CloudGlide: Deconstructing the Landscape of Cloud-Based Analytics

Michail Georgoulakis Misegiannis
geom@in.tum.de
Technische Universität München

Daniel Ritter
daniel.ritter@sap.com
SAP

Viktor Leis
leis@in.tum.de
Technische Universität München

Jana Giceva
jana.giceva@in.tum.de
Technische Universität München

ABSTRACT

Cloud-based analytics now exposes an increasingly vast space of design choices. Key axes include provisioning (static vs. ephemeral), caching (capacity, tiering), scheduling (admission thresholds, parallelism), and pricing (reserved, on-demand, spot); each choice materially affects cost and performance. To navigate this complexity without deploying large-scale infrastructure, we present **CloudGlide**, a *white-box simulation framework* for systematically exploring cloud data analytics trade-offs. CloudGlide pairs a queueing-theoretic model with a discrete-event simulator (DES), ingesting real-world workload traces to provide cost and latency predictions under diverse configurations. Validated on industry traces and standard benchmarks, CloudGlide approximates behavior across existing architectures and supports rapid what-if analyses along the above axes, all without the prohibitive costs of live deployments.

PVLDB Reference Format:
Michail Georgoulakis Misegiannis, Daniel Ritter, Viktor Leis, and Jana Giceva. CloudGlide: Deconstructing the Landscape of Cloud-Based Analytics. PVLDB, 18(13): 5638 - 5651, 2025.
doi:10.14778/3773731.3773739

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/mikegeo98/cloudglide_olap.

1 INTRODUCTION

Modern cloud-based online analytical processing (OLAP) offers a diverse array of systems, architectures, and pricing models. Previously distinct paradigms, such as Data Warehouse as a Service (DWaaS) [11, 19, 50] and Query as a Service (QaaS) [5, 26] are steadily converging. As shown in Fig. 1, DWaaS platforms increasingly embrace pay-per-use scaling [47], while QaaS providers offer provisioned capacity [28], blurring cost and resource control boundaries and expanding the design space for optimization.

On-demand scaling readily handles spiky, unpredictable workloads yet leaves newly spawned nodes cache-cold, degrading performance and inflating costs if the burst is short [47]. Choosing an ill-suited queueing or scaling policy can further inflate delays by up to 7× for the same workload [8]. Caching further complicates matters: opting for more DRAM over slower tiers can lead to latency and cost swings of up to *20*× [36], though some workloads see only marginal gains [62]. Pricing adds yet another axis: adjusting the ratio of reserved to on-demand virtual machines can shift cost by *30–50%* [33], while spot instances can reduce spend by up to *70%* at the risk of preemptions and scheduling challenges [34].

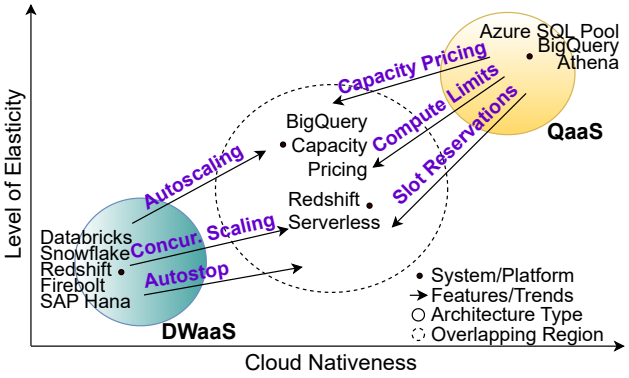


Figure 1: Landscape and trends for cloud-based OLAP

Given these complexities, it is *extremely difficult* to anticipate latency and cost outcomes for a given workload without systematic evaluation or a full-scale trial. This prompts a natural question: ***How can system developers systematically evaluate these design decisions* before *live deployments?***

Existing approaches fall into three categories. *Direct experimentation* tests configurations end-to-end and yields ground-truth results, but can demand substantial engineering effort and hundreds of dollars in cloud fees [64]. In this vein, a TPC-H–based experimental study compares DWaaS and QaaS across data access models, storage formats, and scaling options [66], and the Cloud Analytics Benchmark likewise compares OLAP systems on cloud-relevant metrics such as elasticity and monetary cost [69]. Second, *ML-based autotuning* methods [3, 9, 48, 74] and data-driven frameworks [32, 64] leverage historical telemetry to adapt resource allocations and scaling policies in real-time. However, they typically operate as black boxes—offering limited visibility in what-if scenarios—and require substantial training data to generalize [2], making them less suitable for exploratory, pre-deployment planning.

Proceedings of the VLDB Endowment, Vol. 18, No. 13 ISSN 2150-8097.
doi:10.14778/3773731.3773739

Table 1: Comparison of DWaaS and QaaS Models

	Aspect	DWaaS	QaaS
1	**Resource Provisioning**	User selects cluster size	No user-side provisioning
2	**Resource Isolation**	Isolated at the cluster level	Per-job resources from a shared pool
3	**Data Access and Caching**	On-node DRAM and SSD caching	Direct scan from object storage; no caching
4	**Billing Model**	Fixed rate based on cluster uptime	Pay-per-use (e.g., CPU time, scanned data)

White-box analytical modeling provides a transparent, low-cost alternative to live trials, enabling first-order comparisons before deployment via closed-form equations. While earlier white-box approaches looked at cost-optimal cloud compute placement [39, 72], none have tried to model the broader design space spectrum. Consequently, the design space of cloud-based OLAP remains largely unexplored in academia.

Motivated by these challenges, and aided by recent industry workload traces [52, 68, 71] and benchmarks [21, 65, 69] that expose realistic workload patterns, we propose **CloudGlide**, a *white-box simulation framework* that systematically evaluates diverse system configurations fast and without requiring live system deployments.

CloudGlide is designed for *workload-level* what-if analyses (e.g., for capacity planning, autoscale policy, instance-mix exploration, and offline prototyping), where decisions depend on aggregate behavior across thousands of queries. Traditional cost estimators rely on pre-execution query plans to predict runtime [38]. In contrast, CloudGlide assumes access to each query's post-execution resource footprint—CPU-seconds, scanned bytes, and shuffle (exchange) bytes—rather than the query plan. This information is routinely available in audit logs or workload traces [73]. The resulting plan-agnostic design enables teams to evaluate and compare provisioning, caching, and scaling strategies on their workloads *without touching user SQL*, and to derive actionable insights.

We address three major obstacles in modeling cloud OLAP systems.

(1) **C1:** Providing a system-agnostic execution-engine baseline for fair comparisons across OLAP architectures.
(2) **C2:** Balancing simplicity and accuracy when using closed-form, queueing-theoretic equations to model and reconstruct query execution from post-execution footprints.
(3) **C3:** Selecting representative workloads and leveraging recent industry traces [52, 68, 71] for realistic evaluation.

Our contributions directly address these challenges. We begin with a survey of modern OLAP architectures (§2), followed by CloudGlide's theoretical foundation that combines queueing theory with discrete-event simulation (§3). We validate CloudGlide on TPC-H and TPC-DS across varying data and cluster sizes, achieving a median query-latency QERROR of 1.27. Targeted microbenchmarks—varying repetition and concurrency rates, and workload management strategies—show that the simulator reproduces key multi-query interaction trends (§4). Finally, we illustrate four design-space studies that CloudGlide enables (§5). We also *open source* CloudGlide [24] to foster further research in this evolving domain.

2 BACKGROUND: OLAP IN THE CLOUD

While the design space for cloud OLAP has exploded with a plethora of *architectures* offering diverse capabilities, we begin by presenting an overview of the two **extremes** of the spectrum. We then examine the evolving landscape and how these paradigms converge.

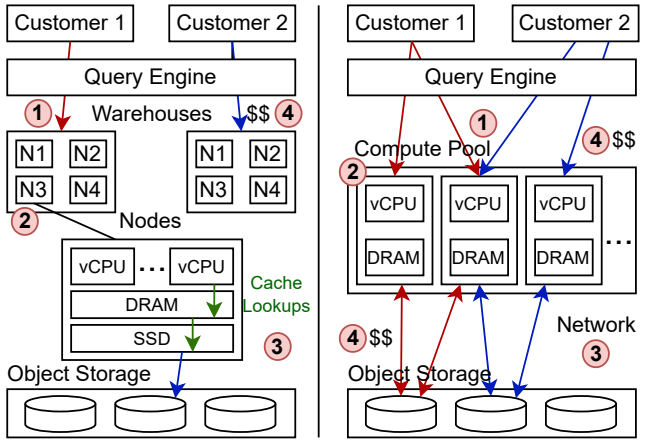
2.1 Endpoints: DWaaS and QaaS

Data-Warehouse-as-a-Service **(DWaaS)** and Query-as-a-Service **(QaaS)** are two fundamental yet contrasting paradigms for cloud-based analytics. DWaaS, exemplified by HANA Cloud [35, 42] or Snowflake [19], relies on pre-provisioned hardware in a disaggregated compute-storage architecture. QaaS, pioneered by BigQuery [26], offers serverless query execution and pay-per-use pricing. These models differ across four dimensions (see Tab. 1, Fig. 2):

1. Resource Provisioning: In DWaaS, users select and pay for a dedicated cluster based on their workload needs, often specifying the cluster size and node type. For example, Redshift offers three instance types and four size options [6]. While this provisioning offers predictable performance, it is prone to over or under-utilization [55], especially when workload demands fluctuate. In contrast, QaaS does not require user-side provisioning, as resources are dynamically allocated from a shared pool on a per-query basis.

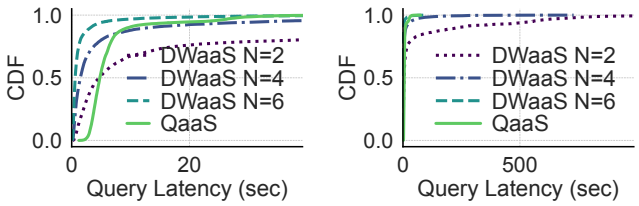
2. Resource Isolation: Both QaaS and DWaaS rely on virtualization or containers for isolation, but differ in granularity. QaaS allocates fine-grained resources from a shared pool—often by spawning multiple containers or micro-VMs on the fly—maximizing resource sharing and cost efficiency, at the expense of more complex scheduling and security measures (e.g., cgroups, chroot). DWaaS enforces isolation more coarsely at the cluster level, hosting multi-tenant workloads on dedicated hardware.

3. Data Access and Caching: Both architectures store primary data in object storage [19, 43]. DWaaS leverages on-node resources (DRAM and SSD) to cache parts of the dataset, thus reducing object storage calls and enabling prefetching. This is especially beneficial for repetitive queries, which account for *80% of Redshift workloads* [47]. Systems like Firebolt [50] even cache data



(a) DWaaS Architecture **(b) QaaS Architecture**

Figure 2: Different architectures for cloud-based systems



(a) Bulk (≤80th percentile) **(b) Full Tail**

Figure 3: CDF of Cloud Analytics Benchmark

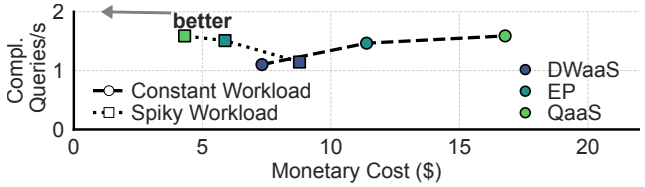


Figure 4: Architecture Tradeoffs for Different Workloads

in DRAM during ingestion. QaaS, on the other hand, scans data directly from object storage over the network, increasing latency compared to on-node caching but still maintaining competitive throughput—particularly since DWaaS SSDs, while cost-effective, are not high-end [70] and cloud network bandwidth has significantly improved [22].

4. Billing Models: DWaaS employs a fixed billing model, charging for cluster uptime regardless of workload intensity. In contrast, QaaS uses a pay-per-use model, with costs determined either by volume of data scanned ($5/TB as of mid-2025 [7, 29, 45]) or by consumed compute resources, measured in units that abstract over CPU time and query duration [28]. Although this approach aligns costs with usage more closely, it makes cost control less predictable.

Performance. To evaluate the differences of DWaaS and QaaS in practice, we ran the Cloud Analytics Benchmark [69] on a 1 TB dataset and replayed 5,000 queries (10 CPU-h) over one hour on Redshift with $N = \{2, 4, 8\}$ `ra3.xlplus` nodes (DWaaS) versus Athena (QaaS). As Fig. 3a shows, a "right-sized" 4-node DWaaS cluster achieves a 3× lower median latency than QaaS, but Athena's elasticity drives 99th-percentile latency down by over 8× (Fig. 3b). These results confirm that DWaaS suits steady, predictable workloads, whereas QaaS excels at handling ad-hoc or large queries that benefit from rapid, on-demand scaling—something that DWaaS could only match through drastic over-provisioning.

2.2 Elastic Pool: Meeting in the Middle

Although DWaaS and QaaS represent two distinct extremes of the cloud OLAP deployment spectrum, the boundaries between them have become increasingly blurred. This convergence has led to a hybrid approach, which we refer to as the *Elastic Pool (EP)* architecture. Systems like Redshift Serverless [12] and BigQuery Capacity Pricing [27] exemplify EP by maintaining a baseline of always-on compute for steady performance, and scaling on demand during bursts—blending DWaaS predictability with QaaS flexibility.

Comparative Evaluation. To evaluate the architectures in practice, we compare DWaaS, QaaS, and EP using two workload patterns: (i) a steady 3 TB workload processed over 1 h, and (ii) an 800 GB workload featuring two large bursts during the same period. We use Redshift with 4 `ra3.xlplus` nodes for DWaaS, Athena for QaaS, and Redshift Serverless (32 processing units) for EP, ensuring comparable overall compute capacity. As shown in Fig. 4, the *steady* workload produces a clear Pareto front. While DWaaS was the most cost-effective choice, its throughput lagged behind EP and QaaS by 13% and 30%, respectively. In the *bursty* scenario, however, the QaaS model emerged as a clear winner for both cost and throughput, delivering results 20% faster and 10% cheaper than EP.

Beyond a Simple Midpoint. EP is not merely a compromise between DWaaS cost-efficiency and QaaS elasticity; it exposes a *policy surface*. Early EP features such as dynamic cluster resizing, on-demand provisioning at peak times, and autostops after inactivity [11, 44] paved the way for greater flexibility. Recent innovations have shifted away from traditional compute cluster abstractions, allowing users to define a base compute capacity while autoscaling manages workload surges, thus preserving control without sacrificing adaptability [47, 58]. Meanwhile, QaaS enhancements, such as alternative pricing models, capacity reservations, and compute limits [28], are delivering more predictable performance and cost.

This progressive convergence on autoscaling from both DWaaS and QaaS (see Fig. 1) opens new questions in resource management:

RQ1: When should an EP system trigger additional resources versus queue incoming jobs?

RQ2: What impact do these scaling decisions have on meeting Service Level Objectives (SLOs)?

RQ3: For autoscaled EP resources, is it more efficient to eagerly populate data caches (DWaaS-style) or rely on ad-hoc network fetches (QaaS-style)?

Prior work analyzed the cost benefits of waiting and scaling policies in compute pools [8, 48]. However, to the best of our knowledge, no study has yet systematically addressed these configuration tradeoffs, leaving the broader design space of autoscaling systems largely unexplored in academia. In §3, we demonstrate how *CloudGlide* systematically captures and explores these dimensions.

2.3 Alternatives

While Function-as-a-Service (FaaS) has been explored as an alternative to QaaS for analytics [17, 46, 51], it has limitations. Unlike QaaS, which supports stateful serverless query processing, FaaS is stateless, constraining inter-function communication. Moreover, its cost model pays off for a narrow set of workloads, due to the high expense of GET/PUT operations on object storage [46]. Even with proposed workarounds such as ephemeral caching [36], alternative isolation approaches for state sharing [62], and tiered storage architectures [16], FaaS has not achieved commercial traction for OLAP workloads and is therefore not considered further in this paper.

Self-hosted solutions (Infrastructure-as-a-Service) [10, 40, 61] deploy analytics software on cloud infrastructure, providing an alternative to fully managed services. They can operate as a DWaaS system or offer QaaS-style, on-demand query processing. Therefore, the same optimization principles extend to these deployments, and CloudGlide is equally well-suited for modeling them.

3 ANALYTICAL MODELING & SIMULATION

Problem Scope. Modern cloud-based OLAP systems present a vast design space, spanning *resource provisioning* (fixed, ephemeral compute), *concurrency handling* (queueing, scaling), *caching strategies*, and *pricing models* (fixed-rate vs pay-per-use). Direct experimentation can be both time- and cost-prohibitive, while learned predictors demand extensive historical data. As a result, developers lack a lightweight tool for rapidly evaluating cost and latency tradeoffs.
Intended Use Case. To address this gap, CloudGlide is built primarily for *offline* capacity planning and what-if exploration. By ingesting full query traces together with per-query resource profiles, it lets teams explore the design space in a query plan-agnostic manner, making it applicable even when query plans are confidential or unavailable. While its focus is offline analysis, CloudGlide could optionally run in a lightweight "look-ahead" mode alongside a production autoscaler to inform real-time scaling decisions.

To support these scenarios effectively, our approach has to: *capture query execution and resource interplay* (in isolation and under concurrency); *remain architecture-agnostic* to compare DWaaS, QaaS, or EP under a single baseline; and *leverage real workload traces* for practical relevance.

3.1 Theoretical Foundations

In **CloudGlide**, we combine a queueing-theory model with a discrete-event simulator. *Queueing theory* helps encapsulate query execution and basic concurrency effects—modeling how queries queue up and share resources—while our *simulator* extends it to handle time-varying workloads (e.g., bursty arrivals, on-demand scaling).

OLAP queries often involve multiple resource-intensive stages: some are dominated by data retrieval bandwidth (*I/O-bound*), others perform non-trivial computations (*CPU-bound*), and in distributed environments, the *shuffle stage*—where data is redistributed across nodes—can also become a significant bottleneck [53]. To capture these dynamics, we treat each stage (I/O, CPU, shuffle) as a limited-capacity resource shared by all active queries. When the number of simultaneous queries exceeds a stage's capacity, additional queries wait in a queue until resources free up. Concretely, we model query execution as a *fork-join*-like structure, an established abstraction for representing parallel sub-tasks that eventually merge [67]. Within this framework, each query consists of three stages: an **I/O stage** that scans data from storage (governed by data size D, bandwidth, and caching), a **CPU stage** that processes the data (limited by CPU load and available compute), and a **shuffle stage** that redistributes data across compute operations (constrained by the network).

We assume that a query's *total work*—its CPU-seconds and data size—remains constant even as parallelism scales with additional nodes or vCPUs. In most cloud OLAP engines, operator selection depends on SQL and table statistics—not on cluster size—so scaling from four to eight nodes, for example, typically only increases each operator's parallelism (e.g., more partitions) rather than altering the *physical operators* in the query plan [1, 13]. While some systems choose different join strategies (e.g., broadcast vs. shuffle) for small tables, these decisions hinge on table size, which does not change with configuration. In our TPC-H evaluation (§4), we observed no physical-operator changes across 2-, 4-, and 8-node clusters; scanned bytes remained identical, and CPU-seconds varied by only 7% (2 → 4) and 12% (2 → 8), both within normal run-to-run variance. By contrast, shuffle volume increases with parallelism [20]. Adaptive re-optimization, where the query plan topology changes in-flight requires plan knowledge and is beyond CloudGlide's scope.

The classical fork–join model defines total service time as the *slowest* of its overlapping stages (here I/O, CPU, shuffle). This can underpredict when stages do not fully coincide. Analysis on Snowset indicates high overlap: for a sample of tenants, a query's dominant phase accounts for 80–85% of total latency on average [71], with the remaining 15–20% often attributable to blocking operators [76] or coordination delays.

To span the spectrum—from no stage overlap to near-full stage overlap—we compare five lightweight service-time estimators and later select the simplest that meets our accuracy/robustness targets (§4). *Mental model:* let $T_{\text{io}}, T_{\text{cpu}}, T_{\text{sh}}$ denote the isolated stage times computed from footprints and stage capacities; let $\delta \geq 0$ be a per-query fixed overhead (parsing/coordination delay), $p \geq 1$, and $k \in \mathbb{N}$. We report $\widehat{T}_\bullet = T_\bullet + \delta$ for each estimator:

$$T_{\max} = \max\left(T_{\text{io}},\ T_{\text{cpu}},\ T_{\text{sh}}\right) \tag{1}$$

$$T_{\text{cpu-only}} = T_{\text{cpu}} \tag{2}$$

$$T_{\text{sum}} = T_{\text{io}} + T_{\text{cpu}} + T_{\text{sh}} \tag{3}$$

$$T_{\text{pm}} = \left(T_{\text{io}}^{p} + T_{\text{cpu}}^{p} + T_{\text{sh}}^{p}\right)^{1/p} \tag{4}$$

$$T_{\text{mw}} = \sum_{i=1}^{k} \max\left(T_{\text{io},i},\ T_{\text{cpu},i},\ T_{\text{sh},i}\right) \tag{5}$$

Intuition. **Max** assumes strong stage overlap (optimistic without δ); **CPU-only** is useful when compute dominates and as a sanity check; **Sum** assumes no overlap (pessimistic bound); **Power-mean** interpolates between sum (p=1) and max ($p\to\infty$), tuning the assumed overlap; **Multi-wave** models multiple shuffle–compute "waves" by chaining k fork–join phases.

We compare all variants in §4.1 and then adopt the simplest model that meets our accuracy and robustness targets. Below, we detail how each of the three stages is modeled.

3.1.1 I/O Stage. In the I/O stage, data may be retrieved from DRAM, SSD, or object storage (e.g., S3 [4]). Because multiple queries share storage and network channels, we adopt a *fair-share* model: if M_{io} queries are actively scanning, each receives $1/M_{\text{io}}$ of the available bandwidth on that tier. This is a simple, optimistic baseline. In practice, richer bandwidth allocators are often used (e.g., Dominant Resource Fairness (DRF) [25], Weighted Fair Queueing (WFQ) [49]) to improve tail latency and isolation. CloudGlide allows swapping in any such policy in place of fair sharing.

Let D (in GiB) be the amount of data each query scans, and $P_{\text{DRAM}}, P_{\text{SSD}}$, and $(1 - P_{\text{DRAM}} - P_{\text{SSD}})$ be the probabilities that a data block is retrieved from DRAM, SSD, or remote storage, respectively. Suppose there are N nodes in the cluster, each with read bandwidths DRAM-BW, SSD-BW, and Net-BW for DRAM, SSD, and the network. Consequently, the expected service time of the I/O stage for a single query is:

$$T_{\text{io}} = M_{\text{io}} \Bigg[P_{\text{DRAM}} \frac{D}{N \times \text{DRAM-BW}} + P_{\text{SSD}} \frac{D}{N \times \text{SSD-BW}} + \left(1 - P_{\text{DRAM}} - P_{\text{SSD}}\right) \frac{D}{N \times \text{Net-BW}} \Bigg] . \tag{6}$$

Table 2: Comparison of Cloud OLAP Architectures in CloudGlide

Parameter	DWaaS	Elastic Pool (EP)	QaaS
Pricing	$N \times T \times R_{\text{node}}$ (Eq. (12))	$\sum_{i=1}^{T_s} c_i \cdot \frac{R_{\text{unit}}}{3600}$ (Eq. (13))	$D \times P_{\text{scan}}$ (Eq. (14))
Caching	Fixed: P_{DRAM}, P_{SSD}	Warm-up: $P_{\text{DRAM}}(n) = P^*(1 - e^{-\gamma n})$ (Eq. (7))	None: $P_{\text{DRAM}} = P_{\text{SSD}} = 0$
Isolation	Shared (M_{io}, $M_{\text{sh}} \geq 1$)	Shared (M_{io}, $M_{\text{sh}} \geq 1$)	Dedicated; $M_{\text{io}} = M_{\text{sh}} = 1$
Resources	Fixed (c)	Dynamic $c(t)$	Per-query c_q
I/O Time	T_{io} (Eq. (6))	T_{io} with cache warm-up (Eq. (7))	$T_{\text{io}}^{\text{QaaS}} = \frac{D}{\text{Net-BW}}$ (Eq. (8))
CPU Time	T_{cpu} (Eq. (9)) + T_{spill} (Eq. (10))	Eq. (9) with time-varying $c(t)$ + Eq. (10)	Eq. (9) with per-query c_q
Shuffle Time	T_{sh} (Eq. (11))	T_{sh} (Eq. (11))	T_{sh} (Eq. (11))

In practice, workloads often exhibit data skew: some nodes handle disproportionately large scans [15]. To account for this, users can lower the effective per-tier bandwidths in the configuration (e.g., reduce Net-BW by 10–20% to reflect observed hotspots).

In an EP system, new nodes start with cold caches. As ephemeral nodes often live only briefly, they may not receive enough repeated queries to amortize cache warm-up [62]. When a node processes many queries, its caches warm toward a steady-state hit rate. We model this with a simple exponential: let n be the number of queries a node has served, and P^* the steady-state DRAM hit probability. Then

$$P_{\text{DRAM}}(n) \;=\; P^*\big(1 - e^{-\gamma n}\big) \quad . \tag{7}$$

Here $\gamma > 0$ controls the warm-up speed (we observe $\gamma \in [0.05, 0.2]$ in our experiments). The exponential captures diminishing returns: once a cache is warm, additional queries yield minimal benefit [62].

In contrast, QaaS environments typically launch *ephemeral containers* that do not retain cache state—thus, we set $P_{\text{DRAM}} = P_{\text{SSD}} = 0$ for those queries, reducing T_{io} to simply:

$$T_{\text{io}}^{\text{QaaS}} \;=\; \frac{D}{\text{Net-BW}} \quad . \tag{8}$$

This underscores why short-lived nodes and ephemeral services may never benefit substantially from caching, unlike longer-lived resources in DWaaS or EP that process many queries per node.

3.1.2 CPU Stage. In the CPU stage, the processing work is divided into sequential and parallelizable components [30]. Following Amdahl's principle [57], let T_0 be the runtime on a single core and P the parallelizable fraction. The sequential part $(1 - P)T_0$ is unaffected by parallelism, while the parallel part corresponds to $W := P\,T_0$ *core-seconds* of compute. We discretize W into unit core-second tasks and assume the query receives c_{eff} effective cores under time slicing (its CPU share, optionally capped per query). Up to c_{eff} tasks are active per second. Hence,

$$T_{\text{cpu}} \;=\; (1 - P)\,T_0 \;+\; \frac{\lceil P\,T_0 \rceil}{c_{\text{eff}}} \quad . \tag{9}$$

Example. If T_0=100 and P=0.9, the sequential share is 10 s. With c_{eff}=10, the parallel part is $\lceil 0.9 \cdot 100 \rceil / 10 = 9$ s, for a total of 19 s.

During a query's execution, the effective core share c_{eff} may change as other queries arrive/finish or the system scales; as explained later in §3.2, the simulator updates c_{eff} at each event and drains the remaining parallel work at the new rate.

Beyond CPU time, CloudGlide also tracks each query's materialized working set $\big(f_{\text{mem}}D\big)$ in memory, where D (GiB) is the query's data volume and $f_{\text{mem}} \in [0, 1]$ is the fraction kept resident. Let Mem_{node} denote per-node memory capacity. If the aggregate resident set of concurrent queries on a node exceeds Mem_{node}, the overflow Δ_{spilled} (GiB) is spilled to SSD. We model the resulting extra I/O time as

$$T_{\text{spill}} \;=\; \frac{\Delta_{\text{spilled}}}{\text{SSD-BW}} \quad , \tag{10}$$

We add this penalty to the CPU stage time $\big(T_{\text{cpu}} \leftarrow T_{\text{cpu}} + T_{\text{spill}}\big)$. CloudGlide does not fully model out-of-memory operators; any nonzero Δ_{spilled} incurs this disk-spill penalty [37].

3.1.3 Shuffle Stage. Many OLAP queries redistribute (shuffle) intermediate data across nodes (e.g., joins, global aggregations). Let D_{sh} (GiB) be the data exchanged among N nodes, each with per-node network bandwidth Net-BW. Ideally, aggregate bandwidth scales to N×Net-BW. With M_{sh} queries shuffling concurrently, each query receives $1/M_{\text{sh}}$ of this total, yielding:

$$T_{\text{sh}} \;=\; \frac{M_{\text{sh}}\,D_{\text{sh}}}{N \times \text{Net-BW}} \quad . \tag{11}$$

In practice, sub-linear speedups can arise from partitioning or synchronization overheads. Some engines (e.g., BigQuery [43]) use a dedicated shuffle service to scale bandwidth separately from compute, yet high concurrency can still saturate this layer and make it a bottleneck [43].

3.1.4 Architectural Differences. Building on the fork–join-based queueing network from §3.1 (**C1**), we extend the framework to capture how DWaaS, EP, and QaaS differ along the four dimensions in §2 (**C2**); Tab. 2 summarizes the parameters.

1. Resource Provisioning. DWaaS uses a fixed cluster of N nodes (total c cores). EP autoscales, exposing a time-varying capacity $c(t)$. QaaS provisions per-query capacity c_q on demand.

2. Resource Isolation. In DWaaS/EP, in-flight queries share I/O (M_{io} concurrent scans) and network (M_{sh} shuffles). CPU capacity is time-sliced among active queries (see Eq. (9)). In QaaS, each query is granted dedicated slots (M_{io}=M_{sh}=1). In practice, however, noisy-neighbor effects may still appear [41, 56].

3. Data Access and Caching. DWaaS maintains steady DRAM/SSD hit rates ($P_{\text{DRAM}}, P_{\text{SSD}}$), EP warms caches over time (Eq. (7)), and QaaS always reads from object storage ($P_{\text{DRAM}} = P_{\text{SSD}} = 0$). All follow the same I/O latency model (Eq. (6)).

4. Billing Model: From a customer's perspective, each architecture's cost reduces to a simple closed-form expression:

$$\text{Price}_{\text{DWaaS}} = N \times T \times R_{\text{node}}, \tag{12}$$

$$\text{Price}_{\text{EP}} = \Big(\sum_{i=1}^{T} c_i\Big) \times \frac{R_{\text{unit}}}{3600} \tag{13}$$

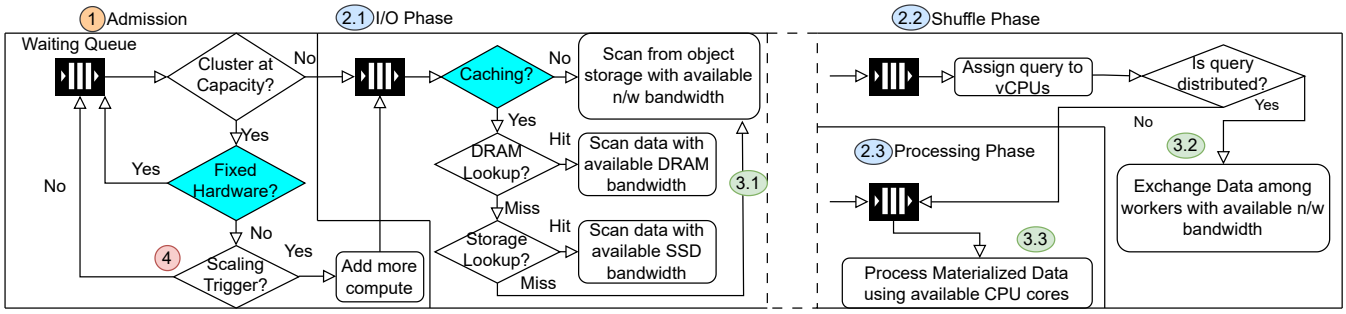


Figure 5: Flowchart of query lifetime in a CloudGlide simulation (DES)

Table 3: Key Configurable Parameters in CloudGlide (*essential inputs in bold*)

Category	Parameters (Symbols)
Query-Level	**Data** (D); **CPU time** (T_0); **Arrival time**; **Shuffled data** (D_{sh}); Materialized fraction(f_{mem}); Overflow (Δ_{spilled})
Workload-Level	Parallel fraction (P); Cache warm-up rate (γ); Steady-state hit rates (P^*_{DRAM}, P^*_{SSD}).
Hardware	**Number of nodes** (N); **CPU cores per node** (c_{node}); Total cores ($C{=}N\,c_{\text{node}}$); **Memory per node** (Mem_{node}); **DRAM bandwidth** (DRAM-BW); **SSD bandwidth** (SSD-BW); **Network bandwidth** (Net-BW); **Node type**.
Scaling	Scaling policy (threshold-based, etc.); scale-up/down trigger; cold-start delay; instance type (on-demand/spot).
Scheduling	Queue discipline (e.g., FCFS, SJF); concurrency limit.
Cost & Pricing	DWaaS \$/node-hour rate (R_{node}); EP unit rate (R_{unit}); QaaS \$/TB scan rate (P_{scan}) ; reserved/spot discounts
Model Variants	Offset constant (δ); Power-mean exponent (p); Multi-wave count (k)

$$\text{Price}_{\text{QaaS}} = D \times P_{\text{scan}} \tag{14}$$

Here, N is the number of nodes, T the total uptime (h), R the per-node rate, c_i the compute units at second i, D the data scanned (TB), and P_{scan} the per-TB scan rate.

Real-world OLAP billing breaks down into granular line items, such as compute (on-demand vs. spot), storage, I/O, and data transfer, rather than an all-in-total bill. CloudGlide handles either detailed component pricing or a single effective rate.

Why a simulator? Closed-form queueing models capture only steady-state trends and do not handle dynamic scaling or more realistic bursty arrivals [68]. By embedding these aspects in a discrete-event framework, CloudGlide replays realistic workloads while adapting resources over time—addressing both **C2** and **C3**.

3.2 Discrete-Event Simulation (DES)

CloudGlide's DES proceeds as follows (see Fig. 5):

1. Query Arrivals: Queries enter the system according to an input trace; each arrival enqueues an *arrival* event.

2. Query Forking: On arrival, a query is either admitted (if resources permit) or queued. Admitted queries spawn three *stage-completion* events (I/O, CPU, shuffle), with service times from the closed-form models (Eqs. (6), (9) and (11)).

3. Service-Time Updates: In DWaaS/EP, whenever a stage admission or completion event fires, we "lazily" recompute the remaining service times of queries sharing that resource via our closed-form model (Tab. 2). On an admission, each active stage's resource share shrinks; on a completion, the freed capacity immediately reduces the estimated remaining times of the survivors. In QaaS, per-query resources are dedicated, so no recalculation is needed.

4. Scaling Policy: In EP mode, crossing the queue-length threshold enqueues a *scale-up* or *scale-down* event. These adjust available I/O, CPU, and shuffle slots (and apply cache warm-up for new nodes via Eq. (7)), after a cold-start delay. DWaaS and QaaS skip this step.

5. Logging: On completion, we log per-query latency and cost (Eqs. (12) to (14)) and aggregate mean and tail metrics for analysis.

Implementation & Simulation Cost. CloudGlide is written in 1,000 lines of Python. Its "lazy-recalculation" strategy—updating remaining service times only upon an event and for affected queries—ensures that runtime scales linearly with the number of events processed. On a single core of an AMD Ryzen 7 PRO 6850U (32 GiB RAM), it processes over 10,000 queries/s, while peak memory usage stays below 50 MiB for 10 K queries, 300 MiB for 50 K, and 3 GiB for 3 M. Processing a 50 K-query Snowset trace takes 5 s, and the largest Snowset workload (3 M queries) completes in under 6 min.

3.3 Essential Parameters and Calibration Effort

Essential Parameters and Calibration Effort: Targeted at OLAP system developers, CloudGlide requires only each query's observed resource footprints (CPU-seconds, scanned and shuffled data) plus cluster specs and pricing; essential inputs are in bold in Tab. 3.

Per-query metrics are accessible in cloud OLAP systems via audit logs or system tables (e.g., BigQuery's *INFORMATION_SCHEMA*, Redshift's *STL*) and in public workload traces [68, 71]. Hardware data are likewise public for DWaaS/EP; when DRAM-BW, SSD-BW, or Net-BW are not exposed (e.g., on Vantage.sh [70]), our repository offers zero-touch SQL microbenchmarks to infer them [24].

Other Parameters: When workload-level inputs are missing, we use empirically grounded defaults from the Snowset trace and our

experiments: cache hit rates are set to $P^*_{\text{DRAM}} = 0.7$, $P^*_{\text{SSD}} = 0.2$, aggregated across Snowset [71]. Parallel fraction is set to $P = 0.95$ based on average CPU-bound behavior in our experiments, and warm-up rate is set to $\gamma = 0.02$ (≈ 50 queries to reach steady state). When richer telemetry is available, users can override the defaults. We include a sensitivity analysis of these defaults (§4.2).

For scheduling and scaling, we mimic industry practice. Our default autoscaler adds nodes when the queue length exceeds 50% of capacity—akin to Concurrency Scaling [11, 19]. We pair this with a hybrid FCFS+SJF scheduling policy (inspired by AutoWLM [59]).

Extensibility and System Comparison. CloudGlide can be extended to a new target system in three steps: (1) ingest per-query resource footprints, (2) specify or infer hardware and bandwidth parameters, and (3) optionally plug in custom scheduling, scaling, or cost logic to yield a fair-share baseline for within-engine what-if analyses. It requires no deep system expertise, and only core workload and hardware information are mandatory; workload-level parameters can be refined via targeted microbenchmarks to boost fidelity. While CloudGlide does not model engine-specific operator implementations, one can compare systems of the same family (e.g., two DWaaS platforms) directly by feeding each system's own per-query traces—letting variations in the query resource footprints drive the comparison. Moreover, because for most queries the choice of physical operators is preserved across cluster sizes, CloudGlide generalizes from a single calibration to arbitrary configurations without plan visibility. The DES then adjusts parallelism, bandwidth, and cache state—capturing both Amdahl's Law and the longer shuffle phases—to adjust the simulation under each scenario. CloudGlide is inherently future-proof: as the cloud landscape evolves—with new architectures and hardware tiers—it can adapt simply by tuning a few parameters or adding minimal glue code.

Aspects Not Modeled. Firstly, CloudGlide models read-only queries. Additionally, CloudGlide abstracts each query into just three resource stages—I/O, CPU, and shuffle—and does not capture operator-level internals or dynamic plan rewrites. Similarly, our fair-share network model treats throughput as a pooled resource and omits fine-grained effects (e.g., per-flow congestion control), which can affect latency under high load. Furthermore, we assume gradual cache warm-up and near-uniform data access; heavy data skew or rapid cache thrashing may incur larger slowdowns than a single "skew factor" can express. A more detailed, plan-aware, or stochastic simulator could address these limitations in future work.

4 EXPERIMENTAL EVALUATION

We now evaluate *CloudGlide*'s single- and workload-level accuracy. We focus on four main questions:

(1) *Model Fidelity (§4.1)*—Which execution-model variant best approximates query processing across different classes of queries?
(2) *Concurrency Effects (§4.2)*—How well does CloudGlide reflect queueing and resource contention under concurrent workloads?
(3) *Scheduling & Scaling (§4.2)*—Do CloudGlide's autoscaling and scheduling policies reproduce trends seen in real cloud systems?
(4) *Robustness (§4.2)*—How sensitive are CloudGlide's predictions to imperfect estimates of workload parameters?

Benchmarks & Workloads. We evaluate CloudGlide on two axes:

(1) *Single-query benchmarks*, to stress different query shapes:

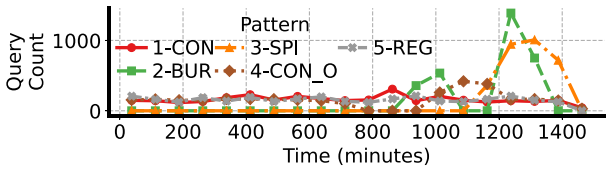


Figure 6: Snowset Patterns 1–5 Query Arrival Timeline

- **TPC-H** (all 22 queries) at SF$\in \{50, 85, 125, 200, 350\}$.
- **TPC-DS** (all 99 queries) at SF100.
- **Resource-Balance queries [63]**, CPU- vs I/O-focused mixes drawn from TPC-DS and JOB.

(2) *Multi-query workloads*, capturing temporal dynamics:

- **CAB** [69], a 5 000+ query mix of TPC-queries.
- **Snowset Trace Patterns** with varied temporal profiles (Fig. 6) [68, 71]. Figure 6 shows their arrival trends, which we will refer to by the codes *1-CON* (constant), *2-BUR* (short bursts), *3-SPI* (one large spike), *4-CON_O* (constant with outlier activity), and *5-REG* (regular repeated jobs).
- **Redbench** query streams with varied repetition rates [65].

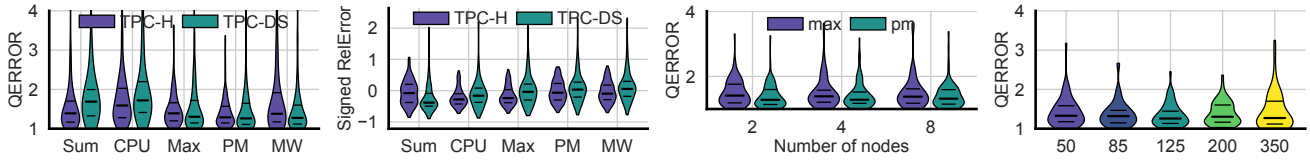
Experimental Setup and Target Systems. We evaluate CloudGlide on representative cloud services: *DWaaS* (Redshift, Snowflake) and *QaaS* (BigQuery, Athena). For DWaaS–like baselines, we parameterize two hardware profiles using provider pricing: *c5d.2xlarge* (8 vCPUs, 16 GiB RAM, \$2.50/hr) to approximate Snowflake, and *ra3.xlplus* (4 vCPUs, 32 GiB RAM, \$1.086/hr) for Redshift pricing, based on documentation [6, 70]. Following measurement-based guidelines [39], we assume 80% of peak network bandwidth; $\approx$ 280 MB/s for *c5d.2xlarge* (scaled from 100 Gbit/s link of bigger *c5d* instances [70]) and 156 MB/s for *ra3.xlplus* (from 10 Gbit/s equivalent bigger instance). DRAM throughput is sustained at 20 GB/s per node, in line with dual-channel DDR4-2400 measurements [31]. For EP-style offerings (e.g., Redshift Serverless), we model each Virtual Processing Unit as 1 vCPU + 8 GiB RAM with $\approx$ 39 MB/s of network throughput (one-quarter that of *ra3.xlplus*) [12].

In QaaS, queries run on short-lived slots without persistent caches; we use scan-based pricing of \$5 per TB (mid-2025 on-demand rates) [5, 28]. Unless stated otherwise, CloudGlide uses $P = 0.95$ (parallel fraction) and $\gamma = 0.02$ (cache warm-up) across all experiments, which were their empirically measured defaults. This uniform configuration ensures that each synthetic replay uses the same settings, enabling direct comparison with real-system results.

4.1 Query Processing Model Evaluation

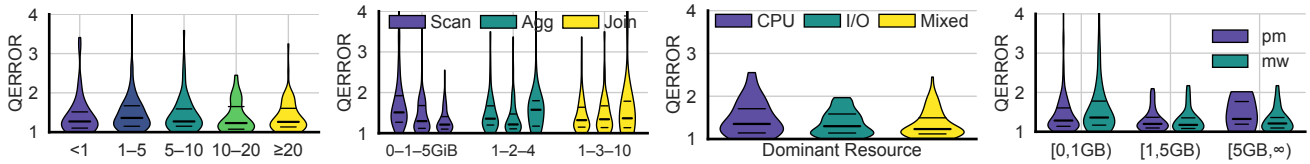
To evaluate CloudGlide's execution-time models and identify when higher-fidelity modeling is required, we compare our five variants *sum*, *cpu*, *max*, *pm* ($p = 1.5$), and *mw* ($k = 3$) (Eqs. (1) to (5)) against measured runtimes on TPC-H (22 queries $\times$ 5 scale factors $\times$ 3 cluster sizes) and TPC-DS (3 cluster sizes $\times$ 99 queries at SF 100).

In Fig. 7a, we show each estimator's QERROR distribution as a violin plot. *sum* and *cpu* exhibit heavy upper tails, with worst-case QERRORs above 5. *max* reduces these extremes (median 1.35 on TPC-H, 1.29 on TPC-DS). The power-mean estimator (*pm*) strikes the best balance: its median QERROR is 1.28 (TPC-H) and 1.26



(a) QERROR by model (b) SRE by model (c) Scale-out in DWaaS (d) TPC-H Scale factors (*SF*)

Figure 7: Model-fidelity validation across TPC-H and TPC-DS, varying cluster sizes and scale factors.



(a) Latency buckets (s) (b) Amount of data or operators (c) Dominant phase (d) Shuffle volume

Figure 8: Model-fidelity validation across latency buckets, query complexity, dominant execution phase, and shuffle volume.

(TPC-DS), and its 90th percentile remains < 2 on both benchmarks. Finally, *mw* performs similarly on TPC-DS—albeit with a few larger outliers—but underperforms on TPC-H, where both *pm* and *max* achieve lower QERROR.

Signed relative error (Fig. 7b) reveals a mild systematic bias toward under-prediction, with outliers on both sides of zero. TPC-H's errors are more tightly clustered (IQR ≈ 0.15 in signed error), while TPC-DS shows greater spread, reflecting more diverse query shapes. As QERROR directly captures multiplicative deviation, we retain it as our principal accuracy metric for the rest of the study.

We next verify the stability of the most accurate estimators, *pm* and *max*, across cluster sizes. Figure 7c plots their QERROR on Redshift hardware as we vary the number of nodes $n \in \{2, 4, 8\}$. The median QERROR remains ≈ 1.3 for all n, with an absolute change $\Delta\text{median} < 0.1$ across these scales, indicating that our node-level parameters extrapolate reliably with n.

Figure 7d presents model accuracy across TPC-H scale factors $s \in [50, 350]$. Smaller scale factors exhibit higher median QERRORs (≈ 1.44 at SF = 50), while larger factors improve toward ≈ 1.24. However, at SF = 350, the QERROR distribution develops a heavier upper tail, suggesting more complex execution behavior or unmodeled overheads at the largest scale factor.

Next, we analyze model fidelity by **runtime** (Fig. 8a). Short queries (< 1 s) are dominated by fixed overheads and are poorly suited to overlap-centric models. Here, *max* performs best: the minimum offset prevents underestimation on tiny jobs without inflating QERROR, yielding the lowest median and tail among all estimators for sub-second queries. As runtimes increase to the 1–5 s range, *max*'s fixed offset becomes overly conservative (median QERROR ≈ 1.5), while *pm*, which we use for queries with runtime ≥ 1 s, recovers to ≈ 1.3. For long-running queries (> 10 s), *pm* converges, reaching a median QERROR of ≈ 1.25, confirming that its assumptions hold best once fixed overheads are amortized.

We then examine **query complexity** (Fig. 8b) as a function of scanned data or operator counts. For **Scan**, bins correspond to total input size (0–1 GiB, 1–5 GiB); for **Agg** and **Join**, they indicate the number of aggregations or joins (1, 2–3, and ≥ 4 aggregations; 1–2, 3–9, and ≥ 10 joins). Larger scans are modeled more accurately (median QERROR ≈ 1.2) as estimates scale linearly with input size. Aggregation-heavy queries stress the simple pipeline model and median QERROR rises to 1.67 for ≥ 4 aggregations. Join-rich queries fall in between (median QERROR ≈ 1.43), as multi-table joins are still pipelined; here, *mw* occasionally outperforms *pm*, reflecting its ability to model multiple "waves" of overlapping phases.

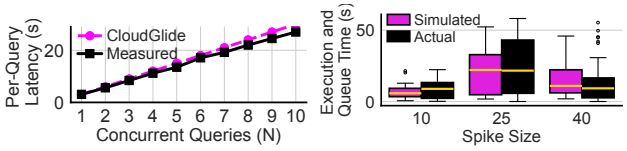
To isolate pure **shuffle-volume effects**, Fig. 8d plots QERROR against the shuffled data. For highly shuffle-intensive queries, *mw* reduces the 75th percentile error by ≈ 0.4 relative to *pm*, albeit at the cost of underperforming on simpler, low-shuffle jobs. This confirms that higher-fidelity (multi-wave) models yield measurable benefits when explicit phase repetition is present, but may otherwise overfit.

Finally, Fig. 8c groups queries by their **dominant resource** (CPU, I/O, or mixed) using the Resource-Balance set (queries from JOB and TPC-DS). CPU-bound queries exhibit the widest spread—their 75th percentile QERROR reaches ≈ 1.87. I/O-bound queries are easier to predict (median ≈ 1.34), while *mixed* queries lie in between, with lower medians but higher outliers. The coexistence of long I/O and CPU phases allows phase overlap to absorb small misestimations on either side, narrowing the overall distribution.

Overall, *pm* maintains the lowest median and tail QERROR across runtime, size, and complexity, and we adopt it as the default execution-time model in the remaining experiments.

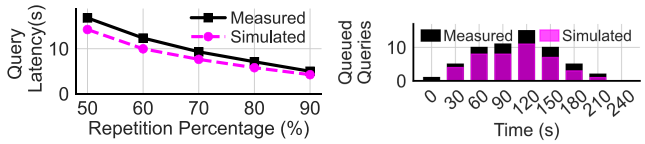
Outliers and Failure Modes While overall accuracy remains high, a small set of queries deviate noticeably. A manual query plan inspection of outlier queries (14, 24, 43, 58, 63, 71, 74 from TPC-DS) reveals two recurring outlier patterns: **(i) Deeply nested CTEs and multi-stage materialization**, which trigger repeated "flush-and-rescan" cycles violate our single-pass assumption; and **(ii) Complex joins and aggregations**—queries with ≥ 5 such operators—where even *mw* fails to capture execution behavior accurately.

Additional experiments also identified complex user-defined functions (**UDFs**) as another outlier class, where per-tuple interpretation costs inflate CPU time beyond the scope of the current phase



(a) TPC-H Q1: Execution times for varying concurrency. **(b) Measured and simulated times across spike intensities.**

Figure 9: Execution time under concurrency.



(a) Query time vs. repetition rate **(b) Queued queries over time**

Figure 11: Repetition and queueing

model. Incorporating explicit materialization penalties, per-tuple UDF overheads, and heuristics for detecting deep CTE chains are promising directions to further narrow these accuracy gaps.

4.2 Concurrent Benchmarks

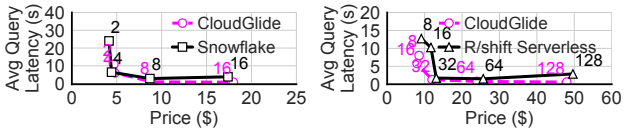
After validating CloudGlide's accuracy on isolated queries, we test whether its per-query models, combined with queueing and resource-sharing logic, remain valid under concurrent execution.

Concurrency Testing. We evaluate contention modeling by launching identical TPC-H Q1 (SF 15) instances in parallel and recording wall-clock response times (Fig. 9a). As concurrency increases, measured latencies rise by up to 8×, yet *CloudGlide* tracks the trend closely: the median prediction error remains within 10% across all load levels, validating its resource-sharing logic.

Workload Spikes. Performance unpredictability often stems from transient concurrency *spikes*. We evaluate this using a synthetic benchmark that injects bursts of 10, 25, or 40 queries within a 20 s window. As shown in Fig. 9b, although spikes are inherently hard to model, *CloudGlide* closely predicts median latencies (within 20%). Accuracy degrades in the tails, as the model does not capture straggler behavior or fairness heuristics under high contention.

Scheduling - Caching. We next evaluate DRAM hit rates and scheduling dynamics using two workloads. Figure 11a shows Redbench results, where queries with increasing repetition in their plans execute progressively faster due to higher cache reuse. *CloudGlide* captures this trend through its modeled hit rates, with a median underestimation of ≈ 16%. Figure 11b tracks queued queries at 30 s intervals during a 5 min spike (*2-BUR*); while *CloudGlide* reproduces the overall queue trajectory, it underpredicts peak backlog by up to 20%. Overall, these results show that *CloudGlide* effectively captures the key caching and scheduling trends.

Scale-Out in DWaaS and EP. We run CAB on Snowflake (2–16 nodes) to assess scale-out behavior (Fig. 10a). Latency predictions converge within 20% at 8 nodes; smaller clusters (2–4) show up to 30% underestimation due to transient bursts and CPU saturation beyond our idealized resource-sharing approach. On Redshift Serverless (Fig. 10b), *CloudGlide* maintains ≤ 25% runtime error and ≤ 10% cost error while scaling capacity from 8–128 vCPUs. Smaller elastic tiers (8–16 vCPUs) remain hardest to predict, as ephemeral rapid resource transitions amplify contention.



(a) DWaaS Scale-Out **(b) Elastic Pool Scale-Out**

Figure 10: *CAB* runs in DWaaS and EP baselines. Markers show the number of DWaaS nodes and VPUs in the EP.

Finally, we evaluate autoscaling on Redshift Serverless, triggering scale-out when the queue length exceeds 50% of CPU cores. As shown in Fig. 12, *CloudGlide* reproduces the observed scale-up intervals within 1–2 min of actual triggers, and predicts costs within 10%. While the simplified time-slicing model can underestimate latencies under contention, *CloudGlide* effectively captures scaling trends across workloads.

Sensitivity Analysis. Figure 13a shows how three representative queries (Q2 I/O, Q5 CPU, and Q9 Shuffle) respond when each stage's parameters vary between −30% to +30%. Compared to Redshift baselines, Q5 (CPU-bound) exhibits the largest deviation, indicating higher sensitivity to CPU parameters, while Q2 (I/O-bound) and Q9 (shuffle-heavy) are less affected.

The ablation in Fig. 13b quantifies how individual workload-level parameters affect mean *QERROR* on a 4-node ra3.xlplus Redshift cluster. Starting from a baseline of 1.31, eliminating the spilled-data penalty raises QERROR by 5%, while reducing the CPU parallel fraction adds 16%. Disabling SSD caching further increases error by 27%, and removing DRAM caching drives a dramatic 78% jump. Finally,

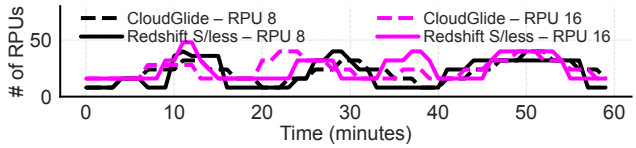
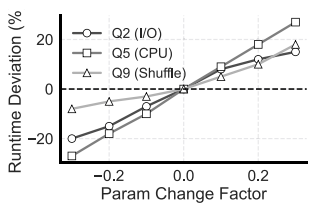


Figure 12: Scaling Behavior Evaluation – Redshift Serverless

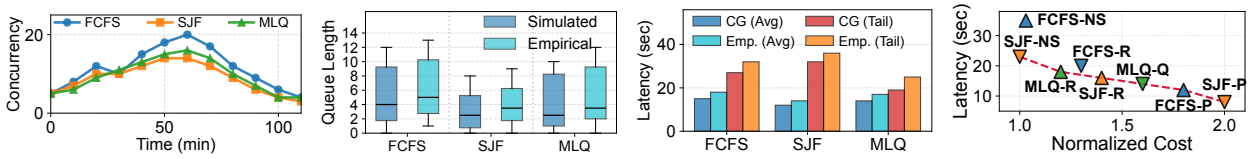


(a) Per-query sensitivity.

Step	Q	(%)
Baseline	1.31	—
Δspill (5.37 to 0 GiB)	1.38	5
P (0.95 → 0.90)	1.52	16
P^*_{SSD} (0.20 → 0.00)	1.66	27
P^*_{DRAM} (0.70 → 0.00)	2.33	78
sum (pm → sum)	3.12	138

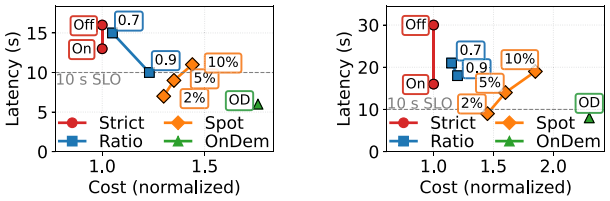
(b) Workload-level ablation.

Figure 13: (a) CloudGlide sensitivity to estimate noise; (b) Ablation of workload parameters and their effect on QERROR.



(a) Concurrency over time. **(b) Queue length distribution.** **(c) Average and tail latencies.** **(d) Cost–latency Pareto front.**

Figure 14: Workload management insights across scheduling and scaling policies.



(a) Workload 1 – Moderate Bursts **(b) Workload 2 – Larger Bursts**

Figure 15: Comparison of concurrency policies. *Ratio* denotes fraction percentage, and *Spot* interruption frequency.

replacing the power-mean with the sum estimator more than doubles QERROR (by 138%). Ablation results underscore that caching and stage-overlap modeling are the primary accuracy levers.

5 DESIGN DECISIONS EXPLORED WITH CLOUDGLIDE

We now analyze four key design decisions that developers face when building or tuning cloud OLAP systems and show how **CloudGlide** informs them: **RQ1** (*scaling vs. queueing*), **RQ2** (*SLO-driven resource management*), **RQ3** (*caching in EP*), and a comparative evaluation of DWaaS, QaaS, and EP under varied workloads.

5.1 Use Case 1: Scaling vs. Queueing

Scenario & Developer Challenge. We evaluate how *scaling* compares with *queueing* under a 15 s mean-latency SLO during load spikes. A developer considers:

- **Queue Only** (`NS`): run a fixed-size cluster and queue excess queries—lower cost, longer waits.
- **Scale-Out** via triggers: `Q` (queue-based), add nodes when queue length exceeds q^*; `R` (reactive), provision when utilization $u > u^*$; `P` (predictive), allocate resources *before* an anticipated burst.

CloudGlide's Role. We replay a mixed workload combining `1-CON` and `2-BUR` (Sect. 4), i.e., steady arrivals plus burst phases. Supported schedulers are *FCFS*, *SJF*, and *MLQ* (Multi-Level Queue) with four priority levels. *CloudGlide* simulates *scheduling–scaling* pairs (e.g., `SJF-R`) under a non-scaling baseline (`NS`) or with one of the scaling triggers (`Q`, `R`, `P`). For each configuration, it logs concurrency, queue length, and mean/tail latencies, and identifies settings that meet the 15 s SLO at minimum cost. For scale-out, developers choose the condition (e.g., queue length $> q^*$, CPU utilization $u > u^*$) and the step size (add +1 node per action vs. larger steps).

Setup: 2-node Redshift cluster; queue threshold $q^* = 8$ (cluster vCPU count); reactive trigger uses CPU utilization with $u^* = 0.70$; predictive trigger `P` fires when the 10 s moving-average of arrivals has a positive slope and the projected queue length exceeds q^*.
Insights. Figures 14a to 14c show different scheduling policies in non-scaling (`-NS`) setups. As expected, `SJF` reduces mean latency by preventing short-query starvation, albeit at the cost of higher tail latencies.

Under a *scalable* system, more factors arise: **Fig. 14d** demonstrates how queue-based, reactive (CPU-driven), and predictive triggers trade off performance and cost. Queue-based triggers are simple but may react too late once the cluster saturates; reactive triggers respond faster but depend on the monitored resource (CPU vs. I/O); predictive triggers can pre-empt bursts but risk overshoot if demand stabilizes. In practice, `-Q` handles mild bursts, while `-R` or `-P` keep latencies below the 15 s SLO at an added cost of +30–100%. The Pareto frontier shows up to 2× cost differences and 2.5× latency variation across scheduling–scaling pairs.

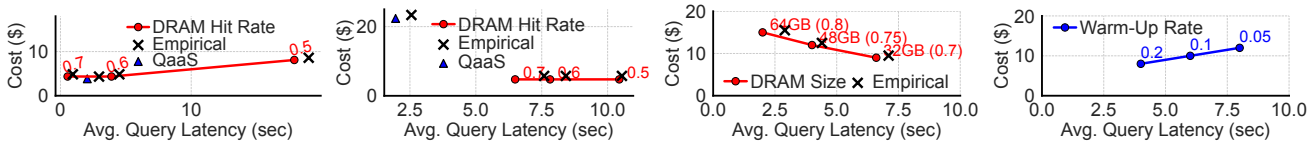
CloudGlide reproduces both the relative ordering of scheduling policies and their absolute cost–latency magnitudes (Figs. 14a to 14d). For a representative scaling configuration `SJF-R`, its P50 latency error is $\leq 15\%$ and its cost MAPE is $\leq 8\%$.

5.2 Use Case 2: SLO-Driven Resource Allocation

Scenario & Developer Challenge. While serverless implies a scale-to-zero architecture, real-world QaaS systems run on shared pools of finite resources (e.g., a baseline of 400 slots). They must serve both **short-running** queries (e.g., with a target 10 s SLO) and more lenient **long-running** queries. Overprovisioning is costly; underprovisioning risks SLO violations for short-running jobs. To mitigate sudden bursts, developers may use *spot instances* at a discount price, but if they are reclaimed, the query is interrupted and restarts at the baseline pool.
CloudGlide's Role. We simulate three **policies** for short queries:

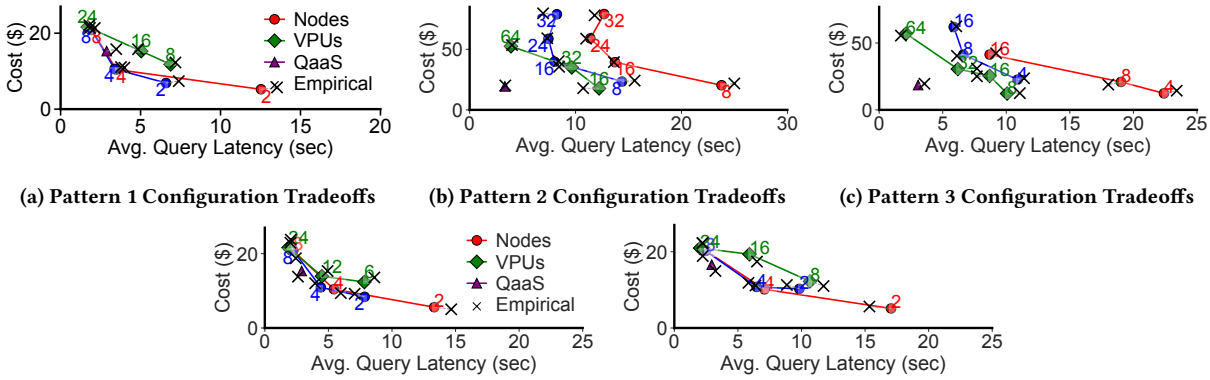
- **Strict Priority (A)**: preemptive scheduling on the 400 baseline slots, where short jobs *always* preempt long jobs to protect the 10 s SLO at peak times.
- **Fixed Ratio (B)**: reserve a static fraction r of baseline slots for short jobs (e.g., $r = 0.5 \Rightarrow 200/400$ slots).
- **Slot Scaling (C)**: Dynamically provision additional *spot slots* at lower cost and fall back to **on-demand** slots when spot capacity is reclaimed; reclaimed queries restart on the baseline pool.

Workloads: two mixes of short/long queries with different spike profiles: moderate bursts (Patterns `1-CON+2-BUR`) and larger, longer spikes (Patterns `1-CON+3-SPI`).



(a) Steady workload: DWaaS vs. QaaS cost–latency tradeoff. **(b) I/O-intensive workload: DWaaS vs. QaaS pricing and latency.** **(c) Effect of DRAM size and hit rate (P^*_{DRAM}) on cost and latency.** **(d) Impact of cache warm-up rate (γ) on average latency.**

Figure 16: Caching vs. No Caching: Latency–cost tradeoffs across DWaaS, QaaS, and EP configurations.



(a) Pattern 1 Configuration Tradeoffs **(b) Pattern 2 Configuration Tradeoffs** **(c) Pattern 3 Configuration Tradeoffs**

(d) Pattern 4 Configuration Tradeoffs **(e) Pattern 5 Configuration Tradeoffs**

Figure 17: Performance-cost tradeoffs for different configurations: DWaaS, Autoscaling DWaaS, Elastic Pool, and QaaS.

Insights. Under moderate load (Fig. 15a), **Strict Priority (A)** struggles to keep short queries under the 10 s SLO. Despite the preemption, multi-tenancy leads to excessive stalling on shared resources. **Fixed Ratio (B)** meets the 10 s SLO when r is sufficiently large by dedicating a fraction of baseline slots to short jobs. **Slot Scaling (C)** lowers mean latency by elastically adding capacity, but frequent reclaims inflate tails due to restarts. An *on-demand* option could yield the best short-query latency but at a significantly higher cost. Across policies, the same workload exhibits up to 2× cost and 3× latency variation.

Under heavier bursts (Fig. 15b), **Fixed Ratio (B)** becomes less effective as the spike saturates the reserved fraction. **Strict Priority (A)** fares better short-term (preemptions effectively free all slots for newly arriving queries) but penalizes long jobs. **Slot Scaling (C)** or on-demand capacity can maintain the SLO, yet require substantially more resources, with frequent reclaims also worsening tail latencies. Cost and performance differences grow to 4× and 2.5×, respectively, suggesting that when such spikes are common, *raising the baseline capacity* can be cheaper and steadier than repeatedly relying on spot/on-demand bursts.

On BigQuery, our simulated short-query latencies under Policy B matched measured latencies within 15%, and cost deviations were under 5% across all spot/interruption scenarios.

5.3 Use Case 3: Caching Considerations

Scenario. A developer observes rising I/O times in a DWaaS cluster with 32 GiB DRAM as repetitive queries push mean latency beyond a 10 s SLO. The developer must decide whether to *(i)* expand DRAM capacity, *(ii)* switch to an autoscaling Elastic Pool (EP) system and tune cache warm-up behavior, or *(iii)* migrate to QaaS (no caching) to restore sub-10 s performance.

Workloads. We evaluate caching behavior using arrival times from `1-CON` and `2-BUR` to capture steady and bursty access patterns. Each workload replays 100 queries with varying repeatability to balance cold and hot queries, using datasets of 50 GB (steady) and 200 GB (I/O-intensive). Experiments run on a 2-node Redshift cluster ($N = 2$) with `ra3.xlplus` nodes.

CloudGlide's Role. *CloudGlide* models three caching options: scaling DRAM from 32 → 64 GiB (raising hit rate 0.70 → 0.80); varying the hypothetical EP warm-up rate γ; and using a QaaS setup with no caching. It estimates latency and cost to quantify how cache size and warm-up dynamics shape overall performance–cost trade-offs.

Insights. As expected, QaaS cannot match DWaaS performance *under low concurrency* (Fig. 16a). However, under sustained usage and as Fig. 16b shows, the pay-per-use model incurs a clear *cost premium* relative to the fixed DWaaS configuration. Increasing DRAM from 32 → 64 GiB and boosting the hit rate P^*_{DRAM} from 0.7 → 0.8 reduces average latency L_{avg} by nearly 2× (Fig. 16c). Similarly, improving the EP cache warm-up rate from $\gamma = 0.05$ to $\gamma = 0.2$ significantly lowers average latency during bursts (Fig. 16d).

Overall, for repetitive workloads, scaling DRAM or enabling autoscaling with a tuned cache warm-up mechanism can sharply reduce I/O times. *CloudGlide* quantifies these gains before hardware or service-level changes are deployed in production.

On Redshift, our model's latency predictions for increased DRAM configurations deviated by less than 15%, cost projections stayed

within 6% of the actual increase from larger node types, and repetitive query latencies aligned within 27% of measured values.

5.4 Use Case 4: Architecture Comparison

Scenario. A BI team must choose between four architectures—**DWaaS**, **Autoscaling DWaaS**, **QaaS**, and **EP**—to balance cost and latency for their workloads. Autoscaling DWaaS expands or shrinks DWaaS cluster size in response to load, EP allocates compute from a shared pool across tenants, offering finer-grained elasticity.
CloudGlide's Role. *CloudGlide* simulates all four architectures under identical arrival traces, logging per-query latencies L_{avg}, L_{95} and cost to map their performance–cost Pareto fronts (Fig. 17a–Fig. 17e).
Workloads. We evaluate the workloads from Sect. 4, each processing a 10 CPU h load and scanning 10 TB of data over two hours.
Insights. Constant Loads (`1-CON, 5-REG`). A fixed DWaaS cluster is most cost-efficient, typically **1.7–2**× cheaper than autoscaling DWaaS or QaaS at comparable latency. For example, the fixed cluster costs \$10 at $L_{\text{avg}} = 5.5$ s, whereas autoscaling ($N_{\max} = 8$) costs \$18 for $L_{\text{avg}} = 4.5$ s—only ≈ 18% faster but 80% more expensive.
Spiky Loads (`2-BUR, 3-SPI`). Serverless and EP configurations yield **1.5–6**× lower latency than fixed DWaaS clusters for comparable or moderately higher cost by elastically scaling capacity during bursts. QaaS, in particular, delivers the largest latency gains under short, spike-heavy workloads, as idle fixed capacity in DWaaS inflates the effective cost per query.
Mixed Workloads (`4-CON_0`). Here's where it gets more interesting: autoscaling DWaaS cuts *tail* latencies by up to **50**% relative to a minimal fixed cluster, while increasing overall cost by only **20**%–**30**%. Those occasional heavy queries make short-term scale-outs more efficient than maintaining an oversized cluster.
Overall Pareto Frontier. Each architecture was Pareto-optimal under at least one workload pattern and configuration—no single design dominated across all conditions. Systematically exploring these workload–capacity trade-offs in **CloudGlide** reveals *which* patterns each architecture handles best and *by how much*. Across all patterns, CloudGlide's cost–latency Pareto fronts closely match empirical results on Redshift, Redshift Serverless, and BigQuery, with mean latency error below **21**% and mean cost error below **7**%.

6 RELATED WORK

While database research has long focused on operator-level cost models and on-premises performance tuning, cloud analytics bring new challenges—elastic compute, pay-per-use pricing, and multi-tenancy—that call for cost-aware, pre-deployment tools.

CloudGlide brings together three research strands—*operator-level models*, *whole–job predictors*, and *cost- or queue-centric cloud analytical models*—into a framework that captures both query execution and workload/resource management. To our knowledge, CloudGlide is the first end-to-end simulator for cloud OLAP.
Operator–level models From System R's per-operator formulas [60] and Volcano/Cascades' rule-based optimizer [30], through cache/TLB-aware scans and joins [18] and RDMA-optimized hash joins [14], decades of work have produced cost models to predict the runtime of a *query plan* under isolated, fixed-hardware assumptions. In contrast, CloudGlide is query plan-agnostic: it collapses operator detail into three stages (I/O, CPU, shuffle) using per-query resource footprints, then layers on concurrency and elastic scaling. By calibrating once with these footprints—rather than predicting them—CloudGlide builds a stable baseline and can simulate complex, dynamic what-if scenarios even when the plan is hidden.
Whole-job predictors. Big-data tools (e.g., PerfOrator [54], SparkTune [13]) advanced modeling to distributed setups by regressing runtimes on disk, CPU, and network "bricks" for a *known DAG*, but remained engine-specific and assumed fixed clusters. CloudGlide adopts their stage-level resource accounting while dispensing with a static DAG: it feeds per-query footprints into an M/M/$c(t)$ queueing network whose server count can scale every second.
Cost- and queue-centric cloud analytical models. Prior white-box models for the cloud condense entire workloads into a handful of metrics—scanned data, processing-seconds—and then optimize VM selection [33] or tuning rules in an M/M/m queue [8]. They rely on closed-form estimates for steady-state throughput and cost. CloudGlide embraces their cost-optimization ethos but adds time-varying arrivals and per-query resource footprints. This **temporal dimension** enables quantifying performance and cost trade-offs across diverse, workload-specific scenarios, as demonstrated in §5.
Black-box Models. Black-box approaches use historical logs to train ML or sampling models that tune indexes, memory, or VM types without revealing system internals [2, 3, 9, 23, 32]. Recent extensions apply cost-aware auto-tuning and dynamic query placement in production [64, 75] and route queries across engines via learned profiles (e.g., BRAD [74]). These methods excel at online adaptation but demand extensive training data and offer limited pre-deployment what-if visibility. CloudGlide fills this gap with a transparent, parameterized simulator for offline trade-off analysis.
Key differences. Unlike operator models (plan-based), big-data predictors (fixed DAGs), cloud models (no temporal information), and black-box ML solutions (data-heavy, opaque, online-focused), CloudGlide's simulator (i) requires no query plan, (ii) works across engines and architectures, (iii) captures elastic scaling, pricing, and bursty arrivals, and (iv) supports fast, offline what-if analyses.

7 CONCLUSION & FUTURE WORK

We presented **CloudGlide**, a white-box simulation framework for modeling performance–cost trade-offs in cloud OLAP architectures. In the cloud, performance and cost are inseparable; with principled assumptions and light calibration, CloudGlide provides clear insights into the impact of parameters like caching, scheduling, and scaling. To our knowledge, it is the first framework to unify multiple OLAP architectures under an intuitive model. Validation on real systems shows that CloudGlide reproduces production behavior *without* costly live testing and supports *what-if* analyses of current and hypothetical configurations. Overall, our results show that, with the right structure, a transparent white-box model can go surprisingly far while remaining interpretable and fast to iterate.

ACKNOWLEDGMENTS

Funded/Co-funded by the European Union (ERC, FDS, 101164-556). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] Josep Aguilar-Saborit and Raghu Ramakrishnan. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *Proc. VLDB Endow.* 13, 12 (2020), 3204–3216.

[2] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD Conference*. ACM, 1009–1024.

[3] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Billian, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.* 14, 7 (2021), 1241–1253. https://doi.org/10.14778/3450980.3450992

[4] Amazon. 2006. AWS S3. https://aws.amazon.com/s3/. Accessed: 2025-06-30.

[5] Amazon. 2008. Amazon Athena. https://aws.amazon.com/athena/. Accessed: 2025-06-30.

[6] Amazon. 2024. Amazon Redshift Pricing. https://aws.amazon.com/redshift/pricing/. Accessed: 2025-06-20. Pricing page as of 2024.

[7] Amazon Web Services. 2025. Amazon Athena Pricing. https://aws.amazon.com/athena/pricing/. Accessed: 2025-06-30.

[8] Pradeep Ambati, Noman Bashir, David E. Irwin, and Prashant J. Shenoy. 2020. Waiting game: optimally provisioning fixed resources for cloud-enabled schedulers. In *SC*. IEEE/ACM, 67.

[9] Christoph Anneser, Nesime Tatbul, David E. Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *Proc. VLDB Endow.* 16, 12 (2023), 3515–3527. https://doi.org/10.14778/3611540.3611544

[10] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD Conference*. ACM, 1383–1394.

[11] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD Conference*. ACM, 2205–2217.

[12] AWS. 2024. Compute Capacity in Redshift Serverless. https://docs.aws.amazon.com/redshift/latest/mgmt/serverless-capacity.html. Accessed: 2024-06-17.

[13] Lorenzo Baldacci and Matteo Golfarelli. 2019. A Cost Model for SPARK SQL. *IEEE Trans. Knowl. Data Eng.* 31, 5 (2019), 819–832.

[14] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing using RDMA. In *SIGMOD Conference*. ACM, 1463–1475.

[15] Paul Beame, Paraschos Koutris, and Dan Suciu. 2013. Communication steps for parallel query processing. In *PODS*. ACM, 273–284.

[16] Thomas Bodner. 2020. Elastic Query Processing on Function as a Service Platforms. In *PhD@VLDB (CEUR Workshop Proceedings)*, Vol. 2652. CEUR-WS.org.

[17] Thomas Bodner, Theo Radig, David Justen, Daniel Ritter, and Tilmann Rabl. 2025. An Empirical Evaluation of Serverless Cloud Infrastructure for Large-Scale Data Processing. In *EDBT*. OpenProceedings.org, 935–948.

[18] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*. www.cidrdb.org, 225–237.

[19] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD Conference*. ACM, 215–226.

[20] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*. 137–150.

[21] George Fraser. 2023. Cloud Data Warehouse Benchmark: Blog: Fivetran. https://www.fivetran.com/blog/warehouse-benchmark Accessed: 2024-06-17.

[22] Fredrik Korsbäck and Lincoln Dale and Dave McGaugh. 2024. Growing AWS Internet Peering with 400 GbE. https://aws.amazon.com/blogs/networking-and-content-delivery/growing-aws-internet-peering-with-400-gbe/.

[23] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. 2009. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *ICDE*. IEEE Computer Society, 592–603.

[24] Michail Georgoulakis. 2025. CloudGlide: Deconstructing the Landscape of Cloud-Based Analytics. https://github.com/mikegeo98/cloudglide_olap. Accessed: 2025-06-11.

[25] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *NSDI*. USENIX Association.

[26] Google. 2010. Google BigQuery. https://cloud.google.com/bigquery. Accessed: 2025-06-30.

[27] Google Cloud. 2024. BigQuery Pricing. https://cloud.google.com/bigquery/pricing. Accessed: 2025-06-30.

[28] Google Cloud. 2024. Understand BigQuery Slots. https://cloud.google.com/bigquery/docs/slots. Accessed: 2024-06-17.

[29] Google Cloud. 2025. BigQuery On-Demand Pricing. https://cloud.google.com/bigquery/pricing. Accessed: 2025-06-30.

[30] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170.

[31] John L. Hennessy and David A. Patterson. 2019. *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann.

[32] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*. www.cidrdb.org, 261–272.

[33] Yu-Ju Hong, Jiachen Xue, and Mithuna Thottethodi. 2011. Dynamic server provisioning to minimize cost in an IaaS cloud. In *SIGMETRICS*. ACM, 147–148.

[34] Dalia Kaulakiene, Christian Thomsen, Torben Bach Pedersen, Ugur Çetintemel, and Tim Kraska. 2015. SpotADAPT: Spot-Aware (re-)Deployment of Analytical Processing Tasks on Amazon EC2. In *DOLAP*. ACM, 59–68.

[35] Kihong Kim, Hyunwook Kim, Jinsu Lee, Taehyung Lee, Alexander Böhm, Norman May, Guido Moerkotte, Daniel Ritter, Ralf Dentzer, Heiko Gerwens, Irena Kofman, and Mihnea Andrei. 2025. Enterprise Application-Database Co-Innovation for Hybrid Transactional/Analytical Processing: A Virtual Data Model and Its Query Optimization Needs. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 485–498. https://doi.org/10.1145/3722212.3724436

[36] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *OSDI*. USENIX Association, 427–444.

[37] Maximilian Kuschewski, Jana Giceva, Thomas Neumann, and Viktor Leis. 2024. High-Performance Query Processing with NVMe Arrays: Spilling without Killing Performance. *Proc. ACM Manag. Data* 2, 6 (2024), 238:1–238:27.

[38] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[39] Viktor Leis and Maximilian Kuschewski. 2021. Towards cost-optimal query processing in the cloud. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1606–1612. https://doi.org/10.14778/3461535.3461549

[40] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *SIGMOD Conference*. ACM, 2530–2542.

[41] Amiya Kumar Maji, Subrata Mitra, Bowen Zhou, Saurabh Bagchi, and Akshat Verma. 2014. Mitigating interference in cloud services by middleware reconfiguration. In *Middleware*. ACM, 277–288.

[42] Norman May, Alexander Böhm, Daniel Ritter, Frank Renkes, Mihnea Andrei, and Wolfgang Lehner. 2025. SAP HANA Cloud: Data Management for Modern Enterprise Applications. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 580–592. https://doi.org/10.1145/3722212.3724452

[43] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472.

[44] Microsoft. 2023. Serverless SQL Pool – Azure Synapse Analytics. https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/on-demand-workspace-overview. Accessed: 2025-06-30.

[45] Microsoft Azure. 2025. Azure Synapse Serverless SQL Pools Pricing. https://azure.microsoft.com/pricing/details/synapse-analytics/serverless/. Accessed: 2025-06-30.

[46] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD Conference*. ACM, 115–130.

[47] Vikram Nathan, Vikramank Y. 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 *SIGMOD Conference Companion*. ACM, 269–279.

[48] Jennifer Ortiz, Brendan Lee, and Magdalena Balazinska. 2016. PerfEnforce Demonstration: Data Analytics with Performance Guarantees. In *SIGMOD Conference*. ACM, 2141–2144.

[49] Chandandeep Singh Pabla. 2009. Completely fair scheduler. *Linux J.* 2009, 184, Article 4 (Aug. 2009), 4 pages.

[50] Mosha Pasumansky and Benjamin Wagner. 2022. Assembling a Query Engine From Spare Parts. In *CDMS@VLDB*.

[51] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD Conference*. ACM, 131–141.

[52] Alibaba Cluster Trace Program. 2018. Cluster Trace v2018. https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md. Accessed: 2024-06-17.

[53] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[54] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. 2016. PerfOrator: eloquent performance models for Resource Optimization. In *SoCC*. ACM, 415–427.

[55] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SoCC*. ACM, 7.

[56] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *CCS*. ACM, 199–212.

[57] David P. Rodgers. 1985. Improvements in Multiprocessor System Design. In *ISCA*. IEEE Computer Society, 225–231.

[58] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In *EuroSys*. ACM, 16:1–16:16.

[59] 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 *SIGMOD Conference Companion*. ACM, 225–237.

[60] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD Conference*. ACM, 23–34.

[61] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *ICDE*. IEEE, 1802–1813.

[62] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *USENIX ATC*. USENIX Association, 419–433.

[63] Tapan Srivastava. 2023. Resource-Balance Workloads: CPU- vs. I/O-Bound Query Mixes. https://github.com/tapansriv/resource-balance-workloads. Accessed: 2025-06-11.

[64] Tapan Srivastava and Raul Castro Fernandez. 2024. Saving Money for Analytical Workloads in the Cloud. *Proc. VLDB Endow.* 17, 11 (2024), 3524–3537.

[65] UTN Data Systems. 2023. Redbench: A Benchmark Suite for Cloud OLAP Query Patterns. https://github.com/utndatasystems/redbench. Accessed: 2025-06-11.

[66] Junjay Tan, Thanaa M. Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David J. DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing A Cloud DBMS: Architectures and Tradeoffs. *Proc. VLDB Endow.* 12, 12 (2019), 2170–2182.

[67] Alexander Thomasian. 2014. Analysis of Fork/Join and Related Queueing Systems. *ACM Comput. Surv.* 47, 2 (2014), 17:1–17:71.

[68] 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 (2024), 3694–3706.

[69] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proc. VLDB Endow.* 16, 6 (2023), 1413–1425.

[70] Vantage. 2025. AWS Instance Comparison. https://www.instances.vantage.sh/.

[71] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI*. USENIX Association, 449–462.

[72] Ziheng Wang, Emanuel Adamiak, and Alex Aiken. 2024. A Model for Query Execution Over Heterogeneous Instances. In *CIDR*. www.cidrdb.org.

[73] Dong Young Yoon, Barzan Mozafari, and Douglas P. Brown. 2015. DBSeer: Pain-free Database Administration through Workload Intelligence. *Proc. VLDB Endow.* 8, 12 (2015), 2036–2039.

[74] Geoffrey X. Yu, Ziniu Wu, Ferdinand Kossmann, Tianyu Li, Markos Markakis, Amadou Latyr Ngom, Samuel Madden, and Tim Kraska. 2024. Blueprinting the Cloud: Unifying and Automatically Optimizing Cloud Data Infrastructures with BRAD. *Proc. VLDB Endow.* 17, 11 (2024), 3629–3643.

[75] Huanchen Zhang, Yihao Liu, and Jiaqi Yan. 2024. Cost-Intelligent Data Analytics in the Cloud. In *CIDR*. www.cidrdb.org.

[76] Xiaoke Zhu, Min Xie, Ting Deng, and Qi Zhang. 2024. HyperBlocker: Accelerating Rule-based Blocking in Entity Resolution using GPUs. *Proc. VLDB Endow.* 18, 2 (2024), 308–321.