PREPRINT / TECHNICAL PAPER

Data Engineering First Principles

A Constraint-Driven Framework for Designing Reliable, Scalable, and AI-Native Data Systems

Eugene Ezenwa Ebem

Co-Founder & Chief Technology Officer, Tagus Technologies LLC

Dallas-Fort Worth, Texas, USA

ORCID: 0009-0005-8470-4922

Version 1.0 | September 2026

DOI: 10.5281/zenodo.22691903

Core thesis: Tools change. Principles don't.

Scholarly integrity note: This paper proposes a conceptual architecture and decision framework. The case study is an illustrative composite intended to demonstrate the method; it does not claim new benchmark results or disclose confidential client systems.

Abstract

Data engineering is often taught through products, frameworks, and platform-specific recipes. That approach can accelerate short-term implementation while weakening architectural judgment when tools, workloads, economics, or organizational constraints change. This paper presents Data Engineering First Principles (DEFP), a constraint-driven framework for reasoning about data systems before selecting technologies. DEFP models an engineering decision as a progression from requirements to measurable constraints, explicit trade-offs, architecture patterns, and only then technology selection. The framework organizes design around six recurring constraint families: data shape, scale, latency, reliability, economics, and governance/security. It then extends those constraints to batch and streaming systems, cloud-native storage and compute, warehouse/lake/lakehouse choices, relational and non-relational databases, and AI-native pipelines involving embeddings, vector retrieval, retrieval-augmented generation, and model feedback loops. A reusable Architecture Trade-off Matrix and a composite enterprise pipeline case study demonstrate how the method can expose hidden assumptions, prevent tool-first architecture, and improve explainability of design decisions. The contribution is not a new data platform; it is a durable reasoning model intended to help engineers move from task execution toward system design.

Keywords: data engineering; first principles; systems thinking; data architecture; cloud data platforms; streaming; DataOps; lakehouse; vector databases; RAG; AI-native data systems

1. Introduction

Modern data engineering sits at the intersection of distributed systems, databases, cloud infrastructure, analytics, machine learning, governance, and software delivery. The ecosystem is productive precisely because it offers many choices, but the same abundance creates a recurring failure mode: architecture begins with a favored tool rather than with the problem.

A team may begin with a streaming platform because 'real time' sounds desirable, adopt a lakehouse because it is fashionable, or add a vector database because an application uses a large language model. Each technology can be appropriate, but none is a first principle. The engineering question is whether the workload's constraints justify the operational, financial, and organizational consequences of that choice.

This paper argues for reversing the sequence. Architecture should emerge from requirements and constraints. Technology should implement the resulting architecture, not define it. This distinction becomes more important as AI-assisted development makes implementation faster: when code generation becomes cheaper, judgment about data semantics, failure modes, reliability, cost, privacy, and system boundaries becomes more valuable.

1.1 Contributions

- A constraint-driven reasoning sequence: Requirements -> Constraints -> Trade-offs -> Architecture -> Technology Selection.
- Six reusable constraint families for data-system design: data shape, scale, latency, reliability, economics, and governance/security.
- An Architecture Trade-off Matrix for comparing candidate patterns without prematurely committing to products.
- A unified application of the framework to batch, streaming, cloud, lakehouse, DataOps, and AI-native/RAG workloads.

- An illustrative enterprise case study showing how first-principles reasoning can diagnose a fragile pipeline and guide redesign.

2. Background and Positioning

Existing literature provides strong foundations for data-intensive system design. Kleppmann emphasizes the trade-offs among reliability, scalability, maintainability, storage, replication, partitioning, and distributed processing. Akidau and colleagues develop rigorous mental models for event time, processing time, windows, watermarks, and correctness in streaming systems. Reis and Housley frame data engineering through a lifecycle that connects generation, storage, ingestion, transformation, and serving. The lakehouse literature argues for unifying data-lake flexibility with warehouse-style management and performance. These works make clear that data architecture is fundamentally a trade-off discipline.

DEFP complements these perspectives by concentrating on the decision process itself. Its goal is to make senior-engineering reasoning explicit and teachable: what must be true before a design is selected, which constraints dominate, which trade-offs are accepted, and how the choice can be explained to technical and non-technical stakeholders.

3. The First-Principles Reasoning Model

A first-principles approach decomposes a design problem into constraints that remain meaningful even when products change. For data engineering, the key move is to delay product selection until the workload has been characterized.

Stage	Primary question	Typical output
1. Requirements	What outcome must the system enable?	Business/technical service objective
2. Constraints	What limits or conditions are non-negotiable?	Latency, scale, quality, cost, compliance bounds
3. Trade-offs	What are we willing to optimize or sacrifice?	Explicit decision record
4. Architecture	Which system pattern best satisfies the constraints?	Batch/streaming, storage, compute, serving pattern
5. Technology	Which products implement the chosen pattern well?	Tool/platform selection
6. Operations	How will we know the system is healthy?	SLOs, observability, tests, incident and cost controls

The order matters. Moving directly from requirements to technology collapses design into product familiarity. Moving through constraints and trade-offs creates an auditable explanation for why the architecture exists.

4. Six Constraint Families

4.1 Data Shape

Structure, mutability, ordering, schema evolution, cardinality, event semantics, file/object size, and whether the workload includes text, images, embeddings, or other multimodal representations.

4.2 Scale

Volume, throughput, concurrency, growth rate, skew, partitionability, and the difference between average and peak load.

4.3 Latency

How quickly data must become usable. Latency should be tied to a decision or user outcome rather than to a vague desire for 'real time.'

4.4 Reliability

Durability, availability, replayability, idempotency, correctness, recovery objectives, lineage, and tolerance for duplicates or late data.

4.5 Economics

Infrastructure cost, engineering labor, operational burden, data-transfer cost, storage tiers, compute elasticity, and the cost of failure.

4.6 Governance and Security

Data classification, privacy, retention, access control, sovereignty, auditability, encryption, model/data lineage, and regulatory obligations.

5. Architecture Trade-off Matrix

The Architecture Trade-off Matrix converts qualitative reasoning into a structured comparison. Scores are not universal truths; they are workload-specific judgments. A useful matrix therefore records both a score and the rationale behind it.

Constraint	Weight	Batch-first	Streaming-first	Hybrid
Freshness < 5 min	High	Low	High	High
Replay simplicity	High	High	Medium	High
Operational simplicity	Medium	High	Low-Med	Medium
Peak throughput	High	Medium	High	High
Cost predictability	Medium	High	Medium	Medium
Late/out-of-order events	High	Medium	High	High
Team maturity	High	High	Low-Med	Medium

A weighted matrix does not replace engineering judgment. It makes assumptions visible. If the business requirement changes from hourly reporting to sub-minute fraud intervention, the weights change and the preferred architecture may change with them.

6. Batch vs. Streaming Is a Constraint Decision

Batch and streaming are often presented as generations of technology, as if streaming were inherently more advanced. A first-principles view treats them as different responses to latency, ordering, cost, correctness, and operational constraints.

Question	Batch tendency	Streaming tendency
When must the result influence a decision?	Minutes to days	Milliseconds to minutes
Can data be recomputed economically?	Usually favorable	May require state/replay strategy
How important is event ordering?	Often simpler	Explicit event-time semantics may be required
Operational complexity tolerance	Lower	Higher
Burst handling	Scheduled/elastic batches	Continuous backpressure and scaling
Failure recovery	Rerun partitions/jobs	Replay checkpoints/state/log

The correct architecture can also be hybrid: continuous ingestion with micro-batch transformation, streaming detection with batch reconciliation, or batch backfills feeding the same serving model as real-time events.

7. Storage and Database Selection from First Principles

Storage selection should begin with access patterns and consistency requirements. Relational systems excel when structured relationships, transactions, constraints, and mature query semantics dominate. Document, key-value, wide-column, and graph systems trade different forms of consistency, query flexibility, distribution, and data modeling. Object storage changes the economics of large analytical datasets by decoupling durable storage from compute.

Workload property	Likely design implication
Strong multi-row transactional invariants	Relational/transactional engine
Massive immutable analytical history	Object storage + analytical table format
Key-based low-latency access at scale	Key-value / distributed NoSQL
Highly connected relationship traversal	Graph-oriented model may be appropriate
Semantic similarity retrieval	Vector index integrated with source-of-truth storage
Frequent schema evolution	Flexible ingestion plus governed schema-on-read/write boundary

8. Warehouse, Lake, and Lakehouse

The warehouse/lake/lakehouse discussion is best understood as a set of trade-offs rather than labels. Warehouses historically prioritize managed schemas, SQL performance, governance, and curated analytical access. Data lakes prioritize inexpensive, open-ended storage and broad data types but can suffer from weak metadata and governance if unmanaged. Lakehouse architectures seek to combine

low-cost object storage and open formats with transactional metadata, governance, and high-performance analytical access.

DEFP asks four questions before selecting a pattern: Who consumes the data? What consistency and governance guarantees are required? What is the cost profile of storage and compute? How portable must the data remain? These questions usually matter more than the platform label.

9. Cloud-Native Mental Models

Cloud migration is not equivalent to cloud-native design. Recreating fixed on-premises clusters inside virtual machines preserves capacity assumptions while changing the billing model. Cloud-native data engineering instead reasons about elasticity, managed services, failure domains, decoupled storage and compute, ephemeral workers, infrastructure automation, and usage-based economics.

- Design for replaceable compute and durable externalized state.
- Scale independently where architecture permits rather than scaling entire stacks together.
- Treat network movement and cross-region transfer as architectural costs.
- Use managed services when reduced operational burden outweighs portability or control concerns.
- Make cost observable alongside latency, throughput, and reliability.

10. Reliability, DataOps, and Observability

A pipeline is not production-grade merely because it completes successfully. Reliability includes whether the output is correct, fresh, complete, explainable, reproducible, and recoverable. DataOps extends software delivery disciplines into data systems through version control, automated testing, CI/CD, environment management, data contracts, lineage, and incident response.

Layer	Example control	Failure detected
Source	Schema/data contract	Breaking producer change
Ingestion	Volume and lag monitors	Dropped or delayed data
Transformation	Unit/integration/data-quality tests	Logic or quality regression
Storage	Partition/table health checks	Corruption, missing partitions, skew
Serving	Freshness and query SLOs	Stale or slow consumer experience
Business semantics	Reconciliation checks	Technically valid but incorrect outcomes

Observability should connect technical telemetry to business semantics. A green cluster with a stale executive metric is not a healthy data product.

11. Cost Engineering as Architecture

Cost is not an after-the-fact optimization. It is an architectural constraint. A design that meets latency and reliability targets but cannot be operated economically is not sustainable. Cost engineering includes storage lifecycle policies, partition design, compute sizing, autoscaling, query efficiency, data retention, transfer costs, and the engineering labor required to operate the system.

A useful design review therefore asks not only 'Can this scale?' but 'What is the marginal cost of scaling, and which component becomes economically dominant first?' This shifts FinOps from reporting into engineering.

12. AI-Native Data Engineering

AI systems increase rather than eliminate the need for disciplined data engineering. Training, fine-tuning, retrieval, evaluation, and inference all depend on governed data flows. Embeddings introduce a derived representation that must be linked to source content, model/version metadata, chunking strategy, permissions, and refresh logic.

12.1 Retrieval-Augmented Generation as a Data Pipeline

A RAG system can be reasoned about as a specialized data product: source acquisition -> parsing -> chunking -> enrichment -> embedding -> indexing -> retrieval -> prompt assembly -> generation -> evaluation/feedback. Each stage has familiar data-engineering concerns such as lineage, idempotency, freshness, schema evolution, access control, and observability.

RAG stage	First-principles question
Ingestion	Which sources are authoritative and how quickly must changes propagate?
Chunking	What unit preserves semantic meaning for the target query?
Embedding	Which model/version produced each vector and when must it be refreshed?
Vector storage	What recall, latency, filtering, scale, and durability are required?
Retrieval	How will relevance and permission constraints be evaluated?
Generation	How are retrieved evidence, prompts, and model outputs traced?
Feedback	How are failures converted into measurable data-quality or retrieval improvements?

12.2 Vector Databases Are Not the Source of Truth

Vector indexes are optimized for similarity search, but semantic retrieval does not remove the need for authoritative source data, transactional metadata, access controls, and reproducible transformation. A robust design treats vectors as derived, versioned artifacts whose provenance can be reconstructed.

13. Illustrative Composite Case Study: Repairing a Fragile Enterprise Pipeline

This case study is deliberately composite and illustrative. It demonstrates the DEFP method without asserting that the described architecture, metrics, or outcomes belong to a specific employer or client.

13.1 Initial State

Assume an enterprise pipeline ingests operational events from multiple producers, performs transformations, and publishes analytical tables used by risk and operations teams. The system has

accumulated symptoms: late data, expensive reruns, inconsistent schemas, poor lineage, duplicated transformation logic, and pressure to make selected decisions faster.

13.2 Tool-First Response

A tool-first response might propose replacing the orchestration platform, introducing streaming everywhere, or migrating all workloads to a new managed platform. Each action could help, but none establishes which problem dominates.

13.3 Constraint Decomposition

Constraint	Illustrative finding	Design consequence
Data shape	Mixed event JSON + relational reference data	Separate ingestion contracts; normalize at governed boundary
Scale	High peaks, moderate daily average	Elastic ingestion/compute; avoid sizing for peak everywhere
Latency	Only a subset needs <5 min freshness	Stream critical path; batch noncritical enrichment
Reliability	Reruns create duplicates	Idempotent writes, checkpoints, replayable raw layer
Economics	Full-history recomputation is expensive	Incremental processing + tiered storage
Governance	Sensitive fields cross multiple zones	Classification, least privilege, lineage, policy enforcement

13.4 Resulting Architecture Pattern

The resulting design is hybrid rather than universally streaming. A durable raw layer preserves replayability. Critical events follow a low-latency path with explicit event-time and idempotency semantics. Noncritical transformations remain scheduled or incremental. Curated data products expose governed schemas, while observability measures freshness, completeness, lag, cost, and reconciliation.

13.5 Why the Method Matters

The value is not that DEFP always chooses a hybrid architecture. The value is that every major component can be traced to a constraint. If latency, cost, regulation, or scale changes, the team can revisit the affected decision rather than redesigning by fashion.

14. From Tool Executor to System Designer

First-principles data engineering is also a professional maturity model. Early-career engineers naturally focus on syntax and task completion. More senior engineers increasingly reason about interfaces, failure modes, trade-offs, organizational constraints, and consequences across the system.

Maturity	Dominant question	Typical behavior
Tool executor	How do I implement this?	Follows known recipes
Pipeline engineer	How do I make the flow work reliably?	Owns end-to-end execution
System designer	Which architecture best fits the constraints?	Compares patterns and trade-offs
Architect/technical lead	How does this decision affect the	Balances technical, economic, security,

	broader platform and organization?	and product outcomes

AI-assisted coding accelerates the lower layers of this progression but does not remove the upper layers. Faster code generation increases the leverage of engineers who can define the right problem, inspect generated solutions critically, and reason about system-wide consequences.

15. Limitations and Future Work

DEFP is a conceptual framework, not an empirically validated scoring model. Constraint weights are context-dependent, and different teams may reasonably score the same architecture differently. The framework also does not replace specialized methods for capacity planning, formal security analysis, distributed-systems verification, or financial modeling.

Future work should evaluate the framework through reproducible architecture exercises and practitioner studies. Useful research questions include whether explicit constraint matrices improve design consistency, whether they reduce unnecessary platform complexity, and whether they help less-experienced engineers explain architectural decisions more effectively. A companion open-source template could standardize decision records and make comparisons across case studies possible.

16. Conclusion

Data engineering will continue to change rapidly at the product layer. New storage engines, orchestration systems, table formats, streaming runtimes, vector stores, and AI frameworks will appear. The durable skill is not memorizing the current catalog; it is reasoning from the properties of the problem.

Data Engineering First Principles formalizes that reasoning as a sequence: requirements, constraints, trade-offs, architecture, technology, and operations. By organizing decisions around data shape, scale, latency, reliability, economics, and governance/security, engineers can build systems whose rationale survives changes in tooling. The framework is intended to make architectural judgment explicit, teachable, and reusable across traditional analytics and AI-native systems.

Tools change. Principles don't.

References

Akidau, T., Chernyak, S., & Lax, R. (2018). Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing. O'Reilly Media.

Armbrust, M., Ghodsi, A., Xin, R., & Zaharia, M. (2021). Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. CIDR 2021.

Dehghani, Z. (2022). Data Mesh: Delivering Data-Driven Value at Scale. O'Reilly Media.

Kleppmann, M. (2017). Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media.

Mell, P., & Grance, T. (2011). The NIST Definition of Cloud Computing. NIST Special Publication 800-145.

Reis, J., & Housley, M. (2022). Fundamentals of Data Engineering: Plan and Build Robust Data Systems. O'Reilly Media.

Stonebraker, M., & Çetintemel, U. (2005). One Size Fits All: An Idea Whose Time Has Come and Gone. Proceedings of ICDE.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster Computing with Working Sets. Proceedings of HotCloud.

Author Note

Eugene Ezenwa Ebem is a data engineering and architecture practitioner with experience across batch, streaming, cloud, governance, and AI-oriented data systems. This paper develops concepts originating in his Data Engineering First Principles book proposal and subsequent technical writing. The present work is an independent technical preprint.