



AI in Cloud-Native Architecture: Rethinking Software Design for Data-Intensive and Machine Learning Systems

Vishwajeet Kharote

Independent Researcher, USA

ORCID: 0009-0002-0791-844X

Utsav Kale

Government Polytechnic, Amravati, India

ORCID: 0009-0005-5350-2949

Gaurav Bhardwaj

APJ Abdul Kalam Technological University, India

ORCID: 0009-0007-6183-3873

Vinay Manchundiya

Independent Researcher, USA

ORCID: 0009-0007-0081-3967

Sajeel Shamsudeen

Independent Researcher, USA

ORCID: 0009-0009-1253-8514

Aishwarya Kannothe Putlumbath

University of Wisconsin Milwaukee, USA

ORCID: 0009-0001-5983-3299

Shirsha Ayyagari

Independent Researcher, USA

ORCID: 0009-0001-8589-1062

Publication History: Received: 01.09.2026; Revised: 29.09.2026; Accepted: 04.10.2026; Published: 15.10.2026.

ABSTRACT: The migration of computing workloads from on-premises data centers to elastic cloud platforms has changed how data is collected, processed, and turned into intelligence. Cloud-native architecture, which treats elasticity, distribution, automation, and failure as first-class design inputs, has become the default foundation for modern data platforms and machine learning systems. Yet AI and data workloads differ in important ways from the web and transactional applications for which cloud-native practice was first developed. They move and transform large volumes of data, depend on long-running training jobs and scarce accelerators, serve models whose behavior is determined by data rather than code alone, and can degrade silently as the world changes. This paper examines cloud-native design from the perspective of AI and data systems and makes four contributions. First, it reinterprets the foundational principles of the approach, including statelessness, loose coupling, and designing for failure, for data pipelines, feature management, and model lifecycles. Second, it surveys the recurring patterns used to build distributed data and AI systems, from logs and feature stores to streaming pipelines, online inference, and resilience strategies for model serving. Third, it examines how training, inference, and continuous data pipelines reshape assumptions about state, consistency, and resource elasticity. Fourth, it analyzes the trade-offs that architects of AI and data platforms



must reconcile, since no single design optimizes for scalability, data correctness, model quality, cost, and operational simplicity at once. The central argument is that building AI on the cloud is not a matter of deploying models on cloud infrastructure. It requires treating data, features, and models as first-class architectural concerns, governed by the same principles of decoupling, versioning, and resilience that cloud-native design applies to services.

KEYWORDS: Artificial intelligence, machine learning systems, data engineering, cloud-native architecture, distributed systems, MLOps, data pipelines, model serving

I. INTRODUCTION

For most of the history of enterprise computing, data and analytics were built around centralized systems that ran on hardware the organization owned. Operational data lived in transactional databases, analytical data was copied in nightly batches into a data warehouse, and predictive models, where they existed at all, were trained offline on extracts and deployed as static rules or scheduled scoring jobs. Capacity was provisioned in advance, scaling meant buying larger machines, and the boundary between the data platform and the applications that consumed it changed slowly. This model was coherent and, for many years, adequate.

The arrival of commodity cloud computing [1] dissolved several of the assumptions on which that model rested. Storage became cheap and effectively unbounded, compute could be requested and released within minutes, and frameworks for distributed processing made it practical to analyze data sets far larger than any single machine could hold [7, 8]. At the same time, machine learning moved from offline analysis into the core of products. Recommendations, search ranking, fraud detection, forecasting, and, more recently, generative AI assistants now run as live services whose quality depends on continuous flows of fresh data. At the scale these systems operate, component failure is a statistical certainty, data arrives late or malformed as a matter of routine, and models drift as the world they describe changes.

Cloud-native architecture is the response to this new reality. It is a set of design principles and patterns oriented toward building applications that are resilient, elastic, and continuously deliverable in distributed environments. The recent rise of AI workloads gives the discipline fresh urgency. Training large models, serving predictions at scale, and moving data through continuous learning pipelines impose demands that ordinary web and transactional applications never did, and they expose the limits of older design assumptions with unusual clarity. The remainder of this paper explores what a cloud-native architecture is the response to this reality for software in general. It is a set of design principles and patterns for building applications that are resilient, elastic, and continuously deliverable in distributed environments. AI and data systems inherit these principles but stress them along different axes. They must manage large and continuously changing data sets, coordinate expensive training jobs across many accelerators, serve models under tight latency budgets, and keep training data, features, and deployed models consistent with one another over time. The remainder of this paper explores what a cloud-native orientation entails for AI and data systems, how established principles and patterns translate to pipelines and models, how AI workloads stress the architecture further, and where the central difficulties lie.

II. FROM TRADITIONAL DATA PLATFORMS TO CLOUD-NATIVE AI SYSTEMS

The distinction between a traditional data platform hosted in the cloud and a genuinely cloud-native AI and data architecture is often misunderstood. Moving an on-premises warehouse, a set of scheduled extraction jobs, and a few trained models onto cloud virtual machines is sometimes described as a cloud migration, but it leaves the fundamental design untouched. The pipelines still run as tightly coupled batch chains, compute is still provisioned for peak load, models are still retrained by hand, and a failure anywhere in the chain still halts everything downstream. Such a platform inherits the billing model of the cloud without inheriting its benefits.

Cloud-native design begins from a different premise. It assumes that the infrastructure beneath a pipeline or model is transient, that individual components will fail and be replaced, and that the system as a whole must continue to produce correct data and reliable predictions even as its parts come and go. This premise reshapes nearly every design decision. Intermediate data cannot be assumed to live safely on a local disk, because the worker holding it may vanish. Pipelines must be idempotent and restartable, because partial runs are inevitable. Tight coupling between producers and consumers of data becomes dangerous, because a schema change in one system can silently corrupt features in another. Manual retraining and deployment become bottlenecks, because a model that must be refreshed as data changes cannot wait for human intervention.



The paradigm shift, then, is from treating data and models as static assets managed by a central platform to treating them as continuously flowing, versioned products that move through a dynamic and partially unreliable environment. This is less a technological change than a philosophical one, and it accounts for why lifting existing data platforms and models into the cloud so often fails to deliver the agility, scale, and model quality that motivated the move.

III. FOUNDATIONAL PRINCIPLES FOR AI AND DATA SYSTEMS

Several principles recur across cloud-native systems. Together they form the conceptual core of the approach. Each takes on a specific meaning when the system's primary outputs are processed data, features, and predictions rather than transactions alone.

3.1 Decomposition into Data and Model Services

Rather than building a single deployable unit, cloud-native systems are commonly decomposed into small, independently deployable services, each responsible for a distinct capability [4]. In AI and data systems, the natural units of decomposition include ingestion, transformation, feature computation, training, evaluation, model serving, and monitoring. This decomposition allows each stage to be developed, scaled, and released on its own schedule. An inference service under heavy load can be scaled without scaling the training cluster, and a defect in one transformation can, if the boundaries are well drawn, be contained rather than propagated into every downstream model.

Decomposition is not free. It trades the simplicity of a single pipeline for the complexity of a distributed system, and the quality of the boundaries largely determines whether that trade is worthwhile. Machine learning systems are especially prone to hidden coupling, in which a change to one input signal, feature, or upstream model silently alters the behavior of others [11]. Boundaries drawn around well-defined data products, with explicit ownership and published contracts, tend to hold up well [21]. Boundaries drawn arbitrarily produce services that must constantly coordinate, reproducing the entanglement of a monolithic pipeline while adding network latency and making failures harder to trace.

3.2 Containerization and Orchestration of ML Workloads

Containers package an application together with its dependencies into a portable, isolated unit that runs consistently across environments. For machine learning, this portability addresses a long-standing source of failure: the difference between the environment in which a model was trained and the one in which it is served, including library versions, numerical behavior, and hardware drivers. Orchestration platforms then manage large numbers of containers, handling placement, scaling, health checking, and the replacement of failed instances [3]. Data and AI workloads extend the demands on the orchestrator. Batch jobs must be scheduled alongside long-running services, distributed training requires groups of workers to start together, and accelerators must be allocated as scarce, often indivisible resources. Frameworks designed specifically for distributed AI applications build on the same disposable-instance assumption while adding the task and actor abstractions that training and reinforcement learning require [15].

3.3 Externalized State: Data, Features, and Model Artifacts

Because any given instance may be terminated at any time, cloud-native services are designed wherever possible to be stateless, with state that must persist externalized to durable stores. In AI and data systems, this principle applies to three kinds of state. Raw and processed data are externalized to object storage, data lakes, and warehouses. Features are externalized to feature stores that serve consistent values to both training and inference. Model weights, configurations, and evaluation results are externalized to model registries that track every version. A stateless processing worker or inference server can then be replicated freely, scaled without ceremony, and replaced without data loss. This principle is one of the sharpest departures from traditional practice, where intermediate data sets lived on local disks and trained models were copied manually between environments, and it is the foundation on which reproducibility, rollback, and auditability rest.

3.4 Loose Coupling, Data Contracts, and Model APIs

Cloud-native components interact through well-defined interfaces, typically network APIs, rather than through shared memory, shared databases, or direct code-level dependencies. In data systems, the equivalent of an API is the data contract: an explicit, versioned agreement about the schema, semantics, freshness, and quality of a data set that one team produces and others consume. Reading another team's tables directly, without such a contract, is the data equivalent of a shared database and produces the same fragility. Models, in turn, are best exposed through stable prediction interfaces that hide the underlying framework, version, and hardware, so that a new model can replace an old one without changes to its consumers [16]. Treating both data contracts and model interfaces as deliberately designed

and carefully versioned artifacts allows producers and consumers to evolve independently and prevents a change in one from cascading uncontrollably into others.

3.5 Automation, Elasticity, and the ML Lifecycle

The cloud offers the ability to acquire and release resources dynamically, but this ability is useful only if the system can respond to changing demand without manual effort. Cloud-native AI and data systems therefore lean heavily on automation across the full lifecycle: automated ingestion and validation of new data, automated feature computation, automated retraining when data or performance shifts, automated evaluation against quality thresholds, and automated, gradual deployment of new model versions. This practice, often described as MLOps, extends continuous delivery from code to data and models [22]. Elasticity, the capacity to grow and shrink with demand, allows such a system to absorb bursts of ingestion, large backfills, and spikes in prediction traffic without provisioning for peak load at all times.

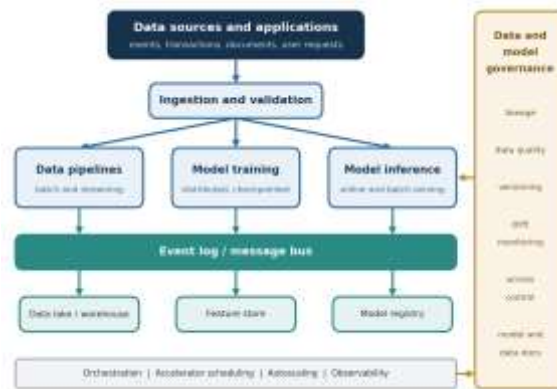


Figure 1. Simplified cloud-native architecture for AI and data systems, showing ingestion, processing, training and inference services, shared data stores, and cross-cutting governance.

IV. DESIGN PATTERNS FOR DISTRIBUTED DATA AND AI SYSTEMS

Principles describe the goals of cloud-native design. Patterns describe the concrete techniques that achieve them. A number of patterns have become standard vocabulary in distributed data and AI systems.

4.1 Data Management Patterns: Logs, Lakes, and Feature Stores

When a system is decomposed into services, the question of how to manage data becomes acute. A common pattern gives each service ownership of its own data, preventing the shared database that would otherwise couple services together. Data and AI systems add a second requirement: the same data must also be made available, reliably and at scale, for analytics and model training. A widely adopted response is to treat an ordered, durable event log as the backbone of the platform [9, 6]. Services publish changes as events, and downstream consumers, including analytical stores, feature pipelines, and training jobs, build their own views from that shared history. Because the log can be replayed, a new model or feature can be computed retroactively over historical data, and a faulty transformation can be corrected and rerun.

To coordinate such operations without a distributed transaction spanning all of them, systems often use a saga, which sequences a series of local transactions and defines compensating actions to undo earlier steps if a later one fails. Related patterns include separating the read and write paths of a system so that each can be optimized independently, and recording To coordinate operations that span several services without a distributed transaction, systems often use a saga, which sequences local transactions and defines compensating actions to undo earlier steps if a later one fails. Related patterns separate the read and write paths of a system so that each can be optimized independently. In AI systems, the most consequential data pattern is the feature store, which computes feature definitions once and serves them both to offline training, as point-in-time-correct historical snapshots, and to online inference, as low-latency current values. By sharing definitions across both paths, the feature store addresses the training and serving skew that otherwise arises when the same feature is implemented twice. Each of these techniques addresses the fundamental difficulty that, once data is distributed, the simple guarantees of a single database no longer apply.



4.2 Communication Patterns: Batch, Streaming, and Online Inference

Services must talk to one another, and the choice of how they communicate carries significant consequences. Data systems have historically relied on batch processing, in which large volumes of data are processed on a schedule [7]. Batch processing is simple to reason about and efficient for large jobs, but it delays insight and models by the length of the batch interval. Stream processing handles data continuously as it arrives, enabling fresh features and near-real-time predictions at the cost of added complexity in handling late, out-of-order, and duplicate events. Unified models of data processing treat batch as a special case of streaming over bounded data, allowing the same logic to serve both [10]. For model serving, synchronous online inference, in which a caller waits for a prediction, is appropriate when the result is needed immediately, while asynchronous or batch inference, in which predictions are computed ahead of time or delivered through a queue, decouples services in time and is often far cheaper.

As the number of services grows, managing cross-cutting concerns such as routing, retries, encryption, and observability becomes burdensome to implement in each service individually. A dedicated infrastructure layer that handles these concerns transparently allows the services themselves to remain focused on their logic. For model serving, this layer also supports traffic-splitting techniques that are central to safe deployment of AI: shadow deployment, in which a new model receives copies of live requests without affecting responses, canary release, in which a small share of traffic is routed to the new model, and online experiments that compare models on business outcomes rather than offline metrics alone.

4.3 Resilience Patterns for Pipelines and Model Serving

Because failure is expected rather than exceptional, cloud-native systems build resilience directly into their structure [5]. A widely used technique detects when a downstream dependency is failing and temporarily stops sending requests to it, allowing it to recover. Another isolates resources so that a failure consuming one pool does not starve the rest of the system. Retries with progressively increasing delays handle transient faults, and sensible timeouts prevent a caller from waiting indefinitely. Data and AI systems require additional forms of resilience. Pipelines must be idempotent, so that a retried step does not duplicate records, and checkpointed, so that a failed job resumes rather than restarts [8]. Incoming data must be validated against expected schemas and distributions before it reaches training or features, so that a malformed upstream change is quarantined rather than learned [12]. Model services need fallbacks, such as a simpler model, cached predictions, or a rule-based default, so that the failure of a complex model degrades the quality of a response rather than removing it. Applied together, these patterns allow a system to degrade gracefully under partial failure rather than collapsing entirely.

V. REASONING ABOUT DISTRIBUTED COMPLEXITY IN DATA AND AI SYSTEMS

Underlying all of these patterns is a body of theory about the limits of distributed systems. A foundational result establishes that a distributed data store cannot simultaneously guarantee consistency, availability, and tolerance of network partitions [2]. Because partitions are unavoidable in any real network, the practical consequence is a forced choice, during a partition, between remaining available and remaining strictly consistent. Cloud-native systems frequently accept a weaker form of consistency, in which replicas converge to the same state over time, in exchange for higher availability. Data and AI systems face a related set of choices along the dimension of time. Freshness, completeness, and cost pull against one another: waiting for late data improves completeness but delays results, while acting on data immediately improves freshness but risks decisions on partial information. Designing for these trade-offs explicitly, through watermarks, late-data handling, and clearly defined freshness guarantees, is a deliberate response to constraints that no amount of engineering can remove [10].

The distributed nature of these systems also makes understanding their behavior far harder than in a monolith. A single prediction may depend on data from many sources, transformed by several pipelines, combined in a feature store, and scored by a model trained weeks earlier on a different snapshot. Diagnosing a problem requires the ability to trace requests across services, aggregate logs, and monitor system metrics, but in AI and data systems this is not sufficient. Observability must extend to the data and the model: lineage that records where each data set and feature came from, data quality metrics that detect missing, delayed, or anomalous inputs, and model metrics that track prediction distributions and accuracy over time. A system can be fully healthy by infrastructure measures while producing systematically wrong results, which is why data and model observability must be treated as a design requirement rather than an operational luxury.



VI. AI WORKLOADS IN CLOUD-NATIVE SYSTEMS

The principles and patterns described so far were shaped largely by web-scale transactional systems. AI workloads share their distributed setting but stress it along different axes. A system that trains, serves, and continuously updates machine learning models does not merely run on a cloud-native platform; it tests the assumptions of state, consistency, and elasticity on which the platform rests. Three tensions are worth drawing out.

6.1 Training and Inference as Distinct Profiles

AI systems contain two workloads with almost opposite characteristics, and conflating them is a common design error. Training is a batch process: long-running, resource-hungry, and often coordinated across many accelerators that must make progress together, whether through data parallelism, in which each worker processes a different slice of data, or model parallelism, in which a model too large for one device is split across several [14]. A training job may run for hours or days, which makes the disposable-instance assumption expensive rather than convenient, since losing an instance late in a job can waste enormous work unless the job checkpoints its progress and can resume. Inference, by contrast, is request-driven and latency-sensitive. It resembles a conventional online service and benefits from the same autoscaling and resilience patterns, with important complications. A model's weights can be large enough that loading them into a freshly scheduled instance violates the latency budget, and large language models add further constraints, since the memory needed to hold intermediate attention state grows with the length of each request and determines how many requests can be served concurrently [17]. Serving systems therefore rely on techniques such as request batching, caching, and careful memory management that have no direct counterpart in ordinary web services [16, 17].

6.2 State, Consistency, and the Data Path

Statelessness is harder to preserve for AI services than for ordinary ones. A model server is nominally stateless, but the model it serves is a large, versioned artifact that must be present before the service can answer a single request, which pushes designers toward warm pools and cached weights rather than instances created purely on demand. Retrieval-augmented generation adds a further layer of state, since a generative model's answers depend on documents retrieved at query time from an index that must itself be kept current and consistent with its sources [18]. The data path introduces consistency concerns of its own. When the same features are computed one way for training and another way for serving, the resulting skew can quietly degrade model quality in production even though every individual component appears healthy. Feature stores and streaming pipelines inherit the eventual-consistency trade-offs discussed earlier, but here the cost of stale or inconsistent data is not a stale web page; it is a systematically worse prediction. Reasoning explicitly about point-in-time correctness, about which version of a model is paired with which version of its features, training data, and retrieval index, becomes part of the architecture rather than an afterthought.

6.3 Elasticity Under Scarce and Costly Resources

Elasticity is the property AI workloads strain most. The accelerators that training and large-model inference depend on are scarcer, costlier, and slower to provision than the commodity compute for which autoscaling was designed. Scaling to zero to save money collides directly with cold-start latency, because bringing an accelerator-backed replica online and loading a large model onto it is not instant. Provisioning for peak wastes expensive capacity; provisioning lean risks queuing or dropped requests when demand spikes. The elastic ideal, matching capacity precisely to demand moment by moment, is therefore only partially attainable, and architects must reconcile cost, latency, and availability with more care than a CPU-bound service would demand. Techniques such as sharing accelerators across smaller models, routing requests to models of different sizes according to difficulty, and scheduling training on interruptible capacity with frequent checkpoints all help, but each introduces its own complexity. Observability shifts accordingly. It is no longer enough to watch latency, errors, and saturation; the system must also observe the quality of its predictions, since a model can fail by growing steadily less accurate as the world drifts away from its training data, all while every infrastructure metric stays green [13].

VII. CHALLENGES AND TRADE-OFFS

The benefits of cloud-native architecture for AI and data systems are real, but so are its costs, and an honest assessment must weigh both.

The most immediate cost is complexity. A single training script and a nightly batch job are simple to build, test, and reason about. A distributed platform of ingestion services, streaming pipelines, feature stores, training clusters, model



registries, and serving layers introduces network communication, partial failure, distributed data, and operational overhead that simply do not exist in the simpler setup. It has been observed that in mature machine learning systems, the model code is often a small fraction of the whole, surrounded by far larger amounts of data collection, verification, configuration, serving, and monitoring infrastructure, and that much of the long-term maintenance burden lives there [11]. For a small team or an early-stage use case, this complexity may exceed any benefit, and a simpler, well-structured design may be the wiser choice. Cloud-native design earns its keep at scale, when the independence of components and the ability to scale them separately outweigh the coordination costs.

A second challenge is data consistency and quality. The distribution of data across independently owned stores, combined with the theoretical limits described above, means that engineers must reason explicitly about consistency, freshness, and completeness in ways that a single database previously handled for them. In AI systems, this challenge is sharpened because errors in data do not usually cause visible failures. A missing field, a changed unit, or a shifted distribution produces a model that still runs and still returns predictions, only worse ones. Systematic data validation, contracts between producers and consumers, and lineage that connects every prediction back to its sources are therefore essential rather than optional [12].

A third challenge is organizational. Cloud-native architecture is closely tied to the way teams are structured and work. Independent data and model services deliver their promised independence only if the teams that own them can also work independently, which requires investment in automation, in shared platform capabilities such as feature stores and model registries, and in a culture that treats data quality and model operations as part of development rather than a separate function. Approaches that treat data as a product owned by domain teams, supported by a self-service platform and federated governance, reflect this insight [21]. Adopting the architecture without the corresponding organizational changes tends to produce the costs of distribution without the benefits, with data scientists, data engineers, and platform engineers each owning a fragment of a system that none can see as a whole.

Finally, security and governance grow more intricate as the number of components, data sets, and network connections increases. Every service boundary is a potential point of attack, and every inter-service connection must be secured, authenticated, and authorized. AI and data systems add further concerns of their own. Training data must be protected against tampering and poisoning, sensitive and personal data must be controlled across many copies and derived features, and models must be protected as valuable assets that can leak information about the data on which they were trained. Documentation practices that record the provenance, composition, and intended use of data sets and the evaluated behavior and limitations of models support accountability and regulatory compliance [19, 20]. These obligations are easiest to meet when lineage, versioning, and access control are designed into the architecture from the start.

VIII. FUTURE DIRECTIONS

Several trends are shaping the continuing evolution of cloud-native design for AI and data. The convergence of data lakes and warehouses into unified platforms that support both analytics and machine learning on the same storage reduces copying and simplifies governance. The move toward finer-grained, serverless units of deployment for both data processing and inference pushes the cloud-native premise of disposable infrastructure further, while raising new challenges for large models with heavy startup costs. The demands of AI workloads are themselves driving specialized scheduling for accelerators, serving layers built for large language models, vector and retrieval infrastructure for generative applications, and data infrastructure designed for continuous training and evaluation. In the opposite direction, AI is increasingly applied to the operation of data systems themselves, from learned components that replace hand-tuned data structures [23] to intelligent tooling for capacity planning, anomaly detection, data quality monitoring, and self-healing. Each of these directions extends the same underlying idea: that the infrastructure should adapt to the needs of data and models automatically, leaving engineers and scientists free to concentrate on the data products and models that deliver value.

IX. CONCLUSION

Cloud-native architecture represents a genuine rethinking of software design rather than a new deployment target. Its principles, decomposition into independent services, disposable and externalized state, loose coupling, and pervasive automation, follow directly from taking the properties of the cloud seriously. Its patterns Cloud-native architecture represents a genuine rethinking of how AI and data systems are designed rather than a new deployment target for existing pipelines and models. Its principles, namely decomposition into independent data and model services,



externalized and versioned state, loose coupling through data contracts and model interfaces, and pervasive automation across the machine learning lifecycle, follow directly from taking the properties of the cloud seriously. Its patterns provide tested solutions to the recurring problems of distributed data, communication, and failure, from event logs and feature stores to streaming pipelines, safe model deployment, and graceful degradation of predictions. Its challenges, namely complexity, data consistency and quality, organization, and security and governance, are not incidental but intrinsic, and they define the conditions under which the approach is worthwhile. AI workloads make the case more pressing still, since training, inference, and continuous data pipelines press hardest on exactly the assumptions about state, consistency, and elasticity that cloud-native design exists to address. The essential insight is that designing AI for the cloud means designing for distribution, for failure, and for continuously changing data as ordinary conditions of operation. Systems built on that insight can achieve a level of scale, resilience, and sustained model quality that traditional designs cannot, but only when data, features, and models are treated as first-class architectural concerns rather than artifacts deployed on top of infrastructure.

REFERENCES

The discussion above draws on well-established work in distributed systems, cloud computing, data engineering, and machine learning systems. Readers seeking primary sources may consult the following foundational works.

1. Mell, P. and Grance, T. The NIST Definition of Cloud Computing. NIST Special Publication 800-145, 2011.
2. Gilbert, S. and Lynch, N. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." ACM SIGACT News, 2002.
3. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. "Borg, Omega, and Kubernetes." ACM Queue, 2016.
4. Newman, S. Building Microservices. O'Reilly Media.
5. Nygard, M. Release It! Design and Deploy Production-Ready Software. Pragmatic Bookshelf.
6. Kleppmann, M. Designing Data-Intensive Applications. O'Reilly Media, 2017.
7. Dean, J. and Ghemawat, S. "MapReduce: Simplified Data Processing on Large Clusters." Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2004.
8. Zaharia, M., et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2012.
9. Kreps, J., Narkhede, N., and Rao, J. "Kafka: A Distributed Messaging System for Log Processing." Proceedings of the NetDB Workshop, 2011.
10. Akidau, T., et al. "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, 2015.
11. Sculley, D., et al. "Hidden Technical Debt in Machine Learning Systems." Advances in Neural Information Processing Systems (NeurIPS), 2015.
12. Breck, E., Polyzotis, N., Roy, S., Whang, S. E., and Zinkevich, M. "Data Validation for Machine Learning." Proceedings of Machine Learning and Systems (MLSys), 2019.
13. Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., and Bouchachia, A. "A Survey on Concept Drift Adaptation." ACM Computing Surveys, 2014.
14. Dean, J., et al. "Large Scale Distributed Deep Networks." Advances in Neural Information Processing Systems (NeurIPS), 2012.
15. Moritz, P., et al. "Ray: A Distributed Framework for Emerging AI Applications." Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2018.
16. Crankshaw, D., et al. "Clipper: A Low-Latency Online Prediction Serving System." Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2017.
17. Kwon, W., et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention." Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2023.
18. Lewis, P., et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." Advances in Neural Information Processing Systems (NeurIPS), 2020.
19. Gebru, T., et al. "Datasheets for Datasets." Communications of the ACM, 2021.
20. Mitchell, M., et al. "Model Cards for Model Reporting." Proceedings of the Conference on Fairness, Accountability, and Transparency (FAT*), 2019.
21. Dehghani, Z. Data Mesh: Delivering Data-Driven Value at Scale. O'Reilly Media, 2022.
22. Huyen, C. Designing Machine Learning Systems. O'Reilly Media, 2022.
23. Kraska, T., Beutel, A., Chi, E. H., Dean, J., and Polyzotis, N. "The Case for Learned Index Structures." Proceedings of the ACM SIGMOD International Conference on Management of Data, 2018.