RELAGENT: LLM Agents as Data Scientists for Relational Learning

Xingyue Huang[1]* **Louis Tichelman**[2,3] **Jinwoo Kim**[4] **Krzysztof Olejniczak**[1]
İsmail İlkan Ceylan[2,3,1]
[1]University of Oxford [2]TU Wien [3]AITHYRA [4]KAIST

Abstract

Relational learning is a challenging problem that has motivated a wide range of approaches, including graph-based models (e.g., graph neural networks, graph transformers), tabular methods (e.g., tabular foundation models), and sequence-based approaches (e.g., large language models), each with its own advantages and limitations. We propose RELAGENT, an LLM-based autonomous data scientist for relational learning, which operates in two phases. In the *search phase*, an LLM agent uses database, validation, and evaluation workspace tools to construct SQL feature programs and select a predictive model. In the *inference phase*, the resulting program is executed without further LLM calls. The final predictor consists of SQL queries and a classical model, enabling fast, deterministic, and intrinsically interpretable predictions: features are human-readable queries, and predictions depend only on the resulting query-defined feature map, enabling scalable deployment using standard database systems.

1 Introduction

Relational databases are the standard paradigm for storing, managing, and processing structured data [1]. In practice, predictive modeling over such data is driven by feature construction: analysts write SQL queries that join tables along foreign-key paths, aggregate signals over time windows, and construct features such as counts, recency statistics, or activity trends for downstream models [2–4]. The space of possible features is vast, and identifying useful combinations often requires iterative trial-and-error over joins, filters, and aggregations. This process is time-consuming, requires substantial domain expertise, and becomes increasingly difficult to scale as database schemas grow in complexity.

Machine learning over relational databases, commonly referred to as relational learning, aims to automate this process. Existing approaches address this challenge through different representations, access patterns, and computational mechanisms (see Table 1). *Tabular methods*, including supervised tabular models [5–7] and tabular foundation models [8, 9], operate on flat feature matrices and can be highly effective, but applying them to relational databases requires either substantial feature engineering or generic aggregations that may fail to capture task-specific relational interactions. *Graph-based models*, such as graph neural networks [10–12], graph transformers [12], and relational foundation models [13–16], typically operate on a graph induced by the database through entity-centric subgraphs. While powerful, these methods rely on a fixed mechanism for accessing relational structure, where the effective context, such as the neighborhood size and depth of aggregation, is chosen in advance and may not necessarily align with the downstream task. *Sequence-based approaches*, such as large language models (LLMs) [17, 18] applied to serialized tables or graphs, offer flexible reasoning interfaces, but their inductive bias is not naturally aligned with structured relational prediction. These limitations point to a complementary direction: keeping relational structure external and executable, while automating the search for task-relevant queries.

*A substantial part of this research was conducted while the author was a predoctoral fellow at AITHYRA.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

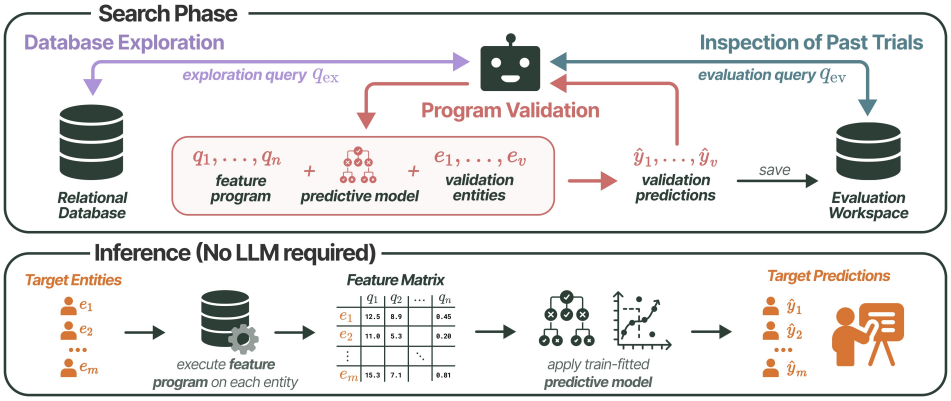


Figure 1: **RELAGENT.** During the search phase, an LLM agent iteratively proposes and refines a *feature program* consisting of SQL feature queries $\{q_1, \ldots, q_n\}$ and a predictive model configuration φ to solve a given task. The agent uses three tools: **(1) database exploration** via read-only SQL exploration queries, **(2) program validation** by executing candidate programs on a validation set and receiving performance metrics, and **(3) inspection of past trials** in the *Evaluation Workspace* via evaluation queries. Once a final program is selected, the agent is no longer needed at inference time.

Our approach. We propose RELAGENT, a framework that casts relational learning as an *agentic search process* over query-based features and predictive models. As illustrated in Figure 1, RELAGENT operates in two phases:

(1) *Search phase*: An LLM agent interacts with the system through three query interfaces: (i) **exploration queries** that inspect the database, (ii) **feature queries** that construct predictive features, and (iii) **evaluation queries** that analyze predictions and diagnostics stored in an evaluation workspace. The agent iteratively proposes feature programs, selects a downstream predictive model (e.g., XGBoost [5], LightGBM [6]), and refines program and model choice based on this feedback.

(2) *Inference phase*: The selected feature program is executed on the target entities, and the resulting features are passed to the predictor fitted from the training examples.

More broadly, RELAGENT defines predictors over query-induced feature maps, with the hypothesis class parameterized by executable queries rather than fixed architectures. The final predictor consists only of an SQL feature program and a predictive model, yielding predictions that are fast, deterministic, and intrinsically interpretable: each prediction can be traced to the selected queries, their materialized feature values, and the downstream model's feature usage. This also gives a different view of relational inductive bias: two target examples with identical outputs under the selected SQL features receive identical predictions, so invariance is induced by the selected queries rather than by a fixed graph encoding, sequence ordering, or neural architecture.

Contributions. We summarize our main contributions as follows:

- We introduce RELAGENT, an agentic framework that formulates relational learning as an autonomous search process over query-based features and predictive models using an LLM interacting with relational databases via tools.
- We propose an execution-grounded learning loop with validation feedback and an evaluation workspace, enabling iterative refinement through structured analysis of prior trials.
- We decouple search and inference, yielding a final predictor that does not require an LLM at inference time, and is fast, deterministic, and interpretable.
- We formulate relational learning over query-defined feature maps, with invariance induced by executable queries rather than fixed architectural biases.
- We empirically evaluate RELAGENT on relational benchmarks, demonstrating competitive performance with strong baselines while retaining interpretability and scalability.

2 Related work

Relational deep learning. Predictive modeling over relational databases traditionally relies on manually materializing task-specific feature tables via joins and aggregations, followed by supervised

Table 1: Comparison of relational learning approaches.

Method	Representation	Access	Mechanism	Inductive bias	Interpretability
Tabular	Tables	Internalized (flatten)	Trees / Attention	Row perm.-equiv.	Feature-level
GNNs	Graph	Internalized (local)	Message passing	Perm.-equiv.	Post-hoc
Graph Transf.	Graph (with PE)	Internalized (global)	Attention	Perm.-equiv.	Post-hoc
LLMs	Sequence	Internalized (text)	Attention	Order-sensitive	Post-hoc
RELAGENT	Database	External (queries)	SQL + Predictors	Query-defined	Intrinsic

tabular models such as XGBoost [5], LightGBM [6], or AutoGluon [7] – a workflow that requires substantial engineering and does not transfer across schemas or targets. Relational Deep Learning (RDL) avoids manual flattening by treating databases as temporal heterogeneous graphs (rows as nodes, primary-foreign key links as edges, training tables as targets) [2], underlying GNN- and Graph Transformer-based models such as RelGNN [19] and RelGT [12]. More recently, relational foundation models, including Griffin [14], Relational Transformers [13], KumoRFM-v1 [15], and KumoRFM-v2 [16], aim to achieve stronger transfer across schemas, databases, and tasks. Another line of work directly utilizes tabular or language foundation models, with examples including GNN+TabPFN-2.5 [20], RDBLearn [9], and LLM-based predictors that serialize relational database as context [13, 17, 21]. RDL is also related to the broader effort of building graph and knowledge graph foundation models [22–26], but is specialized to relational databases with schemas and temporal prediction targets.

LLMs for structured data. LLMs can be applied to structured data by serializing tables and schemas into text, enabling tasks such as tabular prediction [27] and text-to-SQL [28]. Row-level serialization can support few-shot tabular learning, but performance remains sensitive to prompt design and representation choices [29, 30]. Similarly, graph-structured data can be encoded as text, though such approaches are often brittle to formatting and neighborhood choices [18]. Later methods add structure-aware adaptations, including graph prompting and instruction tuning [31–33], or inject structure via graph tokenization [34, 35]. More recently, tool use enables LLMs to interact with structured data through explicit operations rather than pure prompting. ReAct [36] interleaves reasoning with tool calls, Toolformer [37] learns API use, and extensions such as Plan-and-Execute [38] and Reflexion [39] add planning and self-correction. Graph-specific variants further enable structured reasoning via tool invocation [40] or executable programs [41]. Unlike approaches that internalize relational structure by serializing tables or graph neighborhoods into the LLM context, RELAGENT keeps the database external and accessible through SQL.

LLMs as feature engineers. Beyond using LLMs as direct predictors, recent work treats them as feature engineers that generate task-specific transformations for downstream models. Early approaches largely focus on single-table settings: CAAFE [42] and FeatLLM [43] show that LLMs can synthesize semantically meaningful features or rule-based transformations for tabular data, improving few-shot performance while avoiding LLM usage at inference time. More recent systems cast feature engineering as an iterative search process: LLM-FE [44] formulates feature construction as program search guided by validation feedback, and OCTree [45] further incorporates decision tree reasoning to iteratively refine individual rules. For multi-relational databases, ReFuGe [46] extends LLM-based feature generation through a fixed sequential pipeline with aggregate validation feedback and a fixed downstream model. RELAGENT instead exposes three query interfaces: exploration queries over the database, feature queries used as executable programs, and evaluation queries over a persistent workspace of row-level predictions and trial histories. This enables targeted self-correction and joint search over SQL features and downstream models, while keeping the deployed predictor intrinsically interpretable.

3 Preliminaries

Relational databases. A *relational database* is a collection of tables $\mathcal{D} = \{\mathcal{T}_1, \ldots, \mathcal{T}_k\}$. Each table contains rows representing entities or events, with typed columns such as numerical or categorical attributes, timestamps, and primary/foreign keys. We denote the schema by $\mathcal{S}(\mathcal{D})$, which includes table names, column names, types, and primary-foreign key relations defining links across tables.

Entity prediction tasks over relational databases. A supervised entity-level prediction task consists of a relational database (called *context*) $\mathcal{D}$ and target tables $\mathcal{T}_{\text{train}}, \mathcal{T}_{\text{val}}, \mathcal{T}_{\text{test}}$. Each target row is of the form (e, t, y), where e is the target entity, t is the prediction timestamp, and $y \in \mathcal{Y}$ is a classification or regression label. The target tables are split chronologically by cutoffs $t_{\text{train}} < t_{\text{val}} < t_{\text{test}}$, and $\mathcal{D}$ contains only records available up to the test cutoff.

We enforce *point-in-time correctness*: for each target row (e, t, y), feature construction may use only source records with a timestamp strictly before t. For validation and test predictions, we augment the context with training labels, $\widetilde{\mathcal{D}} = \mathcal{D} \cup \{\mathcal{T}_{\text{train}}\}$, and learn a predictor $g : (\widetilde{\mathcal{D}}, e, t) \mapsto \hat{y}$. Thus, validation and test predictions may use historical relational records and training labels, but never validation or test labels.

SQL queries. We distinguish three types of queries. Let $\mathcal{L}$ denote the class of SQL queries available to the agent. An *exploration query* $q_e \in \mathcal{L}$ is a read-only query over the context database $\mathcal{D}$ used for schema and data inspection. A *feature query* $q \in \mathcal{L}$ maps a target entity e, timestamp t, and context $\widetilde{\mathcal{D}}$ to one or more feature values, $q(\widetilde{\mathcal{D}}, e, t) \in \mathcal{X}^{d_q}$, using only records before t; a collection of feature queries $\boldsymbol{q} = (q_1, ..., q_n)$ is a *feature program*. An *evaluation query* is a read-only query over the evaluation workspace, used only for search-time diagnostics and never as a predictive feature.

Feature-construction map. For a target table $\mathcal{T}$ with m rows, a feature program $\boldsymbol{q} = (q_1, \ldots, q_n)$ materializes and aligns query outputs into $\boldsymbol{X} = \Phi_{\boldsymbol{q}}(\widetilde{\mathcal{D}}, \mathcal{T}) \in \mathcal{X}^{m \times d}$, where $d = \sum_{j=1}^{n} d_{q_j}$.

4 Methodology

We present RELAGENT, an agentic framework that uses an LLM agent to autonomously search over SQL-derived features and predictive models (such as XGBoost [5], LightGBM [6], etc.) for relational learning via tool-use. Given an augmented database context $\widetilde{\mathcal{D}}$ and a validation target table $\mathcal{T}_{\text{val}}$, the framework has two stages:

(1) **Search phase.** The agent iteratively explores the database, validates candidate feature programs and predictive models on $\mathcal{T}_{\text{val}}$, and inspects previous trials to select a feature program and model.
(2) **Inference phase.** The best validated program and model are deployed without further LLM calls: the program produces feature vectors, and the fitted predictive model outputs predictions.

4.1 Search phase

Let $\mathcal{L}$ be the class of SQL queries available to the agent and $\mathcal{P}$ the class of predictive models, where each $\varphi \in \mathcal{P}$ determines a model family and its hyperparameters. We write $\mathcal{L}^* = \bigcup_{n \geq 1} \mathcal{L}^n$ for the set of finite-length feature programs. The agent searches over the joint space of feature programs and predictive models, $\mathcal{L}^* \times \mathcal{P}$, to identify a candidate $(\boldsymbol{q}, \varphi)$ with strong validation performance. We formalize the search as a stochastic policy over a structured action space $\mathcal{A}$.

Let $a_t \in \mathcal{A}$ denote the action taken at step t and o_t the tool observation returned after executing a_t. Given the interaction history $H_t = \big(\widetilde{\mathcal{D}},\ \mathcal{T}_{\text{val}},\ (a_1, o_1), \ldots, (a_{t-1}, o_{t-1})\big)$ at step t, the agent samples its next action as $a_t \sim \pi(\cdot \mid H_t)$, where the policy π is implemented by an LLM. The action space can be divided into three action types:

Database exploration. The agent can inspect the augmented database $\widetilde{\mathcal{D}}$ and the label-masked validation table $\mathcal{T}_{\text{val}}$ using two read-only tools:

(1) `execute_query()`: runs read-only SQL exploration queries;
(2) `get_table_info()`: returns table schemas $\mathcal{S}(\widetilde{\mathcal{D}})$, primary/foreign keys, and row counts.

This allows the agent to identify relevant tables, columns, timestamp fields, primary-foreign key paths, and candidate temporal aggregations. For instance, the agent typically first inspects table schemas and then issues aggregate queries to summarize entity frequencies, timestamp ranges, missingness patterns, and relationships between tables. These exploratory queries provide context for subsequent feature construction, but need not directly be used as predictive features unless the agent later includes them in a candidate program.

Program validation. The agent can submit a tuple $(\boldsymbol{q}, \varphi)$ consisting of a feature program $\boldsymbol{q}$ and a predictive model φ for validation through the tool `validate_program()`. The model $\varphi \in$

$\mathcal{P}$ specifies both the model family and its associated hyperparameters. For regression tasks, it additionally includes optimization-related choices, such as the type of regularization (e.g., L_1, L_2), the loss function (e.g., Huber loss), and potential input transformations (e.g., a log-target transformation may be applied, whereby the model is trained on $\log(1+y)$). The validation tool first produces features for training and evaluation entities via the feature construction map induced by $\boldsymbol{q}$.

$$\boldsymbol{X}_{\text{train}} = \Phi_{\boldsymbol{q}}(\widetilde{\mathcal{D}}, \mathcal{T}_{\text{train}}), \qquad \boldsymbol{X}_{\text{val}} = \Phi_{\boldsymbol{q}}(\widetilde{\mathcal{D}}, \mathcal{T}_{\text{val}}).$$

Let $\boldsymbol{y}_{\text{train}}$ and $\boldsymbol{y}_{\text{val}}$ denote the label vectors extracted from $\mathcal{T}_{\text{train}}$ and $\mathcal{T}_{\text{val}}$, respectively. A predictor f, initialized w.r.t the submitted model configuration φ, is fitted to the training data $(\boldsymbol{X}_{\text{train}}, \boldsymbol{y}_{\text{train}})$. Consequently, the tool computes

$$\hat{\boldsymbol{y}}_{\text{val}} = f(\boldsymbol{X}_{\text{val}}), \qquad \text{score}_{\text{val}}(\boldsymbol{q}, \varphi) = M(\hat{\boldsymbol{y}}_{\text{val}}, \boldsymbol{y}_{\text{val}}),$$

where M is the task-specific validation metric, e.g., AUROC for binary classification or negative MAE for regression. The validation response also returns execution feedback, including SQL errors, timeout failures, and model-training errors, allowing the agent to revise later candidate programs.

Reflection via evaluation workspace. After each validation trial, the validation tool records the candidate program and validation artifacts in an *evaluation workspace*. The evaluation workspace up to trial b is a relational database defined as

$$\mathcal{W}^{(\leq b)} = \left\{\widetilde{\mathcal{D}}, \mathcal{T}_{\text{trials}}^{(\leq b)}, \mathcal{T}_{\text{preds}}^{(\leq b)}\right\}$$

where $\widetilde{\mathcal{D}}$ is the augmented context database and for each trial $j \in \{1, \ldots, b\}$ the table $\mathcal{T}_{\text{trials}}^{(\leq b)}$ contains one trial-level record and $\mathcal{T}_{\text{preds}}^{(\leq b)}$ contains one row per evaluated target example with its validation score and ground truth value for each trial. After at least one validation, the agent can inspect the accumulated workspace via `query_eval_workspace()`, which executes evaluation queries against $\mathcal{W}^{(\leq b_{\text{curr}})}$, where b_{curr} is the latest validation index. Prediction records are keyed by trial index and stable `row_id`, with diagnostic metadata stored during validation. This lets the agent align outcomes across trials, inspect errors, aggregate diagnostics, and revise later feature programs, model configurations, or target transformations. See Appendix A for further details.

4.2 Inference phase

After the search, the LLM agent is removed from the prediction pipeline. Let B be the total number of validate actions for the policy rollout. We then simply select the feature program and predictive model that achieved the highest validation score, i.e.,

$$b^\star := \underset{1 \leq b \leq B}{\arg\max}\ \text{score}_{\text{val}}(\boldsymbol{q}^{(b)}, \varphi^{(b)}), \qquad (\boldsymbol{q}^\star, \varphi^\star) := (\boldsymbol{q}^{(b^\star)}, \varphi^{(b^\star)}).$$

In practice, we execute $K = 5$ independent policy rollouts and select the candidate that achieves the highest validation score across all searches, which we refer to as *cross-rollout model selection*. For test evaluation, the feature construction map $\Phi_{\boldsymbol{q}^\star}$ induced by $\boldsymbol{q}^\star$ is applied to train, validation and test tables, yielding $\boldsymbol{X}_{\text{train}}$, $\boldsymbol{X}_{\text{val}}$ and $\boldsymbol{X}_{\text{test}}$ and a predictor $f^\star$ – initialized w.r.t the model configuration $\varphi^\star$ – is fitted to training data $(\boldsymbol{X}_{\text{train}}, \boldsymbol{y}_{\text{train}})$ and validation data $(\boldsymbol{X}_{\text{val}}, \boldsymbol{y}_{\text{val}})$. Finally, test predictions are computed as $\hat{\boldsymbol{y}}_{\text{test}} = f^\star(\boldsymbol{X}_{\text{test}})$.

4.3 Properties

Intrinsic interpretability. Each feature is defined by an explicit SQL query, making predictions directly inspectable through the selected queries and the downstream model's feature usage. We illustrate this in Appendix B, where the agent discovers semantically meaningful feature blocks such as recency, post activity, and commenting behavior.

Expressivity. Since RELAGENT discovers SQL programs during search, its expressivity is bounded by the available query language $\mathcal{L}$ rather than a fixed architecture. SQL can express motifs such as triangle participation via self-joins and distinctness constraints, which standard Message Passing Neural Networks cannot. In Appendix C, the agent discovers a filtered three-way self-join for directed triangle detection, illustrating how query-based predictors can represent logical relational patterns beyond fixed message-passing architectures, though not always within a fixed search budget.

In contrast to automated feature generation approaches such as deep feature synthesis [4], SQL can capture joint conditions across multiple rows of a child table rather than only column-wise aggregations, enabling strictly more expressive feature programs (see Appendix D).

Query-induced invariance. Similar to expressivity, *relational invariance*, i.e., invariance to permutations of row identifiers and table orderings, is induced by the query language $\mathcal{L}$, not a model architecture: as long as each selected SQL feature is relationally invariant, so is the resulting predictor. In our experiments queries that access row identifiers were not explicitly forbidden, however we found that in all final selected programs, only relationally invariant queries were chosen (see Appendix G).

5 Experiments

We evaluate RELAGENT along five axes: performance on classification and regression tasks, component ablations, backbone robustness, search-budget scaling (Appendix E), and qualitative search behavior study (Appendix B). We also include a discussion of Deep Feature Synthesis in Appendix D.

5.1 Setup

Databases and tasks. We evaluate our method on three relational benchmarks, RelBenchV1 [2], RelBenchV2[2] [3], and 4DBInfer [47]. These benchmarks span a diverse set of domains, including e-commerce, social and Q&A platforms, medicine, academia, and sports, and cover a range of temporal prediction tasks such as churn prediction, engagement prediction, and lifetime value (LTV) estimation. Full dataset statistics are provided in Table 14.

Baselines. We compare against four groups of baselines : supervised tabular models, relational deep learning methods, relational foundation models, and LLM as feature engineers approaches:

- **Supervised tabular models.** We include native tabular predictors and relational-to-tabular pipelines: XGBoost [5], LightGBM [6], AutoGluon over flattened tables [7], and human feature-engineering pipelines such as DS+LightGBM, DS+AutoGluon, and DS+TabPFN-2.5 [2, 16].
- **Relational deep learning.** We include graph neural networks such as GraphSAGE [10], GAT [11], PNA [48], and RelGNN [19]; graph transformers such as HGT/HGT_{PE} [49] and RelGT [12]; and language-model-based relational predictors, including LLM_1 [13], LLM_2 [17].
- **Relational foundation models.** We compare against Griffin [14], RT_{zero} [13], Rel-Zero [21], KumoRFM-v1 [15], and KumoRFM-v2 [16]. We also include hybrid or tabular foundation-model pipelines, including GNN+TabPFN-2.5 [20], RDBLearn [9], and TabPFN-2.5 [8].
- **LLMs-as-feature-engineers.** We include FeatLLM [43] and ReFuGe [46] as LLM-based feature-synthesis baselines on RelBenchV1 and report FeatLLM results from Kim et al. [46].

Implementations & evaluations. RELAGENT uses GPT-5.2 [50] as the LLM backbone, CAMEL [51] as the agent framework, and DUCKDB [52] as the SQL engine. The agent selects among seven tree-based predictors: LightGBM variants with standard boosting, DART, and GOSS [6]; Random Forest [53]; XGBoost and XGBoost+DART [5, 54]; and CatBoost [55]. We run five independent search rollouts and select the final model with the highest validation score. We report average AUROC for classification, normalized MAE for regression, and average rank over methods with complete results. See efficiency analysis in Appendix F and the full experimental details in Appendix G. Code is available at `https://github.com/HxyScotthuang/RelAgent`.

5.2 Entity classification

We evaluate RELAGENT on entity classification tasks from RelBenchV1, 4DBInfer, and RelBenchV2, comparing against supervised tabular pipelines, supervised relational models, zero-shot relational foundation models, and LLM-as-feature-engineer. Results are shown in Tables 2 to 4.

Observe that on RelBenchV1, RELAGENT achieves the second-best average AUROC, reaching 78.38, behind only the proprietary relational foundation model KumoRFM-v2 (79.60). On the other hand, RELAGENT obtains the best overall rank (3.17), indicating that its performance is consistently strong across tasks rather than isolated wins.

[2]We did not include MIMIC dataset due to licensing constraints.

Table 2: **Test results on the entity classification tasks in RelBenchV1.** Higher is better (AUROC). We **bold** the best result and underline the second best result.

	Method	f1		avito		event		trial	amazon		stack		hm	Avg	Rank
		dnf	top3	click	visit	repeat	ignore	out	user	item	eng	badge	churn	↑	↓
Sup. Tabular	LightGBM	68.56	73.92	53.60	53.05	68.04	79.93	70.09	52.22	62.54	63.39	63.43	55.21	63.66	18.25
	DS+LightGBM	69.80	82.40	64.27	64.46	75.42	84.23	72.00	67.60	81.80	90.30	86.20	69.00	75.62	8.42
	DS+AutoGluon	70.01	82.53	65.66	65.87	74.85	82.31	70.51	68.12	80.45	89.94	85.38	68.71	75.36	9.25
	DS+TabPFN-2.5	71.16	77.70	64.21	64.53	76.43	85.70	70.75	66.28	79.83	89.07	85.08	68.31	74.92	10.25
Sup. Relational	GraphSAGE	72.62	75.54	65.90	66.20	76.89	81.62	68.60	70.42	**82.81**	90.59	88.86	69.88	75.83	6.75
	HGT	70.77	70.75	63.76	64.32	64.96	82.47	58.37	66.43	77.97	88.47	86.08	66.95	71.78	13.58
	HGT_{PE}	71.17	76.27	64.57	64.95	65.36	81.61	59.21	66.19	78.03	88.17	85.66	65.69	72.24	12.75
	RelGNN	75.29	85.69	68.23	66.18	79.61	86.18	71.24	**70.99**	82.64	**90.75**	**88.98**	70.93	78.06	3.83
	RelGT	75.87	83.52	68.30	66.78	76.09	81.57	68.61	70.39	82.55	90.53	86.32	69.27	76.65	6.38
Zero-Shot Foundational	LLM_1	75.80	91.40	59.80	62.70	71.40	69.30	57.40	58.10	62.10	78.00	80.00	59.80	68.82	15.04
	LLM_2	80.03	87.11	53.36	54.07	70.11	68.65	59.17	62.55	73.41	81.23	79.99	63.81	69.46	14.75
	Rel-Zero	71.84	70.64	62.28	56.17	68.12	61.32	59.02	60.07	64.10	69.46	62.12	55.95	63.42	17.67
	TabPFN-2.5	61.21	77.35	54.44	51.45	69.10	70.91	71.74	57.14	59.30	59.66	52.26	54.99	61.63	18.08
	Griffin	57.70	82.50	45.90	60.70	71.88	83.27	51.00	62.30	69.00	77.50	73.50	60.20	66.29	16.58
	RT_{zero}	81.20	89.30	59.50	61.80	73.22	77.47	51.80	64.00	70.90	75.70	80.10	62.80	70.65	14.08
	GNN+TabPFN	64.26	74.67	60.68	65.35	73.58	84.13	56.68	64.30	77.89	87.48	85.34	64.34	71.56	13.75
	RDBLearn	70.87	79.69	**69.04**	65.49	75.04	82.52	71.58	67.57	82.07	89.39	85.26	68.05	75.55	8.92
	KumoRFM-v1	82.41	91.07	64.85	64.11	76.08	89.20	70.79	67.29	79.93	87.09	80.00	67.71	76.71	8.71
	KumoRFM-v2	**84.59**	**92.18**	67.42	**69.41**	**81.66**	**90.83**	72.03	69.10	82.17	89.40	87.15	69.27	**79.60**	3.38
LLM as FE.	FeatLLM	70.43	52.72	48.69	51.15	46.67	47.88	57.35	-	55.11	42.56	-	-	-	-
	ReFuGe	76.26	83.04	63.20	66.43	77.14	83.40	**72.22**	-	67.00	87.66	83.80	68.10	-	-
	RELAGENT	78.34	85.23	68.36	67.79	78.20	87.25	71.86	70.78	82.84	90.41	88.42	**71.07**	78.38	**3.17**

Supervised models. RELAGENT substantially improves over LightGBM on raw tables and outperforms human feature-engineering pipelines with LightGBM, AutoGluon, and TabPFN-2.5. This suggests that the agent can automate relational feature engineering while remaining competitive with human-designed pipelines. RELAGENT achieves the highest average AUROC and the best average rank, outperforming all relational supervised baselines. Thus, agentic search via SQL is competitive with end-to-end supervised relational architectures on this benchmark.

Foundation models. RELAGENT also outperforms all open-source zero-shot foundation-models in average AUROC. Only the closed-source KumoRFM-v2 achieves a higher average AUROC (79.60 vs. 78.38), while RELAGENT obtains the best average rank (3.17 vs. 3.38). Unlike foundation models used directly as predictors, RELAGENT uses the LLM only during search; inference is performed without further LLM calls.

LLM as feature engineers. We also report FeatLLM and ReFuGe as LLM-based feature-engineering references. On overlapping tasks, RELAGENT is competitive with or stronger than ReFuGe, while supporting evaluation-workspace queries and joint feature/model selection.

Table 3: **Test results on the entity classification tasks in RelBenchV2.** Higher is better (AUROC). We **bold** the best and underline the second best result.

Method	ratebeer			arxiv	Avg	Rank
	beer	user	dormant	citation	↑	↓
LightGBM	76.21	83.92	75.79	71.21	76.78	4.75
GraphSAGE	78.67	94.27	80.51	82.50	83.99	2.75
KumoRFM-v1	75.06	91.38	77.10	80.62	81.04	4.25
KumoRFM-v2	83.84	97.43	80.65	81.71	85.91	2.25
RELAGENT	**84.70**	**98.63**	**83.33**	**82.62**	**87.32**	**1.00**

Table 4: **Test results on the entity classification tasks in 4DBInfer.** Higher is better (AUROC). We **bold** the best and underline the second best result.

	Method	AB	OB	RR	SE		Avg	Rank
		churn	ctr	cvr	vote	churn	↑	↓
Sup. Tab.	XGBoost	50.00	50.00	50.00	49.68	50.84	50.10	11.50
	AutoGluon	50.00	49.69	50.96	50.81	50.00	50.29	11.50
	DFS+XGBoost	69.22	54.21	79.06	86.75	82.51	74.35	10.00
	DFS+AutoGluon	72.91	54.94	80.08	88.49	83.96	76.08	8.40
Sup. Rel.	GAT	76.22	61.46	82.84	88.53	86.45	79.10	6.40
	HGT	77.30	62.60	84.95	88.17	86.70	79.94	4.20
	PNA	76.45	62.49	83.67	88.96	86.64	79.64	4.60
	GraphSAGE	75.71	62.39	84.70	88.61	85.58	79.40	5.20
Found.	RDBLearn	77.41	54.47	84.69	88.45	87.96	78.60	5.00
	KumoRFM-v1	75.59	61.11	84.36	88.92	87.87	79.17	5.50
	KumoRFM-v2	77.06	61.97	84.36	88.50	87.92	79.96	4.70
	RELAGENT	**79.44**	**62.63**	**86.35**	**89.45**	**89.03**	**81.38**	**1.00**

Table 5: **Test results on entity regression tasks in RelBenchV1.** Lower is better (MAE). Average performance $\boldsymbol{\mu}_n$ is normalized relative to the LightGBM baseline. We **bold** the best result and underline the second best result.

	Method	f1	avito	event	trial		amazon		stack	hm	$\boldsymbol{\mu}_n$	Rank
		pos	ctr	attend	adverse	success	user	item	votes	sales	↓	↓
	Global Median	4.399	0.043	0.264	57.533	0.462	16.783	64.234	0.068	0.076	1.062	13.22
	Entity Median	8.519	0.046	0.269	57.930	0.441	17.423	66.436	0.069	0.078	1.190	15.06
Sup. Tabular	LightGBM	4.170	0.041	0.264	44.011	0.425	16.783	60.569	0.068	0.076	1.000	10.61
	DS+LightGBM	3.963	0.044	0.284	40.581	0.407	13.928	**41.122**	0.065	0.036	0.879	5.56
	DS+AutoGluon	4.251	0.045	0.256	44.706	0.430	14.399	45.390	0.068	0.043	0.918	8.39
	DS+TabPFN-2.5	4.373	0.044	0.318	47.168	0.432	15.631	47.908	0.080	0.059	1.009	11.11
Sup. Relational	GraphSAGE	4.022	0.041	0.258	44.473	0.400	14.313	50.053	0.065	0.056	0.918	7.17
	HGT	4.226	0.046	0.264	45.169	0.443	15.412	55.868	0.068	0.064	0.988	11.61
	HGT_{PE}	4.392	0.048	0.261	42.648	0.440	15.864	55.849	0.068	0.064	0.992	10.83
	RelGNN	3.798	0.037	0.238	44.461	**0.301**	14.230	48.767	0.065	0.054	0.861	4.94
	RelGT	3.917	0.035	0.250	43.992	0.326	14.267	48.922	0.065	0.054	0.869	5.33
Zero-shot Foundational	TabPFN-2.5	4.446	0.044	0.527	55.187	0.469	17.513	60.996	0.104	0.096	1.258	15.78
	Griffin	4.460	0.050	0.461	78.232	0.463	35.590	53.214	0.092	0.151	1.471	16.44
	RT_{zero}	2.901	0.058	0.379	73.999	0.455	18.802	57.996	0.110	0.089	1.240	14.89
	RDBLearn	3.834	0.034	0.237	43.913	0.424	14.540	48.559	0.068	0.064	0.906	6.33
	KumoRFM-v1	**2.747**	0.035	0.264	58.231	0.417	16.161	55.254	0.065	0.040	0.908	7.78
	KumoRFM-v2	2.854	0.034	**0.235**	43.293	0.433	**13.921**	46.992	**0.064**	**0.034**	0.822	3.11
	RELAGENT	4.019	**0.033**	0.241	**37.194**	0.386	13.949	41.765	**0.064**	0.035	**0.817**	**2.83**

RELAGENT on RelBenchV2 and 4DBInfer. RELAGENT achieves the best AUROC on all RelBenchV2 and 4DBInfer entity classification tasks, improving the averages from KumoRFM-v2 from 85.91 to 87.32 and from 79.96 to 81.38, respectively; see Tables 3 and 4. On 4DBInfer, it also outperforms DFS-based baselines, suggesting that guided SQL search can improve over generic relational aggregation. Unlike fixed DFS aggregations, SQL feature programs can express task-specific row-level interactions, such as co-occurrence predicates within a related table; see Appendix D.

5.3 Entity regression

We evaluate RELAGENT on entity regression tasks in RelBenchV1 and RelBenchV2; results are shown in Tables 5 and 6. RELAGENT achieves the best normalized average performance and average rank on RelBenchV1, improving over KumoRFM-v2, and performs best on both RelBenchV2 regression tasks.

Table 6: **Test results on the regression tasks in RelBenchV2.** Lower is better (MAE). Average performance μ_n is normalized relative to the LightGBM baseline. We **bold** the best and underline the second best result among all models.

Method	ratebeer	arxiv	μ_n	Rank
	user-count	publication	↓	↓
Global Median	15.124	0.577	0.872	5.50
Entity Median	13.079	0.874	1.079	5.50
LightGBM	20.350	0.577	1.000	6.00
GraphSAGE	7.374	0.513	0.626	3.00
KumoRFM-v1	11.063	0.518	0.721	4.00
KumoRFM-v2	7.298	0.487	0.601	2.00
RELAGENT	**6.021**	**0.462**	**0.548**	**1.00**

We attribute this to the flexibility of program-level search. Regression tasks in these benchmarks often require task-specific choices beyond relational aggregation: targets such as lifetime value and sales are often heavy-tailed, and different tasks benefit from different temporal windows, model families, and objectives. RELAGENT searches over these choices jointly. In the selected programs, the agent consistently chooses L_1 objectives and applies log-target transformations in seven of nine RelBenchV1 tasks, while also selecting task-specific SQL features and model configurations. This suggests that the gains come from optimizing the full predictive program, rather than from SQL feature construction alone.

5.4 Ablations

We perform ablations to isolate the contribution of iterative feedback, semantic schema information, evaluation workspace, as well as the downstream supervised predictor and backbone LLM choice.

Unless otherwise stated, all ablations use the same evaluation protocol as the main experiments and report the same task-level metrics aggregated across datasets. We report the averaged ablation results in Table 7, and the full results in Table 15.

Iterative feedback loop. To isolate the value of iterative validation and revision, we compare RELAGENT with a variant that does not allow program validation tools and is asked to output the final program and predictive model configuration directly. Removing this loop reduces average AUROC from 78.38 to 66.86, suggesting that the gains come not only from SQL features, but also from repeated validation, where the agent observes execution outcomes, debugs failures, and revises subsequent feature programs and model choices.

Evaluation workspace. We compare RELAGENT with a variant that receives validation feedback but cannot access the accumulated workspace $\mathcal{W}^{(\leq b)}$. Removing this access reduces average AUROC from 78.38 to 75.78, showing that the workspace provides signal beyond scalar validation scores. In the `rel-stack/eng` case study in Appendix B, the agent uses workspace diagnostics to identify constant or low-SHAP vote features and replaces them with voting-breadth features, such as the number of distinct posts voted on within recent windows. This illustrates how the workspace supports targeted error analysis.

Table 7: **Ablation results on the entity classification tasks in RelBenchV1.** Higher is better (AUROC).

Model Variants	Avg ↑
RELAGENT	**78.38**
w/o feedback	66.86
w/o eval workspace	75.78
w/o semantic	74.44
rule-only	69.92
catboost-only	75.95
lightgbm-only	75.94
xgboost-only	73.42
xgboost-dart-only	74.97
tabpfn-v2.5-only	78.13
with Deepseek-V4	76.00
with Qwen3-32B	70.13

Semantic schema information. To assess the role of database semantics, we compare against an anonymized-schema variant where dataset names, table names, column names, and task descriptions are replaced by synthetic identifiers. This reduces average AUROC from 78.38 to 74.44. A closer look at the trace suggests that semantic clues help the agent identify task-relevant features such as recency, activity frequency, trends, and relational paths; without them, it falls back more often to generic counts, missingness statistics, and broad aggregates. The anonymized variant still performs reasonably well, indicating that tool-grounded search can recover useful relational signals even without human-readable schema semantics.

Predictor choice. We compare RELAGENT with fixed-predictor variants while keeping SQL feature search unchanged. Restricting the predictor to CatBoost, LightGBM, XGBoost, or XGBoost-DART reduces average AUROC from 78.38 to 75.95, 75.94, 73.42, and 74.97, respectively. Notably, TabPFN-2.5 achieves 78.13, approaching the performance of the full system with a fixed downstream predictor. A rule-only variant, where the agent writes Python rules to combine SQL-derived features without fitting a supervised predictor, achieves only 69.92. These results indicate that task-specific SQL feature discovery should be paired with an effective learned downstream predictor.

Backbone robustness. We evaluate alternative LLM backbones by rerunning the full pipeline with the same tools, search budget, and evaluation protocol. We choose DeepSeek-V4-Pro [56] as a strong lower-cost backbone for token-intensive search. We also evaluate Qwen3-32B [57] as a much smaller open-source backbone that can be hosted locally. DeepSeek-V4 reaches 76.00 average AUROC, close to the GPT-5.2 result of 78.38, while Qwen3-32B reaches 70.13. Thus, stronger backbones improve search quality, while the tool-augmented design remains effective with smaller models.

SQL feature discovery. We evaluate the quality of SQL features discovered by RELAGENT while fixing the downstream predictor to TabPFN-2.5, shown in Table 8 (a). Compared with Deep Feature Synthesis (DFS), human-curated features, and entity-table-only features, RELAGENT achieves the highest average AUROC of 78.13, outperforming DFS (75.55) and human-curated features (74.92) by 2.58 and 3.21 points, re-

Table 8: **SQL feature analysis.** (a) Average AUROC on RelBenchV1 classification with a fixed TabPFN-2.5 predictor. (b) Test AUROC under restricted SQL operations on three representative tasks.

(a) SQL feature discovery

Feature representation	AUROC ↑
Entity-table only	61.63
Human-curated	74.92
DFS	75.55
RELAGENT	**78.13**

(b) SQL expressiveness ablations

Task	Full	1-hop	No temporal	No cond.
hm-churn	**71.07**	70.47	68.34	69.10
amazon-user	**70.78**	69.33	64.69	70.43
amazon-item	**82.84**	82.52	73.90	82.08

spectively. This demonstrates that the gains are not solely attributable to downstream predictor selection, but also arise from discovering task-specific relational features.

SQL expressiveness. We investigate which relational operations contribute to RELAGENT's predictive performance by restricting the SQL search space to one-hop joins, removing temporal features, or disabling conditional aggregations. As shown in Table 8 (b), removing temporal features causes the largest performance degradation across all three tasks, reducing AUROC by 2.73, 6.09, and 8.94 points, respectively. In contrast, restricting join depth or conditional aggregations leads to smaller degradations. These results suggest that task-specific temporal feature construction is a particularly important benefit of SQL program synthesis.

6 Conclusion

We introduced RELAGENT, an LLM-agent framework for relational learning that searches over SQL feature programs and downstream tabular predictors. By keeping relational structure external and executable, RELAGENT uses the LLM only during search; the deployed predictor consists only of SQL feature construction and a standard predictive model. This yields fast, deterministic, and inspectable predictions without LLM calls at inference time.

Empirically, RELAGENT achieves competitive or state-of-the-art results across classification and regression tasks, often matching or exceeding supervised relational models and relational foundation-model baselines. These results suggest that program-level search is a strong alternative to fixed relational architectures, especially when tasks require task-specific joins, temporal windows, objectives, or target transformations. Its main limitations are search cost and imperfect search reliability: the agent may miss useful deep joins, high-order relational patterns, or features from long free-text fields within a fixed budget; see Appendix J. Overall, RELAGENT shows that LLMs can construct executable relational learning programs rather than serve only as direct predictors.

Acknowledgment

This work was supported in part by the National Research Foundation of Korea (RS-2024-00351212) funded by the Korean government (MSIT). AITHYRA is supported by the Austrian Academy of Sciences and the not-for-profit Boehringer Ingelheim Stiftung.

References

[1] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 1970.

[2] J. Robinson, R. Ranjan, W. Hu, K. Huang, J. Han, A. Dobles, M. Fey, J. E. Lenssen, Y. Yuan, Z. Zhang, X. He, and J. Leskovec. RelBench: A benchmark for deep learning on relational databases. In *NeurIPS*, 2024.

[3] Justin Gu, Rishabh Ranjan, Charilaos Kanatsoulis, Haiming Tang, Martin Jurkovic, Valter Hudovernik, Mark Znidar, Pranshu Chaturvedi, Parth Shroff, Fengyu Li, et al. Relbench v2: A large-scale benchmark and repository for relational data. *arXiv preprint arXiv:2602.12606*, 2026.

[4] 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)*. IEEE, 2015.

[5] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016.

[6] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T. Liu. LightGBM: A highly efficient gradient boosting decision tree. In *NeurIPS*, 2017.

[7] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola. AutoGluon-Tabular: Robust and accurate automl for structured data. *CoRR*, 2020.

[8] N. Hollmann, S. Müller, L. Purucker, A. Krishnakumar, M. Körfer, S. B. Hoo, R. T. Schirrmeister, and F. Hutter. Accurate predictions on small data with a tabular foundation model. *Nature*, 2025.

[9] 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.

[10] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NeurIPS*, 2017.

[11] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. In *ICLR*, 2018.

[12] Vijay Prakash Dwivedi, Sri Jaladi, Yangyi Shen, Federico Lopez, Charilaos I. Kanatsoulis, Rishi Puri, Matthias Fey, and Jure Leskovec. Relational graph transformer. In *ICLR*, 2026.

[13] Rishabh Ranjan, Valter Hudovernik, Mark Znidar, Charilaos I. Kanatsoulis, Roshan Reddy Upendra, Mahmoud Mohammadi, Joe Meyer, Tom Palczewski, Carlos Guestrin, and Jure Leskovec. Relational transformer: Toward zero-shot foundation models for relational data. In *ICLR*, 2026.

[14] Yanbo Wang, Xiyuan Wang, Quan Gan, Minjie Wang, Qibin Yang, David Wipf, and Muhan Zhang. Griffin: Towards a graph-centric relational database foundation model. In *ICML*, 2025.

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

[16] 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.

[17] M. Wydmuch, L. Borchmann, and F. Graliński. Tackling prediction tasks in relational databases with llms. *CoRR*, 2411.11829, 2024.

[18] Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. Talk like a graph: Encoding graphs for large language models. In *ICLR*, 2024.

[19] Tianlang Chen, Charilaos Kanatsoulis, and Jure Leskovec. Relgnn: Composite message passing for relational deep learning. In *ICML*, 2025.

[20] Joe Meyer, Tom Palczewski, Afreen Shaikh, Mahmoud Mohammadi, Dinesh K. Ramprasath, Karan Paresh, Roshan U. Reddy, and Mark Li. Relational in-context learning on structured data via neighborhood aggregation and structural information. In *Proceedings of the AAAI-26 Summer Symposium Series: AI in Business: Intelligent Transformation and Management*, 2026.

[21] Fang Wu, Vijay Prakash Dwivedi, and Jure Leskovec. Large language models are good relational learners. In *ACL*, 2025.

[22] Mikhail Galkin, Xinyu Yuan, Hesham Mostafa, Jian Tang, and Zhaocheng Zhu. Towards foundation models for knowledge graph reasoning. In *ICLR*, 2024.

[23] Xingyue Huang, Pablo Barceló, Michael M. Bronstein, İsmail İlkan Ceylan, Mikhail Galkin, Juan L Reutter, and Miguel Romero Orth. How expressive are knowledge graph foundation models? In *ICML*, 2025.

[24] Xingyue Huang, Mikhail Galkin, Michael M Bronstein, and Ismail Ilkan Ceylan. Hyper: A foundation model for inductive link prediction with knowledge hypergraphs. In *ICLR*, 2026.

[25] Yucheng Zhang, Beatrice Bevilacqua, Mikhail Galkin, and Bruno Ribeiro. TRIX: A more expressive model for zero-shot domain transfer in knowledge graphs. In *LoG*, 2024.

[26] Jinwoo Kim, Xingyue Huang, Krzysztof Olejniczak, Kyungbin Min, Michael Bronstein, Seunghoon Hong, and İsmail İlkan Ceylan. Flock: A knowledge graph foundation model via learning on random walks. In *ICLR*, 2026.

[27] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. Tabert: Pretraining for joint understanding of textual and tabular data. In *ACL*, 2020.

[28] Qian Liu, Bei Chen, Jiaqi Guo, Morteza Ziyadi, Zeqi Lin, Weizhu Chen, and Jian-Guang Lou. Tapex: Table pre-training via learning a neural sql executor. In *ICLR*, 2022.

[29] Stefan Hegselmann, Alejandro Buendia, Hunter Lang, Monica Agrawal, Xiaoyi Jiang, and David Sontag. Tabllm: Few-shot classification of tabular data with large language models. In *International Conference on Artificial Intelligence and Statistics*, 2023.

[30] Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. In *NeurIPS*, 2023.

[31] Jiabin Tang, Yuhao Yang, Wei Wei, Lei Shi, Lixin Su, Suqi Cheng, Dawei Yin, and Chao Huang. Graphgpt: graph instruction tuning for large language models. In *SIGIR*, 2024.

[32] Ruosong Ye, Caiqi Zhang, Runhui Wang, Shuyuan Xu, and Yongfeng Zhang. Language is all a graph needs. *EACL*, 2024.

[33] Yuanfu Sun, Zhengnan Ma, Yi Fang, Jing Ma, and Qiaoyu Tan. Graphicl: Unlocking graph learning potential in llms through structured prompt design. In *Findings of the Association for Computational Linguistics: NAACL 2025*, 2025.

[34] Bryan Perozzi, Bahare Fatemi, Dustin Zelle, Anton Tsitsulin, Mehran Kazemi, Rami Al-Rfou, and Jonathan Halcrow. Let your graph do the talking: Encoding structured data for llms. *arXiv preprint arXiv:2402.05862*, 2024.

[35] Runjin Chen, Tong Zhao, Ajay Kumar Jaiswal, Neil Shah, and Zhangyang Wang. Llaga: Large language and graph assistant. In *ICML*, 2024.

[36] 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.

[37] 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. In *NeurIPS*, 2023.

[38] 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 *ACL*, 2023.

[39] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

[40] Rongzheng Wang, Shuang Liang, Qizhi Chen, Jiasheng Zhang, and Ke Qin. Graphtool-instruction: Revolutionizing graph reasoning in llms through decomposed subtask instruction. In *KDD*, 2025.

[41] Ben Finkelshtein, Silviu Cucerzan, Sujay Kumar Jauhar, and Ryen W White. Actions speak louder than prompts: A large-scale study of llms for graph inference. In *ICLR*, 2026.

[42] Noah Hollmann, Samuel Müller, and Frank Hutter. Large language models for automated data science: Introducing caafe for context-aware automated feature engineering. In *NeurIPS*, 2023.

[43] Sungwon Han, Jinsung Yoon, Sercan O Arik, and Tomas Pfister. Large language models can automatically engineer features for few-shot tabular learning. In *ICML*, 2024.

[44] 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.

[45] 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. In *NeurIPS*, 2024.

[46] 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.

[47] 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 rdbs. In *NeurIPS*, 2024.

[48] G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Veličković. Principal neighbourhood aggregation for graph nets. In *NeurIPS*, 2020.

[49] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous graph transformer. In *Proceedings of The Web Conference 2020*, 2020.

[50] OpenAI. Introducing gpt-5.2. https://openai.com/index/introducing-gpt-5-2/, 2025. Accessed: 2026-04-22.

[51] Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for" mind" exploration of large language model society. In *NeurIPS*, 2023.

[52] Hannes Mühleisen and Mark Raasveldt. *duckdb: DBI Package for the DuckDB Database Management System*, 2026.

[53] Leo Breiman. Random forests. *Machine learning*, 2001.

[54] R Gilad-Bachrach and K Rashmi. Dart: Dropouts meet multiple additive regression trees. In *AIStats*, 2015.

[55] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. Catboost: Unbiased boosting with categorical features. *arXiv preprint arXiv:1706.09516*, 2017.

[56] DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. `https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro/blob/main/DeepSeek_V4.pdf`, 2026. Technical report.

[57] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[58] DeepSeek. DeepSeek API pricing. `https://api-docs.deepseek.com/quick_start/pricing`, 2026.

[59] OpenAI. GPT-5.2 API pricing. `https://openai.com/api/pricing/`, 2026.

A Evaluation workspace details

The *evaluation workspace* is a persistent DUCKDB database that accumulates trial results across the agent's search phase. After each call to `validate_program()`, the harness writes the outcome to the workspace; the agent reads it back through `query_eval_workspace(sql)`, which executes any read-only SQL query against the accumulated records.

Core tables. The base workspace maintains two tables, illustrated in Table 9[3]. The `trials` table stores one record per validation trial, capturing the trial identifier, the names and a stable content hash of the submitted SQL feature blocks, the chosen downstream model, the prediction split, the primary validation metric, and a JSON serialization of all computed metrics. The `eval_predictions` table stores one row per evaluated target example, recording the entity identifier, ground-truth label, model output score, and optionally predicted class. The key design choice is that `row_id` is positionally stable across trials on the same split: it is the integer index of the row in the validation target table, assigned once and reused. This allows the agent to join two trials directly on `row_id` and compare predictions on the same examples.

A representative diagnostic query used by the agent that compares entity-level errors between two trials and finds examples of the 3rd trial fixed is:

```
SELECT p1.entity_id, p1.label, p1.score AS score_v1, p3.score AS score_v3
FROM eval_predictions p1
JOIN eval_predictions p3 ON p1.row_id = p3.row_id
WHERE p1.trial_id = 'val_0001' AND p3.trial_id = 'val_0003'
  AND ABS(p3.score - CAST(p3.label AS DOUBLE))
    < ABS(p1.score - CAST(p1.label AS DOUBLE))
ORDER BY ABS(p1.score - CAST(p1.label AS DOUBLE)) DESC LIMIT 20
```

Evaluation query usage. To characterize how the agent uses `query_eval_workspace()` in practice, we analyzed agent runs across all benchmarks. A majority of runs issue at least one evaluation query, and Figure 2 summarizes the distribution of these queries by intent. The dominant patterns reflect an iterative diagnostic strategy:

(1) *Trial tracking* (20.6%) covers queries that rank or retrieve trial records by validation score to monitor search progress.
(2) *Error analysis* (17.8%) computes per-row prediction errors, often sorted to expose the worst-predicted examples.
(3) *Score and distribution statistics* (10.5%) aggregates prediction scores and class-conditional summaries for a single trial.
(4) *Cross-trial prediction comparison* (8.5%) joins two trials on the stable `row_id` to identify examples whose predictions changed after a feature revision.
(5) *Prediction inspection* (7.1%) retrieves ranked lists of false-positive or false-negative examples.
(6) *Feature importance ranking* (6.0%) and *feature diagnostic checks* (5.2%) inspect feature-level signals to identify low-value or potentially defective feature blocks.
(7) The remaining queries (24.2%) cover miscellaneous aggregates, metadata lookups, and other auxiliary diagnostics.

Together, these patterns reveal a structured self-diagnostic loop: the agent first tracks which trial is best, then drills into row-level failures, compares the impact of individual feature changes, and finally uses feature-level evidence to diagnose possible causes and guide the next feature revision.

Anti-leakage constraint. Workspace tables are analysis-only: they contain validation predictions and labels, and must not be used by feature queries or predictive models. The `query_eval_workspace()` interface is a read-only diagnostic interface over the evaluation workspace

$$\mathcal{W}^{(\le b)} = \left\{ \widetilde{\mathcal{D}}, \mathcal{T}_{\text{trials}}^{(\le b)}, \mathcal{T}_{\text{preds}}^{(\le b)} \right\}.$$

[3]Table shows key fields; the full schema additionally includes `parent_trial_id`, `created_at`, `resolved_model_config`, `notes` in `trials` and `eval_cutoff` in `eval_predictions`.

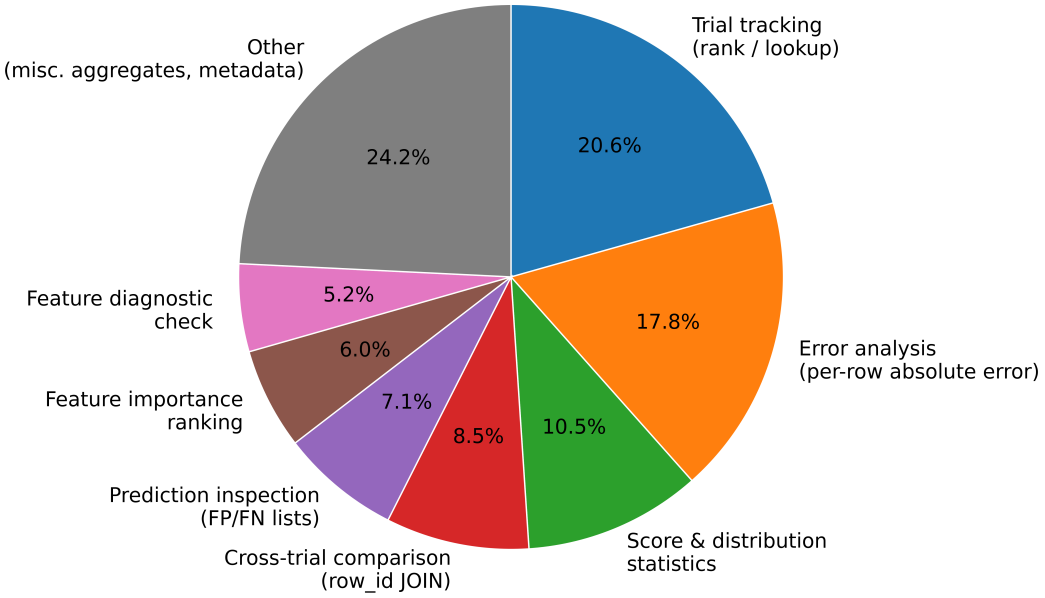


Figure 2: Distribution of evaluation workspace queries across all agent runs.

Field	Role
`trial_id`	primary key
`trial_name`	trial label
`split`	data split
`model_choice`	model configuration
`feature_block_names`	feature groups
`feature_query_hash`	program identifier
`primary_metric`	validation metric
`primary_score`	validation score
`metrics_json`	serialized metrics

(a) `trials`

Field	Role
`trial_id`	foreign key to `trials`
`row_id`	stable row index
`entity_id`	target entity
`label`	ground-truth label
`score`	predicted score
`predicted_class`	predicted class
`split`	data split

(b) `eval_predictions`

Table 9: **Schema of the evaluation workspace.** The `trials` table stores trial-level metadata, model configurations, feature-program identifiers, and validation metrics. The `eval_predictions` table stores row-level validation outputs for each trial. The field `row_id` is stable within each split, enabling cross-trial comparisons on the same target examples.

Thus, evaluation queries may join validation outcomes back to tables in $\widetilde{\mathcal{D}}$ for error analysis, for example, to aggregate errors by entity attributes, timestamps, or relational context. However, these queries are never used as predictive features. Candidate feature programs submitted to `validate_program()` are executed only through the feature-query path, and the validation harness does not expose workspace tables such as `trials` or `eval_predictions` to feature construction during validation or inference.

Point-in-time correctness. For each validation or test split, the system exposes the target rows through `eval_table`, which contains `row_id`, the entity key, and the prediction timestamp. Feature queries are evaluated against this split-specific table and are merged by `row_id`. Thus, two target rows with the same entity but different timestamps can receive different feature values. The prompts instruct the agent to filter all timestamped source tables using predicates of the form `source_time < eval_table.timestamp`. In addition, our validation system checks that submitted feature queries are anchored on `eval_table` and return `row_id`; queries that cannot be aligned row-wise are rejected.

B Case study

To better understand the search process of RELAGENT, we analyze one complete agent run in detail. The goal of this case study is to characterize how the agent structures its search.

We study a run on the `user-engagement` task using the `rel-stack` dataset. Stack Exchange is a network of question-and-answer websites organized around different topics. The prediction task is to determine, for a user at a given prediction timestamp, whether that user will make any votes, posts, or comments in the following three months.

B.1 Agent search summary

The analyzed run consists of 45 conversation turns and 30 validation trials. Of these validation trials, 28 produced validation scores, while 2 failed due to execution issues: one timeout and one SQL error. The agent made 16 direct SQL exploration calls over the training database, consisting of one table-listing call, eight schema/table-info calls, and seven exploratory SQL queries. It also made 7 calls to the evaluation workspace, which stores validation histories and row-level predictions from previous trials.

Across the run, the agent proposed 178 distinct feature columns in successful validation diagnostics and ultimately selected a final champion program with 45 merged features using the XGBoost DART model. The complete conversation trace is provided in the `https://anonymous.4open.science/r/RelAgent`.

We manually segmented the agent's search into four semantic blocks. This segmentation is not used by the agent itself; rather, it is an analysis of the trace intended to expose the structure of the search.

- **Initial database exploration and baseline programs (Turn 0-1).** The agent first inspects the database structure. It lists the available tables, including `badges` [4], `comments`, `post history`, `post links`, `posts`, `users`, `votes`, and the training table, and then queries schemas and row counts. It observes that the classification problem is strongly imbalanced. The first submitted program uses LightGBM with standard gradient boosting and contains three main feature groups: user-profile features, activity counts over posts, comments, and votes, and recency features measuring time since previous activity. This first validation reaches about 71.72 AUROC on the validation set. A second, similar validation gives nearly the same score.
- **Revision of vote and activity features using validation feedback. (Turn 2-6).** After the initial baselines, the agent consults previous trial results in the evaluation workspace and notices that vote-related features, such as upvotes, downvotes, and accepted-vote counts, were either constant or had low SHAP value. The agent removes these columns and replaces them with voting-*breadth* features, counting the number of *distinct* posts a user voted on within recent windows. This produces the first clear performance increase, with validation AUROC rising to about 78.20. At this point, the feature program contains the four semantic blocks that remain the core for most later trials: recency, activity windows, comments and votes, and user profile. The agent then tries additional aggregates, such as active-day counts and velocity ratios, but these expanded variants reduce validation performance and the agent decides to focus on model changes instead.
- **Model-family search with the core SQL fixed. (Turn 7-28).** In this phase the agent explores variations of the predictive model while keeping the feature set mostly fixed. It first evaluates a DART variant of LightGBM and improves the validation score to about 78.80. Other LightGBM configurations, including GOSS, stronger regularization, smaller tree depth, and different learning rates, do not improve over this DART configuration. The agent then switches to *XGBoost*-DART, again using the same four core SQL blocks. This produces a large jump in validation AUROC to approximately 89.90. It then validates nearby XGBoost-style configurations, varying depth, learning rate, and child-weight parameters, and also compares against plain XGBoost and CatBoost. These trials remain close to the XGBoost-DART score but do not exceed the first XGBoost-DART trial.
- **Incremental feature expansion on top of the fixed core (Turn 29-44).** In the final part of the run, the agent keeps XGBoost-DART and the four-query core fixed while adding feature blocks

[4] *Badges* on Stack Exchange are achievement-based indicators awarded to users for specific contributions and participation milestones within the platform's community.

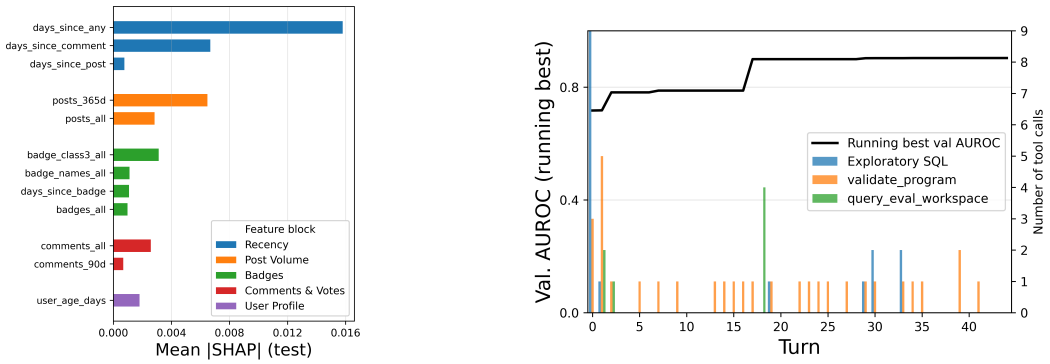
from additional tables: It adds badge-based features, including recent badge counts, and then adds post-history features based on user-related events in the post-history table. Both additions slightly improve validation performance, reaching about 90.30 AUROC. The agent also tests post-link features, but these lower the score and are not retained. The badge features are then refined with badge-class splits, days since last badge, and badge-velocity ratios. A richer post-history variant is also tried but later reverted after a small performance drop. Trial 29, which combines the four core blocks with badge and post-history is selected as the final champion.

An overview of the number of tool calls per turn as well as the running best validation AUROC is provided in Figure 3b).

B.2 Target features found by the agent

Since the final predictor is based on explicit SQL features, its representation can be inspected directly. Note that a single SQL query can construct multiple feature columns. We refer to all features originating from the same query as one feature *block*. The final program in the inspected run consists of seven such blocks. Figure 3a reports the mean absolute SHAP values of all features with mean absolute SHAP value greater than 0.0005, grouped by feature block. Features within a block are typically semantically related.

The most important block, **Recency**, measures how recently the user was active before the prediction time. It includes features such as the number of days since the user's last post, comment, or vote. A second block, **Post Volume**, captures how much the user has posted historically and in recent rolling windows, including lifetime question and answer counts as well as windowed post activity. The **Badges** block represents badge acquisition behavior, including badge counts, distinct badge names, class-specific badge counts, recency of the last badge, and short-versus-long window velocity ratios. The **Comments and Votes** block measures comment and voting behavior in rolling windows, including the number of distinct posts voted on. This block emerged from the agent's correction of constant vote-type features during the search. Finally, the **User Profile** block contains basic user-level information, primarily account age and whether the user appears in the user table.



(a) **Feature-block importance** (b) **Search trajectory**

Figure 3: Case-study summary for the inspected `rel-stack/user-engagement` run. Figure (a) shows the mean absolute SHAP values of the most influential test-set features, grouped by the feature block from which they originate. Figure (b) shows the running best validation AUROC over conversation turns together with the number of tool calls made in each turn.

C Synthetic experiments on directed triangle detection

To evaluate whether the agent can discover a precise relational pattern from labeled examples alone without any schema hints about the underlying rule, we construct a controlled synthetic benchmark based on directed triangle membership.

Setup. Each dataset consists of a single directed edge relation `R(src, dst)` drawn from an Erdős–Rényi random graph ($n = 300$ nodes, edge probability $p = 0.02$). Training uses 5 independent



Figure 4: **Synthetic directed triangle tasks.** Green nodes indicate positives (satisfying $Q(x)$) and gray nodes indicate negatives (not satisfying $Q(x)$). Left: loop-free setting. Right: self-loop setting.

graphs (1500 nodes total); validation and test each use 2 graphs (600 nodes). Node IDs are globally unique across graphs, so the agent must learn a *transferable* structural rule rather than memorize node identities. The agent can write SQL feature queries over relational table `R`, and a LightGBM classifier is trained automatically on those features. The agent observes validation AUROC after each trial and iteratively refines its queries up to a budget of 30 trials, the same as the setup in the main experiments.

We distinguish two closely related predicates. The first is the directed 3-walk closure predicate

$$P(x) = \exists y, z : \ \mathtt{R}(x, y) \wedge \mathtt{R}(y, z) \wedge \mathtt{R}(z, x),$$

which can be computed by a three-way self-join. The second is the *proper* directed 3-cycle predicate

$$Q(x) = \exists y, z : \ \mathtt{R}(x, y) \wedge \mathtt{R}(y, z) \wedge \mathtt{R}(z, x) \wedge x \neq y \wedge y \neq z \wedge z \neq x.$$

The two predicates coincide on loop-free graphs, but differ when self-loops are present. As illustrated in Figure 4, the loop-free task and the self-loop task differ in whether degenerate directed walks are possible. In Task 1, positives are nodes participating in a directed 3-cycle, and the absence of self-loops makes the distinctness constraint implicit. In Task 2, self-loops introduce spurious matches for the naive three-edge walk, so the agent must distinguish proper directed cycles from degenerate self-loop walks.

Note that both of these predicates $P(x), Q(x)$ are not expressible by standard message-passing graph neural networks, since it is not definable in graded modal logic.

Task 1: directed triangle detection on loop-free graphs. In the first task, all self-loops are removed from `R` (by setting $p_{\text{sl}} = 0.0$), and the binary label for node x is given by $Q(x)$. Since no edge of the form $\mathtt{R}(u, u)$ exists, any satisfying assignment to $P(x)$ must already use three distinct nodes. Therefore, on this task,

$$P(x) \equiv Q(x).$$

Consequently, the agent only needs to discover the directed 3-cycle pattern: the raw three-way self-join for $P(x)$ is already an exact feature for the target. This task tests whether the agent can find the relevant relational motif from examples, but it does not require the distinctness constraints to be identified separately.

Task 2: directed triangle detection on graphs with self-loops. The second task keeps the same target label $Q(x)$, but adds self-loops to the graph. Each node u receives a self-edge $\mathtt{R}(u, u)$ independently with probability p_{sl}. In this setting, $P(x)$ and $Q(x)$ are no longer equivalent. In particular, when $p_{\text{sl}} = 1.0$, every node satisfies $P(x)$ through the degenerate walk

$$x \to x \to x \to x,$$

even if it does not participate in any proper directed 3-cycle. Thus, the naive three-way self-join over-predicts positives and thus loses discriminative power. Consequently, solving Task 2 requires the agent to discover the proper-cycle condition $Q(x)$, either by explicitly enforcing the inequalities $x \neq y$, $y \neq z$, and $z \neq x$, or equivalently by filtering self-loop edges before performing the directed 3-cycle join, e.g., `WHERE src != dst`.

Table 10: **Directed triangle detection on synthetic graphs** (p_{sl}: self-loop probability). Each row is one independent agent run. "Discovery trial" is the trial at which the winning query was first proposed; "—" indicates the agent never reached AUROC 1.0.

Setup	p_{sl}	Discovery trial	Agent strategy
No self-loops	0.0	1	Raw `cycle3_cnt`; no inequality constraint required
Partial self-loops	0.4	1	`WHERE src` $\neq$ `dst` written immediately
Partial self-loops	0.4	—	Proxy features only; inequality constraint never written
Hard self-loops	1.0	14	Filter + directed join combined at trial 14
Hard self-loops	1.0	—	Raw `cycle3_cnt + cycle2_cnt`; no inequality constraint
Hard self-loops	1.0	—	`cycle3_out + recip_out`; no inequality constraint

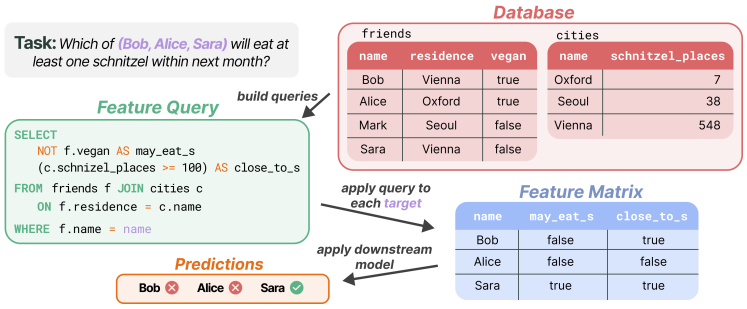


Figure 5: Illustrative example of SQL-based feature construction over relational data.

Results. Table 10 summarises representative run per setup. On Task 1 (no self-loop), the agent reliably solves the task on the first trial, since the standard 3-way self-join is sufficient and LightGBM learns the threshold automatically. Task 2 (with self-loop) separates the two predicates. With partial self-loops ($p_{\text{sl}} = 0.4$), the agent sometimes discovers the `src != dst` filter immediately, recovering the proper-cycle signal. In other runs, it relies on proxy features that correlate with $Q(x)$ but do not exactly encode the target. With self-loops on every node ($p_{\text{sl}} = 1.0$), the distinction becomes necessary: the unfiltered predicate $P(x)$ fires for all nodes through degenerate self-loop walks. In one of three independent runs, the agent reaches AUROC 1.0 by first discovering self-loop filtering in an imperfect feature and later combining it with the directed 3-cycle join at trial 14. The winning query is:

```
WITH edges AS (SELECT src, dst FROM R WHERE src != dst),
cycle3 AS (
  SELECT e1.src AS node_id, COUNT(*) AS cycle3_cnt
  FROM edges e1
  JOIN edges e2 ON e2.src = e1.dst
  JOIN edges e3 ON e3.src = e2.dst AND e3.dst = e1.src
  GROUP BY e1.src
)
SELECT e.node_id, COALESCE(c.cycle3_cnt, 0) AS cycle3_cnt
FROM eval_table e LEFT JOIN cycle3 c USING(node_id)
```

The two failing runs never wrote `src` $\neq$ `dst`; they relied on raw (inflated) cycle counts, which provide a strong but imperfect signal because every node with a self-loop accumulates at least one spurious 3-cycle path ($u \to u \to u \to u$), inflating its count regardless of true triangle membership. Task 1 shows that the agent can discover the directed 3-cycle pattern when P and Q coincide; Task 2 shows that self-loop variants make the distinction between P and Q observable, and therefore test whether the agent can recover the additional first-order distinctness constraint from validation feedback.

D Deep feature synthesis (DFS) vs. agentic feature construction

D.1 Expressivity

RELAGENT constructs a task-dependent feature map from a relational database to a feature matrix by synthesizing SQL feature queries with an LLM agent and applying a downstream predictor (see Figure 5 as an example). This invites a comparison to RDBLearn [9], which also adopts a two-stage pipeline by first summarizing relational data into a single table and then applying a tabular foundation model [8]. However, in contrast to RELAGENT, this mapping from relational database to a single table is not task-dependent but instead relies on a fixed, task-agnostic procedure, leading to a gap in expressivity.

Specifically, the deep feature synthesis (DFS) [4] used in Xu et al. [9] constructs features via predefined, column-wise aggregations. As first observed by Hudovernik et al. [16] such operations are inherently limited, as they cannot capture interactions *across rows*.

To illustrate this limitation, consider a database with a target table $\mathcal{T}_1$ and a child table $\mathcal{T}_2$ containing binary features A and B. For each entity in $\mathcal{T}_1$, define the target $y = 1$ if and only if A and B co-occur in at least one related row of $\mathcal{T}_2$. Any model operating on column-wise aggregations will fail to correctly distinguish these cases, since the joint occurrence information is not preserved.

In contrast, RELAGENT can directly express this target function via the following SQL query:

```
SELECT SUM(CASE WHEN A = 1 AND B = 1 THEN 1 ELSE 0 END) > 0 AS y
FROM T2
WHERE t1_id = entity
```

D.2 Interpretability

Each feature used by RELAGENT is defined by an explicit SQL query. Predictions can therefore be inspected through the selected queries and the downstream model's feature usage (see also the case study in Section 3). This yields a transparent pipeline in which both feature construction and prediction remain accessible and interpretable.

Importantly, through the evaluation workspace, RELAGENT can iteratively assess candidate features and retain only those that are most informative for the task. As a result, the final feature set is compact and focused on relevant signals, rather than relying on a large collection of generic aggregations.

In contrast, DFS-based approaches rely on predefined aggregation primitives applied in a breadth-first manner over the relational schema. While this provides broad coverage, it often leads to a large number of features. As shown in Figure 6, RELAGENT achieves competitive to superior performance while producing significantly fewer features (note the log-scale in Figure 6). Due to the expansion mechanism of DFS, the number of generated features grows with the degree of the target table and its related tables. In strongly connected relational databases, this can result in a substantial increase in feature columns (see the `rel-event` dataset in Figure 6). In contrast, the number of features created by RELAGENT stays constant over different tasks and datasets.

E Search budget and convergence

Setup. We study how RELAGENT scales with the number of allowed agent turns. For each task, we compute the best validation score achieved up to turn k, and normalize the improvement by the final best validation score reached within the full 60-turn budget. Thus, a value of 100% indicates that the best final validation score has already been recovered by turn k.

Results. As shown in Figure 7, most gains occur early in the search. On classification tasks, RELAGENT often recovers a large fraction of its final validation gain within the first 5–10 turns, especially on 4DBInfer and RelBenchV2 classification. Regression tasks converge more gradually, reflecting the larger role of objective choice, target transformations, and model configuration in addition to feature discovery. Across all benchmark groups, performance largely saturates before the full 60-turn budget, suggesting that the agent benefits from iterative feedback but does not require long unbounded search in most cases.

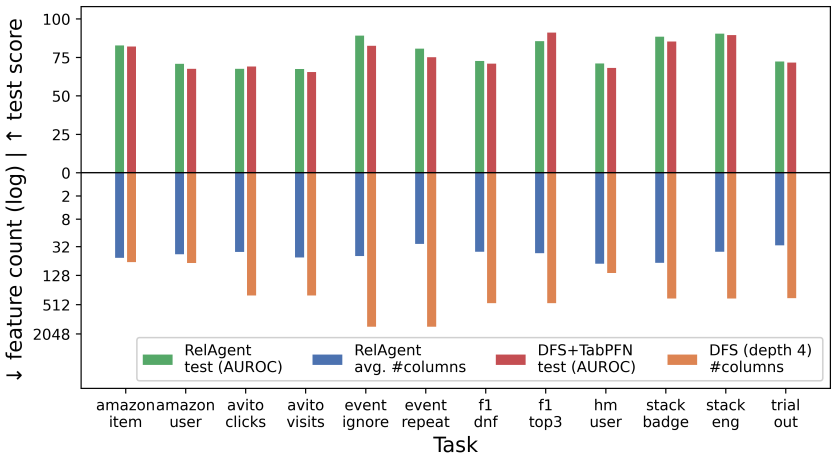


Figure 6: Test scores (AUROC) on RelBenchV1 classification tasks of RELAGENT and DFS+TabPFN [4, 9] (↑) as well as the **number of feature columns produced per task** by RELAGENT and DFS (depth 4) (↓).

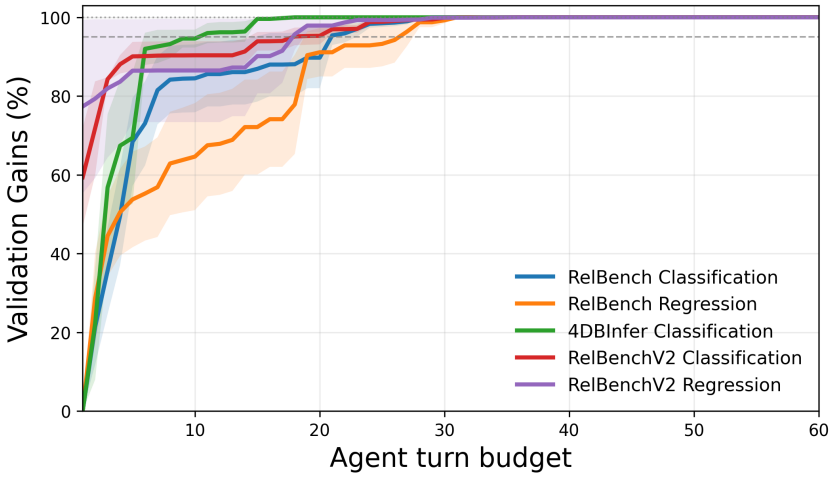


Figure 7: **Search-budget scaling.** The fraction of final validation gain recovered as a function of agent turn budget. Shaded regions indicate variation across tasks.

F Efficiency analysis

We analyze the wall-clock time and token consumption of our agent across three backbone LLMs: GPT-5.2, Qwen3-32B, and DeepSeek-V4-Pro. For each completed run, we parse the agent's conversation trace and count tokens. Wall-clock time is measured from the first to the last event in the trace. Thinking tokens, i.e., the content within `<think>` ... `</think>` delimiters, are reported separately because they consume inference compute but are never passed back to the agent as context.

Setup. Each run consists of up to 60 agent turns over a single RelBench task. At each turn, the model receives the full conversation history and produces a response that may include zero or more tool invocations (SQL queries, `validate_program`, etc.); all tool results are collected and appended before the next turn begins. Since a single turn can issue multiple tool calls, the total tool invocations per run can exceed the number of turns. All three models are queried through the same internal API gateway with temperature $= 1$ and a maximum of 8 000 output tokens per turn. GPT-5.2 and DeepSeek-V4-Pro are accessed via LiteLLM API calls through an internal API gateway. Qwen3-32B is served locally using vLLM on 2× NVIDIA RTX 6000 Ada (48 GB each, tensor-parallel). All models use temperature $= 1$ and a maximum of 8 000 output tokens per turn.

Table 11: **Efficiency of three backbone LLMs** per agent run (mean $\pm$ std) averaged over RelBench V1 classification and regression runs. Thinking tokens are `<think>` chain-of-thought content generated by Qwen3-32B, not re-injected as context.

	GPT-5.2	**Qwen3-32B**	**DeepSeek-V4-Pro**
Wall-clock time (min)	35.2 ± 27.8	97.5 ± 63.9	44.5 ± 49.1
Agent turns per run	41.1 ± 11.7	44.2 ± 15.2	40.7 ± 31.0
Tool invocations / run	47.2 ± 10.9	25.2 ± 6.7	31.0 ± 16.1
Input tokens	$22{,}270 \pm 7{,}356$	$6{,}343 \pm 2{,}509$	$8{,}404 \pm 3{,}600$
Output tokens	$20{,}671 \pm 5{,}884$	$31{,}873 \pm 11{,}118$	$26{,}744 \pm 35{,}034$
Thinking tokens	—	$36{,}782 \pm 10{,}870$	—
Total tokens	$42{,}941 \pm 11{,}911$	$74{,}998 \pm 22{,}267$	$35{,}148 \pm 38{,}634$
Trace-token cost / run (USD)	$\approx \mathbf{\$1.57}$	$\mathbf{\$0}$	$\approx \mathbf{\$0.11}$

Table 12: **Effect of the search-turn budget.** Average test AUROC on RelBenchV1 classification tasks. Candidate selection is based on validation performance.

Turns	10	20	30	60
Test AUROC	77.81	77.48	**78.87**	78.38

Time efficiency, token efficiency, and cost. GPT-5.2 and DeepSeek-V4-Pro complete a run in 35 and 45 minutes on average, respectively. Qwen3-32B is $2.8\times$ slower (98 min) due to the latency of its extended chain-of-thought generation over two locally-hosted RTX 6000 GPUs.

Input tokens are counted from the conversation trace using the `cl100k_base` tokenizer (system prompt counted once plus all tool results). GPT-5.2 has substantially more input tokens ($\approx$22K) than Qwen3-32B and DeepSeek-V4-Pro ($\approx$6–8K) as it issues roughly twice as many tool calls per run, accumulating more SQL query results in its input stream.

In terms of total tokens, Qwen3-32B generates $1.7\times$ more tokens than GPT-5.2; 49% of Qwen3-32B's tokens are hidden thinking tokens never re-injected as context, so the visible output (32K) is only $1.5\times$ larger than GPT-5.2's (21K).

In terms of cost, DeepSeek-V4-Pro costs [58] only \$0.11 vs. \$1.57 for GPT-5.2 [59], which is $14\times$ cheaper thanks to DeepSeek's lower per-token price (\$1.74/\$3.48 vs. \$15/\$60 per 1M input/output tokens). Qwen3-32B costs \$0 as a locally-hosted model. Costs are estimated from the final conversation trace under an ideal 100% prompt-cache hit rate: each message is counted once rather than re-counting the full prompt at every API turn.

Diminishing returns. RELAGENT achieves near-full-budget performance within ten search turns, recovering approximately 80% and 90% of the final validation improvement within 10 and 20 turns, respectively. We further examine the effect of independent rollouts through best-of-R selection, where the best candidate among the first R rollouts is selected by validation performance. Increasing the number of rollouts from one to five yields incremental test AUROC improvements of $(0.37, 0.67, 0.00, 0.24)$ on RelBenchV1, $(0.17, 0.00, 0.00, 0.01)$ on RelBenchV2, and $(0.56, 0.03, 1.77, 0.48)$ on 4DBInfer. While RelBenchV1 and RelBenchV2 exhibit diminishing marginal gains, 4DBInfer benefits substantially from the fourth rollout. Overall, these results suggest that moderate search budgets already capture much of the available performance, although additional exploration can occasionally discover substantially better programs.

G Experimental details

G.1 Search setup

For each task, we run five independent searches with up to 60 agent turns each. The agent uses GPT-5.2 at temperature 1 and up to $8{,}000$ output tokens per turn, with a 900 second time-out per turn. Among the five searches, we select the one with the highest best-trial validation score as the final

Table 13: **Best-of-R rollout analysis.** Incremental change in average test AUROC when increasing the number of independent rollouts from $R-1$ to R. Candidates are selected by validation performance.

Benchmark	$+R_2$	$+R_3$	$+R_4$	$+R_5$
RelBenchV1	+0.37	+0.67	+0.00	+0.24
RelBenchV2	+0.17	+0.00	+0.00	+0.01
4DBInfer	+0.56	+0.03	+1.77	+0.48

Table 14: Database summary and statistics across RelBenchV1, RelBenchV2, and 4DBInfer.

Dataset	Domain	Tasks	#Tasks	#Tables	#Rows	#Cols
RelBenchV1						
f1	Sports	Race outcome	3	9	74,063	67
avito	E-commerce	User engagement, CTR	3	8	20,679,117	42
event	Social	Event participation	3	5	41,328,337	128
trial	Medical	Clinical trial outcome	3	15	5,434,924	140
amazon	E-commerce	Churn, LTV	4	3	15,000,713	15
stack	Social	Community activities	3	7	4,247,264	52
hm	E-commerce	Churn, LTV	2	3	16,664,809	37
RelBenchV2						
ratebeer	E-commerce	Engagement, Churn	4	13	13,787,005	221
arxiv	Academic Citations	Publications	2	6	2,146,112	21
4DBInfer						
AB	E-commerce	Churn	1	3	24,291,489	15
OB	Social	CTR	1	8	2,170,441,217	31
RR	E-commerce	CVR	1	3	23,033,676	11
SE	Social	Churn, Popularity	2	7	5,399,818	49

predictor. We refer to this process as cross-rollout model selection. The agent selects among seven tree-based model families: LightGBM with standard boosting (`gbdt`), DART (`dart`), and GOSS (`goss`) [6]; XGBoost standard (`xgboost`) and XGBoost+DART (`xgb_dart`) [5, 54]; Random Forest (`rf`) via LightGBM; and CatBoost (`catboost`) [55]. For each model choice the agent also tunes key hyperparameters: number of estimators n_T, learning rate η, and tree depth d. For regression, the agent additionally selects the training objective ($L1$/MAE or $L2$/MSE) and whether to log-transform the target. All SQL queries are executed via DuckDB [52]. Prompts are reported in Appendix H.

G.2 Selected programs

Tables 16 and 17 report, for each task, the number of SQL feature queries in the selected program, the chosen downstream model, and the key hyperparameters of the best-trial configuration. Among

Table 15: **Full Ablation results on the entity classification tasks in RelBenchV1.** Higher is better (AUROC).

	Model Variants	f1		avito		event		trial	amazon		stack		hm	**Avg**
		dnf	top	click	visit	repeat	ignore	out	user	item	eng	badge	churn	↑
Architecture	RELAGENT	78.34	85.23	68.36	67.79	78.20	87.25	71.86	70.78	82.84	90.41	88.42	71.07	78.38
	w/o feedback	58.43	70.31	53.22	50.27	72.06	87.16	66.24	66.48	81.80	65.96	60.06	70.37	66.86
	w/o eval workspace	64.94	79.83	68.76	70.79	75.98	84.55	65.23	70.21	82.24	88.86	87.67	70.37	75.78
	w/o semantic	65.44	77.70	67.09	67.42	73.39	81.23	62.20	70.23	82.32	89.45	85.70	71.07	74.44
Predictor	rule-only	62.64	83.45	61.15	64.38	66.79	82.72	55.55	61.06	75.30	88.91	77.13	59.90	69.92
	catboost-only	67.64	80.27	66.74	67.57	73.38	81.16	72.98	70.46	82.71	90.18	88.04	70.35	75.95
	lightgbm-only	73.30	81.22	66.50	61.47	78.53	84.21	65.29	67.54	82.28	90.23	89.52	71.21	75.94
LLM	with Deepseek-V4	68.97	86.81	67.77	67.67	77.64	82.50	74.10	69.96	82.06	90.43	87.67	70.97	76.00
	with Qwen3-32B	57.67	58.04	65.10	64.62	74.73	75.91	71.36	69.22	79.92	89.11	65.30	70.62	70.13

all the model families, CatBoost is the most frequently selected model for classification tasks (14 of 21 tasks), followed by XGBoost-based variants. The number of SQL feature queries per task ranges from 3 to 19, with a median of 7. For regression, the agent consistently selects L1/MAE objectives across all tasks, and applies a log-transformation of the target in 9 of 11 regression tasks, consistent with the skewed distributions common in LTV and activity-count targets.

Table 16: Selected programs for classification tasks. n_T: number of estimators; η: learning rate; d: tree depth; **Val.**: best validation AUROC (%) of the selected run; **Test**: test AUROC (%) of the selected run; **Std**: standard deviation of test AUROC across 5 runs.

Benchmark	Task	#SQL	Model	n_T	η	d	Val.	Test	Std
RelBenchV1	f1/driver-dnf	7	gbdt	500	0.030	5	66.44	78.34	3.72
	f1/driver-top3	10	catboost	500	0.030	6	74.74	85.23	6.89
	avito/user-clicks	9	catboost	500	0.050	8	67.11	68.36	1.69
	avito/user-visits	11	catboost	500	0.032	5	71.29	67.79	0.12
	event/user-repeat	7	xgboost	500	0.050	5	78.21	78.20	2.94
	event/user-ignore	5	catboost	400	0.050	6	76.66	87.25	1.30
	trial/study-outcome	3	catboost	500	0.012	5	66.19	71.86	2.18
	amazon/user-churn	11	catboost	500	0.060	8	70.65	70.78	0.51
	amazon/item-churn	9	catboost	500	0.060	9	82.39	82.84	0.23
	stack/user-engagement	7	xgb_dart	500	0.050	6	90.33	90.41	0.19
	stack/user-badge	13	xgboost	500	0.050	7	89.65	88.42	1.27
	hm/user-churn	19	catboost	500	0.050	9	71.50	71.07	0.27
RelBenchV2	ratebeer/beer-churn	5	catboost	500	0.060	8	90.27	84.70	0.93
	ratebeer/user-churn	3	xgb_dart	500	0.050	9	98.90	98.63	0.83
	ratebeer/brewer-dormant	15	catboost	500	0.050	8	84.80	83.33	0.33
	arxiv/paper-citation	8	catboost	500	0.050	6	82.47	82.62	0.14
4DBInfer	amazon/user-churn	13	catboost	500	0.050	8	79.90	79.44	0.25
	outbrain/ad-ctr	4	catboost	500	0.040	4	63.89	62.63	4.42
	retailrocket/item-cvr	7	catboost	500	0.060	8	88.65	86.35	0.73
	stackexchange/post-upvote	13	xgboost	500	0.050	7	96.24	89.45	3.31
	stackexchange/user-churn	12	dart	500	0.050	7	87.16	89.03	0.21

Table 17: Selected programs for regression tasks. All selected programs use an L1 (MAE) objective. The "Log" column indicates whether the agent applied a log-transformation to the regression target; MAE is reported in the original space after inverting the transformation. **Val.**: best validation MAE (%) of the selected run; **Test**: test MAE of the selected run; **Std**: standard deviation of test MAE across 5 runs.

Benchmark	Task	#SQL	Model	n_T	η	d	Obj.	Log	Val.	Test	Std
RelBenchV1	f1/driver-position	10	goss	500	0.050	6	L1	—	3.598	4.019	0.153
	avito/ad-ctr	5	gbdt	500	0.050	7	L1	✓	0.030	0.033	0.000
	event/user-attendance	7	dart	500	0.040	4	L1	✓	0.238	0.241	0.009
	trial/study-adverse	7	xgboost	500	0.070	7	L1	✓	37.439	37.194	3.553
	trial/site-success	5	xgboost	500	0.050	6	L1	—	0.400	0.386	0.018
	amazon/user-ltv	7	xgboost	500	0.030	7	L1	✓	11.771	13.949	0.060
	amazon/item-ltv	5	xgboost	500	0.050	8	L1	✓	36.370	41.765	0.332
	stack/post-votes	7	xgboost	500	0.040	8	L1	✓	0.058	0.064	0.001
	hm/item-sales	7	xgb_dart	500	0.050	8	L1	✓	0.043	0.035	0.001
RelBenchV2	ratebeer/user-count	6	xgboost	500	0.040	8	L1	✓	4.367	6.021	0.107
	arxiv/author-publication	7	goss	500	0.025	10	L1	✓	0.420	0.462	0.016

Variance among single searches. Tables 16 and 17 also report the standard deviation of test scores across the 5 searches that were performed for the cross-rollout model selection. On most tasks, the standard deviation is below 2 AUROC points (classification) or small relative to the score magnitude (regression), indicating that individual runs are broadly representative of overall performance. Tasks with higher variance (e.g., `f1/driver-top3` σ=6.89, `trial/study-outcome` σ=2.18) tend to be smaller datasets where randomness in program search has a larger effect.

Table 18: Mean and standard deviations of test AUROC (%) among 3 cross-rollout model selections with each 5 searches for entity classification tasks in RelBenchV2.

Task	arxiv/citation	ratebeer/beer	ratebeer/dormant	ratebeer/user
Mean	82.59	84.55	83.01	98.54
Std.	0.031	0.135	0.239	0.072

Variance among cross-rollout model selections. For entity classification tasks in RelBenchV2, we additionally report the mean and standard deviation of test AUROC among 3 independent cross-rollout model selections, each consisting of 5 searches (Table 18). Due to resource constraints, we omit other tasks and datasets. Across all four tasks, the standard deviation remains below 0.25 AUROC points, indicating that this best-of-K selection procedure yields stable final performance with low sensitivity to the randomness of individual rollouts.

Validation-test distribution shift. On some small datasets (e.g., `f1/driver-top3` and `trial/study-outcome`), **Val.** exceeds **Test**, suggesting mild overfitting to the validation split; however, these gaps are small and primarily occur in low-sample regimes. Most datasets show closely matched validation and test performance, indicating that validation performance is generally a reliable criterion for model selection rather than a source of systematic overfitting.

Relational invariance. Relational invariance, i.e., invariance to permutations of row identifiers and table orderings, is induced by the selected SQL feature queries: if no query depends on row identifiers, the resulting predictions are invariant as well. In our experiments, queries that directly access row identifiers were not explicitly forbidden. We therefore inspected all final selected feature programs across all tasks and found that none accesses a row identifier of any table. We did, however, find instances in which a query conditions on a categorical identifier, e.g., `sales_channel_id` in `hm/user-churn`. This does not violate relational invariance: such identifiers are fixed categorical encodings with semantic meaning (e.g., online vs. offline), not row identifiers, and are thus invariant to row permutations.

G.3 SQL query analysis

To better understand the programs discovered by RELAGENT, we analyze the selected program for each task (Tables 16–17). Across 32 selected programs, the agent produces 264 feature queries, with a median of 7 queries per program and a range of 3–19. Note that no two selected queries are identical, suggesting that the agent synthesizes task-specific SQL programs rather than reusing fixed templates. Table 19 summarizes the structural properties of these queries.

Join depth and structure. Half of all queries join exactly one additional table; 7% are direct scans with no join; 44% join two or more tables (maximum 9). Common Table Expressions (CTEs, written as `WITH name AS (...)`) appear in 70% of queries, reflecting a preference for staged, named intermediate results over deeply nested subqueries.

Temporal structure. Every query references `eval_table`, which exposes the target row and its prediction timestamp. This enforces row-wise alignment by construction. For timestamped event tables, temporal non-leakage is achieved through explicit predicates comparing event timestamps against the corresponding `eval_table` cutoff; 73% of queries contain interval-based temporal filters, while the remaining queries are mostly static/profile features or non-windowed aggregates. Surprisingly, the agent independently arrives at the same canonical aggregation horizons, e.g., 30, 90, 365, 180, 7, 14, and 60 days, that human analysts use for churn, LTV, and activity modelling, with no instruction to do so.

Aggregate repertoire. `COUNT` is the most common aggregate (62%), followed by conditional logic via `CASE`/`IF` expressions (59%), `SUM` (39%), and `MAX`/`AVG` (31%/30%). Division-based ratio features (e.g., recent activity rate over lifetime count) appear in 9% of queries.

Table 19: Structural properties of the 264 feature queries across all 32 selected programs (one per task). All percentage counts refer to the fraction of queries (out of 264) that exhibit each property.

Property	**# queries (%)**
Query structure	
Queries per program (median / range)	7 / 3–19
Joins per query (mean / median)	1.84 / 1
0 joins (direct scan)	18 (7%)
1 join	131 (50%)
2 joins	58 (22%)
3 joins	24 (9%)
4+ joins	33 (13%)
Using CTEs (`WITH` clauses)	186 (70%)
Temporal patterns	
Contains interval-based temporal filter	193 (73%)
Contains recency feature (days since last event)	94 (36%)
Contains 7-day window	40 (15%)
Contains 30-day window	78 (30%)
Contains 90-day window	70 (27%)
Contains 365-day window	57 (22%)
Contains 180-day window	40 (15%)
Aggregate functions	
`COUNT`	163 (62%)
`CASE/IF` conditional	155 (59%)
`SUM`	104 (39%)
`MAX`	83 (31%)
`AVG`	80 (30%)
Division ratio features	25 (9%)

H Prompts

We list the prompts sent to the agent at each stage of a run. Curly-brace tokens (`{entity_col}`, `{target_col}`, etc.) are filled in per task at runtime. The system prompt is sent once at the start of each run. Follow-up prompts are injected after each agent turn until the turn budget is exhausted, at which point the wrap-up prompt is sent instead.

H.1 Tool descriptions

The agent has access to the following tools. Descriptions are passed to the LLM as part of the function-calling schema.

`execute_query`

```
Execute a SQL query and return results.

Args:
    query (str): The SQL query to execute.

Returns:
    For SELECT queries: list of dicts (column -> value per row).
    For non-SELECT operations: status dict with 'status' and 'message'.
    For errors: error message string starting with "Error:".

Note: SELECT queries without a LIMIT clause are automatically
capped at 200 rows.
```

`get_table_info`

```
Get comprehensive information about table(s) in the database.

Returns schema, primary keys, foreign keys, and row counts.
If table_name is provided, returns info for that table;
```

```
otherwise returns info for all tables.

Args:
    table_name (str, optional): Name of a specific table.
        If None, returns info for all tables.
```

validate_program

```
Test a predictive program using a wrapped model on a data split.

Args:
    feature_queries_json: JSON string of feature queries.
        Format: [{"name": "query_name", "sql": "SELECT ..."}]
        Each query must be anchored on eval_table, return row_id,
        and produce feature columns for each target row. eval_table
        contains the split rows currently being scored, including
        row_id, {entity_col}, and {timestamp_col}. For timestamped
        source tables, feature queries must filter source records to
        occur before eval_table.{timestamp_col}.
        `train_table` is available and contains labeled training rows.
    model_choice: One of 7 learners: "gbdt", "rf", "dart", "goss",
        "xgboost", "xgb_dart", "catboost".
        - gbdt:     LightGBM standard gradient boosting
        - rf:       Random Forest via LightGBM
        - dart:     LightGBM with DART dropout regularization
        - goss:     LightGBM with gradient-based subsampling
        - xgboost:  XGBoost (second-order gradients)
        - xgb_dart: XGBoost with DART dropout
        - catboost: CatBoost (ordered boosting)
    model_config_json: JSON dict of hyperparameters (optional).
        Common keys: n_estimators, learning_rate, max_depth,
                     subsample, colsample_bytree.
        LightGBM variants also accept: min_child_samples,
                     lambda_l1, lambda_l2.
        XGBoost variants also accept: min_child_weight,
                     reg_alpha, reg_lambda.
        CatBoost also accepts: l2_leaf_reg.
        Out-of-bounds values are clamped. Unknown keys are ignored.

Returns:
    Formatted string with metrics, diagnostics, and trial history.
```

get_trial_history

```
Get a summary of all previous validation trials.

Returns:
    Formatted string showing trial history with scores and
    approach summaries.
```

query_eval_workspace

```
Query the evaluation workspace to analyse trial results.

The workspace is a DuckDB database with the following tables:

  trials
    trial_id TEXT, trial_name TEXT, parent_trial_id TEXT,
    created_at TIMESTAMPTZ, split TEXT, model_choice TEXT,
    resolved_model_config TEXT, feature_query_hash TEXT,
    feature_block_names TEXT, primary_metric TEXT,
    primary_score DOUBLE, metrics_json TEXT, notes TEXT


  eval_predictions
    trial_id TEXT, row_id INTEGER, entity_id TEXT, label TEXT,
```

```
    score DOUBLE, predicted_class TEXT, split TEXT,
    eval_cutoff TIMESTAMPTZ

row_id is stable across trials for the same split, so you can
join two trials on row_id to compare predictions on the same examples.

IMPORTANT: These tables are analysis artifacts only. Do NOT use
them as feature sources in your SQL feature queries.

Args:
    sql: A SELECT query to run against the workspace.

Returns:
    Query results (up to 500 rows) or an error message.
```

H.2 System prompt

System Prompt: Classification

```
You are a data scientist building a predictive pipeline for a
ENTITY CLASSIFICATION task.

GOAL: Find a set of SQL feature queries + a model choice that
accurately predict the class label ({target_col}) for ALL entities
in the validation set.

You have the following tools:
1. SQL tools (execute_query, get_table_info, etc.)
   -- explore and query the database
2. validate_program(feature_queries_json, model_choice, model_config_json)
   -- train and evaluate your feature pipeline on the validation split
3. get_trial_history()
   -- see what you've already tried and their scores
4. query_eval_workspace(sql)
   -- analyze the evaluation workspace (trials, eval_predictions)

Rules:
- Use SQL tools to explore the database. Do NOT guess table/column names.
- Start by running SHOW TABLES and PRAGMA table_info('table') to
  understand the schema.
- train_table contains labeled training examples. Use it to learn patterns.
- eval_table contains the current rows being scored (val/test input keys).
- Always anchor feature SQL on eval_table (not train_table) -- eval_table
  is swapped with the correct split at each call.
- Write SQL queries that extract features PER TARGET ROW.
- eval_table contains one row per target example, including row_id,
  {entity_col}, and {timestamp_col}.
- Each feature query must be anchored on eval_table and return row_id.
- For timestamped source tables, include temporal filters of the form
  source_time < eval_table.{timestamp_col}.
- Do not group only by {entity_col}: the same entity may appear at
  multiple prediction timestamps. Group by row_id, and merge features
  back by row_id.
- Call validate_program() with your SQL feature blocks, model_choice,
  and optional model_config_json to run on the validation set.
- Iterate based on the metrics, diagnostics, and error examples returned.

- For BINARY classification, the model outputs probability scores (0-1).
- Think about features like: entity frequency, recency, aggregates from
  related tables.
- The validation tool returns metrics (AUROC, F1, etc.)
  + diagnostics.

=== WRAPPED MODEL MODE ===

Your validate_program() tool accepts:
```

```
  - feature_queries_json: SQL feature queries (same as before)
  - model_choice: one of the 7 learners below
  - model_config_json: optional JSON dict of hyperparameters

Available models:
  1. "gbdt"     -- Standard Gradient Boosted Trees. Fast, strong default.
                   Config: n_estimators (50-500), learning_rate (0.01-0.3),
                           max_depth (2-10), min_child_samples (1-100),
                           subsample (0.5-1.0), colsample_bytree (0.5-1.0)
                   Regularization: lambda_l1 (0.0-10.0), lambda_l2 (0.0-10.0)
  2. "rf"       -- Random Forest (bagging; less sensitive to learning rate).
                   Config: same keys as gbdt
  3. "dart"     -- DART Boosting (dropout regularization).
                   Config: same keys as gbdt
  4. "goss"     -- GOSS (gradient-based subsampling; fast on large datasets).
                   Config: same keys as gbdt
  5. "xgboost"  -- XGBoost (second-order gradients; different regularization).
                   Config: n_estimators (50-500), learning_rate (0.01-0.3),
                           max_depth (2-10), min_child_weight (1-100),
                           subsample (0.5-1.0), colsample_bytree (0.5-1.0)
                   Regularization: reg_alpha (0.0-10.0), reg_lambda (0.0-10.0)
  6. "xgb_dart" -- XGBoost + DART dropout.
                   Config: same keys as xgboost
  7. "catboost" -- CatBoost (ordered boosting; robust on heterogeneous features).
                   Config: n_estimators (50-500), learning_rate (0.01-0.3),
                           max_depth (2-10), l2_leaf_reg (0.1-10.0)

Categorical features (all 7 learners):
  Add "categorical_features" inside model_config_json as a list of
  "<query_name>__<col>" names to treat columns natively as categorical.
  High-cardinality columns (>~4000 levels) should be bucketed in SQL first.

Constraints:
  - Do NOT write free-form training code, custom objectives, or ensembles.
  - Do NOT perform hyperparameter search loops -- one config per call.
  - Out-of-bounds config values are clamped automatically.
  - Omitted config fields use strong defaults.
  - The environment handles train/val splitting, fitting, and evaluation.

The tool returns: metrics, resolved model config, row counts after merges,
missingness rates, warnings, and best/worst prediction examples.
Use this feedback to iterate on your SQL features.

=== EVALUATION WORKSPACE ===
After each validate_program() call, the full evaluation output is
persisted to a queryable workspace.
Use query_eval_workspace(sql) to analyse results.

Workspace tables:

  trials
    trial_id TEXT, trial_name TEXT, parent_trial_id TEXT,
    created_at TIMESTAMPTZ, split TEXT, model_choice TEXT,
    resolved_model_config TEXT, feature_query_hash TEXT,
    feature_block_names TEXT, primary_metric TEXT,
    primary_score DOUBLE, metrics_json TEXT, notes TEXT

  eval_predictions
    trial_id TEXT, row_id INTEGER, entity_id TEXT, label TEXT,
    score DOUBLE, predicted_class TEXT, split TEXT,
    eval_cutoff TIMESTAMPTZ

row_id is positionally stable across trials for the same split, so
you can join two trials on row_id to compare predictions on the same
examples.

IMPORTANT: Always run PRAGMA table_info('tablename') to verify exact
```

```
column names. Column names differ across datasets -- never assume a
column like 'id', 'type', 'date', or 'count' exists without checking.
```

The regression system prompt is identical to the classification one above, except for the goal statement, task-type bullet points, and the model menu (each learner gains a regression objective key:

- `regression_l1`/`regression_l2`/`huber` for LightGBM variants,
- `reg:absoluteerror`/`reg:squarederror` for XGBoost variants).

```
You are a data scientist building a predictive pipeline for a
ENTITY REGRESSION task.

GOAL: Find a set of SQL feature queries + a model choice that
accurately predict the numerical value ({target_col}) for ALL
entities in the validation set.

[... tool inventory and Rules identical to classification ...]

- The model outputs a numeric value for each entity.
- The validation tool returns metrics (MAE)
  + diagnostics.
- Think about: historical averages, trends, aggregates from related
  tables.
- Primary objective for this run is MAE: minimize absolute error.


=== WRAPPED MODEL MODE ===

Your validate_program() tool accepts:
  - feature_queries_json: SQL feature queries (same as before)
  - model_choice: one of the 7 learners below
  - model_config_json: optional JSON dict of hyperparameters

Available models:
  1. "gbdt"     -- Standard Gradient Boosted Trees. Fast, strong default.
                   Config: n_estimators (50-500), learning_rate (0.01-0.3),
                           max_depth (2-10), min_child_samples (1-100),
                           subsample (0.5-1.0), colsample_bytree (0.5-1.0)
                   objective: "regression_l1" (MAE), "regression_l2" (MSE),
                              "huber"
                   Default: "regression_l1" -- directly minimises eval metric.
  2. "rf"       -- Random Forest (bagging; less sensitive to learning rate).
                   Config: same keys as gbdt (including objective)
  3. "dart"     -- DART Boosting (dropout regularization).
                   Config: same keys as gbdt (including objective)
  4. "goss"     -- GOSS (gradient-based subsampling; fast on large datasets).
                   Config: same keys as gbdt (including objective)
  5. "xgboost"  -- XGBoost (second-order gradients).
                   Config: n_estimators (50-500), learning_rate (0.01-0.3),
                           max_depth (2-10), min_child_weight (1-100),
                           subsample (0.5-1.0), colsample_bytree (0.5-1.0)
                   objective: "reg:absoluteerror" (MAE),
                              "reg:squarederror" (MSE),
                              "reg:pseudohubererror" (Huber)
  6. "xgb_dart" -- XGBoost + DART dropout.
                   Config: same keys as xgboost (including objective)
  7. "catboost" -- CatBoost (ordered boosting).
                   Config: n_estimators (50-500), learning_rate (0.01-0.3),
                           max_depth (2-10), l2_leaf_reg (0.1-10.0)

  For skewed targets: add "log_transform_target": true to
  model_config_json. The harness fits on log1p(y) and reports MAE
  back in the original scale.

Categorical features (all 7 learners):
  Add "categorical_features" inside model_config_json as a list of
```

```
  "<query_name>__<col>" names to treat columns natively as categorical.

Constraints:
  - Do NOT write free-form training code, custom objectives, or ensembles.
  - Do NOT perform hyperparameter search loops -- one config per call.
  - Out-of-bounds config values are clamped automatically.
  - The environment handles train/val splitting, fitting, and evaluation.

[... feedback description and Evaluation Workspace identical to classification ...]
```

H.3 Execution Prompts

Execute Prompt (First User Message)

```
Task: {task_type}
Task Description: {task_description}
Dataset: {dataset_name}

Entity column: {entity_col}
Timestamp column: {timestamp_col}
Target column: {target_col}
Validation set size: {n_val} target rows

WRAPPED MODEL MODE: You propose SQL features + a model choice.
The environment trains and evaluates.

Steps:
1. Run SHOW TABLES to see available tables
2. Run PRAGMA table_info('train_table') and inspect other tables
3. Explore data distributions (SELECT COUNT(*), sample rows, etc.)
4. Design SQL feature queries that extract per-target-row signals
   anchored on eval_table and keyed by row_id
5. Call validate_program() with your features, model_choice, and
   optional config

Example call:
  validate_program(
    feature_queries_json='[{"name": "basic_stats",
                            "sql": "SELECT ..."}]',
    model_choice="gbdt",
    model_config_json='{}'
  )

Start with gbdt and simple features. Iterate by improving SQL
features based on diagnostics.

You have query_eval_workspace(sql): after each validate_program(),
analyze trials and eval_predictions as described in the system prompt.
```

Follow-up Prompt (Intermediate Turns)

```
Continue improving your pipeline.
- If you haven't validated yet, call validate_program() now.
- Full evaluation results are available in the workspace after each trial.
  Use query_eval_workspace() for error analysis.
- Call get_trial_history() to see all past attempts.
- Use what you find in the workspace to guide your next SQL feature
  improvements.
```

Wrapup Prompt (Final Turn)

```
This is your last turn. If you haven't submitted a validation yet, do so now.
Call validate_program() with your best SQL queries, model choice, and model configuration.
If you already have results, call get_trial_history() to confirm your best score.
```

Table 20: Task descriptions of RelBench V1 (12 classification, 9 regression).

Dataset	Task	Type	Description
rel-amazon	user-churn	Cls	Predict 1 if the customer does not review any product in the next 3 months, and 0 otherwise.
	item-churn	Cls	Predict 1 if the product does not receive any reviews in the next 3 months.
	user-ltv	Reg	Predict the $ value of the total products a user buys and reviews in the next 3 months.
	item-ltv	Reg	Predict the $ value of the total purchases and reviews a product receives in the next 3 months.
rel-stack	user-engagement	Cls	Predict if a user will make any votes, posts, or comments in the next 3 months.
	user-badge	Cls	Predict if a user will receive a new badge in the next 3 months.
	post-votes	Reg	Predict how many votes a user post will receive in the next 3 months.
rel-trial	study-outcome	Cls	Predict if the trial will achieve its primary outcome ($p < 0.05$).
	study-adverse	Reg	Predict the number of patients with severe adverse events or death.
	site-success	Reg	Predict the success rate of a trial site in the next 1 year.
rel-f1	driver-dnf	Cls	Predict if a driver will DNF in a race in the next 1 month.
	driver-top3	Cls	Predict if a driver will qualify in the top 3 for a race in the next 1 month.
	driver-position	Reg	Predict the average finishing position of a driver across all races in the next 2 months.
rel-hm	user-churn	Cls	Predict whether a customer will have no transactions in the next week.
	item-sales	Reg	Predict the total sales (sum of prices) for an article in the next week.
rel-event	user-repeat	Cls	Predict if a user will attend an event in the next 7 days (given attendance in the last 14 days).
	user-ignore	Cls	Predict if a user will ignore more than 2 event invitations in the next 7 days.
	user-attendance	Reg	Predict how many events a user will respond yes/maybe to in the next 7 days.
rel-avito	user-visits	Cls	Predict whether a user will visit more than one ad in the next 4 days.
	user-clicks	Cls	Predict whether a user will click on more than one ad in the next 4 days.
	ad-ctr	Reg	Given that an ad will be clicked in the next 4 days, predict its click-through rate.

H.4 Task Descriptions

Tables 20–22 list the `{task_description}` string injected into the execute prompt for each task.

I Broader impacts

RELAGENT may reduce the manual effort required to build relational prediction systems and improve explainability by producing explicit SQL feature programs. However, relational databases may contain sensitive attributes or proxy variables, and an autonomous feature-search system may discover features that encode demographic, behavioral, or otherwise sensitive signals. The method should therefore be used with appropriate data governance, privacy controls, and fairness auditing, especially in high-stakes domains such as healthcare, finance, hiring, or public services.

Table 21: Task descriptions of RelBench V2 (4 classification, 2 regression).

Dataset	Task	Type	Description
rel-arxiv	paper-citation	Cls	Predict whether a paper will receive at least one citation in the next 6 months.
	author-publication	Reg	Predict how many papers an author will publish in the next 6 months.
rel-ratebeer	beer-churn	Cls	Predict whether a beer will receive no rating in the next 90 days.
	user-churn	Cls	Predict whether a user will give no beer rating in the next 90 days.
	brewer-dormant	Cls	Predict whether a brewer will release zero new beers in the next 365 days.
	user-count	Reg	Predict the number of beer ratings a user will give in the next 90 days.

Table 22: Task descriptions of 4DBInfer (5 classification).

Dataset	Task	Type	Description
amazon	user-churn	Cls	Predict whether an Amazon reviewer will stop writing reviews in the next time window.
outbrain	ad-ctr	Cls	Predict whether a user will click on a promoted content recommendation on Outbrain.
retailrocket	item-cvr	Cls	Predict whether a user will convert (purchase) after viewing an item on RetailRocket.
stackexchange	post-upvote	Cls	Predict whether a StackExchange post will receive an upvote.
	user-churn	Cls	Predict whether a StackExchange user will stop posting or participating.

J Limitations

RELAGENT inherits both the strengths and weaknesses of SQL-based search. First, the search space grows combinatorially with the number of tables, join paths, time windows, filters, and aggregation operators, so the agent may miss useful deep or high-order relational features within a fixed turn budget. Second, although SQL can express many relational patterns, discovering the correct query can require brittle multi-step exploration. Third, the current implementation primarily exploits relational structures and statistics, and does not fully process long free-text fields or multimodal attributes. Finally, because the agent searches using validation feedback, care is required to prevent leakage and overfitting to the validation set.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes].

   Justification: The abstract and introduction state the main scope of the paper: an LLM-agent framework for relational learning that searches over SQL feature programs and downstream predictive models, with the LLM removed at inference time. The empirical claims are supported by results on RelBenchV1, RelBenchV2, and 4DBInfer, and the paper states that the final predictor is fast, deterministic, and interpretable because it consists of explicit SQL feature programs and a standard predictive model.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes].

   Justification: The conclusion discusses the main limitation that agentic search is costlier than standard model selection and does not guarantee finding the best SQL program. The experiments and appendices further discuss search budget, efficiency, backbone dependence, and failure cases in the synthetic triangle experiment.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best

judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A].

   Justification: The paper does not present formal theoretical results or theorems. The properties section provides a conceptual discussion of interpretability, expressivity, and query-induced invariance, supported by examples and experiments rather than formal proofs.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper?

   Answer: [Yes].

   Justification: The paper describes the benchmark datasets, task splits, evaluation metrics, model choices, agent prompts, tool interfaces, search budget, selected model configurations, and efficiency details. The appendix includes dataset statistics, prompts, task descriptions, selected programs, ablations, and search setup. We also include the github repo for reproducing the experiments

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.

(c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes].

   Justification: The paper provides an anonymized code URL and uses publicly available benchmark datasets where licenses permit access. The experimental details, prompts, and selected configurations are included in the appendix. The MIMIC dataset from RelBenchV2 is excluded due to licensing constraints.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details necessary to understand the results?

   Answer: [Yes].

   Justification: The paper specifies the benchmarks, tasks, metrics, train/validation/test setup, agent search budget, number of independent searches, model families, hyperparameters, SQL engine, LLM backbones, and final model-selection protocol. Additional task descriptions and selected programs are provided in the appendix.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes].

Justification: The main benchmark tables report the test score of the selected run and the standard deviation of test scores across five independent agent runs; see Table 16 and Table 17. These standard deviations quantify run-to-run variability from the stochastic program-search procedure under fixed data splits and evaluation protocols. We discuss the observed variance in the experimental analysis: most tasks have small standard deviations, while larger values occur primarily on low-sample datasets. We do not claim formal statistical significance and do not report confidence intervals or hypothesis tests. Efficiency statistics are additionally reported as mean $\pm$ standard deviation.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources needed to reproduce the experiments?

   Answer: [Yes].

   Justification: The efficiency appendix reports wall-clock time, agent turns, tool invocations, token counts, and estimated cost per run for GPT-5.2, DeepSeek-V4-Pro, and Qwen3-32B. It also states that Qwen3-32B is served locally using vLLM on 2 NVIDIA RTX 6000 Ada GPUs with 48GB memory each. The paper describes API-based access for the other LLM backbones.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics?

Answer: [Yes].

Justification: The work uses existing relational learning benchmarks and does not introduce human-subject experiments, scraped personal datasets, or high-risk model releases. The paper respects licensing constraints by excluding MIMIC from RelBenchV2.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes].

Justification: The paper discusses both positive impacts, such as reducing manual feature-engineering effort and improving auditability through explicit SQL feature programs, and negative impacts, such as the risk that autonomous feature search may discover sensitive or proxy variables in relational databases. It also recommends privacy controls, data governance, and fairness auditing for high-stakes applications.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse?

Answer: [N/A].

Justification: The paper does not release a new high-risk pretrained model, scraped dataset, or generative system. The released asset is code for an experimental relational-learning framework evaluated on existing benchmarks.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.

- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets used in the paper properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes].

    Justification: The paper cites the datasets, software libraries, and model families used in the experiments, including RelBench, 4DBInfer, DuckDB, LightGBM, XGBoost, CatBoost, CAMEL, and the LLM backbones. The paper also notes that MIMIC is excluded due to licensing constraints.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes].

    Justification: The paper releases code for RELAGENT and documents the system through method descriptions, tool schemas, prompts, task descriptions, selected programs, and experimental settings. No new dataset is introduced.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation?

Answer: [N/A].

Justification: The paper does not involve crowdsourcing or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A].

Justification: The paper does not involve crowdsourcing or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes].

Justification: LLM usage is central to the proposed method. The paper describes the LLM agent, tool interfaces, prompts, search procedure, LLM backbones, inference-time removal of the LLM, and efficiency/cost analysis.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.