LakeHelm: Zero-Shot Lakehouse Advisor for Joint Engine-Format Selection and Configuration

Zhongwei Xu
University of Michigan
Ann Arbor, Michigan, USA
xzw@umich.edu

Siyuan Dong
University of Michigan
Ann Arbor, Michigan, USA
dougdong@umich.edu

Haotian Gong
University of Michigan
Ann Arbor, Michigan, USA
haotiang@umich.edu

Donna Pham
University of Michigan
Ann Arbor, Michigan, USA
phdonn@umich.edu

Lin Ma
University of Michigan
Ann Arbor, Michigan, USA
linmacse@umich.edu

ABSTRACT

Lakehouse systems unify the strengths of data lakes and data warehouses and are rapidly becoming a dominant architecture for analytic data management. The lakehouse architecture decouples system design into interoperable subsystems—execution engines (e.g., Spark, Trino, Presto) and table formats (e.g., Delta Lake, Iceberg, Hudi)—giving users flexibility to mix and match. However, jointly selecting and configuring these subsystems is hard: subsystem choices and configurations interact in complex ways, and online trial-and-error is costly (or infeasible when migration is required). Although there is extensive work on database tuning, most methods target a single subsystem and thus miss cross-dependencies; many also rely on iterative online tuning that is prohibitively expensive. In this work, we present LakeHelm, a zero-shot lakehouse advisor that jointly recommends an engine–format pair and its configuration without online feedback. LakeHelm uses a dual-gate Mixture-of-Experts model: separate gates specialize in engine and format choices, and experts learn configuration surrogates for each subsystem combination. To enhance generalization, we augment training data with generated SQL templates and synthesized workloads, layered atop collected runs that explore the configuration space. Evaluated across five standard benchmarks (TPC-DS, TPC-H, JOB, SSB, SSB-Flat), LakeHelm delivers competitive execution times—averaging 1.35× speedup over a fixed overall-best lakehouse configuration across a large number of workload variations. It achieves this via zero-shot inference on unseen workloads in seconds, without costly online experimentation.

PVLDB Reference Format:
Zhongwei Xu, Siyuan Dong, Haotian Gong, Donna Pham, and Lin Ma. LakeHelm: Zero-Shot Lakehouse Advisor for Joint Engine-Format Selection and Configuration. PVLDB, 19(8): 1768 - 1781, 2026.
doi:10.14778/3811243.3811250

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/umich-db/LakeHelm.

Proceedings of the VLDB Endowment, Vol. 19, No. 8 ISSN 2150-8097.
doi:10.14778/3811243.3811250

1 INTRODUCTION

With the rapid growth and increasing heterogeneity of data generated by web services, applications, and machine learning pipelines, organizations increasingly adopt the lakehouse architecture as a data processing solution that combines the flexibility and cost-efficiency of data lakes with key data warehouse capabilities [5, 8]. Such an architecture decouples storage and compute, allowing users to independently select different *subsystems*, including query engines (Spark [4], Presto [32], Trino [37]) and open table formats (e.g., Delta [10], Iceberg [2], Hudi [1]). This modularity alleviates platform lock-in and avoids tight coupling to a monolithic system. The growing ecosystem of stateless engines and open formats also broadens options for new adopters and enables seamless upgrades and component replacement for existing deployments.

However, while the lakehouse architecture's modular nature provides flexibility, it also introduces complexity: organizations must determine which combination of open table formats and query engines is most suitable for their specific workloads and subsequently configure these components' knobs properly. **This fundamental challenge—selecting the efficient subsystems combination with proper configuration for specific workloads—is critical for achieving desired performance outcomes.** Empirically evaluating every possible combination is prohibitively expensive, as it requires reloading raw data to rebuild metadata structures for each table format [8], followed by executing resource-intensive workloads across different engine configurations. Existing auto-tuning work for lakehouses has primarily targeted a single subsystem (e.g., Spark tuning [23, 25, 42, 43, 49]), leaving cross-subsystem interactions unaddressed. To our knowledge, no tool holistically tunes engines *and* table formats to optimize end-to-end performance. This paper presents such a tool and confronts two key challenges:

Challenge 1: Complex multi-subsystem interactions— Lakehouses comprise interoperable subsystems, each exposing many configuration knobs; choices in one subsystem strongly influence desirable settings in others. Consequently, users face a compounded problem: first selecting subsystems (e.g., a table format like Iceberg and an execution engine like Spark) that best suit the workload, then optimizing configurations within each while accounting

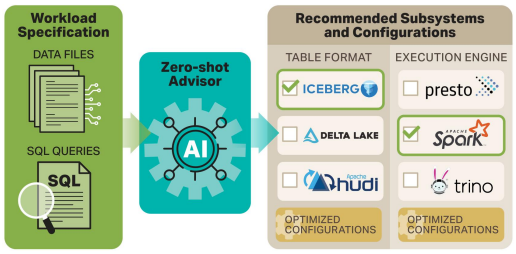
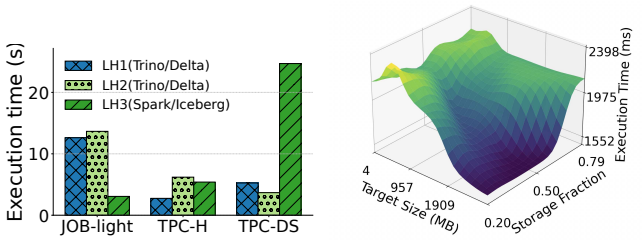


Figure 1: LakeHelm: Zero-Shot Lakehouse Advisor



(a) Performance across workloads. **(b)** Knob interaction effects.

Figure 2: Impact of subsystems and configurations. (a) Performance of the best lakehouse (**LH**), including subsystems and configurations, for each workload across all workloads. (b) Relationship between Spark's storage fraction (x-axis), Iceberg's target file size (y-axis), and the TPC-DS query execution time (z-axis).

for cross-subsystem coupling. This yields a massive combinatorial search space. A large body of prior work examines database tuning—especially ML-based methods that model and optimize high-dimensional knob spaces [13, 19, 23, 39, 41, 43, 45, 47, 49]. However, these approaches predominantly target a *single* system (e.g., Spark or PostgreSQL), and thus do not address the fundamental challenge of jointly selecting and tuning subsystems across categories or modeling their cross-subsystem interactions.

Challenge 2: Generalization to unseen workloads—Most tuning approaches rely on online adaptation: iteratively evaluating candidate configurations and updating a model until convergence. Even with knowledge transfer from prior workloads [46], they typically require tens to hundreds of trials. This dependence is ill-suited to end-to-end lakehouse tuning: changing subsystems entails costly data rewrites and workload migrations, and even configuration changes (e.g., file-size settings) can force expensive reloads. These constraints motivate a **zero-shot** approach that recommends subsystem combinations and configurations for unseen workloads without online iterations. The central obstacle is *data quantity and diversity*: a zero-shot model needs broad, representative training data to generalize across workload patterns, yet existing datasets (e.g., standard benchmarks) lack the breadth required for lakehouse deployments. While prior work, E2ETune [15], also explores zero-shot tuning, it targets a single system rather than the multi-subsystem lakehouse setting, where evaluation costs and generalization challenges are dramatically amplified.

Therefore, we propose **LakeHelm**, a zero-shot lakehouse tuning advisor that recommends promising subsystems and configurations given a workload specification (i.e., data files and queries), as shown in Figure 1. It addresses the above challenges through two key innovations. *First*, it uses a **novel dual-gate Mixture-of-Experts** (MoE [16]) architecture that explicitly models cross-subsystem interactions. Our insight is that different subsystem combinations exhibit fundamentally different performance characteristics that require specialized experts, while the subsystem selection itself depends on workload patterns that can be learned by dedicated gating networks. Furthermore, it employs training strategies specialized for lakehouses: different training stages separately train gates for table format and execution engine selection, enabling fine-grained specialization; another stage trains each expert to learn configuration optimization for specific subsystem combinations, while leveraging the unique signals available in our domain—explicit ground truth for best subsystem choices from collected execution traces. *Second*, LakeHelm employs **comprehensive data augmentation** techniques combining workload synthesis and large language model (LLM)-generated queries to enable robust generalization. Traditional benchmarks have limited queries and exhibit biases toward specific subsystem combinations, which lack sufficient quantity and diversity for training effective zero-shot models. Our approach generates synthetic workloads with controllable complexity while preserving realistic execution characteristics.

Across five standard benchmarks (TPC-DS, TPC-H, JOB, SSB, SSB-Flat [21, 28, 29, 34, 35]), LakeHelm outperforms existing methods and delivers competitive runtime on unseen workloads. It selects strong engine–format combinations with well-tuned configurations that, on average, 1.4× speedup over a fixed overall-best lakehouse configuration across a large number of workload variations. Although it does not guarantee optimality, inference completes in seconds and yields high-quality recommendations without costly online experimentation. As the first attempt toward joint engine–format lakehouse tuning, LakeHelm exhibits robust zero-shot generalization—even when target workloads differ markedly from the training data—demonstrating its ability to navigate the complex joint optimization space. Finally, zero-shot tuning is not mutually exclusive with online tuning: when resources permit, users can apply existing subsystem-specific methods (e.g., Spark-focused tuners) on top of LakeHelm's recommendations to further improve performance.

2 MOTIVATION

2.1 Complex Multi-Subsystem Interactions

Recent work has demonstrated the significant performance gain achieved by tuning subsystem-specific configurations in lakehouses, particularly Spark configurations [23, 25, 42, 49]. Our initial experiments further demonstrate the importance of selecting the proper subsystem combinations and jointly tuning their configurations.

First, we evaluate three representative engine–format combinations on three benchmark workloads (Figure 2a). For each workload, we exhaustively tune all subsystem combinations and configurations (cost in Section 2.2), yielding a workload-specific best lakehouse—**LH1**, **LH2**, **LH3**. Two workloads achieve their best time with Trino–Delta, while one favors Spark–Iceberg, highlighting workload sensitivity to subsystem choice. We then apply each workload's best lakehouse to the other workloads. Performance shifts

sharply: **LH3** is best on JOB-light but worst on TPC-DS. Even the same subsystem combination (Trino–Delta) behaves very differently under different configurations (cf. **LH1** vs. **LH2**), and relative rankings can invert. These results underscore the need to tune both subsystem combinations and their configurations.

This unified approach, though necessary, substantially complicates optimization. As shown in Figure 2b, knobs from the execution engine and table format can interact through complex interdependencies. The figure highlights the interaction between Spark's storage fraction and Iceberg's target file size, revealing counterintuitive performance patterns on TPC-DS queries. Surprisingly, the best performance arises with the smallest storage fraction (0.20) and the largest target file size (2048MB), contrary to typical intuitions about memory allocation and file-granularity trade-offs. With minimal storage fraction, larger target files appear to compensate by reducing I/O and metadata overhead despite less memory for caching. This interdependence creates a challenging search space: small deviations from the optimal pairings cause significant slowdowns, making the space difficult to navigate effectively.

2.2 Generalization to Unseen Workloads

Another challenge for joint engine–format lakehouse tuning is the high cost of evaluating different subsystem combinations and configurations, which motivates a zero-shot tuning approach.

Prior work shows that configuration evaluation can account for up to 90% of total tuning time in a single database [7, 24]. In response, several methods reduce the number of tuning iterations [19, 20, 49], and some even recommend knobs in zero-shot using generalizable models without configuration evaluation [15]. These costs are amplified in lakehouse tuning. Table-format changes often require data migration, whose cost can scale exponentially with growing enterprise data volumes. Even within the same format, changing certain configurations (e.g., file size) can force data rewrites. Switching table formats incurs cumbersome workload migration, and configuration updates often require engine relaunches. Furthermore, even after migrating data and workloads to a specific combination and configuration, executing the workload itself can take substantial time and resources. In our first experiment (Section 2.1), exhaustive tuning to find the best lakehouse for each workload required **2 days** on average on a 15-node cluster over a 10 GB benchmark dataset. As modern workloads grow, this cost becomes increasingly prohibitive.

These costs motivate a zero-shot approach that tunes the lakehouse without iterative subsystem or configuration evaluation. This approach requires a generalizable tuning model, which in turn demands abundant, representative training data; however, current benchmarks provide limited quantity and diversity. Most standard benchmarks contain only tens to hundreds of queries, and their workloads can be biased toward particular engine–format combinations because of narrow query patterns and historical optimization choices. Models trained on such data struggle to provide robust recommendations for unseen workloads—precisely what zero-shot methods require. Therefore, increasing the quantity and diversity of training workloads is desirable; however, doing so is more challenging than naively generating random queries, which neither guarantees patterns that exercise different subsystems nor avoids prohibitive evaluation costs during the training stage.

Again, zero-shot tuning is not exclusive to iterative tuning; users can apply other online tuners when resources permit, where zero-shot methods provide strong initial subsystems and configurations.

3 SOLUTION OVERVIEW

In this section, we first discuss the problem formulation and then introduce the main components of our solution, LakeHelm.

3.1 Problem Formulation

Problem formulation. The subsystem-selection problem in lakehouse systems is to choose, for a given workload W on dataset P, the engine–format pair and configuration that minimize a performance metric. Let $T = \{t_1, \ldots, t_n\}$ denote the available table formats and $E = \{e_1, \ldots, e_m\}$ the compute engines, with the candidate set $C = T \times E$. For each candidate $c_i = (t_a, e_b) \in C$, let $CK_{c_i} = TK_a \times EK_b$ be its joint knob space, formed by the table-format knobs TK_a and the engine knobs EK_b. The optimal configuration for c_i is

$$ck^*_{c_i} = \arg\min_{ck \in CK_{c_i}} M(W, P, c_i, ck), \tag{1}$$

where M is a performance metric. The goal is to find the global optimum

$$(c^*, ck^*_{c^*}) = \arg\min_{c_i \in C,\, ck \in CK_{c_i}} M(W, P, c_i, ck). \tag{2}$$

We use execution time as the metric M, but the formulation and solution should be generally applicable to other metrics (e.g., cost).

3.2 System Architecture

Figure 3 presents the high-level architecture of our proposed solution. The diagram highlights how our two core contributions are integrated across two main stages: (1) **MoE Architecture for Modeling, Training, and Inference** — where the MoE framework addresses complex subsystem interactions and provides optimization recommendations, and (2) **Training Data Collection & Augmentation** — where data augmentation enhances the training corpus to address the generalization challenge to unseen workloads.

3.2.1 MoE Architecture for Modeling, Training, and Inference (Section 4). To model the complexity inherent in lakehouse performance tuning, we adopt a Mixture-of-Experts (MoE) architecture, which has achieved strong results [11, 33] in recent systems such as DeepSeek-R1 [9] and Mixtral [18], thanks to sparse conditional computation that routes inputs to specialized subnetworks. MoE naturally separates gating and expert components, enabling the gating network to specialize in subsystem selection while individual experts focus on configuration optimization. However, naively applying MoE to lakehouse tuning does not suffice; LakeHelm must address several critical challenges.

Modeling challenges and solutions. A straightforward MoE design would connect a single gate to the workload embedding and route inputs to generic experts, expecting them to discover specialties and jointly predict both lakehouse subsystems and configurations. This burdens the single gate with learning both engine- and table-format structure, creating a tight coupling that impedes

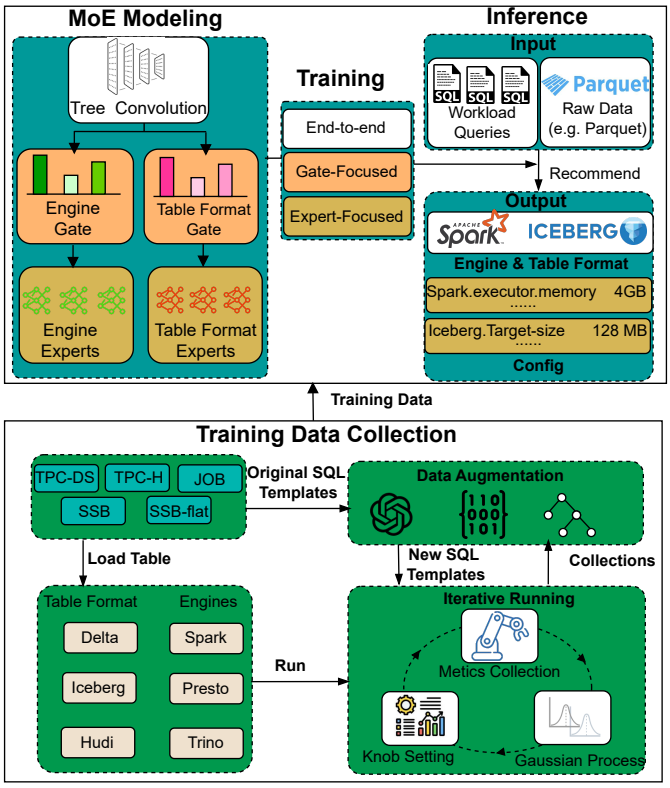


Figure 3: LakeHelm Architecture. Mixture-of-Experts framework addresses complex subsystem interactions and provides optimization recommendations; data augmentation enhances the training corpus to address generalization to unseen workloads.

convergence. To address this, LakeHelm employs a *dual-gate* architecture with dedicated gates for each subsystem type—one for table formats and one for execution engines—allowing each gate to specialize in its respective decision space (Section 4.1).

Training challenges and solutions. End-to-end training of the full network can be unstable because pipeline components (e.g., gate and expert models) have different convergence speeds and computational characteristics. Moreover, end-to-end supervision typically requires identifying the optimal subsystem combination and configuration for each workload, which prevents using suboptimal configurations and data from other subsystem combinations, severely limiting the training signal. LakeHelm addresses this with a specialized three-stage training regime—supervised gate pre-training and expert-specific training—that aligns convergence across components and fully utilizes the training data (Section 4.2).

As shown in Figure 4, our architecture integrates three coordinated components: a *tree convolution* model that encodes SQL queries into numeric embeddings; the *dual-gate* design that separately captures engine and table-format dependencies; and *expert models* that mirror the dual-subsystem structure—separate expert networks for engine and table-format configurations. This design allows each expert to specialize in configuration optimization patterns for a subsystem, while capturing the subsystem interactions.

During inference, given a user workload (queries and data files), the system feeds query features into the MoE pipeline, producing three recommendations: (i) table format, (ii) execution engine, and (iii) configuration knobs. This integrated approach delivers end-to-end optimization guidance for diverse workloads.

3.2.2 Training Data Collection with Data Augmentation (Section 5). A generalizable zero-shot model requires both proper modeling and high-quality training data. However, collecting such data faces critical challenges that motivate our augmentation approach. First, gathering sufficient samples across all subsystem combinations and configurations is computationally expensive and time-consuming. Second, existing benchmarks offer limited diversity and may not capture the full spectrum of query patterns and execution scenarios.

To address these challenges, LakeHelm first uses LLMs (ChatGPT [31]) to generate new SQL templates spanning a wide range of complexity, thereby increasing workload diversity (Section 5.1). It then applies a sampling-based workload synthesis technique to further expand the training set without additional query-execution cost (Section 5.2). As illustrated in Figure 3, our data-collection platform supports loading and executing queries across multiple lakehouse subsystem combinations—*table formats* (Delta, Iceberg, Hudi) and *execution engines* (Trino, Presto, Spark).

The data-collection procedure loads benchmark tables from TPC-DS [35], TPC-H [36], JOB-Light [40], SSB [30], and SSB-Flat [29], and executes queries across different engine–format combinations. To collect training data under varying configurations, LakeHelm begins with default settings, measures execution metrics, and then uses Gaussian Process—common in prior tuning methods [15, 19, 20, 39]—to iteratively propose new configurations based on observed runtimes; each workload undergoes up to 50 tuning iterations. The resulting dataset combines original benchmark queries with LLM-generated ones, providing broad workload coverage.

3.3 Assumptions and Limitations

We focus on three table formats and three execution engines commonly used in lakehouse deployments [8, 17, 25, 49], and we optimize for average query runtime. The problem formulation and framework are general and should extend to other lakehouse subsystems and performance targets. We assume the raw data is stored as Parquet [3] files on disk, which all the three table format supports, though our solution is independent of the raw data format.

Our approach assumes a fixed hardware environment—a 7-node cluster (Section 7.1.5). Resource constraints prevent us from collecting training data and evaluating on additional hardware. Incorporating hardware features and training models that generalize across environments is promising future work.

We tune the most impactful configuration knobs for each subsystem. Specifically, we select 5 knobs for table formats and 8 knobs for execution engines using prior knob selection methods [19, 39, 46]. These studies show selected knobs can already capture most of the attainable performance; expanding the knob set is future work.

Finally, although our method does not guarantee global optimality, as the first attempt at zero-shot, end-to-end lakehouse tuning, it achieves highly competitive performance, which significantly outperforms existing techniques (Section 7.2). It can be combined with subsystem-specific online optimization when resources permit.

4 MIXTURE OF EXPERTS ARCHITECTURE

In this section, we introduce our design of LakeHelm's MoE modeling, including the model architecture and training methodology.

4.1 Mixture-of-Experts Model Architecture

Understanding the complexity inherent in lakehouse performance tuning necessitates the recognition of two fundamental interaction paradigms within the optimization process. The primary paradigm involves modeling the intricate relationships between workload characteristics and subsystem combinations, while the secondary paradigm encompasses the complex interdependencies among specific subsystem components. The MoE architecture addresses these challenges through its inherently modular yet holistic design: the gating network specializes in subsystem selection decisions, while individual experts focus on configuration optimization for specific subsystem combinations. This architectural decomposition facilitates the learning of complex mappings between workload characteristics and the joint optimization of subsystem selection.

Nevertheless, naively applying to lakehouse tuning presents significant challenges. A straightforward MoE design would connect a single gate to the workload embedding and route each input to several generic experts, expecting them to discover specialties and jointly predict both lakehouse subsystems and configurations. However, this naive approach faces two critical obstacles: (1) with limited training data, experts often fail to converge on distinct specialties, and (2) even when they specialize, directly predicting optimal configurations remains challenging due to the vast configuration space and the high cost of obtaining ground-truth optima.

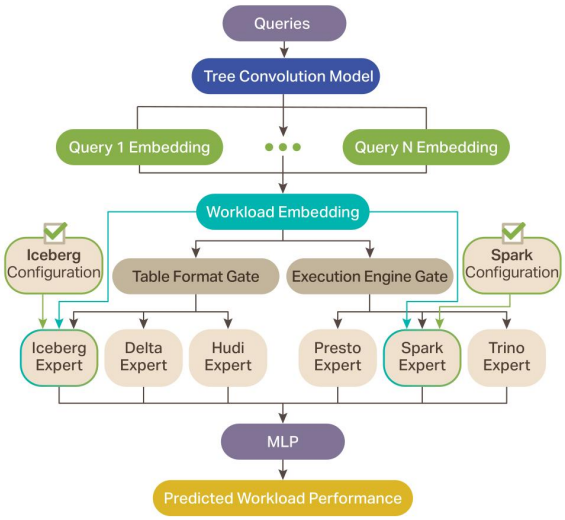


Figure 4: MoE Model Structure for LakeHelm

LakeHelm address these challenges by aligning the MoE's gates and experts with explicit subsystem boundaries. As shown in Figure 4, LakeHelm dedicates one gate per subsystem type—one for table formats and one for execution engines. Each gate selects exactly one expert, with each expert tied to a single concrete subsystem (e.g., Iceberg or Spark). When a workload embedding arrives, the gates select their respective subsystems (one for format and one for engine), and the two corresponding experts collaborate to recommend configurations for these subsystems. This customized role assignment simplifies training and enhances data efficiency while preserving the joint model's ability to capture cross-subsystem interactions through end-to-end training.

Rather than directly regressing on optimal configurations, LakeHelm adopts a proven two-step strategy from prior tuning work [15, 19, 20, 39]. Each expert functions as a surrogate model that takes a candidate configuration and outputs a latent vector. These two latent vectors (one per active expert) are fused by a multilayer perceptron (MLP) to predict overall workload latency. At inference time, after the gates select their experts, the model samples a large pool of configurations for each subsystem, scores them using the surrogate models, and returns the best choice.

We now discuss the details of LakeHelm's MoE modeling, including gate models, expert models, and the loss function.

4.1.1 Workload Representation and Gate Models. LakeHelm first converts the input workload into a feature representation using a tree convolution network, which has been successfully applied to query modeling [26, 27]. LakeHelm utilizes the query's logical plan rather than the physical plan, as accessing physical plans necessitates expensive resource allocation for metadata generation in data lake environments. LakeHelm's encoding approach employs a two-component embedding structure: [operator, operator_columns] for all logical plan nodes, where operator represents the operation type (e.g., Scan, Filter, Join, Aggregate) and operator_columns denotes the specific columns involved in that operation.

Node features are augmented with Parquet file statistics, including column histograms and row counts. LakeHelm uses operator-specific column definitions: Scan operators incorporate row counts from file metadata, while Join operators utilize cardinality estimation values derived from joined column statistics. The tree convolution model processes these enriched node features through an MLP-based kernel network. At each convolution step, the kernel combines encodings from the current node with those of its children, capturing hierarchical interactions across the query plan.

The workload representation serves as input to the gate model. The output labels of the gate models represent the chosen subsystem combination. If the model predicts that the workload will achieve the best execution time using Spark with Iceberg, the table format gate will output Iceberg, and the execution gate will output Spark.

4.1.2 Expert Models and Performance Prediction. The expert models take as input the workload representation, the gate outputs, and candidate configuration knobs. The gate outputs are integer-encoded indicators of the selected subsystems: for the engine expert, an additional column records the table format chosen by the format gate, and vice versa, allowing the experts to capture cross-subsystem interactions. The configuration knobs specify candidate subsystem-specific settings; during inference, after the gates determine the subsystem choices, LakeHelm samples a large set of configurations for each expert and evaluates them through the expert models. The two activated experts (engine and format) process these features through their specialized networks to produce

intermediate embeddings that capture subsystem behavior and interactions. These embeddings are then concatenated and passed to a post-MLP head, which outputs the final performance prediction as a ratio-based target, defined as the predicted workload execution time divided by the best execution time observed for the same workload during exhaustive tuning in the data collection process (Section 3.2.2).

4.1.3 Loss Function. During training, LakeHelm encodes each workload using the tree-convolution model to obtain an embedding that feeds into gate models to produce format and engine selection probabilities. Then the corresponding expert networks and the post-MLP make the final performance prediction. LakeHelm's training objective integrates three complementary losses to balance learning across all these components.

Cross-Entropy Loss (Gate Models). LakeHelm uses cross-entropy loss for the gate models, given their classification nature:

$$\mathcal{L}_{\text{CE}} = -\Big[\log p_{\text{format},f^*} + \log p_{\text{eng},c^*}\Big], \tag{3}$$

where p_{format,f^*} is the predicted probability of selecting the best format f^* for the given workload, and p_{eng,c^*} is the predicted probability of selecting the best engine c^*. These probabilities come from the softmax outputs of the respective gate networks. f^* and c^* are the best subsystems identified in data collection (Section 3.2.2).

Mean Squared Loss (Expert and Performance Models). LakeHelm utilizes the MSE loss for the expert and performance prediction models. The loss is weighted by the joint Gumbel-softmax probabilities of both engine and table format gates:

$$\mathcal{L}_{\text{MSE}} = (\hat{r} - r)^2 \cdot p^{\text{gumbel}}_{\text{eng},c^*} \cdot p^{\text{gumbel}}_{\text{format},f^*}, \tag{4}$$

where $p^{\text{gumbel}}_{\text{eng},c^*}$ and $p^{\text{gumbel}}_{\text{format},f^*}$ denote the Gumbel-softmax probabilities of the correct engine c^* and format f^*.

Diversity Regularization (Gate Balance). To prevent collapse where gates assign all probability mass to a single subsystem, LakeHelm enforces distributional balance across the batch:

$$\mathcal{L}_{\text{div}} = \sum_j \left(\bar{p}_{\text{format},j} - \tfrac{1}{|\text{formats}|}\right)^2 + \sum_k \left(\bar{p}_{\text{eng},k} - \tfrac{1}{|\text{engines}|}\right)^2, \tag{5}$$

where $\bar{p}_{\text{format},j}$ and $\bar{p}_{\text{eng},k}$ are average selection probabilities across all samples in the batch. This term regularizes the gate distributions and improves exploration.

Total Objective and Gradient Connections. The final objective for each workload integrates all components:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{MSE}} + \mathcal{L}_{\text{CE}} + \lambda_{\text{div}} \cdot \mathcal{L}_{\text{div}}. \tag{6}$$

During training, the gradients for each batch of workloads flow through the entire architecture to jointly optimize the tree convolution encoder, gate networks, and expert models.

4.2 Training Strategy

Despite architectural advantages for complex optimization, training our MoE design presents two challenges. First, expert starvation occurs when some experts receive insufficient samples, creating a feedback loop where the gate network increasingly avoids underutilized experts. Traditional MoE solutions [33] address this by adding auxiliary losses to encourage balanced expert utilization. However, this approach requires many more epochs for the gate to converge, which is impractical in our setting: the tree-convolution encoder is computationally expensive and prone to overfitting, causing convergence mismatches when training the gates, encoder, and experts jointly, even with additional regularization. Second, our optimization problem has exactly one best subsystem-configuration pair per workload, resulting in sparse training signals compared to conventional machine learning scenarios where multiple valid solutions exist.

To address these challenges, we design a specialized training approach with three coordinated stages in each training epoch: end-to-end training that uses the loss discussed in Section 4.1.3, gate-focused training that resolves expert starvation and convergence issues, and expert-focused training that overcomes sparse training signals. As illustrated in Figure 5, a training epoch consists of first flowing data through end-to-end training, followed by activating both gate-focused and expert-focused training stages. During the gate-focused and expert-focused stages, multiple sub-epochs are performed with different data volumes. Finally, throughout all stages, the model's best state based on validation loss serves as the initialization for subsequent stages, maintaining training stability and ensuring convergence quality across the entire pipeline.

To better illustrate how training data flows in each stage, we denote the number of workloads in the training dataset as N, the number of subsystem combinations as C, and the number of configurations per combination as M. In total, the training dataset contains $N \times C \times M$ records.

End-to-end Training: This stage integrates all components where workload embeddings produced by the tree convolution model are processed by gate networks to generate subsystem selection probabilities, which are then concatenated with configuration features and passed to experts for execution time prediction. End-to-end training uses only N samples where each workload contributes its best subsystem-configuration pair, enabling joint optimization of all model components.

Gate-focused Training: Gate-focused training uses N directly observable best subsystem choices as strong supervision. Each workload embedding is paired with its best subsystem combination, creating N training samples that are processed across multiple epochs. This approach prevents collapse into trivial solutions and accelerates convergence by separating gate training from end-to-end optimization. Additionally, LakeHelm's diversity regularization (Section 4.1.3) distributes probability mass across subsystems, reducing expert underutilization.

Expert-focused Training: As shown in Figure 5, this exposes every expert to the complete dataset within its domain. Each expert is trained individually across every workload and configuration tuple within its assigned combination, processing $N \times M$ samples in total across multiple sub-epochs. Crucially, for each expert's training, all data within its specific combination is utilized, including suboptimal and poor configurations. This comprehensive exposure enables the model to identify inferior configurations and predict correspondingly high performance ratios, ensuring accurate performance models before gate integration and comprehensive data utilization across heterogeneous subsystem distributions.

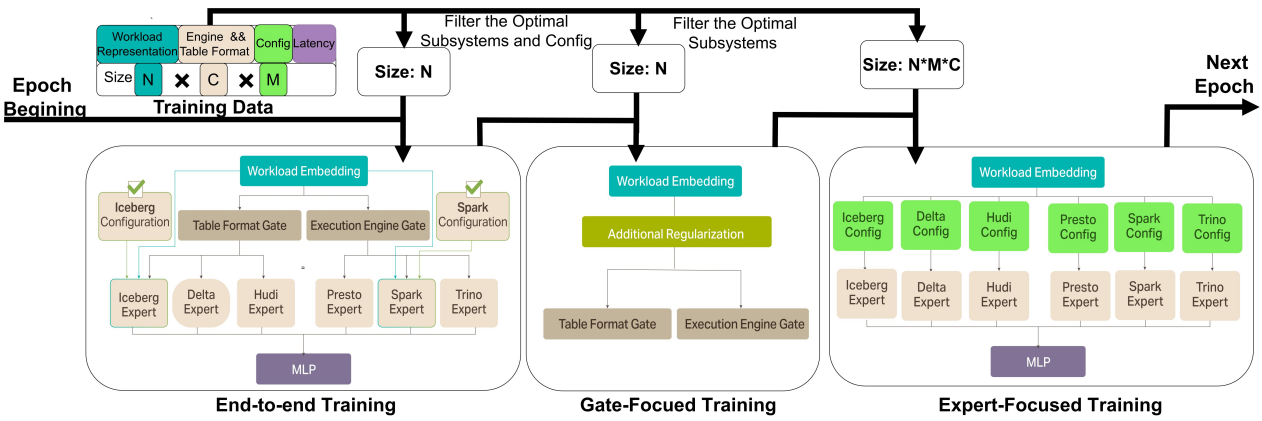


Figure 5: MoE Training with a Multi-Stage Strategy. The upper black arrows illustrate how training data flows through each training stage. Each epoch begins with end-to-end training, followed by gate-focused training that resolves expert starvation and convergence issues, and expert-focused training that overcomes sparse training signals. The next epoch starts with the saved model from the previous epoch.

5 DATA AUGMENTATION

This section presents our data augmentation methodology, which integrates two complementary approaches: (1) **LLM-generated queries**, which provide fine-grained control over query complexity and mitigate benchmark biases; and (2) **workload synthesis**, which expands limited execution traces into rich training corpora. Together, these strategies address both data scarcity and representational imbalance, enabling robust training of MoE models.

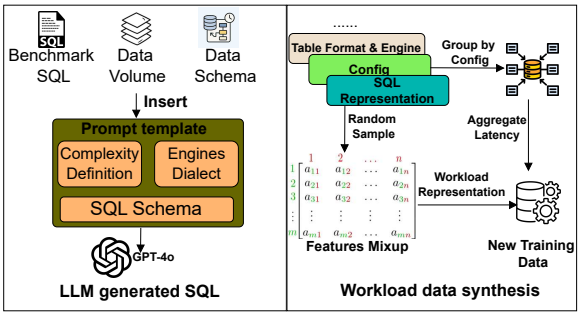


Figure 6: Proposed augmentation methods. The left panel displays LLM-generated SQL templates; the right panel shows synthesized workloads with no additional execution cost.

5.1 LLM-generated Queries

Zero-shot models must generalize across workloads, which in turn requires training data that is representative—especially near decision boundaries between subsystem combinations. Standard benchmarks (e.g., TPC-DS [35], SSB [30]) typically provide few query templates and often favor specific subsystems due to implementation choices and historical optimizations. Therefore, recent work utilizes LLMs to generate SQL templates and expand training data [15], but without explicit control over query complexity and with frequent empty result sets, limiting their generalization value.

As shown in Figure 6, LakeHelm introduces operator-level complexity controls for LLM-generated queries. We assign scores in 1,2,3 to six primary SQL operators—scan, join, aggregate, sort, project, and nested query—and define a query's total complexity as the sum of its operator scores. For example, a "simple" join (score 1) uses two tables with a primary–foreign key; a "moderate" join (score 2) spans more tables or non-key columns and may include filters; and a "complex" join (score 3) may involve three or more tables, subqueries, or non-equijoins.

LakeHelm enumerates both total-complexity tiers and per-operator levels to sample balanced combinations across the space. Prompts include schema details, data scale, and representative templates, guiding the LLM (GPT-4o) to generate queries within targeted complexity bands. This structured prompting improves syntactic correctness and dialect compatibility across engines.

To address empty outputs, LakeHelm anchors generation with validated benchmark queries and reusable subqueries, executes each candidate, and filters those yielding empty result sets. This preserves realistic execution characteristics.

The result is a balanced dataset with controllable complexity and meaningful coverage: operator-level scoring provides interpretable control, and systematic validation ensures a useful correlation between SQL complexity and runtime performance, yielding a richer, more representative training set than benchmarks alone.

5.2 Workload Synthesis

Although LLMs can increase the number of distinct queries, exhaustively executing workloads across subsystem combinations and configurations remains computationally expensive; consequently, only a limited number of workloads can be run in practice. Moreover, a single workload has exactly one best lakehouse (subsystems plus configuration), yielding an insufficient training signal for MoE.

To overcome this, we design a workload-synthesis framework that generates additional training data from execution traces without extra runs. A single exhaustive tuning pass for a workload

already measures every query under many subsystem and configuration settings. We therefore resample queries within this pass to assemble new workloads, reusing the recorded metrics for each lakehouse setting at zero additional execution cost.

Naive query sampling, however, can entrench subsystem preferences: uniform selection often favors dominant combinations (e.g., Iceberg+Spark), producing high volume but low diversity and offering little coverage of decision boundaries where the optimal choice is uncertain. We address this with *balanced sampling*: we group queries by their best subsystem combination and sample equally across groups, while increasing the selection probability of minority combinations. For training, we use 80% balanced sampling and 20% random sampling; for testing, we use fully random sampling to reflect real-world distributions.

As shown in Figure 6, each augmented data point contains three components: (1) an SQL representation capturing query semantics, (2) configuration settings defining the execution environment, and (3) execution time as the performance metric. This triplet preserves the characteristics needed for accurate performance prediction.

Concretely, LakeHelm constructs synthetic workloads by sampling many subsets of n queries executed under the same configuration. Assume it samples queries $q_1, \ldots, q_n$ with corresponding execution times $t_1, \ldots, t_n$. The workload feature is a weighted sum, $\sum_{i=1}^{n} \alpha_i \cdot \text{feature}(q_i)$ and the aggregated execution time is $\sum_{i=1}^{n} \alpha_i \cdot t_i$, where coefficients α_i control each query's contribution. This formulation aligns synthesized features with measured performance, turning a small set of expensive trials into a large pool of labeled workloads. It also preserves ground-truth fidelity and provides the MoE with both data volume and variety for efficient offline training.

6 INFERENCE WITH LAKEHELM

After training, LakeHelm generates subsystem and configuration recommendations through a three-step inference process.

In the first step, LakeHelm extracts vectorized representations for unseen workloads. It uses logical plans from Spark and augments them with statistical metadata, including column histograms and row counts from Parquet files. These inputs are fed into the trained tree model to produce SQL-level features, which are then aggregated to obtain the overall workload feature vector F_W.

Next, the two gating networks (engine and format) each process the workload features F_W and output a probability distribution over available subsystems, selecting the one with the highest probability: $s^* = \arg\max_i \text{softmax}(\mathcal{G}(F_W))_i$.

Finally, LakeHelm samples the configuration space, which contains both discrete knobs (*e.g.*, number of partitions) and continuous knobs (*e.g.*, memory allocation ratios). For continuous knobs, LakeHelm defines bounded ranges and discretizes them to create an enumerable configuration space $C = C_1 \times C_2 \times \cdots \times C_n$. The corresponding expert models then evaluate different configurations and recommend the one that minimizes predicted execution time: $c^* = \arg\min_{c \in C} \mathcal{E}_{s^*}(F_W, c)$, which is indexed back to the configuration space to obtain the best configuration.

7 EVALUATION

7.1 Experiment Setup

7.1.1 Datasets. To evaluate LakeHelm across diverse data scales and workloads, we use five widely adopted analytical benchmarks: TPC-H [36], TPC-DS [35], JOB [40], SSB [30], and SSB-Flat [30]. We also vary each benchmark across three scale factors, `sf={1, 10, 100}`, representing small, medium, and large data sizes. Note that JOB is evaluated only at `sf=10GB`, since its dataset is fixed in size and does not support scalable data generation across scale factors.

As discussed in Section 3.2.2, we perform exhaustive tuning (up to 50 iterations) on both original and LLM-generated queries to collect execution data for training and evlauation. For each benchmark and scale factor, we tune configurations for all SQL queries across 9 subsystem combinations, including three query engines (Spark, Presto, and Trino) and three table formats (Hudi, Iceberg, and Delta). The configuration vector contains 12 parameters: 8 for the compute engine (such as Spark's executor cores, executor memory, and parallelism settings) and 4 for the table format (such as Delta's file size).

The total number of augmented workloads in our experiments is approximately 3,000, resulting in around 324,000 execution samples across all benchmarks, scale factors, subsystem combinations, and configurations.

7.1.2 Baselines. We compare LakeHelm against these baselines:

Baseline #1 (Best-Fixed): We employ an overall best fixed subsystem combination(Trino + iceberg) and configuration. Specifically, we select the combination that achieves the best performance most frequently in our collected execution data and use the configuration knobs that demonstrate the best average performance across the data.

Baseline #2 (GPT-5): We use GPT-5 [31] to directly recommend the subsystem combination and configuration in a zero-shot manner; the suggested settings are applied once without any feedback.

Baseline #3 (GPT-4o): We use GPT-4o [31] under the same zero-shot protocol as GPT-5.

Baseline #4 (Two-Stage): We compare LakeHelm's MoE model with a two-stage modeling approach: the first model recommends subsystem combinations, and the second recommends configurations for the selected subsystems. We implement the first model using decision trees and neural networks (NNs) for classification, and the second model using NNs trained separately as regressors.

Baseline #5 (λ-Tune): λ-Tune[13] is an online tuning method that iteratively executes the target workload, observes feedback, and updates configuration choices across multiple rounds, resulting in substantially higher tuning overhead at larger scale factors. Since changing subsystem combination online is infeasible, we employ λ-Tune for each test workload with the overall best-performing subsystem combination observed in our collected data.

7.1.3 Experiment Methodology. We evaluate LakeHelm and baselines in a zero-shot setting. For each experiment, we select one benchmark as the target benchmark and exclude *all* execution data from this benchmark from the training set. We then train LakeHelm with a training dataset that aggregates execution data from the remaining benchmarks across all scale factors (`sf={1, 10, 100}`).

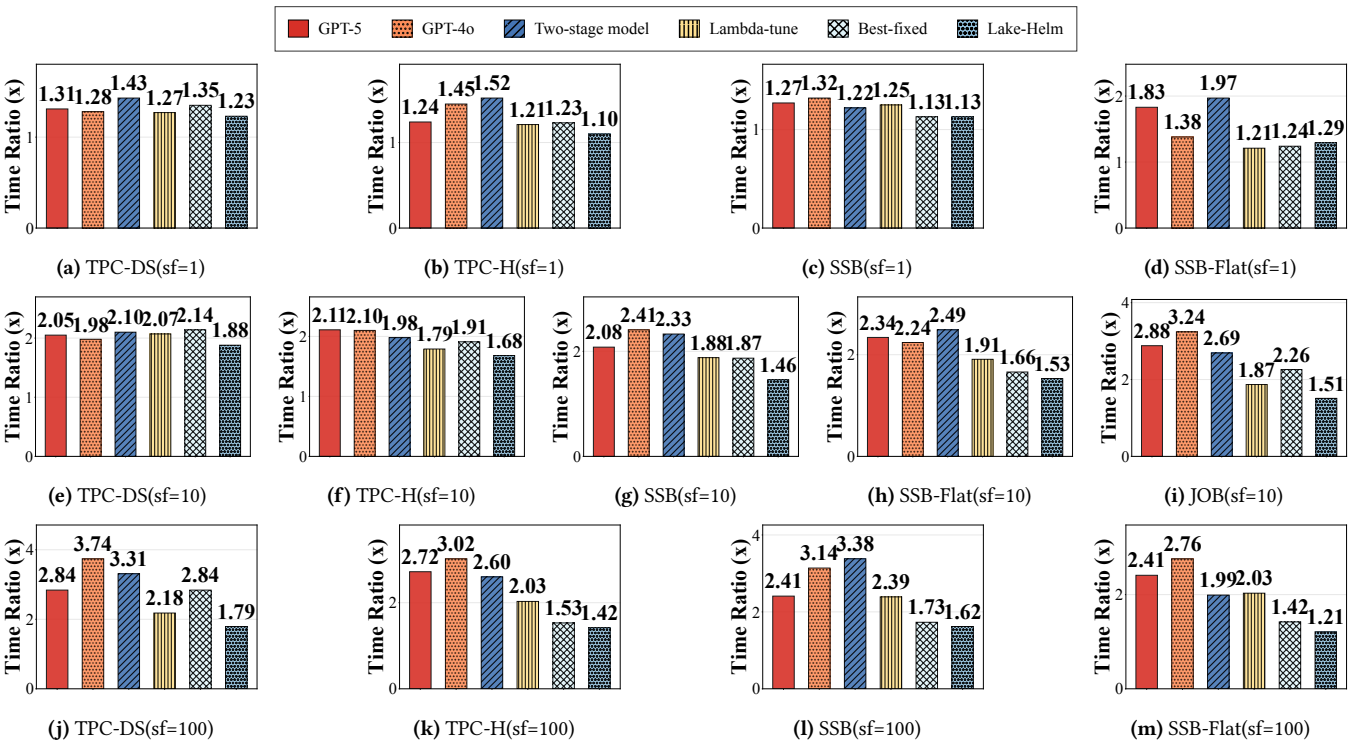


Figure 7: Zero-shot performance evaluation of baseline performance under different scale factors. Rows correspond to scale factors of 1GB, 10GB, and 100GB, while columns represent different benchmarks.

For each benchmark, we evaluate the lakehouse recommendations only on the *original benchmark queries* and report results separately for each scale factor. However, treating each benchmark as a single workload yields too few test points. To broaden coverage, we adopt a workload augmentation strategy similar to Section 5.2: for each benchmark, we use Bagging [6] to generate 10 workloads by sampling 10%, 20%, …, 100% of the original benchmark queries.

All methods make recommendations per sampled workload. We then report the average result across all 10 sampled workloads per benchmark and scale factor. For LakeHelm and Two-Stage, after the gate models/classifiers first recommend the subsystem combination, they uniformly sample 50,000 configurations, use the expert models/regressors to predict execution time, and return the configuration with the minimum predicted time as the recommendation.

To speed up evaluation, we map each recommended subsystem combination and configuration to the nearest executed configuration in our collected execution data using a normalized (0–1) distance metric over the configuration space. On average, the nearest neighbor is within 0.03, and we observe negligible latency differences at this granularity. We therefore use the latency of the nearest executed configuration as the recommendation's realized latency.

7.1.4 Metrics. We use **Time Ratio** as the primary evaluation metric. For each workload, we compute the ratio of the execution time under the recommended subsystem combination and configuration to the best observed execution time from our collected execution data under exhaustive tuning (lower is better; the optimum is 1.0).

7.1.5 Computational Resources. All experiments were conducted on the University of Michigan Great Lakes Cluster [38] using CPU nodes with dual Intel Xeon Gold 6154 processors. Data staging used 3 lightweight nodes, while execution-time profiling used a 4-nodes cluster. Each query engine was allocated identical resources (4 nodes, 8 cores and 64 GB RAM per node), and model training was performed on a single NVIDIA RTX 4090 GPU.

7.2 Main Results

7.2.1 Zero-Shot Performance Evaluation. We present our main results via a zero-shot evaluation across five benchmarks. Figure 7 reports **time ratio** under different scale factors.

Overall performance and scale-stratified results. Across all reported test points (TPC-DS/TPC-H/SSB/SSB-Flat at `sf=1/10/100GB` plus JOB at `sf=10GB`), LakeHelm achieves the best overall performance with an average time ratio of **1.45**. The strongest baseline across all settings is **Best-Fixed**, with an average time ratio of **2.03**, yielding a **1.40**× overall improvement.

This advantage is consistently observed across all scale regimes when compared against the *best-performing baseline at each scale*. At `sf=1GB`, the strongest baseline is λ-**Tune** (**1.23**), while LakeHelm achieves a lower average ratio of **1.19**. At `sf=10GB`, λ-**Tune** again emerges as the best baseline (**1.90**), whereas LakeHelm further improves to **1.61**. At `sf=100GB`, the best-performing baseline shifts to **Best-Fixed** (**1.95**), yet LakeHelm remains stable at **1.51**, demonstrating robust performance under large-scale execution.

Beyond average performance, LakeHelm also demonstrates strong worst-case robustness at large scale. At sf=100GB, LakeHelm stays below a time ratio of **1.80** on three out of four benchmarks (TPC-DS, TPC-H, and SSB-Flat). In contrast, the strongest baseline at this scale, **Best-Fixed**, exceeds **2.80** on TPC-DS and remains above **1.50** on all benchmarks. This indicates that LakeHelm not only improves mean performance, but also substantially reduces large-scale performance degradation in unfavorable execution regimes.

Key insights. Based on the main results above, we distill the following five key insights that explain where performance gaps arise and how they evolve across data scales and workloads.

Key insight #1 (small-scale saturation). At sf=1GB, performance differences between methods are relatively small: LakeHelm averages **1.19**, while Best-Fixed, λ-Tune, GPT-4o, and GPT-5 average **1.24**, **1.23**, **1.36**, and **1.41**, respectively. This "small-gap" regime is expected because small datasets often fit in memory and incur limited shuffle/spill pressure, so the penalty of suboptimal configurations is muted and subsystem-combination choice dominates.

Key insight #2 (scale amplifies configuration sensitivity). As the scale increases, the gap widens substantially, and joint configuration tuning across subsystems becomes critical. This trend is reflected in the relative degradation of baselines: λ-Tune exhibits the smallest performance degradation as scale grows, and GPT-5, despite higher reasoning cost, often recommends better configurations (knobs) than other LLM-based baselines, even though it does not consistently identify better subsystem combinations.

Key insight #3 (query complexity drives tuning upside). For benchmarks dominated by simpler SQL (fewer joins, predictable scans), configuration changes yield smaller marginal gains because the execution path is stable and less likely to trigger expensive shuffles/spills. For example, SSB at sf=1GB shows minimal separation (LakeHelm **1.13** vs. Best-Fixed **1.13**), indicating that once a reasonable subsystem is chosen, further knob tuning provides diminishing returns. In contrast, complex workloads with deeper join trees and heavier shuffles (notably TPC-DS and JOB) show much larger gains from workload-aware configuration tuning, especially at sf=10/100GB.

Key insight #4 (scale magnifies subsystem combination gaps). In most cases, increasing the scale factor makes the leading engine–format combination stand out more clearly, resulting in a larger and more stable performance margin. As data grows, bottlenecks such as memory pressure, spill, shuffle, and I/O saturation become dominant, which amplifies performance differences and allows the strongest combination to establish a clearer lead. For a subset of queries, however, scale growth can also change the dominant combination: e.g., many TPC-DS queries shift from favoring Trino at small scale to Spark at larger scale, reflecting different robustness under scale-triggered shuffle and memory pressure.

Key insight #5 (GPT-5 vs. GPT-4o is scale-dependent). GPT-5 is consistently better than GPT-4o at large scale factors (e.g., sf=100GB: **2.60** vs. **3.17**), suggesting stronger ability to propose configurations that mitigate memory/shuffle pressure. At small scale, their performance is closer because subsystem-combination choice dominates and the upside of better configuration reasoning is limited.

Method	LakeHelm	Two-Stage	GPT-5	GPT-4o
Inference Time	**58.5 s**	47.9 s	507 s	146 s
λ-Tune (sf=1/10/100GB)	~1 h / ~3 h / ~8 h			

Table 1: Average inference (or tuning) time comparison. λ-Tune shows scale-dependent tuning overhead, while other methods provide near constant inference latency.

System-level interpretation (compute engines and lake formats). A concrete example of the scale-driven dynamics above is the Trino–Spark reversal on TPC-DS with Iceberg: Trino+Iceberg performs better at sf=1GB, yet Spark+Iceberg becomes superior at sf=100GB. This occurs because Spark provides more explicit control over shuffle scheduling and spill tolerance, allowing it to better handle large data movement under shuffle-intensive joins, while Iceberg's metadata pruning further amplifies these effects at large scale.

7.2.2 Time Cost. Table 1 compares the inference/tuning time of different methods. Zero-shot methods (LakeHelm/Two-Stage/GPT-5/GPT-4o) provide near-constant inference latency (10s/100s seconds), while λ-Tune incurs scale-dependent overhead (hours at sf=10/100GB). Furthermore, λ-Tune tunes only under a fixed subsystem combination, since switching subsystems online is highly challenging and often impractical.

7.3 Ablation Study

To comprehensively evaluate the contribution of each component in our framework, we conducted systematic ablation studies to verify the effectiveness of our proposed methods.

7.3.1 Ablation Study on Training Strategy and Inference Strategy. We first study the impact of different training strategies in LakeHelm by selectively removing gate-focused training, expert-focused training, or both. Table 2 summarizes the results under zero-shot evaluation across scale factors. In addition, we include a variant that removes the *dual-gate* design, where the two-layer routing is replaced by a single unified gate.

Removing expert-focused training leads to only minor degradation at small and medium scales. At sf=1GB and sf=10GB, the time ratio increases modestly across all benchmarks (e.g., TPC-DS from **1.23** to **1.33** at sf=1GB, and from **1.88** to **2.12** at sf=10GB). However, the performance drop becomes *significantly amplified* at sf=100GB, where expert removal degrades TPC-DS from **1.79** to **3.19**, and SSB-Flat from **1.21** to **2.97**. This trend indicates that expert specialization is particularly critical at large scale, where configuration-level decisions strongly interact with memory pressure, shuffle behavior, and spill dynamics.

In contrast, removing gate-focused training consistently degrades performance across *all* scale factors. Even at sf=1GB, time ratios increase noticeably (e.g., TPC-DS rises from **1.23** to **1.51**), and the degradation persists and compounds at sf=10GB and sf=100GB. This behavior highlights that incorrect or unstable routing among subsystem combinations cannot be compensated by configuration tuning alone, and that accurate combination selection is a prerequisite for effective end-to-end optimization.

Method	TPC-DS			TPC-H			SSB			SSB-Flat			JOB
	1GB	10GB	100GB	1GB	10GB	100GB	1GB	10GB	100GB	1GB	10GB	100GB	10GB
LakeHelm	1.23	1.88	1.79	1.10	1.68	1.42	1.13	1.46	1.62	1.29	1.53	1.21	1.51
- Expert Focused	1.33	2.12	3.19	1.41	1.77	2.67	1.23	2.01	2.81	1.39	2.16	2.97	2.27
- Gate Focused	1.51	2.80	2.71	1.63	1.82	2.16	1.47	1.81	2.32	1.42	2.74	2.58	2.81
- Both Focused	1.62	2.91	3.36	1.80	1.87	2.66	1.58	2.11	2.49	1.67	2.80	2.73	2.93
- Double Gate	1.49	2.31	2.42	1.40	1.66	2.08	1.31	1.72	1.83	1.38	1.77	1.45	2.16
- Inference Strategy	1.54	2.35	4.12	1.64	1.91	3.11	1.63	1.86	2.79	1.52	1.82	2.38	4.11

Table 2: Ablation study on LakeHelm's training and inference strategies. Results are reported as **Time Ratio** at `sf=1GB, 10GB, 100GB`.

Method	TPC-DS	TPC-H	SSB	SSB-Flat	JOB
LakeHelm	1.88	1.68	1.46	1.53	1.51
- Gen SQL	2.40	1.86	1.71	2.34	2.66
- Workload Syn	2.51	2.13	2.01	2.51	3.13
+ SQL Storm	1.96	1.78	1.52	1.43	1.88

Table 3: Ablation study on data augmentation strategies. All results are reported at a fixed `sf=10GB` (Time Ratio; lower is better).

When both focused training strategies are removed, performance deteriorates the most across all benchmarks and scales. The resulting time ratios approach or exceed those of strong baselines, confirming that LakeHelm relies on *both* accurate combination selection (gate) and fine-grained configuration modeling (experts) to achieve robust performance.

Beyond the sample-and-rank inference paradigm, we also evaluate directly predicting a concrete configuration at inference time, allowing LakeHelm to output a single end-to-end configuration without candidate enumeration. However, direct configuration prediction is inherently challenging due to the high-dimensional configuration space and sharp performance cliffs, which significantly hinder training convergence and often lead to extreme or poorly calibrated predictions. These observations motivate our inference strategy (Section 6) to stabilize learning and improve robustness.

7.3.2 Ablation Study on Data Augmentation. We next evaluate the effect of different data augmentation strategies, with results reported in Table 3. All experiments in this section use the full LakeHelm training and inference strategy.

Removing either generated SQL queries or workload synthesis substantially degrades performance across all benchmarks. For example, removing generated SQL increases the time ratio on TPC-DS from **1.88** to **2.40**, and on SSB-Flat from **1.53** to **2.34**. Removing workload synthesis leads to even larger degradation, indicating that capturing workload-level structure and cross-query interactions is critical for generalization.

Replacing our LLM-generated SQL queries with SQL Storm yields inferior performance in most (4/5) cases, though the performance of SQL Storm is competitive. Across all benchmarks, SQL Storm reduces time ratios substantially compared to the "no Gen SQL" setting (e.g., TPC-DS improves from **2.40** to **1.96**), and its average time ratio is close to LakeHelm (SQL Storm: ~**1.71** vs. Ours: ~**1.61**).

However, as shown in Table 4, LakeHelm attains this performance with *lower acquisition cost* and *more controllable workload quality*. For a fair comparison, we generate approximately **200 SQL queries using SQLStorm**, so that the number of *non-empty* SQLs from SQLStorm is comparable to that produced by our method.

Instead of relying on large-scale free-form SQL generation, LakeHelm imposes finer-grained complexity control to systematically cover different join depths and operator patterns, enabling more balanced structural coverage and reducing variance caused by extreme or degenerate queries.

Table 4 provides a workload-level view of how different SQL sources populate structurally diverse regimes that are relevant for lakehouse evaluation, reporting non-empty template coverage, join complexity distribution, and summary statistics on referenced tables and predicates. While the latter metrics are included for completeness, our analysis primarily focuses on *join complexity*, as it captures the dominant dimension of structural diversity across workloads. As shown in the table, different SQL sources exhibit markedly different coverage profiles: LakeHelm produces a more diverse and balanced join complexity distribution compared to SQLStorm, which tends to concentrate queries within a narrower join-depth range. In conclusion, our generated workloads exercise a broader spectrum of execution regimes during training, which is essential for learning robust, workload-aware recommendations, particularly at larger scale factors such as `sf=100GB`.

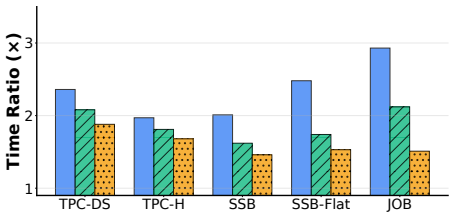


Figure 8: Effects of training data scale on LakeHelm performance across three scales (1/3, 2/3, and complete dataset).

7.3.3 Ablation study on scale of training data. We delve deeper into the effects of training data scale on the quality of configurations recommended by the LakeHelm. Following the evaluation settings outlined in Section 7.1.4, we modify the scale of the training dataset for our analysis. Our experiments are conducted across three different scales: 1/3, 2/3, and the complete training dataset, with results

Metric	TPC-DS			TPC-H			SSB			SSB-flat			JOB		
	Bench	SQLStorm	Ours	Bench	SQLStorm	Ours	Bench	SQLStorm	Ours	Bench	SQLStorm	Ours	Bench	SQLStorm	Ours
Non-empty / Distinct templates	106/106	103/200	124/150	22/22	105/200	116/150	13/13	92/200	128/150	13/13	92/200	120/150	120/120	108/200	120/150
Join complexity distribution (L/M/H)	103/0/3	48/50/5	41/41/42	22/0/0	5/61/39	38/38/40	13/0/0	36/52/4	42/42/44	13/0/0	92/0/0	120/0/0	118/2/0	21/73/14	40/40/40
Referenced Tables per SQL (Min / Max / Avg)	3/12/7.5	3/11/7.4	3/12/8.2	2/8/4.5	6/12/9.3	6/13/10.2	2/5/3.0	4/10/6.8	4/11/7.6	1/1/1.0	1/1/1.0	1/1/1.0	4/15/8.0	5/13/8.6	5/14/9.8
Predicates per SQL (Min / Max / Avg)	4/25/12.0	4/14/8.3	4/16/9.7	2/12/6.0	6/19/11.3	5/20/12.6	3/15/8.0	3/13/7.9	3/14/8.8	0/10/4.0	0/15/8.0	0/16/9.2	6/40/18.0	4/17/9.3	2/18/10.5

Table 4: The **join complexity distribution (L/M/H)** column reports the *number of SQL templates* whose join counts fall into low (0–3 joins), medium (4–6 joins), and high (7+ joins) categories. For referenced tables and predicates, we report **min / max / avg.** per SQL template.

visualized in Figure 8. The figure demonstrates the progressive improvement in time ratio as training data scale increases. The findings illustrate a notable positive relationship between training data size and the enhancement in configuration quality suggested by the LakeHelm. Time ratio shows consistent improvements across all scales, with the trend persisting even when utilizing the entire training set. This suggests that LakeHelm performance can be further optimized with an expanded training dataset, indicating that the method has not yet reached performance ceiling.

7.4 Experimental Cost Evaluation

This section analyzes the computational and financial costs of our system. camera

7.4.1 Workload Generation Cost. The workload generation process is *scale-factor agnostic*: SQL templates are generated once per benchmark and reused across all scale factors (`sf=1/10/100GB`).

We employ GPT-4o to generate additional SQL templates to augment benchmark workloads. Across all benchmarks, SQL generation required approximately 17 hours and incurred a total cost of about $20 for 200 templates per benchmark. Workload synthesis via linear interpolation requires only a few minutes and introduces negligible overhead.

7.4.2 Training Data Collection Cost. Training data collection comprises two sequential stages: data loading and iterative latency collection. For both stages, we utilized the computational infrastructure detailed in Section 7.1. The data loading phase required 30 hours to load all benchmark tables with Spark across different table formats, resulting in a total storage footprint of 402 GB across all datasets. The iterative execution metrics collection stage required roughly 450 hours of computation time for each benchmark across all subsystem combinations. This phase systematically executed queries to gather performance metrics across the full configuration space explored by our Gaussian process surrogate model

7.4.3 Model Training Cost. Model training is performed on a single NVIDIA RTX 4090 GPU. The complete training process requires 12–18 hours depending on the training strategy.

8 RELATED WORK

Prior research relevant to our problem spans two key areas. First, studies on open table formats and query engine optimization examine how format policies and engine configurations shape performance, but typically treat subsystems in isolation [17, 23, 25, 42, 43, 49]. Second, data augmentation approaches broaden training coverage to improve generalization. Alternative strategies employ feature amplification techniques and LLM-based query generation to synthesize diverse, representative workloads cost-effectively [12, 48].

Query engine and data layout optimization. There is a long line of research on automatic tuning for query engines. Early work such as OtterTune [39], CDBTune [45], and QTune [22] framed tuning as a configuration search problem, applying Bayesian optimization or reinforcement learning to reduce manual knob setting. More recent approaches, including UDO [41] and LlamaTune [19], focus on unifying logical and physical tuning decisions or reducing sample complexity for faster convergence. At the same time, commercial and open-source engines (e.g., Spark) integrate built-in techniques such as cost-based optimization, adaptive query execution, and runtime knob tuning to improve robustness [44], but they generally assume the underlying data layout is static. Beyond query engine tuning, research has also explored data layout and compaction strategies. Notably, Starfish [14], though designed for Hadoop, demonstrated the value of learned resource allocation models for large-scale data pipelines, highlighting the importance of joint optimization across system layers. However, in lakehouses, engine optimizers assume fixed data layout while layout optimizers treat query engine as a black box, leading to underexplored cross-subsystem interactions such as the interplay between Iceberg's split size and Spark's shuffle behavior.

Data augmentation. To synthesize more representative workloads, diversifying data variety rather than simply increasing volume is crucial. Alternative augmentation strategies have emerged that employ feature amplification techniques. These approaches synthesize workloads cost-effectively by interpolating or extrapolating existing workload characteristics, thereby avoiding expensive plan generation or query execution requirements. More recently, large language models (LLMs) have shown significant promise for workload synthesis. Research has demonstrated LLMs' capability for generating semantically diverse SQL queries [12]. Additionally, PBench [48] combines traditional benchmark components with GPT-enhanced query generation to approximate target performance statistics through example-based prompting.

9 CONCLUSION

This paper explores an innovative method for end-to-end zero-shot advisory services that recommend both subsystem combinations and corresponding configurations. To achieve this goal, we introduce a novel MoE pipeline with a specialized training strategy and a data augmentation method. Through extensive experiments on five benchmarks and ablation studies, we demonstrate improvements in both latency and choice accuracy at an acceptable cost.

ACKNOWLEDGMENTS

This work was supported (in part) by the Google ML and Systems Junior Faculty Award, the Google JAX AI Stack Research Award, and the NVIDIA Academic Grant Program Award.

REFERENCES

[1] Apache Hudi Project. [n.d.]. Apache Hudi: Upserts, Deletes and Incremental Processing on Big Data. https://hudi.apache.org. Accessed: 2025-07-29.

[2] Apache Iceberg Project. [n.d.]. Apache Iceberg: Open Table Format for Huge Analytic Datasets. https://iceberg.apache.org. Accessed: 2025-07-29.

[3] Apache Parquet Project. 2025. Apache Parquet: an open source, column-oriented data file format. https://parquet.apache.org/. Accessed: 2026-04-01.

[4] Apache Spark Project. 2025. Unified engine for large-scale data analytics. https://spark.apache.org/. Accessed: 2026-04-01.

[5] Michael Armbrust, Tathagata Das, Shixuan Zhu, Reynold S. Xin, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms Unifying Data Warehousing and Advanced Analytics. In *11th Annual Conference on Innovative Data Systems Research (CIDR)*.

[6] Leo Breiman. 1996. Bagging predictors. *Machine learning* 24, 2 (1996), 123–140.

[7] Baoqing Cai, Yu Liu, Lin Ma, Pingqi Huang, Bingcheng Lian, Ke Zhou, Jia Yuan, Jie Yang, Xiaofan Cai, and Peijun Wu. 2025. SCompression: Enhancing Database Knob Tuning Efficiency Through Slice-Based OLTP Workload Compression. *Proceedings of the VLDB Endowment* (2025).

[8] Jesús Camacho-Rodríguez, Ashvin Agrawal, Anja Gruenheid, Ashit Gosalia, Cristian Petculescu, Josep Aguilar-Saborit, Avrilia Floratou, Carlo Curino, and Raghu Ramakrishnan. 2024. LST-Bench: Benchmarking Log-Structured Tables in the Cloud. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–26.

[9] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv preprint arXiv:2501.12948. Accessed: 2025-01-22.

[10] Delta Lake Project. [n.d.]. Delta Lake Documentation. https://delta.io. Accessed: 2025-07-29.

[11] William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research* 23, 120 (2022), 1–39.

[12] 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).

[13] Victor Giannakouris and Immanuel Trummer. 2025. λ-Tune: Harnessing Large Language Models for Automated Database System Tuning. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[14] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A self-tuning system for big data analytics. In *5th Biennial Conference on Innovative Data Systems Research (CIDR 11)*. 261–272.

[15] Xinmei Huang, Haoyang Li, Jing Zhang, Xinxin Zhao, Zhiming Yao, Yiyan Li, Tieying Zhang, Jianjun Chen, Hong Chen, and Cuiping Li. 2025. E2etune: End-to-end knob tuning via fine-tuned generative language model. *Proceedings of the VLDB Endowment* (2025).

[16] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. 1991. Adaptive mixtures of local experts. *Neural computation* 3, 1 (1991), 79–87.

[17] Paras Jain, Peter Kraft, Conor Power, Tathagata Das, Ion Stoica, and Matei Zaharia. 2023. Analyzing and Comparing Lakehouse Storage Systems.. In *CIDR*.

[18] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. arXiv preprint arXiv:2401.04088. https://doi.org/10.48550/ARXIV.2401.04088 Accessed: 2025-07-13.

[19] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: sample-efficient DBMS configuration tuning. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2953–2965.

[20] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2025. GPTuner: An LLM-Based Database Tuning System. *ACM SIGMOD Record* 54, 1 (2025), 101–110.

[21] Viktor Leis, Andrey Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really?. In *Proceedings of the VLDB Endowment*, Vol. 9. VLDB Endowment, 204–215.

[22] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[23] Yang Li, Huaijun Jiang, Yu Shen, Yide Fang, Xiaofeng Yang, Danqing Huang, Xinyi Zhang, Wentao Zhang, Ce Zhang, Peng Chen, et al. 2023. Towards general and efficient online tuning for spark. *arXiv preprint arXiv:2309.01901* (2023).

[24] Wan Shen Lim, Lin Ma, William Zhang, Matthew Butrovich, Samuel Arch, and Andrew Pavlo. 2024. Hit the Gym: Accelerating Query Execution to Efficiently Bootstrap Behavior Models for Self-Driving Database Management Systems. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3680–3693.

[25] Chenghao Lyu, Qi Fan, Philippe Guyard, and Yanlei Diao. 2024. A Spark Optimizer for Adaptive, Fine-Grained Parameter Tuning. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3565–3579.

[26] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD*. 1275–1288. https://doi.org/10.1145/3448016.3452838

[27] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-structured deep neural network models for query performance prediction. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1733–1746.

[28] Patrick O'Neil, Betty O'Neil, and Xuedong Chen. 2009. The star schema benchmark and augmented fact table indexing. In *Performance Engineering and Optimization Techniques for Database Systems*. Springer, 237–252.

[29] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. Star Schema Benchmark - Flattened Version. Flattened variant of SSB used in analytical database research.

[30] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. *The Star Schema Benchmark and Augmented Fact Table Indexing*. Springer-Verlag, Berlin, Heidelberg, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17 Accessed: 2026-04-01.

[31] OpenAI. 2025. ChatGPT. https://chat.openai.com/. Large language model; accessed September 28, 2025.

[32] Presto Project. 2025. Query Everything. Fast. Open Source. https://prestodb.io/. Accessed: 2026-04-01.

[33] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538* (2017).

[34] Transaction Processing Performance Council. 2014. *TPC Benchmark H (Decision Support) Revision 2.17.1*. Technical Report. Transaction Processing Performance Council. http://www.tpc.org/tpch/ Accessed: 2026-04-01.

[35] Transaction Processing Performance Council. 2021. TPC-DS: TPC Benchmark DS (Decision Support) Standard Specification. http://www.tpc.org/tpcds/ Available at: http://www.tpc.org/tpcds/. Accessed: 2026-04-01.

[36] Transaction Processing Performance Council. 2021. TPC-H: TPC Benchmark H (Ad-Hoc, Decision Support) Standard Specification. http://www.tpc.org/tpch/ Available at: http://www.tpc.org/tpch/. Accessed: 2026-04-01.

[37] Trino Project. 2025. Trino, a query engine that runs at ludicrous speed. https://trino.io/. Accessed: 2026-04-01.

[38] University of Michigan Information and Technology Services. 2026. Great Lakes High Performance Computing Cluster. https://its.umich.edu/advanced-research-computing/high-performance-computing/great-lakes. Accessed: 2026-04-01.

[39] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.

[40] Viktor Leis and Andrey Gubichev and Atanas Mirchev and Peter Boncz and Alfons Kemper and Thomas Neumann. 2015. Join Order Benchmark (JOB).

https://github.com/gregrahn/join-order-benchmark. Accessed: 2026-04-01.

[41] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: universal database optimization using reinforcement learning. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3402–3414.

[42] Yixin Wu, Xiuqi Huang, Zhongjia Wei, Hang Cheng, Chaohui Xin, Zuzhi Chen, Binbin Chen, Yufei Wu, Hao Wang, Tieying Zhang, Rui Shi, Xiaofeng Gao, Yuming Liang, Pengwei Zhao, and Guihai Chen. 2024. Towards Resource Efficiency: Practical Insights into Large-Scale Spark Workloads at ByteDance. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3759–3771. https://doi.org/10.14778/3685800.3685804

[43] Jinhan Xin, Kai Hwang, and Zhibin Yu. 2022. Locat: Low-overhead online configuration auto-tuning of spark sql applications. In *Proceedings of the 2022 International Conference on Management of Data*. 674–684.

[44] Runhui Xue, Yuxin Bu, Ayush Somani, Hongyu Fan, Reynold Xin, and Matei Zaharia. 2024. Adaptive and Robust Query Execution for Lakehouses at Scale. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3947–3960. https://www.vldb.org/pvldb/vol17/p3947-bu.pdf Accessed: 2026-04-01.

[45] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*. 415–432. https://doi.org/10.1145/3299869.3300085

[46] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating database tuning with hyper-parameter optimization: a comprehensive experimental evaluation. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1808–1821.

[47] Xinyi Zhang, Hong Wu, Yang Li, Zhengju Tang, Jian Tan, Feifei Li, and Bin Cui. 2023. An efficient transfer learning based configuration adviser for database tuning. *Proceedings of the VLDB Endowment* 17, 3 (2023), 539–552.

[48] Yan Zhou, Chunwei Liu, Bhuvan Urgaonkar, Zhengle Wang, Magnus Mueller, Chao Zhang, Songyue Zhang, Pascal Pfeil, Dominik Horn, Zhengchun Liu, et al. 2025. PBench: Workload Synthesizer with Real Statistics for Cloud Analytics Benchmarking. *arXiv preprint arXiv:2506.16379* (2025).

[49] Yiwen Zhu, Rathijit Sen, Brian Kroth, Sergiy Matusevych, Andreas Mueller, Tengfei Huang, Rahul Challapalli, Weihan Tang, Xin He, Mo Liu, Estera Kot, Sule Kahraman, Arshdeep Sekhon, Dario Bernal, Aditya Lakra, Shaily Fozdar, Dhruv Relwani, Rui Fang, Long Tian, Karuna Krishna, Ashit Gosalia, Carlo Curino, and Subru Krishnan. 2025. Rockhopper: A Robust Optimizer for Spark Configuration Tuning in Production Environment. In *Companion of the 2025 International Conference on Management of Data (SIGMOD '25 Companion)*. 743–756. https://doi.org/10.1145/3722212.3724451