DATA PROTECTION & SOVEREIGNTY

# Analytics Without the Warehouse

A warehouse fuses storage and compute into one coupled system. Pulse decouples them into two separate concerns — so your analytics scale with your data, not your planning.

August 2026 · v1.0 · 17 min read

01 The thesis: decoupling storage from compute
02 The mechanism: compute that queries without holding
03 Layer zero: storage is the object store, never the engine
04 Keeping the query small: pruning, then rollups
05 Scaling the compute out: sharding, then spilling
06 Sovereign by inheritance, not by re-implementation
07 Analyse the data without exposing who it's about
08 Bring, or upload, any data — into your own storage
09 Composable by design: bridging into the stack a team already runs

## Executive summary

Pulse is Kilimanjaro's Data Intelligence plane for operational and agentic systems — four products for working with data: Lakehouse and Stream for storage, Compute and Feed for activity. This paper is about two of them. Lakehouse holds data as object storage; Compute is a stateless engine that reads it, computes an answer, and retains nothing. Where that boundary between the two is drawn is the paper's whole argument — both how Pulse scales and how it stays sovereign per tenant follow directly from it.

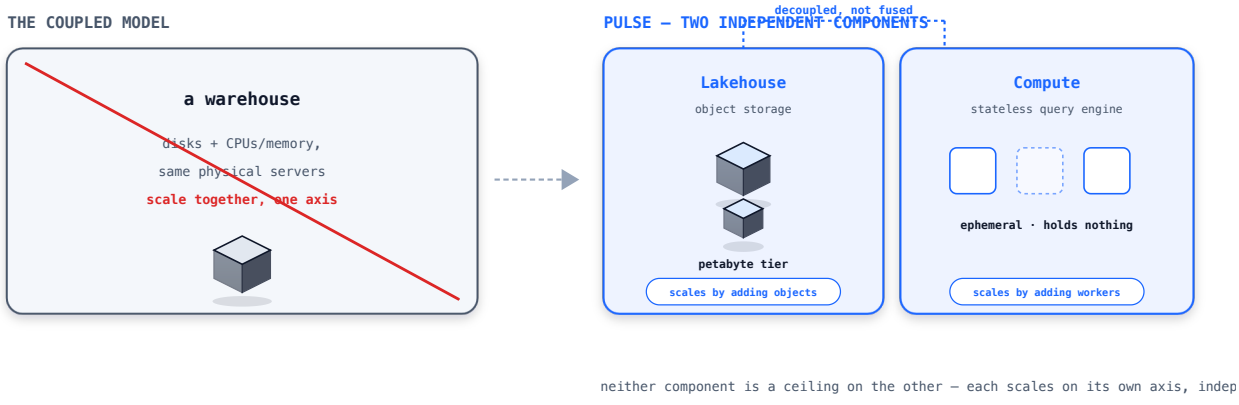
For scale, the consequence is that a query's cost tracks what it touches, not what exists — each stateless worker prunes to the slice a query actually touches, so a petabyte dataset is served without any single worker ever having to hold it. For governance, the same statelessness means compute inherits Kilimanjaro's per-tenant isolation and region-pinning by construction rather than by policy, and the columns that identify a person stay sealed under a per-subject key while the rest of the dataset is queried in the clear.

The paper works through the mechanics that keep a query small as data grows — pruning, rollups, and fleet sharding — and closes by showing how Pulse composes with the warehouse a team already runs, for the narrow class of workload that calls for one.

## 01 The thesis: decoupling storage from compute

A traditional data warehouse fuses storage and compute inside the same physical servers: disks and the CPUs/memory that query them scale together, provisioned and running as one system whether or not a given moment's workload actually needs both. Pulse decouples them into two of the four products that make up its Data Intelligence plane. The Lakehouse is a storage layer — Hive-partitioned Parquet on object storage. Compute is a separate layer — a stateless query engine that reads from it and holds nothing between queries. (Pulse's other two products, Stream and Feed, are outside this paper's scope.)

This is not a packaging choice — it is the architectural fact the rest of the paper rests on. Because storage is an object store and compute is stateless, the two scale on independent axes: storage grows by adding cheap objects, compute grows by adding stateless workers, and neither is a ceiling on the other. Snowflake and Databricks separate storage and compute internally too; the difference here is that Pulse runs compute as an ephemeral, per-tenant engine over the storage layer rather than a shared, always-on cluster provisioned ahead of demand. Everything that follows — how queries stay small as data grows, how sovereignty holds per tenant, and how Pulse composes with the tools already in a data team's stack — is a consequence of that decoupling.

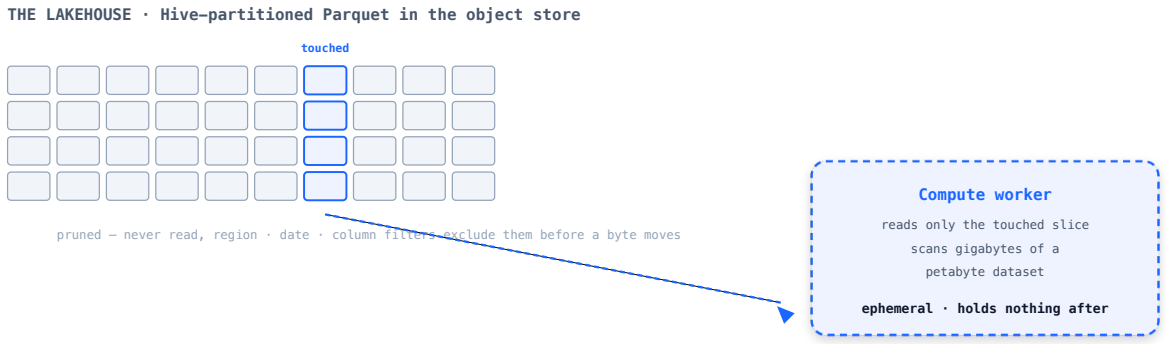



A traditional warehouse couples storage and compute inside one system — the two scale together on the same servers, whether or not a given workload needs both. Pulse decouples them into two independently-scaling components: the Lakehouse (object storage, the petabyte tier) and Compute (a stateless engine that reads it) — neither is a ceiling on the other.

## 02 The mechanism: compute that queries without holding

DuckDB runs in-process and single-node by design — that simplicity is exactly why it fits this architecture. In Pulse, a query acquires a worker from the per-project, per-region pool, reads Parquet from the Lakehouse through a partition-pruned scan, computes its answer, and releases the worker — holding no state between queries. The dataset itself never sits inside the engine; it sits in the object store, the tier built to hold arbitrarily large data cheaply, and the engine simply visits it.

The consequence is direct: dataset size is not bounded by any node's memory. A petabyte Parquet dataset lives in the object store; a query against it materialises only what it touches. So the design question Pulse optimises for is never "how big is the dataset" but "how small can the engine keep the working set" — and four layers answer that, each shrinking what a query actually reads below the layer before it.




Most of the object store, for any given query, is pruned before a byte moves — region, date, and column filters exclude it up front. A stateless, ephemeral Compute worker reads only the touched slice, computes an answer, and holds nothing after. A query's working set is the data it touches, not the size of the table it lives in.

## 03 Layer zero: storage is the object store, never the engine

The foundation is that the petabyte tier is the object store, not the engine. Datasets are Hive-partitioned Parquet in object storage; the Compute engine reads them with a partition-scoped scan and keeps nothing between queries. The object store is unbounded and cheap; the engine is ephemeral and stateless. Because a query's working set is the data it touches, not the size of the dataset it lives in, the dataset's total size never has to fit anywhere at once.

Four layers build on that foundation, each a different way of keeping "the data a query touches" smaller than "the data that exists": prune it before reading, pre-aggregate it so the hot path never reads raw, shard it across workers when it is genuinely large, and spill it to disk when a single worker's slice still exceeds memory. The table above is the map; the next two sections walk it.

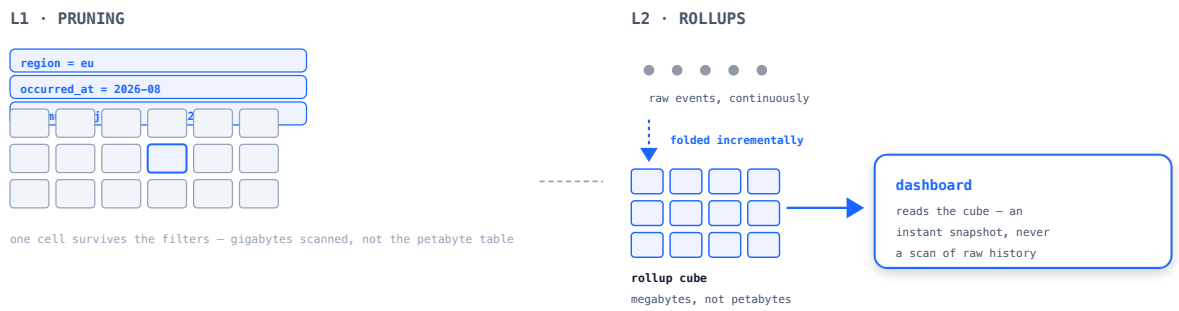
| Layer | Mechanism | What it bounds |
|---|---|---|
| L0 · Object-store storage | Hive-partitioned Parquet in object storage; the engine reads it, never holds it | Working set = data touched, not dataset size |
| L1 · Partition pruning + columnar pushdown | Read only the touched partitions, the projected columns, the row-groups whose statistics pass the predicate | A selective query scans gigabytes of a petabyte dataset |
| L2 · Rollups | Continuously-maintained materialised aggregates; the dashboard path reads cubes, not raw events | Petabyte of raw → gigabytes of aggregate; interactive reads never scan history |
| L3 · Scatter-gather | Shard a large multi-partition query across the stateless worker pool; a coordinator merges mergeable partials | Compute scales with worker count; each worker's working set stays bounded |
| L4 · Out-of-core | Larger-than-memory joins, sorts, and aggregations spill to the worker's disk | A single worker's ceiling is disk and throughput, not RAM |

The four working-set-reducing layers over a stateless compute fleet. Each layer shrinks what a query actually reads, below the layer before it; each layer is described in the order a query meets it.

## 04 Keeping the query small: pruning, then rollups

The dominant case never materialises the dataset at all. A query filtered to one region, a time window, and a handful of columns reads only the Hive partitions those filters select, only the columns the query projects, and only the Parquet row-groups whose zone-map statistics pass the predicate — everything else on disk is skipped before a byte is read. A selective query against a petabyte dataset therefore scans gigabytes, and a per-query scanned-bytes and partitions ceiling makes that a hard bound rather than a hope. This is L1, and it is built into the query executor: partition pruning plus columnar and row-group pushdown is the difference between "the query is the size of the answer" and "the query is the size of the table."

The interactive and dashboard path goes further and reads pre-aggregated rollups instead of raw events. A Rollup is a continuously-maintained materialised aggregate — a petabyte of raw events maintained, incrementally, as gigabytes of aggregate cubes off the event stream. A dashboard's "instant snapshot" reads a cube; it never scans history, so most high-frequency read load never touches the raw tier. This is L2, and it is what keeps the working set small for the queries a business actually runs all day, as opposed to the occasional full-history sweep.

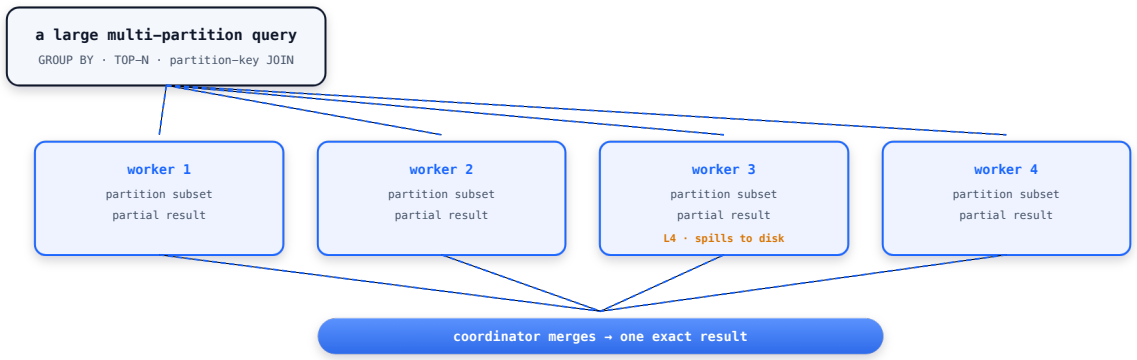



Pruning narrows a Hive-partitioned grid, by region, time window, and projected columns, down to the one touched slice — gigabytes scanned, not the petabyte table. Separately, a stream of raw events folds continuously into a rollup cube; the dashboard reads that cube as an instant snapshot and never scans raw history.

## 05 Scaling the compute out: sharding, then spilling

For a query that genuinely spans many partitions, the work is sharded across the stateless worker pool: N workers each own a partition subset, compute partial results, and a coordinator merges them. This is MPP-lite, and it is exact for the operations that dominate analytics — filters, GROUP BY, aggregations, TOP-N, distinct-count approximations, and joins on the partition key are all associative and mergeable, so partial-then-merge returns the same answer a single pass would. Compute scales with worker count while each worker's working set stays bounded regardless of total dataset size. This is L3, the architecture that shards the engine for multi-node scale — an orchestration the platform builds over the fleet, delivered as a growth-phase capability as query shapes cross the single-worker threshold.

Below the shard level, a single worker still executes larger-than-memory: hash joins, sorts, and aggregations spill to the worker's temporary disk rather than failing. So a worker's real ceiling is disk and throughput, not RAM — and L3 scales that horizontally by adding workers. Read the four layers together and the shape is clear: L0 removes the dataset from the engine, L1 and L2 shrink what a query reads, L3 spreads what remains across a fleet, and L4 keeps any single worker from being memory-bound. Across all four, the dataset never has to fit in a single node.

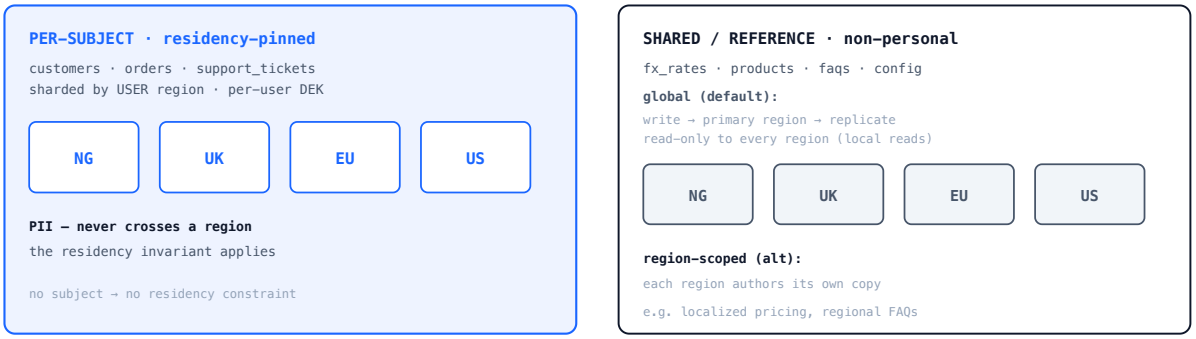



A large multi-partition query fans out to N stateless workers, each owning a partition subset and computing a partial result; a coordinator merges the mergeable partials into one exact answer. If any single worker's slice still exceeds memory, that worker spills to disk rather than failing.

## 06 Sovereign by inheritance, not by re-implementation

A stateless per-tenant engine inherits the substrate's sovereignty cleanly. Sovereignty in Kilimanjaro is a substrate property — per-tenant isolation, regional sharding, row-level security, policy-based access — inherited by every product by default, not a feature each product re-invents. Compute inherits it by construction: a stateless worker runs per project and per region, cred-fenced to exactly that project and region's storage prefix by a read-only, region-scoped credential, with nothing pooled across tenants. Region-pinning is a physical fence, not app logic — the credential a worker holds cannot read another region's bytes, so residency is enforced by what the key can reach, not by a check the code remembers to run.

Access control follows the same inheritance. Each dataset is a policy resource in its own right: fine-grained, per-dataset access — default-deny, deny-wins, with attribute predicates evaluated by the policy ceiling before the query runs, refusing the whole query if any referenced dataset is denied — is built on the substrate's policy model rather than bolted on. Two properties of that isolation are worth calling out directly. There is no cross-tenant data-sharing surface at all — isolation by construction means there is nothing to leak through, a deliberate contrast to platforms built around sharing across orgs. And per-subject row-filtering is fenced by the query surface guard, the database's own row security, and the set-role guard together — defense in depth, not a single role trusted to hold the line.

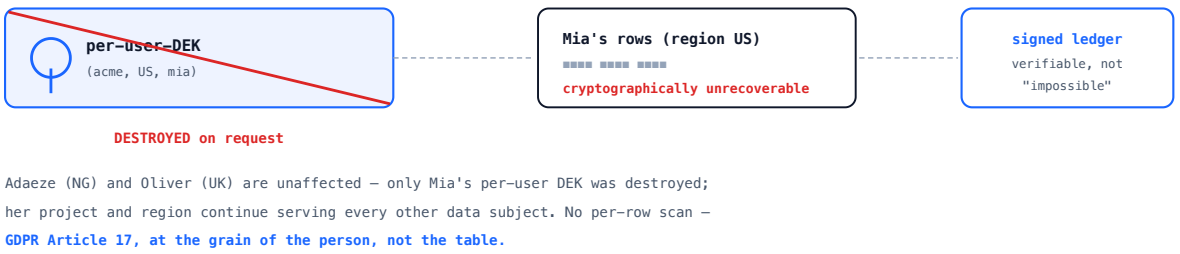



A dataset declares its residency class. Per-subject data is region-pinned and sharded by the subject's region; shared reference data is non-personal. The engine inherits this from the substrate — a stateless worker per project and region, fenced to that project and region's storage prefix — rather than re-implementing isolation of its own.

## 07 Analyse the data without exposing who it’s about

The flagship governance property of Pulse analytics is that you can analyse over a dataset while the identifying columns stay cryptographically sealed. Two design decisions compose it. First, encrypted columns stay encrypted in analytics by design: the operator reveal gate that can unseal a protected field is a collections-only mechanism and is deliberately not extended to datasets, so no analyst path in the analytics engine unseals a sealed column. Second, the identifying ("linkage") columns are sealed as ciphertext with AES-256-GCM-SIV — an authenticated cipher that is nonce-misuse-resistant, so a mishandled nonce degrades gracefully instead of catastrophically leaking, which is a stronger property than plain GCM offers, not a caveat on it.

The key structure is what makes the confidentiality per-subject rather than per-table. Each subject’s identifying data is sealed under that subject’s own 256-bit field key; a domain-separated HKDF-SHA-256 derivation produces a distinct per-cell subkey, so no working key is reused across cells. The cipher’s associated data binds the full identity of the cell — project, region, collection, column, and subject — so a sealed cell is cryptographically bound to its exact location and cannot be relocated across columns, collections, subjects, or tenants; a moved cell fails its tag. Every failure mode resolves to the same safe state: a missing key, a tag or identity mismatch, or malformed input returns null — never plaintext, never ciphertext, never a catchable error. On the analytics route, that fail-closed null is the state the identifying columns resolve to; the per-subject in-view decrypt is the built mechanism, exercised only for an authorised subject-key holder, and the analytics engine has no operator reveal path at all.



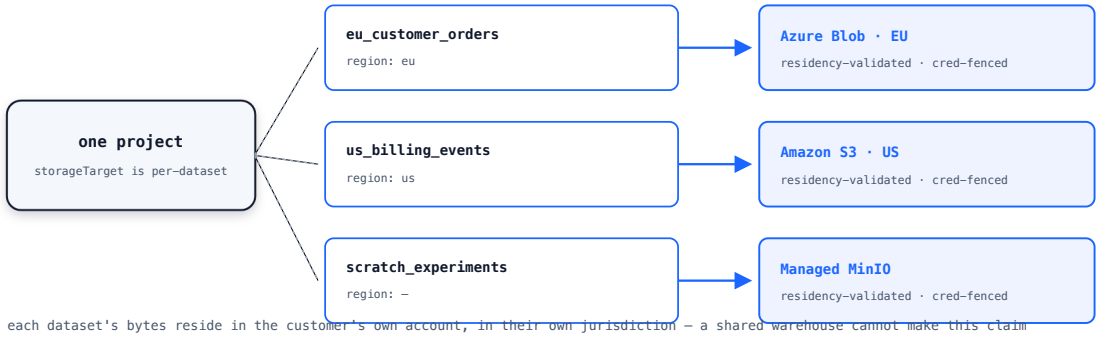

Identifying columns are sealed per subject; the analytical columns stay in the clear. Destroying a subject’s key makes their identifying cells permanently undecryptable — presenting as null — across every dataset, rollup, and object-store version, with no per-row scan and no repartitioning. You analyse the non-identifying columns while identities stay cryptographically sealed.

Erasure falls out of the same structure. Because the identity is a key and not a row, destroying a subject's field key makes their derived per-cell keys unrecoverable, and their identifying cells become permanently undecryptable — presenting as null — across every dataset, rollup, and object-store version derived from the source collection, including every Parquet version and object-version, with no scan and no repartitioning: the object store holds only ciphertext, so the key destruction reaches it at once. The source collection itself follows the platform's live-scoped erasure model — the field key destruction is immediate there too, and any physical database backup of that collection ages out on its own retention window. This is erasure by crypto-shred, the analytics-side tie-in to Provable Decommissioning. None of this is column masking — a role toggle over cleartext — and the paper keeps that distinction sharp: it is per-subject cryptographic sealing, and the confidentiality is a property of the key hierarchy, not of a policy that could be reconfigured.

## 08 Bring, or upload, any data — into your own storage

The engine is general, not operational-only. Data arrives through one governed write path with many adapters: a no-code upload wizard that infers, previews, and commits a schema from CSV, Excel, or JSON, and — for programmatic ingest — newline-delimited JSON as the single write primitive every other source funnels through, so schema conformance, residency derivation, classification, metering, and audit are enforced once, in one place, and no source can bypass governance. Parquet is the at-rest format datasets live in, not an upload format.

Storage targeting is where the sovereignty claim reaches its sharpest point. Storage is a per-dataset property: a dataset can point at customer-supplied storage rather than the managed default, across a matrix of seven providers — Amazon S3, any S3-compatible endpoint, Google Cloud Storage, Azure Blob, self-hosted MinIO, Cloudflare R2, and Backblaze B2 — with no per-provider tier-gating, every provider available regardless of plan. Because targeting is per-dataset, an EU dataset can reside in the customer's Azure Blob EU account, a US dataset in their S3 US bucket, and a scratch dataset on managed storage — each dataset's analytical bytes physically resident in the customer's own account, in their own jurisdiction, a claim a shared warehouse structurally cannot make. The residency invariant, the credential fence, and crypto-shred erasure all still hold, because the linkage keys are platform-custodied wherever the ciphertext bytes live.

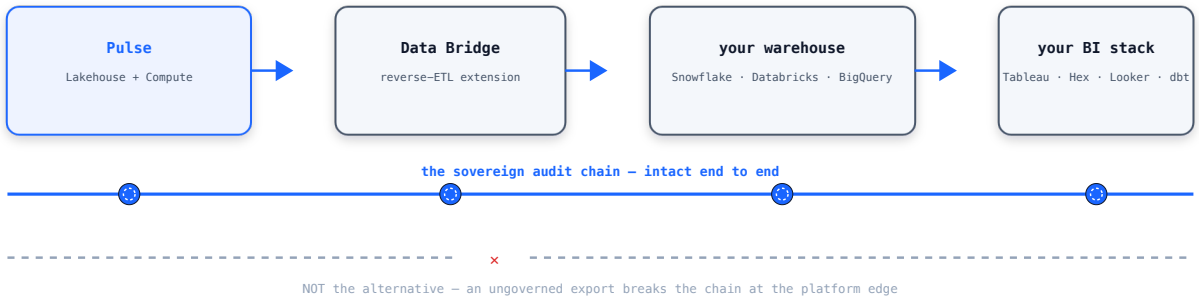



Storage targeting is a per-dataset declaration, not a project-wide setting. One project can hold an EU dataset bound to the customer's own Azure Blob account, a US dataset in their own S3 bucket, and a scratch dataset on managed storage — each binding residency-validated and cred-fenced at declare time.

## 09 Composable by design: bridging into the stack a team already runs

A small set of workloads — a global join on a non-partition key, a rank or sort over hundreds of terabytes — are native territory for a distributed-shuffle engine, and Pulse does not re-implement that inside a stateless fleet. Instead, the Data Bridge fleet reverse-ETLs Pulse and Stack data directly into the warehouse a team already runs — Snowflake, Databricks, BigQuery — so their analysts keep using the BI and notebook stack they already have: Tableau, Hex, Looker, dbt.

What makes this composable rather than a hand-off is what survives the crossing: the sovereign audit chain and the governance travel with the data across the bridge, so a workload that moves from Pulse into a warehouse never leaves the platform's chain of custody behind. Direct engine access, for the record, is DuckDB SQL over an HTTP JSON API — connectivity to external BI tools is bridge-delivered by design, which keeps the query surface simple while the ecosystem around it stays open. Pulse does not have to be the whole stack for every workload to be the sovereign core of it.




A Data Bridge reverse-ETLs Pulse and Stack data into the warehouse a team already runs, so their existing BI and notebook stack connects there as it always has — with the sovereign audit chain running unbroken across the whole path, not left behind at the platform edge.

## What we disclose

The scaling techniques in this paper — partition pruning, columnar pushdown, rollups, scatter-gather sharding, out-of-core execution — describe how the engine reduces what a query reads. No number in this paper is a benchmark, and the paper does not claim unlimited scale.

A native MPP warehouse — Snowflake, Databricks, Trino — is the better fit for one narrow class of query: a global high-cardinality shuffle, such as a full-dataset join on a non-partition key or a sort/rank over hundreds of terabytes, where a cross-node data shuffle is unavoidable. This is a rare pattern in the selective, partition-scoped queries Pulse is built for. Where it does occur: scatter-gather covers the mergeable cases, a rollup pre-materialises the rest, and the Data Bridge covers whatever remains by routing to the warehouse a team already runs — a governed path over the same Parquet, keeping the residency and role fences intact. No bridge is described as live until its per-bridge build status is confirmed.

The confidentiality property is cryptographic, not a masking policy: identifying columns are sealed per subject with AES-256-GCM-SIV under a per-subject key hierarchy and fail closed to null, and the operator reveal gate that exists for collections is deliberately absent from analytics.

# See how the unbundled model fits your data

Talk to a Sovereign Engineer about which of your analytical workloads stay sovereign in-platform, and where a governed bridge to the warehouse you already run is the right composition.

Talk to a Sovereign Engineer →

See how Pulse works