OctoSelector: Efficient and Effective Batch-Aware Model Selection for Large Language Models

GUANGXUE ZHANG, University of California, Irvine, USA
YIMING LIN, University of California, Berkeley, USA
SHARAD MEHROTRA, University of California, Irvine, USA

Large Language Models (LLMs) vary significantly in metrics such as accuracy, latency, and cost, making it challenging for users and applications to decide which model to invoke for each query. This paper presents OctoSelector, a framework for LLM selection that satisfies user-defined objectives and constraints across multiple metrics. In the pre-processing phase, OctoSelector learns difficulty-aware representations of queries based on both input and output complexity, clustering them into similar difficulty groups to enable efficient performance estimation across multiple LLMs. During inference, OctoSelector supports LLM selection for batched workload, formulating it as an Integer Linear Programming (ILP) problem that optimizes a user-defined objective (e.g., minimizing cost or latency, or maximizing accuracy) while enforcing constraints on other metrics. We evaluate OctoSelector on two types of tasks: NL2SQL using the Spider and BIRD benchmarks, and sentiment analysis using the IMDb benchmark. When optimizing for cost under accuracy and latency constraints, OctoSelector achieves up to a 67.7% cost reduction on NL2SQL tasks for batched workloads compared to state-of-the-art approaches.

CCS Concepts: • **Information systems** → **Data analytics**; **Query optimization**.

Additional Key Words and Phrases: Large Language Models, Text-to-SQL, Model Selection

ACM Reference Format:
GuangXue Zhang, Yiming Lin, and Sharad Mehrotra. 2026. OctoSelector: Efficient and Effective Batch-Aware Model Selection for Large Language Models. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 218 (June 2026), 26 pages. https://doi.org/10.1145/3802095

1 Introduction

In recent years, large language models (LLMs) have become a core component in a wide range of natural language applications, data-driven tasks, such as data wrangling [11, 13, 14, 32, 47], tabular data treatment [9, 30, 34], and Text-to-SQL [18, 19, 21, 27, 28, 46]. While much research focused on improving model architectures or training pipelines, an equally critical challenge arises: given a pool of advanced LLMs, how can we systematically select the most suitable ones to achieve both cost efficiency and performance effectiveness in applications.

Compared to open-source LLMs, proprietary LLMs such as GPT [4], Gemini [1], and Claude [2] offer several key advantages. These models exhibit strong generalization capabilities and state-of-the-art performance across diverse tasks, largely due to their massive parameter scales and proprietary training pipelines. More importantly, they are easily accessible via API, requiring no infrastructure management, model training, or fine-tuning from the user. By typically just

Authors' Contact Information: GuangXue Zhang, University of California, Irvine, Irvine, USA, gxzhang7@uci.edu; Yiming Lin, University of California, Berkeley, Berkeley, USA, yiminglin@berkeley.edu; Sharad Mehrotra, University of California, Irvine, Irvine, USA, sharad@ics.uci.edu.

ACM 2836-6573/2026/6-ART218
https://doi.org/10.1145/3802095

specifying a model name users can invoke powerful LLMs while offloading concerns such as versioning, scaling, and maintenance.

Despite their advantages, proprietary LLMs exhibit varying performance in terms of accuracy, latency, and pricing, making model selection a non-trivial decision. The primary concern with proprietary LLMs lies in their API usage cost, which can be substantial at scale. For example, OpenAI offers multiple LLM APIs, with GPT-4o reported to perform better on complex tasks than GPT-4o-mini but at 16× the cost [3]. Similarly, Opus, part of Anthropic's Claude 3.5 model achieves higher accuracy on various benchmarks than the lightweight version Haiku but can be 18 to 60 × more expensive [5].

A naive strategy that sends all queries to the most powerful model can lead to excessive costs, while processing easy queries with expensive models wastes budget without meaningful accuracy gain. Conversely, complex queries may fail when routed to lightweight models resulting in inaccurate outputs and wasted latency. As a result, inefficient model selection can lead to rapidly accumulating wasted cost and latency, posing a major bottleneck for scalable applications when processing large volumes of data.

Prior Work on LLM Routing: Recent work has investigated LLM routing strategies to reduce inference costs while maintaining task performance. A straightforward method called HybridLLM [17] classifies queries into two categories—easy and hard—and routes them to small or large LLMs accordingly, based on estimated difficulty. In contrast to the binary difficulty classification used in prior work, Smoothie [23] adopts a clustering-based retrieval approach: it identifies k nearest neighbor queries in an embedding space and routes the input to the LLM that achieved the best performance on these similar queries. Another line of work adopts cascade execution strategies such as FrugalGPT [12] and Automix [7], where a query is first processed by a small and cheap model, and only escalated to a more powerful LLM if the initial response is verified to be unsatisfied in term of accuracy or confidence. Although this reduces the number of expensive model calls, it introduces additional latency and redundant computation, especially when large portions of queries eventually require escalation. A recent strategy is GraphRouter [20] that introduces a set-level routing framework using a graph neural network (GNN), where tasks, queries, and LLMs are represented as nodes. Edges between queries and LLMs encode performance and cost.

Compared Aspect	HybridLLM	FrugalGPT	GraphRouter	Automix	Smoothie	RouteLLM	OctoSel.(our)
Support batch-level routing	-	-	-	-	-	-	✓
Input–output difficulty–aware feature encoding	-	-	-	-	-	-	✓
Clustering-based routing strategy	-	-	-	-	✓	-	✓
Objective optimization (e.g., cost-efficient)	✓	✓	✓	✓	-	✓	✓
Support user-specified constraints	-	-	-	-	-	-	✓
Cross-model routing across multiple LLMs	-	✓	✓	✓	✓	-	✓
Single-stage routing architecture(vs. cascade)	✓	-	✓	-	✓	✓	✓

Table 1. Comparison between OctoSelector and representative LLM routing systems. ✓indicates that the feature is supported, while - denotes absence.

Our goal in this paper is to develop an LLM routing strategy for situations when LLM invocations are used as inference mechanisms within data processing systems (e.g., ThalamusDB [25] or LOTUS [41]) or in data science applications where such inferences must be performed over a large batch of tasks. We further wish to design routing strategies that empower users to explore tradeoffs across diverse metrics – cost, accuracy, and latency – inherent in different LLMs. In particular, the routing strategy must allow specification of constraints over one or more metrics, and choose LLMs such that those constraints are met in aggregate over the entire workload, while optimizing other metrics.

Existing approaches (with the exception of GraphRouter) operate at the individual query level, selecting an LLM per query in isolation without considering global routing efficiency across the entire query set. GraphRouter, while it enables global optimization over a single objective such as minimizing cost, it lacks support for multi-objective routing with constraints, such as jointly optimizing for cost while satisfying accuracy or latency requirements. Furthermore, while GraphRouter makes decisions at the global (set) level, the routing mechanisms uses a single query for each LLM invocation which ignores the potential benefit of batching multiple queries into one prompt, in terms of overall cost and latency reduction. Table 1 summarizes a comparative analysis between OctoSelector- the routing mechanism proposed in this paper with representative LLM routing frameworks across different dimensions. Another approach is to fine-tune a small language model (LM) with labeled data, which may be effective for simple tasks such as classification, even with non-trivial labeling effort. Such models typically perform much worse on more complex tasks than advanced LLMs [43]. OctoSelector is instead designed to support LLM-powered data processing tasks, ***independent of task complexity and model choice.***

Our Approach – OctoSelector: We propose OctoSelector, a general framework for cost-efficient and performance-aware LLM model selection. OctoSelector addresses several challenges of resolving LLM router under cost-performance trade-offs. First, it supports an effective feature representation of input queries that captures their relative difficulty with respect to a specific task domain. Without such a representation, it would be difficult to distinguish which queries can be handled by lightweight models and which require more capable LLMs. Second, it supports effective ways to determine how well different LLMs perform for queries of varying levels of difficulty across diverse metrics. Third, it supports ways to batch queries into single prompts and to assign batches to models so as to jointly optimize for performance, cost, and latency across the entire workload. Unlike heuristic-based routing, this approach allows for globally coordinated decisions that can leverage batching opportunities and model capabilities more effectively.

OctoSelector operates in two phases: the *pre-processing* and the *inference* phases. In the *pre-processing* phase, OctoSelector learns a feature extraction model from training dataset of natural language queries, which produces vector representations that encode the relative difficulty of each query within a given task domain. In OctoSelector, difficulty is treated as an abstract concept rather than an actual class label. We identify the source of difficulty by task type: for code generation and text generation, the difficulty often arises from both the input prompt (including query and task instruction) and the expected response as output, which may be long or structurally complex. In contrast, for task domain like sentiment analysis, where outputs are typically short (e.g., a label or scalar value), difficulty is largely derived from the semantic or linguistic ambiguity of the input query itself. Depending on the task type, the feature extraction model is trained on the input query alone or jointly with the expected output, capturing the aspects of difficulty that are most relevant to the task. Using these difficulty-aware features, we cluster the queries in the training set with similar difficulty. We operate under the assumption that LLMs tend to exhibit similar performance across queries of similar difficulty. This enables us to approximate query-level performance (e.g., cost, latency, accuracy) using cluster-level statistics, reducing the need for exhaustive model-query evaluation. The resulting performance estimates are stored in a lookup table, which will be referenced during inference.

In the *inference* phase, given a set of incoming (development) queries, OctoSelector formulates the query-to-model assignment as an integer linear programming (ILP) problem, jointly optimizing for cost, latency, and accuracy under user-defined constraints. Unlike prior work that routes queries individually, OctoSelector employs a batching strategy: multiple queries that share same context and instructions are grouped into a single prompt, enabling significant cost and latency savings while preserving task feasibility. Furthermore, it supports streaming inferences when queries arrive sequentially and LLM assignment decision must be made at query arrival time. In this setting, we adopt a local optimization strategy based on linear programming, leveraging cluster-level performance estimates to make real-time decisions.

Contributions: This paper's contributions are summarized as follows. We 1) formalize the model selection problem in a general form (Sec. 2); 2) develop a difficulty-aware query feature representation, leveraging unsupervised clustering to approximate query-level performance metrics using cluster-level statistics, which enables cost-efficient and scalable estimation (Sec. 3); 3) develop an integer linear programming based optimization strategy of model selection for batched and streaming settings (Sec. 4); 4) Evaluate OctoSelector for two case studies – NL2SQL tasks and sentiment analysis (Sec. 5), and conduct comprehensive experiments to evaluate OctoSelector in these domains, demonstrating up to 70% cost reduction while maintaining or improving task feasibility compared to single-LLM execution baselines.

2 Problem Setting

Let L be an LLM model that takes a natural language question q as input and returns a natural language response r as output. We denote one invocation of the LLM L on a prompt q as $L(q) = r$. Considering a workload $Q = \{q_1, q_2, \ldots, q_n\}$, a set of candidate LLMs $\mathcal{L} = \{L_1, L_2, \ldots, L_m\}$, we aim to design a good plan that assigns one LLM $L_i \in \mathcal{L}$ per question $q_j \in Q$ satisfying user-defined constraints while optimizing user-defined objectives.

We first introduce three metrics used to define the constraints or objectives, i.e., cost, latency, and accuracy of an LLM invocation, denoted by $C(q, L)$, $T(q, L)$, and $A(q, L)$, respectively.

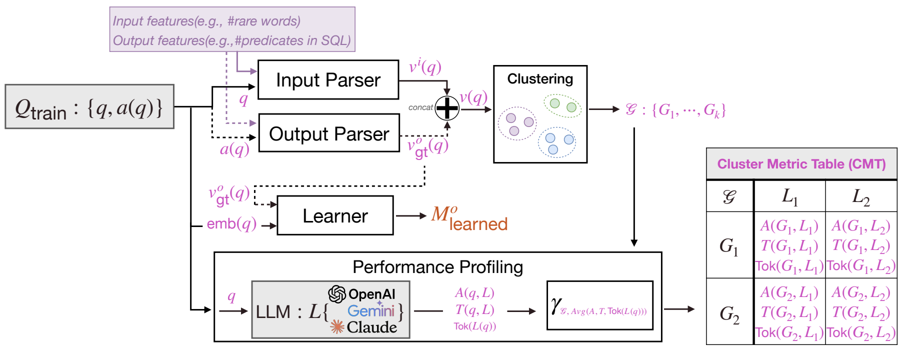
Cost $C(q, L)$ per LLM invocation on question q using LLM L is determined by the number of tokens in the input q and the response $L(q)$. In particular, $C(q, L) = L.IC * \text{Tok}(q) + L.OC * \text{Tok}(L(q))$, where $\text{Tok}(\cdot)$ denotes the number of tokens, and $L.IC$ and $L.OC$ denote the unit cost for input and output tokens, respectively. For example, IC is \$0.15 and OC is \$0.60 per one million tokens for gpt-4o-mini [3]. Latency $T(q, L)$ is defined as the time between sending a prompt and receiving the response from LLM server, consisting of network communication latency and the model's internal processing time. The network communication latency refers to a fixed time incurred by transmitting a request to and receiving a response from the LLM server. Accuracy $A(q, L)$ per LLM invocation is defined as a boolean function that returns True if the response $L(q)$ matches the ground-truth answer of q, and False otherwise. Based on the metrics defined for a single LLM invocation, it is straightforward to extend them to the query workload Q, where cost, latency, and accuracy are defined as the average values over all invocations on Q. Let $C(Q, \mathcal{L})$, $T(Q, \mathcal{L})$, and $A(Q, \mathcal{L})$ denote the average cost, latency, and accuracy over the workload Q paired with LLMs from $\mathcal{L}$, where $C(Q, \mathcal{L}) = \frac{\sum_{q \in Q} C(q, f(q))}{|Q|}$, and $f(q)$ denotes the LLM $L \in \mathcal{L}$ assigned to q by OctoSelector. (The definitions for the other metrics are analogous and are omitted here.)

2.1 Problem Formulation

Given a query workload $Q = \{q_1, q_2, \ldots, q_n\}$, a set of LLMs $\mathcal{L} = \{L_1, L_2, \ldots, L_m\}$, and predefined metrics $C(Q, \mathcal{L})$, $T(Q, \mathcal{L})$, and
$A(Q, \mathcal{L})$, we present the high-level problem formulation by first discussing the possible combinations of constraints and objectives based on these three metrics, and then by relaxing the LLM assignment problem under both batching and query at a time scenarios.

Space of Constraints and Objectives. We describe below three cases supported in OctoSelector, based on the number of constraints and objectives.

(1) Single objective. Given a workload Q and LLMs $\mathcal{L}$, users may specify an objective from one of the metrics $C(Q, \mathcal{L})$, $T(Q, \mathcal{L})$, or $A(Q, \mathcal{L})$, and optionally place the remaining (partial) metrics



$\mathcal{G}$	L_1	L_2
G_1	$A(G_1, L_1)$ $T(G_1, L_1)$ Tok(G_1, L_1)	$A(G_1, L_2)$ $T(G_1, L_2)$ Tok(G_1, L_2)
G_2	$A(G_2, L_1)$ $T(G_2, L_1)$ Tok(G_2, L_1)	$A(G_2, L_2)$ $T(G_2, L_2)$ Tok(G_2, L_2)

Fig. 1. Pre-Processing Component Diagram

as constraints. For example, one may maximize accuracy $A(Q, \mathcal{L})$ while constraining cost $C(Q, \mathcal{L})$ and/or latency $T(Q, \mathcal{L})$ under user-defined thresholds (e.g., $C(Q, \mathcal{L}) \leq \sigma_C$).

(2) Multi-objectives. Users are also allowed to set more than one objective with optional constraints. For example, one can simultaneously maximize accuracy $A(Q, \mathcal{L})$ and minimize cost $C(Q, \mathcal{L})$ while constraining latency $T(Q, \mathcal{L})$ under thresholds (e.g., $T(Q, \mathcal{L}) \leq \sigma_T$).

(3) Zero-objective. Users may also specify only constraints without an explicit objective. For example, one may require staying within a cost budget and ensuring a minimum acceptable accuracy without caring about latency. In this case, any feasible solution suffices, and OctoSelector returns one from the pareto solution set, i.e., one that dominates the others, if such a solution exists.

While we focus on describing solutions for the single-objective case, which is most commonly used in practice, we also discuss our approaches to support other cases in Section 4.

Batching Strategy. For a given set of queries, we refer to *batching* as a strategy that grouping a subset of queries into a single prompt for LLM invocation. The batching strategy follows the policy below: queries within one prompt share context and instruction. For example, two questions in NL2SQL task from the same database can potentially share the same schema description. As a result, executing prompt with multiple queries reduces cost and latency, compared with same queries but each one in single prompt. We formulate this batching strategy in Section 4.

We evaluate OctoSelector through two case studies: NL2SQL, which translates natural language into SQL, and sentiment analysis, which classifies text sentiment. Evaluation results from both are presented in Section 5.

3 OctoSelector: Pre-Processing

Let the training data set at the pre-processing state consist of a set of natural language queries $Q_{\text{train}} = \{q_1, \cdots, q_n\}$ and let their ground-truth answers be $A_{\text{train}} = \{a(q_1), \cdots, a(q_n)\}$), where $a(q_i)$ refers to the (ground-truth) output of q_i. Note that human labels (i.e., ground-truth outputs) during training are not mandatory for OctoSelector, as they can be generated using oracle LLMs (e.g., voting from a few advanced LLMs). This one-time labeling cost can be easily amortized over many LLM inferences, the number of which typically far exceeds the number of labels in realistic deployments. [1]

[1]While OctoSelector assumes the availability of ground truth for model evaluation, this is not strictly necessary. In cases where ground truth is unavailable, a ensemble techniques can be employed on a sample to estimate each model's performance. Such an approach has been extensively studied in ML literature, with solutions based on EM-based latent truth discovery [16]. In this work we maintain the simplifying assumption about availability of ground truth since our contributions – optimizing the cost/quality/latency tradeoffs of a workload – do not rely on the mechanism used to calibrate/profile models.

The pre-processing pipeline of OctoSelector transforms query/answer pairs in such an input and output training data into a a numerical features that capture the level of difficulty of the queries. Based on such a feature representation, OctoSelector clusters the queries creating groups $\mathcal{G} : \{G_1, \cdots, G_k\}$ of queries that have a similar levels of difficulty. The queries in a group are executed over the various target LLMs to gather statistics about their performance resulting in a profile of the LLM based on the training data. These performance statistics are then summarized into a Cluster Metric Table (CMT) which will be used during inference time in assigning optimal LLMs to execute for each query. We describe the different components of the pre-processing pipeline in more detail below.

3.1 From Queries to Clusters

The first step of pre-processing is to extract and represent queries based on difficulty-related features. The difficulty-related features for a query can be derived from input for tasks such as classification, or from both input and output, for generative tasks such as code generation. Features capturing query difficulty, in general, are domain and task specific, though OctoSelector supports a set of default features which can be used. A set of such default features listed in Table 2 correspond to linguistic complexity indicators [22, 39], which we used for both NL2SQL and sentiment analysis tasks, that capture the syntactic and semantic difficulty of the natural language query. OctoSelector supports parse rules to extract such default features from input queries. Such features extracted from a query q are referred to as input-based features of q denoted as $v^i(q)$, short as v^i. Features derived from the output, $a(q)$ to a query q, used in OctoSelector are denoted as v^o.

Note that since the $a(q)$ is available in training data, during pre-processing, the output features v^o can be derived from the corresponding outputs. Such features would not be available during the inference phase, and, thus, as discussed in the following subsection, we will learn an additional model to estimate the output-based feature values from the input query q.

Output-based features are task-dependent and capture the domain-related difficulty revealed from the answer to the query. For the NL2SQL task, the output-based features we used in OctoSelector are shown in Table 3 which are based on the complexity of SQL constructs used in the SQL output.

Although OctoSelector provides a set of default input-based difficulty features (which can be used broadly across diverse tasks), a set of output-based features (intended for capturing the difficulty of translating a specific NL query into SQL), such features are customizable and changeable based on the specificity of the task.

The final feature representation of query q is formed by concatenating the input and output vectors: $v(q) = [v^i; v^o]$; or $v(q) = [v^i]$ alone when output features are not used (e.g., in sentiment analysis).

Once the feature representation $v(q)$ for a query q has been determined, the second step of pre-processing applies unsupervised learning to cluster the feature vectors $v(q)$. This clustering process is motivated by the observation that for a given task, LLM performance varies significantly across queries of different difficulty levels. In OctoSelector, "difficulty" is an abstract and task-dependent notion rather than a direct class label. Two queries have similar difficulty if 1) they share similar features representation as discussed above, and 2) they exhibit similar metrics profiles (accuracy, latency, and cost) across a given set of LLMs. Since we cannot observe a query's difficulty directly, by clustering queries in the feature space, we naturally partition them into groups $\mathcal{G} : \{G_1, ..., G_k\}$ of queries of similar level of difficulty. These clusters then enable cluster-level performance estimation (through the Cluster Metric Table (CMT) in Section 3.3 for different LLMs, which OctoSelector uses to optimize model selection under user constraints. OctoSelector employs K-Means clustering [36, 37] as its default algorithm. We determine the optimal number of clusters k through the elbow method [15]. The chosen k corresponds to the elbow point where increasing the number of clusters results in only a small improvement in clustering quality, as measured by the reduction in intra-cluster variance.

Table 2. Linguistic Features and corresponding definitions

Linguistic Features	**Definition (Scale)**
Rare Word Proportion	Proportion of low-frequency words in the input text.
Readability Score	Flesch-Kincaid Grade [44] Level estimating reading difficulty (higher = harder).
Average Sentence Length	For multi-sentence inputs, this is defined as the mean number of tokens per sentence. For single-sentence inputs, it reduces to the total number of words in the input.
Parse Tree Depth	Maximum depth of the syntactic parse tree [38] (captures structural complexity).
Subordinate Clause Ratio	Ratio of subordinate clauses to total sentences (e.g., "although", "because").
Part of Speech(POS) ratios	Proportion of each POS tag (adjective, adverb, verb, noun) relative to the total number of words in the input.

3.2 Output Model Learner

The Learner is involved for the task such as NL2SQL where both input and output features are used for query difficulty feature representation. During pre-processing, input and output (ground truth answers of the query) are both available in the training dataset. We could, thus, extract difficulty-related features from both input and output via parsers, and learn clusters over such a feature representation. During inference, however, the development data only contains query input, and therefore the parser is no longer applicable to generate the output features, i.e. output features are unobservable. Thus, we need a mechanism to predict the output feature values from the input query at inference time. OctoSelector supports a model learning step (referred to as *Learner* in Figure 1) that learns a model M^o_{learned}, which is used to determine the output feature values based on input query during inference time.

The Learner is a supervised model. For NL2SQL task, OctoSelector adapts a Random Forest (RF) model as the Learner. To train the Learner (RF model), the input is the embedded queries $\text{emb}(q)$, and the supervision is the output-based feature vector parsed from the ground truth SQL in Q_{train}, denoted as v^o_{gt} in Figure 1. Since the RF model requires numerical inputs, we apply a word-embedding model to each natural language query q, yielding $\text{emb}(q)$. Notice that the Learner takes $\text{emb}(q)$ as input, which differs from the input-based difficulty-related features $v^i(q)$. This is because the Learner is designed to predict the output feature representation from the semantic content of the query, rather than from the difficulty features $v^i(q)$ used for clustering. In OctoSelector, we uses the TF-IDF [42] as the word embedding model, which represents each natural language query as a high-dimension numerical vector where each dimension corresponds to a vocabulary term weighted by its importance.

Table 3. SQL Syntax Features and corresponding definitions

Feature Group	**Feature Name**	**Definition**
Clause Complexity	Num Predicates	Number of predicates in WHERE clause.
	Num Nested Queries	Number of nested subqueries in the SQL statement.
	Num GROUP BY	Number of GROUP BY clauses.
	Num HAVING	Number of HAVING constraints used.
	Has ORDER BY	Binary indicator for presence of ORDER BY clause.
	Has LIMIT	Binary indicator for presence of LIMIT constraint.
	Has DISTINCT	Binary indicator for use of DISTINCT keyword.
	Has LIKE	Binary indicator for use of LIKE operator.
SQL Concept Richness	Num SQL Concepts	Total number of unique SQL syntax used in the query.
	Num Joins	Number of join operations (e.g., INNER JOIN, LEFT JOIN).
	Num Aggregations	Number of aggregation functions (e.g., SUM, AVG).
	Num Logical Ops	Number of logical operators (e.g., AND, OR, NOT).
Schema Diversity	Num Distinct Attrs	Number of distinct attributes accessed in the query.

3.3 Profiling and Cluster Metric Table (CMT)

LLM Performance Profiling. For each natural language query q in the training data Q_{train}, we profile all candidate LLMs by measuring each invocation's performance in terms of accuracy $A(q, L)$, latency $T(q, L)$, and query's response $L(q)$ token length, noted as $\text{Tok}(L(q))$. We record the query's response $L(q)$ token length instead of cost is because that following Section 2, the cost is calculated from a query's input token length (i.e. prompt length) and response token length. Since the prompt length can be counted directly, we only record the output token length for later cost calculation.

The prompt that we used for LLM invocation and metric computations are task-dependent. In general, prompt should include task instruction, necessary context, and query (question). For instance, in NL2SQL, the instruction might be "convert this natural language query to SQL", and the context is query associated table schema. For sentiment analysis in movie reviews, the context we used is external information, such as the movie storyline description from IMDb. The latency and cost metrics definitions are as defined in Section 2, while the accuracy is task-dependent or decided by users.

Specifically, OctoSelector supports two tasks, NL2SQL and sentiment analysis. For NL2SQL, the training set Q_{train} contains natural language queries as input and each corresponding SQL query as ground truth labels. We adopt *Execution Accuracy (EX)* [35, 45] as the evaluation metric for NL2SQL task. This metric measures whether the execution result of the SQL query generated by the LLM $L_j(q)$ is identical to that of the ground truth SQL query $R_{gt}(q)$ on the associated database. Formally, $A(q, L_j) = EX(L_j(q), R_{gt}(q))$, where $EX(\cdot)$ is an indicator function returning 1 if the two execution results are identical, and 0 otherwise.

For sentiment analysis, the class label is a discrete numerical values (e.g., a rating score), and we adopt mean-squared-error (MSE) as the accuracy metric, to measure the distance between the ground truth class label and the model's prediction. When the class label is categorical (e.g., positive/negative sentiment), we instead measure accuracy by the proportion of exact matches between predicted and ground-truth labels.

CMT Construction. We construct Cluster Metric Table (CMT) to enable efficient query-level performance estimation via cluster-level approximation. This approach leverages the similar difficulty property of clusters in $\mathcal{G}$: queries within each cluster exhibit similar LLM performance characteristics.

Given the performance profile of training queries $q_i \in Q_{train}$ on each LLM L_j (i.e., accuracy, latency, and output token length), we define **cluster-level metrics**. For example, the cluster-level accuracy of L_j on cluster G_k is

$$A(G_k, L_j) = \frac{1}{|G_k|} \sum_{q \in G_k} A(q, L_j),$$

where $|G_k|$ is the number of queries in cluster G_k. Analogously, we define cluster-level latency and cluster-level output token length.

During inference, the CMT serves as a lookup table and we estimate the expected accuracy, latency and output token length of a query q by locating the cluster it belongs to based on its feature vector $v(q)$. Specifically, we estimate the accuracy of q on an LLM L_j by taking the cluster-level accuracy of L_j on the cluster G_i that q belongs to:

$$A(q, L_j) \approx A(G_i, L_j), \quad \text{where } v(q) \in G_i,\ G_i \in \mathcal{G},\ j \in \{1, \dots, m\}.$$

The same approximation applies to latency and output token length.

The CMT stores the aggregated performance result from profiling for all $G_i \in \mathcal{G}$ and $L_j \in L$, allowing efficient lookup at inference time.

CMT Source of Bias. The CMT serves as LLM performances estimation table at inference time, thus its quality can affect the OctoSelector's assignment plan of development data. The main sources of bias in the CMT arise from: (1) distribution shift between the training data used to compute CMT entries and the inference-time queries mapped to each cluster, as well as small clusters that can yield noisy or skewed averages; (2) cluster assignment errors caused from the quality of the clustering model that may map queries to the wrong cluster. It biases the cluster-level performance aggregation used to construct the CMT; if it occurs at inference time it biases the performance estimation obtained from the CMT lookup; (3) performance profiling and measurement noise when computing performance on the training set—accuracy may be biased by evaluation artifacts and model output variability (including hallucinations), while latency can fluctuate due to server-side conditions.

4 OctoSelector: Inference with Batching Strategy

The inference process in OctoSelector (Figure 2) takes as input a development dataset of natural language queries (Q_{dev}) and user-specified optimization objectives (e.g., minimizing cost) and

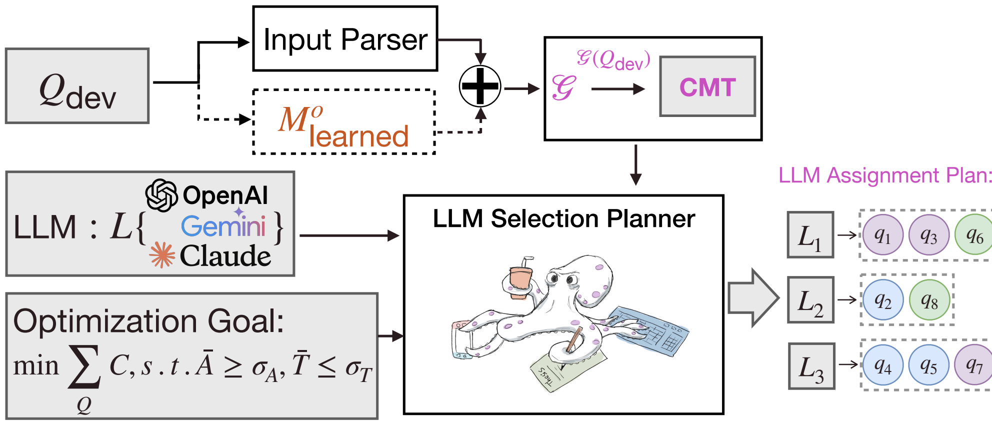


Fig. 2. Inferencing in OctoSelector

constraints (e.g., average accuracy per query ≥ X, average latency per query ≤ Y second) to generate an LLM assignment plan that allocates queries to different LLMs.

In this phase, each query $q \in Q_{\text{dev}}$ is transformed into a feature vector using the input and output parsers (Figure 1). Query q is then assigned to its respective cluster based on its feature representation $v(q)$. The inference task is, thus, transformed into a set of group of queries, with each group corresponding to one cluster. The selection planner takes such an input and uses the cluster metric table (Figure 1), learned during pre-processing, to assign queries to LLMs. It postulates the LLM assignment task as an an Integer Linear Programming (ILP) problem to satisfy the user-defined constraints. The exact formulation of the problem is described in Section 4.4. Grouping queries with shared context into a single prompt reduces both cost and latency. For instance, in the NL2SQL task, queries from the same database reuse identical schema descriptions; batching them ensures the schema is transmitted only once, thereby lowering token cost and time.

We first define the notion of a batch and the procedure for batch creation (Section 4.2). We then describe how to compute batch-level metrics—accuracy, latency, and cost (Section 4.3). Finally, we present our LLM assignment problem formulation and solution approach (Section 4.4).

4.1 Applicable Applications for Batching

Multiple applications can benefit from batching strategy and we provide two concrete applications below. First, movie reviews sentiment analysis where user should send a set of reviews of one given movie to LLM and ask for each review's sentiment. Consider the movie introduction as helpful context, instead of sending one review in a prompt with each context, OctoSelector allows to send a batch of queries together with one copy of context in a single prompt, saving overall cost and latency. Second, for NL2SQL application with batching strategy gets benefit when many NL queries target the same underlying database. This occurs in analytic workloads over semantic-operator databases (e.g., LOTUS, ThalamusDB), where analysts process large complaint tables whose entries contain free-text descriptions. Each complaint (e.g., "battery swelling", "device overheating", "screen flickering") is converted into an NL2SQL task that queries the same Products and Reviews tables, and the resulting SQL outputs are then aggregated for downstream analytics. Similar high-volume batching occurs in multi-user applications such as e-commerce ("Where is my order?") and ride-sharing ("Where is my driver?"). Notice that batching is performed within backend with each request retaining its authenticated context, so no cross-user data is exposed. OctoSelector operates as a routing middle layer and only optimizes model assignment for cost-quality tradeoffs.

4.2 Batch Definition and Creation

For each invocation to LLM, the expected prompts are assumed to contain a set of instructions and context, referred to as *preamble*, $D : \{D_1, \cdots, D_M\}$, which should associate with one or multiple queries in the dataset. For example, in the NL2SQL task, the preamble consists of the task instruction and the database schema.

Batch Definition. A batch b is defined as one or a group of queries that share the same preamble, and can thus be jointly processed within a single LLM invocation. In OctoSelector, each batch b contains at most n_s queries.

While each LLM enforces a maximum prompt length, these limits are typically very large (e.g., 128k tokens for GPT-4o at Tier 1 [4]). For short-query workloads such as NL2SQL, this constraint rarely limits the choice of n_s. However, in tasks with longer queries or more complex preambles, the prompt length may still impose a non-trivial bound on n_s. In practice, the effective upper bound on n_s is often determined empirically: batching too many queries together can cause semantic interference and degraded performance well before the raw prompt-length limit is reached. Similar effects have been reported in prior studies on long-context usage and prompt packing, which show that quality loss arises primarily from information dilution and cross-query interference rather than exceeding context length [12, 33].

Batch Creation. We consider the following inputs for batch creation: a set of natural language queries for inference: $Q_{\text{dev}} : \{q_1, \cdots, q_n\}$, their assicoated preambles, $D : \{D_1, \cdots, D_M\}$, LLMs $L : \{L_1, \cdots, L_m\}$, similar difficulty clusters $G : \{G_1, \cdots, G_k\}$, and the Cluster Metric Table (CMT), which summarizes the cluster-level performance of different LLMs. Based on these inputs, we now describe the batch creation procedure.

Let $Q_{(D_i,G_j)}$ be the queries that share a preamble D_i and are grouped in cluster G_j. Inside preamble D_i, for every cluster G_j we slice its queries into blocks of exactly n_s queries each (last block, refer to as *left over*, may have $< n_s$ queries). This process creates

$$\lceil \frac{|Q_{(D_i,G_j)}|}{n_s} \rceil + \textbf{left over}_{ij}$$

number of batches for the given preamble D_i and cluster G_j, where the $|\cdot|$ denotes the numbers of queries. We refer to a batch, b, created through the above described process that contains n_s queries, i.e., $|b| = n_s$, a as *full pure batch.*

Dealing with Under-filled batches. Both *left-over* batches and clusters containing fewer than n_s queries will produce under-filled batches. We consider two strategies for handling them:

(1) *Retain as-is:* Each under-filled batch is kept without merging. This may result in up to $k \times M$ under-filled batches, where k is the number of clusters and M is the number of preambles.
(2) *Merge:* Inside the same preamble, greedily merge under-filled batches from different clusters to form filled batches of size n_s. This yields *mixed batches*, as opposed to *pure batches* formed without merging. While this strategy reduces the total number of batches compared to (1), it may still result in up to M under-filled batches in the worst case.

In order to minimize the total number of batches, we choose the merge strategy as a default. In general, two types of batches may arise in OctoSelector: *full pure batches* that contain n_s number of queries belonging to a given cluster, and *residual batches*, which may either be mixed batches or under-filled pure batches. We next describe how performance and cost metric described earlier for queries can be extended to corresponding concepts for batches. These metrics will be used in formulating the LLM selection challenge as an ILP problem (Section 4.4).

4.3 Batch based Metrics

In this section we describe how accuracy, latency, and cost can be computed for different LLMs based on the corresponding metrics defined for queries for both *full pure batches* and *residual batches*.

Batch Accuracy. Batch accuracy measures the average accuracy of individual queries within a batch. Estimating batch accuracy can be challenging due to two reasons: First, LLM performance is affected by the hallucination problem [24], making it challenging to determine whether accuracy variations stem from batching or model instability. Second, while query batching can provide beneficial cross-query context, it also risks attention dilution [8, 26]. Given these uncertainties which are difficult to model, we adopt a simplified approach that treats each query as independent and compute batch accuracy as the average accuracy of its constituent queries. Given a batch b and an LLM L_j, the batch accuracy can be computed as follows:

$$A(b, L_j) = \frac{1}{|b|} \sum_{i=1}^{k} n_i A(G_i, L_j) \tag{1}$$

where n_i is the number of queries of cluster G_i in the batch, and the set of clusters are $G_1, G_2, \ldots, G_k$.

Our experimental results will show that while LLM behavior could deviate from the idealistic setting of queries being considered as independent, the mapping of batches to LLMs OctoSelector achieves (while modeling batch accuracy as the average accuracy of independent queries) brings significant cost-savings while ensuring desired accuracy levels.

Batch Latency. The Cluster Metric Table (CMT) only estimates the latency for single-query prompt, we now build a latency model for a prompt with batched queries. From Section 2 we claim that the latency arises from network and LLM processing time. Since the internal of each part is opaque to the client, OctoSelector adopts the parameters which capture the aggregate latency occurred on specific LLM. Thus, the latency is LLM-dependent, differing from their unique server location, API read/write delay, and model processing capacity. Therefore, we model latency as the sum of three components: a coefficient for input size, a coefficient for output size, and a constant model-specific overhead. The main saving from latency perspective between single-query prompt and batched-query prompt is that the input length is reduced and thus the average latency per query is lower. The detailed latency computation and the empirical result for batch latency parameters of each LLM are provided in Appendix A.1 in [6].

Let Preamble(b) denote the shared preamble of batch b (e.g., NL2SQL instructions + database schema), and let $n_i(b)$ be the number of queries in b that belong to cluster G_i, note that $n_i(b) \le n_s$. Define $\mu_j^{\text{in}}(G_i)$ as the *per-query* average input-token count (*excluding the preamble*) for cluster G_i, and $\mu_j^{\text{out}}(G_i)$ as the *per-query* average output-token count for model L_j on cluster G_i (both estimated offline in CMT), where by previous notation in Section 2 $\mu_j^{\text{out}}(G_i) = \text{Tok}(L_j(q \in G_i))$.

For batch b executed on L_j, we use the following unified latency model:

$$\begin{aligned} T(b, L_j) = \theta_j^{\text{in}} \Big[\text{Tok}(\text{Preamble}(b)) + \sum_{i=1}^{K} n_i(b)\, \mu_j^{\text{in}}(G_i) \Big] \\ + \theta_j^{\text{out}} \sum_{i=1}^{K} n_i(b)\, \mu_j^{\text{out}}(G_i) + \tau_j \end{aligned} \tag{2}$$

Here, $\mathrm{Tok}(\cdot)$ is the model-specific tokenizer; $\mathrm{Tok}(\mathrm{Preamble}(b))$ can be computed exactly once for each preamble and cached; $(\theta_j^{\mathrm{in}}, \theta_j^{\mathrm{out}})$ are the input/output per-token coefficients, and τ_j is a LLM-dependent coefficient.

We empirically determine the coefficient value $\theta_j^{\mathrm{in}}, \theta_j^{\mathrm{out}}$ and τ_j in the *Performance Profiling* module, where we measure the latency across varying batch sizes estimate the parameters via linear regression.

Batch Cost. The batching strategy reduces cost by allowing multiple queries to share same instruction and context within a single prompt. For a batch of size $|b|$, this avoids repeating the shared preamble $|b| - 1$ times compared to single-query execution.

The cost for processing batch b on LLM L_j is computed as:

$$C(b, L_j) = L_j.IC \left[\mathrm{Tok}(\mathrm{Preamble}(b)) + \sum_{i=1}^{K} n_i(b)\, \mu_j^{\mathrm{in}}(G_i)\right] + L_j.OC \sum_{i=1}^{K} n_i(b)\, \mu_j^{\mathrm{out}}(G_i), \tag{3}$$

where $L_j.IC$ and $L_j.OC$ is the input and output unit price of LLM L_j, respectively. Other terms are reused from latency model in Eq. 2.

Full Pure versus Residual Batches. The above models for accuracy, cost, and latency apply for both full pure, as well as, residual batches. When batches are full, we can significantly simplify the the expressions. For instance, for pure batches, the expression for simply reduces to $A(G_i, L_j)$, the accuracy for a query in the cluster for a given LLM. Likewise the latency expression Eq. 2 can be simplified by replacing the term $\sum_{i=1}^{K} n_i(b)$ by n_s, the size of the full batches, across all clusters. We further approximate $\mathrm{Tok}(\mathrm{Preamble}(b))$ in the expressions above for latency and cost for a full pure cluster by a cluster-level average for G_i, which can be computed offline. With such a replacement, latency for a full batch and cost of a full batch on a given LLM L_j depends only on (G_i, n_s). Such an approximation essentially treats expected latency and cost of any two full batches belonging to the same cluster on the same LLM as being equal – its value is considered as the expected latency/cost. We refer to the corresponding cost and latency for a pure full batch belong to cluster G_i on LLM L_j by $C_{fp}(G_i, L_j)$ and $T_{fp}(G_i, L_j)$ respectively. Note that instead of approximating batch latency/cost as above, we could use the actual composition of the batch to better estimate cost and latency. We will discuss our rationale for estimating the metrics as above in next section.

4.4 LLM Assignment Problem

We now formulate the LLM assignment problem, which is a mapping of batch to LLM based on the competing criteria - cost, accuracy, latency. The LLM assignment problem can be expressed as an optimization problem wherein we optimize one (or more of the metrics) and specify constraints on the other. We focus in OctoSelector on the variant where we minimize total cost, while putting constraints on accuracy and latency, defined by σ_A and σ_T, respectively. These constraints are on the average overall queries (expected sense) based on the LLM profile learned during the pre-processing phase. To formalize the assignment problem, let us define the following notations.

Batch Assignment An assignment of batches to LLM is expressed as $\{L_1 : \mathcal{B}_1, \cdots, L_m : \mathcal{B}_m\}$, where, $\mathcal{B}_i$ is the set of batches assigned to LLM L_i, for all $i = \{1, 2, \ldots, m\}$. Since a batch is assigned to a single LLM, $\mathcal{B}_i \cap \mathcal{B}_j = \emptyset$, $i \neq j$, and $\bigcup \mathcal{B}_i$ covers all batches, $\forall i, j \in \{1, \cdots m\}$.

Full pure batches assignment variables. We denote by $x_{i,j} \in \mathbb{N}^+$ the numbers of full pure batches originating from cluster G_i that are assigned to LLM L_j.

Residual batches assignment variables. Let $y_{b,j} \in \{0,1\}$ be the binary decision variable for residual batches, i.e., $b \in \mathcal{B}^{\text{res}}$ assigned to LLM L_j. Each residual batch is assigned to exactly one LLM, i.e., $\sum_{j=1}^{m} y_{b,j} = 1$.

Note that for an given batch assignment, the total number of batches assigned to LLM L_j, denoted as $|\mathcal{B}_j|$, is the sum of the number of full pure batches and the number of residual batches assigned to L_j. Specifically, $|B_j| = \sum_{i=1}^{K} x_{i,j} + \sum_{b \in \mathcal{B}^{\text{res}}} y_{b,j}$.

Example of Assignment Assume inside preamble D_1 we have : cluster ID G_1 contains full pure batches: $\{b_1, b_2, b_3, b_4\}$, and residual batches: $\{b_5, b_6\}$, which will be assigned to LLM L_1, L_2. If an example of assignment plan leads to variable solution as: $x_{1,1} = 3$, $x_{1,2} = 1$, and $y_{b_5,1} = 0$, $y_{b_5,2} = 1$, $y_{b_6,1} = 1$, $y_{b_6,2} = 0$, this indicating that, any three batches inside the G_1 will be assigned to L_1, and the rest will be assigned to L_2, and for residual batches, b_5 will be assigned to L_2, wherein b_6 will be assigned to L_1.

Given an assignment plan $\mathcal{A}$, we can determine the cost, accuracy and latency using the models described in Section 4.3 based with the assignment variables.

In particular, the cost of a given assignment corresponds to the cost of assigning $x_{i,j}$ full batches from cluster G_i to LLM L_j, plus the cost of any residual batches assigned to it. Thus, the total cost of an assignment can be expressed as:

$$\text{cost}(\mathcal{A}) = \sum_{i=1}^{K} \sum_{j=1}^{m} x_{i,j}\, C_{\text{fp}}(G_i, L_j) \;+\; \sum_{b \in \mathcal{B}^{\text{res}}} \sum_{j=1}^{m} y_{b,j}\, C(b, L_j) \tag{4}$$

In the formula above, the cost of $\mathcal{A}$, is computed by adding up the costs of assigning $x_{i,j}$ full batches of queries belonging to cluster G_i to LLM L_j with the cost of residual batch assignments to various LLMs. We can similarly compute the accuracy and latency metrics for an assignment $\mathcal{A}$ as follows:

$$\begin{aligned}\text{accuracy}(\mathcal{A}) = \frac{1}{N}\Bigg[&\sum_{i=1}^{K} \sum_{j=1}^{m} x_{i,j}\, n_s\, A(G_i, L_j) \\ &+ \sum_{b \in \mathcal{B}^{\text{res}}} \sum_{j=1}^{m} y_{b,j} \sum_{i=1}^{K} n_i(b)\, A(G_i, L_j) \Bigg] \end{aligned} \tag{5}$$

$$\text{latency}(\mathcal{A}) = \frac{1}{N}\left[\sum_{i=1}^{K} \sum_{j=1}^{m} x_{i,j}\, T_{\text{fp}}(G_i, L_j) + \sum_{b \in \mathcal{B}^{\text{res}}} \sum_{j=1}^{m} y_{b,j}\, T(b, L_j) \right], \tag{6}$$

where $T(b, L_j)$ and $C(b, L_j)$ are the unified batch latency (Eq. 2) and batch cost (Eq. 3), respectively. The terms $T_{\text{fp}}(G_i, L_j)$ and $C_{\text{fp}}(G_i, L_j)$ are full pure batch latency and cost expression, obtained by instantiating a full pure batch from cluster G_i with $|b| = n_s$ and replacing $\text{Tok}(\text{Preamble}(b))$ by the cluster-level average token length on preambles (detailed in subsection 4.3.

We can now formulate the optimization problem as:

$$\min \text{cost}(\mathcal{A}), \text{s.t. accuracy}(\mathcal{A}) \geq \sigma_A,\ \text{latency}(\mathcal{A}) \leq \sigma_T \tag{7}$$

and

$$\begin{aligned} \sum_{j=1}^{m} x_{i,j} &= \left|\mathcal{B}^{\text{full pure}} \in G_i\right|, \quad i = 1, \ldots, K, \\ \sum_{j=1}^{m} y_{b,j} &= 1, \forall\, b \in \mathcal{B}^{\text{res}}, \\ x_{i,j} \in \mathbb{N}^{+},&\ \ y_{b,j} \in \{0,1\}, \end{aligned}$$

where the expression cost($\mathcal{A}$), accuracy($\mathcal{A}$), latency($\mathcal{A}$) follow the formulation in Eq. 4-6, respectively.

The Eq. 7 minimizes total cost by summing contributions from full pure batches and residual batches. Accuracy and latency constraints are enforced at the *query* level: full pure terms are count-weighted via $x_{i,j}$ (each full batch from G_i contributes n_s queries), while residual terms average over the actual composition of each residual batch via $y_{b,j}$. The assignment constraints ensure that all full pure batches originating from each cluster are accounted for and that every residual batch is routed to exactly one LLM.

Note that in the above characterization, we have assumed the cost, quality, and latency of a given full pure batch of queries from the same cluster assigned to the same LLM to be essentially equal. While this assumption is necessary for our approach to ensure accuracy, as our method is based on similar difficulty clustering, it is not strictly required for latency or cost. In practice, latency and cost depend on the input and output sizes of the prompt. Since we know the actual sizes of the batches, we could in principle adopt a more fine-grained model that expresses the individual cost of each full pure batch, similar to our treatment of residual batches. However, doing so would require introducing a binary assignment variable for every batch, specifically, the full pure batch assignment variable $x_{i,j}$ becomes $x_{b,j} \in \{0,1\}$. If a cluster G_i contains $|\mathcal{B}_i^{\text{full pure}}|$ numbers of full pure batches, individual assignment variable numbers of $x_{b,j}$ equal to the $|\mathcal{B}_i^{\text{full pure}}|$. Instead, by assuming that full pure batches from the same cluster share the same metrics, we reduce the numbers of full pure batches assignment variables to $x_{i,j}$. Within the cluster G_i, numbers of variables $x_{i,j}$ is bounded by numbers of candidate LLMs m, and overall number of variable $x_{i,j}$ is bounded by Km, where K is the total cluster number.

5 Evaluation

5.1 Experimental Setup

Evaluation Tasks & Benchmarks We evaluate OctoSelector on two tasks, *NL2SQL* and *sentiment analysis*. NL2SQL is chosen for its importance in democratizing access to databases. Additionally, to measure the difficulty per NL2SQL task, it considers syntax features from both the input (e.g., linguistic features in Table 2) and the output (e.g., SQL features in Table 3), making it a representative task for evaluation. We evaluate NL2SQL using two popular benchmarks, Spider [45] and BIRD [29].

Spider Benchmark. We employ 7,000 queries from the Spider training set (`train_spider`) in the pre-processing stage to compute the cluster metric table, and 1,034 queries from the development set for inference.

BIRD Benchmark. Compared with Spider, queries in BIRD exhibit higher difficulty with larger table schema [35]. In OctoSelector, 9,428 queries from the training set are used for pre-processing, and 1,534 queries from the development set are employed for inference.

We evaluate the second task, sentiment analysis, using the IMDb benchmark [40], where each task predicts a movie's rating score based on its corresponding review.

IMDb Benchmark [40]. The IMDb dataset consists of movie reviews labeled with sentiment (rating scores from 1 to 10) for sentiment analysis tasks. We use 4,000 reviews with labels during the pre-processing stage in OctoSelector and another 1,000 reviews without labels for inference.

Candidate LLMs We evaluate OctoSelector using proprietary LLMs listed in Table 4, chosen for their popularity in real-world data processing tasks and accessibility through commercial APIs. OctoSelector works when taking all LLMs listed in Table 4 as the LLM pool. However,

LLM	Company	Input Price/1M	Output Price/1M
GPT-4o Mini	OpenAI	0.15	0.6
GPT-3.5 Turbo	OpenAI	3	6
GPT-4o	OpenAI	2.5	10
Gemini 1.5 Flash 8B	Google	0.0375	0.15
Gemini 1.5 Flash	Google	0.075	0.3
Gemini 1.5 Pro	Google	1.25	5
Claude 3.5 Haiku	Anthropic	0.8	4
Claude 3.5 Sonnet	Anthropic	3	15

Table 4. LLM Pricing Comparison (March 2025 rates)

in a collection of models, if a small subset exhibits dominated performance (e.g., lower cost and higher accuracy), it can obscure the observations for the remaining majority of models, since those dominated models are unlikely to be selected. For example, as will be shown later, Gemini models are overall faster and cheaper than Claude models, resulting in few Claude models being selected and, consequently, limited observations for them. Therefore, to increase the diversity of observations, we group all LLMs from the same provider into one LLM pool used in OctoSelector and for other baselines. This allows us to make more fine-grained observations across models from different providers, which is also also a common practical setting to select models from the same provider.

Baselines. We consider the following three baselines to compare against OctoSelector:

Baseline 1: Our first baseline is the *single LLM approach*, which uses a single LLM to process all queries in the same batch created by OctoSelector.

Baseline 2: The second baseline is an improved version of the first one. Instead of using a single LLM to process all queries in one batch, it selects the cheapest LLM from the candidate models that satisfy the user-specified constraints. How these constraints are set will be clarified shortly in the evaluation setup.

Baseline 3: The third baseline is GraphRouter [20], which is designed for assigning LLMs per query by optimizing an objective (e.g., minimize cost) under user-specified constraints, a setting similar to OctoSelector. Since GraphRouter shows better performance than HybridLLM [17] and FrugalGPT [12], we only compared against GraphRouter [20]. However, GraphRouter natively does not support batching. We extend it with a batching mechanism to enable a fair comparison with OctoSelector, and denote this enhanced version as *GraphRouter Plus* or *GR+*. Specifically, given the LLM assignments produced by GraphRouter, we first group queries into clusters such that queries within the same cluster share the same LLM assignment. We then apply the same batching strategy as in OctoSelector to create batches for each cluster, allowing queries with the same assigned LLM and shared context to be batched together, thereby reducing cost and latency.

Evaluation Setups. While OctoSelector supports flexible optimization objectives over various metrics such as cost, latency, and accuracy, we focus on minimizing cost while constraining accuracy (σ_A) and latency (σ_T) in our evaluation. This setup is common for LLM-powered tasks, where the goal is to minimize cost while ensuring that accuracy (and/or latency) meets user-specified thresholds.

The accuracy threshold σ_A for a benchmark is defined over a range $[l, r]$, where l and r denote the lowest and highest accuracy of Baseline 1 (i.e., using a fixed LLM for all queries) when employing different LLMs from the candidate pool. For example, consider σ_A for Spider in Table 5, which spans $[0.661, 0.815]$, where 0.661 and 0.815 correspond to the accuracies of Baseline 1 using GPT-3.5-Turbo and GPT-4o, respectively. All baselines have no meaningful data points outside this range, and the same applies to OctoSelector. For example, the highest possible accuracy that OctoSelector can achieve is by employing the best model (e.g., GPT-4o among all OpenAI models) to execute

Benchmark	Constraints	Ranges
Spider	σ_A	$[0.661, 0.815]$
	σ_T(sec)	$[0.248, 0.583]$
BIRD	σ_A	$[0.398, 0.485]$
	σ_T(sec)	$[0.457, 0.974]$
IMDb	σ_A	$[1.98, 3.51]$
	σ_T(sec)	$[0.421, 1.71]$

Table 5. Constraint value range of accuracy (σ_A) and latency (σ_T) set across benchmarks.

every single query. We enforce a similar constraint range on latency, defined by the smallest and largest latency of Baseline 1 when using different LLMs.

We further discretize each range into 20 evenly spaced values, resulting in a total of 400 combinations of accuracy and latency test pairs.

5.2 Feasibility Regions

We first define the feasibility rate for OctoSelector as follows. If the LLM assignment plan created by a method has average estimated accuracy and latency that both satisfy a given constraint pair on accuracy and latency (σ_A, σ_T), i.e. there exists an optimal solution of Eq 7, we call such a plan *feasible* on (σ_A, σ_T).

We evaluate OctoSelector not just on a single constraint pair, but over a range on each of the constraint (Table 5). Specifically, each constraint range is uniformly divided into 20 values, and OctoSelector is evaluated at 20×20 (σ_A, σ_T) in total 400 pairs per benchmark. Consider the 20 by 20 pairs as a two-dimensional grid, we define the *feasibility region* for a method as the set of feasible constraint pairs, and further define the *feasibility rate (%)* as the number of feasible constraint pairs over the total number of pairs (i.e., 400).

Figure 3 shows the feasibility regions of each single LLM (provided by OpenAI) compared with OctoSelector on the Spider benchmark. The three circled points for each LLM represent the corresponding average accuracy and latency of Baseline 1 (*single LLM*), and we consider the bottom-left area of each circled point as the feasibility region of the corresponding LLM. The light blue area in Figure 3 denotes the feasibility region of OctoSelector, which is strictly larger than any of the three individual feasibility regions. For Baseline 2, defined as the union of the three feasibility regions (in this case, gpt-4o-mini, gpt-3.5, and gpt-4o), it always selects the cheapest model. The corresponding assignment plan of Baseline 2 is thus the cheapest feasible solution to the given constraint pair. We list the exact feasibility rates of the baselines for each LLM provider across all benchmarks in Table 6, where OctoSelector has the highest feasibility rates than all baselines for all LLMs.

	GPT						Gemini						Claude				
	4o-mini	3.5	4o	Union	GR+	OctoSel.	1.5-flash-8b	1.5-flash	1.5-pro	Union	GR+	OctoSel.	haiku	sonnet	Union	GR+	OctoSel.
Spider	5.5%	27.5%	19.5%	32.0%	30.2%	**42.0%**	75.0%	76.0%	28.5%	84.3%	76%	**94.5%**	0.2%	3.8%	3.8%	1.5%	**4.5%**
BIRD	6.0%	4.8%	40.5%	43.0%	9.4%	**56.8%**	0.0%	18.0%	18.0%	28.0%	24%	**37.7%**	2.0%	12.0%	12.0%	10.5%	**16.0%**
IMDb	85.5%	90.2%	90.0%	94.8%	86.5%	**95.0%**	4.8%	40.0%	72.0%	76.0%	9.5%	**79.0%**	46.8%	66.5%	66.5%	48%	**76.0%**

Table 6. Feasibility rate (%) across benchmarks, compared OctoSelector with three baselines: single-model, cheapest feasible solution on union of feasible regions, and GraphRouter plus.

5.3 Evaluation Results

For each LLM provider, we first compute each LLM's feasibility region within the constraint ranges shown in Table 5. Each cell in the feasibility region in Figure 3 corresponds to a constraint pair

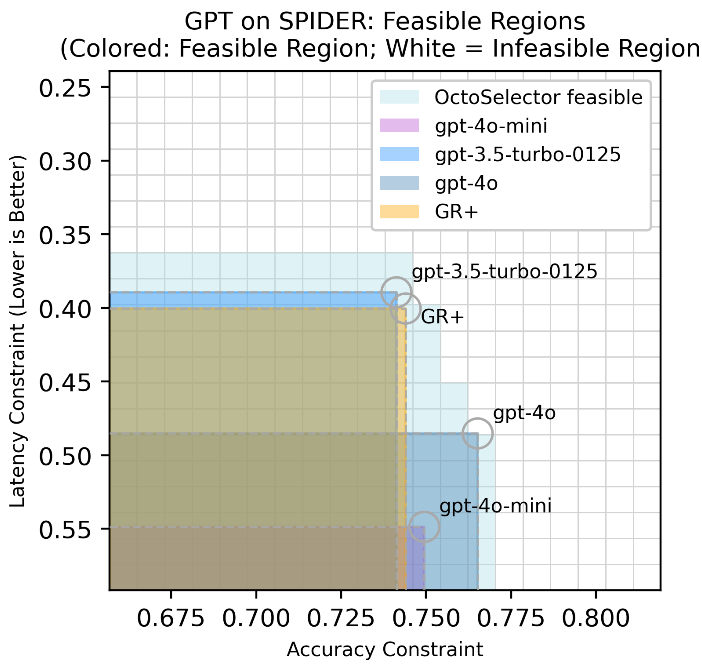


Fig. 3. Feasible region of LLMs from OpenAI provider compared with OctoSelector. The circled points of each GPT model denote the single LLM baseline result performance of average accuracy and latency value.

(σ_A, σ_T), for which OctoSelector returns an LLM assignment plan that satisfies the pair while minimizing cost.

Experiment 1: Feasibility Rate. We measure the feasibility rate, defined as the percentage of queries that satisfy the specified constraints across all baselines using different LLMs, as shown in Table 6. A higher feasibility rate indicates a better approach, as it reflects a greater ability to find assignment plans that fit more queries within the given constraints. In this experiment, OctoSelector has the highest feasibility rate among all baselines across all LLMs, demonstrating its great capability to offer an effective plan.

Experiment 2: Baseline 1 vs. OctoSelector. We compare OctoSelector against Baseline 1, i.e., the single-LLM method, in Figure 4. The y-axis shows the percentage of cost reduction, defined as $(1 - \frac{C_{\text{OctoSelector}}}{C_{\text{baseline}}}) \times 100\%$. We report the cost reduction that is averaged over all cells within the feasible ranges per method, that is, over the queries that satisfy both constraints for the enumerated constraint pairs. The x-axis shows the compared single-LLM model names. OctoSelector has a significantly lower cost than the single-LLM method. In particular, on the Spider benchmark, within the feasibility region of the LLM gpt-4o, OctoSelector achieves an average cost reduction of 81.5% compared to the cost incurred by a single-LLM gpt-4o invocation. In addition, OctoSelector achieves 40–95% cost savings; especially on the IMDb benchmark, the highest saving reaches 95.1% compared to gpt-3.5-turbo.

Experiment 3: Baseline 2 vs. OctoSelector. We compare OctoSelector with Baseline 2 in Figure 5. Recall that Baseline 2 is an improvement over the single LLM method, where it selects the model with the lowest cost from the candidates that satisfy the constraints (i.e., among the union of feasibility regions of all candidate LLMs from the same provider). We similarly report the average cost reduction, and observe a substantial improvement with OctoSelector. Across benchmarks,

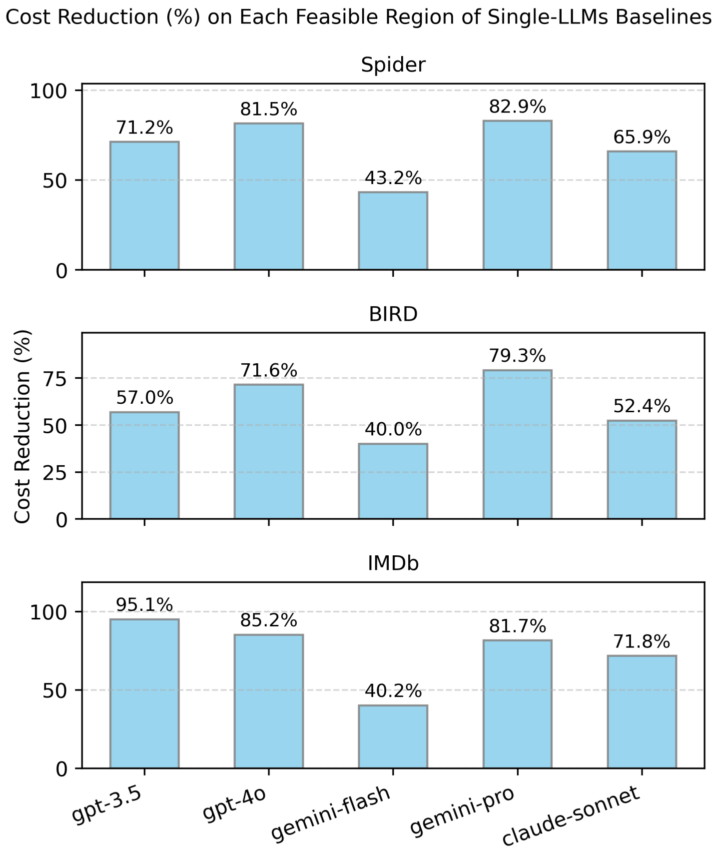


Fig. 4. Cost reduction (in percentage) compared with single-LLM baselines on their correspond feasible regions.

OctoSelector achieves 30.8% to 68% cost reduction, with particularly high savings on the IMDb benchmark, where the average reduction exceeds 60%.

Experiment 4: Baseline 3 vs. OctoSelector. We compare Baseline 3, GraphRouter Plus (GR+), with OctoSelector, and report the average cost reduction in Figure 6. OctoSelector achieves significant cost reductions across all three benchmarks and all LLM providers. The peak saving of 94.8% occurs on the IMDb benchmark when compared with GPT models. This is because GraphRouter tends to select gpt-3.5-turbo for most queries, which is relatively expensive according to the pricing table 4.

Experiment 5: Effect of Number of Query Clusters. We conduct an ablation study to examine how the number of query clusters affects the performance of OctoSelector. We report the cost reduction by comparing with OctoSelector against the single-LLM method (i.e., Baseline 1) as well as against the cheapest solution on GPT union feasible regions (i.e., Baseline 2) shown in Figure 7, which demonstrates the robustness of OctoSelector across different cluster number K, as the cost savings are within a fluctuation of 10%. Furthermore, we report the cluster statistics shown in Appendix A.3 in [6] By increasing K, each cluster size is becoming more balanced, as the standard deviation of cluster size is getting smaller. We choose $K = 15$ for the NL2SQL Spider benchmark experiment, where cluster statistics becomes relatively stable.

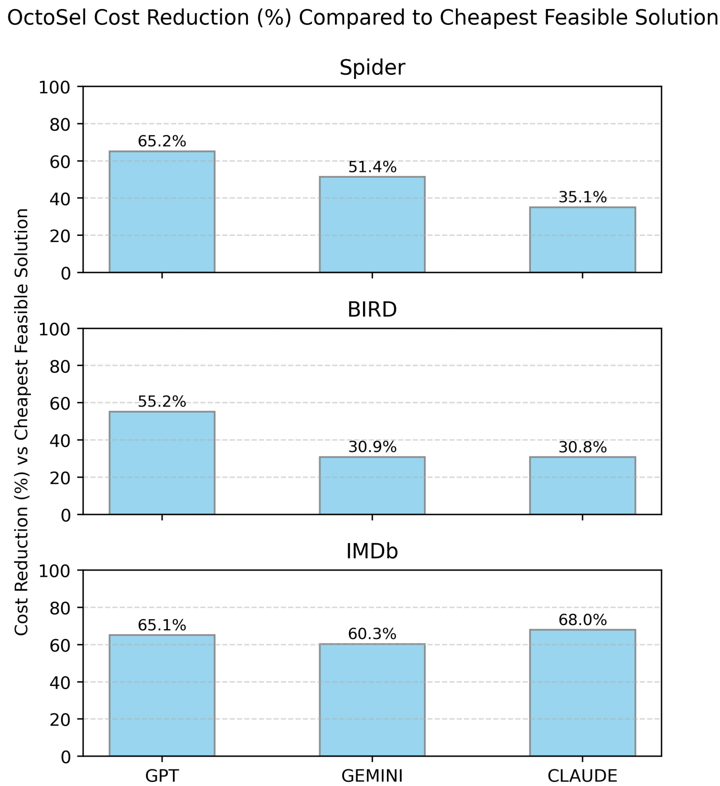


Fig. 5. Cost reduction (in percentage) compared to the cost of cheapest feasible solution on the union feasibility regions of (same provider's) single LLMs.

Experiment 6: Effect of Batch Size n_s. We examine the best batch size (n_s) for each LLM in GPT series, in both Spider and BIRD benchmark. The best batch size for a given LLM is considered to achieve the highest average accuracy, since the average latency and cost strictly decreased with larger batch size. For Spider benchmark, the Figure 8 indicates that the best batch size for each LLM within the same provider are close, though in BIRD benchmark (Appendix A.4 in [6]) the highest average accuracy for gpt-3.5-turbo is achieved at $n_s = 4$, while the rest two are achieved at $n_s = 6$. Thus, we evaluate OctoSelector with single-LLM baseline sharing the same batch size, where for Spider benchmark we use $n_s = 8$ while for BIRD case $n_s = 6$.

Experiment 7: Baseline 1 vs. non-batching vs. batching. To illustrate the effectiveness of optimization algorithms that used in OctoSelector, we compare the single-LLM performance (Baseline) with OctoSelector but not using batching strategy, i.e., one prompt only contain one query with necessary task instruction and context. This non-batching can also be consider as specific case of batching strategy with $n_s = 1$. In this experiment, we use Spider benchmark and single-LLMs in GPT series as baseline, and compute the cost reduction between the reference and the non-batching, as well as with batching strategy. From Figure 9 we can observe that, first, even with non-batching strategy, OctoSelector can achieve large cost savings ($\geq 69.8\%$ in dark blue bars), except for the gpt-4o-mini since it's the lowest cost that OctoSelector can achieve. Second, from non-batching (dark blue bar) to batching (light blue bar, with $n_s = 8$ in this case), OctoSelector further reduces the cost up to 44.2%, which strongly proves the effectiveness of batching strategy.

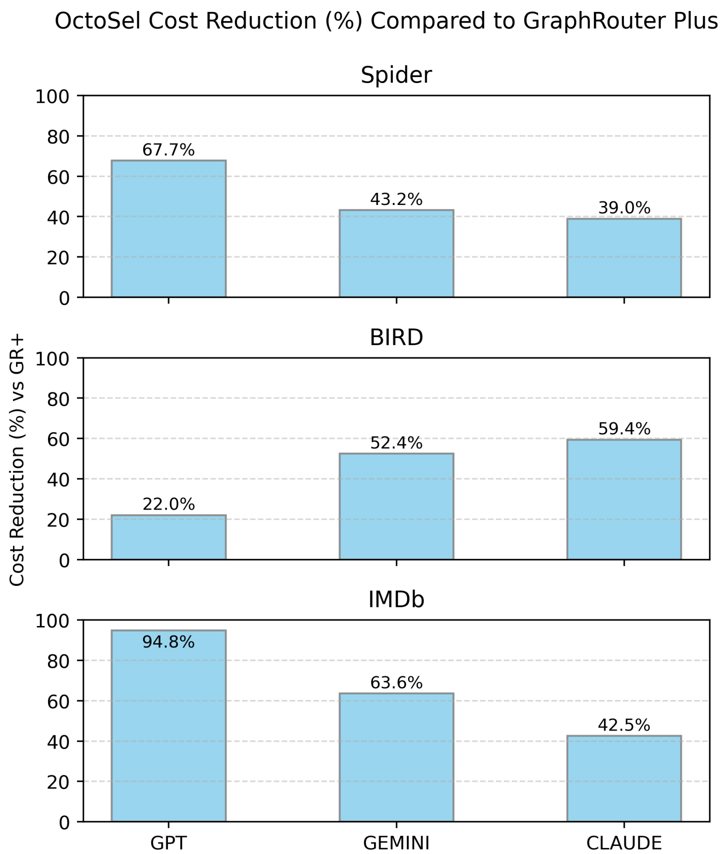


Fig. 6. Cost reduction (in percentage) compared to the cost of GraphRouter Plus (GR+) on the GR+'s feasibility regions

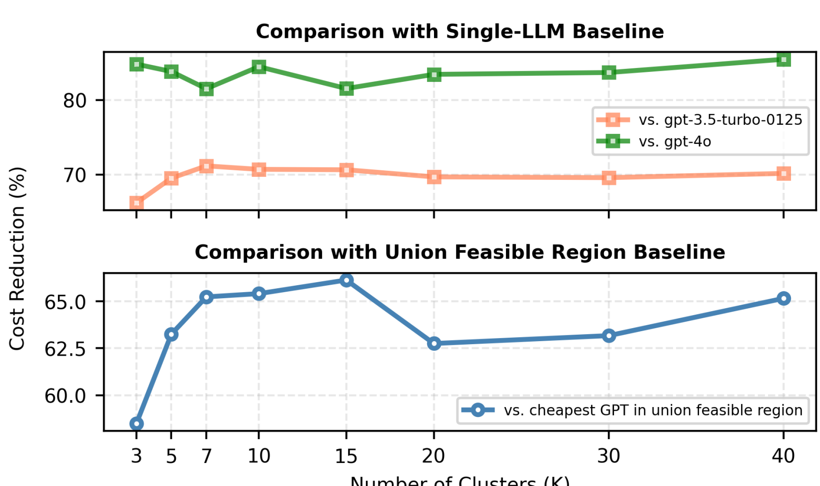


Fig. 7. Cost reduction on different cluster numbers K

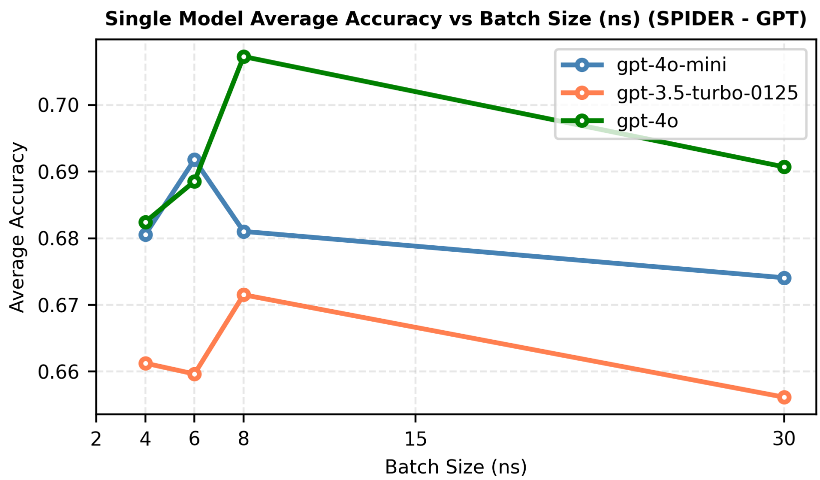


Fig. 8. Average accuracy of each single LLM (GPT) series vs. different batch size n_s on Spider benchmark.

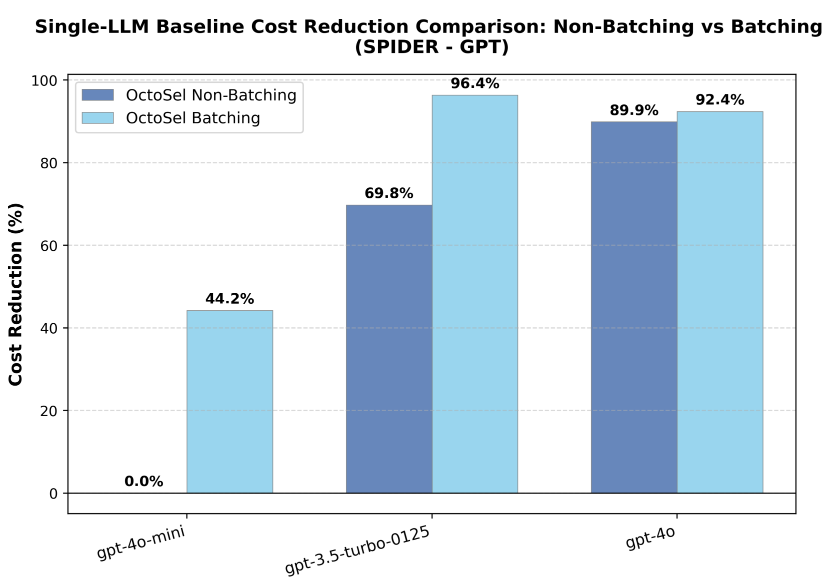


Fig. 9. Cost reduction comparing single-LLM (GPT) with non-batching and batching strategy.

5.4 Results of Model Selection

We illustrate the LLM assignment result on Spider benchmark across three LLM series, i.e., GPT, Gemini, and Claude in Table 7, where we select three different constraints pairs (two pairs for Claude) for each LLM series as example. The selected constraint pairs focus on different representative user preferences, i.e., low accuracy and medium latency, medium latency and low latency, and high accuracy but high latency. The first two columns are the constraints values, on accuracy (σ_A) and latency (σ_T), respectively, the third column is the estimated total cost that executing all

σ_A	σ_T	Cost	gpt-4o-mini, gpt-3.5, gpt-4o
0.720	0.520	0.049	[63,56], [2,14], [0,0]
0.725	0.435	0.158	[29,37], [36,33], [0,0]
0.764	0.551	0.229	[32,31], [12,1], [21,38]

σ_A	σ_T	Cost	gemini-1.5-flash-8b, 1.5-flash, 1.5-pro
0.781	0.251	0.007	[49,61], [16,9], [0,0]
0.789	0.357	0.011	[19,30], [45,40], [1,0]
0.812	0.464	0.147	[6,2], [18,7], [41,61]

σ_A	σ_T	Cost	claude-3.5-haiku, claude-3.5-sonnet
0.550	0.567	0.236	[64,42], [1,28]
0.700	0.678	0.382	[18,26], [47,44]

Table 7. Assignment examples across GPT, Gemini, and Claude series LLM on Spider benchmark; Aggregated assignment results are in last column, indicating [#full batches, #residual batches].

queries under the assignment plan. The last column is the assignment plan. The actual plan in OctoSelector is per-batch assignment, for demonstration, we aggregate by the batch type and their assigned model names. For example, in Table 7, the first row in GPT series model with value $[63, 56]$ means there are in total 63 full pure batches and 56 residual batches assigned to gpt-4o-mini, 2 full pure batches and 14 residual batches assigned to gpt-3.5. OctoSelector recommends substantially different assignment plans for the three selected constraint pairs, demonstrating its ability to adapt to diverse user constraints while maintaining effectiveness. Similar assignment pattern appears in Gemini and Claude series models in Table 7, respectively.

Selection with Dominated Models. Dominance in our setting is evaluated at the cluster level and thus depends on the task. A model can be dominated on accuracy, latency, and cost in some clusters but not others, so it may still be selected in the final assignment for queries in which clusters it remains competitive. In contrast, if a model is dominated on all three metrics across all clusters, it offers no advantage under any constraint and will not be selected by OctoSelector. In practice, for a given task, such a fully dominated model is uncommon within a single vendor's model family, though it may arise when comparing models across different vendors.

5.5 Impact of Pre-processing Data Size

We analyze the impact of using different training data size during pre-processing and how the performance in terms of cost reduction varies during inference. We use NL2SQL task on Spider benchmark 7000 training queries as full set, and using GPT series LLMs as candidates. We randomly shuffle and select three typical size, i.e., 10%, 50% and 100% of the full set as representative training size, and evaluate each on full development data (1034 queries). We compare the cost reduction against three baselines described in Section 5.3, and the results are shown in Table 8. Notice that gpt-4o-mini on its feasible region is the minimum price, thus, we omit the comparison but only list the case for gpt-3.5-turbo and gpt-4o. The cost reduction improvement is most occurred from 10% to 50%, while the gains from 50% to 100% are relatively smaller. This result suggests that OctoSelector's can achieve significant cost savings without using full training dataset. This observation suggests that the pre-processing cost can be reduced by profiling only 50% of the training set. In terms of cost amortization, profiling the full training set requires about 10K inference queries to recover the profiling cost, whereas profiling only 50% of the training data reduces this

Baseline/Training set size	**10%**	**50%**	**100%**
vs. Single-LLM baseline (gpt-3.5-turbo)	58.5	68.5	71.2
vs. Single-LLM baseline (gpt-4o)	69.8	79.6	81.5
vs. Union Feasible Region baseline (GPT)	55.2	62.7	65.2
vs. GR Plus baseline	56.9	65.6	67.7

Table 8. Cost reduction of OctoSelector trained with 10%, 50%, 100% training data size compared with three baselines.

threshold to about 5K queries. In addition, we report the training data size impact on cluster models quality in Appendix A.2 in [6].

5.6 Additional Analysis in Appendix

We conducted several other experiments to evaluate OctoSelector, which are listed in Appendix [6]:

Appendix A.6: In addition to processing a set workload with batching strategy, OctoSelector also support query-at-a-time strategy, where the optimization for LLM assignment is performed at query-level, and thus global optimization on objective (such as cost) is impossible. We focus on reducing the cost while maintain a minimum numbers of queries that violate constraints (on accuracy and latency);

Appendix A.2: While Section 5.5 reports the impact of training data size on OctoSelector cost reduction, we also examine its effect on cluster quality. For each LLM and metric (accuracy, latency, or cost), we compute an R^2 score that measures how much of the metric's variance is explained by the cluster assignment. Higher R^2 indicates more homogeneous clusters. The results show that R^2 increases with training data and stabilizes at around 80% of the full training set.

6 Conclusion

In this paper, we introduced OctoSelector, a framework for LLM selection across diverse tasks. OctoSelector first models query difficulty by extracting features from both input and output, and groups queries into similar difficulty clusters. Based on this clustering, it formulates LLM assignment as an integer linear programming problem that jointly optimizes for accuracy, latency, and cost. We proposed batching strategy for LLM assignment—leveraging shared context to reduce overhead. Experimental results demonstrate that OctoSelector consistently achieves higher feasibility rates and better cost-effectiveness than the baselines, without compromising performance.

We position OctoSelector as a general framework in which only feature extraction component requires modest effort to specify task-dependent features. Once features are defined, the remaining components - clustering, performance estimation and optimization for LLM assignment - are task-independent.

While OctoSelector allows adding new LLMs and workloads, there are several directions of future work. One direction for enhancing OctoSelector is the implementation of guardrail mechanisms [10] for LLM response validation. Furthermore, we plan to integrate OctoSelector into systems with LLM-based semantic operators. Systems such as LOTUS [41] and Palimpzest [31] support SQL query with natural language predicates evaluated by LLMs. OctoSelector can be integrated as a backend routing layer to decide which LLM should execute at each semantic operator.

Acknowledgments

We gratefully acknowledge the support of the following grants NSF-2540011, 2420846, 2419844, 2245372, 2133391, 2113930, 2008993, 1952247.

References

[1] 2025. *https://ai.google.dev/gemini-api/docs/models*.

[2] 2025. *https://docs.anthropic.com/en/docs/about-claude/models/all-models*.

[3] 2025. *https://openai.com/api/pricing/*.

[4] 2025. *https://platform.openai.com/docs/models/gpt-4o*.

[5] 2025. *https://www.anthropic.com/pricing#anthropic-api*.

[6] 2026. *https://drive.google.com/file/d/10udgPuUg-XksCmab9jzUMyutc56jbBOK/view?usp=sharing*.

[7] Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, et al. 2024. Automix: Automatically mixing language models. *Advances in Neural Information Processing Systems* 37 (2024), 131000–131034.

[8] Shengnan An, Zexiong Ma, Zeqi Lin, Nanning Zheng, Jian-Guang Lou, and Weizhu Chen. 2024. Make your llm fully utilize the context. *Advances in Neural Information Processing Systems* 37 (2024), 62160–62188.

[9] Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. 2023. Language models enable simple systems for generating structured views of heterogeneous data lakes. *arXiv preprint arXiv:2304.09433* (2023).

[10] Suriya Ganesh Ayyamperumal and Limin Ge. 2024. Current state of LLM Risks and AI Guardrails. *arXiv preprint arXiv:2406.12934* (2024).

[11] Fabian Biester, Mohamed Abdelaal, and Daniel Del Gaudio. 2024. Llmclean: Context-aware tabular data cleaning via llm-generated ofds. In *European Conference on Advances in Databases and Information Systems*. Springer, 68–78.

[12] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176* (2023).

[13] Wei-Hao Chen, Weixi Tong, Amanda Case, and Tianyi Zhang. 2025. Dango: A Mixed-Initiative Data Wrangling System using Large Language Model. *arXiv preprint arXiv:2503.03154* (2025).

[14] Yibin Chen, Yifu Yuan, Zeyu Zhang, Yan Zheng, Jinyi Liu, Fei Ni, and Jianye Hao. 2024. Sheetagent: A generalist agent for spreadsheet reasoning and manipulation via large language models. In *ICML 2024 Workshop on LLMs and Cognition*.

[15] Mengyao Cui et al. 2020. Introduction to the k-means clustering algorithm based on the elbow method. *Accounting, Auditing and Finance* 1, 1 (2020), 5–8.

[16] Alexander Philip Dawid and Allan M Skene. 1979. Maximum likelihood estimation of observer error-rates using the EM algorithm. *Journal of the Royal Statistical Society: Series C (Applied Statistics)* 28, 1 (1979), 20–28.

[17] Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Ruhle, Laks VS Lakshmanan, and Ahmed Hassan Awadallah. 2024. Hybrid llm: Cost-efficient and quality-aware query routing. *arXiv preprint arXiv:2404.14618* (2024).

[18] Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. 2024. Combining small language models and large language models for zero-shot NL2SQL. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2750–2763.

[19] Yuankai Fan, Zhenying He, Tonghui Ren, Can Huang, Yinan Jing, Kai Zhang, and X Sean Wang. 2024. Metasql: A generate-then-rank framework for natural language to sql translation. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1765–1778.

[20] Tao Feng, Yanzhen Shen, and Jiaxuan You. 2024. Graphrouter: A graph-based router for llm selections. *arXiv preprint arXiv:2410.03834* (2024).

[21] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-sql empowered by large language models: A benchmark evaluation. *arXiv preprint arXiv:2308.15363* (2023).

[22] Vidhya Govindaraju, Ce Zhang, and Christopher Ré. 2013. Understanding tables in context using standard NLP toolkits. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 658–664.

[23] Neel Guha, Mayee Chen, Trevor Chow, Ishan Khare, and Christopher Re. 2024. Smoothie: Label free language model routing. *Advances in Neural Information Processing Systems* 37 (2024), 127645–127672.

[24] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2025. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems* 43, 2 (2025), 1–55.

[25] Saehan Jo and Immanuel Trummer. 2024. Thalamusdb: Approximate query processing on multi-modal data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[26] Danny Leybzon and Corentin Kervadec. 2024. Learning, forgetting, remembering: Insights from tracking llm memorization during training. In *Proceedings of the 7th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*. 43–57.

[27] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The dawn of natural language to SQL: are we fully ready? *arXiv preprint arXiv:2406.01265* (2024).

[28] Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[29] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems* 36 (2023), 42330–42357.

[30] Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. 2023. Table-gpt: Table-tuned gpt for diverse table tasks. *arXiv preprint arXiv:2310.09263* (2023).

[31] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baile Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, Rana Shahout, et al. 2025. Palimpzest: Optimizing ai-powered analytics with declarative query processing. In *Proceedings of the Conference on Innovative Database Research (CIDR)*. 2.

[32] Lei Liu, So Hasegawa, Shailaja Keyur Sampat, Maria Xenochristou, Wei-Peng Chen, Takashi Kato, Taisei Kakibuchi, and Tatsuya Asai. 2024. AutoDW: Automatic Data Wrangling Leveraging Large Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 2041–2052.

[33] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the Middle: How Language Models Use Long Contexts. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://arxiv.org/abs/2307.03172

[34] Tongyu Liu, Ju Fan, Nan Tang, Guoliang Li, and Xiaoyong Du. 2024. Controllable tabular data synthesis using diffusion models. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–29.

[35] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2024. A Survey of NL2SQL with Large Language Models: Where are we, and where are we going? *arXiv preprint arXiv:2408.05109* (2024).

[36] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[37] James MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, Vol. 5. University of California press, 281–298.

[38] Sean Massung, ChengXiang Zhai, and Julia Hockenmaier. 2013. Structural parse tree features for text representation. In *2013 IEEE seventh international conference on semantic computing*. IEEE, 9–16.

[39] Yael Moros-Daval, Fernando Martínez-Plumed, and José Hernández-Orallo. 2024. Language Task Difficulty Prediction Through LLM-Annotated Meta-Features. In *ECAI 2024*. IOS Press, 2434–2441.

[40] Aditya Pal, Abhilash Barigidad, and Abhijit Mustafi. 2020. IMDb Movie Reviews Dataset. https://doi.org/10.21227/zm1y-b270

[41] Liana Patel, Siddharth Jha, Melissa Pan, Harshit Gupta, Parth Asawa, Carlos Guestrin, and Matei Zaharia. 2024. Semantic Operators: A Declarative Model for Rich, AI-based Data Processing. *arXiv preprint arXiv:2407.11418* (2024).

[42] Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, Vol. 242. Citeseer, 29–48.

[43] Lei Sheng and Xu Shuai Shuai. 2025. Slm-sql: An exploration of small language models for text-to-sql. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*. 1497–1512.

[44] Marina Solnyshkina, Radif Zamaletdinov, Ludmila Gorodetskaya, and Azat Gabitov. 2017. Evaluating text complexity and Flesch-Kincaid grade level. *Journal of social studies education research* 8, 3 (2017), 238–248.

[45] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887* (2018).

[46] Chao Zhang, Yuren Mao, Yijiang Fan, Yu Mi, Yunjun Gao, Lu Chen, Dongfang Lou, and Jinshu Lin. 2024. FinSQL: model-agnostic LLMs-based text-to-SQL framework for financial analysis. In *Companion of the 2024 International Conference on Management of Data*. 93–105.

[47] Zeyu Zhang, Paul Groth, Iacer Calixto, and Sebastian Schelter. 2024. Directions Towards Efficient and Automated Data Wrangling with Large Language Models. In *2024 IEEE 40th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 301–304.

Received October 2025; revised January 2026; accepted February 2026