Query-Centric Optimization of AI Workflows via Approximate Query Processing and Proxy Models

Huayi Wang
Georgia Institute of Technology
Atlanta, USA
hwang762@gatech.edu

Jun Xu
Georgia Institute of Technology
Atlanta, USA
jx@cc.gatech.edu

Gromit Yeuk-Yin Chan
Adobe Research
San Jose, USA
ychan@adobe.com

ABSTRACT

Many modern AI workflows—ranging from LLM post-training pipelines to agentic reasoning tasks—can be expressed as declarative queries whose expensive predicate is evaluated by a large model or reward function. We propose a *query-centric* formulation of these workflows and show that classical database techniques, namely approximate query processing (AQP) and proxy-model (PM) based filtering, can substantially reduce the number of expensive model invocations without changing the database, model and pipeline through exposing the invocations that could be avoided with preserving aggregate or selected-output quality. AQP treats the workflow as an online aggregation problem: it progressively samples records, maintains a running aggregate estimate with an approximate confidence interval and an empirical stopping rule based on interval narrowing and stabilization. PM trains a lightweight, CPU-resident decision tree on a small set of oracle-labeled examples and uses it to pre-filter records using structural features, with oracle confirmation and sampled correction for aggregate queries and candidate selection. We also propose a unified plan on a shared label cache where they yield the AQP estimate and train the proxy to select the best strategy at negligible CPU overhead. We evaluate on TPC-DS aggregate queries and real LLM post-training pipelines including math reasoning, general instruction following, and code generation. On TPC-DS, AQP keeps aggregate error under 10% using about 10% to 15% of oracle calls in the balanced-distribution experiments, and PM reduces oracle calls by 60–70% on several natural-label workloads, with only challenges faced on small-table cases. On LLM pipelines, fixed-budget sampling provides a cost-quality reference, and PM uses up to 19 times fewer RM calls than full scoring with similar accuracies like 91.16% vs 92.95% on LLaMA-3.1-8B Math.

PVLDB Reference Format:
Huayi Wang, Jun Xu, and Gromit Yeuk-Yin Chan. Query-Centric Optimization of AI Workflows via Approximate Query Processing and Proxy Models. PVLDB, 14(1): XXX-XXX, 2027.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Why1221/Query-Centric-AI-optimization.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

The rapid growth of expensive Large Language Model (LLM) computation in AI workflows [4, 13]—from training foundation models to invoking them hundreds of times in agentic task pipelines [3, 28]—has created new opportunities to optimize latency and cost. More importantly, many of the current AI-centric workflows can be expressed as database queries. Consider the following two examples:

Example 1.1. A recurring pattern in modern LLM post-training pipelines — explicitly documented in both the Llama 2 and Llama 3 technical reports [10, 36] — is what can be understood as the following SQL query operation:

```
SELECT * FROM model_outputs
ORDER BY reward_model_score DESC LIMIT K
```

Demonstrated in Figure 1, for each training prompt, N candidate responses are sampled from the current model, scored by a reward model, and only the top-scoring response is retained as supervised fine-tuning data for the next training round. Here, N denotes the generated pool size and K the number retained with descending reward order in the query. This generate-rank-select loop is a central mechanism by which the model bootstraps its own training data quality without additional human annotation.

Example 1.2. Agentic reasoning tasks—workflows in which an AI model must gather context, apply multi-step inference, and produce a decision or action—are typically treated as orchestration problems, implemented through bespoke application logic spread across pipelines, queues, and LLM API calls. Yet consider a canonical example: *"computing the average order value among customers whose recent orders suggest churn risk"*, a task that today is commonly delegated to an agent framework with hand-written retrieval, prompt assembly, and filtering logic [3, 31]. As illustrated in Figure 2, this workflow can be expressed as the following single declarative query:

```
SELECT customer_id, AVG(order_total)
FROM orders WHERE llm_classify(order_description,
'churn_risk') = 'HIGH' GROUP BY customer_id
```

where `llm_classify` itself may encapsulate a multi-step agentic pipeline, yet appears in the query simply as a semantic predicate that filters records before a standard aggregation is applied. This observation suggests that agentic reasoning tasks can be naturally viewed as declarative queries with semantic predicates.

These examples illustrate query-centric AI workflows, where the final result determines how much model evaluation is useful: an aggregate summarizes many records, while a selection retains only a few. This exposes oracle evaluations that contribute little to the requested result, and we avoid them through aggregate sampling and structural proxy filtering, leaving the oracle unchanged.

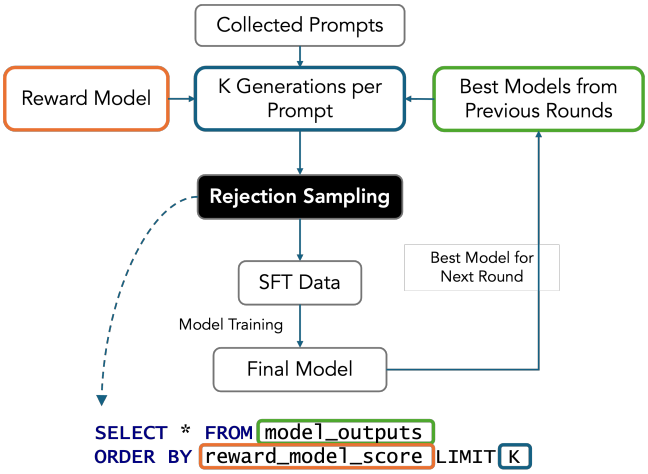


Figure 1: Illustration of rejection sampling as a SQL query in the LLM post-training recipe.

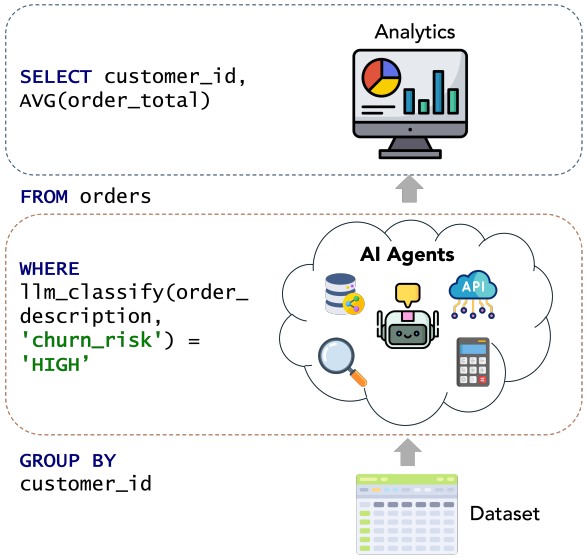


Figure 2: Illustration of an analytical query involving AI agents as a User Defined Function (UDF).

Our Approach. We use standard database and statistics tools: progressive uniform sampling estimates aggregates, with an approximate interval guiding the sampling budget. A CPU decision tree provides a complementary proxy from structural features, such as response length and formatting. Oracle confirmation and sampled correction support aggregate queries, while a gate and filter reduce RM calls for response selection. The query formulation connects these database techniques to the cost and quality of the workflow's final output. Our shared execution reuses uniform oracle labels and original data representations for both estimation and proxy training, avoiding a second labeling budget and further data transformations. It completes the plan with fewer remaining oracle calls first and reuses cached labels if the other plan continues. Unlike optimizing agreement with individual oracle labels, we evaluate the numerical aggregate or selected response required by the query. Related Work distinguishes this execution design from declarative AI systems [21, 31, 35], prediction-based approximation [40, 41] and embedding-based query optimization approaches [6, 9, 18].

We conduct a comprehensive experimental evaluation on two classes of query-centric AI workloads. The first is a set of TPC-DS queries that replay known predicate labels to measure oracle-row demand; these experiments demonstrate the effectiveness of the AQP-based approach, showing that progressive sampling achieves <10% aggregate error while invoking the oracle on only around 10% of tuples under adjusted balanced label distributions. Natural small-table queries can remain difficult at a roughly 20% budget. An additional shared-execution experiment measures the cost of an early PM output followed by AQP, without requiring full evaluation. The second is a suite of real LLM post-training pipelines spanning math reasoning (GSM8K), general instruction following (UltraFeedback), and code generation (HumanEval+, MBPP), where we evaluate PM selection against fixed-budget reward-model sampling. The proxy-based filter uses up to 19 times fewer RM calls than full scoring, including training; for LLaMA-3.1-8B Math, accuracy changes from 92.95% to 91.16%.

Contributions. Our main contributions are as follows:

- We formulate AI workflows as aggregate or selected-output queries and use their result requirements to identify and avoid unnecessary oracle evaluations through AQP and structural proxy filtering.
- We design shared AQP and PM execution that reuses oracle labels, preserves both estimates, and returns the plan with fewer remaining oracle calls first.
- We evaluate both aggregate strategies on TPC-DS and PM selection on real LLM candidate pools, using up to 19 times fewer RM calls than full scoring on the evaluated structured LLM tasks, including training, and oracle-call reductions of 85% to 90% with <10% aggregate error in the balanced TPC-DS experiments. Additional experiments compare CPU BARGAIN and embedding proxies and measure query-result agreement and variability.

2 BACKGROUND AND PROBLEM

In this section, we describe the background, as well as the inputs, outputs, and goals of our query-centric AI workflow. To begin with, we explain the characteristics of common AI workflows that can be expressed as queries. Consider the following query examples:

```
-- Q1
SELECT AVG(churn_rate), predicted_segment
FROM customer_analytics_db
GROUP BY predicted_segment
-- Q2
SELECT genre
FROM movie_db
WHERE predicted_rating > 4
-- Q3
SELECT *
FROM loan_application_db
WHERE predicted_default = True
```

```sql
-- Q4
SELECT t1.user, t2.movie
FROM user_db as t1
LEFT JOIN movie_db as t2
ON t1.predicted_preferred_genre = t2.genre
--Q5
SELECT *
FROM census_db
ORDER BY predicted_income ASC
LIMIT 50000
```

The queries **Q1-5** demonstrate various needs of predictive AI workflows to output designated tables. In a straightforward data science pipeline, we first fetch all the rows into memory and input them into the workflows. Then, after acquiring the predictions, we store them in a separate column and aggregate or filter the rows to output the results. Yet from a data management perspective, the above workflow causes a lot of data processing on all rows in the database, but only a small fraction of rows appear in the final output. For example, in **Q3**, if the default rate is only 10%, a reliable proxy or early filter may reject many likely-negative entries before they reach the full AI pipeline. To sum up, a query-centric AI workflow contains two types of speed-up opportunities. First, there is a *filter* condition (**Q2-4**) where we only need to focus on a subset of labels. We could derive a simpler pipeline that optimizes the subset's task. Query clauses like `WHERE`, `JOIN`, and `LIMIT` could trigger such a condition. Secondly, there is also an *aggregate* condition (**Q1**) where we could consider different weights on different labels. For example, for conditions like `GROUP BY`, rows with a particular label might contain similar value distribution on the numerical column so that we do not need to query many of them to get the average.

Furthermore, given the needs of feature engineering, the AI pipeline might need to query lots of columns to produce the predictions. Many predictive systems are powered by expensive AI pipelines that contain complex (1) feature engineering and (2) model architecture. We refer to these inference steps as *targeted AI pipelines*. The pipeline queries the data needing predictions from sources and acquires the data in a tabular data format. Then, it transforms the input data into a target schema. For example, records are transformed into labels by a classification pipeline, numbers by a regression pipeline, and clusters by a clustering pipeline. Yet, as inference pipelines grow more sophisticated, they become increasingly expensive to execute. We call the supplied pipeline an *oracle*, whose labels or scores define the query result to approximate.

We summarize these problems into two goals. Given a database, a query, and an AI pipeline, we want to (1) derive an estimator of the query result based on samples to reduce the number of records sent through the AI pipeline (**G1**); and (2) build a simpler proxy model that estimates the AI workflow's result on the query efficiently and accurately (**G2**). While many query optimization techniques for ML inference have been proposed to speed up the process [13, 16, 23, 39], our design also aims to operate without changing the database architecture. In the industrial setting, it is preferred to have minimal changes on the query processor and the layers underneath due to extra costs incurred on database migration and maintenance. Besides, the database already contains several optimization features like caching and indexing that we can leverage.

2.1 Inputs, Outputs, and Goals

Our strategies take a query, a dataset, and an AI workflow as input, and output an approximated query result. Our goal is to reduce the number of records sent through the AI pipeline and the cost of its model invocations. The application identifies the expensive predicate or score function, and ordinary relational operations produce its input relation. Our prototype does not discover arbitrary model calls or search join orders. Aggregate error, selected-response quality, and exact winner agreement are separate evaluation objectives.

To be specific, for the targeted ML pipeline, we expect the output to be either a *classification* label or a *regression* value. Our scope of the queries is as follows:

(1) *Aggregate Queries* on a numerical column with the group-by clause based on ML pipeline.
(2) *Selection Queries* on particular rows with the filter clause based on ML pipeline.
(3) *Limit Queries* that control the size of output with sorting criteria based on the ML pipeline.

From these queries, we can observe that some predicted labels or values are more important than others. For example, we only need to focus on those data points whose predicted values are in a certain range. Thus, a query-centric AI workflow, unlike typical Machine Learning (ML) pipelines that assign equal importance to each row's predictions, results in different preferences on the pipeline output. Our strategies first collect a uniform sample of oracle outputs. The same labels support an aggregate estimate and training a proxy for the query's required output as we will elaborate in Section 4.

3 RELATED WORK

Our work draws on and departs from three bodies of literature. We first review classical approximate query processing in Section 3.1, which provides the statistical foundation for our AQP formulation. We then discuss query optimization systems that handle ML predicates inside a database system in Section 3.2, which share our goal of reducing oracle invocations through sampling or proxy models. Finally, we review LLM pipeline optimization work in Section 3.3, including declarative systems that exploit query structure. Our focus is reducing calls to a supplied oracle according to aggregate and selected-output requirements.

3.1 Approximate Query Processing

Online aggregation [11] introduced the idea of computing aggregate results progressively as rows are scanned: after each batch of samples, the system updates the aggregate estimate and its confidence interval, allowing users to inspect the current precision. WanderJoin [19] extends this framework to queries with joins. The core difficulty is that uniform sampling from each table independently produces few join matches when join selectivity is low, making the estimator high-variance. WanderJoin resolves this by performing random walks over the join graph and weighting their contributions to obtain an unbiased aggregate estimate.

Our AQP formulation applies this framework to AI workflows, where each sampled value requires an expensive oracle or reward-model call. Our experiments start from a materialized relation and focus on reducing oracle invocations. The approximate intervals

guide the sampling budget, while empirical error is reported separately. BlinkDB [1] uses stratified samples, while VerdictDB [27] provides AQP through middleware without changing the database. Our unchanged-engine design is compatible with such integration.

3.2 Query Optimization with ML Predicates

Prior work on optimizing queries with expensive ML predicates falls into several related settings. NoScope [16] uses specialized cascades for video queries, while BlazeIt [15] uses proxy models for video aggregation and LIMIT queries. TASTI [18] constructs semantic indexes for unstructured data, including video, text, and speech. Its embeddings supply proxy scores without training a separate proxy for each query.

For general ML predicates expressed as UDFs, PP [23] and CORE [39] rewrite the query plan to insert proxy filters ahead of costly ML UDFs inside the DBMS. CORE relaxes PP's independence assumption across predicates using branch-and-bound search, achieving up to 63% higher throughput than PP. ABae [17] uses a learned proxy to allocate sampling effort across strata. Aero [13] monitors UDF statistics at runtime and adapts predicate evaluation.

Our proxy deliberately relies on classical database and data-science tools rather than neural encoders. Cheap, explicit features (length, symbol density, formatting, lexical statistics) feed a single decision tree that is inexpensive to fit, easy to inspect, and whose splits are comparisons expressible as SQL predicates. It evaluates extraction, fitting, and prediction on CPU outside the engine.

3.3 LLM Pipeline Optimization

A parallel line of work optimizes LLM-powered workflows from the machine learning and systems perspective. These efforts fall into three categories: designing a declarative structure of LLM pipelines, routing queries to cheaper models, and reducing generation cost in Best-of-N sampling.

First, LOTUS [28] introduces semantic operators, a declarative programming model that extends relational algebra with LLM-based operations such as semantic filtering, joining, and ranking over unstructured text. Abacus [29] selects implementations of semantic operators under quality, cost, or latency constraints. TAG [3] frames natural language questions as table-augmented generation tasks, enabling accurate analytical query answering without Text-to-SQL. DocETL [31] takes an agentic approach to pipeline design, rewriting user-specified LLM pipelines into decomposed plans for document processing. These systems optimize the pipeline itself. We treat it as a black box and reduce oracle calls through sampling and filtering, so the two lines of work are complementary.

For optimization through model routing, FrugalGPT [4] routes each query to the cheapest LLM that can answer it accurately. Task Cascades [32] decomposes a task into a cascade of cheaper sub-operations, escalating only uncertain records to the expensive oracle. The approaches can reduce expensive model calls through choices of models, document portions, or tasks. Our evaluation instead keeps the oracle unchanged.

Best-of-N selection is used in LLM post-training [10, 36]. Speculative Rejection [34] and the Pandora's Box formulation [14] reduce candidate-generation work. Our comparison holds generated candidates fixed but reduces RM scoring. Generation and fine-tuning costs are outside the reported savings.

Closest approximation methods. BARGAIN [40] targets agreement with oracle labels or precision and recall for filtering only. Our objective also covers the numerical aggregate or selected response requested by the query. Thus, in Section 5.1, we compare the authors' AT implementation on a subset of TPC-DS queries using the same CPU proxy and training rows.

Another recent study from Chung *et al.* [6] uses lightweight CPU classifiers over embeddings from neural networks for semantic filtering and ranking. Its gains depend on representation quality and the amortization of embedding and training costs. Our RM cascade uses structural features to predict continuous scores and reduce scoring through a gate and candidate filter. As we will show in Section 5.2, surface features used in our proxy model retain competitive selection quality while avoiding neural feature-extraction cost on our workloads. For aggregates, our shared execution preserves a uniform-sample AQP estimate and uses the same labels to train PM. The remaining oracle requests determine which configured plan completes first, with label reuse if the other plan continues as we will elaborate in Section 4.3.

Approximate and declarative AI queries. ThalamusDB [12] uses model scores and user calibration to bound multimodal query results. UQE [9] supports aggregate and retrieval queries, using embedding-based stratification for aggregation and active learning for retrieval. Palimpzest [21] chooses AI-operator implementations, while DocETL [31] rewrites document-processing tasks. ZenDB [20] uses document hierarchies, and QUEST [35] reduces extraction work through retrieval and document-specific plans. SEED [5] compiles data-curation cascades using generated code and small models. The systems establish the value of declarative AI optimization. Our query-centric design uses aggregate and selected-output requirements to coordinate uniform AQP with a structural CPU proxy for a supplied oracle.

Cross-prediction [41] combines fitted predictions and sampled labels to obtain corrected statistical estimates. Our shared execution instead preserves a proxy-independent AQP estimate alongside the PM result. Uniform labels serve both estimation and proxy training, and a shared cache eliminates duplicate oracle calls. Comparing the remaining requests determines which configured plan completes first, making its result available before the other plan finishes.

Our experiments fix the input relation or candidate pool and oracle to isolate call reduction at comparable aggregate or selection quality. In contrast, ThalamusDB and UQE use calibration or embedding-based strata; Palimpzest and DocETL optimize operator implementations or task decompositions; QUEST and ZenDB optimize document extraction through retrieval or document hierarchies, whereas our queries operate on materialized records or generated response candidates. Their full-pipeline results would thus include effects beyond our estimation and selection strategies. We therefore discuss these systems qualitatively rather than use them as direct experimental baselines.

4 QUERY-CENTRIC AI WORKFLOW INFERENCE

The core idea of an inference optimization from a query-centric AI workflow is that we do not need to fetch all the data points and invoke the AI pipeline on each of them to obtain accurate results that require predictions.

For example, if the query aims at collecting the `AVG(c1)` where the predicted values at `c2` are equal to `"label 1"` while all values in `c1` are equal to 0, the average is zero whenever at least one row qualifies. No further labels are needed to compute that value.

On the other hand, if we do not aggregate the rows in the query result (e.g. `SELECT`), we still do not need to invoke the AI pipeline on much of the database if the query has a filter by prediction labels. Therefore, there exists an execution plan that decides how many AI-pipeline invocations we need, as well as how they should be constructed. In the following sections, we describe two main strategies to process the queries without invoking the AI pipeline on all data from the database, their tradeoffs, and how to interplay them together to optimize query performance.

4.1 Strategy AQP: Treating the Workflow as an Approximate Query Processing Problem

The first strategy treats these workflows as an Approximate Query Processing (AQP) problem and estimates aggregate results from samples. We use an online aggregation approach [11] to provide approximate results with confidence intervals and gradually improve with an increasing number of samples. Given an `AGG(expression)` and any samples where `AGG` could be `SUM`, `AVG`, `COUNT` or `VAR` and `expression` could involve any columns in the database, the desired accuracy target for the estimator $\hat{Y}$ is

$$Pr[|\hat{Y} - \texttt{AGG(expression)}| \leq \epsilon] \geq \alpha. \tag{1}$$

Here ϵ is the requested half-width and α the confidence level. Each round draws a uniform batch with replacement and updates the estimate and approximate half-width. The width guides the sampling budget, and a stalled width is reported with its consumed budget. Equation 1 states the desired coverage. The normal interval below is a fixed-budget approximation, not a simultaneous guarantee after repeated inspection. Repeated draws retain their multiplicities, while cached labels avoid repeated oracle calls.

4.1.1 Illustration: AQP with `SUM`. For concreteness, we first address the following `SUM` query:

```
SELECT SUM(c1) WHERE c2 = "label 1"
```

Let z_γ indicate whether row γ satisfies the oracle predicate, and let a_γ be its value in `c1`. A uniform draw has probability $p(\gamma) = 1/N$, so $z_\gamma a_\gamma / p(\gamma)$ estimates the full `SUM` without bias. For a batch of b draws, the round estimate is $v_i = (N/b) \sum_{\gamma \text{ drawn}} z_\gamma a_\gamma$, with repeated draws counted repeatedly. Averaging independent round estimates reduces variance as more samples are collected. The interval below summarizes the uncertainty of the estimate and guides the sampling budget.

4.1.2 AGG Functions. SUM and COUNT use scaled sample totals, with $a_\gamma = 1$ for COUNT. AVG uses the ratio of the sampled qualifying sum to the qualifying count. A round with no qualifying row has an undefined AVG and is excluded from the finite round estimates. VAR requires an estimator based on the appropriate sample moments. Let the finite aggregate estimate on the i-th round be v_i. Then the estimator of $\hat{Y}$ with n trials is: $\hat{Y} = \frac{1}{n} \sum_{i=1}^{n} v_i$. To compute the approximate half-width in Equation 2, we estimate the sample standard deviation: $s = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (v_i - \hat{Y})^2}$. With the estimator $\hat{Y}$ and the sample standard deviation s, the approximate half-width is

$$h_n = \frac{z_{(1+\alpha)/2} s}{\sqrt{n}}. \tag{2}$$

Here n counts finite round estimates and $z_{0.975} = 1.96$ for $\alpha = 0.95$. The reported half-width describes empirical precision at the current budget, rather than certifying coverage after repeated inspection. For the relative-error experiments, let r count sampling rounds and let $n(r)$ count their finite estimates. We use $H_r = 100 h_{n(r)} / |\hat{Y}_{n(r)}|$ and a target of 10%. A plateau requires finite current and earlier widths, and is detected when

$$H_r > 10, \quad H_{r-5} > 0, \quad \frac{H_{r-5} - H_r}{H_{r-5}} < 0.05, \quad r \geq 7. \tag{3}$$

The patience window avoids reacting to a single round. When a plateau is detected, the system reports the current interval and the oracle cost consumed so far, and the user can stop or escalate to full evaluation (Section 5.1).

Other AGG functions, such as MIN and MAX, are difficult for the sampling method. They usually do not follow a certain distribution like Gaussian. Assuming these values depend on one unique row only, and each independent round samples b distinct rows uniformly, the expected number and standard deviation of sampling trials needed will be $\frac{N}{b}$ and $\sqrt{(1 - \frac{b}{N})/(\frac{b}{N})^2}$ where N is the number of rows and b is the sample size. Each round hits the extreme row with probability b/N, so the waiting time is geometric. With b draws with replacement, the hit probability instead is $1-(1-1/N)^b$. If $b \ll N$, the waiting-time variation is large. If b is not much smaller than N, sampling becomes comparable to a full table scan. In both cases, a full scan is preferable. To address the limitation on these AGG functions, we introduce the second strategy that we should construct queries to select points with target labels with small cost, in the next section.

4.1.3 LIMIT Queries. Beyond AGG functions, Strategy AQP extends naturally to `LIMIT` queries of the form:

```
SELECT * ORDER BY score DESC LIMIT k
```

This pattern appears directly in LLM post-training pipelines: for each prompt, N candidate responses are generated and scored by a reward model, and only the top-k responses are retained [10]. Rather than invoking the oracle on all N candidates, we ask: how many candidates must we sample so that the global top-k are likely to be included?

Candidate coverage. For a uniform sample of fixed size $m \geq k$ without replacement, the probability of including every global top-k candidate is

$$P_k(m, N) = \prod_{i=0}^{k-1} \frac{m-i}{N-i}. \tag{4}$$

The identity counts subsets containing the designated top-k, without assumptions about score independence. A fixed budget can be

chosen to satisfy $P_k(m, N) \geq \alpha$. For $k = 1$, the probability is m/N. For large N and fixed k, the required budget is approximately

$$m \approx N\alpha^{1/k}. \tag{5}$$

The fraction increases with k, because every designated candidate must be present. Winner recovery and task quality therefore have different sampling requirements.

Adaptive Stopping in Practice. Equation 4 is exact for fixed-size uniform sampling. For applications that accept an empirical quality tradeoff, a score-stabilization heuristic instead uses a tolerance $\delta > 0$ and checks whether the worst score among the current top-k changes little:

$$|r_{(k)}^{(m)} - r_{(k)}^{(m-1)}| \leq \delta, \tag{6}$$

where $r_{(k)}^{(m)}$ denotes the k-th highest reward score observed after drawing m ($m \geq k + 1$) candidates. Equation 6 records score stabilization rather than a probability of recovering unseen candidates. An unchanged score can occur before a better candidate is sampled. We retain the rule as an empirical option, separate from the fixed-size identity in Equation 4. Our evaluation (Section 5.2) shows that scoring a small fraction of candidates already gets close to full-RM quality, at 1.9× to 8.0× lower estimated oracle scoring time at the offline stabilization points.

4.2 Strategy PM: Proxy Model to Increase Throughput on Model Inference

The second strategy to avoid feeding all data to the AI workflow is to develop a predictive model that **handles the "easy" input** before passing the remaining data to the AI pipelines. We define easy data as rows whose pipeline outcome can be predicted with high confidence by a rule-based model, or whose prediction error does not affect the *final query result*. For example, errors on candidates outside the returned set need not change a selection query's output. Thus, the idea is to train an efficient proxy model with limited ground truth samples and process the majority of the data while retaining useful result quality, thus increasing the throughput of the query.

4.2.1 Illustration: SELECT rows with label "1". Let us consider a simple query: `SELECT * FROM table where label = '1'` A lightweight proxy predicts the predicate label from inexpensive row features. First, a uniform sample obtains oracle labels, or reuses those already collected by AQP. Then, the sampled labels train a query-specific decision tree. Afterward, its predictions determine which remaining rows receive oracle evaluation. For aggregate queries, a random audit also accounts for positives among excluded rows. The proxy's purpose is to reduce work for the query result, rather than reproduce every oracle prediction.

4.2.2 Overview of Proxy Model. We use a Decision Tree (DT) because it is inexpensive to fit and its feature comparisons can be expressed as SQL predicates. The proxy learns from a small uniform set of oracle-labeled examples, including samples already collected by AQP. Figure 3 illustrates the workflow: a sampled subset provides oracle labels for a regression tree, which then predicts scores for the candidates of unseen query groups from surface features. In the RM workload, the oracle is the reward model and a query group

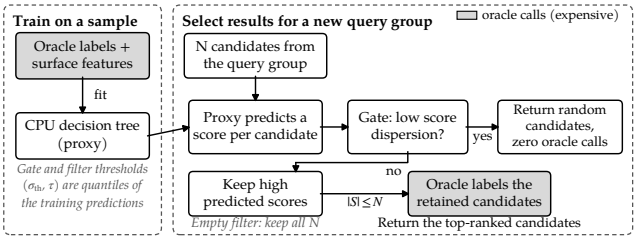


Figure 3: Strategy PM. A CPU proxy trained on a small oracle-labeled sample predicts a score for each candidate in a query group, and only retained candidates receive oracle labels. Reward-model scoring of LLM responses is one instance; Table 1 gives the cascade presets used there.

is the set of candidate responses for one prompt. Our evaluated training budgets are 5% or 10% of prompts in the LLM workload and a sweep of row fractions in TPC-DS. The budgets are workload settings rather than a universal minimum. We use a decision tree because it is cheap to fit and its splits are simple comparisons that translate directly into SQL predicates. In our prototype, feature extraction, fitting, and prediction run on CPU outside the database.

Aggregate correction. For TPC-DS aggregates, PM includes known training labels exactly and obtains oracle labels for all remaining rows that the proxy does not confidently exclude. The remaining predicted-negative pool U receives a uniform random audit at a configurable fraction; Section 5.1 gives the fractions used in our experiments. Let C contain the training and confirmed rows, and let $A \subseteq U$ be the audit sample. The corrected SUM is

$$\hat{Y}_{\text{PM}} = \sum_{i \in C} z_i a_i + \frac{|U|}{|A|} \sum_{i \in A} z_i a_i. \tag{7}$$

An empty pool contributes zero, and a non-empty pool receives at least one audit row. COUNT uses $a_i = 1$. AVG divides the corrected SUM by the corrected positive count. The sampled correction accounts for positives among excluded rows. It is separate from the per-prompt selection cascade below.

4.2.3 Feature Engineering. The proxy uses cheap features from the input record, including length, symbol density, formatting, and lexical statistics. For LLM responses, answer markers and numeric structure can help predict oracle scores without a neural encoder. Smaller models and embeddings can also observe such properties, so our claim concerns extraction cost and usefulness. Section 5.2 compares representations separately from the regression model.

4.2.4 Cascade Gating. We now expand the rejection-sampling cascade, which is the form used throughout our LLM evaluation (Section 5.2). The query is `SELECT * ORDER BY rm_score DESC LIMIT k` over the N candidates of each prompt, and the proxy is a decision tree that predicts the oracle score from surface features. We specialize to top-1 ($k = 1$) because our evaluated workflow selects one of 32 responses per prompt. Each prompt forms an independent query group for the cascade's two coupled tests.

Variance gate (group-level skip). The tree first scores all N candidates of a prompt and we measure the within-prompt dispersion

Table 1: Cascade presets used in the LLM experiments.

Task and training condition	Preset	q_σ	q_s
Math/Code, mean RM < 6.5	A	off	0.60
Math/Code, mean RM ≥ 6.5	B	0.99	0.50
General, Qwen3-8B	A	off	0.90
General, LLaMA-3.1-8B	A	off	0.97
General, Mistral-7B-V0.3	A	off	0.30
General, Qwen3-32B	A	off	0.88

of these predictions, $\sigma_{\text{pred}} = \text{std}(\hat{s}_1, \ldots, \hat{s}_N)$. When σ_{pred} falls below the threshold σ_{th}, the gate returns one candidate using a fixed random seed and issues zero oracle calls for the prompt. Low predicted dispersion identifies groups for which the proxy provides little ranking information. The gate exploits the empirical quality of candidates in structured tasks, rather than certifying equality of their true oracle scores.

Candidate filter (within-group pruning). If a prompt survives the variance gate, we avoid scoring all N candidates by pruning candidates with low predicted scores. We retain the survivor set $S = \{ c : \hat{s}(c) \geq \tau \}$, where τ is a score threshold, forward only S to the oracle, and select $\arg\max_{c \in S}$ over the true oracle scores. If S is empty, meaning no candidate clears τ, we fall back to forwarding all N candidates to obtain a full-oracle selection for that prompt. This costs $|S|$ calls for nonempty S (N when empty).

Both thresholds are computed on the training sample alone, after fitting the tree to oracle labels. Let q_s denote the score quantile and q_σ the dispersion quantile. We set $\tau = Q_{q_s}(\{\hat{s}(c) : c \text{ in training}\})$ and $\sigma_{\text{th}} = Q_{q_\sigma}(\{\sigma_{\text{pred}}(p) : p \text{ in training}\})$. The standard deviation uses divisor N. No held-out labels enter threshold selection. The filter is one-sided, and the oracle ranks only the retained candidates. A nonempty survivor set does not establish exact top-k recovery. The supplementary agreement analysis is in our technical report [37].

Two cascade presets. Preset A disables the gate and applies only the candidate filter. Preset B enables both tests. General instruction following uses A, while Math and Code use B when the mean training oracle score is at least 6.5 and A otherwise. Table 1 separates the preset parameters from the workflow figure. The policy uses the observed quality of training candidates to adjust the scoring budget.

Scope. Overall, PM targets selection queries where any sufficiently good candidate is an acceptable answer, as in Math and Code, where a pool often holds several correct responses (Section 5.2). It does not recover the exact top-scoring candidate. When that candidate matters, as on the open-ended General task, it saves little and loses quality. Users who need the exact winner would need a fallback mechanism in which we will discuss below.

4.2.5 Fallback or Early Termination. The implemented cascade returns a candidate without oracle calls when the variance gate fires. If the gate does not fire, an empty filtered set triggers full-oracle scoring. The mean training oracle score selects the preset, and there is no separate reliability test on the gate. Users requiring the exact full-oracle winner can disable approximate pruning and score all candidates. For approximate selection, the measured task quality and agreement describe the resulting tradeoff.

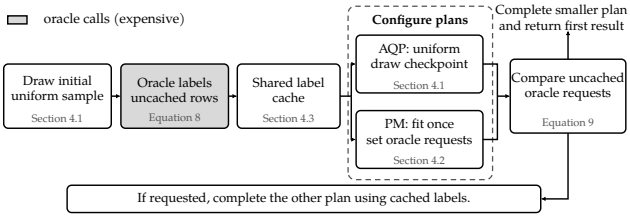


Figure 4: Shared AQP and PM execution. Initial uniform labels support AQP and train PM. Comparing remaining uncached requests determines which configured plan completes first; the other can continue using cached labels.

4.3 Deciding Which Strategy to Use

Query structure determines the output objective. Aggregates can use either estimator, while PM supports approximate selection and LIMIT admits fixed-size sampling. Low selectivity and variable qualifying values can make AQP less precise. PM helps when inexpensive features identify useful records, but class imbalance alone does not establish a good proxy. The strategies share labels rather than requiring a second sampling phase. Our 5% to 10% initial budgets are workload settings, not universal minimums.

Shared samples and oracle cost. Shared execution avoids paying twice for oracle labels needed by AQP and PM. First, ordinary joins and filters produce the input relation for the supplied expensive function. Second, uniform oracle sampling gives an initial AQP estimate. Third, the already labeled sample trains a CPU proxy without a second training-label budget. Figure 4 shows how both plans reuse the labels throughout execution.

If L contains cached row labels and a plan requests rows D, only $D \setminus L$ needs oracle evaluation. Total cost is

$$C = C_{\text{features}} + C_{\text{fit}} + C_{\text{predict}} + C_{\text{route}} + \sum_{i \in L_{\text{final}}} c_f(i). \tag{8}$$

Each cached oracle row is charged once, while the CPU terms account for feature extraction, fitting, prediction, and routing.

Output order and continuation. After fitting PM, its confirmation set and configured audit determine requests D_{PM}. A chosen AQP checkpoint has a uniform draw schedule with distinct row set D_{AQP}. Before observing their unknown labels, compare

$$c_{\text{PM}} = |D_{\text{PM}} \setminus L|, \qquad c_{\text{AQP}} = |D_{\text{AQP}} \setminus L|. \tag{9}$$

If $c_{\text{PM}} < c_{\text{AQP}}$, we complete PM and return its estimate first. Otherwise, we complete AQP first, with ties also favoring AQP. The other plan can then continue on the cached labels if the user wants its result too. Until either plan completes, the AQP estimate from the initial sample serves as a running answer.

Overall, the plans share labels but not samples: AQP averages only its own uniform draws, and PM's proxy and audit weights are fixed once fitted, so either plan can finish first without changing the other's estimate. The coordination also adds little CPU work. In Section 5.2, on the LLM workloads, feature extraction, tree fitting, and prediction together take a few seconds, roughly 1% of the estimated time for full reward-model scoring (167–10,891 s) and at most about 5% in the worst case.

5 EVALUATION

In this section, we evaluate our query-centric optimization on two kinds of AI workflows. Our goals are:

- Using AI workflows derived from TPC-DS queries, we demonstrate that under balanced distributions AGG results can be approximated within 10% error at the original stabilization reference of 10% to 15% of tuples, corresponding to 85% to 90% fewer oracle calls.
- Using LLM post-training tasks with Reward Models as an example, we evaluate Strategy PM against fixed-budget sampling and use up to 19 times fewer RM calls than full scoring, including training. We report task-specific quality gaps and their variability.

Implementation Details. We run the CPU experiments on a workstation with an AMD Ryzen 9 9950X CPU and 96 GB of DRAM. All proxy feature extraction, fitting, and prediction run on CPU. We use a single NVIDIA GeForce RTX 5090 GPU for Qwen3-8B, LLaMA-3.1-8B, and Mistral-7B-V0.3 generation. Qwen3-32B generation uses eight A100 80 GB GPUs.

5.1 Evaluating AGG Queries with TPC-DS

Experimental Setup. We choose TPC-DS as the evaluation workload for two reasons. First, real-world AI-agentic query workflows are largely black-box: their internal predicates and data distributions are opaque, making controlled experimentation difficult. Second, TPC-DS is a widely adopted benchmark that covers representative SQL query patterns with diverse aggregate functions, providing a well-understood setting in which we can precisely control label distributions and isolate the effect of each strategy.

Specifically, we select eight representative TPC-DS queries (Q7, 13, 50, 52, 53, 54, 55, 56) spanning three aggregate functions (SUM, AVG, and COUNT). For each query, we split the original WHERE predicate into two parts: the *predicted* predicate that we could predict using columns in the table and the other structural filters and joins. We construct a data table by joining the relevant fact and dimension tables according to the original query's join conditions, materializing the filters explicitly. The predictive predicate is converted into a binary row label. We then evaluate a single scalar aggregate (SUM, AVG, or COUNT) over all rows satisfying the learned predicate, stripping the original query's GROUP BY, ORDER BY, and LIMIT clauses to isolate the core problem—accurately estimating an aggregate under a predictive predicate with limited oracle calls that mimics an AI workflow.

Since the value distributions of TPC-DS are referenced from real world data tables, applying the original WHERE predicate to dimension columns (e.g., `i_category`, `i_class` for Q53) to produce a natural label distribution limits our opportunities to stress test our strategies with a comprehensive set of ranges of label distributions. Thus, we also set the positive-label rate to 30% and 50% to simulate more balanced distributions that may arise in practice. We define the oracle budget as the number of calls to retrieve these labels from the column. Using known predicate labels, we measure oracle calls and relative aggregate error against full evaluation.

We repeat each configuration multiple times. We report mean aggregate error, with shading showing the 25th to 75th percentiles in the error curves. Both strategies use uniform random sampling to select the data for calling the oracle initially.

Compared methods and settings. We compare the following methods using the same natural predicate labels and full-oracle aggregate.

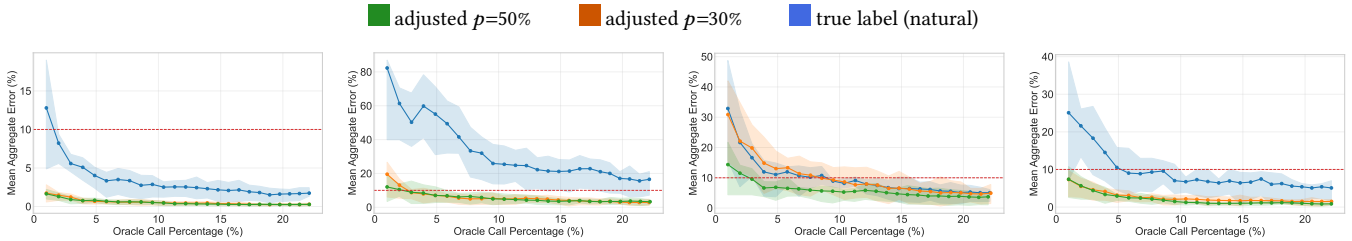
- **AQP** estimates the aggregate from uniform 1% batches. AQP (stop) in Table 2 returns at the first plateau in Equation 3, or at the 25-round cap. It does not stop at the first width below 10% or assume that every returned estimate meets that target.
- **PM** in Table 2 uses oracle confirmation and random auditing, as described in Section 4.2.2. Training uses 5% of rows, except 10% for Q54. The audit fraction is 10% by default, 5% for Q50, and 2% for Q54.
- **BARGAIN AT** uses the authors' adaptive-threshold implementation [40]. AT-90, AT-95, and AT-99 target row accuracies of 90%, 95%, and 99%, respectively. They use failure probability 0.05 and $M = 20$, with the target adjusted for known training labels.

PM and AT share training rows and the same CPU decision tree, using one CPU thread for fitting and prediction. We count every distinct oracle row, including training, calibration, confirmation, and audit. We repeat the comparison multiple times with paired seeds and report mean relative aggregate error. We also repeat shared execution multiple times per query to measure label reuse and output order.

Results for Strategy AQP. We present the results of Strategy AQP under the settings described above in Figures 5 and 6. Figure 5 measures *the real errors of the queries under different percentages of oracle calls* and Figure 6 measures *the estimated CI with increasing numbers of sampling rounds*. For Figure 5, the dots along the line represent the increasing number of sampling trials (e.g., the 4th dot means we have done four trials of sampling). Each trial draws 1% of the data uniformly with replacement so that the total of oracle calls might be less than the total number of samples due to overlapping. Under the adjusted positive-label rates of 30% and 50%, aggregate error falls below 10% with less than 10% of oracle calls. Under natural TPC-DS labels, several larger relations retain low error, while Q13 and Q54 (small tables with fewer than 10,000 rows) can remain above the target after evaluating more than 20% of their rows. In the additional batch-size ablation, 1% offers a near-optimal aggregate-error and oracle-call trade-off among the tested settings at a moderate oracle budget. Smaller batches generally use fewer oracle calls but have higher error; larger batches can reduce error at substantially higher cost. We hypothesize that our sampling strategy obeys the Law of Large Numbers and it might be beneficial for small data to just go through the original AI workflows.

Figure 6 shows the original stabilization reference points at around 10 to 13 rounds. The markers describe the plotted interval trends and correspond to roughly 85% to 90% fewer oracle calls. Under balanced distributions, the approximate interval has narrowed below 10% at those points. Under natural labels, the small-table cases retain wide intervals.

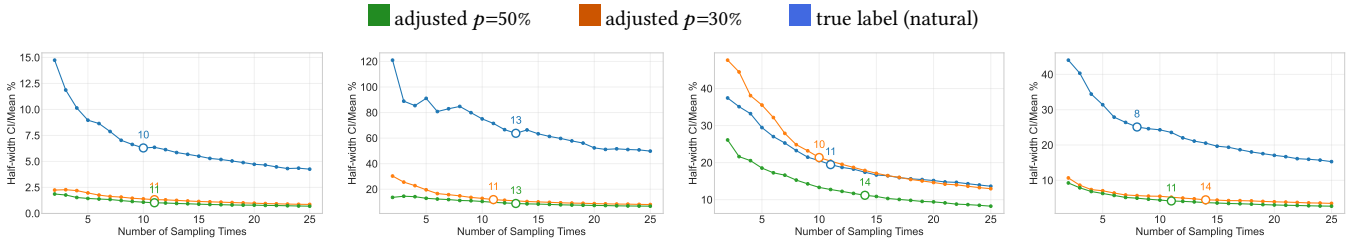
Results for Strategy PM. For Strategy PM, we use the natural TPC-DS labels rather than randomized labels, because randomized labels would destroy learnable structure for the decision tree. Figure 7



(a) Q7 (p=1.4%,n = 529, 437) (b) Q13 (p=1.0%, n = 6, 477) (c) Q50 (p=26.3%, n = 2, 311) (d) Q52 (p=1.7%,n = 86, 313)

(e) Q53 (p=4.1%,n = 541, 763) (f) Q54 (p=6.1%,n = 2, 201) (g) Q55 (p=1.7%,n = 94, 025) (h) Q56 (p=4.5%,n = 20, 165)

Figure 5: Strategy AQP: mean aggregate error vs. oracle-call percentage for all eight TPC-DS queries. Shaded regions show the 25th–75th percentile range; the dashed red line marks 10% error. p denotes the natural positive-label rate and n denotes the table size. Under balanced distributions (green, orange), sampling converges below 10% error within 10% of oracle calls. Under the natural distribution (blue), Q13 and Q54 remain above 10% error at a 20% oracle budget. Their achieved error is reported without requiring full evaluation.



(a) Q7 (AVG, p=1.4%) (b) Q13 (AVG, p=1.0%) (c) Q50 (COUNT, p=26.3%) (d) Q52 (SUM, p=1.7%)

(e) Q53 (SUM, p=4.1%) (f) Q54 (COUNT, p=6.1%) (g) Q55 (COUNT, p=1.7%) (h) Q56 (SUM, p=4.5%)

Figure 6: Strategy AQP: relative half-width of the approximate 95% interval (half-width divided by the current sample mean) vs. number of sampling rounds, for all eight queries. p denotes the natural positive-label rate. Circles mark the offline stabilization reference points. Under balanced distributions the half-width is below 10% at these points; under natural labels, Q13 and Q54 remain above 10%.

retains the original aggregate errors and additional oracle-call rates. The latter count confirmation and audit after training, rather than a separate full-evaluation policy. Training calls are added when reporting total oracle cost. Its original audit settings differ from the final comparison above. Overall, 5 out of 8 queries (Q7, 52, 53, 55, 56) performed as expected, in the sense that the errors reached or stayed at a low margin (< 10%) as the training samples increased to at most 20%, with additional oracle calls decreasing along the way. In total, the oracle calls are around 30–40%, implying a 60–70% reduction in oracle calls to the expensive AI pipeline for these configurations. Q50 and Q54 require many oracle confirmations because the tree confidently excludes fewer rows. On Q13, aggregate error remains high even when PM excludes more rows. The results identify where proxy filtering saves oracle work and where retaining the AQP estimate is valuable. An additional training-size ablation under the final comparison settings shows that 5% training for most queries (10% for Q54) gives a near-optimal aggregate-error and oracle-call trade-off when oracle cost is prioritized. Larger training samples

can lower error on some queries, but usually increase total calls and do not improve error consistently.

Comparison with BARGAIN. Table 2 compares AQP at its stopping point, PM, and BARGAIN AT [40] on natural labels. Query-result accuracy exposes savings that row-accuracy targets alone do not capture. On Q7, PM gives 2.04% aggregate error at 19.75% oracle cost, compared with 4.12% at 18.07% for AT-90. On Q53, AQP gives 1.33% error at 22.12% cost, while AT-95 gives 21.61% error at 37.55% cost. The differences show why the aggregate objective belongs in the execution decision. AQP here uses the online plateau test in Equation 3, or the 25-round cap. The stabilization markers in Figure 6 are separate offline reference points.

AT-99 further reduces aggregate error but uses 75.89% to 100% of oracle rows. For Q7, it gives 0.20% error at 87.42% cost. In the interest of space, the full AT-99 results are deferred to our technical report [37]. Higher row agreement does not ensure a fixed aggregate-error bound, and the methods offer different cost-quality tradeoffs.

Ablation study: Shared labels and output order. We compare shared and separate execution of AQP and PM on the eight natural-label queries. In the shared setting, PM trains on distinct rows from five 1% AQP rounds (ten for Q54). The nominal 5%/10% budgets correspond to Table 2, but repeated draws give slightly fewer training rows. PM uses the same thresholds and audit fractions. Before any further labels are collected, we compare the uncached rows needed by PM and by a 25-round AQP run (Equation 9) and finish the smaller one first.

PM completes first in 75 of 160 (8 queries × 20 trials) runs. On Q7 it gives 1.98% error at 19.65% oracle cost. Continuing to the AQP checkpoint gives 1.66% error at 34.22% combined cost, saving 7.55 percentage points over separate execution. Q50 completes AQP first at 22.08% cost and 4.21% error. Every replay preserves both estimates under either completion order. Q13 remains difficult, with 31.25% PM error and 18.41% AQP error. Overall, we show that label reuse reduces total oracle work while allowing the caller to use the first completed result without paying for both plans.

Ablation study: Stalled intervals. On Q13 and Q54, the interval half-width stays above the 10% target. We compare three policies once the plateau rule (Eq. 3) fires. Stopping at the plateau uses 12.35% of rows on Q13 and 15.42% on Q54 but leaves 30.74% and 19.55% error. Continuing to the 25-round cap costs about 22% of rows and lowers the errors to 18.11% and 14.32%. Labeling all remaining rows removes the error on Q13 at 100% cost and gives 3.79% error at 76.62% cost on Q54. The plateau rule therefore flags hard queries, but it does not by itself make them cheap to answer.

The complete shared-execution and stopping-policy results are deferred to our technical report [37].

5.2 Evaluating Queries Representing Reward-Model-Based Fine-Tuning on LLMs

LLM pipeline workload. To evaluate query-centric system across diverse task domains in post-training pipelines, we select three representative pipelines: a *Math* fine-tuning task based on GSM8K [7], a *General* task based on UltraFeedback [8], and a *Code* task targeting code generation (a combination of two widely-used coding benchmarks HumanEval+ [22] and MBPP [2]). For each prompt from the pipeline, we generate $N = 32$ candidate responses following standard practice in RLHF rejection sampling [10].

Each pipeline uses a task-specific Reward Model (RM), with Math and Code scored by Skywork-Reward-V2-Qwen3-8B [33] and General scored by UltraRM-13B [26], as the oracle to score the generated candidates from LLMs on the questions in the dataset and select the top ones for fine-tuning in the next iteration. Thus, our goal is to reduce reward-model scoring calls while maintaining answer quality; candidate generation and fine-tuning costs are orthogonal to this evaluation. For each domain we evaluate four generator models with varying numbers of parameters to understand our improvements on fine-tuning with different model scales: Qwen3-8B [38], LLaMA-3.1-8B [10], Mistral-7B-V0.3 [24] and Qwen3-32B [38].

Table 2: Oracle calls (% of rows) / aggregate error (%) on natural TPC-DS labels. Lower is better for both; bold marks the fewest in each row.

Query	AQP (stop)	PM	AT-90	AT-95
Q7	21.35 / 2.16	19.75 / 2.04	**18.07** / 4.12	42.54 / **0.90**
Q13	12.35 / 30.74	20.15 / 26.45	**12.22** / 35.15	28.39 / **23.48**
Q50	**13.67** / 8.47	77.05 / **5.23**	81.55 / 15.00	93.04 / 6.49
Q52	**14.60** / **10.88**	20.64 / 11.37	16.24 / 69.75	40.71 / 46.61
Q53	22.12 / **1.33**	24.23 / 1.54	**17.36** / 28.44	37.55 / 21.61
Q54	**15.42** / 19.55	72.77 / 22.36	71.60 / 32.11	75.97 / **15.37**
Q55	**12.94** / **4.12**	20.49 / 5.84	13.14 / 60.04	42.60 / 46.27
Q56	**12.92** / 13.65	29.89 / **11.27**	32.40 / 60.93	57.58 / 41.76

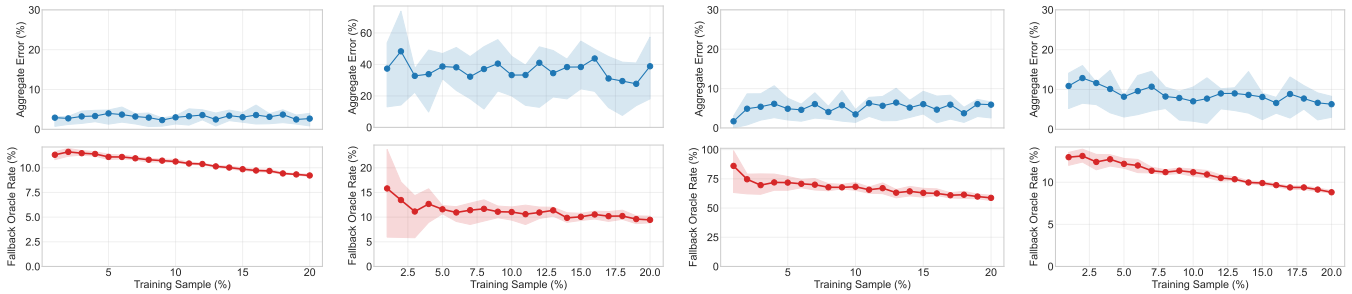
Table 3: PM versus embedding proxies. Each cell shows quality / RM calls (%). Quality is accuracy (%) for Math and Code and mean RM score for General; higher quality and fewer calls are better. Bold marks the best quality and the fewest calls in each row.

Task	Generator	PM	MiniLM	Nomic
Math	Qwen3-8B	**94.21** / **6.26**	92.11 / 7.00	93.26 / 6.75
	LLaMA-3.1-8B	91.16 / **5.23**	**91.68** / 8.97	90.53 / 5.84
	Mistral-7B-V0.3	**75.56** / 46.19	72.00 / 44.30	72.56 / **43.57**
	Qwen3-32B	95.26 / 5.18	95.05 / **5.02**	**95.79** / 6.92
General	Qwen3-8B	**5.69** / 84.90	5.43 / **76.17**	5.64 / 82.10
	LLaMA-3.1-8B	**5.76** / 79.67	4.42 / **65.21**	4.78 / 71.53
	Mistral-7B-V0.3	**7.67** / 70.81	7.54 / **67.81**	7.55 / 67.82
	Qwen3-32B	**5.82** / 69.07	5.23 / **57.46**	5.55 / 69.33
Code	Qwen3-8B	**79.14** / **10.26**	78.57 / 10.99	77.82 / 11.98
	LLaMA-3.1-8B	**66.49** / **10.18**	65.24 / 13.97	65.78 / 11.79
	Mistral-7B-V0.3	**53.57** / 44.35	50.56 / 47.23	49.25 / **43.00**
	Qwen3-32B	**85.20** / **5.60**	84.85 / 13.60	83.96 / 22.30

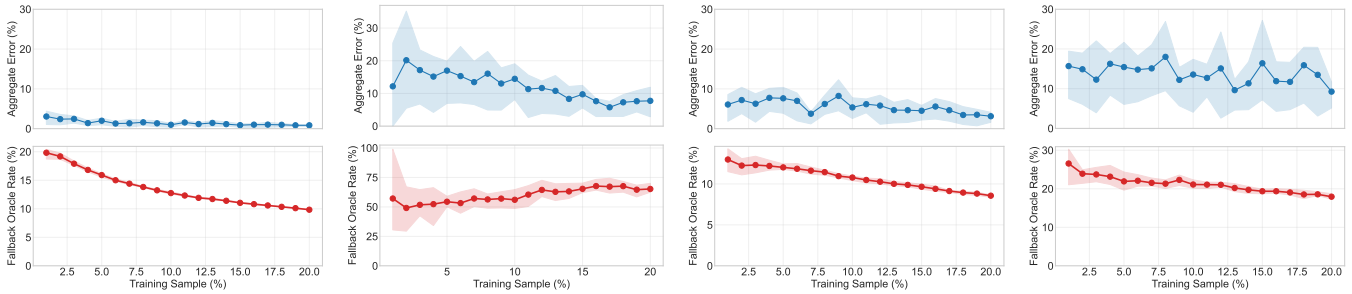
Quality is accuracy (%) for Math and Code, and mean RM score for General.

Table 4: CPU feature-extraction time for 128 responses.

Method	Min (s)	Median (s)	Max (s)
PM	0.009	0.016	0.052
MiniLM	1.498	2.293	2.814
Nomic	16.391	25.732	84.770



(a) Q7 (AVG, p=1.4%) (b) Q13 (AVG, p=1.0%) (c) Q50 (COUNT, p=26.3%) (d) Q52 (SUM, p=1.7%)



(e) Q53 (SUM, p=4.1%) (f) Q54 (COUNT, p=6.1%) (g) Q55 (COUNT, p=1.7%) (h) Q56 (SUM, p=4.5%)

Figure 7: Strategy PM aggregate error vs. training sample percentage under the natural label distribution. Each configuration is evaluated multiple times, and shaded regions show the 25th to 75th percentiles. The plotted fallback rate counts confirmation and audit calls, excluding training. It does not indicate mandatory full evaluation.

Compared methods. Full RM scores all 32 candidates and returns the highest-scoring response. **Fixed-budget sampling** scores a uniform subset and returns its best response. **PM** applies the gate and filter in Section 4.2.

As mentioned earlier in Section 3.3, we compare **Surface**, our PM's decision tree using 23 surface features, with **MiniLM** [30] and **Nomic** [25], which use the same decision tree with embeddings. MiniLM uses a 256-token limit, while Nomic encodes the complete response. All three predict RM scores using the same training data and cascade rules. Table 4 measures CPU feature-extraction time on matched batches of 128 responses using four threads.

Accuracy metrics. For Math and Code tasks, accuracy is defined as exact-match correctness—whether the numerical answer matches the ground truth and whether test cases pass, respectively. For the General task, we use the UltraRM-13B score of the selected response as the quality metric. Comparing with full RM measures how well each method preserves the pipeline's selection objective while reducing RM calls. In Table 3, quality is accuracy (%) for Math and Code and mean RM score for General.

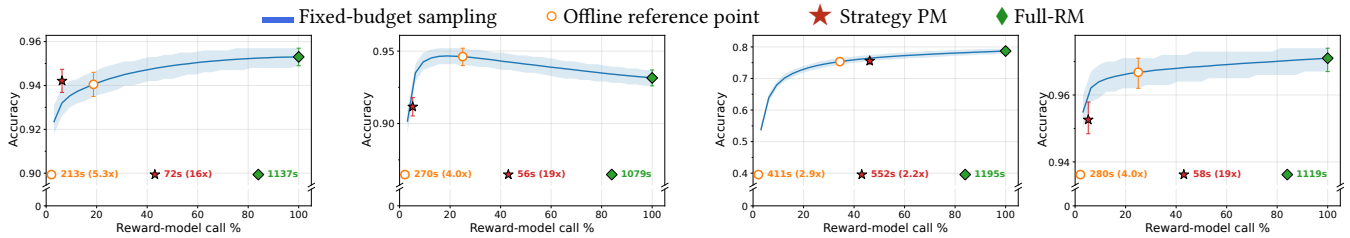
RM calls. The sampling reference evaluates RM@m for predetermined $m = 1, \ldots, 32$. The orange point marks the original offline location where the increase in expected maximum RM score first falls below 0.1. It is not an execution of Equation 6. PM uses the original 5% or 10% prompt-training budgets and the presets in Table 1. Calls include scoring the training candidates and are normalized by full-workload scoring. The times in Figure 8 are estimated RM scoring times.

Results. We report the results in Figure 8. The fixed-budget curves show that scoring a subset of candidates preserves quality close to full RM, and PM further reduces scoring on Math and Code. For LLaMA-3.1-8B Math, PM uses 5.23% of full scoring, including training calls, or 19.1 times fewer calls than full RM. Accuracy on the test prompts is 91.16%, compared with 92.95% for full RM, a gap of 1.79 percentage points.

The General task gives more modest savings, because structural features are less informative for free-form responses. PM uses 69.07% to 84.90% of full scoring across generators, with RM-score gaps of 0.40 to 0.64. Math and Code often have several correct candidates, allowing PM to avoid most scoring calls. General requires more scoring to preserve the quality of the selected response.

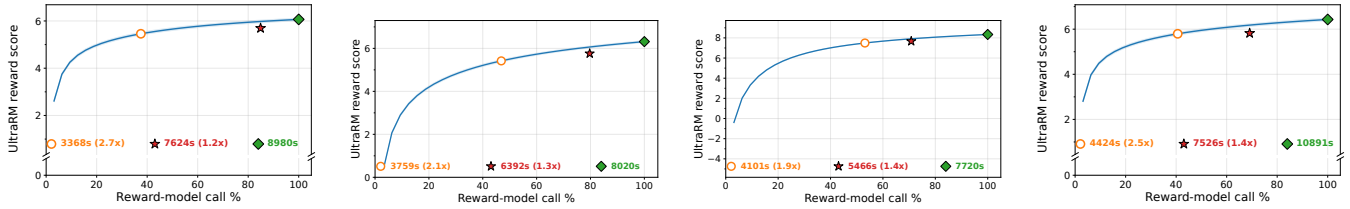
Surface feature versus embeddings. We compare our PM's surface features (Surface) with MiniLM and Nomic embeddings on quality and RM calls (Table 3) and CPU preparation time (Table 4). The comparison tests whether embeddings improve the quality and scoring savings enough to justify their preparation cost.

Quality and RM calls. Table 3 shows that Surface achieves competitive quality and RM-call savings without embeddings. Surface gives higher quality in 11 of 12 cases against each embedding proxy. For Qwen3-32B Code, Surface gives 85.20% accuracy at 5.60% of calls, compared with 84.85% at 13.60% for MiniLM and 83.96% at 22.30% for Nomic. Embeddings sometimes reduce calls further, particularly on General, but with lower mean RM scores. MiniLM improves LLaMA-3.1-8B Math accuracy from 91.16% to 91.68%, while increasing calls from 5.23% to 8.97%. Even the smaller MiniLM encoder therefore offers no consistent improvement over Surface.



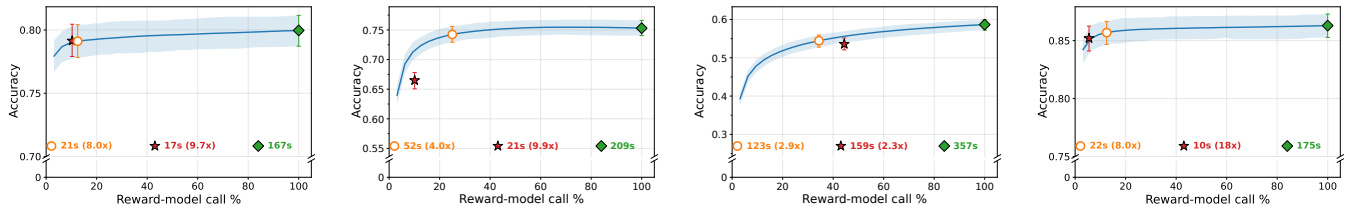
(a) Qwen3-8B (b) LLaMA-3.1-8B (c) Mistral-7B-V0.3 (d) Qwen3-32B

(A) Math Reasoning (GSM8K). Accuracy is measured by exact-match correctness of the numerical answer.



(e) Qwen3-8B (f) LLaMA-3.1-8B (g) Mistral-7B-V0.3 (h) Qwen3-32B

(B) General Instruction Following (UltraFeedback). Quality is measured by the UltraRM-13B reward score of the selected response, as no ground-truth labels exist for this open-ended domain.



(i) Qwen3-8B (j) LLaMA-3.1-8B (k) Mistral-7B-V0.3 (l) Qwen3-32B

(C) Code Generation (HumanEval+, MBPP). Accuracy is measured by pass@1.

Figure 8: Oracle cost and quality on three LLM tasks. Shading and error bars show the 25th to 75th percentiles from prompt resampling with the original proxy. Annotations show estimated RM time and speedup. Sampling, offline, and Full-RM quality use all prompts. PM quality uses test prompts. PM calls include training and are normalized by scoring all prompts.

CPU preparation time. Table 4 shows that Surface's feature extraction takes 0.009 to 0.052 seconds per 128 responses, compared with 1.498 to 2.814 seconds for MiniLM and 16.391 to 84.770 seconds for Nomic. On matched batches, Surface's extraction is 54 to 173 times faster than MiniLM extraction and 1,469 to 2,029 times faster than Nomic extraction. Together, Tables 3 and 4 show that Surface retains competitive quality and RM-call savings at substantially lower preparation cost. A decision tree with inexpensive structural features can therefore exploit the workflow's selection requirement without the cost of generating embeddings.

The complete predictor comparison, fitting-time measurements, variability and response-agreement analyses, and discussion of continued sampling are deferred to our technical report [37].

6 CONCLUSION

We presented a query-centric formulation of AI workflows and showed that two classical database techniques—approximate query processing and proxy-based predicate filtering—directly reduce the number of expensive model invocations without modifying the underlying models or pipelines. Experiments on TPC-DS confirm that the two strategies are complementary: Strategy AQP provides reliable reduction under balanced label distributions, while Strategy PM provides additional gains when structural features support a reliable proxy. PM uses up to 19 times fewer RM calls than full scoring in the evaluated structured tasks, including training. For LLaMA-3.1-8B Math, accuracy is 91.16%, compared with 92.95% for full RM. Shared execution avoids duplicate oracle calls across estimation and proxy training, and returns the plan with fewer remaining requests first. The results demonstrate how query-result requirements make existing database techniques effective for reducing AI workflow cost.

REFERENCES

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *European Conference on Computer Systems*. 29–42. https://doi.org/10.1145/2465351.2465355

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).

[3] Asim Biswal, Liana Patel, Siddarth Jha, Amog Kamsetty, Shu Liu, Joseph E. Gonzalez, Carlos Guestrin, and Matei Zaharia. 2024. Text2SQL is Not Enough: Unifying AI and Databases with TAG. arXiv:2408.14717 [cs.DB] https://arxiv.org/abs/2408.14717

[4] Lingjiao Chen, Matei Zaharia, and James Zou. 2024. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. *Transactions on Machine Learning Research* (2024). https://openreview.net/forum?id=cSimKw5p6R

[5] Zui Chen, Lei Cao, Sam Madden, Tim Kraska, Zeyuan Shang, Ju Fan, Nan Tang, Zihui Gu, Chunwei Liu, and Michael Cafarella. 2023. SEED: Domain-Specific Data Curation With Large Language Models. arXiv:2310.00749. https://arxiv.org/abs/2310.00749

[6] Yeounoh Chung, Rushabh Desai, Jian He, Yu Xiao, Thibaud Hottelier, Yves-Laurent Kom Samo, Pushkar Khadilkar, Xianshun Chen, Sam Idicula, Fatma Özcan, Alon Halevy, and Yannis Papakonstantinou. 2026. 100x Cost & Latency Reduction: Performance Analysis of AI Query Approximation using Lightweight Proxy Models: [Experiments & Analysis]. *Proceedings of the ACM on Management of Data* 4, 3, Article 125 (2026), 23 pages. https://doi.org/10.1145/3802002

[7] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training Verifiers to Solve Math Word Problems. *arXiv preprint arXiv:2110.14168* (2021).

[8] Ganqu Cui, Lifan Yuan, Ning Ding, Guanming Yao, Bingxiang He, Wei Zhu, Yuan Ni, Guotong Xie, Ruobing Xie, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2023. UltraFeedback: Boosting Language Models with Scaled AI Feedback. arXiv:2310.01377 [cs.CL] https://arxiv.org/abs/2310.01377v2

[9] Hanjun Dai, Bethany Yixin Wang, Xingchen Wan, Bo Dai, Sherry Yang, Azade Nova, Pengcheng Yin, Phitchaya Mangpo Phothilimthana, Charles Sutton, and Dale Schuurmans. 2024. UQE: A Query Engine for Unstructured Databases. In *Advances in Neural Information Processing Systems*, Vol. 37. https://proceedings.neurips.cc/paper_files/paper/2024/hash/34b3a40ec9752c1ae48fe85fef8fe8dc-Abstract-Conference.html

[10] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[11] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online aggregation. *SIGMOD Rec.* 26, 2 (June 1997), 171–182. https://doi.org/10.1145/253262.253291

[12] Saehan Jo and Immanuel Trummer. 2024. ThalamusDB: Approximate Query Processing on Multi-Modal Data. *Proceedings of the ACM on Management of Data* 2, 3, Article 186 (2024). https://doi.org/10.1145/3654989

[13] Gaurav Tarlok Kakkar, Jiashen Cao, Aubhro Sengupta, Joy Arulraj, and Hyesoon Kim. 2025. Aero: Adaptive Query Processing of ML Queries. *Proc. ACM Manag. Data* 3, 3, Article 174 (June 2025), 27 pages. https://doi.org/10.1145/3725408

[14] Yusuf Kalayci, Vinod Raman, and Shaddin Dughmi. 2025. Optimal Stopping vs Best-of-N for Inference Time Optimization. arXiv:2510.01394 [cs.LG] https://arxiv.org/abs/2510.01394

[15] Daniel Kang, Peter Bailis, and Matei Zaharia. 2019. BlazeIt: optimizing declarative aggregation and limit queries for neural network-based video analytics. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 533–546. https://doi.org/10.14778/3372716.3372725

[16] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: optimizing neural network queries over video at scale. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1586–1597. https://doi.org/10.14778/3137628.3137664

[17] Daniel Kang, John Guibas, Peter Bailis, Tatsunori Hashimoto, Yi Sun, and Matei Zaharia. 2021. Accelerating approximate aggregation queries with expensive predicates. *Proc. VLDB Endow.* 14, 11 (July 2021), 2341–2354. https://doi.org/10.14778/3476249.3476285

[18] Daniel Kang, John Guibas, Peter D. Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2022. TASTI: Semantic Indexes for Machine Learning-Based Queries over Unstructured Data. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1934–1947. https://doi.org/10.1145/3514221.3517897

[19] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*. 615–629.

[20] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeighami, Aditya G. Parameswaran, and Eugene Wu. 2025. Querying Templatized Document Collections with Large Language Models. In *IEEE International Conference on Data Engineering*. 2422–2435. https://doi.org/10.1109/ICDE65448.2025.00183

[21] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, and Gerardo Vitagliano. 2025. Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing. In *Conference on Innovative Data Systems Research*. https://www.vldb.org/cidrdb/papers/2025/p12-liu.pdf

[22] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://proceedings.neurips.cc/paper_files/paper/2023/hash/43e9d647ccd3e4b7b5baab53f0368686-Abstract.html

[23] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1493–1508. https://doi.org/10.1145/3183713.3183751

[24] Mistral AI. 2024. Mistral-7B-v0.3. https://huggingface.co/mistralai/Mistral-7B-v0.3. Apache 2.0 License.

[25] Nomic AI. 2024. nomic-embed-text-v1.5 Model Card. Hugging Face. https://huggingface.co/nomic-ai/nomic-embed-text-v1.5 Accessed 26 September 2026.

[26] OpenBMB. 2023. UltraRM-13b Model Card. Hugging Face. https://huggingface.co/openbmb/UltraRM-13b Accessed 26 September 2026.

[27] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. VerdictDB: Universalizing Approximate Query Processing. In *Proceedings of the 2018 International Conference on Management of Data*. 1461–1476. https://doi.org/10.1145/3183713.3196905

[28] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2025. Semantic Operators and Their Optimization: Enabling LLM-Based Data Processing with Accuracy Guarantees in LOTUS. *Proc. VLDB Endow.* 18, 11 (July 2025), 4171–4184. https://doi.org/10.14778/3749646.3749685

[29] Matthew Russo, Chunwei Liu, Sivaprasad Sudhir, Gerardo Vitagliano, Michael Cafarella, Tim Kraska, and Samuel Madden. 2026. Abacus: A cost-based optimizer for Semantic Operator Systems. *Proceedings of the VLDB Endowment* 19, 5 (2026), 1060–1073. https://doi.org/10.14778/3796195.3796215

[30] Sentence Transformers. n.d.. all-MiniLM-L6-v2 Model Card. Hugging Face. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2 Accessed 26 September 2026.

[31] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G. Parameswaran, and Eugene Wu. 2025. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *Proc. VLDB Endow.* 18, 9 (May 2025), 3035–3048. https://doi.org/10.14778/3746405.3746426

[32] Shreya Shankar, Sepanta Zeighami, and Aditya Parameswaran. 2026. Task Cascades for Efficient Unstructured Data Processing. *Proc. ACM Manag. Data* 4, 1, Article 88 (April 2026), 26 pages. https://doi.org/10.1145/3786702

[33] Skywork. 2025. Skywork-Reward-V2-Qwen3-8B Model Card. Hugging Face. https://huggingface.co/Skywork/Skywork-Reward-V2-Qwen3-8B Accessed 26 September 2026.

[34] Hanshi Sun, Momin Haider, Ruiqi Zhang, Huitao Yang, Jiahao Qiu, Ming Yin, Mengdi Wang, Peter L. Bartlett, and Andrea Zanette. 2024. Fast best-of-N decoding via speculative rejection. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 1025, 23 pages.

[35] Zhaoze Sun, Chengliang Chai, Qiyan Deng, Kaisen Jin, Xinyu Guo, Han Han, Ye Yuan, Guoren Wang, and Lei Cao. 2025. QUEST: Query Optimization in Unstructured Document Analysis. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4560–4573. https://doi.org/10.14778/3749646.3749713

[36] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL] https://arxiv.org/abs/2307.09288

[37] Huayi Wang, Jun Xu, and Gromit Yeuk-Yin Chan. 2026. Query-Centric Optimization of AI Workflows via Approximate Query Processing and Proxy Models. arXiv:2607.00254 [cs.DB] https://arxiv.org/abs/2607.00254

[38] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[39] Zhihui Yang, Zuozhi Wang, Yicong Huang, Yao Lu, Chen Li, and X. Sean Wang. 2022. Optimizing machine learning inference queries with correlative proxy models. *Proc. VLDB Endow.* 15, 10 (June 2022), 2032–2044. https://doi.org/10.14778/3547305.3547310

[40] Sepanta Zeighami, Shreya Shankar, and Aditya Parameswaran. 2025. Cut Costs, Not Accuracy: LLM-Powered Data Processing with Guarantees. *Proceedings of the ACM on Management of Data* 3, 6 (2025), 1–26. https://doi.org/10.1145/3769776

[41] Tijana Zrnic and Emmanuel J. Candès. 2024. Cross-Prediction-Powered Inference. *Proceedings of the National Academy of Sciences* 121, 15 (2024), e2322083121. https://doi.org/10.1073/pnas.2322083121