ESP-JETA
ESP Journal of Engineering & Technology Advancements
ISSN: 2583-2646/ Volume 6 Issue 4 October-December 2026
Paper Id: JETA-V6I4P105/ Page No: 46-56

Original Article

AI-Native Data Architectures for Agentic Applications: A Comparative Evaluation of Lakehouse, Warehouse, and OLAP Technologies

Mohit Mittal
Senior Software Engineer, Intercontinental Exchange (ICE)

Received Date: 12 October 2026 Revised Date: 16 October 2026 Accepted Date: 20 October 2026

Abstract: *Large language model (LLM) agents are emerging as a new class of client for analytical data systems. Unlike a human analyst or a dashboard, an agent explores a database by issuing bursts of heterogeneous, partially redundant queries, requests individual entities during a reasoning loop, and expects the data it reads to reflect writes that occurred seconds earlier. Recent database research has termed this behavior agentic speculation and argued that data systems should be redesigned around it. Practitioners, meanwhile, must choose today among three mature architecture families: the cloud data warehouse, the open- table- format lakehouse, and the real- time OLAP engine. This manuscript compares the three for agentic workloads. We first derive seven data- layer requirements for agentic applications from the literature on agent behavior, cloud analytics workloads, and text- to- SQL benchmarks. We then define a reproducible agentic query workload with four parts: schema and sample probes, a speculative analytic burst with 57% exact- repeat queries, entity point lookups, and a micro- batch freshness test. We run it on single- node, open- source proxies for each family: DuckDB native storage (warehouse- style), Delta Lake tables scanned by DuckDB (lakehouse), and ClickHouse MergeTree through chDB (real- time OLAP), all on 5 million order events. The warehouse- style proxy had the lowest median latency on the speculative burst (7. 7.2 ms against 21. 3 ms and 24. 3 ms). The OLAP proxy was 15 times faster than the lakehouse proxy on entity lookups (2. 2.8 ms against 42. 1 ms). The lakehouse proxy made micro- batch writes visible roughly seven to ten times more slowly than the other two (50. 3 ms median, against 5. 5.2 ms and 7. 7.3 ms), and it accumulated 100 new data files after 100 appends. Per- query snapshot resolution accounted for about 17% of the lakehouse' s burst latency. No single architecture met every requirement. We therefore propose an AI- native access layer that sits above any of the three and supplies semantic metadata, freshness watermarks, result caching, and cost guardrails. We also provide a requirement- to- architecture selection matrix. The results are small- scale and proxy- based. We report them as directional evidence with a released methodology, not as vendor rankings.*

Keywords: *Agentic AI, Data Architecture, Lakehouse, Data Warehouse, Real-Time OLAP, Delta Lake, Apache Iceberg, Clickhouse, Duckdb, Text-To-SQL, Model Context Protocol, Benchmarking.*

I. INTRODUCTION

For twenty years the analytical data architecture has been geared towards two types of user: those who write SQL or use business-intelligence tools, and scheduled pipelines that supply reports and feed machine-learning training processes. Both of these user types are predictable. An analyst prepares a number of carefully constructed queries and then the dashboard updates at regular intervals.

Each of the three architecture families that are currently in widespread use has developed in order to meet the needs of these users. The cloud data warehouse separates storage from elastic computing by means of a proprietary, fully managed columnar store [1], [2], [3]. The lakehouse provides ACID transactions and warehouse-level performance for open file formats stored in object storage [4], [5], [6]. Real-time OLAP engines give up some generality in order to offer sub-second, high-concurrency service for data that has just been ingested [7], [8], [9].

LLM agents are a different type of user. An agent that is based on the reason-and-act pattern [10] and that is linked to data via a standard tool interface such as the Model Context Protocol (MCP) [11] is quite unlike a person using a SQL editor. Liu et al. investigated agents carrying out data tasks and introduced the concept of agentic speculation, which refers to agents

This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/2.0/)

generating a large number of exploratory queries characterised by scale, heterogeneity, redundancy, and steerability, and using these queries to examine schemas, test hypotheses, and eventually arrive at an answer [12]. They claim that data systems should adopt an "agent-first" approach. The benchmarks support this view. On Spider 2.0, which requires agents to carry out real enterprise text-to-SQL tasks on production-scale schemas in both cloud warehouses and local databases, the best agent reported in the original study managed to solve only 21.3% of the tasks, compared with 91.2% on Spider 1.0 and 73.0% on BIRD [13], [14]. What distinguishes the various benchmarks is the amount of exploration and multi-query reasoning that a given task demands, and it is this exploration that represents the load which the data layer has to take on.

Those people who are developing agentic applications thus have to make a definite decision, since the current evidence does not address this issue. Studies that compare cloud analytical systems assess latency and cost under TPC-type or trace-based workloads which are designed for human users and scheduled consumers [15], [16]. The research on lakehouse table formats looks at load times, performance on TPC-DS queries, and update behaviour [17], [18]. Analyses of fleets indicate that actual warehouse workloads are repetitive, involve a lot of writing, and are dominated by short queries, which is already a deviation from the assumptions of TPC [19]. None of these studies employs a workload that resembles that of an agent, and none of them evaluates the various architecture families in relation to the requirements that an agent brings—for example, tolerance to bursts, entity-level access, read-your-writes freshness, open access for a number of engines, and metadata that an LLM can reason over.

The manuscript presents four contributions. The first of these is to derive seven data-layer requirements for agentic applications and to base each of them on published evidence (as set out in Section 3). The second contribution is the definition of a small, reproducible agentic query workload which includes speculation, redundancy, entity access, and streaming freshness (in Section 4). The third involves carrying out a controlled comparison of open-source proxies for the warehouse, lakehouse, and real-time OLAP families, using the same data and the same query streams and having the results verified across different engines (in Section 5). The fourth is to convert the findings into a requirement-to-architecture selection matrix and to design an AI-native access layer that is positioned above the storage architecture that an organisation currently uses (in Section 6). Section 7 looks at possible threats to validity, and Section 8 provides the conclusion.

II. BACKGROUND AND RELATED WORK

A. Cloud Data Warehouses

The modern cloud data warehouse is characterised by the separation of durable storage from elastic, independently scalable computing resources. Snowflake's approach involves storing the data in immutable, compressed columnar micro-partitions on cloud object storage and keeping the metadata in a transactional service while scaling up the virtual warehouses as required [1]. A subsequent analysis of Snowflake's elastic query engine when used with disaggregated storage outlines the caching and scheduling mechanisms that are necessary in order to achieve a fast performance as a result of that separation [2]. Google's Dremel was the first to introduce interactive, columnar, massively parallel SQL for use with nested data and forms the basis of BigQuery [3]. The advantages of such a warehouse include a single, governed SQL interface, ACID semantics which are completely managed by the vendor, well-developed workload management, and result caching. Its disadvantages are proprietary storage, a form of storage that other systems generally only obtain via exports or connectors, and, in the past, ingestion methods based on batch or micro-batch loading. Google's Napa makes the underlying trade-off explicit by allowing clients to select their own balance between query performance, data freshness, and cost [20]. This three-way trade-off is once again present in every architecture that is examined here.

B. The Lakehouse

The architecture of the lakehouse adds a transactional metadata layer onto open columnar files, usually Parquet, that are stored in object storage. This allows both warehouse and machine-learning workloads to use a single copy of the data [4]. Delta Lake achieves ACID compliance by using a write-ahead transaction log which is stored together with the data [5]. Photon and other vectorized engines have narrowed the performance gap considerably in comparison with proprietary warehouses [6]. Apache Iceberg and Apache Hudi address the same issue using different metadata and update mechanisms. Jain et al. carried out a comparison of the three formats at a scale of 3 TB TPC-DS. Overall, Delta was able to run queries 1.4 times faster than Hudi and 1.7 times faster than Iceberg. For a table consisting of 200,000 files, Delta's distributed metadata processing started queries 7 to 12 times faster than Iceberg's single-node processing. Merge-on-read updates gave up lower write amplification in exchange for read amplification [17]. LST-Bench contends that traditional benchmarks do not reflect how these log-structured tables degrade and recover when undergoing continuous modification and maintenance, and it provides a framework for examining that behaviour [18]. The ecosystem of formats is moving towards convergence. The Iceberg v3 specification includes deletion

vectors, row lineage, and a VARIANT type for semi-structured data, with implementations designed to be compatible between Delta Lake and Parquet [21]. DuckLake 1.0 (in April 2026) goes even further by storing all lakehouse metadata in a SQL database rather than in files on object storage and by inlining small inserts in the catalog in order to avoid producing many tiny files [22]. This design directly addresses the small-file problem that our freshness results highlight (Section 5.4).

C. Real-Time OLAP Engines

Real-time OLAP engines are designed to support both user-facing and machine-facing analytics, providing sub-second latency even when there is high query concurrency on data that is continuously being ingested. Apache Druid combines streaming ingestion with the use of immutable, indexed segments [8]. Apache Pinot was developed at LinkedIn in order to provide analytics for hundreds of millions of members [9]. ClickHouse's MergeTree family stores the data in sorted, compressed column parts; it has a sparse primary index and data-skipping indexes which allow it to read only the granules that a query requires, and background merges are used to consolidate the parts over time [7]. These systems have inherited the column-store execution methods that made analytical databases so fast [23]. As a result, they offer weaker support for multi-table joins, row-level updates, and transactional semantics than do data warehouses or lakehouses [7].

D. Agents as Data Consumers

Research on using LLMs as database clients has progressed from simple text-to-SQL queries to more complex agentic workflows involving multiple steps. Fernandez et al.claimed that LLMs would have a disruptive effect on data management in the areas of discovery, integration, and querying [24]. The benchmarks have shifted from the type of single queries seen in Spider to the large and noisy databases used in BIRD [14] and then on to the enterprise workflows featured in Spider 2.0 [13]. Studies in the field of prompt engineering, such as DAIL-SQL, have shown just how much task performance relies on the way the schema is represented and on the examples provided in the context [25]. With regard to the data system, the most direct description is that offered by Liu et al. in their work on agentic speculation [12]. These agents carry out a great many more queries than people do. The types of queries vary, ranging from metadata probes to heavy aggregations. A large number of them are repeated or very similar to each other. Moreover, the agents can be guided by the hints that the system sends them. Liu et al. suggest some research directions concerning agent-oriented query interfaces, speculative query processing, and agentic memory stores. Our study builds on that position paper by providing a measurement of how the various current architecture families perform when faced with a workload that has these characteristics.

E. Comparative Evaluation of Analytical Systems

Tan et al. carried out a comparison of cloud DBMS architectures with respect to their performance and cost, demonstrating that the choice of storage, the use of caching, and the pricing model can reverse the performance rankings [15]. The Cloud Analytics Benchmark obtained query-arrival patterns from actual Snowflake telemetry. It discovered that 57.6% of the queries involve both reads and writes, and not just reads, and that the load of a single customer can vary by more than an order of magnitude within a day [16]. The Redshift fleet analysis revealed that in half of all clusters 80% of the queries are exact repetitions of previous ones, and that more than half of the queries are completed in less than 100 ms [19]. These results are relevant in this context since agentic load strengthens both of these properties: query streams become more repetitive and more latency-sensitive. Table 1 provides a summary of how this body of literature relates to the present study.

Table 1: Related Work and Its Coverage of Agentic Data-Layer Requirements.

Work	Architecture(s)	Workload basis	Agent-shaped workload?	Freshness measured?	Relation to this study
Snowflake, Dremel, Napa [1], [3], [20]	Warehouse	Production / system design	No	Napa: tunable freshness	Warehouse design principles; freshness/performance/cost trade-off
Lakehouse, Delta, Photon [4], [5], [6]	Lakehouse	TPC-DS, production	No	No	Lakehouse design principles
Jain et al.; LST-Bench [17], [18]	Lakehouse formats	TPC-DS, modification + maintenance cycles	No	Partly (degradation under updates)	Motivates our append-fragmentation test

ClickHouse, Druid, Pinot [7], [8], [9]	Real-time OLAP	Production, ClickBench-style	No	Ingest-to-query in design	OLAP design principles
Tan et al.; CAB; Redshift fleet [15], [16], [19]	Cloud DBMSs	TPC, Snowflake and Redshift traces	No (human/scheduled)	No	Repetition, short queries, cost variability
Spider 2.0, BIRD, DAIL-SQL [13], [14], [25]	Warehouses, SQLite, others	Text-to-SQL tasks	Yes (task accuracy)	No	Agent capability, not data-system behavior
Agent-first data systems [12]	Architecture-agnostic	Agent traces on data tasks	Yes	No	Characterizes speculation; we measure its impact on three families
This study	All three (proxies)	Agentic query workload	Yes	Yes (read-your-writes, fragmentation)	—

III. REQUIREMENTS OF AGENTIC APPLICATIONS ON THE DATA LAYER

The following seven requirements are obtained from the literature mentioned above, and each one is stated in such a way that it can be tested or at least evaluated.

R1: The ability to tolerate speculative bursts. When the agents are exploring [12], they send out scaled, heterogeneous, and redundant bursts of queries, and currently it is still difficult for agents to handle enterprise workflows that require multi-step exploration [13]. The data layer needs to respond quickly to a large number of small to medium-sized analytic queries and should make use of redundancy by means of caching. Data from the fleet indicate that exact repetition is already a common feature of workloads from the human era [19].

R2: Access at the level of individual entities. The actions carried out by an agent typically relate to a particular customer, order, instrument, or ticket. When you need to retrieve the entity's recent history during the reasoning process, this involves a point or local-range query, not a full scan.

R3: The freshness associated with read-your-writes. Agents are now acting on operational data. If an agent or the system that it works with writes an event, then later reasoning must be able to see it. Freshness has to be a measurable feature of the architecture, not merely an assumption based on scheduling, and Napa demonstrates that it has a direct trade-off with query performance and cost [20].

R4: Open and supported by multiple engines. Agentic applications integrate SQL analytics, feature computation, embedding generation, and model evaluation. It is more difficult to extend architectures that rely on a single engine than those that can be used by multiple engines [4], [21].

R5: Machine-readable semantics and governance. In order to function properly, agents require schemas that are discoverable, governed metric definitions, lineage information, and access policies which can be obtained as context. The way in which schema is represented has a strong impact on text-to-SQL accuracy [25], and the enterprise schemas included in Spider 2.0 are a major source of difficulty [13].

R6: Under bursty loads the cost remains elastic and predictable. Since agent traffic is bursty and can be increased by speculation, cost controls must be designed to limit the worst-case expenditure, not just the average expenditure, because real cloud workloads already vary by more than an order of magnitude within a single day [16].

R7: Multimodal retrieval. By means of vector similarity search [26], [27] agents combine structured queries with a retrieval of information from unstructured content. The architecture has to either contain the vector indexes or be able to integrate smoothly with a system that does. The values of R1 to R3 can be obtained on a single node, as described in Section 5, while those of R4 to R7 depend on the ecosystem, the service model, and the pricing, and therefore they are evaluated based on published evidence in Section 6.

IV. METHODOLOGY

A. Architecture Proxies

To compare commercial services directly would be to combine considerations such as architecture with network placement, pricing tier, cluster size, and proprietary optimisations. Instead, we create one open-source proxy for each architecture family, run all three on the same hardware, and ensure that each proxy remains closely aligned with the storage model which defines its family (Figure 1).

- Warehouse-style (W): DuckDB 1.5.5 [28] features its native storage file. Just as in a warehouse, one engine has control of a managed columnar format together with row-group min/max statistics, and the catalog and engine are closely linked.
- Lakehouse (L): a Delta Lake table created using delta-rs 1.6.6 (comprising ten Parquet files each containing 500,000 rows when loaded), and it is scanned by the same DuckDB engine. Prior to carrying out each query, the proxy determines the most recent table snapshot from the Delta transaction log in order to get the current list of files, since an engine processing an open table format has to do this in order to see the new commits [5]. The fact that W and L make use of the same execution engine means that any differences between them can be attributed to the storage and metadata layer.
- For real-time OLAP (O): ClickHouse 26.7 was used via chDB 4.4.0, with a MergeTree table ordered by (customer_id, ts), dictionary encoding for the categorical columns where low-cardinality was applied, and background part merging [7].

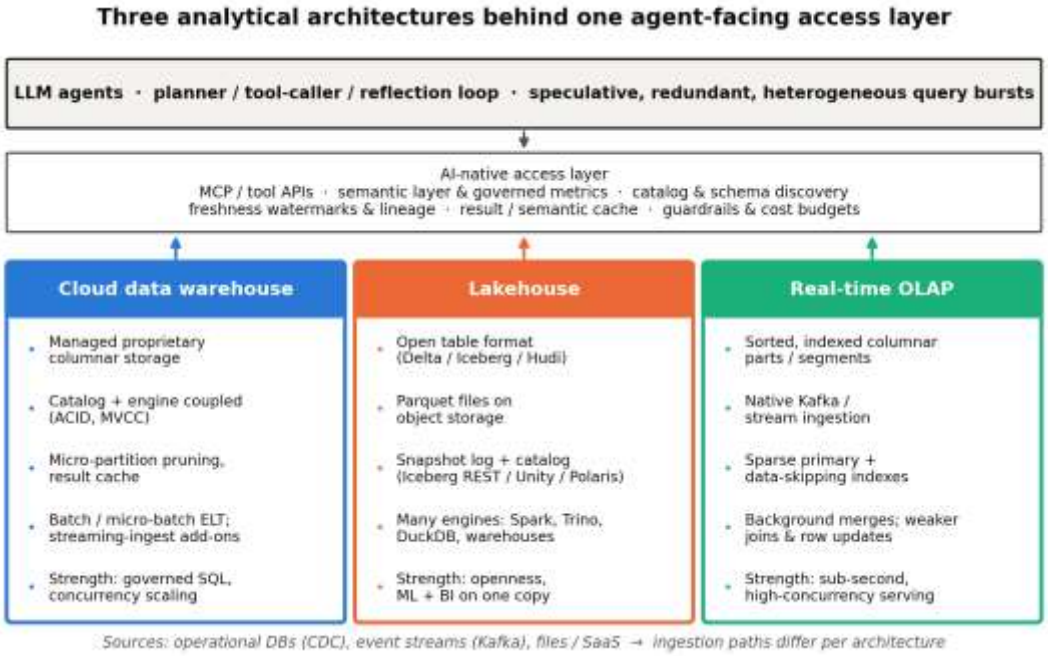


Figure 1: The Three Architecture Families Compared in This Study and the AI-Native Access Layer Proposed in Section 6. Each Family Reaches the Agent Through the Same Access Layer; What Differs Is the Storage Model, How Data Is Ingested, And How the Catalog Is Coupled to the Engine.

B. Dataset

We create a synthetic order-event table consisting of 5,000,000 rows over a period of 90 days and including ten columns: order_id, ts, customer_id, product_id, region, category, channel, qty, price, and status. Customer activity is based on a Zipf distribution (with an exponent of 1.3 and 200,000 customers) in order to mimic the hot-entity skew that is typical in transactional systems. There are 20,000 products, 8 regions, 20 categories, 4 channels, and 5 order statuses, the proportions of which are realistic. Prices follow a log-normal distribution. The rows are inserted in chronological order of timestamp so that zone maps and file statistics can be used for pruning. The generator is seeded and each engine receives the same Arrow data.

C. Agentic Query Workload

The workload has four parts, each tied to an agent behavior and a requirement:

- The schema and sample probes (R1, R5). There are six types of probe (including the sample rows, the number of rows, the time range, and the distinct values for the region, category, and status), each of which is run five times, giving a total of

30 queries. These probes represent the operations carried out by the schema discovery and value enumeration agents before they generate SQL [12], [13]. We record the per-query p50 and p95 latency.

- Speculative analytic burst (R1). This involves 240 queries taken from six different templates (including filtered group-bys, top-k products, daily revenue series, status distributions, distinct-customer counts, and per-customer aggregates) with randomly selected parameters. Fifty-six point seven percent of the queries are identical to one that had been executed before, illustrating the extent of the scale, the heterogeneity, and the redundancy characteristic of agentic speculation [12] as well as the kind of repetition observed in fleet workloads [19]. We record the p50 and p95 latency values as well as the total burst time.
- For entity point lookups (R2), we carry out 20 queries of the type «last 20 orders for customer *c*», which correspond to the mid-reasoning retrieval of one entity's history, and we measure the p50 and p95 latency.
- For micro-batch freshness (R3), we perform 100 appends, each consisting of 1,000 rows; after each append there is a read-your-write query which has to display the newly added rows. A variety of queries are then executed both before and after compaction. We record the commit latency, the commit-to-visible latency, the number of storage units (that is, row groups, files, or parts), and the scan latency before and after compaction.

D. Procedure and Validity Controls

The experiments were carried out within a single Linux container which had 2 vCPUs and 8 GB of RAM allocated on the local disk. For each engine we first carried out a warm-up run, followed by three measured runs for each component of the workload, and the median run (that is, the one with the median total time) was reported. The totals for bursts per engine varied by no more than 12% across the different runs, and a complete independent re-execution reproduced each of the reported medians within a margin of 12%. Correctness was verified. All three engines produced the same number of rows and the same total revenue both before and after the freshness test (5,000,000 rows; 5,701,633,052 revenue units prior and 5,100,000 rows; 5,816,253,960 revenue units after). Latency was measured on the client side and takes into account the process of materialising the results in Python. In the case of chDB this involves the CSV serialisation of the results, which introduces a small and roughly constant extra cost. The benchmark script and the original results can be obtained from the author upon request, and the workload can be regenerated using the seeds mentioned.

V. RESULTS

A. Probes and Speculative Bursts (R1)

Figure 2 and Table 2 provide a summary of the latency. For schema and sample probes, the warehouse-style and OLAP proxies had medians of 6.3 ms and 6.4 ms respectively, while the lakehouse proxy took about twice as long—12.3 ms. A large part of the lakehouse proxy's higher latency is due to it resolving a snapshot before each probe. During the 240-query speculative burst, the warehouse-style proxy was by far the fastest: it had a median latency of 7.2 ms, a p95 value of 20.2 ms, and the whole burst took 2.2 seconds. The lakehouse proxy had a median latency of 21.3 ms (amounting to 5.6 seconds in total), and the OLAP proxy took 24.3 ms (5.7 seconds in total). The OLAP proxy's sort key, which is (customer_id, ts), is favourable for access to entities. Since the burst consists mainly of region-, category-, and time-filtered aggregates that cannot use this key, ClickHouse has to scan more granules than the row groups in DuckDB, which are ordered by timestamp, require. This is a design decision, not a defect of the engine, and it demonstrates that a real-time OLAP table is optimised for the access path that is most common, while an agent's exploratory queries seldom follow any one particular path.

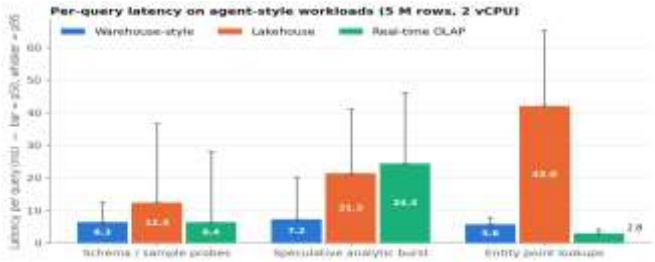


Figure 2 Shows the Per-Query Latency for the Three Latency-Oriented Workload Components (Bar Representing the P50, Whisker the P95; Based on the Median of Three Passes, 5 million Rows, 2 Vcpu). The Warehouse-Style Proxy Achieves an Advantage During Speculative Analytic Bursts, The OLAP Proxy Performs Better During Entity Lookups, And the Lakehouse Proxy Ends Up Paying the Price for Per-Query Snapshot Resolution and for the Lack of an Entity-Oriented Layout.

Redundancy presents a substantial opportunity that has so far been unused; since only 43.3% of the queries made in the burst were unique, a result cache based on the normalized query text could have dealt with 56.7% of them without having to access storage. In this setup, none of the three proxies had a cross-query result cache turned on. It is typical for commercial warehouses to offer such a feature, which is therefore an advantage for the warehouse and one that our proxy does not take into account.

B. Entity Point Lookups (R2)

Entity access resulted in the greatest difference. The OLAP proxy responded with "the last 20 orders for customer *c*" in 2.8 ms at the median (p95 4.1 ms) since its main sort key enables it to read only the granules belonging to that customer. The warehouse-style proxy required 5.6 ms (p95 7.8 ms). Although the row-group statistics provide only limited pruning when the customer column is unsorted, its native storage format makes a full column scan inexpensive. The lakehouse proxy needed 42.1 ms (p95 65.3 ms), which is 15 times the OLAP median, since each lookup has to resolve a snapshot and then scan every file; file-level min/max statistics are unable to prune on customer_id when the files are arranged by time.

C. Metadata Overhead in the Lakehouse

In order to separate metadata resolution from the scanning process, we re-ran the lakehouse burst and lookups using a cached snapshot. As a result, the median burst latency decreased from 21.3 ms to 17.6 ms and the lookup latency dropped from 42.1 ms to 37.9 ms. Therefore, in this setup snapshot resolution amounts to about 4 ms per query, which is roughly 17% of the median burst latency. When the files are between ten and a hundred in number the overhead is small. However, when using object storage, there are remote catalog round-trips and the tables consist of thousands of files—this is the situation in which Jain et al. measured metadata-dominated start-up for large file counts [17]—the overhead increases. If an agent is making hundreds of speculative queries then it has to bear this cost on each one unless the access layer caches the snapshots within a bounded staleness window. This is precisely the freshness trade-off that Section 6 makes explicit.

Table 2: Summary Of Measured Results (Median of Three Passes; Ms Unless Stated).

Metric	Warehouse-style (DuckDB native)	Lakehouse (Delta + DuckDB)	Real-time OLAP (ClickHouse/chDB)
Initial load of 5 M rows (s)	2.53	1.78	4.43
Probe latency p50 / p95	6.3 / 12.6	12.3 / 36.6	6.4 / 28.2
Speculative burst p50 / p95	**7.2 / 20.2**	21.3 / 41.0	24.3 / 46.1
Burst total, 240 queries (s)	**2.20**	5.63	5.71
Burst p50 with cached snapshot	—	17.6	—
Entity lookup p50 / p95	5.6 / 7.8	42.1 / 65.3	**2.8 / 4.1**
Micro-batch commit p50 / p95	**4.6 / 5.4**	28.0 / 50.2	4.9 / 8.4
Commit-to-visible p50 / p95	**5.2 / 6.1**	50.3 / 87.1	7.3 / 11.7
Storage units before → after 100 appends → after compaction	41 → 42 → 42 row groups	10 → 110 → 1 files	1 → 5 → 1 parts
Compaction time (s)	0.07 (checkpoint)	2.12	1.71
Mixed-scan p50 fragmented → compacted	4.9 → 5.5	50.6 → 58.9	4.3 → 2.8

D. Freshness and Fragmentation (R3)

The micro-batch test shows the greatest differences between the architectures (see Figure 3). When 1,000 rows were appended, the median time taken was 4.6 ms using the warehouse-style proxy and 4.9 ms with the OLAP proxy. The data appeared to a reader after 5.2 ms and 7.3 ms respectively. With the lakehouse proxy a commit lasted 28.0 ms: this consisted of writing a new Parquet file and making a transaction-log entry. The time from commit to when the data became visible was 50.3 ms at the median and 87.1 ms at the p95 level, since the reader also has to resolve the new snapshot. These absolute figures are low since all the operations are local. What is important is the ratio and the underlying structural cause. Each lakehouse commit produces at least one data file and one log entry. After 100 appends the Delta table contained 110 data files, having increased from 10. The OLAP proxy's background merger had already reduced its new parts to 5, and the warehouse-style proxy added just one row group.

The way in which compaction was carried out varied from family to family. Combining the parts of the OLAP proxy into a single unit improved its mixed-query median from 4.3 ms to 2.8 ms. However, compacting the Delta table into a single file—

which took 2.1 s—resulted in a slower mixed-query median, *dr*opping from 50.6 ms to 58.9 ms. By having one large file, the file-level parallelism and the time-based file skipping that were available with the original ten-file arrangement were lost. The takeaway for agentic deployments is that lakehouse maintenance should aim for a file size and clustering that are suitable for the query mix, rather than just trying to reduce the number of files. LST-Bench was developed in order to examine precisely this interaction between maintenance and performance over time [18]. More recent designs focus on addressing the fundamental cause. DuckLake incorporates small inserts directly into its SQL catalog until they can be flushed in bulk [22], and Iceberg v3 deletion vectors reduce the write amplification caused by row-level changes [21].

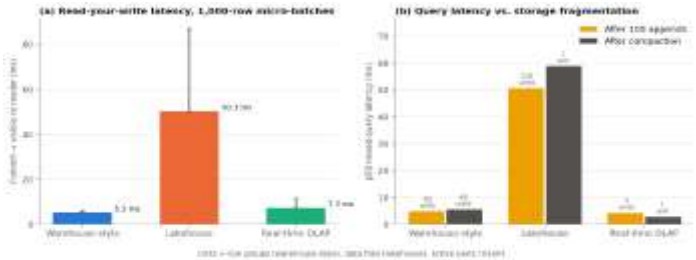


Figure 3 (A): The Commit-To-Visible Latency Is Given for Micro-Batches Consisting Of 1,000 Rows (The Bar Represents the P50 And the Whisker the P95). In (B) The Median Latency of a Mixed Query Set Is Shown After 100 Appends and After Compaction, With the Number of Storage Units (That Is, Row Groups, Data Files, Or Active Parts) Noted. When Lakehouse Performs a Commit, It Creates Files That Then Have to Be Compacted; The OLAP Engine Merges the Parts in the Background; And the Warehouse-Style Store Incorporates the Small Appends into The Existing Row Groups.

VI. DISCUSSION: THE SELECTION AND LAYERING OF ARCHITECTURES FOR AGENTS

A. Requirement-to-Architecture Matrix

Table 3 includes our measurements for R1 to R3 together with the published data for R4 to R7. The ratings are given relative to one another in this context: ●●● means strong, ●● means adequate and ● means weak. These ratings refer to the architecture family rather than to any specific product, and a feature provided by a vendor can cause a particular system to be rated higher or lower.

Table 3: Fitness Of Each Architecture Family Against Agentic Data-Layer Requirements.

Requirement	Warehouse	Lakehouse	Real-time OLAP	Basis
R1 Speculative burst tolerance	●●●	●●	●●	§5.1; warehouse result caches exploit redundancy [19]
R2 Entity-level access	●●	●	●●●	§5.2; sort-key/sparse-index design [7]
R3 Read-your-writes freshness	●●	● (●● with inlining / streaming writers)	●●●	§5.4; native stream ingestion [8], [9]; DuckLake inlining [22]
R4 Open multi-engine access	● (●● via Iceberg interop)	●●●	●	Open table formats [4], [21]
R5 Machine-readable semantics & governance	●●●	●● (●●● with unified catalog)	●	Managed catalogs vs. open catalogs; schema context drives agent accuracy [25], [13]
R6 Elastic, predictable cost under bursts	●●	●●	●● (●●● at high QPS)	Pricing and elasticity dominate at cloud scale [15], [16]
R7 Multimodal/vector retrieval	●●	●●	●●	Vector search is increasingly offered as an extension or integration in all three; purpose-built vector DBMSs and ANN

				indexes remain the reference point [26], [27]

No column is strong on every row. The warehouse is the best default for exploratory analytic agents: data analysts' copilots and planning agents that mostly aggregate. The real-time OLAP engine is the best serving layer for operational agents that look up entities and act on fresh events. The lakehouse is the best system of record when the same data must feed agents, feature pipelines, and model training through several engines. For these agents it needs a serving or caching tier in front of it and deliberate file-layout management behind it.

B. The AI-Native Access Layer

Since no one architecture is superior, we claim that the 'AI-native' aspect of an agentic data architecture should be placed above the storage architecture, in an access layer that every agent has to go through (see Figure 1). The functions of this layer are based on the measurements:

- The use of result and snapshot caching with a limit on staleness means that 56.7% of the speculative bursts involved repeating earlier queries and that the cost of resolving a snapshot was approximately 17% of the lakehouse burst latency. A cache that is based on the normalized query text and the table version eliminates redundant speculation. When queries are served from a snapshot that is at most Δ seconds old, the per-query metadata resolution is removed, and Δ can be set on a per-tool basis.
- Freshness is evident in each response, with differences between the various architectures amounting to an order of magnitude and depending on the maintenance status. It should therefore be the case that the layer includes, together with each result, the commit version or timestamp it represents, so that an agent or its guardrail can determine whether it is fresh enough for the intended action.
- Routing that is aware of the workload sends entity lookups to the OLAP tier or to an entity-sorted replica, exploratory aggregates to the warehouse or lakehouse engine, and full-history analytics to the lakehouse; since each of the workload components mentioned in Section 5 had a different winner, routing based on query shape is able to regain the best result from each.
- Semantic context as a tool—metric definitions, schema descriptions, sample values, and lineage should all be made available via MCP-style tools [11]. Instead of having to carry out expensive probes, agents can discover the structure by making simple metadata requests. In this way, the context burden that Spider 2.0 and DAIL-SQL point out [13], [25] is dealt with.
- Budgets for speculation and guardrails; per-agent limits were imposed on the number of queries, the amount of scanned bytes, and the cost per task in order to bound the worst case caused by bursty agent loads [16]. Additionally, the layer is also able to provide steering suggestions, for example 'this table is partitioned by date; add a date filter', in keeping with Liu et al.'s steerability property [12].

C. Reference Patterns

The following three deployment approaches are outlined. *In the warehouse-centric* model, the warehouse acts as the system of record, the access layer provides semantic tools and includes caching, and a small OLAP or key-value layer is used for entity lookups. *With the lakehouse-centric* approach, the open-table-format lakehouse functions as the system of record and a real-time OLAP engine or streaming-ingest catalog takes in the new writes and serves the operational agents. This is referred to as the "speed layer" pattern, adapted for agents. For domains that involve a high volume of events such as trading operations, fraud detection, or observability, the OLAP engine is the primary component and the history is stored in a lakehouse for training and audit purposes. In all these patterns, it is the freshness watermark of the access layer that enables an agent to reason correctly across the various tiers which have different levels of freshness.

VII. THREATS TO VALIDITY AND LIMITATIONS

Construct validity: Our proxies are standing in for entire architecture families, not for individual products. DuckDB's native storage has no cloud warehouse features such as elasticity, result caching, or network hops. Delta Lake when running on local disk underestimates the latency associated with object storage and the catalog, so the actual metadata overhead of a lakehouse is probably greater. chDB is ClickHouse embedded without a server-side query cache. The absolute figures therefore refer to these proxies. The relative patterns—which architecture performs better on each component and why—are based on the structural characteristics that the proxies have in common with their respective families, as documented in Sections 2 and 5. Internal validity: W and L use the same execution engine, which separates out the influence of storage but could advantage the lakehouse compared to engines that have weaker Parquet readers. The results were consistent across different runs and also in

an independent re-execution, and the correctness has been verified, although the measurements are based on single-threaded client streams using only 2 vCPUs. External validity: At 5 million rows, all the data fits into memory. Concurrency, multi-node scaling, cost, and object storage were not included in the study. The workload is synthetic and has been derived from published descriptions of agent behaviour [12], not from actual traces of production agents. Reliability: The workload is seeded and specified in a way that makes it fully reproducible. We encourage others to carry out replications on a larger scale and using managed services, and to extend the work towards concurrency by using CAB-style arrival patterns [16].

VIII. CONCLUSION AND FUTURE WORK

Agents represent a new type of analytical client; they carry out speculation in bursts, retrieve individual entities, and demand up-to-date data. When these behaviours are taken into account, each of the three main architecture families has strengths in some areas and weaknesses in others. In our controlled test, a warehouse-style store was able to respond to speculative bursts three times faster than the lakehouse and the OLAP proxies. A real-time OLAP store carried out entity lookups 15 times faster than the lakehouse and made micro-batch writes visible almost as quickly as the warehouse-style store. The lakehouse, being the most open of the three, had to pay a price for its openness in the form of slower per-query metadata resolution, delayed write visibility, and file fragmentation which compaction has to manage carefully. The practical takeaway is that the selection of a storage architecture should be based on an organisation's system-of-record requirements, particularly with regard to openness and governance. A data architecture is considered to be AI-native because of the access layer situated above it: this layer caches speculative redundancy, attaches freshness watermarks, routes queries according to their shape, exposes semantics as tools, and allocates a budget for speculation. Future work will include (i) repeating the evaluation with managed cloud services that use object storage and have concurrent agents, (ii) replacing the synthetic workload with traces obtained from production agents, and (iii) building and evaluating the access layer end to end, while measuring both agent task success and latency.

IX. REFERENCES

[1] Dageville, B., Cruanes, T., et al. (2016). The Snowflake elastic data warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 215–226).

[2] Vuppalapati, M., Miron, J., et al. (2020). Building an elastic query engine on disaggregated storage. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[3] Melnik, S., Gubarev, A., et al. (2010). Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment, 3*(1–2), 330–339.

[4] Armbrust, M., Ghodsi, A., et al. (2021). Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[5] Armbrust, M., Das, T., et al. (2020). Delta Lake: High-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment, 13*(12), 3411–3424.

[6] Behm, A., Palkar, S., et al. (2022). Photon: A fast query engine for lakehouse systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[7] Schulze, R., Schreiber, T., et al. (2024). ClickHouse, lightning fast analytics for everyone. *Proceedings of the VLDB Endowment, 17*(12), 3731–3744.

[8] Yang, F., Tschetter, E., et al. (2014). Druid: A real-time analytical data store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 157–168).

[9] Im, J.-F., Gopalakrishna, K., et al. (2018). Pinot: Realtime OLAP for 530 million users. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

[10] Yao, S., Zhao, J., et al. (2023). ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*.

[11] Anthropic. (2024). *Introducing the Model Context Protocol*.

[12] Liu, S., Ponnapalli, S., et al. (2026). Supporting our AI overlords: Redesigning data systems to be agent-first. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[13] Lei, F., Chen, J., et al. (2025). Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows. In *International Conference on Learning Representations (ICLR)*.

[14] Li, J., Hui, B., et al. (2023). Can LLM already serve as a database interface? A big bench for large-scale database grounded text-to-SQLs. In *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*.

[15] Tan, J., Ghanem, T., et al. (2019). Choosing a cloud DBMS: Architectures and tradeoffs. *Proceedings of the VLDB Endowment, 12*(12).

[16] van Renen, A., & Leis, V. (2023). Cloud analytics benchmark. *Proceedings of the VLDB Endowment, 16*(6), 1413–1425.

[17] Jain, P., Kraft, P., et al. (2023). Analyzing and comparing lakehouse storage systems. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[18] Camacho-Rodríguez, J., Agrawal, A., et al. (2024). LST-Bench: Benchmarking log-structured tables in the cloud. *Proceedings of the ACM on Management of Data, 2*(1).
[19] van Renen, A., Horn, D., et al. (2024). Why TPC is not enough: An analysis of the Amazon Redshift fleet. *Proceedings of the VLDB Endowment, 17*(11), 3694–3706.
[20] Agiwal, A., Khandelwal, K., et al. (2021). Napa: Powering scalable data warehousing with robust query performance at Google. *Proceedings of the VLDB Endowment, 14*(12).
[21] Blue, R., Weeks, D., & Narayanan, A. (2025). *Apache Iceberg™ v3: Moving the ecosystem towards unification*. Databricks Blog.
[22] DuckDB Team. (2026). *DuckLake v1.0: The lakehouse format built on SQL reaches production-readiness*.
[23] Balaji, R. (2025) 'AgenticSecOps for Microservice APIs: Policy-Constrained Multi-Agent Investigation and Response to Application-Layer Incidents', ESP Journal of Engineering & Technology Advancements, 5(3), pp. 212-221. doi: 10.56472/25832646/JETA-V5I3P127
[24] Abadi, D., Boncz, P., et al. (2013). The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases, 5*(3), 197–280.
[25] Castro Fernandez, R., Elmore, A. J., et al. (2023). How large language models will disrupt data management. *Proceedings of the VLDB Endowment, 16*(11).
[26] Gao, D., Wang, H., et al. (2024). Text-to-SQL empowered by large language models: A benchmark evaluation. *Proceedings of the VLDB Endowment, 17*(5).
[27] Pan, J. J., Wang, J., & Li, G. (2024). A survey of vector database management systems. *The VLDB Journal, 33*.
[28] Malkov, Y. A., & Yashunin, D. A. (2020). Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 42*(4), 824–836.
[29] Raasveldt, M., & Mühleisen, H. (2019). DuckDB: An embeddable analytical database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 1981–1984).