



(REVIEW ARTICLE)



A LAYERED REFERENCE FRAMEWORK FOR SCALABLE DATA ENGINEERING AND AI APPLICATIONS

Santosh Kumar Maddali *

Sr. Staff Data Engineer, USA.

* Corresponding Author

World Journal of Advanced Research and Reviews, 2026, 31(03), 1805–1814

Publication history: Received on 18 August 2026; revised on 24 September 2026; accepted on 26 September 2026

Article DOI: <https://doi.org/10.30574/wjarr.2026.31.3.2489>

Copyright © 2026 Author(s) retain the copyright of this article. This article is published under the terms of the Creative Commons Attribution License 4.0

ABSTRACT

Modern organizations depend on data platforms that must serve two audiences at once: analytical workloads that expect consistent, well-governed tables, and artificial intelligence (AI) workloads that expect fresh, versioned, feature-ready data delivered to training and inference systems. Most platforms were not designed for both, and scaling one audience often degrades the other. This paper consolidates established results from the systems and machine learning literature into a layered reference framework for data engineering platforms that scale across volume, velocity, workload diversity, and change. The framework organizes responsibilities into ingestion, storage, processing, feature and model, and serving layers, bound together by cross-cutting schema contracts, quality gates, lineage, and observability. Seven design principles are derived from established systems literature: separating storage from compute, declarative pipeline definitions, idempotent and replayable processing, explicit schema contracts, incremental computation, unified batch and streaming semantics, and automated validation gates. The paper then shows how the same framework supports AI applications through training data management, feature pipelines, model serving feedback loops, and retrieval-augmented applications. An analytical comparison of lambda, kappa, and lakehouse-based architectures against the stated requirements is provided, followed by a discussion of open challenges including cost governance, semantic drift, and the operational burden of unstructured data. The resulting framework provides a vendor-neutral architectural blueprint that systematically connects scalability requirements, platform responsibilities, operational controls, and AI lifecycle needs, enabling engineering teams to evaluate and evolve data and AI platforms incrementally.

Keywords: Data Engineering, Scalable Architecture, Machine Learning Pipelines, Stream Processing, Lakehouse, Data Quality, Mlops, Feature Engineering

1 INTRODUCTION

Data engineering has moved from a supporting discipline to a core capability of the modern enterprise. The same pipelines that once fed nightly reports now feed dashboards refreshed every few minutes, machine learning models retrained on a schedule, and interactive applications that call models at request time. Each of these consumers places different demands on the platform. Reporting values consistency and auditability. Model training values reproducibility and access to long histories. Online inference values low-latency and freshness. A platform that is scaled for one consumer without regard for the others tends to accumulate parallel copies of the same data, each with its own

definitions, quality rules, and failure modes. The technical debt that results has been documented extensively in the machine learning systems literature [1], [2].

The word scalable is often used narrowly to mean the ability to process more bytes. In practice, scalability for data and AI platforms has at least four dimensions: scaling with data volume and arrival rate, scaling with the number and diversity of workloads, scaling with the rate of change in sources and schemas, and scaling with the number of teams that build on the platform. A framework that addresses only the first dimension will still fail in production, because the other three dimensions are where most operational incidents originate [3], [4].

This paper organizes established results into a layered reference framework that treats these dimensions together. The framework is deliberately vendor-neutral. It draws on well-established systems research, including distributed batch processing [5], [6], unified stream and batch semantics [7], [8], durable event logs [9], columnar analytical storage [10], transactional table formats on object storage [11], [12], and the production machine learning literature [1], [3], [4], [13]. The framework does not prescribe a specific product. Instead, it specifies responsibilities per layer, the contracts between layers, and the design principles that allow each layer to scale independently.

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 describes how the framework was derived. Section 4 states the scalability requirements. Section 5 presents the layered framework. Section 6 derives seven design principles. Section 7 applies the framework to AI applications. Section 8 provides an analytical comparison of architectural patterns. Section 9 discusses open challenges, and Section 10 concludes.

The scope of this paper is deliberately limited. The framework and principles are a synthesis and organization of the cited public literature. The paper does not describe the systems, data, or practices of any specific organization, does not report measurements from any deployment, and introduces no proprietary method. Every layer, principle, and comparison that follows is traceable to the published work reviewed in Section 2.

2 RELATED WORK

2.1 Distributed Data Processing

The MapReduce programming model [5], built on a distributed file system [14], established the pattern of moving computation to data and tolerating machine failure through re-execution of deterministic tasks. Apache Spark generalized this model with in-memory resilient datasets and a unified engine for batch, interactive, and iterative workloads [6], while Spark SQL added a relational optimizer over the same engine [15]. For unbounded data, the Dataflow model separated the questions of what is computed, where in event time, when in processing time, and how refinements relate [7]. Apache Flink demonstrated that a single streaming runtime can execute both batch and streaming programs with exactly once state semantics [8]. Durable, partitioned logs such as Kafka provide the transport that decouples producers from consumers and allows replay [9]. These systems form the computational substrate of the processing layer described later in this paper.

2.2 Storage Architectures

Columnar storage and tree-structured query execution allowed interactive analysis over very large nested datasets [10]. The lakehouse pattern proposes placing transactional table formats directly on low-cost object storage so that data warehousing and machine learning workloads can share a single copy of the data [11], [12]. Kleppmann surveys the broader design space of replication, partitioning, and consistency that constrains any such storage layer [16]. Earlier architectural proposals, including the lambda architecture with separate batch and speed layers [17], and the kappa proposal that replaces the batch layer with log replay [18], remain influential and are compared analytically in Section 8. Stonebraker and Cetintemel argued that no single engine serves all workloads well [19], an observation that motivates the layered separation used here.

2.3 Machine Learning Systems and Operations

Sculley et al. identified data dependencies, configuration debt, and feedback loops as sources of hidden technical debt in machine learning systems [1]. Polyzotis et al. framed data management for production machine learning as a first-class research problem covering understanding, validation, cleaning, and preparation [3]. TFX described a production-scale platform integrating data validation, transformation, training, and serving [13], and the ML Test Score provides a rubric for assessing readiness [4]. Schelter et al. showed how declarative constraints can automate data quality verification at scale [20]. Paleyes et al. surveyed deployment challenges across many case studies [2], and Kreuzberger et al. consolidated MLOps into a set of principles and an architecture [21]. Lifecycle tooling for tracking experiments and models has been described in MLflow [22]. Frameworks for large-scale training and distributed AI workloads include TensorFlow [23] and Ray [24]. Data mesh argues for domain ownership and data as a product as organizational

scaling mechanisms [25], and the Ground project proposed a shared data context service for lineage and metadata [26]. This paper builds on these threads by locating each concern within one coherent framework.

3 FRAMEWORK DERIVATION APPROACH

The framework is derived through a structured synthesis of the systems and machine learning literature reviewed in Section 2. First, recurring scalability concerns were grouped into four requirement categories: volume and velocity, workload heterogeneity, correctness under change, and operational and cost constraints. Second, architectural responsibilities described across distributed processing, transactional storage, streaming, and production machine learning literature were mapped to bounded platform layers. Third, recurring mechanisms for reliability and operational control were consolidated into cross-cutting capabilities. Finally, the design principles were derived by identifying architectural properties that directly address one or more scalability requirements. The resulting framework therefore represents a traceable synthesis of established architectural concepts rather than an organization-specific implementation. Table 1 records the traceability from each requirement category to the principles and capabilities that address it.

Table 1: Traceability from scalability requirements to framework principles

Scalability requirement	Supporting principles and capabilities
Volume, velocity, and variety	Separate storage from compute; incremental computation; idempotent and replayable processing
Workload heterogeneity	Separate storage from compute; unified batch and streaming semantics
Correctness under change	Explicit schema contracts; idempotent and replayable processing; automated validation gates
Operational and cost constraints	Incremental computation; automated validation gates; cross-cutting observability and cost management

4 SCALABILITY REQUIREMENTS

This section states the requirements that the framework must satisfy. They are expressed as properties rather than numeric targets, because targets are specific to an organization and change over time.

4.1 Volume, Velocity, and Variety

The platform must accept data whose total size grows without a predetermined bound, whose arrival rate varies by orders of magnitude between quiet and peak periods, and whose shape ranges from fixed schema relational records to semi-structured events and unstructured documents, images, and audio. Capacity should be expandable without requiring fundamental architectural redesign. This requirement favors architectures that decouple storage and computation, since tightly co-located designs can make independent scaling more difficult and may require storage and compute resources to grow together [11], [12].

4.2 Workload Heterogeneity

Analytical queries, scheduled transformations, continuous stream processing, model training, and online inference must run against the same governed data without interfering with one another. Isolation is required so that a heavy training job cannot starve a latency sensitive dashboard. The observation that one engine cannot serve every workload equally well [19] implies that the framework must allow multiple engines to read and write shared storage through open formats.

4.3 Correctness Under Change

Sources change schema, semantics, and volume. Pipelines are edited by many engineers. Models are retrained. The platform must preserve correctness across these changes, which requires explicit contracts at layer boundaries, versioning of both data and code, and the ability to reprocess historical data deterministically. Late and out of order events must be handled according to stated semantics rather than silently dropped [7].

4.4 Operational and Cost Constraints

Every layer must expose health, freshness, and cost signals. Failures must be diagnosable from lineage and metadata rather than from reading code. Costs must be attributable to the workload and team that incurred them, since elastic

infrastructure removes the natural ceiling that a fixed cluster provided. Production readiness criteria for machine learning systems, such as those in the ML Test Score [4], apply equally to the data pipelines that feed those systems.

5 A LAYERED REFERENCE FRAMEWORK

Figure 1 shows the framework. It organizes the layered separation described in the lakehouse and production machine learning literature [12], [13], [21] into a single structure. Five layers carry data upward from sources to consumers. A set of cross-cutting capabilities applies to every layer. Each layer has a bounded responsibility, a defined interface to the layer above, and a scaling mechanism that does not depend on the other layers.

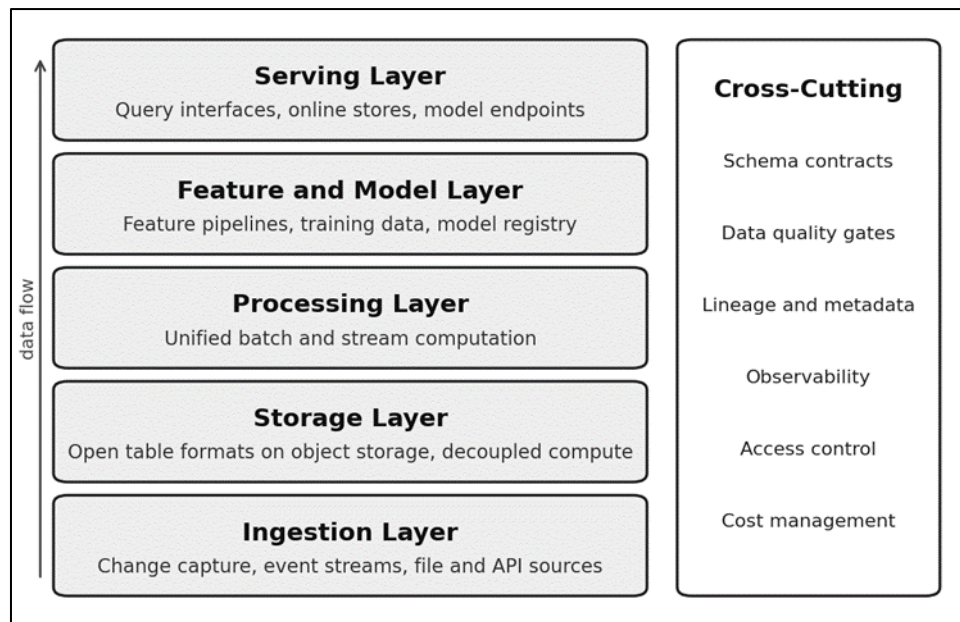


Figure 1: Layered reference framework with cross-cutting capabilities

5.1 Ingestion Layer

The ingestion layer moves data from operational sources into the platform with minimal transformation. It supports three source classes: change data capture from transactional databases, event streams from applications and devices, and periodic file or API extracts. Its output is an immutable, append-only record of what arrived, when it arrived, and from where. A durable partitioned log is the natural interface for this layer because it decouples producer rate from consumer rate and allows any consumer to replay from an offset [9]. The scaling mechanism is partitioning: adding partitions increases parallelism without changing the producer or consumer code.

5.2 Storage Layer

The storage layer holds data in open columnar formats on object storage, with a transactional table layer providing atomic commits, schema enforcement, and time travel [11]. Separating storage from compute is a foundational scaling decision in the framework. It allows storage to grow independently of compute, allows multiple engines to read the same tables, and allows historical versions to be retained for reproducibility [12]. Data is organized into progressively refined zones: a raw zone that preserves source data, a curated zone with enforced schemas and quality controls, and a consumption zone shaped for analytical or AI workloads. The scaling mechanism is elastic object storage, which provides highly scalable capacity while allowing throughput to increase through partitioning, parallel access, and independently provisioned compute resources.

5.3 Processing Layer

The processing layer transforms data between zones. It provides both batch and streaming execution under a single semantic model, so that a transformation written once can run over a bounded historical range or an unbounded live stream [7], [8]. Transformations are expressed declaratively and are organized as a directed acyclic graph with explicit dependencies. The scaling mechanism is horizontal: elastic compute pools are sized per workload and released when idle. Because storage is separate, processing clusters are disposable and can be created per team, per environment, or per job [6], [15].

5.4 Feature and Model Layer

The feature and model layer is where the framework becomes specific to AI applications. Following the component structure described in the MLOps and production machine learning literature [13], [21], it contains feature pipelines that derive model inputs from curated tables, a feature store with offline and online views that share a single definition, training dataset snapshots that are versioned and reproducible, and a model registry that records lineage from data version to model artifact [13], [21], [22]. Placing this layer above the processing layer ensures that features are derived from governed data rather than from ad hoc extracts, which addresses several of the debt sources identified by Sculley et al. [1]. The scaling mechanism is reuse: a feature defined once is materialized once and consumed by many models.

5.5 Serving Layer

The serving layer exposes data and models to consumers. It includes analytical query interfaces over the consumption zone, low-latency key-value stores for online features, model inference endpoints, and application programming interfaces for downstream systems. The serving layer primarily exposes outputs produced by lower layers and should avoid becoming the authoritative location for reusable transformation or business logic. Limited request-time processing may still be required for inference, ranking, filtering, or response composition. The scaling mechanism is replication and caching, which are appropriate for read-heavy access patterns [16].

5.6 Cross-Cutting Capabilities

Six capabilities apply across all layers. Schema contracts define the shape and meaning of every interface between layers and are versioned as code. Data quality gates evaluate declarative constraints at zone boundaries and block promotion when constraints fail [20]. Lineage and metadata record how every table, feature, and model was produced [26]. Observability exposes freshness, volume, latency, and error signals per pipeline. Access control applies consistent policy regardless of the engine used to read the data. Cost management attributes spend to workloads and teams. Table 2 summarizes the layers.

Table 2: Layer responsibilities, interfaces, and scaling mechanisms

Layer	Primary responsibility	Interface to next layer	Scaling mechanism
Ingestion	Capture source data faithfully	Immutable partitioned log	Partitioning
Storage	Durable, versioned, open format tables	Transactional table API	Object store capacity
Processing	Batch and stream transformation	Curated and consumption zones	Elastic compute pools
Feature and Model	Features, training sets, model artifacts	Feature store and registry	Definition reuse
Serving	Query, online lookup, inference	Query and inference endpoints	Replication and caching

6 DESIGN PRINCIPLES

The following principles are derived from the requirements in Section 4 and the literature reviewed in Section 2. Each principle is stated with its rationale and the failure it prevents.

6.1 Separate Storage from Compute

Data should be held in open formats on object storage so that any suitable engine can attach to it. This principle allows the storage and processing layers to scale independently and permits multiple engines to coexist [11], [12], [19]. It reduces the risk that a single cluster must be sized for peak compute and total storage simultaneously.

6.2 Declarative Pipeline Definitions

Transformations should be declared as queries or configuration rather than as imperative programs wherever the logic permits. Declarative definitions can be optimized by the engine [15], executed under a common batch and stream model [7], validated statically, and understood by tools that produce lineage. This principle reduces the risk that pipeline logic becomes opaque to everyone except its original author.

6.3 Idempotent and Replayable Processing

Transformations should produce the same output when run again over the same input, and pipelines should be able to reprocess history from the ingestion log [9], [18]. Idempotence makes retries safe and makes backfills routine. This principle reduces the risk that a partial run leaves duplicated or missing records that can only be repaired by hand.

6.4 Explicit Schema Contracts

The schema and semantics of each interface between layers should be declared, versioned, and enforced at write time [3], [11]. Producers may add fields and widen types; breaking changes should require a new version. This principle reduces the risk that an upstream change silently corrupts downstream tables and models.

6.5 Incremental Computation

Pipelines should process only data that has changed since the previous successful execution, using persisted progress markers such as ingestion log offsets, table versions, or equivalent checkpoints [7], [11]. Full recomputation should be reserved for backfills or recovery scenarios. This principle keeps cost more closely proportional to the volume of change than to total historical data volume and reduces the risk that pipeline runtime grows until it exceeds its schedule window.

6.6 Unified Batch and Streaming Semantics

Where practical, the same transformation definition should support both batch and streaming execution while preserving semantically consistent results. Equivalent event-time, windowing, state, and late-data policies should be applied so that differences arise primarily from execution mode and latency rather than duplicated business logic [7], [8]. This principle reduces the risk of separate batch and streaming implementations diverging over time, one of the principal concerns associated with dual-path architectures [18].

6.7 Automated Validation Gates

Data should be promoted between zones only after declarative quality checks pass [20], and models should be promoted only after data, feature, and model tests pass [4], [13]. Gates should be automated and their results recorded as metadata. This principle reduces the risk that bad data reaches consumers and is discovered by them rather than by the platform.

7 APPLYING THE FRAMEWORK TO AI APPLICATIONS

AI applications are consumers of the platform, but they are unusually demanding consumers. This section shows how the framework meets four recurring needs.

7.1 Training Data Management

Model training requires a dataset that can be reproduced exactly at a later date. The storage layer provides this through table versioning and time travel [11]. A training run records the table versions it read, and the model registry stores that lineage alongside the artifact [22]. Data validation before training detects distribution shifts and schema anomalies that would otherwise surface as unexplained model regressions [3], [13]. Because training datasets are derived from governed and versioned curated data rather than from unmanaged extracts, the framework improves training data consistency, lineage, and reproducibility while reducing the risk of undocumented differences between successive training runs [1], [3].

7.2 Feature Pipelines

Features are defined once in the feature and model layer and materialized to both an offline table for training and an online store for inference. The offline and online views are produced by the same declarative definition executed in batch and streaming modes respectively, which is possible only because the processing layer provides unified semantics (Principle 6.6). Using a shared feature definition for offline training and online inference also reduces training-serving skew by minimizing differences between feature computation paths. Point-in-time correctness is enforced by joining features to labels on event time, using the event-time and watermark concepts formalized in the Dataflow model [7] to decide when a feature value was actually available. Feature reuse across models is the primary scaling mechanism of this layer. Feature reuse can substantially reduce the marginal engineering and computation required to support additional models that depend on existing features.

7.3 Model Serving and Feedback Loops

The serving layer exposes model endpoints that read online features and return predictions. Predictions, the features used, and any subsequently observed outcomes are written back through the ingestion layer as first-class events. This closes the loop: monitoring for drift, evaluating models against realized outcomes, and assembling the next training set all become ordinary pipelines over ordinary tables, as the MLOps literature recommends [21]. Sculley et al. warned about hidden feedback loops [1]; the framework makes the loop explicit and observable rather than hidden. Distributed execution frameworks designed for AI workloads [24] can be attached to the processing layer for training and batch inference without changing the storage contract.

7.4 Retrieval-Augmented Applications

Applications built on large pretrained models, including transformer-based language models [27], [28], increasingly retrieve organizational content at request time to ground their outputs [29], [30], a practice intended to reduce unsupported or ungrounded generation [31]. Within this framework, a retrieval index is treated as a consumption zone table with an additional vector representation managed by a vector store [32], and the pipeline that chunks, embeds, and indexes documents is an ordinary processing layer pipeline subject to the same schema contracts, quality gates, lineage, and access control as any other. Treating retrieval indexes this way ensures that freshness, provenance, and permissions are handled by the platform rather than reimplemented per application.

8 ANALYTICAL ASSESSMENT

This section compares three architectural patterns against the requirements of Section 4. The comparison is qualitative and analytical; it reflects the structural properties of each pattern rather than measured performance, which depends on implementation and workload. The lambda architecture maintains separate batch and speed layers with a merge at query time [17]. The kappa architecture keeps a single streaming layer and reprocesses history by replaying the log [18]. The lakehouse-based framework described in this paper places transactional tables on object storage and executes batch and streaming transformations through a unified processing layer [12]. The properties assigned to each pattern are taken from the structural descriptions in the cited sources [12], [17], [18], not from observed deployments. Table 3 summarizes the assessment.

Table 3: Qualitative comparison of architectural patterns against scalability requirements

Requirement	Lambda	Kappa	Layered lakehouse framework
Volume and velocity	Scales, but duplicates compute across two layers	Scales through log partitioning; history replay can be expensive	Scales through object storage and elastic compute; well suited to incremental processing
Workload heterogeneity	Batch and stream only; AI support typically requires additional feature or storage components	Stream centric; large analytical workloads may require derived or materialized views	Multiple engines share open tables; explicit feature and model layer included
Correctness under change	Two implementations of logic can diverge	Single implementation; replay can reconstruct derived state when deterministic processing is maintained	Single declarative definition; versioned tables and contracts
Reproducibility for training	Historical reproducibility depends on retention and versioning strategy	Possible through replay, but costly for long histories	Native through table versioning and time travel
Operational burden	High; two systems to monitor and reconcile	Moderate; log retention and replay must be managed	Moderate; many components but one storage contract
Cost attribution	Difficult across merged layers	Simpler; single pipeline per dataset	Per workload through disposable compute pools

The assessment shows that the layered framework does not dominate the alternatives on every dimension. Its operational surface may be larger than a pure kappa architecture because it introduces additional storage, governance, and AI lifecycle components. Its principal advantage is structural: among the patterns compared here, it makes shared governed storage and explicit cross-cutting contracts foundational elements for analytical, batch, streaming, and AI workloads. Lambda and kappa implementations can incorporate similar capabilities by adding shared storage, governance, and lifecycle services, causing their resulting architectures to approach the layered framework described here. This makes incremental adoption possible without requiring organizations to discard existing processing engines.

9 CHALLENGES AND OPEN PROBLEMS

Several challenges remain that the framework structures but does not solve.

- **Cost governance.** Elastic compute removes the natural ceiling that a fixed cluster imposed. Without attribution and budgets enforced at the workload level, spend grows with usage rather than with value. Cost signals must be treated as first-class observability data, a concern raised across deployment case studies [2].
- **Semantic drift.** Schema contracts catch structural change but not changes in meaning, such as a field that keeps its type while its business definition changes. Detecting semantic drift requires statistical monitoring of distributions and, ultimately, human review [3], [20].
- **Unstructured data.** Documents, images, and audio do not fit the tabular contracts that the framework relies on. Metadata tables can describe such assets, but validation, lineage, and quality rules for unstructured content are less mature than their tabular counterparts [3], [12], and evaluating generative outputs built on such content requires evaluation and validation methods that are still maturing [33], [34].
- **Organizational scaling.** The framework provides technical contracts, but it does not by itself assign ownership. Data mesh style domain ownership [25] is complementary and addresses the question of who is accountable for each contract.
- **Model and data co-evolution.** As models are retrained on data that includes their own prior predictions, the feedback loop identified by Sculley et al. [1] becomes a governance problem. The framework makes the loop observable, but policies for when to intervene remain application specific.

10 CONCLUSION

This paper developed a layered reference framework by systematically consolidating established results from distributed data systems, streaming architectures, transactional storage, and production machine learning research. The framework separates ingestion, storage, processing, feature and model, and serving responsibilities, and binds them with cross-cutting schema contracts, quality gates, lineage, observability, access control, and cost management. Seven design principles were derived from the systems literature: separating storage from compute, declarative pipelines, idempotent and replayable processing, explicit schema contracts, incremental computation, unified batch and streaming semantics, and automated validation gates. The framework was shown to support training data management, feature pipelines, serving feedback loops, and retrieval-augmented applications without special-purpose infrastructure. An analytical comparison showed that lambda and kappa implementations can approach the proposed framework as shared governed storage, cross-cutting contracts, and lifecycle services are added.

The principal limitation of this work is that the framework is evaluated analytically rather than through a controlled empirical implementation. The assessment therefore establishes architectural relationships and trade-offs rather than measured performance advantages. Future work will implement the framework under representative analytical, streaming, training, and retrieval workloads to evaluate latency, reprocessing cost, resource isolation, and feature reuse. Further work will also extend the contract model to unstructured and generative AI assets.

STATEMENTS ON COMPLIANCE WITH ETHICAL STANDARDS

Acknowledgments

The author would like to thank industry peers and reviewers for their constructive feedback on earlier drafts of this work. No external funding was received for this research.

Disclosure of conflict of interest

The author declares no conflict of interest.

REFERENCES

- [1] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J. F. Crespo, and D. Dennison, "Hidden Technical Debt in Machine Learning Systems," *Advances in Neural Information Processing Systems* 28 (NIPS 2015), pp. 2503-2511, 2015.
- [2] A. Paleyes, R. G. Urma, and N. D. Lawrence, "Challenges in Deploying Machine Learning: A Survey of Case Studies," *ACM Computing Surveys*, vol. 55, no. 6, Article 114, 2022. DOI: 10.1145/3533378
- [3] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data Management Challenges in Production Machine Learning," *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, pp. 1723-1726, 2017. DOI: 10.1145/3035918.3054782
- [4] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction," *Proceedings of the 2017 IEEE International Conference on Big Data*, pp. 1123-1132, 2017. DOI: 10.1109/BigData.2017.8258038
- [5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107-113, 2008. DOI: 10.1145/1327452.1327492
- [6] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A Unified Engine for Big Data Processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56-65, 2016. DOI: 10.1145/2934664
- [7] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1792-1803, 2015. DOI: 10.14778/2824032.2824076
- [8] [8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine," *IEEE Data Engineering Bulletin*, vol. 38, no. 4, pp. 28-38, 2015.
- [9] [9] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A Distributed Messaging System for Log Processing," *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB)*, 2011.
- [10] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive Analysis of Web-Scale Datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330-339, 2010. DOI: 10.14778/1920841.1920886
- [11] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Luszczak, M. Switakowski, M. Szafranski, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia, "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3411-3424, 2020. DOI: 10.14778/3415478.3415560
- [12] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics," *Proceedings of the 11th Conference on Innovative Data Systems Research (CIDR)*, 2021.
- [13] D. Baylor, E. Breck, H. T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich, "TFX: A TensorFlow-Based Production-Scale Machine Learning Platform," *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1387-1395, 2017. DOI: 10.1145/3097983.3098021
- [14] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google File System," *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 29-43, 2003. DOI: 10.1145/945445.945450
- [15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: Relational Data Processing in Spark," *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 1383-1394, 2015. DOI: 10.1145/2723372.2742797
- [16] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, 1st ed., O'Reilly Media, 2017. ISBN: 978-1-4493-7332-0
- [17] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Real-Time Data Systems*, Manning Publications, 2015. ISBN: 978-1-61729-034-3
- [18] J. Kreps, "Questioning the Lambda Architecture," O'Reilly Radar, 2014. Available: <https://www.oreilly.com/radar/questioning-the-lambda-architecture/>

- [19] M. Stonebraker and U. Cetintemel, "One Size Fits All: An Idea Whose Time Has Come and Gone," *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pp. 2-11, 2005. DOI: 10.1109/ICDE.2005.1
- [20] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, and A. Grafberger, "Automating Large-Scale Data Quality Verification," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1781-1794, 2018. DOI: 10.14778/3229863.3229867
- [21] D. Kreuzberger, N. Kuehl, and S. Hirschl, "Machine Learning Operations (MLOps): Overview, Definition, and Architecture," *IEEE Access*, vol. 11, pp. 31866-31879, 2023. DOI: 10.1109/ACCESS.2023.3262138
- [22] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, and C. Zumar, "Accelerating the Machine Learning Lifecycle with MLflow," *IEEE Data Engineering Bulletin*, vol. 41, no. 4, pp. 39-45, 2018.
- [23] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A System for Large-Scale Machine Learning," *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 265-283, 2016.
- [24] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A Distributed Framework for Emerging AI Applications," *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 561-577, 2018.
- [25] Z. Dehghani, *Data Mesh: Delivering Data-Driven Value at Scale*, O'Reilly Media, 2022. ISBN: 978-1-4920-9239-1
- [26] J. M. Hellerstein, V. Sreekanti, J. E. Gonzalez, J. Dalton, A. Dey, S. Nag, K. Ramachandran, S. Arora, A. Bhattacharyya, S. Das, M. Donsky, G. Fierro, C. She, C. Steinbach, V. Subramanian, and E. Sun, "Ground: A Data Context Service," *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.
- [27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, pp. 5998-6008, 2017.
- [28] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling Laws for Neural Language Models," *arXiv preprint arXiv:2001.08361*, 2020.
- [29] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Kuttler, M. Lewis, W. T. Yih, T. Rocktaschel, S. Riedel, and D. Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*, pp. 9459-9474, 2020.
- [30] W. Fan, Y. Ding, L. Ning, S. Wang, H. Li, D. Yin, T. S. Chua, and Q. Li, "A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models," *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 6491-6501, 2024. DOI: 10.1145/3637528.3671470
- [31] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu, "A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, Article 42, 2025. DOI: 10.1145/3703155
- [32] J. J. Pan, J. Wang, and G. Li, "Survey of Vector Database Management Systems," *The VLDB Journal*, vol. 33, no. 5, pp. 1591-1615, 2024. DOI: 10.1007/s00778-024-00864-x
- [33] Y. Chang, X. Wang, J. Wang, Y. Wu, L. Yang, K. Zhu, H. Chen, X. Yi, C. Wang, Y. Wang, W. Ye, Y. Zhang, Y. Chang, P. S. Yu, Q. Yang, and X. Xie, "A Survey on Evaluation of Large Language Models," *ACM Transactions on Intelligent Systems and Technology*, vol. 15, no. 3, Article 39, 2024. DOI: 10.1145/3641289
- [34] S. Shankar, J. D. Zamfirescu-Pereira, B. Hartmann, A. G. Parameswaran, and I. Arawjo, "Who Validates the Validators? Aligning LLM-Assisted Evaluation of LLM Outputs with Human Preferences," *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST)*, Article 131, 2024. DOI: 10.1145/3654777.3676450