supports compression, checking, and revision (Tian et al., 2026; Chen et al., 2026). These representations describe the current state of task completion and support continued execution.

STEPREL applies this perspective to relational feature search. The Program Store maintains the incumbent as named SQL blocks, while persistent search evidence records candidate revisions, validation outcomes, and execution failures. Context reconstruction combines the database brief, the complete incumbent, and selected search evidence. Records omitted from the selected view remain in persistent storage. Checkpoint size can grow with the incumbent, but it does not accumulate the complete interaction history.

H.5 EXECUTION-GROUNDED PROGRAM REVISION

Tool-using language models interleave generation with observations from external systems. ReAct established a general reasoning–action loop, while Toolformer studied how models can learn to invoke external APIs (Yao et al., 2022; Schick et al., 2023). Planning and reflection methods extend this pattern by decomposing actions or retaining feedback from previous attempts (Wang et al., 2023b; Shinn et al., 2023).

For generated programs, execution provides a more objective feedback channel than natural-language self-assessment. LEVER combines candidate generation with execution-based verification, while self-debugging methods use compiler, runtime, or test feedback to revise code (Ni et al., 2023; Chen et al., 2024). The relevant observation for relational feature search is that syntactically plausible SQL may still fail to execute, return the wrong prediction grain, time out, or yield an uninformative feature matrix. Search must therefore retain not only scalar validation scores, but also structured information about program identity, execution status, and downstream diagnostics.

RelAgent operationalizes this idea through an evaluation workspace that stores trial metadata and row-level validation predictions, allowing the agent to compare errors, inspect feature importance, and analyze changes between trials (Huang et al., 2026). ReFuGe uses a more structured pipeline in which evaluation feedback filters and revises generated feature candidates (Kim et al., 2026). STEPREL links candidate revisions and feedback to stored program versions, then selects search evidence for subsequent context reconstruction.

H.6 DATA ACCESS AND EVALUATION PROTOCOLS

Data leakage can occur even when a conventional train–test split is present. Features may encode target-derived information, preprocessing may be fitted using held-out examples, or temporal records unavailable at prediction time may enter the model (Kaufman et al., 2011; Kapoor & Narayanan, 2023). Relational databases make these failures particularly subtle because information can be reached through multi-hop joins, nested subqueries, derived aggregates, and multiple

timestamps. Validation-guided search introduces an additional risk: repeated program selection can overfit the validation split even when test labels remain hidden.

Related work on tool-using agents enforces information and action boundaries. CaMeL separates trusted control flow from untrusted tool outputs, while Progent and MiniScope enforce programmable or least-privilege authorization policies (Debenedetti et al., 2025; Shi et al., 2025; Zhu et al., 2025). Text-to-SQL evaluations similarly distinguish query correctness from compliance with database access policies (Fei et al., 2026).

Temporal and data-access controls are also part of existing relational prediction systems. RelBench provides temporal task splits and data access relative to prediction time (Robinson et al., 2024). RelAgent separates feature queries from its analysis-only evaluation workspace and instructs queries to apply row-level temporal predicates (Huang et al., 2026). Our experiments use shared data-access boundaries: generated SQL sees target-free anchors and the active database snapshot, labels remain inside the evaluator, and the selected program is frozen before test execution. These controls define the evaluation protocol. They do not establish that every generated feature satisfies all task-specific availability constraints.