# Parsing Is Not Executing: Decentralized Compliance for Agentic Query Plan Routing

Ranjan Sinha
rsinha@us.ibm.com
IBM
San Jose, California, USA

## Abstract

An engine that parses a Substrait plan is not necessarily one that executes it correctly. We encountered this firsthand while building the Substrait Compliance Framework, a decentralized system for testing query plan interoperability across heterogeneous engines. The framework ships multi-language SDKs (Java, Python, Rust), 1,480 function-level test cases spanning 11 categories, and serialized TPC-H and TPC-DS benchmark plans with CI/CD-ready reporting. Testing three production engines showed that one achieves 22/22 syntactic TPC-H acceptance but falls short on full semantic validation, function-level pass rates of 94% on non-string categories mask dramatic per-category swings, and a prototype evidence-aware routing agent reduced semantic mismatches from 15 to 5 on a 50-item mixed workload (66.7% reduction) while exposing a string-processing category where no available engine meets the quality threshold.

## CCS Concepts

• **Information systems → Query languages**; • **Software and its engineering** → *Interoperability*; • **Computing methodologies** → *Artificial intelligence.*

## Keywords

Substrait, interoperability testing, decentralized compliance, AI agents, data systems, query plan portability, federated governance

**ACM Reference Format:**
Ranjan Sinha. 2026. Parsing Is Not Executing: Decentralized Compliance for Agentic Query Plan Routing. In *Proceedings of Supporting Our AI Overlords Workshop at CAIS 2026 (SAO 2026).* ACM, New York, NY, USA, 5 pages.

## 1 Introduction

Who writes your analytical queries? Increasingly, the answer is not a person. AI agents now select data sources, construct query plans, orchestrate multi-engine pipelines, and interpret results with minimal human oversight [3, 16]. This shift matters for infrastructure because agents operating at pipeline speed do not stop to inspect whether a `DECIMAL` rounded differently than expected, do not cross-reference results against domain intuition, and rarely validate outputs before passing them downstream. They trust the interchange format. When that trust is misplaced, errors compound silently across the pipeline.

Substrait [15] defines a portable intermediate representation for relational algebra, decoupling plan *producers* from *consumers*. The promise is clear: write a plan once, execute it faithfully anywhere. But SQL's decades of dialect fragmentation [5, 7] suggest caution. Even at the plan level, engines can parse a Substrait plan correctly while producing wrong results on aggregation edge cases, window function framing, or implicit casts [10]. Parsing is not executing.

This paper describes the Substrait Compliance Framework, a system we built to make these divergences visible. We report on testing three production Substrait consumers (anonymized as Engine A, B, and C) against 1,480 function-level tests and benchmark queries from TPC-H and TPC-DS. Engine B went from passing 2 of 22 TPC-H queries to all 22 after integrating the framework into its CI pipeline. Engine C parses every TPC-H plan without error yet does not achieve full semantic fidelity. And a naive routing agent that ignores compliance evidence produces 15 semantic mismatches on a 50-item workload; one that consults it reduces this to 5, while flagging a string-processing category where the ecosystem itself falls short.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
*SAO 2026, San Jose, California*
© 2026 Copyright held by the owner/author(s).

## 2 The Interoperability Gap

Data systems were designed for careful human operators who understand engine quirks, write queries in specific dialects, and manually verify results [16]. An agent composing an analytical workflow cannot do any of this. It expresses a plan once and expects faithful execution across whichever Substrait consumer it targets. The agent will not inspect type coercion rules or null propagation semantics; it relies on the interchange format entirely. To be fair, human operators are not reliably better at catching these issues: production data systems accumulate silent semantic bugs that go undetected for years. The difference is less about detection ability and more about feedback loops. A human analyst notices when a dashboard result looks implausible against domain experience; an agent in an automated pipeline passes results downstream without that sanity check. Our framework aims to close this gap by giving agents structured evidence that substitutes for the informal verification humans perform, and potentially surpasses it, since an agent can programmatically verify category-level compliance before every routing decision, something no human operator does consistently.

Why is this reliance risky? Because syntactic agreement has never guaranteed semantic equivalence in query processing [9]. Casting behavior, function definitions, null handling, optimizer assumptions: all vary across engines even when the surface syntax is shared [7]. Substrait operates at the plan level rather than the query text level, which helps, but the deeper challenge carries over.

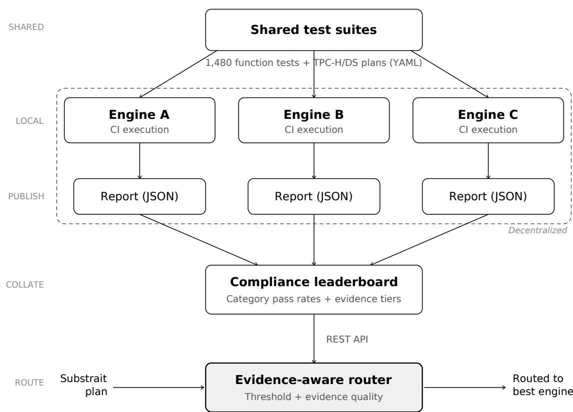


**Figure 1: Compliance framework workflow. Shared test suites are executed locally by each engine within its own CI pipeline (dashed region). Structured reports feed a public leaderboard, which the evidence-aware router queries before dispatching Substrait plans to the best-matching engine.**

Agentic workloads make this worse. An agent in an automated pipeline does not squint at the output and notice that revenue looks off; it propagates errors silently. And agents increasingly compose plans that span multiple engines in a single workflow. Every engine boundary is another chance for semantics to diverge. When agents generate high-throughput, speculative query variants [16], those mismatches compound fast.

One might argue that centralized conformance testing (a single authority validating all engines) would solve the problem. In practice, it does not scale well. It creates bottlenecks, conflicts with deployment constraints (some engines require local infrastructure or proprietary connectors), handles sensitive diagnostic artifacts poorly [4], and reduces conformance to a binary badge when agents actually need category-level granularity. Lakehouse architectures with pipeline-level atomicity [17] reinforce the point: testing belongs inside existing CI pipelines, not in an external service.

# 3 The Substrait Compliance Framework

We built a decentralized compliance framework with three layers: shared community artifacts (versioned test suites, metadata conventions, reporting schemas), engine-local execution, and standardized result publication. Figure 1 shows the end-to-end pipeline, from shared test suites through engine-local CI execution and structured report publication to the compliance leaderboard, which the evidence-aware router queries before dispatching each plan.

## 3.1 Multi-Language SDK

The framework ships SDKs in Java, Python, and Rust. Each provides an idiomatic implementation of a common contract: Java uses interfaces, Python uses abstract base classes, Rust uses traits. Engine developers pull the SDK into their own repository and implement the `ComplianceEngine` interface. The interface has four methods. Two are metadata: `getEngineInfo()` for engine name, version, and vendor, and `getCapabilities()` for declaring supported relations, functions, and types through an `EngineCapabilities` descriptor. The other two do the real work: `validatePlan()` checks structural validity of a Substrait plan, and `executePlan()` runs it and returns tabular results. All execution happens locally. No data leaves the engine team's environment.

Internally, the SDK uses a four-layer architecture with dependencies pointing inward (infrastructure, domain, application, presentation). Domain objects are immutable, which simplified concurrent test execution.

The capability declaration system turned out to be more important than we initially expected. An engine that does not implement geospatial functions declares this via `EngineCapabilities`, and the runner skips those tests automatically. Results carry four states: `PASSED`, `FAILED`, `SKIPPED`, `ERROR`. Without this distinction, a leaderboard conflates absence with failure and penalizes engines that report honestly.

## 3.2 Test Suite

The framework includes two complementary suites.

**Function-level tests.** Approximately 1,480 test cases across 115 files covering eleven categories: arithmetic (500+ tests), arithmetic decimal (80+ tests), string (319 tests), comparison (150+ tests), datetime (312 tests), boolean (27 tests), logarithmic, rounding, mathematical (100+ tests combined), list operations (2 tests), and aggregate (13 tests). Tests include trigonometric, bitwise, and other specialized operations. Each test specifies input data, a Substrait plan targeting a specific function with defined edge conditions (nulls, overflow, empty groups), and expected output with explicit floating-point tolerances.

**Benchmark queries.** 22 TPC-H queries [18] and 99 TPC-DS queries [12], serialized as Substrait plans in binary and JSON formats. TPC-H uses scale factor 0.01 (approximately 86,630 rows across 8 tables); TPC-DS uses minimal sample data (2-3 rows per table, 11 of 24 tables implemented, three sales channels: store, catalog, and web). Queries range from simple to very complex. Critically, these are *semantic fidelity* tests, not performance benchmarks. We measure correctness, not throughput [6].

## 3.3 Developer Workflow

We optimized for low friction. An engine team picks the SDK matching their language, implements `ComplianceEngine`, loads the test suite, runs it through `ComplianceRunner`, and wires it into CI/CD with our GitHub Actions template (one file, four configuration variables). Test suites are loaded from YAML files by default, but the loader interface is pluggable for teams that need alternative formats. Optionally, results go to the leaderboard.

Results flow through a hierarchical aggregation pipeline into a `ComplianceReport` (per-category pass rates, exportable as JSON, Markdown, HTML, or CSV). The public leaderboard sits behind a REST API with JWT authentication, rate limiting, and webhook support for downstream notifications. It separates TPC-H results from function-level results and assigns status tiers (Excellent, Good, Fair) based on pass rates.

## 4 Empirical Results

We ran three production Substrait consumers against the full suite.[1] A fourth (Engine D) is under integration but not yet mature enough for inclusion in this evaluation.

### 4.1 TPC-H

Engine A and Engine B both achieve 22/22 (100%) on semantic validation. Engine B's trajectory is worth noting: it started at 2/22 before the compliance framework was integrated into its development workflow. That progression, from near-total failure to full compliance, suggests the framework does not just measure quality but can drive it. Engine C tells a different story. It parses all 22 plans via its Calcite-based [2] Substrait path and passes 5/5 in a focused subset, but full 22-query semantic coverage lags behind Engine A and Engine B. An engine that accepts every plan is not the same as one that executes every plan correctly. For agents, syntactic acceptance without semantic verification is arguably worse than outright rejection, because it creates a false signal of correctness.

### 4.2 TPC-DS

Engine A and Engine C both reach 87/99 (87.9%) on TPC-DS syntactic coverage; all 87 pass structural validation against minimal data. The 12 failures cluster around complex window functions with multiple partitions, nested aggregations with `ROLLUP`/`CUBE`, and advanced correlated subquery patterns. We suspect several of these reflect specification ambiguity rather than implementation bugs. Engine B's TPC-DS results are not yet available; its 75% score in the routing evaluation (Table 1) is extrapolated from TPC-H performance rather than directly measured.

### 4.3 Function-Level Coverage

Of the approximately 1,480 function-level tests in the suite, 581 non-string tests were applicable to Engine B after filtering for declared engine capabilities. Engine B achieves 546/581 (94.0%) on this non-string subset; string-category results (319 tests) are reported separately below. Categories with mature Substrait extension definitions (arithmetic, comparison, boolean) reach near complete coverage. Categories involving string operations tell a different story: Engine B passes 58.3% (186/319) of string tests, and Engine A reaches 73.4% (234/319). This is not an outlier. Categories involving newer or less-specified extensions show similarly low pass rates and wide inter-engine divergence. An engine that nails boolean while stumbling on strings looks very different depending on what your plan needs. An agent composing a plan that calls `concat` and `regexp_replace` should not assume the same level of fidelity as one calling `and` and `or`. The aggregate score hides exactly this distinction.

### 4.4 Complementary Strengths

No engine dominates every dimension. Engine A provides the strongest semantic TPC-H reference path and broad TPC-DS coverage. Engine B serves as the primary function-level executor with strong overall coverage. Engine C contributes multi-engine path validation and TPC-DS structural testing. This is a complementarity that rewards routing strategies aware of it and punishes those that assume one engine handles everything equally well.

[1]Engine identifiers are anonymized. All three are mature, open-source analytical engines with active Substrait integration.

## 5 Agents and Compliance

### 5.1 Three Roles

The most straightforward use of the leaderboard is for routing. An agent queries category-level pass rates through the REST API, identifies which engines have known weaknesses in specific semantic areas, and avoids routing plans through gaps in coverage. Tagliabue et al. [16] describe this as the "data systems for agents" direction; our compliance API fits that vision.

Compliance evidence can also be *produced* through standard CI automation rather than consumed by agents. An engine's CI pipeline runs the suite on every commit, detects regressions, and submits structured reports. This is deterministic code, not agentic behavior: the GitHub Actions template executes a fixed sequence of steps (download SDK, build adapter, run tests, publish report). Quality gates fail builds when pass rates drop below configurable thresholds, catching semantic regressions before release. In lakehouse architectures with data branching [17], these runs can be isolated on a branch so test artifacts never touch production data.

A third role is more speculative. An agent monitoring query telemetry could notice that production plans heavily perform interval arithmetic, which happens to be underrepresented in the test suite, and flag the gap. We have not built this yet, but the plugin architecture was designed with it in mind; extension points for suites, loaders, reporters, and validators accept community contributions without core SDK changes.

### 5.2 Routing Evaluation

We benchmarked compliance across TPC-H (22 queries, approximately 86K rows), TPC-DS (99 queries), and function-level tests spanning eleven categories (1,480+ test cases total), then fed the measured pass rates into a prototype evidence-aware router. A "mismatch" occurs when a query is routed to an engine that produces incorrect results or fails to execute. The evaluation workload uses a 50-item subset: 20 TPC-H queries, 10 TPC-DS queries, and 20 function tests (10 arithmetic, 5 string, 5 boolean). Engine C was excluded due to incomplete TPC-H coverage. The naive router sends everything to Engine B, which we chose as the default because it represents the path of least resistance in practice: an embedded, zero-configuration engine that many automated pipelines adopt without evaluating compliance profiles. The evidence-aware router picks the engine with the highest pass rate above an 85% threshold, preferring real integration evidence over extrapolated estimates. When no engine meets the threshold, it selects the best available option and flags it as low-confidence.

The results (Tables 1 and 2) tell a more nuanced story than a clean sweep would. The naive policy produced 15 mismatches. The evidence-aware router reduced this to 5, a 66.7% improvement. All 10 avoided mismatches were TPC-DS queries: Engine B's TPC-DS score of 75% was extrapolated from its TPC-H performance with a complexity discount, since even fully TPC-H-compliant engines reach only 87.9% on the more demanding TPC-DS suite. The router correctly redirected these to Engine A (87.9%, real integration).

**Table 1: Compliance evidence driving routing decisions. Engine B's TPC-DS score is extrapolated; all other scores are from real integration testing.**

| Category | Eng. A | Eng. B | Selected |
|---|---|---|---|
| TPC-H | 100% R | 100% R | A (20) |
| TPC-DS | **87.9%** R | 75% E | A (10) |
| Arith. | 90% R | **95%** R | B (10) |
| String | 73.4% R | 58.3% R | A (5)† |
| Bool. | 100% R | 100% R | B (5) |
| **Total distribution** | | | A:35 B:15 |

R=REAL_INT. E=EXTRAP. †Below threshold

**Table 2: Mismatch avoidance by workload category.**

| Category | Items | Naive | Aware | Avoided |
|---|---|---|---|---|
| TPC-H | 20 | 0 | 0 | 0 |
| TPC-DS | 10 | 10 | 0 | 10 |
| Arith. | 10 | 0 | 0 | 0 |
| String | 5 | 5 | 5 | 0 |
| Boolean | 5 | 0 | 0 | 0 |
| **Total** | **50** | **15** | **5** | **10** |

For arithmetic, Engine B won legitimately (95% vs. 90%, both real integration). For boolean, both engines scored 100%.

The 5 remaining mismatches are string tests, and they expose a genuine limitation. Engine A passes 73.4% (234/319) of string tests; Engine B passes 58.3% (186/319). Both fall below the 85% threshold. The router selected Engine A as the lesser of two problems, but it cannot fix a category where no available engine is adequate. This is an honest result: compliance-aware routing helps where the evidence discriminates between engines, but it cannot conjure quality that does not exist. The string category is a gap the Substrait ecosystem needs to close at the implementation level rather than through routing.

The evidence-aware policy selected Engine A for 70% of queries and Engine B for 30%, compared to 100% Engine B under naive routing. Had Engine A been the naive default, the TPC-DS mismatches would vanish without routing logic, though the router would still improve arithmetic selection and flag the string gap.

## 6 Related Work

The lakehouse architecture [1] created the multi-engine topology that makes plan-level interoperability a practical concern. Work on lakehouse storage formats [8] surfaced analogous challenges between Delta Lake, Iceberg, and Hudi; we face the same class of problem at the query plan layer. Velox [11] took the shared-runtime approach: one execution library embedded across systems. Our model assumes independently developed engines, matching the Substrait ecosystem and the broader trend toward specialized engines [14]. Shankar et al. [13] showed that continuous validation outperforms one-time certification in ML pipelines; the same principle applies to plan semantics. TPC-DS [12] provides richer query structures than TPC-H and motivates our ongoing suite expansion.

Database conformance testing predates our work by decades. NIST maintained SQL suites, and JDBC/ODBC specifications include their own tests. What we do differently is test at the *plan level* rather than the query text level, which matters when agents bypass SQL entirely. The testing is also *decentralized*: engines run suites in their own CI instead of submitting to a central lab. And the output is *category-granular* rather than a binary pass/fail verdict, because an agent deciding where to route a plan cares about which specific operators are reliable, not just whether the engine passed an aggregate compliance check.

## 7 Discussion

The obvious weakness is self-reporting bias. Engines can game their results, and we have no mechanism to prevent it yet. Cryptographic attestation is on the roadmap, though it brings its own complications around key management. Coverage is another concern: 1,480 tests sound like a lot until you consider the space of possible plans. Some of the 12 failing TPC-DS queries may not be "bugs" at all, but areas where the Substrait specification has not settled on a single correct interpretation.

Environmental variation (hardware, OS, compiler) can confound cross-engine comparison. We considered mandating Docker but rejected it: added onboarding friction defeats the decentralized premise. Leaderboard rankings could also be misread as performance rankings; we use "compliance" rather than "benchmark" throughout, but conflation risk persists [6].

The routing evaluation is small (50 items, two engines in the routing pool, threshold-based policy). A production router would need plan decomposition across engines, dynamic evidence updates as engines ship new versions, and latency-aware selection that balances correctness against execution cost.

More broadly, decentralized conformance testing raises governance questions our framework surfaces but does not answer. Who decides when a disputed test case is "correct"? How should the community handle selective category reporting? These are socio-technical problems that will matter more as the ecosystem grows.

## 8 Conclusion

Can decentralized, category-level conformance testing produce evidence that agents can actually use for routing? Based on testing three production engines, the answer is a qualified yes. Engine A and Engine B pass every TPC-H query semantically; Engine C parses them all but stumbles on execution. TPC-DS reaches 87.9%. The 94% non-string function pass rate hides category swings large enough to change routing decisions. A prototype router reduced mismatches from 15 to 5 (66.7%) by routing away from an engine with only extrapolated evidence. The 5 residual string failures show where routing alone cannot help: both engines fall well short of the 85% threshold, and no amount of evidence-aware selection can fix a category where the ecosystem itself is immature. Engine B's progression from 2/22 to 22/22 on TPC-H may be the most practically significant result: the framework did not just reveal a gap, it helped close one. Next steps include expanding TPC-DS coverage, integrating Engine D, and feeding conformance evidence into cost-based optimizers [14].

## Acknowledgments

We thank the SAO Workshop organizers for the opportunity to discuss these ideas. We are grateful to colleagues at IBM Research for early feedback on the compliance framework design, and to the Substrait and Apache Arrow communities for ongoing collaboration on interoperability standards. We also thank the anonymous reviewers for their constructive comments.

## References

[1] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Proc. CIDR 2021*.
[2] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proc. SIGMOD 2018*. ACM, 221–230.
[3] Databricks. 2025. Trustworthy AI in the Agentic Lakehouse. arXiv preprint arXiv:2511.16402.
[4] Cynthia Dwork. 2006. Differential privacy. In *Automata, Languages and Programming*, 1–12. Springer.
[5] Andrew Eisenberg and Jim Melton. 1999. SQL:1999, formerly known as SQL3. *ACM SIGMOD Record* 28, 1 (1999), 131–138.
[6] Jim Gray. 1993. *The Benchmark Handbook for Database and Transaction Processing Systems* (2nd ed.). Morgan Kaufmann.
[7] Alon Halevy. 2005. Why your data won't mix. *Queue* 3, 8 (2005), 50–58.
[8] Paras Jain et al. 2023. Analyzing and Comparing Lakehouse Storage Systems. In *Proc. CIDR 2023*.
[9] Jim Melton and Alan R. Simon. 2001. *SQL:1999: Understanding Relational Language Components*. Morgan Kaufmann.
[10] Andy Pavlo et al. 2024. What Goes Around Comes Around... And Around... *SIGMOD Record* 53, 2 (2024), 21–37.
[11] Pedro Pedreira et al. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (2022), 3372–3384.
[12] Meikel Poess and Chris Floyd. 2000. New TPC Benchmarks for Decision Support and Web Commerce. *ACM SIGMOD Record* 29, 4 (2000), 64–71.
[13] Shreya Shankar et al. 2022. Operationalizing Machine Learning: An Interview Study. arXiv preprint arXiv:2209.09125.
[14] Michael Stonebraker and Ugur Çetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proc. ICDE 2005*. IEEE, 2–11.
[15] Substrait Project. Substrait: Cross-language Serialization for Relational Algebra. Retrieved May 17, 2026 from https://substrait.io/.
[16] Jacopo Tagliabue et al. 2025. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. In *Proc. CIDR 2026*. arXiv preprint arXiv:2509.00997.
[17] Jacopo Tagliabue et al. 2026. Building a Correct-by-Design Lakehouse. arXiv preprint arXiv:2602.02335.
[18] Transaction Processing Performance Council. TPC Benchmark H Standard Specification. Retrieved May 17, 2026 from http://www.tpc.org/tpch/.