AutoSLO: Practical Latency SLOs on Cloud Data Warehouses – Extended Version

Markos Markakis
MIT CSAIL
Cambridge, MA, USA
markakis@mit.edu

Tim Kraska
MIT CSAIL
Cambridge, MA, USA
kraska@mit.edu

ABSTRACT

Modern cloud data warehouses decouple compute from storage, making it easy for organizations to access the same underlying data with multiple compute clusters. This flexibility is often used for performance isolation among diverse workloads, so that each workload meets its latency service-level objective (SLO) more reliably. For example, interactive dashboards, ad hoc analysis, and batch jobs can each run on separate clusters. However, this dedicated-cluster approach requires each compute cluster to be continuously scaled to adapt to workload evolution, with over-provisioning wasting resources and under-provisioning risking SLO violations.

We present AutoSLO, a latency-SLO-aware workload management framework for multi-cluster cloud data warehouses. AutoSLO operates across three timescales through three key components. First, a periodic Policy Tuner plans proactive cluster scaling actions and tunes configuration parameters, using simulations of history-derived workload forecasts. Second, an SLO-aware reactive Autoscaler adjusts the active cluster set when recent workload behavior deviates from the forecast. Third, an online Query Router reacts to live load when placing each query, using a concurrency-aware latency predictor to avoid SLO violations.

On realistic Redbench workloads, AutoSLO successfully meets latency SLOs of varying strictness, reducing cost by a mean of 26.4% compared to the per-scenario next-best baseline. Component-level evaluations show that the Query Router and Autoscaler respectively reduce SLO violation rates by a mean of 47.8% and 93.7%, relative to their corresponding alternatives. Finally, we show that the Policy Tuner can reduce the SLO violation rate by a mean of 44.6% using a single day of workload history, and that each component is efficient given its intended operating timescale.

1 INTRODUCTION

The Problem. Modern cloud data warehouses such as Amazon Redshift Serverless [3] and Snowflake [52] decouple compute from storage, enabling multiple compute clusters to access the same underlying data. For example, multiple Amazon Redshift Serverless workgroups can access the same data through Datashares [5], while Snowflake natively supports multi-cluster warehouses [50].

Customers have embraced this feature [55], because it enables coarse-grained performance isolation by allowing the separation of workloads with different latency service-level objectives (SLOs). For example, interactive dashboards may need to respond within a few seconds, nightly ETL workloads may need to finish before business hours, and data-science workloads may tolerate some additional latency as long as costs are kept bounded.

However, even if workloads are each assigned to a separate cluster, current systems do not allow SLOs to be specified directly. This means that administrators have to experiment with the cluster's settings (e.g. number of nodes or qualitative price-performance hint [39]) until the desired latency SLO is met. Even after such tuning, there is no control over what happens as queries *within* each workload may interfere with one another. As a recent work by the Amazon Redshift team notes, *"large ad-hoc queries can have a severe negative impact on the overall performance of the cluster, since they can occupy compute resources and cause cache thrashing."* [39]

In this work, we take a different approach: we treat latency SLOs as first-class inputs and use the mechanisms exposed by cloud data warehouses to meet them cost-efficiently. The goal is not cross-workload performance isolation for its own sake, but rather meeting the SLO *of each individual query*, while minimizing infrastructure cost. This requires deciding which clusters to use, when to scale them, and where to route each incoming query.

What Success Looks Like. If each cluster could be spun up instantaneously and immediately offer its full performance, the task at hand would be simple: the system could wait until demand appears, create exactly the resources needed, and route queries accordingly. However, it takes time to provision new resources, and additional time for caches and other internal state to warm up. As a result, to cost-efficiently meet latency SLOs, we must manage cluster configuration across multiple timescales, as summarized in Table 1:

- **Plan ($\approx$ 24 hrs):** Enterprise workloads often exhibit recurring patterns [55]. By extracting and anticipating these patterns, the system can make long-term cluster-management and configuration decisions proactively, such as when to spin up or tear down clusters and how to tune the policies used during execution. Because these decisions are made offline and amortized over longer periods, one can afford to evaluate a richer set of alternatives.
- **Adjust ($\approx$ 5 mins):** Forecasts are imperfect, and workloads can evolve. The system must be able to respond by spinning up or tearing down clusters to compensate for unexpected bursts, lulls, or shifts in workload mix. Unlike planning, adjustment operates under tighter time constraints and therefore makes more localized decisions within the policy chosen by the planner.
- **React ($\approx$ 1 s):** Finally, each arriving query must be assigned to one of the currently active clusters. Since the goal is no longer workload performance isolation, but rather meeting query-level latency SLOs cost-efficiently, this decision must account for the query's own latency SLO, its impact on already-running queries, and the infrastructure cost implied by the routing decision.

These behaviors are complementary. Planning reduces dependence on slow reactive scaling by preparing resources before expected demand arrives. Adjustment handles forecast error and unexpected workload changes. Reaction handles the per-query effects of concurrency and contention, which remain present even under a

Table 1: No prior work exhibits all key desired behaviors.

Key Desired Behavior	**Plan** resources based on history	**Adjust** resources based on demand	**React** to concurrent live load
Timescale	≈ 24 hrs	≈ 5 mins	≈ 1 s
ResTune [63]	✓ *Replay-based knob tuning on replica*	✗	✗
WiseDB [33]	✓ *Tree model for fixed per-template routing*	✗	✗
Das et al. [16]	✗	✓ *Container sizing w/ token bucket*	✗
SLAOrchestrator [40–42]	✗	✓ *Utilization- or simulation-based pool resizing*	✗
BRAD [26, 62]	✗	✓ *Blueprint beam search*	✗
Auto-WLM [49]	✗	✓ *Queueing-based horizontal scaling*	✗
RAIS [39]	✓ *Multi-forecast parameter tuning*	✓ *Queueing-based horizontal scaling*	✗
AutoSLO	✓ *Multi-forecast spinup planning & tuning*	✓ *Simulation-based best-cluster spinup*	✓ *Concurrency-aware routing*
Component	Policy Tuner	Autoscaler	Query Router
Details in...	Section 7	Section 6	Section 5

well-chosen cluster configuration. Together, they allow a system to leverage multiple clusters to meet query-level latency SLOs without requiring rigid workload-to-cluster assignments.

How Prior Work Falls Short. Several automatic workload management mechanisms have been proposed in recent years. However, as shown in Table 1, none exhibits all three **key desired behaviors**.

Some systems can only **plan** for a known or forecasted workload, using replays [63] or modeling [33]. However, planning alone is insufficient: forecasts are imperfect, workload mixes can shift, and live query-level contention can affect SLO adherence and must be managed. Other systems can only **adjust** resources online in response to demand, using various forms of short-term performance modeling/assumptions under different available resources [16, 26, 40–42, 49, 62]. However, spinning up new clusters can require non-negligible time, so purely reactive approaches will transiently leave the system under-provisioned, risking SLO violations.

Finally, existing systems route queries using static workload classes, query templates, or estimated resource requirements [33, 39, 42, 49, 62]. They do not **react** to live concurrent query load by explicitly considering how placing a new query on a particular cluster affects SLO adherence (for both the new query and the already-running ones). This is how RAIS [39] still falls short of addressing the full problem: since it only exposes a *qualitative* price-performance slider, rather than a way to specify latency SLOs, it cannot provide SLO-aware query placement.

Our Approach. AutoSLO addresses the gaps discussed above by jointly providing all three key desired behaviors: planning from historical workload patterns, adjusting resources when reality deviates from the plan, and reacting to live concurrent load when routing each query. It does so through three corresponding components: the Policy Tuner, the Autoscaler, and the Query Router.

The Policy Tuner can **plan** using the historical workload. It generates and simulates forecasted workloads to determine proactive cluster-management actions and configure parameters. By ensuring that the right resources are available at the right time (shortly before demand materializes), it reduces dependence on reactive scaling and mitigates cluster spinup delays.

The Autoscaler can **adjust** the active cluster set when observed workload behavior deviates from the forecast. Its important contributions include cluster spinup/teardown triggers, which identify opportune moments for resource adjustments, and a cluster spinup size selector, which uses short-term simulations to estimate the SLO and cost implications of different scaling actions.

Finally, the Query Router can **react** to live load. When a query q arrives, it selects on which active cluster it should be executed by managing a multi-way tradeoff among (1) the risk that q will violate its SLO; (2) the risk q poses to the SLO adherence of already-running queries; and (3) the resource cost implied by each routing decision.

Contributions. In summary, we make the following contributions:

- We formulate the problem of SLO-aware multi-cluster workload management for cloud data warehouses, where the goal is bounded SLO violation rate at minimal infrastructure cost.
- We present AutoSLO, a multi-timescale framework that provides the three key desired behaviors needed for this problem: planning from historical workload patterns, adjusting resources when workloads deviate from the forecast, and reacting to live concurrent load when routing individual queries.
- We develop the components of AutoSLO: a forecast-driven Policy Tuner for proactive cluster management and tuning, a simulation-based Autoscaler for cost- and SLO-aware scaling, and a concurrency-aware Query Router for per-query placement.
- We evaluate AutoSLO on realistic Redbench-generated workloads and show that it can successfully meet latency SLOs of varying strictness, reducing cost by a mean of 26.4% compared to the per-scenario next-best baseline.
- We also evaluate per-component performance and find that the Query Router and Autoscaler reduce the SLO violation rate by a mean of 47.8% and 93.7% respectively, compared to corresponding alternatives on TPC-DS-derived workloads, while the Policy Tuner can reduce the SLO violation rate by a mean of 44.6% using a single day of workload history. Each component is also shown to be efficient enough for its operating timescale.

2 PROBLEM FORMULATION

We now define the workload and system setting more precisely.

Online, Diverse Query Arrivals. Each query q is issued online at time $q.t_a$. At each point in time, no information about future query arrivals is known exactly. Each query q has varying resource demands (e.g. CPU, memory, I/O), which may interact with those of concurrently executing queries, impacting their latency.

Latency SLOs. Each query q has a latency service-level objective $SLO(q)$, treated as an input provided by the application or administrator. For example, the SLO may be inferred from the query's source application, user-defined priority, or other submission-time metadata. A query q has *met* it whenever its *client-side* latency $L(q) \leq SLO(q)$. Otherwise, q has *violated* its SLO.

Clusters. Multiple clusters can query the same data, with the vendor managing concurrency control and consistency. Each cluster c has a *size* $S(c)$ describing its computational capacity. The queries running on c are denoted Q_c. Instead of statically assigning each workload to a cluster, we allow routing queries independently:

Definition 1: Routing Policy

Let $C_{q.t_a}$ be the active cluster set when query q arrives, and $\mathcal{H}_{q.t_a}$ be the historical workload observed until then, including prior query arrivals, routing decisions, completions, and SLOs. A routing policy $\mathcal{R}$ determines which active cluster to execute q on:

$$\mathcal{R}(q, C_{q.t_a}, \mathcal{H}_{q.t_a}) \mapsto c \in C_{q.t_a}$$

Adjustable Cluster Pool. New clusters can be requested from the vendor, and existing clusters can be released. Spinning up a new cluster c takes time δ; tearing down an existing cluster takes time ζ. We present δ and ζ as constants, but our techniques also support sampling them. This leads to another decision point:

Definition 2: Scaling Policy

At time t, given the active cluster set C_t and the historical workload $\mathcal{H}_t$, a scaling policy $\mathcal{S}$ outputs a new active cluster set C_{new}:

$$\mathcal{S}(t, C_t, \mathcal{H}_t) \mapsto C_{new}$$

Usage-Based Billing. Let $\mathcal{P} = (\mathcal{R}, \mathcal{S})$ denote a pair of policies. The cost $k^{\mathcal{P}}(c, Q)$ of a cluster c over workload Q with these policies is the product of: (i) A constant *price* p; (ii) c's size $S(c)$; and (iii) c's *active time* while Q is executed, measured by a timer as follows:

- The timer is initially paused.
- If paused, it starts running when a query arrives at c.
- If running, it pauses when both are true: (i) there are no running queries on c; and (ii) at least m seconds (often $m = 60$ [4, 51]) have elapsed since the timer started running.

Billing interacts with routing: using an idle cluster restarts its timer, while consolidating reduces cost but can cause interference.

Target-Aware Comparison. For a workload Q under policies $\mathcal{P}$, let the *SLO violation rate* as $v^{\mathcal{P}}(Q) = \frac{1}{|Q|} \sum_{q \in Q} \mathbf{1}[L^{\mathcal{P}}(q) > SLO(q)]$ and the *total cost* as $k^{\mathcal{P}}(Q) = \sum_c k^{\mathcal{P}}(c, Q)$. We also allow specifying an SLO violation rate *target* τ. Not treating τ as a hard constraint allows comparing policies even when their SLO violation rate is above τ. In particular, we compare policies with SLO violation rates $V = \{v^{\mathcal{P}_1}, \ldots, v^{\mathcal{P}_n}\}$ and costs $K = \{k^{\mathcal{P}_1}, \ldots, k^{\mathcal{P}_n}\}$ using:

$$\textsc{BestWithTarget}(V, K, \tau) = \operatorname*{lex\,min}_{(v,k) \in \{V,K\}} (\max(0,\ v - \tau),\ k)$$

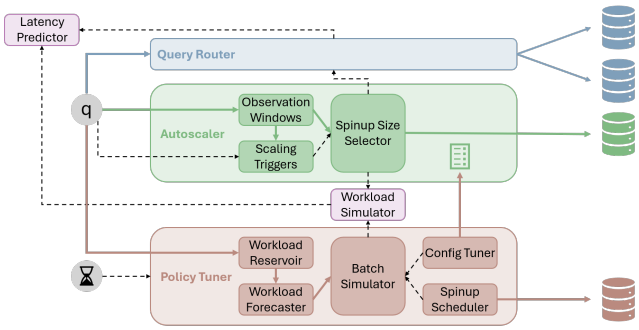


Figure 1: The architecture of AutoSLO. Solid lines indicate input/output flow, while dashed lines indicate control flow.

Problem Definition. We want to find the best policy under the lexicographic target-aware objective above. Concretely:

Definition 3: SLO-aware Multi-Cluster Workload Management

For an SLO violation rate target $\tau \geq 0$, select a policy pair $\mathcal{P}^\star$ to meet τ first and reduce cost second:

$$\mathcal{P}^\star = \{\mathcal{P} \mid (v_{\mathcal{P}}, k_{\mathcal{P}}) = \textsc{BestWithTarget}(V, K, \tau)\}$$

3 OVERVIEW OF AUTOSLO

To address the problem above, we design AutoSLO to demonstrate the three **key desired behaviors** from Section 1. For our technical exposition in the following Sections, unlike our top-down introduction, we follow a query's path through the system bottom-up.

As shown in Figure 1, an incoming query q first encounters the Query Router, which can **react** to the current workload and decide which active cluster to forward q to. This decision is powered by the Latency Predictor, which estimates how different placements would affect both q and the already-running queries. Sections 4 and 5 describe the predictor and routing algorithm, respectively.

Each query is also added to the Autoscaler's sliding Observation Windows and trips the Scaling Triggers, which decide whether to **adjust** the active cluster set. If a spinup is triggered, the Spinup Size Selector simulates a short-term workload forecast under different hypothetical cluster additions to balance SLO adherence and cost. It uses the Workload Simulator, a simple event-driven simulator that can replay query arrivals and uses the Latency Predictor to determine query completions. Section 6 expands on this process.

Finally, each query is added to the Workload Reservoir within the Policy Tuner, which is triggered periodically (e.g. every 24 hours). The Workload Forecaster uses the reservoir to generate forecasted workloads, which are efficiently simulated using the Batch Simulator. Based on these simulations, the Spinup Scheduler optimizes cluster spinup timing for reliably recurring load, while the Configuration Tuner then tunes Autoscaler parameters governing on-demand autoscaling. Section 7 covers this functionality.

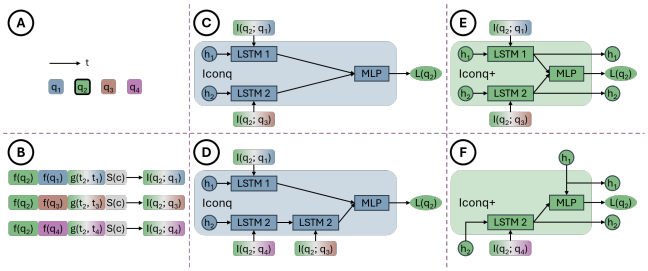


Figure 2: When predicting the latency of q_2, Iconq+ ingests the interaction feature vectors of later neighbors (q_3 and q_4) in chronological order, enabling incremental inference.

4 LATENCY PREDICTOR

We start with Iconq [58], a recent LSTM-based concurrent query latency estimation model, but modify it to address the needs of AutoSLO. In particular, we need support for clusters of different sizes (in the Query Router), while we also plan to use the model to determine query completion times during simulations (in the Workload Simulator). Before discuss how we address these requirements in our Latency Predictor, called Iconq+, we present Iconq.

4.1 Background: Iconq

Iconq was developed for single-cluster query scheduling, where arriving queries can be queued and/or reordered to reduce mean latency. At a high level, Iconq predicts the latency of a query q by ingesting a sequence of *interaction feature vectors*, derived from q and its neighbors, with an LSTM. We define q_n to be a neighbor of q if their executions overlap. Each interaction feature vector concerns the target query q and a particular neighbor q_n and includes: query-plan-derived features for q and q_n; relative timing information about their arrivals; and concurrency-unaware latency estimates for each query, $\hat{\ell}_q$ and $\hat{\ell}_{q_n}$, derived from a fast decision-tree-based sub-model (Stage [57]). Importantly, Iconq ingests neighbors arriving before q and after q separately, enabling latency prediction updates for already-running queries to account for submissions of new ones.

4.2 Query Featurization

Iconq assumes all queries run on the same cluster. In AutoSLO, however, we must predict latencies across clusters of different sizes. We therefore extend the *interaction feature vector* with features related to cluster size, including: (i) the raw cluster size; (ii) $\log_2(\text{size})$ to capture non-linear scaling; (iii) $1/\text{size}$ to capture inverse-capacity effects; (iv) $\hat{\ell}_q \cdot \text{size}$ and $\hat{\ell}_{q_n} \cdot \text{size}$ as proxies for the compute work implied by each query's isolated latency; and (v) $(\hat{\ell}_q + \hat{\ell}_{q_n})/\text{size}$ as a capacity-normalized pairwise contention signal. In Figure 2**B**, we see the interaction feature vectors for the queries in Figure 2**A**, when predicting the latency of q_2. Each includes query plan features and Stage latency predictions per query $f(q)$, relative timing features $g(t; t_{neighbor})$ and cluster size features $S(c)$ as above.

4.3 Censored Observations

To predict the latency of q, Iconq uses features derived from q and its neighbors. When scheduling online, precise neighbor sets are known at decision time. In AutoSLO, however, we also use the Latency Predictor in the Workload Simulator. There, whether two queries overlap can depend on their simulated (predicted) latencies, which require overlap knowledge. Hard inclusion of neighbors therefore creates a feedback loop, where early prediction errors alter downstream neighbor sets and lead to compounding errors.

To mitigate this, we also train Iconq+ on censored observations using sampled partial neighbor sets, where labels are treated as *lower bounds* (i.e. loss is zero if $pred \geq label$). This lets us extract more from the training data. For example, if q ran from $t = 0$ to $t = 10$ and q' arrived at $t = 7$, then we know that $L(q) = 10$ in the presence of q'; however, we also know that $L(q) \geq 7$ without q', since q was still running when q' arrived.

Censoring also lets us train on queries aborted due to timeouts or errors, with time-to-failure as their censored label. This is especially important for small clusters, where timeouts can be more common; discarding timed-out queries would over-represent successful executions and bias the model toward latency under-prediction.

4.4 Incremental Predictions

Iconq uses a bidirectional LSTM to ingest a sequence of interaction feature vectors. Per Figure 2**C**/**D**, when predicting the latency of q_2, neighbors arriving before q_2 (i.e. q_1) are ingested in chronological order, while neighbors arriving after q_2 (i.e. q_3 and q_4) are ingested in reverse chronological order. This makes both sequences end near q_2's arrival, emphasizing nearby interactions in the LSTM state. However, it also means that whenever a new neighbor arrives (e.g. q_4 in panel **D**), every neighbor after q_2 must be re-ingested.

Although the absolute re-ingestion overhead per query can be small, it can add up during simulations, increasing the runtime of the Autoscaler and the Policy Tuner. This is especially true because simulations interleave predictions and CPU-heavy state bookkeeping, effectively mandating CPU inference. To avoid repeated re-ingestion, we modify Iconq+ to also process after-neighbors in chronological order, and add support for incremental inference from cached model state (Figure 2**E**/**F**).

5 QUERY ROUTER

The Query Router routes each incoming query q to an active cluster, leveraging the Latency Predictor to balance three concerns: (1) the ability of q to meet its SLO, (2) the impact of q on the SLO adherence of currently running queries, and (3) the cost implications of the routing decision. The Query Router implements Route (Algorithm 1). It first inspects each cluster c, gathering the SLO-violating queries (line 2) and the total projected cost, from c's spinup until the running queries are predicted to drain (line 3). In the same pass, it builds each cluster's counterfactual query neighbor map, as if the incoming query q were placed there (line 4).

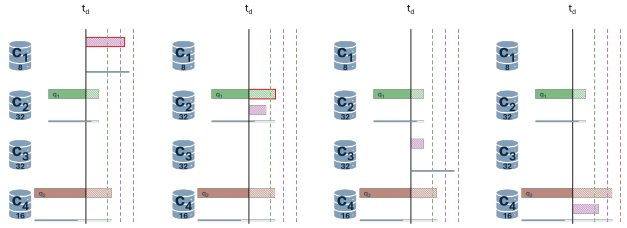
It then runs Iconq+ inference on these counterfactual neighbor maps in a single batch (line 5), yielding updated model predictions $\hat{L}_c^{\text{after}}$ and model states M_c^{after}. As explained in Section 4.4, Iconq+ updates the latency predictions and model states for already-running queries *incrementally* within this step. Route then applies a monotonicity guard for stability (lines 6-8): the latency prediction for each running query is never updated downwards, reflecting the

Algorithm 1 Overall workflow of the Query Router (ROUTE).

Inputs: Incoming query q, active cluster set $C = \{c\}$ with latency predictions $\hat{L}_c^{\text{before}}$ and latency predictor states M_c^{before} for running queries.

Outputs: Selected cluster $c^\star$, updated latency predictions $\hat{L}_{c^\star}$ and latency predictor states $M_{c^\star}$

1: **for** $c \in C$ **do**
2: $\quad v_c^{\text{before}} \leftarrow \sum_{q_i \in Q_c} \mathbf{1}[\hat{L}_c^{\text{before}}(q_i) > \text{SLO}(q_i)]$
3: $\quad k_c^{\text{before}} \leftarrow \text{CLUSTERCOSTUNTILDRAINED}(c, Q_c, \hat{L}_c^{\text{before}})$
4: $\quad \mathcal{N}_c \leftarrow \text{PERQUERYNEIGHBORSIFWEROUTE}(c, q)$
5: $\{L_c^{\text{after}}, M_c^{\text{after}}\}_{c \in C} \leftarrow \text{PREDICTWITHICONQ+}(\{\mathcal{N}_c, M_c^{\text{before}}\}_{c \in C})$
6: **for** $c \in C$ **do**
7: $\quad$ **for** $q_i \in Q_c$ **do**
8: $\quad\quad \hat{L}_c^{\text{after}}(q_i) \leftarrow \max(\hat{L}_c^{\text{after}}(q_i), \hat{L}_c^{\text{before}}(q_i), q.t_{\text{arrival}} - q_i.t_{\text{arrival}})$
9: **for** $c \in C$ **do**
10: $\quad Q_c^{\text{after}} \leftarrow Q_c \cup \{q\}$
11: $\quad v_c^{\text{after}} \leftarrow \sum_{q_i \in Q_c^{\text{after}}} \mathbf{1}[\hat{L}_c^{\text{after}}(q_i) > \text{SLO}(q_i)]$
12: $\quad k_c^{\text{after}} \leftarrow \text{CLUSTERCOSTUNTILDRAINED}(c, Q_c^{\text{after}}, \hat{L}_c^{\text{after}})$
13: $\quad \Delta v_c \leftarrow v_c^{\text{after}} - v_c^{\text{before}}$
14: $\quad \Delta k_c \leftarrow k_c^{\text{after}} - k_c^{\text{before}}$
15: $c^\star \leftarrow \text{lexicographic-argmin}_{c \in C} \ (\Delta v_c, \Delta k_c)$
16: **return** $c^\star, \hat{L}_{c^\star}^{\text{after}}, M_{c^\star}^{\text{after}}$



(a) q **SLO miss** **(b)** q_1 **SLO miss** **(c) Expensive** **(d) Balanced**

Figure 3: The Query Router balances SLOs and cost. The x-axis is time. Each query is a rectangle and same-color dashed lines show when it will miss its SLO. Hasing indicates predicted latencies; a red outline marks an SLO miss. Thin blue lines show billed time; dark blue indicates the minimum.

fact that later queries generally cannot make an earlier query faster, and cannot be smaller than its observed latency so far.

For each candidate cluster c, the algorithm then counts the SLO-violating queries and implied cost in a manner similar to the baseline state (lines 9-12), ultimately calculating the marginal impact of the routing decision (lines 13-14). It then identifies the cluster $c^\star$ that minimizes marginal SLO violations and cost (in this order) and returns it, together with the corresponding updated latency predictions and latency predictor states (lines 15-16).

Figure 3 shows an example, where query q (purple) is routed among 4 clusters, with 2 active queries q_1 (green) and $q2$ (brown). Routing q to c_1 would make it miss its SLO, while routing it to c_2 would make q_1 miss its own SLO. Routing it to c_3 would imply no SLO violations, but it would start a new billing window, leading to high cost. The best option is therefore c_4; there are no SLO violations, while the marginal cost impact is minimized.

By explicitly reasoning about query interference, SLO violation risk, and resource cost, the Query Router can make decisions that balance performance and efficiency in real time.

Algorithm 1 Computational Time Complexity. Before and after model inference, ROUTE performs a constant amount of work per running query, implying $O(N)$, where $N = \sum_c |Q_c|$. For line 5, each LSTM step is $O(dh + Lh^2)$ (to ingest the input and update the hidden state per layer), assuming input dimension d, hidden size h and L layers. We will treat this as constant, since we use a fixed model. We perform two steps per running query q_{running}: one to update q_{running}'s latency prediction if q is added to its neighbors (see Section 4.4), and another while ingesting q_{running} as a neighbor of q while predicting the latency of q. Line 5 is then also $O(N)$, meaning that ROUTE's overall time complexity is $O(N)$.

Algorithm 2 The spinup trigger of the Autoscaler (MAYBESPINUP).

Inputs: Completion window $\mathcal{W}^c$, active cluster set $C = \{c\}$ with predictions $\hat{L}_c$

Configuration: Trigger SLO threshold τ_{trigger}, minimum observations θ

State: Known active cluster set $\mathcal{K}$, change bound t_{change} (shared with **Algorithm 3**), in-flight flag f (shared with **Algorithm 3**)

Outputs: Whether to invoke the Spinup Size Selector.

1: **if** $\mathcal{K} \neq C$ **then**
2: $\quad \mathcal{K} \leftarrow C$
3: $\quad t_{\text{change}} \leftarrow \max(t_{\text{change}}, q.t_{\text{arrival}})$
4: $\quad f \leftarrow \text{FALSE}$
5: $R \leftarrow \{(q, \hat{L}_c(q)) \mid (q.t_{\text{arrival}} \geq t_{\text{change}}) \wedge (q \in Q_c, c \in C)\}$
6: $D \leftarrow \{(q, L(q)) \mid (q.t_{\text{arrival}} \geq t_{\text{change}}) \wedge (q \in \mathcal{W}^c)\}$
7: **if** $\neg f$ **and** $|R| + |D| \geq \theta$ **then**
8: $\quad v_{\text{cur}} \leftarrow \frac{1}{|R|+|D|} \sum_{(q,\ell) \in \{R \cup D\}} \mathbf{1}[\ell > \text{SLO}(q)]$
9: $\quad$ **if** $v_{\text{cur}} > \tau_{\text{trigger}}$ **then**
10: $\quad\quad f \leftarrow \text{TRUE}$
11: $\quad\quad$ **return** TRUE
12: **return** FALSE

6 AUTOSCALER

The Query Router routes each query *among the active cluster set* C; however, it may be that C is no longer appropriate. The Autoscaler addresses this, using three sub-components (see Figure 1).

6.1 Observation Windows

The Autoscaler runs in a background thread and maintains two trailing Observation Windows over the last w seconds. It appends each incoming query to the *arrival window* $\mathcal{W}^a$ while it appends each completed query (alongside its arrival time and latency) to the *completion window* $\mathcal{W}^c$, truncating each as needed. Each query arrival also trips two Scaling Triggers, explained next.

6.2 Scaling Triggers

6.2.1 Teardown Trigger. The teardown trigger determines whether one of the active clusters should be torn down. For an active cluster c, it fires when three conditions are simultaneously met: (i) no

queries are currently running on c; (ii) no query has been routed to c for the last T_{idle} seconds; and (iii) c was spun up at least $T_{\text{min\_lifetime}}$ seconds earlier. If all conditions are met, the Autoscaler initiates the teardown of c. T_{idle} and $T_{\text{min\_lifetime}}$ can be periodically optimized by the Policy Tuner, as we will examine in Section 7.5.

6.2.2 Spinup Trigger. The spinup trigger determines whether a new cluster should be spun up, based on whether recent SLO adherence is unsatisfactory. As described in Algorithm 2, it first checks whether the active cluster set has changed since the last invocation; if so, it saves it, records the current arrival time as t_{change} and clears the in-flight flag f (lines 1–4). The algorithm then assembles R (line 5), the set of running queries that arrived after t_{change}, and D (line 6), the set of queries from $\mathcal{W}^c$ that arrived after t_{change}, alongside their predicted and realized latencies, respectively. If the combined size of these two sets is at least θ and the in-flight flag is not set (line 7), the algorithm computes the SLO violation rate among $R \cup D$ (line 8) and compares it to τ_{trigger} (line 9). The trigger fires only if this threshold is exceeded, at which point the in-flight flag is also set (lines 10-12). The parameters τ_{trigger} and θ can be periodically optimized by the Policy Tuner, as we will examine in Section 7.5. If the spinup trigger fires, the Autoscaler invokes the Spinup Size Selector, explained next in Section 6.3.

Algorithm 2 Computational Time Complexity. The comparison on line 1 takes time $O(\min(|\mathcal{K}|, |C|))$, while constructing R and D on lines 5-6 requires time $O(N + |\mathcal{W}_c|)$, where $N = \sum_c |Q_c|$. Line 8 (if reached) operates on a subset of these query sets, so the overall complexity is $O(\min(|\mathcal{K}|, |C|) + N + |\mathcal{W}_c|)$.

6.3 Spinup Size Selector

Once the spinup trigger fires, the Spinup Size Selector must decide what cluster it would be most beneficial to spin up, if any. At a high level, for each eligible cluster size, it hypothesizes a cluster spinup of that size and compares the downstream SLO violation rate and cost. It does this by simulating a short-term workload forecast, where queries arrive according to copies of the arrival window $\mathcal{W}^a$, are routed as usual by the Query Router, and complete based on their predicted latencies. Crucially, it also evaluates a *do-nothing baseline* (no new cluster added), and only recommends a spinup if some candidate strictly improves upon this baseline.

Remember that a new cluster will not be available instantly, but only after a spinup delay δ. It is vital to capture this *preparation period* in the simulation, because it may generate a workload backlog that the new cluster will have to help relieve, once available. However, the events of this preparation period are independent of the size of the requested cluster, so that it is sufficient to compute the simulation state at the end of the preparation period once.

More precisely, the Spinup Size Selector operates according to Algorithm 3. Lines 1–11 describe the preparation period: copies of the queries in the arrival window are issued (lines 3–4 and 10) and routed (lines 8–9), while queries that are predicted to have completed are removed from tracking (line 7). Once the new cluster is available (lines 1–2 and 5–6), replay stops and we snapshot the cluster set state (line 11). We extend the candidate set with the do-nothing baseline $\emptyset$ (line 12) and, for each possible action, initialize the cluster set from the snapshot (lines 13–17).

Algorithm 3 The Spinup Size Selector (FindBestSpinupSize).

Inputs: Arrival window $\mathcal{W}^a$, active cluster set $C = \{c\}$ with latency predictions $\hat{L}_c$, decision time t_d, target SLO violation rate τ_{target}

Configuration: Eligible cluster sizes $\mathcal{S}$, spinup delay δ, minimum completions μ, observation window width w

State: Change bound t_{change}, in-flight flag f (both shared with **Algorithm 2**)

Outputs: Cluster size s to spin up, or $\emptyset$ if none advisable.

1: $t_{\text{available}} \leftarrow t_d + \delta$; $i \leftarrow 0$
2: **while true do**
3: $\quad q \leftarrow \mathcal{W}^a[i \mod |\mathcal{W}^a|]$
4: $\quad q.t_{\text{arrival}} \leftarrow q.t_{\text{arrival}} + w \cdot \lfloor i/|\mathcal{W}^a| \rfloor$
5: $\quad$ **if** $q.t_{\text{arrival}} \geq t_{\text{available}}$ **then**
6: $\quad\quad$ **break**
7: $\quad$ CollectFinishedUntil(C, $q.t_{\text{arrival}}$)
8: $\quad (c^\star, \hat{L}_{c^\star}, M_{c^\star}) \leftarrow$ Route(q, C)
9: $\quad c^\star$.AddQuery(q, $\hat{L}_{c^\star}, M_{c^\star}$)
10: $\quad i \leftarrow i + 1$
11: $(C_{\text{snap}}, i_{\text{snap}}) \leftarrow (C, i)$
12: $\mathcal{S}' \leftarrow \mathcal{S} \cup \{\emptyset\}$
13: **for** $s \in \mathcal{S}'$ **do**
14: $\quad C_s \leftarrow \text{copy}(C_{\text{snap}})$
15: $\quad$ **if** $s \neq \emptyset$ **then**
16: $\quad\quad C_s \leftarrow C_s \cup \{$NewCluster(size $= s$)$\}$
17: $\quad i \leftarrow i_{\text{snap}}$
18: $\quad D \leftarrow \emptyset$
19: $\quad$ **while** $|D| < \mu$ **do**
20: $\quad\quad q \leftarrow \mathcal{W}^a[i \mod |\mathcal{W}^a|]$
21: $\quad\quad q.t_{\text{arrival}} \leftarrow q.t_{\text{arrival}} + w \cdot \lfloor i/|\mathcal{W}^a| \rfloor$
22: $\quad\quad F \leftarrow$ CollectFinishedUntil(C_s, $q.t_{\text{arrival}}$)
23: $\quad\quad D \leftarrow D \cup \{(q', L(q')) \mid (q' \in F) \wedge (q'.t_{\text{arrival}} \geq t_{\text{available}})\}$
24: $\quad\quad (c^\star, \hat{L}_{c^\star}, M_{c^\star}) \leftarrow$ Route(q, C_s)
25: $\quad\quad c^\star$.AddQuery(q, $\hat{L}_{c^\star}$, $M_{c^\star}$)
26: $\quad\quad i \leftarrow i + 1$
27: $\quad D \leftarrow D \cup_{c \in C_s} \{(q', \hat{L}_c(q')) \mid q' \in Q_c\}$
28: $\quad v_s \leftarrow \frac{1}{|D|} \sum_{(q,\ell) \in D} \mathbf{1}[\ell > \text{SLO}(q)]$
29: $\quad k_s \leftarrow \sum_{c \in C_s}$ ClusterCostUntilDrained(c, Q_c, $\hat{L}_c$)
30: $(v_{\text{best}}, k_{\text{best}}) \leftarrow$ BestWithTarget($\{(v_s, k_s)\}_{s \in \mathcal{S}'}, \tau_{\text{target}}$)
31: **if** $(v_{\text{best}}, k_{\text{best}}) = (v_\emptyset, k_\emptyset)$ **then**
32: $\quad t_{\text{change}} \leftarrow t_d + \delta$
33: $\quad f \leftarrow$ False
34: $\quad$ **return** $\emptyset$
35: **return** $\max\{s \in \mathcal{S} \mid (v_{\text{best}}, k_{\text{best}}) = (v_s, k_s)\}$

We then continue until we have simulated μ query completions (lines 18–19). The core simulation loop is mostly the same (lines 20–26), with the exception that we record the latency of completing queries that arrived after $t_{\text{available}}$ (lines 22–23). After exiting the simulation, we augment this set of μ completed queries with the predicted latencies of outstanding queries (line 27) and compute the SLO violation rate and cost (lines 28–29). We then find the action that provides the best balance of SLO violation rate and cost according to an SLO violation rate target τ_{target} (line 30, see the end of Section 2). If $\emptyset$ wins, we return it, suppressing trigger

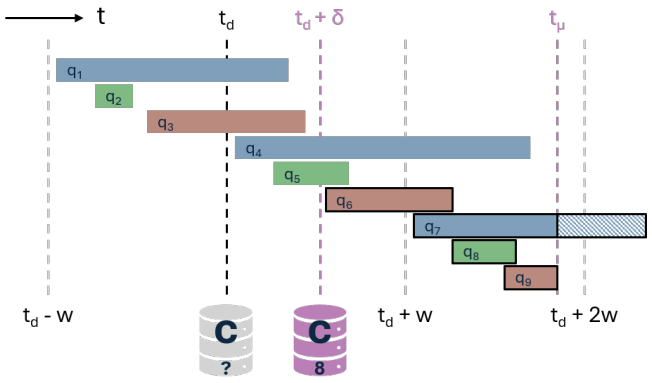


Figure 4: The Spinup Size Selector evaluates each scaling action at t_d by simulating copies of the arrival window $\mathcal{W}^a$. It stops once μ queries (here $\mu = 3$) issued after the new cluster is available ($t_d + \delta$; queries outlined in black) finish executing.

re-evaluation for δ seconds to prevent immediate re-firing (lines 31–34, see Algorithm 2). Otherwise, we return the largest cluster tied for the best outcome (line 35), erring on the side of caution when multiple sizes perform equally well. The parameters μ and w can be periodically optimized by the Policy Tuner, as we will examine in Section 7.5.

Figure 4 shows an example for $\mu = 3$, where the x-axis is time and each rectangle is a query. The query arrivals q_1–q_3 in the arrival window $\mathcal{W}^a$ are replayed twice until q_9's completion at t_μ marks the third completion after $t_d + \delta$, the time the new (purple) cluster of size 8 became available. The simulated latencies of q_6, q_8 and q_9, and the predicted latency of q_7 as of t_μ, contribute to v_8 and k_8.

Algorithm 3 Computational Time Complexity. In the preparation phase (lines 2-10), the number of query arrivals will be $O(|\mathcal{W}^a| \cdot \frac{\delta}{w})$. Let the number of query arrivals required on lines 20-26 until μ queries complete be F_s when the new cluster has size s (or $s = \emptyset$), and let $F^{\max} = \max_{s \in \mathcal{S}'} F_s$. Let each arriving query q encounter R_q running queries, among which any finished queries must be identified ($O(R_q)$) before q is routed ($O(R_q)$), and let $R^{\max} = \max_q R_q$. For the drain phase and SLO/cost calculations (lines 27-29) we similarly incur $O(R_q)$. Therefore, the overall complexity of FindBestSpinupSize is $O((|\mathcal{W}^a| \cdot \frac{\delta}{w} + F^{\max}) \cdot R^{\max})$.

7 POLICY TUNER

As described in Section 6, the Autoscaler can use the recent workload to adjust the active cluster. However, over longer time horizons, additional optimization opportunities open up. The Policy Tuner, invoked at the granularity of several hours[1], captures such opportunities, as follows. Each incoming query is added to the Workload Reservoir (Section 7.1), which can be used by Workload Forecaster (Section 7.2) to draw forecasted workloads. The Batch Simulator (Section 7.3) can efficiently simulate these workloads under different configurations, empowering the Spinup Scheduler (Section 7.4) to schedule proactive cluster spinups and the Configuration Tuner (Section 7.5) to optimize Autoscaler knobs.

7.1 Workload Reservoir

Each query that reaches AutoSLO is also added to the Workload Reservoir. It consists of two tables indexed by calendar date and hour of day. The *count table* records, for each (*date*, *hour*, *query_template*) triple, how many times that query template was issued. The *arrivals table* records the within-hour second offset of every query arrival in each (*date*, *hour*) bin. Both tables have configurable sampling policies to keep their storage footprint bounded, but they importantly do not need to be kept in memory, since they are only used by the infrequently invoked Policy Tuner.

7.2 Workload Forecaster

The Workload Forecaster generates forecasted workloads for the day ahead and partitions them into disjoint training and validation splits. In our current default implementation, it iterates over hours of the day and samples query texts from all available same-day-and-hour-of-week bins. It applies a decay to their sampling probability, so that samples from k weeks ago are weighted by λ^{k-1} (with $\lambda \in (0, 1)$). To assign arrival times to each forecasted query, it calculates interarrival deciles from the *arrivals table* (over the same-day-and-hour-of-week bins) and samples uniformly within each decile. For sparse bins, it falls back to uniform sampling.

Note that the forecasted workloads only need to approximate the overall load contours of the real workload, so that coarse scaling decisions can be planned ahead of time. Alternative forecasting and sampling policies are also supported (e.g., a single prior day, or an equally-weighted 7-day window), but the described policy best balanced simplicity and predictive performance in our experience.

7.3 Batch Simulator

SimulateBatch is the shared evaluation primitive invoked by every phase of the Policy Tuner. It accepts a set of *execution configurations* $\{E_i\}$ and a set of workloads W, and returns a result matrix where entry $\mathcal{R}[i][j]$ holds the simulation outcome for configuration E_i on workload $W[j]$. All $|\{E_i\}| \times |W|$ simulations are dispatched to a process pool and run in parallel in an optimized fashion. Each *execution configuration* specifies any scheduled cluster spinups for the workload ahead (cluster size and spinup time), as well as values for the Autoscaler configuration parameters: T_{idle} and $T_{\text{min\_lifetime}}$ (Section 6.2.1); τ_{trigger} and θ (Section 6.2.2); and μ and w (Section 6.3).

7.4 Spinup Scheduler

7.4.1 High-level Workflow. The Spinup Scheduler augments execution configurations with scheduled cluster spinups for periods of predictable congestion. Conceptually, it greedily evolves an initial execution configuration by adding spinups over at most η rounds. In each round it simulates the current configuration on the training workloads, identifies promising times at which additional capacity may help reduce SLO violations, and accepts the best spinup candidate only if it improves upon the existing configuration.

Algorithm 4 implements this search for a set of initial configurations $\mathcal{E} = \{E_i^0\}_{i \in \mathcal{I}}$. It maintains the best configuration found so far for each candidate, E_i^{best}, and an index set of candidates that

[1] In our exposition and experiments, we will assume that the Policy Tuner is invoked at the end of each day and optimizes over the day ahead, but this is not strictly required.

Algorithm 4 The Spinup Scheduler (ScheduleSpinups).

Inputs: Initial execution configurations $\mathcal{E} = \{E_i^0\}_{i \in \mathcal{I}}$, training workloads W^{tr}, validation workloads W^{val}, target SLO violation rate τ

Configuration: Maximum spinups η, aggregation function α, eligible cluster sizes $\mathcal{S}$, maximum attempts per round ϕ

Outputs: Optimized config $E^\star$ with scheduled spinups

1: $E_i^{\text{best}} \leftarrow E_i^0$ for all $i \in \mathcal{I}$
2: alive_at_all $\leftarrow \mathcal{I}$
3: **for** $r \in [0, \ldots, \eta)$ **do**
4: $\quad \{\Lambda_i\}_{i \in \text{alive\_at\_all}} \leftarrow \text{SimulateBatch}(W^{\text{tr}}, \{E_i^{\text{best}}\}_{i \in \text{alive\_at\_all}})$
5: $\quad$ **for** $i \in$ alive_at_all **do**
6: $\quad\quad v_i^{\text{base}}, k_i^{\text{base}} \leftarrow \text{AggPerf}(\Lambda_i, \alpha)$
7: $\quad\quad \Gamma_i \leftarrow \text{FindGoodSpinupTime}(\Lambda_i, \tau_{\text{target}})$
8: $\quad\quad$ **if** $\Gamma_i = \emptyset$ **then** alive_at_all $\leftarrow$ alive_at_all $\setminus \{i\}$
9: $\quad$ attempt$_i \leftarrow 0$ for all $i \in$ alive_at_all
10: $\quad$ alive_this_round $\leftarrow$ alive_at_all
11: $\quad$ **while** alive_this_round $\neq \emptyset$ **do**
12: $\quad\quad$ **for** $(i, s) \in$ alive_this_round $\times \mathcal{S}$ **do**
13: $\quad\quad\quad E_{i,s} \leftarrow \text{AddSpinup}(E_i^{\text{best}}, \text{time} = \Gamma_i(\text{attempt}_i), \text{size} = s)$
14: $\quad\quad \{\Lambda_{i,s}\} \leftarrow \text{SimulateBatch}(W^{\text{tr}}, \{E_{i,s}\})$
15: $\quad\quad$ **for** $i \in$ alive_this_round **do**
16: $\quad\quad\quad v_{i,s}, k_{i,s} \leftarrow \text{AggPerf}(\Lambda_{i,s}, \alpha)$ for all $s \in \mathcal{S}$
17: $\quad\quad\quad E_i^{\text{b}} \leftarrow \text{BestWithTarget}(\{(v_{i,s}, k_{i,s})\}_s \cup \{(v_i^{\text{base}}, k_i^{\text{base}})\}, \tau_{\text{target}})$
18: $\quad\quad\quad$ **if** $E_i^{\text{b}} \neq E_i^{\text{best}}$ **then**
19: $\quad\quad\quad\quad E_i^{\text{best}} \leftarrow E_i^{\text{b}}$
20: $\quad\quad\quad\quad$ alive_this_round $\leftarrow$ alive_this_round $\setminus \{i\}$
21: $\quad\quad\quad$ **else if** attempt$_i + 1 < \min(|\Gamma_i|, \phi)$ **then**
22: $\quad\quad\quad\quad$ attempt$_i \leftarrow$ attempt$_i + 1$
23: $\quad\quad\quad$ **else**
24: $\quad\quad\quad\quad$ alive_this_round $\leftarrow$ alive_this_round $\setminus \{i\}$
25: $\quad\quad\quad\quad$ alive_at_all $\leftarrow$ alive_at_all $\setminus \{i\}$
26: $\{\Lambda_i^{\text{val}}\} \leftarrow \text{SimulateBatch}(W^{\text{val}}, \{E_i^{\text{best}}\}_{i \in \mathcal{I}})$
27: $v_i^{\text{val}}, k_i^{\text{val}} \leftarrow \text{AggPerf}(\Lambda_i^{\text{val}}, \alpha)$ for all $i \in \mathcal{I}$
28: $E^\star \leftarrow \text{BestWithTarget}(\{(v_i^{\text{val}}, k_i^{\text{val}})\}_{i \in \mathcal{I}}, \tau)$
29: **return** $E^\star$

remain eligible for further spinups, alive_at_all (lines 1–2). The outer loop performs up to η greedy rounds (line 3). At the beginning of each round, all still-active candidates are simulated on the training workloads in a single batch (line 4). For each candidate, the scheduler aggregates its current SLO violation rate and cost (across the training workloads), then calls FindGoodSpinupTime (Algorithm 5) to obtain a ranked list Γ_i of promising spinup times (lines 5–7). Candidates with no promising times are removed from alive_at_all (line 8).

For the remaining candidates, the scheduler searches for one spinup to add in the current round. It initializes each candidate's attempt counter and places all active candidates into alive_this_round (lines 9–10). The inner loop then evaluates candidates in attempt waves (lines 11–25). In each wave, every still-alive candidate i is paired with every eligible cluster size $s \in \mathcal{S}$, producing a trial configuration that adds a spinup at the candidate's currently attempted time $\Gamma_i(\text{attempt}_i)$ with size s (lines 12–13). These trial configurations are simulated in parallel on the training workloads, before we aggregate their violation rates and costs (lines 14–16).

The scheduler then produces, for each candidate, a local best E_i^b among all new configurations and the current baseline using BestWithTarget (line 17). If it differs from the previous best, it is promoted to E_i^{best} and the candidate leaves the current round because it has accepted a spinup (lines 18-20). If no cluster size improves the candidate at the current time, the scheduler advances to the next promising time in Γ_i, up to the per-round attempt limit ϕ (lines 21–22). If the candidate exhausts all allowed attempts without finding an improvement, it is removed both from the current round and from future rounds (lines 24–25). Thus, each outer round adds at most one spinup per candidate, and candidates stop being considered once no useful spinup can be found.

After the greedy training phase, we evaluate the best configuration for each initial candidate on the validation workloads (lines 26–27). We then select the configuration with the best validation cost subject to the target SLO violation rate τ and return it as $E^\star$ (lines 28–29). The batching in Algorithm 4 does not change the greedy search logic; it simply improves simulation throughput.

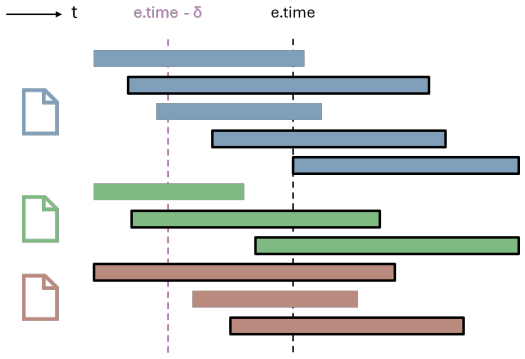


Figure 5: Candidate spinup time determination. Here, queries with bold outlines violated their SLOs. The indicated time is the first candidate spinup time for $\tau_{\text{target}} = 0.6$ and $k = 3$.

Algorithm 4 Computational Time Complexity. Let B the maximum parallel simulation batch size and s the maximum simulation duration. Then the overall computational time complexity of ScheduleSpinups is $O\big(|\mathcal{I}| \cdot \frac{s}{B} \cdot \big(\eta \cdot |\mathcal{S}| \cdot \phi \cdot |W^{\text{tr}}| + |W^{\text{val}}|\big)\big)$.

7.4.2 Finding Good Scheduled Spinup Times. Per Section 7.4.1, the Spinup Scheduler needs to determine promising times for scheduled spinups. Algorithm 5 achieves this in a single linear scan. It begins by building a unified sorted timeline of query arrival and completion events across all per-training-workload simulation logs for the same execution configuration, pre-gathering per-query SLO violation information (lines 1–2). It then initializes per-workload running-sum accumulators V_i, N_i, delinquency flags D_i, the global delinquent-workload count n_D, and the epoch-tracking state (lines 3–4). For each event e, the running sums for the affected workload are updated (lines 6–10), and the workload's delinquency flag and n_D are refreshed (lines 11–13). Using running sums means that SLO adherence is evaluated in $O(1)$ per event. Zero-length intervals

Algorithm 5 Congestion point detection (FindGoodSpinupTime).

Inputs: Per-workload simulation logs $\{\Lambda_i\}$, target SLO violation rate τ_{target}

Configuration: Delinquency threshold k, spinup delay δ, spacing $\sigma_{\min}$

Outputs: List Γ of promising spinup placement times.

1: events $\leftarrow$ All query arrival/completion events from simulation logs $\{\Lambda_i\}$
2: events $\leftarrow$ Sort(events, {time, Asc)}
3: $V_i \leftarrow 0$, $N_i \leftarrow 0$, $D_i \leftarrow$ false for all i
4: $n_D \leftarrow 0$; inEpoch $\leftarrow$ false; $\Gamma \leftarrow [\,]$
5: **for** $j = 0$ **to** |events| $- 2$ **do**
6: $e \leftarrow$ events$[j]$
7: **if** e.type = Start **then**
8: $V_{e.i}$ += $e.v$; $N_{e.i}$ += 1
9: **else**
10: $V_{e.i}$ −= $e.v$; $N_{e.i}$ −= 1
11: $d' \leftarrow \frac{V_{e.i}}{N_{e.i}} \geq \tau_{\text{target}}$
12: **if** $d' \neq D_{e.i}$ **then**
13: n_D += $(d' \; ? +1 : -1)$; $D_{e.i} \leftarrow d'$
14: **if** events$[j{+}1]$.time = e.time: **continue**
15: **if** $n_D \geq k$ **and not** inEpoch **then**
16: inEpoch $\leftarrow$ true
17: $t_{\text{cand}} \leftarrow e.\text{time} - \delta$
18: **else if** $n_D < k$ **and** inEpoch **then**
19: inEpoch $\leftarrow$ false
20: **if** $\{t \in \Gamma \mid |t - (t_{\text{cand}})| < \sigma_{\min}\} = \emptyset$ **then**
21: Γ.Append(t_{cand})
22: **if** inEpoch **and** $\{t \in \Gamma \mid |t - (t_{\text{cand}})| < \sigma_{\min}\} = \emptyset$ **then**
23: Γ.Append(t_{cand})
24: **return** Γ

between simultaneous events are skipped (line 14). When n_D first reaches the threshold k, a *congestion epoch* begins and a candidate spinup time is recorded δ seconds earlier (lines 15–17); when n_D drops back below k, the epoch ends and the candidate spinup time is appended to Γ only if it is at least $\sigma_{\min}$ ahead of the previous candidate (lines 18–21). Any open epoch at the end of the timeline is closed after the loop (lines 22–23). The returned list Γ (line 24) is implicitly sorted in ascending order by placement time.

Figure 5 shows an example with 3 simulation logs, with queries colored blue, green and red respectively. Bold outlines indicate SLO violations. For $\tau_{\text{target}} = 0.6$ and $k = 3$, all 3 workloads have more than 60% of their running queries violating the SLO once the fifth blue query arrives, suggesting a candidate spinup time δ s earlier.

Algorithm 5 Computational Time Complexity. Let $Q = \sum_i |\Lambda_i|$, so that |events| $= 2Q$. The event timeline is sorted and then processed once, so that FindGoodSpinupTime is $O(Q \log Q)$.

7.5 Configuration Tuner

After we determine the scheduled spinups, the Configuration Tuner optimizes the parameters of the Autoscaler. It first generates a set of candidate execution configurations using a configurable strategy; supported options include `grid` (exhaustive cross-product of specified values), `random` (budget-limited random sample from the grid), `coordinate_descent` (iterative per-parameter optimization), and `adaptive_batch` (progressive widening with early stopping). It then evaluates all candidate configurations on the training workloads in parallel via SimulateBatch and retains only the top-k with respect to a configurable aggregate metric over the training workloads (e.g. mean). These are re-evaluated (using SimulateBatch) on the validation workloads, and the configuration with the best aggregate validation performance is returned.

8 EVALUATION

We implemented AutoSLO in around 26k lines of Python [35]. After explaining our evaluation setup (Section 8.1), we will present end-to-end experiments (Section 8.2) and assess the effectiveness (Sections 8.7- 8.3) and efficiency (Section 8.8) of each component. In Tables 2– 5, VR denotes the SLO violation rate and bold/underline indicates the best/second-best VR per row.

8.1 Setup

8.1.1 Configuration. We use a machine with two 20-core 2.10 GHz Intel Xeon Gold 6230 CPUs [23] and 256 GiB of memory, running Linux 6.9.7-arch1-1. From this machine, we use `psycopg2` and `boto3` to interact with Amazon Redshift Serverless. For continuity, we will refer to workgroups as "clusters" below. We use the current Amazon Redshift Serverless price for `us-east-1` ($0.375/RPU/$h$), where an RPU is the unit of cluster size.

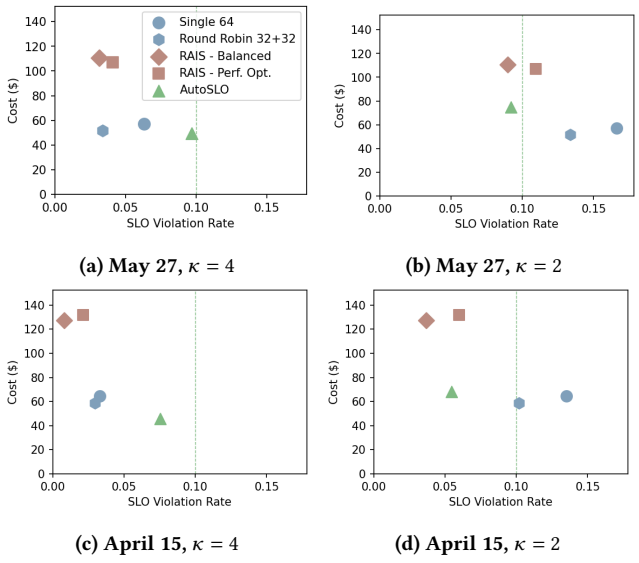
8.1.2 Workloads. All experiments are executed against TPC-DS SF 1000 (1 TB). Several of our experiments use a workload consisting of 3 copies of one query per benchmark query template, for a total of 297 queries, shuffled and issued according to a Poisson process with rate λ. We will call such workloads BaseWorkload(λ). Other experiments will each describe the workload(s) they use.

8.1.3 SLOs. We run BaseWorkload in a closed loop against a 16-RPU Amazon Redshift Serverless cluster and record the maximum observed latency $\ell_{\text{baseline}}(t)$ per query template t. Below, we let the SLO of each query q of template t be $\kappa \cdot \ell_{\text{baseline}}(t)$, with higher κ indicating looser/easier to satisfy SLOs.

8.2 End-to-end Effectiveness

We begin by evaluating AutoSLO end-to-end using Redbench [56], which creates realistic workloads based on the real production traces in Redset [55]. Redset includes query timing and complexity information for real Amazon Redshift customer workloads, but lacks actual query texts and data. Redbench bridges this gap by mapping each Redset query to appropriate query texts from TPC-DS [44]. We obtain an executable version of the SELECT workload faced by a particular Redset cluster (provisioned cluster 157) using Redbench's join matching algorithm. For practical purposes, we compress the interarrival times in the workload by a factor of 6 during simulation/execution, so that each day's workload takes 4 hours to execute. In the Policy Tuner, this compression is only applied in the Batch Simulator (i.e. forecasting uses real times).

Based on this workload, we derive 4 experimental scenarios as the cross product of two variables, as shown in Figure 6. First, we vary the workload day: we use the last Monday (May 27, 2024) and



(a) May 27, $\kappa = 4$ **(b) May 27, $\kappa = 2$**

(c) April 15, $\kappa = 4$ **(d) April 15, $\kappa = 2$**

Figure 6: End-to-end performance of AutoSLO.

middle Monday (April 15, 2024) of the workload, to ensure non-overlapping recent workload histories in the Policy Tuner. Then, we examine two different levels of SLO difficulty, for $\kappa \in \{2, 4\}$. Across scenarios, our SLO violation rate target per Definition 3 is $\tau_{\text{target}} = 0.1$, shown as a dashed green line.

We compare the performance of AutoSLO (green triangle), after using the Policy Tuner over 30 days of past data, against two families of baselines. In blue, we have a pair of naive baselines using a total of 64 RPU each, either in a single cluster (circle) or in two clusters with 32 RPU each, with round-robin query routing (hexagon). In red, we have a single cluster using Redshift Serverless AI-driven Scaling [39], with the price-performance slider set to either "balanced" (position 50, diamond) or "high performance" (position 100, square), and a specified maximum of 128 RPU[2].

As evident, AutoSLO provides the best performance across scenarios, meeting τ_{target} for all 4 scenarios and reducing cost by a mean of 26.4% compared to the next-best baseline per scenario. Compared to the only other baseline that meets τ_{target} in all 4 scenarios (RAIS - Balanced), AutoSLO reduces cost by a mean of 49.6%.

Per Section 1, neither family of baselines allows explicitly specifying SLOs or τ_{target}, which translates to ineffective cost-performance tradeoffs. While the SLOs are loose ($\kappa = 4$, Figures 6a and 6c), all baselines meet τ_{target} but remain unaware of it, spending additional resources to reduce latency in a regime where it no longer matters. When the SLOs tighten ($\kappa = 2$, Figures 6b and 6d), no baseline can be informed, since they are SLO-unaware; they each act the same as before, leading to higher SLO violation rates, with the naive baselines all missing τ_{target}. AutoSLO instead adapts, increasing spending just enough to meet τ_{target}.

[2]We observe that throughout the execution of our workloads, RAIS bills for the full 128 RPU (according to the billing timer of Section 2) regardless of the slider setting.

8.2 End-to-end SLO Adherence: Findings

AutoSLO can cost-effectively maintain latency SLOs on realistic workloads at the specified target level, reducing cost by a mean of 49.2% against the cheapest RAIS baseline per scenario.

8.3 Latency Predictor (Iconq+)

8.3.1 Variants. To assess our Latency Predictor, we compare four model variants in an ablation study: Iconq includes no changes [58]; +Size enhances the interaction feature vectors with cluster size information (Section 4.2); +Censored additionally trains the model on censored observations sampled with probability 0.5% (Section 4.3); finally, Iconq+ also supports incremental predictions (Section 4.4).

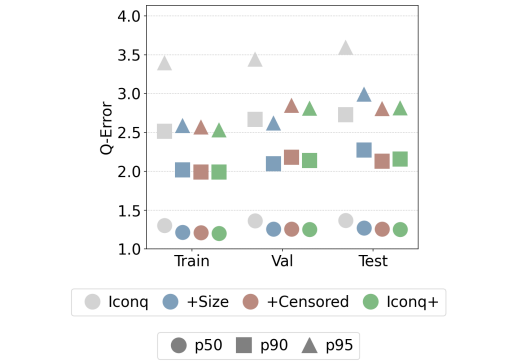
8.3.2 Training Details. We train each variant on data obtained by executing BaseWorkload for $\lambda \in [0.05, 0.1, 0.2, 0.5, 1.0]$ on each of 4, 8, 16 and 32 RPU. On each cluster size, we also execute four "phased" versions of BaseWorkload, alternating between x queries at $\lambda = 0.05$ and 10 queries at λ_{high}, for $x \in \{40, 15\}$ and $\lambda_{high} \in \{0.2, 0.5\}$. For practicality, we impose an one-hour per-query timeout when executing these workload runs. For all four variants, we also perform two training stability modifications compared to the original Iconq paper [58]. First, we only train the underlying Stage [57] model (used to derive neighbor-unaware latency predictions included in interaction feature vectors as query complexity proxies) on isolated query executions, i.e., executions with no neighbors. This makes the resulting predictions better reflect intrinsic query complexity rather than interference from concurrent queries. Second, we interpret data labels and model latency predictions in *logarithmic space*, guaranteeing positivity after re-exponentiation and aligning naturally with multiplicative error. This means we no longer need to heavily penalize negative/tiny predictions in the loss function, which can now use a simpler Q-error-based objective.

8.3.3 Results. Figure 7a evaluates the accuracy of each model. It shows the 50th, 90th and 95th percentile of the Q-error achieved by each variant on each of its folds. As evident, adding the cluster-size-related features in +Size offers noticeable benefits on the test set, in addition to being necessary for cross-cluster-size comparisons when routing. +Censored extends these benefits further, especially at p90 and p95, with Iconq+ achieving similar accuracy.

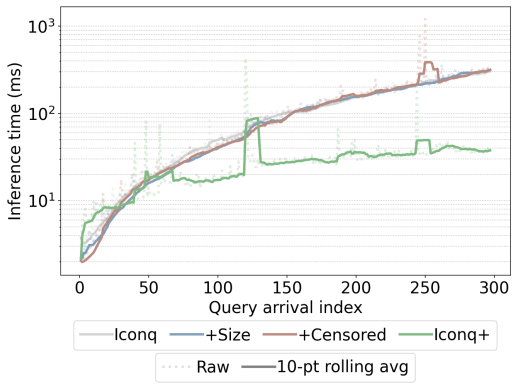
Figure 7b concerns the efficiency of the different variants on BaseWorkload with $\lambda = 0.1$. For each query arrival, we predict its own latency and update the latency predictions of previous queries, assuming no query has completed yet. This is not reflective of a real execution of this particular workload, but this experiment is only stress-testing total inference latency per arrival under increasing load. As evident, inference time grows with increasing congestion, but Iconq+ can use incremental inference to sustain sub-40 ms inference latency. In contrast, every other variant requires around 300 ms by the end of the workload.

8.3 Latency Predictor (Iconq+): Findings

The adaptations of Iconq+ are effective: adding cluster size features and censored observations noticeably prediction accuracy, while incremental inference keeps inference time manageable.



(a) Accuracy



(b) Efficiency

Figure 7: Iconq+ delivers superior accuracy and efficiency.

8.4 Query Router

We start by assessing the performance of the Query Router. We compare three query routing approaches. Round-Robin cycles among the active clusters, as a naive baseline. Stage + Cost Opt. uses the underlying Stage [57] model of our trained Iconq+ instance to derive concurrency-unaware latency predictions on each active cluster and then routes each query to a cluster where its SLO will be met, breaking ties by cost. Such concurrency-agnostic routing mirrors the approach of published descriptions from cloud vendors [39, 49]. Ours (Iconq+ + Cost Opt.) represents our approach as described in Section 5, where the concurrency-aware Iconq+ model is used and the risk to the SLOs of running queries is also assessed.

We evaluate each approach in 8 scenarios, derived as follows: (i) we run BaseWorkload with either $\lambda = 0.1$ or $\lambda = 0.2$; (ii) we define SLOs using either $\kappa = 4$ or $\kappa = 2$; and (iii) we either route between two clusters with 16 RPU each ($C = [16, 16]$), or between one cluster each with 32 RPU and 16 RPU ($C = [32, 16]$).

Table 2 shows the results. Query latency prediction with Stage already outperforms Round-Robin, reducing the SLO violation rate by a mean of 24.3%. It also reduces cost by a mean of 19.3%, since the Query Router also considers cost. However, using Iconq+ can improve even further over Round-Robin, achieving a mean SLO violation rate reduction of 47.8% while still lowering cost.

Table 2: Query Router evaluation.

λ	κ	C	Round Robin		Stage + Cost.Opt.		Iconq + Cost.Opt.	
			VR	Cost	VR	Cost	VR	Cost
0.1	4	[32, 16]	0.0269	$15.78	**0.0135**	$11.01	0.0168	$11.17
0.1	4	[16, 16]	0.0976	$10.40	0.0370	$7.99	**0.0303**	$8.91
0.1	2	[32, 16]	0.0842	$15.78	0.0808	$10.14	**0.0505**	$12.28
0.1	2	[16, 16]	0.2088	$10.57	0.1414	$7.43	**0.1246**	$8.77
0.2	4	[32, 16]	0.1414	$7.92	0.0707	$7.56	**0.0202**	$7.60
0.2	4	[16, 16]	0.3064	$5.41	0.2761	$4.81	**0.1919**	$5.29
0.2	2	[32, 16]	0.1919	$8.13	0.1919	$6.92	**0.1044**	$7.71
0.2	2	[16, 16]	0.4512	$5.48	0.5118	$5.20	**0.3300**	$4.98
Mean Δ			—	—	-24.3%	-19.3%	**-47.8%**	-12.9%

Table 3: Spinup Size Selector steady-state evaluation.

λ	κ	C	No-Op		Horizontal		Vertical w/Ours		Add Cluster w/Ours	
			VR	Cost	VR	Cost	VR	Cost	VR	Cost
0.1	4	[8]	0.9833	$2.81	0.8167	$1.99	0.2000	$1.79	**0.0167**	$2.16
0.1	4	[16]	0.1000	$1.65	**0.0000**	$2.67	0.0833	$1.59	**0.0000**	$2.35
0.1	2	[8]	0.9833	$2.86	1.0000	$2.04	**0.0167**	$3.40	0.1000	$2.09
0.1	2	[16]	0.3667	$1.74	0.0667	$3.14	**0.0167**	$3.40	0.0833	$2.81
0.2	4	[8]	1.0000	$3.27	1.0000	$3.55	**0.0333**	$3.11	**0.0333**	$2.88
0.2	4	[16]	0.8333	$2.21	0.1167	$1.83	0.0333	$2.14	**0.0000**	$2.71
0.2	2	[8]	1.0000	$3.31	0.9500	$3.34	**0.0667**	$3.12	**0.0667**	$3.15
0.2	2	[16]	0.9000	$2.05	0.4167	$2.26	0.0667	$2.29	**0.0500**	$2.75
Mean Δ			—	—	-42.7%	+11.0%	-83.6%	+9.1%	**-93.7%**	+11.8%

8.4 Query Router: Findings

Our Query Router outperforms the alternatives, reducing SLO violation rate by a mean of 47.8% and cost by a mean of 12.9%.

8.5 Autoscaler – Spinup Size Selector

We next focus on the cluster sizes selected by the Scaling Controller within the Autoscaler. We compare four approaches to modifying the active cluster set. No-Op makes no changes to the active clusters. Horizontal spins up an additional cluster of the same size as the existing one, using our Query Router to route between the two. Add Cluster w/ Ours represents our approach described in Section 6. Vertical w/ Ours uses the simulation-based decision-making of our method, but instead of considering what single cluster to *add*, it considers what single cluster to *have* (i.e. it stop routing to the existing cluster once the new one becomes available).

We evaluate each approach in 8 scenarios, derived as in Section 8.4, but we start with only a single cluster of size either 8 or 16 RPU. We execute 20% of the workload queries, at which point we *forcibly trigger* autoscaling. We then let the following 60% of the workload elapse, for scaling to take effect and any congestion-affected queries around the autoscaling point to drain. We report the SLO violation rate and cost over the last 20% of the workload, once a steady state has been reached.

The results are presented in Table 3. The cluster size selected by the Spinup Size Selector outperforms horizontal scaling: Add Cluster w/ Ours achieves a mean SLO violation rate reduction of

Table 4: Autoscaler Spinup Trigger evaluation.

λ	κ	C	No-Op		Queue@32		Observed		Ours	
			VR	Cost	VR	Cost	VR	Cost	VR	Cost
0.1	4	[8]	0.9697	$4.56	0.4276	$6.10	0.4007	$5.88	**0.3199**	$7.31
0.1	4	[16]	0.2896	$5.31	0.3064	$5.09	**0.2222**	$7.18	0.2626	$7.83
0.1	2	[8]	0.9865	$4.71	0.4242	$8.67	0.8013	$6.38	**0.2458**	$11.10
0.1	2	[16]	0.4882	$5.24	0.4916	$5.25	0.2727	$8.68	**0.2660**	$8.74
0.2	4	[8]	0.9899	$4.13	**0.2896**	$5.54	0.9327	$5.04	0.3098	$5.19
0.2	4	[16]	0.8114	$3.71	0.4141	$5.72	**0.3434**	$5.91	0.4007	$4.95
0.2	2	[8]	0.9966	$4.16	0.4680	$5.37	0.5791	$5.91	**0.4040**	$6.02
0.2	2	[16]	0.9495	$3.65	0.4882	$5.45	0.4680	$5.77	**0.3670**	$7.27
Mean Δ			—	—	-41.0%	+35.1%	-37.6%	+43.3%	**-54.6%**	+64.1%

93.7%, compared to 42.7% for Horizontal, while the two methods impact cost similarly. Relying on our algorithm for vertical scaling is also promising, with Vertical w/ Ours achieving a mean SLO violation rate reduction of 83.6%. However, Add Cluster w/ Ours performs best on average, making effective use of the pre-existing cluster.

8.5 Autoscaler – Spinup Size Selector: Findings

The Spinup Size Selector recommends appropriate cluster sizes. Our approach, where we use the new cluster alongside the existing one, achieves a mean SLO violation rate reduction of 93.7%.

8.6 Autoscaler – Spinup Trigger

In the previous experiment, we focused on *which* cluster the Autoscaler spins up, once triggered; we now also consider *when* it is triggered. We compare four approaches to triggering autoscaling, all of which then use the standard Spinup Size Selector. No-Op is never triggered. Queue@32 is triggered whenever there are 32 or more outstanding queries, approximating queuing-based autoscaling schemes present in Auto-WLM [49] and RAIS [39]. Observed is similar to Algorithm 2 but without R (line 5); it only acts on *observed* latencies. Finally, Ours represents our approach described in Section 6.2.2, which augments Observed with the projected SLO violation status of currently-running queries.

We use the same 8 experimental scenarios as Section 8.5. Since each approach can decide when and what to spin up, we compare SLO violation rate and cost throughout the whole workload. As shown in Table 4, Ours clearly outperforms the alternatives on average, reducing the SLO violation rate by a mean of 54.6%.

8.6 Autoscaler – Spinup Trigger: Findings

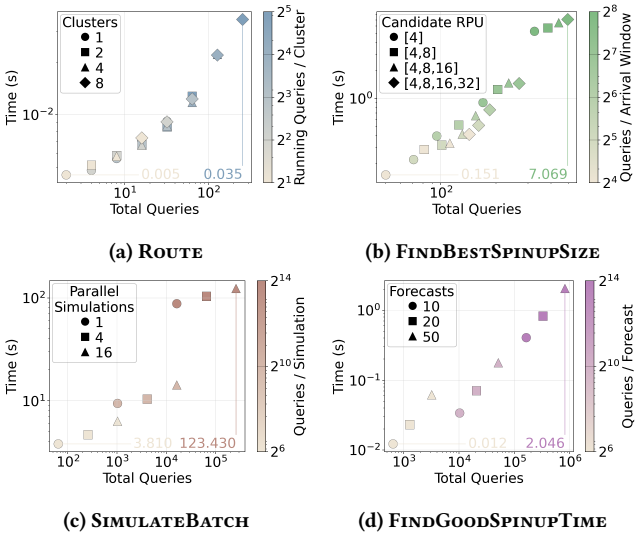
Our Autoscaler spinup trigger, combining observed and projected SLO violations, outperforms the alternatives and delivers a mean SLO violation rate reduction of 54.6%.

8.7 Policy Tuner

We next evaluate the impact of the Policy Tuner on the downstream performance of the execution configuration it produces. In particular, for the day/κ scenarios used in Section 8.2, we use the Policy Tuner to optimize a configuration on 0, 1, 7 or 30 days of past data. We then use each optimized configuration to execute the target workload. As seen in Table 5, the Policy Tuner can leverage even a single day of workload history to reduce the SLO violation rate by a mean of 44.6%, increasing cost by a mean of just 11.8%. With access to 30 days of history, the mean SLO violation rate reduction further improves to 46.1%, but most of the tuning benefit can be already reaped with minimal history.

Table 5: Policy Tuner evaluation.

Day	κ	No Past Data		Past 1 Day		Past 7 Days		Past 30 Days	
		VR	Cost	VR	Cost	VR	Cost	VR	Cost
5/27	4	0.1997	$41.47	0.1149	$46.14	**0.0639**	$51.75	0.0970	$49.25
5/27	2	0.2162	$61.72	0.1092	$71.39	0.1293	$68.38	**0.0920**	$74.89
4/15	4	0.0833	$42.64	**0.0664**	$44.15	0.0848	$45.65	0.0752	$45.82
4/15	2	0.1600	$59.80	**0.0546**	$69.78	0.0575	$68.11	**0.0546**	$68.01
Mean Δ		—	—	-44.6%	+11.8%	-42.6%	+14.1%	**-46.1%**	+15.3%



(a) Route (b) FindBestSpinupSize (c) SimulateBatch (d) FindGoodSpinupTime

Figure 8: Efficiency evaluation of key AutoSLO algorithms.

8.8 Efficiency

We close with efficiency micro-experiments. In Figure 8, each point represents the median execution time across 10 repetitions.

8.8.1 Query Router Efficiency. The Query Router is invoked in the critical path of each query, so it needs to be highly efficient. We assess this by sweeping two variables: the number of active clusters and the number of running queries per cluster. As evident from Figure 8a, the Query Router achieves a decision-making time under 10 ms in most scenarios, with a maximum of only 35 ms.

8.8.2 Autoscaler Efficiency. The Autoscaler runs in a background thread, as described in Section 6. Still, it must be reasonably efficient for the scaling decision to remain timely. We assess its efficiency

by sweeping two variables: the number of candidate cluster sizes and the arrival rate of queries in the Arrival Window. As seen in Figure 8b, its decision-making time is around 7 s even in the most challenging case, providing fast responsiveness for reactive scaling.

8.8.3 Policy Tuner Efficiency. The Policy Tuner is invoked periodically, as described in Section 7, so its overhead can be somewhat higher. Still, it must not consume excessive resources that could be otherwise used for workload execution. Aside from bookkeeping, two expensive functions are used by the Spinup Scheduler and the Configuration Tuner: SimulateBatch and FindGoodSpinupTime.

For SimulateBatch, we sweep two variables: the number of queries in each simulated workload and the number of workloads simulated in parallel. As seen in Figure 8c, batch simulation parallelizes well and finishes in around 2 minutes even with over 15,000 queries, well within the performance requirements of an infrequent, background Policy Tuner invocation. For FindGoodSpinupTime, we sweep the number of queries in each forecasted workload and the number of forecasts considered. As Figure 8d shows, the algorithm completes in slightly over 2 s in the worst case.

8.8 Efficiency: Findings

Each critical algorithm of AutoSLO is efficient, meeting the latency requirements of its operating timescale.

9 RELATED WORK

9.1 SLOs for Cloud Data Systems

As shown in the top part of Table 6, no prior work on cloud data systems offers SLOs and exhibits all three **key desired behaviors**.

Only offline planning. Some works **plan** for the future workload without adjusting resources online or reacting at routing time. ResTune [63] reduces resource utilization by replaying the workload on a replica and tuning knobs using constrained optimization. WiseDB [33] trains a decision model on pre-selected query templates to derive fixed template-to-cluster routing decisions.

Only online adjustment. On the flip-side, some works only **adjust** resources based on online load, neither reacting to this load at routing time nor planning future resource allocation. Das et al. [16] use an adapted version of the token bucket algorithm [54] to balance the estimated resource demand of each customer. SLAOrchestrator [40–42] suggests and enforces per-query latency SLOs over workloads of ad hoc SELECT-PROJECT-JOIN (SPJ) queries, by having users select their desired cost/performance tier among provided options. Most recently, BRAD [62] focuses on how to select, configure and route among *different engines*. An additional number of works focuses on *vendor-side* optimization, packing customer workloads onto machines [7, 15, 28, 31, 37, 38, 53, 59], unlike our focus on optimizing the requirements of each and every workload. Earlier works have also evaluated alternatives like having the user fully define the autoscaling triggers and exact corrective actions [48, 64] or using reinforcement learning [34].

No explicit SLOs. The currently published mechanism of Amazon Redshift, as introduced in Auto-WLM [49] and refined in RAIS [39], balances **adjusting** and **planning** but does not optimize for explicit SLOs, nor does it **react** to observed load through concurrency-aware predictions. Instead, autoscaling is triggered by query queue length, with a qualitative price-performance sensitivity slider controlling the aggressiveness. Query routing relies on concurrency-unaware latency predictions, as described in Stage [57].

Fixed resources. A final class of systems assumes a fixed amount of compute per customer. Some support SLOs but do not **react** to query interactions when routing [12–14, 29, 60]; others **react** to query-level interactions but do not optimize for SLOs [2, 22, 45, 47, 58].

Table 6: Coverage of related work.

Section 9.1: SLOs for Cloud Data Systems

Category	Has SLOs	Plans offline	Adjusts online	Reacts when routing
Only offline planning [33, 63]	✓	✓	✗	✗
Only online adjustment [7, 15, 16, 26, 28, 31] [34, 37, 38, 40–42] [48, 53, 59, 62, 64]	✓	✗	✓	✗
No explicit SLOs [39, 49, 57]	✗	✓	✓	✗
Fixed, no interference [12–14, 29, 60]	✓	✗	✗	✗
Fixed, no SLOs [2, 22, 45, 47, 58]	✗	✗	✗	✓

Section 9.2: SLOs for Other Cloud Systems

Domain	Unlike queries the scheduled tasks have...
Serverless Computing [1, 19, 24]	Opaque, function-level behavior
Application Placement [9, 17, 18, 32, 43, 46]	Longer lifetimes, SLOs over inbound requests
ML Model Serving [10, 25, 36]	Similar computational demands
LLM Serving [8, 20, 21, 27, 30, 61, 65]	Regular structure, only prefix-based caching

9.2 SLOs for Other Cloud Systems

Per the bottom part of Table 6, related work on SLOs for other cloud systems can also be instructive, although not directly applicable.

Serverless Computing. One area of focus is scheduling tasks in serverless function-as-a-service offerings (e.g. AWS Lambda [6]). In this domain, re-executing the *same* function on warm resources can be faster, but there is no notion of a cross-function "cache". This is unlike database queries, where the same cached data can be useful to many syntactically different queries. Nevertheless, some concerns are similar. Hermod [24] balances function performance and resource efficiency through an execution-time-agnostic, but cost- and load-aware hybrid policy, based on simulation-derived insights. FaaSCache [19] casts function keep-alive as a caching problem, reducing cold-start overhead using a variant of the Greedy-Dual [11] policy. Palette load balancing [1] tags each invocation with a user-provided locality hint (*color*); scheduling then maintains a map

from colors to instances. Such works prize generality and treat the computational demands of each function as opaque. However, when dealing with database queries, latency prediction is more tractable and a critical source of scheduling-relevant information.

Application Placement. Works like Paragon [17], Quasar [18] and Mage [46] optimize application placement (called *scheduling* in this literature) in datacenters with heterogeneous platforms and resources. They treat resource allocation and assignment as a recommendation problem, using limited profiling to balance hardware utilization and the desired performance. Works like Heracles [32], PARTIES [9] and CLITE [43] explore co-locating such latency-sensitive applications on the same resources as "background", throughput-focused work. In this context, SLOs are defined for *user requests*, while the scheduled entity is the *long-running application*. This is different than routing each individual SLO-bound database query.

ML Model Serving. Another line of work studies resource allocation for cloud-deployed ML models subject to inference latency SLOs. For example, Mendoza et al. [36] discuss an interference-aware model co-location policy, based on a random forest regressor trained on offline model/hardware profiling. Mudi [10] further attempts to co-locate inference tasks with model training while minimizing interference, by estimating inference latency as a piece-wise linear function of allocated resources. CascadeServe [25] uses model output quality as an additional optimization knob, optimizing the design and placement of *model cascades* that invoke higher-latency inference for "harder" requests only. These systems exploit the relative uniformity of inference requests to a given model, whereas analytical queries can differ substantially in operators, data access patterns, and resource demands

LLM Serving. LLM serving introduces more per-request variability and state management (e.g. KV-cache entries) compared to traditional model serving. Several works deal with these issues for individual requests: Orca [61] proposed dealing with variable-length decode phases using iteration-level scheduling and selective batching. PagedAttention [27] uses OS-inspired techniques to reduce KV-cache fragmentation and improve throughput. DistServe [65] and TetriInfer [21] explore prefill-decode disaggregation, so that the resource allocation for each phase can be separately optimized. SOLA [20] seeks to balance the different SLOs of prefill and decode through state-aware scheduling. HotPrefix [30] addresses the increasing prevalence of requests with the same prefix (e.g. system prompt) by tracking and leveraging the hotness of each prefix while managing the KV-cache. More recent works look at optimizing compound AI systems built around LLMs. Murakkab [8] proposes a declarative abstraction that enables under-the-hood efficiency-driven optimizations. However, LLM requests still have a comparatively regular structure (centered on prefill/decode) and only prefix-based caching. Database queries have more diverse plans and richer forms of data reuse, leading to a much larger decision space.

10 CONCLUSION

We presented AutoSLO, a framework for cost-efficiently meeting latency SLOs on a multi-cluster cloud data warehouse, comprised of a proactive Policy Tuner that **plans** infrastructure scaling, a reactive Autoscaler that **adjusts** resources based on workload fluctuations and an online Query Router that **reacts** to concurrent load. We showed that these components can work together to successfully meet latency SLOs of varying strictness, reducing cost by a mean of 26.4% compared to the per-scenario next-best baseline on realistic Redbench workloads. Component-level evaluations showed that the Query Router and Autoscaler respectively reduce SLO violation rates by a mean of 47.8% and 93.7%, relative to their corresponding alternatives. Finally, we showed that the Policy Tuner can reduce the SLO violation rate by a mean of 44.6% using a single day of workload history, and that each component is efficient given its intended operating timescale.

ACKNOWLEDGMENTS

This research was supported by Amazon, Google, and Intel as part of the MIT Data Systems and AI Lab (DSAIL). This research was also sponsored by the Department of the Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. 2023. Palette Load Balancing: Locality Hints for Serverless Functions. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 365–380. https://doi.org/10.1145/3552326.3567496

[2] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. 2008. QShuffler: Getting the Query Mix Right. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, Piscataway, NJ, USA, 1415–1417. https://doi.org/10.1109/ICDE.2008.4497574

[3] Amazon Web Services. 2026. Amazon Redshift. https://aws.amazon.com/redshift/. Retrieved June 1, 2026.

[4] Amazon Web Services. 2026. Amazon Redshift Pricing. https://aws.amazon.com/redshift/pricing/. Retrieved March 17, 2026.

[5] Amazon Web Services. 2026. Datashares - Amazon Redshift. https://docs.aws.amazon.com/redshift/latest/mgmt/query-editor-v2-datashare-using.html. Retrieved March 19, 2026.

[6] Amazon Web Services. 2026. Serverless Computing - AWS Lambda. https://aws.amazon.com/lambda/. Retrieved March 20, 2026.

[7] Pankaj Arora, Surajit Chaudhuri, Sudipto Das, Junfeng Dong, Cyril George, Ajay Kalhan, Arnd Christian König, Willis Lang, Changsong Li, Feng Li, et al. 2023. Flexible Resource Allocation for Relational Database-as-a-Service. *Proceedings of the VLDB Endowment* 16, 13 (2023), 4202–4215.

[8] Gohar Irfan Chaudhry, Esha Choukse, Íñigo Goiri, Rodrigo Fonseca, Adam Belay, and Ricardo Bianchini. 2025. Towards Resource-Efficient Compound AI Systems. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems* (Banff, AB, Canada) *(HotOS '25)*. Association for Computing Machinery, New York, NY, USA, 218–224. https://doi.org/10.1145/3713082.3730377

[9] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3297858.3304005

[10] Wenyan Chen, Chengzhi Lu, Huanle Xu, Kejiang Ye, and Chengzhong Xu. 2025. Multiplexing Dynamic Deep Learning Workloads with SLO-awareness in GPU Clusters. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 589–604. https://doi.org/10.1145/3689031.3696074

[11] Ludmila Cherkasova. 1998. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy.* Hewlett-Packard Laboratories, Palo Alto, CA, USA.

[12] Yun Chi, Hakan Hacígümüş, Wang-Pin Hsiung, and Jeffrey F. Naughton. 2013. Distribution-based Query Scheduling. *Proc. VLDB Endow.* 6, 9 (jul 2013), 673–684. https://doi.org/10.14778/2536360.2536367

[13] Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Junichi Tatemura. 2011. SLA-tree: a framework for efficiently supporting SLA-based decisions in cloud computing. In *Proceedings of the 14th International Conference on Extending Database Technology* (Uppsala, Sweden) *(EDBT/ICDT '11)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/1951365.1951383

[14] Yun Chi, Hyun Jin Moon, and Hakan Hacigümüş. 2011. iCBS: incremental cost-based scheduling under piecewise linear SLAs. *Proceedings of the VLDB Endowment* 4, 9 (2011), 563–574.

[15] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. 2011. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/1989323.1989357

[16] Sudipto Das, Feng Li, Vivek R. Narasayya, and Arnd Christian König. 2016. Automated Demand-driven Resource Scaling in Relational Database-as-a-Service. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1923–1934. https://doi.org/10.1145/2882903.2903733

[17] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.* 48, 4 (March 2013), 77–88. https://doi.org/10.1145/2499368.2451125

[18] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 127–144. https://doi.org/10.1145/2541940.2541941

[19] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 386–400. https://doi.org/10.1145/3445814.3446757

[20] Ke Hong, Xiuhong Li, Lufang Chen, Qiuli Mao, Guohao Dai, Xuefei Ning, Shengen Yan, Yun Liang, and Yu Wang. 2025. SOLA: Optimizing SLO Attainment for Large Language Model Serving with State-Aware Scheduling. In *Eighth Conference on Machine Learning and Systems*. https://openreview.net/forum?id=ubIvpetAd6

[21] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, and Yizhou Shan. 2024. Inference without Interference: Disaggregate LLM Inference for Mixed Downstream Workloads. arXiv:2401.11181 [cs.DC] https://arxiv.org/abs/2401.11181

[22] Yuwei Huang and Guoliang Li. 2024. Laser: Buffer-Aware Learned Query Scheduling in Master-Standby Databases. *Proc. VLDB Endow.* 18, 3 (Nov. 2024), 743–755. https://doi.org/10.14778/3712221.3712239

[23] Intel Corporation. 2019. *Intel Xeon Gold 6230 CPU.* Intel Corporation. Retrieved July 31, 2024 from https://ark.intel.com/content/www/us/en/ark/products/192437/intel-xeon-gold-6230-processor-27-5m-cache-2-10-ghz.html

[24] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2022. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 289–305. https://doi.org/10.1145/3542929.3563468

[25] Ferdi Kossmann, Ziniu Wu, Alex Turk, Nesime Tatbul, Lei Cao, and Samuel Madden. 2024. CascadeServe: Unlocking Model Cascades for Inference Serving. arXiv:2406.14424 [cs.DC] https://arxiv.org/abs/2406.14424

[26] Tim Kraska, Tianyu Li, Samuel Madden, Markos Markakis, Amadou Ngom, Ziniu Wu, and Geoffrey X. Yu. 2023. Check Out the Big Brain on BRAD: Simplifying Cloud Data Processing with Learned Automated Data Meshes. *Proc. VLDB Endow.* 16, 11 (jul 2023), 3293–3301. https://doi.org/10.14778/3611479.3611526

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[28] Willis Lang, Srinath Shankar, Jignesh M. Patel, and Ajay Kalhan. 2014. Towards Multi-Tenant Performance SLOs. *IEEE Transactions on Knowledge and Data Engineering* 26, 6 (2014), 1447–1463. https://doi.org/10.1109/TKDE.2013.74

[29] Philipp Leitner, Waldemar Hummer, Benjamin Satzger, Christian Inzinger, and Schahram Dustdar. 2012. Cost-efficient and application SLA-aware client side request scheduling in an infrastructure-as-a-service cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, IEEE, Piscataway, NJ, USA, 213–220.

[30] Yuhang Li, Rong Gu, Chengying Huan, Zhibin Wang, Renjie Yao, Chen Tian, and Guihai Chen. 2025. HotPrefix: Hotness-Aware KV Cache Scheduling for Efficient Prefix Sharing in LLM Inference Systems. *Proc. ACM Manag. Data* 3, 4, Article 250 (Sept. 2025), 27 pages. https://doi.org/10.1145/3749168

[31] Ziyang Liu, Hakan Hacıgümüş, Hyun Jin Moon, Yun Chi, and Wang-Pin Hsiung. 2013. PMAX: tenant placement in multitenant databases for profit maximization. In *Proceedings of the 16th International Conference on Extending Database Technology* (Genoa, Italy) *(EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 442–453. https://doi.org/10.1145/2452376.2452428

[32] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: improving resource efficiency at scale. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 450–462. https://doi.org/10.1145/2872887.2749475

[33] Ryan Marcus and Olga Papaemmanouil. 2016. WiSeDB: a learning-based workload management advisor for cloud databases. *Proc. VLDB Endow.* 9, 10 (jun 2016), 780–791. https://doi.org/10.14778/2977797.2977804

[34] Ryan Marcus and Olga Papaemmanouil. 2017. Releasing Cloud Databases for the Chains of Performance Prediction Models.. In *CIDR*.

[35] Markos Markakis. 2026. AutoSLO Code. https://github.com/mmarkakis/autoslo.

[36] Daniel Mendoza, Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. Interference-Aware Scheduling for Inference Serving. In *Proceedings of the 1st Workshop on Machine Learning and Systems* (Online, United Kingdom) *(EuroMLSys '21)*. Association for Computing Machinery, New York, NY, USA, 80–88. https://doi.org/10.1145/3437984.3458837

[37] Vivek Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramuli, and Surajit Chaudhuri. 2013. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *CIDR*.

[38] Vivek Narasayya, Sudipto Das, Manoj Syamala, Surajit Chaudhuri, Feng Li, and Hyunjung Park. 2013. A demonstration of SQLVM: performance isolation in multi-tenant relational database-as-a-service. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1077–1080. https://doi.org/10.1145/2463676.2463686

[39] Vikram Nathan, Vikramank Singh, Zhengchun Liu, Mohammad Rahman, Andreas Kipf, Dominik Horn, Davide Pagano, Gaurav Saxena, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Intelligent Scaling in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 269–279. https://doi.org/10.1145/3626246.3653394

[40] Jennifer Ortiz, Victor Teixeira De Almeida, and Magdalena Balazinska. 2015. Changing the Face of Database Cloud Services with Personalized Service Level Agreements. In *CIDR*.

[41] Jennifer Ortiz, Brendan Lee, and Magdalena Balazinska. 2016. PerfEnforce Demonstration: Data Analytics with Performance Guarantees. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 2141–2144. https://doi.org/10.1145/2882903.2899402

[42] Jennifer Ortiz, Brendan Lee, Magdalena Balazinska, Johannes Gehrke, and Joseph L Hellerstein. 2018. SLAOrchestrator: Reducing the Cost of Performance SLAs for Cloud Data Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Berkeley, CA, USA, 547–560.

[43] Tirthak Patel and Devesh Tiwari. 2020. CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Piscataway, NJ, USA, 193–206. https://doi.org/10.1109/HPCA47549.2020.00025

[44] Meikel Poess, Bryan Smith, Lubor Kollar, and Paul Larson. 2002. TPC-DS, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 582–587. https://doi.org/10.1145/564691.564759

[45] Uwe Rohm, Klemens Bohm, and H-J Schek. 2001. Cache-aware query routing in a cluster of databases. In *Proceedings 17th International Conference on Data Engineering*. IEEE, IEEE, Piscataway, NJ, USA, 641–650.

[46] Francisco Romero and Christina Delimitrou. 2018. Mage: online and interference-aware scheduling for multi-scale heterogeneous systems. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 19, 13 pages. https://doi.org/10.1145/3243176.3243183

[47] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1228–1242. https://doi.org/10.1145/3514221.3526158

[48] Sherif Sakr and Anna Liu. 2012. SLA-based and consumer-centric dynamic provisioning for cloud databases. In *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, IEEE, Piscataway, NJ, USA, 360–367.

[49] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan (Murali) Narayanaswamy. 2023. Auto-WLM: Machine Learning Enhanced Workload Management in Amazon Redshift. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 225–237. https://doi.org/10.1145/3555041.3589677

[50] Snowflake. 2026. Multi-cluster warehouses | Snowflake Documentation. https://docs.snowflake.com/en/user-guide/warehouses-multicluster. Retrieved March 19, 2026.

[51] Snowflake. 2026. Pricing Options. https://www.snowflake.com/en/pricing-options/. Retrieved March 17, 2026.

[52] Snowflake. 2026. Snowflake AI Data Cloud. https://www.snowflake.com/en/. Retrieved June 1, 2026.

[53] Rebecca Taft, Willis Lang, Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, and David DeWitt. 2016. STeP: Scalable Tenant Placement for Managing Database-as-a-Service Deployments. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 388–400. https://doi.org/10.1145/2987550.2987575

[54] A.S. Tanenbaum and D. Wetherall. 2011. *Computer Networks*. Pearson Prentice Hall. https://books.google.com/books?id=T7lGswEACAAJ

[55] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (July 2024), 3694–3706. https://doi.org/10.14778/3681954.3682031

[56] Johannes Wehrstein, Roman Heinrich, Mihail Stoian, Skander Krid, Martin Stemmer, Andreas Kipf, Carsten Binnig, and Muhammad El-Hindi. 2025. Redbench: Workload Synthesis From Cloud Traces. arXiv:2511.13059 [cs.DB] https://arxiv.org/abs/2511.13059

[57] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 280–294. https://doi.org/10.1145/3626246.3653391

[58] Ziniu Wu, Markos Markakis, Chunwei Liu, Peter Baile Chen, Balakrishnan Narayanaswamy, Tim Kraska, and Samuel Madden. 2025. Improving DBMS Scheduling Decisions with Fine-grained Performance Prediction on Concurrent Queries – Extended. arXiv:2501.16256 [cs.DB] https://arxiv.org/abs/2501.16256

[59] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Hyun Jin Moon, Calton Pu, and Hakan Hacigümüş. 2011. Intelligent management of virtualized resources for database systems in cloud environment. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, Piscataway, NJ, USA, 87–98. https://doi.org/10.1109/ICDE.2011.5767928

[60] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Calton Pu, and Hakan HacigümüŞ. 2011. ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 15, 14 pages. https://doi.org/10.1145/2038916.2038931

[61] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX symposium on operating systems design and implementation (OSDI 22)*. USENIX Association, Berkeley, CA, USA, 521–538.

[62] Geoffrey X. Yu, Ziniu Wu, Ferdi Kossmann, Tianyu Li, Markos Markakis, Amadou Ngom, Samuel Madden, and Tim Kraska. 2024. Blueprinting the Cloud: Unifying and Automatically Optimizing Cloud Data Infrastructures with BRAD. *Proc. VLDB Endow.* 17, 11 (Aug 2024), 3629–3643. https://doi.org/10.14778/3681954.3682026

[63] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2102–2114. https://doi.org/10.1145/3448016.3457291

[64] Liang Zhao, Sherif Sakr, and Anna Liu. 2013. A framework for consumer-centric SLA management of cloud-hosted databases. *IEEE Transactions on Services Computing* 8, 4 (2013), 534–549.

[65] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. DistServe: disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation* (Santa Clara, CA, USA) *(OSDI'24)*. USENIX Association, USA, Article 11, 18 pages.