A Rust Runtime for Categorical Data Systems: Implemented Semantic Kernel, Evidence Boundaries, and Production Roadmap

Matthew Long
The YonedaAI Collaboration
YonedaAI Research Collective
Chicago, IL
matthew@yonedaai.com · https://yonedaai.com

23 July 2026

Provisional paper status: OBSTRUCTED

Abstract

Categorical database theory supplies an attractive account of schemas, instances, mappings, and data migration. It does not, by itself, supply a production runtime. The engineering gap includes serialization, diagnostics, validation, compilation, physical planning, transactions, cancellation, connectors, deployment, and evidence that abstraction does not damage performance. This paper asks which part of that gap the accepted CatDB Rust evidence actually closes.

The accepted Slice 4 boundary contains six Rust crates and 4,051 lines of Rust source and tests. It implements closed serialized values; finite schema, path, mapping, and instance validation; mapping identity and composition; a bounded finite pullback migration Δ_F; and fixture verification. Its shared 33-case corpus contains 29 semantic cases and four structural-only cases under capability set `slice-4`. A fresh isolated Rust run passes all 29 semantic cases. An independently implemented Haskell process computes the same 29 results with zero disagreements. A separate Lean development machine-checks selected path and mapping laws, but it does not verify the Rust source.

The accepted result is a research kernel, not a production database runtime. There is no query IR, physical planner, plan cache, storage connector, SQLite or PostgreSQL execution path, transaction layer, asynchronous executor, server, WebAssembly target result, operation-replay engine, or performance experiment at either boundary. At the accepted Slice 4 boundary, Sigma and Pi return explicit unsupported outcomes, and information-loss and schema-diff envelopes are represented but not computed. The direct equation checker is deliberately incomplete.

We reconstruct the implemented boundary, state exact claim labels, analyze concrete counterexamples to common overclaims, compare the design with categorical database implementations and conventional Rust query engines, and specify a falsifiable path

from the semantic kernel to a compiled runtime. The narrow systems thesis remains an ENGINEERING HYPOTHESIS: a versioned categorical layer may be compiled into validated native plans and kept outside the hot execution path. No performance or production-readiness claim is made.

Contents

1 Introduction

1.1 The problem in plain language

A schema mapping is easy to draw and surprisingly hard to execute safely. Suppose one source stores customers and orders in normalized tables while a consumer expects one document per customer. A semantic mapping may say that the consumer's `orders` field follows a relationship path, that money is measured in cents, and that two paths to an account identifier must agree. Those statements are more informative than a pile of column names. They still leave a runtime with practical work: reject an ill-typed path, preserve the equation, choose where the join runs, control a transaction, report why a mapping failed, and avoid recompiling an unchanged plan.

Category theory helps describe the meaning of the transformation. It does not make network calls, enforce a database isolation level, estimate a join, or cancel a remote query. Rust can help structure the implementation and exclude classes of memory errors. It does not prove that cents were converted to dollars or that two schemas mean the same thing. A credible runtime paper must keep these obligations separate.

The distinction matters especially in an ambitious research program. Architecture diagrams can contain an IR, a planner, connectors, a server, and a WebAssembly target long before those components exist. Conversely, a small kernel may contain useful executable evidence even though it is not yet a system. This paper treats the manifest, source, tests, and release records as the boundary. Names in a roadmap are not counted as implementation.

1.2 Research question

The paper asks:

> Which semantic and runtime obligations are implemented by the accepted CatDB Rust boundary, how strong is the evidence for them, which obligations remain unsupported, and what additional contracts and experiments are required before the workspace can support production-runtime claims?

The answer has two parts. First, the accepted kernel implements a meaningful finite slice: typed schema and mapping operations, finite instances, and bounded Δ, connected to shared Rust/Haskell fixtures and selected Lean laws. Second, nearly every claim involving physical

data systems remains unimplemented. Work after Slice 4 is excluded from the terminal paper evidence.

1.3 Bounded thesis

> For the closed finite profile admitted by `slice-4`, the accepted Rust kernel computes deterministic structured outcomes for schema, path, mapping, instance, and bounded pullback-migration operations, and its 29 semantic fixture outcomes agree exactly with an independent Haskell evaluator.

Evidence status. COMPUTATIONAL. The claim is finite and corpus-indexed..

The corresponding production thesis is weaker and prospective:

> A future CatDB runtime may compile versioned semantic obligations into backend-native plans, cache the compiled artifacts, and keep the categorical layer outside the hot row-processing path.

Evidence status. ENGINEERING HYPOTHESIS. No backend, planner, or performance run exists..

1.4 Contributions

This provisional paper contributes:

1. a source-grounded reconstruction of the six-crate Rust workspace and its exact dependency boundary;
2. a separation among serialized representation, semantic validation, abstract laws, and physical execution;
3. a release-evidence ladder connecting shared fixtures, Rust computation, independent Haskell computation, and selected Lean theorems;
4. a claim register that rejects production, safety, equivalence, and performance overstatements;
5. adversarial examples showing why safe Rust, cross-language agreement, immutable artifacts, async frameworks, and successful WASM compilation do not establish database correctness; and
6. a staged architecture and benchmark contract with falsification gates.

1.5 Nonclaims

This paper does not claim that CatDB is production-ready; that Rust proves semantic correctness; that Lean verifies the Rust implementation; that cross-language agreement proves general equivalence; that all three functorial migrations are implemented; that any backend is connected; or that the semantic layer is free at runtime. It does not report a performance number.

2 Background and related systems

2.1 Categorical data migration is established prior art

In the classic functorial account, a schema is a small category and an instance is a set-valued functor. A schema mapping induces three data migration functors commonly written Σ_F, Δ_F, and Π_F [15]. Relational foundations connect this language to executable relational operations and composition results [16]. Algebraic data integration and algebraic databases extend the treatment toward types, equations, queries, and practical integration [12, 13].

This literature makes several aspects of CatDB non-novel: categories as schemas, functor-like mappings, path composition, and the existence of the three data migration directions. Implementations also predate CatDB. CQL offers an executable categorical query and integration language and explicitly distinguishes itself from a DBMS [5]. Catlab provides finite category presentations and attributed C-sets in Julia [1].

The possible CatDB contribution is therefore systems-specific. It would need to show how versioned categorical semantics interact with storage connectors, planning, materialization, provenance, workspaces, and distribution, while comparing directly with existing categorical tools. Reimplementing a small subset in Rust is useful infrastructure, not by itself a new database model.

2.2 Rust narrows some implementation risks, not semantic risk

Rust's ownership discipline moves many lifetime and aliasing checks into compilation [11]. RustBelt gives a machine-checked safety account for a realistic language subset and selected libraries while making the obligations of unsafe extensions explicit [8]. These results justify careful statements about language-level memory safety. They do not imply that an arbitrary safe Rust program implements its intended mathematics.

Cargo workspaces share resolution and a lockfile among members [10]. Serde can reject unknown fields in self-describing formats through `deny_unknown_fields` [14]. The current kernel uses both mechanisms. Neither is a proof of protocol compatibility across future IR versions.

2.3 Physical execution has its own mature design space

Volcano's operator interface and extensible dataflow architecture separate mechanism from operator semantics [6]. Cascades adds rule-based transformation, memoization, and physical-property reasoning [7]. SQLite and PostgreSQL both expose cost-based planning choices in their documentation [9, 17]. Apache DataFusion is an extensible Rust query engine with logical and physical optimizer rules and an Arrow-oriented execution architecture [3].

Consequently, a future CatDB compiler should not build an optimizer merely to have one. It must identify the extra semantic properties that constrain ordinary rewrites, feed those properties into a proven execution framework or a justified custom engine, and measure the cost of doing so.

2.4 Candidate technologies are not current features

Tokio provides tasks, scheduling, asynchronous I/O, synchronization, and runtime support [18]. Apache Arrow specifies a language-independent columnar memory and IPC format [2]. WebAssembly specifies a portable low-level code format [19], and Wasmtime offers a Rust embedding API [4]. These are credible candidates for a runtime roadmap. None appears in the audited Cargo dependency graph.

3 Evidence discipline

3.1 Four distinct questions

A runtime evaluation becomes clearer when it asks four questions separately:

1. **Representation:** Can a value be decoded, encoded, and normalized under a closed contract?
2. **Semantics:** Does the implementation validate or compute the intended operation?
3. **Law:** Does a proof assistant establish a stated invariant in a specified formal representation?
4. **Execution:** Does a backend plan preserve the semantic result, transaction, resource, and failure contracts?

The accepted Slice 4 repository boundary has substantial evidence for the first two, selected evidence for the third, and no evidence for the fourth.

3.2 Claim labels

Label	Interpretation in this paper
COMPUTATIONAL	Reproduced by executable code over a stated finite surface.
MACHINE-CHECKED	Checked by Lean's kernel for named declarations only.
SUPPORTED BY PRIOR LITERATURE	Established externally; not a CatDB result.
ENGINEERING HYPOTHESIS	Plausible implementation proposition without current CatDB evidence.
OPEN	Well-posed research question without a result.
OBSTRUCTED	Required component, experiment, or review does not exist.
REJECTED AS WRITTEN	Contradicted by the live source or stronger than its evidence.
NONCLAIM	Explicitly excluded from the result.

3.3 Accepted boundary and method

The accepted source audit uses the Slice 4 acceptance commit

056915834fcc2f172a4a72efe3629f4f753e45cf.

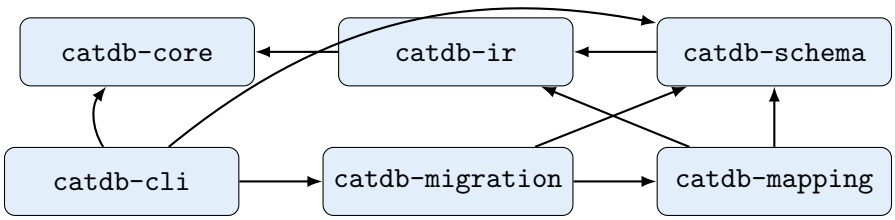
That commit closes Slices 1–4 and is the maximum code-evidence boundary used for every computational claim in this paper. We enumerated every workspace member and member manifest, resolved the dependency graph, read the implementation and tests, traced accepted Slice 0–4 commits into the accepted commit, and reran all accepted gates from an isolated archive. We also searched the accepted manifests, lockfile, and Rust tree for names associated with the proposed runtime subsystems. Later working-tree changes are outside the evidence boundary and are neither counted nor characterized here.

This method prevents two common errors. First, a directory name or roadmap entry is not implementation evidence. Second, a direct-name textual guard is not a dependency proof when Cargo permits workspace aliases and inherited dependencies. The resolved manifest graph is the relevant boundary.

4 The implemented Rust kernel

4.1 Workspace shape

The root manifest declares exactly six members:



Rank	Crate	Direct project dependencies	Implemented responsibility
0	`catdb-core`	none	IDs, scalar envelopes, diagnostic codes and phases
1	`catdb-ir`	core	closed serialized IR and structural normalizers
2	`catdb-schema`	core, IR	schema, path, and direct-equation validation
3	`catdb-mapping`	core, IR, schema	total mapping validation, identity, composition
4	`catdb-migration`	core, IR, schema, mapping	finite instance validation and bounded Δ
9	`catdb-cli`	all five	fixture verification and Rust/Haskell comparison

Table 2: Accepted Slice 4 workspace. Rank is architectural reading order, not a Cargo feature.

The workspace is version 0.1.0, uses Rust edition 2024, and declares an Apache-2.0 license field. Its direct external dependencies are only Serde, `serde_json`, and `tempfile`. The full Rust source and test surface contains 4,051 lines at the accepted Slice 4 commit.

4.2 Core identifiers, scalars, and diagnostics

The core crate defines a transparent string identifier, a closed scalar type enumeration, scalar values, diagnostic codes, phases, and structured diagnostics. Integers are signed 64-bit values at the IR boundary. Diagnostics contain stable machine fields rather than using host exception prose as the comparison contract.

This is a sensible kernel boundary. A parser can preserve a submitted JSON Pointer and semantic references even when the human-readable rendering changes. Independent implementations can compare the code, phase, pointer, and references directly. However, the current identifier is not a compact numeric runtime ID. The program's proposed "numeric identifiers" remain OBSTRUCTED.

4.3 Closed language-neutral IR

The IR crate defines closed Serde values for schemas, paths, mappings, instances, migration requests and outcomes, information-loss results, schema diffs, and fixture envelopes. Named records deny unknown fields. Canonical serialization sorts arrays only after input validation has had an opportunity to observe duplicates and submitted positions.

Contract 4.1 (Representation is not execution)**.** For every IR operation O, the paper distinguishes a decoder decode_O, a normalizer $\mathsf{normalize}_O$, a validator $\mathsf{validate}_O$, and, when implemented, an evaluator eval_O. Successful decoding or normalization does not imply successful validation or evaluation.

This contract catches a real boundary in the accepted source. The instance normalizer orders carriers and assignments but does not validate them. The loss and diff normalizers order supplied output structures but do not compute loss or a diff. A well-formed output envelope can still make a false semantic assertion if supplied by an untrusted caller.

4.4 Finite schemas and typed paths

The schema crate validates the current identifier syntax, duplicate declarations, arrow endpoints, required attributes, and the parallel typing of equation paths. A path carries a start object and a sequence of arrows. Typing folds the arrows from the start object:

$$\mathrm{type}_S(A; f_1, \ldots, f_n) = \begin{cases} A \to A, & n = 0, \\ A \to B_n, & \mathrm{src}(f_1) = A \wedge \bigwedge_{i<n} \mathrm{tgt}(f_i) = \mathrm{src}(f_{i+1}), \\ \mathsf{error}, & \text{otherwise.} \end{cases}$$

Binary path composition requires the first path's target to equal the second path's source and concatenates arrows in traversal order. Deterministic path keys make canonical order independent of declaration order.

The equation boundary is deliberately small. The checker accepts syntactic path equality or one directly declared target equation, with orientation ignored. It does not compute the congruence generated by the equations.

Counterexample 4.2 (Transitive equation). Let the target schema declare $p = q$ and $q = r$. A mapped source equation with images p and r is valid in the generated equivalence relation. The direct checker need not accept it because no single declaration says $p = r$.

Evidence status. Schema and path behavior is COMPUTATIONAL for the admitted fixture profile. A complete equational decision procedure is REJECTED AS WRITTEN..

4.5 Mappings

A raw mapping contains:

$$F_0 : \mathrm{Obj}(S) \to \mathrm{Obj}(T), \qquad F_1(f) : \mathsf{Path}_T(F_0 A, F_0 B),$$

together with attribute expressions in the current typed profile. The mapping crate requires total object, arrow, and attribute assignments, checks endpoint and scalar typing, and uses the bounded equation checker for preservation. Identity mappings send each object to itself, each arrow to its singleton path, and each attribute to its direct expression.

Composition substitutes object images, target paths, and attribute expressions, then validates the result. This design means composition does not silently bless a malformed intermediate value.

Proposition 4.3 (Source-level mapping property)**.** *For every mapping value returned successfully by the current identity or composition functions, the implementation has rerun its bounded mapping validator before returning.*

Proof. This is a claim about the inspected call paths, not a mathematical theorem of all Rust executions. The identity constructor builds a total mapping and calls validation. The composition function validates both inputs, constructs the substituted result, and validates that result before returning success. □

Evidence status. COMPUTATIONAL source and test evidence. The general categorical laws belong to the separate Lean boundary described in Section 5..

4.6 Finite instances

The migration crate validates a closed, finite, total instance profile. A carrier assigns a finite set $I(A)$ to each schema object. Every arrow $f : A \to B$ is interpreted as a total function $I(f) : I(A) \to I(B)$. Required attributes are total and typed. Path equations are checked pointwise.

For a path $p = f_1; \cdots ; f_n$, evaluation is ordinary function composition:

$$I(p)(x) = I(f_n)(\cdots I(f_2)(I(f_1)(x)) \cdots).$$

The validator checks exact schema identity, duplicate carrier and generator records, carrier coverage, assignment coverage and totality, carrier membership, scalar types, and equations. Error selection retains submitted array positions in the shared diagnostic contract.

The profile excludes nulls, optional attributes, partial arrows, infinite carriers, quotient representations, and richer value-domain operations. Those are not defects hidden by the paper; they define the current admitted domain.

4.7 Bounded pullback migration

For a lawful mapping $F : S \to T$ and a finite T-instance J, pullback migration is represented by precomposition:

$$\Delta_F(J) = J \circ F.$$

On objects, the implementation selects $J(F(A))$. On a source arrow, it evaluates the mapped target path. On an attribute, it evaluates the admitted path-then-attribute expression. The result is validated again before success.

Example 4.4 (Path attribute). Suppose S has `Order.customerEmail`, while T represents the same value by the path `Order.customer; Customer.email`. The admitted Δ_F evaluator follows the mapped relationship path and then reads the target attribute for each source carrier element.

Only this finite pullback direction is implemented. Requests for Σ_F and Π_F return exact unsupported outcomes. The code does not perform a physical migration, create a table, open a transaction, cut over a service, or roll back a failed deployment.
Evidence status. COMPUTATIONAL on thirteen migration fixtures: two successful finite Δ examples, nine validation failures, and two explicit unsupported outcomes..

4.8 Fixture CLI

The CLI exposes only fixture verification and Rust/Haskell comparison. It supports capability sets `slice-1` and `slice-4`. The verifier:

1. recursively enumerates JSON fixtures;
2. closed-decodes and exact-round-trips each fixture;
3. computes an actual Rust outcome for admitted operations;
4. compares the computed result with the golden expected value;
5. marks loss and diff as structural-only; and
6. reports exact counts.

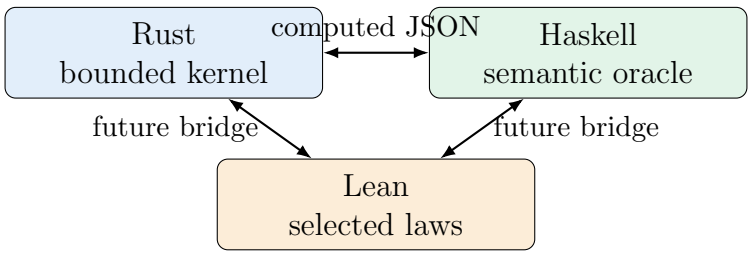
The comparison command launches Cabal as a process in the Haskell package, passes explicit tool paths when configured, receives a fresh temporary JSON report, verifies the report identity and coverage, and compares each independently computed semantic result.

This is a good research harness. It is not a reusable runtime API. It also should not be exposed to hostile fixture trees without resolving the retained symlink-traversal hardening note.

5 Cross-language and formal evidence

5.1 Why three implementations?

The research program uses one shared fixture corpus with three different roles:



Rust is the proposed production language. Haskell is an independent executable semantic oracle. Lean is a proof representation for high-value laws. The roles do not collapse into one another.

5.2 Release-evidence ladder

Slice	Artifact	Positive evidence	Retained boundary
0	closed fixture contract	JSON Schema, manifest integrity, reference closure	no semantic implementation
1	Rust schema, path, and mapping kernel	16 semantic and 8 structural cases in the original 24-case corpus	direct equations only; no instance, query, or backend semantics
2	independent Haskell oracle	exact Slice 1 process-boundary agreement and bounded properties	computational, not proof; no migration semantics in that slice
3	Lean path and mapping kernel	kernel checks for `PATH-02`, `PATH-03`, `PATH-04`, `MAP-03`, `MAP-04`, and `MAP-05`	no Rust equivalence, parsing, migration, SQL, or systems theorem
4	finite instances and Δ	accepted 33 cases: 29 semantic, four structural, zero disagreement	no Sigma/Pi, physical execution, performance, or distribution

The accepted implementation and acceptance commits for Slices 0–4 are ancestors of the Slice 4 acceptance commit. The release records also retain failed review rounds and nonblocking findings. Those code reviews are evidence for the slices; they are not peer review of this manuscript.

5.3 What differential agreement establishes

Under capability set `slice-4`, both implementations closed-decode all 33 fixtures. They classify 29 as semantic and four as structural-only. On the 29 semantic cases, the fresh comparison reports zero disagreements.

Proposition 5.1 (Finite differential result)**.** *For the 29 fixture inputs in the accepted manifest that capability set `slice-4` classifies as semantic, the fresh Rust and Haskell processes emit equal machine-comparison JSON outcomes.*

Evidence status. COMPUTATIONAL. The equality is finite and tied to the manifest digest..

This result is stronger than one implementation comparing itself with a golden file. It can expose independent parsing, normalization, computation, or diagnostic differences. It remains weaker than a correctness proof.

Counterexample 5.2 (Shared defect). If both implementations apply the same mistaken ordering rule to equations, they agree exactly and are both wrong. The Slice 4 history contains a concrete variant: derived Haskell ordering once disagreed with the normative serialized path key, and the fixture boundary had to expose and repair it.

5.4 What Lean establishes

The separate Lean slice defines indexed typed paths, identity and composition, typed schema mappings, and lawful mapping composition. Kernel evidence applies exactly to the six named invariant IDs. It does not parse the JSON fixture language, reproduce Rust diagnostics, validate finite instances, or compute Δ. There is no refinement proof connecting a Rust value to the Lean representation.

Therefore, the statement "Lean verifies CatDB's Rust runtime" is REJECTED AS WRITTEN. The accurate statement is that Lean machine-checks selected abstract path and mapping laws relevant to the runtime design.

6 Fresh accepted reproduction

6.1 Toolchain and isolation

The accepted run used Rust 1.94.0, Cargo 1.94.0, Python 3.12.9, Cabal 3.16.1.0, and GHC 9.14.1. Commit `0569158` was extracted with `git archive`; build state and captured output were placed in fresh temporary directories. Existing release evidence was not overwritten. Explicit Homebrew Cabal and GHC paths avoided obsolete tools elsewhere on the host.

6.2 Commands

```
python3 scripts/validate-fixtures.py \
  --schema fixtures/schema-v1.schema.json \
  --cases fixtures/cases
python3 scripts/audit-fixture-manifest.py fixtures/manifest-v1.json
python3 -m unittest discover -s scripts/tests -p 'test_*.py'

cargo fmt --all --check
cargo clippy --workspace --all-targets --all-features -- -D warnings
cargo test --workspace --all-features

cargo run -p catdb-cli -- fixtures verify fixtures/cases \
  --implementation rust --capability-set slice-1
cargo run -p catdb-cli -- fixtures verify fixtures/cases \
  --implementation rust --capability-set slice-4
```

```
CATDB_CABAL=/opt/homebrew/bin/cabal \
CATDB_GHC=/opt/homebrew/bin/ghc \
cargo run -p catdb-cli -- fixtures compare fixtures/cases \
  --left rust --right haskell --capability-set slice-4
```

6.3 Results

Gate	Semantic	Structural	Failure/disagreement
Fixture schema and manifest	33 closed cases		0
Rust `slice-1` on accepted corpus	16	17	0
Rust `slice-4`	29	4	0
Rust/Haskell `slice-4`	29	4	0

Table 4: Fresh accepted Slice 4 fixture results. The corpus has 12 successful, 19 error, and two unsupported expected outcomes.

The Python suite ran 19 tests. The Rust workspace ran 26 test functions with no failure or ignored test. Formatting and Clippy with warnings denied passed. The explicit cross-language path built Haskell and succeeded, but emitted Haddock missing-documentation warnings. This paper describes the semantic comparison as successful, not the entire command as warning-free.

6.4 Why counts are part of the claim

Reporting only "all tests pass" hides capability boundaries. The accepted `slice-1` runner structurally round-trips seventeen cases that it does not evaluate. The `slice-4` runner still has four structural-only cases: two information-loss and two schema-diff fixtures. Those four are not semantically evaluated. Totals must be derived from the accepted manifest rather than frozen as magic numbers.

7 Runtime claims the code does not support

7.1 Negative dependency and source audit

At the accepted Slice 4 boundary, the resolved workspace has only Serde, `serde_json`, and `tempfile` as external dependencies. The manifest, lockfile, and Rust crate tree contain no occurrence of the runtime technologies or components `tokio`, `sqlx`, `rusqlite`, PostgreSQL bindings, `wasmtime`, service frameworks, planner, connector, server, storage, replica, or distributed runtime terms. No `async` or `await` appears. The absence audit does not prove that future code cannot implement these components; it establishes that this snapshot does not.

Program claim	Current status	Evidence required before promotion
compact semantic IR	COMPUTATIONAL, bounded	profile versioning plus serialization, memory, and lookup baselines
numeric identifiers	REJECTED AS WRITTEN	implemented interning and stable external-identity contract
immutable compiled schemas	OBSTRUCTED	compiled artifact type, dependency digest, concurrency and invalidation tests
Rust trait boundaries	OBSTRUCTED	connector, planner, and executor traits with compatibility tests
async execution	OBSTRUCTED	runtime dependency plus cancellation, backpressure, and blocking-I/O tests
SQLite execution	OBSTRUCTED	connector, lowering, transaction, and differential-query evidence
PostgreSQL execution	OBSTRUCTED	capability negotiation, dialect, transaction, and differential evidence
logical and physical planning	OBSTRUCTED	operator IR, rewrites, physical properties, and result-preservation tests
plan caching	OBSTRUCTED	complete cache key, invalidation rules, and stale-plan counterexamples
deterministic operation replay	OBSTRUCTED	operation algebra, log, state machine, crash and reorder tests
WASM compatibility	OBSTRUCTED	target build, host contract, supported-feature matrix, and measurements
embedded deployment	OBSTRUCTED	stable embedding API and reference application
server deployment	OBSTRUCTED	protocol, service, authentication, resource and failure tests
native-equivalent performance	OBSTRUCTED	pre-registered native SQL comparison with raw observations
distributed runtime	OBSTRUCTED	replication implementation and convergence/consistency evidence

7.2 "Compact" and "immutable" need operational definitions

The serialized IR is small in conceptual scope, but no measurement compares its heap footprint, decode time, lookup cost, or compiled representation with a baseline. Calling it compact without a metric is OPEN.

Likewise, source values can be treated immutably while a cache points to the wrong revision. A production compiled artifact should bind at least:

$$k = H(s, m, c, v, p, b),$$

where s is the schema revision, m the mapping revision, c the backend capability profile, v scalar and collation semantics, p policy, and b compiler build. Omitting one dependency can yield a perfectly immutable stale plan.

7.3 No physical execution

The current Δ evaluator operates on finite JSON-shaped in-memory instances. It does not translate a mapping to SQL. No transaction is opened. No source capability is negotiated. No cursor, prepared statement, index, or network connection exists. Accordingly, "storage-independent execution" is a program objective, not a Paper 15 result.

7.4 No performance evidence

The fixture suite checks exact results and diagnostics. It does not time the kernel under a controlled benchmark protocol. The repository has no native SQL baseline because it has no SQL execution path. The central claim that the semantic layer can stay outside the hot path remains testable but OBSTRUCTED.

8 Rust safety without category errors

8.1 First-party source observation

No `unsafe` block occurs in the current CatDB crate source. This is a useful local observation. It is not a whole-program or supply-chain theorem: the standard library, compiler, operating system, and transitive dependencies are outside that scan. Future database connectors may introduce FFI.

8.2 Three kinds of correctness

Counterexample 8.1 (Safe Rust, wrong semantics). A total safe Rust function copies an integer amount without applying the mapping's scale conversion. It has no dangling pointer and can pass every memory-safety check while returning the wrong business value.

Counterexample 8.2 (Correct mapping, unsafe cutover). A future connector computes every mapped row correctly but commits half the target tables before a process crash. Semantic validity does not establish transaction atomicity or recovery behavior.

Kind	Example failure	Relevant evidence
Memory safety	use-after-free in a connector buffer	Rust type system, unsafe audit, fuzzing, dependency review
Semantic correctness	cents mapped as dollars	fixtures, differential evaluator, proof or refinement bridge
Database correctness	partial target update after failure	transaction tests, isolation model, fault injection

Table 6: Different correctness obligations require different evidence.

8.3 Panic and denial-of-service boundaries

The inspected migration implementation uses internal `expect` calls only after validation establishes the assumed lookup. That pattern can be reasonable, but this paper does not claim panic freedom. A hostile input tree, pathological allocation request, dependency bug, or violated future invariant can still matter. Production gates should include fuzzing, allocation limits, panic policy, and resource-exhaustion tests.

9 Adversarial counterexamples

This section collects cases that a production architecture must answer before promoting a claim.

Counterexample 9.1 (Agreement is not truth). Rust and Haskell share a mistaken interpretation of a scalar date. All 29 fixtures avoid the disputed date, so exact agreement remains zero-disagreement while both implementations fail on the next input.

Counterexample 9.2 (Diagnostic equivalence is not general). The accepted Slice 4 release records two unfixtured multi-fault cases in which Rust and Haskell may select different first diagnostics. The admitted 33-case comparison remains true, but general diagnostic equivalence does not follow.

Counterexample 9.3 (Accepted Slice 4 structural loss result). A caller supplies a well-formed information-loss envelope containing an empty finding list for a many-to-one mapping. At the accepted Slice 4 boundary, the IR can normalize the envelope; no loss analyzer recomputes whether the assertion is true.

Counterexample 9.4 (Stale capability plan). A SQLite backend revision stops supporting a collation-preserving pushdown. If the plan cache key omits the capability revision, the immutable old plan is reused and returns a different order or comparison result.

Counterexample 9.5 (Cancellation without cleanup). A future federated request times out while a blocking driver call continues on a worker thread and holds a database connection. Selecting an async runtime does not define cancellation, cleanup, or source-side interruption.

Counterexample 9.6 (WASM target mismatch). The semantic crates compile for `wasm32-unknown-unknown`, but the selected SQLite connector requires filesystem and native-thread behavior unavailable in the host. Compilation of a subset is not deployment compatibility.

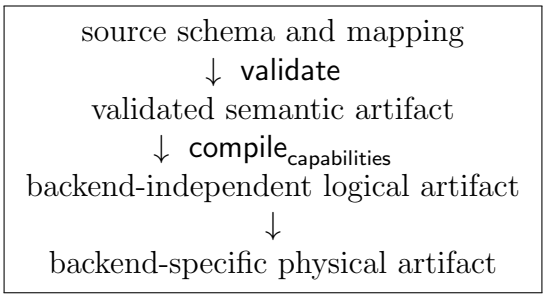
Counterexample 9.7 (Identifier compression changes meaning). A compiled artifact replaces stable external identifiers with small integers. After a schema rebuild, integer 17 names a different object. Reusing a plan by numeric ID alone applies the old mapping to the new object.

Counterexample 9.8 (Fast wrong benchmark). A benchmark compares native SQL with provenance disabled against CatDB output that omits required provenance fields. Lower latency is achieved by changing the requested result contract, so the comparison is invalid.

10 A falsifiable production architecture

10.1 Separate source, validated, compiled, and physical values

The proposed runtime should expose four artifact kinds rather than one overloaded IR:



A source value preserves user evidence and diagnostic positions. A validated value carries a successful profile and dependency set. A logical artifact contains normalized operators and semantic obligations. A physical artifact contains a backend choice, pushed operations, residual operations, parameters, and a result-reconciliation contract.

Requirement 10.1 (No unvalidated constructor path)**.** *Every public function that produces a validated, logical, or physical artifact must either validate its source inputs or accept only an opaque value whose constructor already did so.*

10.2 Compiled schema representation

Numeric interning can reduce lookup and storage costs, but external semantic identity must remain stable. One possible structure is:

$$\mathsf{CompiledSchema} = (\mathit{sourceDigest}, \mathit{objects}, \mathit{arrows}, \mathit{attributes}, \mathit{equations}, \mathit{indexes}),$$

where every numeric ID is local to *sourceDigest*. Serialization of an ID without its artifact identity should be rejected.

Compilation should precompute path endpoint tables, attribute-domain tables, equation-normal-form data for the declared fragment, dependency edges, and backend-independent statistics. Whether that representation is actually compact is a measurement question.

10.3 Trait boundaries

Rust traits are useful only after their semantic contracts are fixed. A connector boundary needs more than methods named `scan` and `join`. It must expose typed capabilities:

$$\mathsf{capabilities}(B) = (\text{operators}, \text{scalar semantics}, \text{collation}, \text{transactions}, \\ \text{limits}, \text{freshness}, \text{failure modes}).$$

A planner may push an operator only when the backend capability witness and the mapping obligations establish equivalence for the request. Unsupported work must remain residual or make the plan infeasible.

10.4 Logical and physical planning

Conventional logical operators—scan, project, filter, join, aggregate, union, sort, limit—remain appropriate. CatDB adds metadata rather than replacing the algebra:

- schema and mapping revision dependencies;
- path and equation obligations;
- identity and reconciliation policy;
- provenance requirements;
- allowed information loss or approximation;
- source authority and update policy; and
- result freshness and consistency.

These properties constrain transformations. They do not eliminate cost-based planning. Volcano, Cascades, SQLite, PostgreSQL, and DataFusion remain the proper baseline family.

10.5 Plan caching

A cache entry should record:

$$\mathsf{PlanKey} = H(\mathsf{LogicalQuery}, \mathsf{schemaRev}, \mathsf{mappingRev}, \mathsf{capabilitiesRev}, \\ \mathsf{ScalarProfile}, \mathsf{PolicyRev}, \mathsf{CompilerRev}).$$

The result should carry the key and the proof obligations discharged during lowering. A cache hit is valid only when every dependency is identical or an explicit compatibility relation has been established.

10.6 Asynchronous execution

An async executor requires a contract before an implementation:

- parent and child cancellation propagation;
- timeout ownership and source-side interruption;
- bounded queues and backpressure;
- isolation of blocking drivers;
- transaction and connection lifetime;
- partial-result policy;
- deterministic error aggregation; and
- provenance for retries and fallback.

Tokio is a plausible mechanism, not the contract itself. The runtime should test cancellation at every await and blocking boundary.

10.7 Embedded, server, and WASM modes

Embedded and server deployments expose different failure, trust, and resource boundaries. They should not be one compile-time flag described as equivalent. Similarly, WASM compatibility should be reported by feature set:

Target	Required test	Example limitation
native embedded kernel	library API and host integration	process shares caller resources
native server	protocol, auth, quotas, crash recovery	network and tenancy boundary
browser WASM	compile, instantiate, deterministic fixture run	no native DB driver
WASI component	component interface and allowed host calls	capability and filesystem policy
embedded Wasmtime plugin	host import/export and resource limits	runtime and sandbox configuration

11 Implementation roadmap

11.1 R1: stabilize the semantic kernel

The next accepted semantic release should version the IR, define compatibility rules, expand generative and mutation testing, replace the direct-equation checker with either a declared

decidable fragment or explicit proof obligations, and implement reviewed information-loss and schema-diff analyses.

Exit gate:

- compatibility fixtures across at least two IR versions;
- fuzz and mutation reports;
- expanded Rust/Haskell properties with adversarial multi-fault cases;
- exact Lean theorem coverage and audited translations; and
- no semantic claim based only on normalizing a supplied result.

11.2 R2: compiled artifact

Introduce a validated source value and a compiled schema value with local numeric interning. Bind every compiled value to content digests and a compiler revision. Measure memory, serialization, lookup, and compile time against the current string-keyed IR.

Exit gate: deterministic compilation, stale-artifact rejection fixtures, and published raw size and timing observations.

11.3 R3: SQLite reference backend

SQLite is a useful first backend because it supports embedded evaluation and a well-documented planner. The implementation should lower a deliberately small logical subset, preserve bound parameters, use explicit transactions, and compare every result with a direct native SQL baseline.

Exit gate:

- schema creation and reflection contract;
- typed projection, selection, and join lowering;
- transaction failure and rollback tests;
- collation and null-semantics matrix;
- `EXPLAIN QUERY PLAN` capture; and
- differential row and provenance results.

11.4 R4: PostgreSQL and capability negotiation

PostgreSQL should not be treated as "SQLite over a socket." Its type, collation, isolation, prepared-statement, extension, and planner behavior deserve an explicit capability profile. The connector should prove that every pushed expression lies inside that profile.

Exit gate: dialect and capability fixtures, transaction/isolation tests, prepared-plan invalidation, and differential results against native SQL.

11.5 R5: planner, cache, and async executor

Only after two backends expose real differences should a planner choose among pushdown, local execution, or materialization. Add a complete plan key and stale-plan tests before claiming cache reuse. Add asynchronous execution after cancellation and backpressure semantics are specified.

Exit gate: forced alternative plans with equal results, cache invalidation counterexamples, deterministic fault injection, and resource-bound tests.

11.6 R6: deployment and replay

Embedded, server, and WASM profiles should be separately supported. Operation replay begins with an operation algebra that states idempotence, order, authority, and conflict rules; deterministic fixture ordering is not a replay system.

Exit gate: target build matrix, API compatibility, crash/restart traces, operation permutations, and linkage to the distributed consistency papers.

12 Benchmark and evaluation contract

12.1 Semantic compilation

Measure decode, validation, normalization, mapping composition, finite instance validation, Δ, and compilation separately. Independent variables should include:

- objects, arrows, attributes, and equations;
- path length and equation-fragment complexity;
- mapping fan-out and composition depth;
- carrier cardinality and assignment count; and
- valid versus invalid inputs, including diagnostic position.

Report median, tail, distribution, and allocation behavior. Do not infer asymptotic complexity from the 33 golden fixtures.

12.2 Backend execution

For each backend query, compare:

1. direct native SQL;
2. CatDB-compiled SQL with semantic checks disabled after compilation;
3. cold CatDB compilation plus execution;
4. warm cached plan execution;
5. local residual processing;

6. materialized execution; and
7. provenance-enabled execution.

All alternatives must bind the same logical query, schema/mapping revisions, scalar semantics, and result contract. Native baselines should use equivalent indexes and parameters. Query text, physical plan, data generator, result digest, and raw timing data must be retained.

12.3 Falsification

The "outside the hot path" hypothesis is falsified if steady-state execution requires categorical interpretation per row or if semantic indirection adds material overhead that cannot be compiled or amortized for the target workload. The broader performance claim is falsified if CatDB changes results, omits required provenance, compares different transaction guarantees, or hides compilation and cache-maintenance costs.

12.4 Suggested matrix

Question	Baseline	Metric	Required evidence
IR compactness	current JSON/string IR	bytes, allocations, decode/lookup time	raw profiles and artifact digests
validation scale	schema, path, and mapping functions	time and peak memory by size	generators and raw samples
Δ scale	direct reference evaluator	time and memory by carrier/path size	equal result digests
compile overhead	direct SQL preparation	cold/warm latency	compile and cache-key traces
native preservation	hand-written SQL	latency, throughput, plan shape	equal semantics and indexes
connector overhead	native driver	CPU, allocations, transfer bytes	same transaction and result
provenance cost	provenance-off CatDB	latency, bytes, storage	same rows plus declared evidence
cancellation	native cancellation	cleanup time and leaked resources	fault-injection trace
WASM feasibility	native embedded kernel	size, startup, supported fixtures	host and feature matrix

13 Claim register

ID	Claim	Status	Basis
P15-C01	The workspace has six CatDB crates.	COMPUTATIONAL	manifest audit
P15-C02	Named IR records reject unknown fields.	COMPUTATIONAL	source/tests
P15-C03	Schema/path validation covers Slice 1 fixtures.	COMPUTATIONAL	fixture run
P15-C04	General equational reasoning is complete.	REJECTED AS WRITTEN	direct checker
P15-C05	Mapping identity and composition exist in Rust.	COMPUTATIONAL	source/tests
P15-C06	Selected abstract path/mapping laws are checked.	MACHINE-CHECKED	Slice 3
P15-C07	Lean verifies the Rust implementation.	REJECTED AS WRITTEN	no bridge
P15-C08	Finite instance validation exists for the profile.	COMPUTATIONAL	Slice 4
P15-C09	Finite pullback Δ exists for the profile.	COMPUTATIONAL	13 fixtures
P15-C10	Sigma and Pi are implemented.	REJECTED AS WRITTEN	unsupported
P15-C11	Rust/Haskell agree on 29 semantic fixtures.	COMPUTATIONAL	fresh compare
P15-C12	Rust/Haskell are generally equivalent.	REJECTED AS WRITTEN	finite corpus
P15-C13	An async executor exists.	OBSTRUCTED	absent source
P15-C14	SQLite/PostgreSQL connectors exist.	OBSTRUCTED	absent source
P15-C15	A planner and plan cache exist.	OBSTRUCTED	absent source
P15-C16	A server exists.	OBSTRUCTED	absent source
P15-C17	WASM deployment is supported.	OBSTRUCTED	no build evidence
P15-C18	Deterministic operation replay exists.	OBSTRUCTED	no operation engine
P15-C19	Native performance is preserved.	OBSTRUCTED	no benchmark
P15-C20	Safe Rust limits classes of memory error.	SUPPORTED BY PRIOR LITERATURE	Rust literature
P15-C21	First-party crate source has no unsafe block.	COMPUTATIONAL	source search
P15-C22	Whole-program memory safety is proved.	REJECTED AS WRITTEN	no such proof
P15-C23	Semantics can stay outside the hot path.	ENGINEERING HYPOTHESIS	future benchmark
P15-C24	Immutable compiled schemas improve reuse.	ENGINEERING HYPOTHESIS	future artifact
P15-C25	CatDB is production-ready.	REJECTED AS WRITTEN	research kernel

14 Limitations and threats to validity

14.1 Small, curated corpus

The accepted thirty-three fixtures are enough to make a semantic boundary executable, not enough to characterize a database runtime. The corpus is curated and hand-computable. It does not sample large schemas, deep equations, hostile documents, high-cardinality instances, concurrent use, or backend behavior.

14.2 Non-independent requirements

The repository authors designed the fixture contract and implementations. The Haskell process is independently coded but follows the same written contract. Shared specification errors remain possible. External paper review has not run.

14.3 Separate formal model

Lean coverage is exact but narrow. Without a verified translation or refinement proof, a theorem about the Lean mapping representation cannot be silently transferred to Serde decoding, Rust diagnostics, or the migration evaluator.

14.4 Snapshot drift

This paper records one accepted commit. Other agents were working in the repository during the broader program, so the live working tree is not an evidence substitute. Paper 15 review and reproduction must remain pinned to the Slice 4 acceptance commit.

14.5 No security or performance study

No threat model, fuzz campaign, supply-chain audit, memory profile, or backend benchmark is reported. The absence of first-party unsafe blocks is neither a security certification nor a performance result.

15 Open questions

1. Which decidable equation fragment offers sufficient modeling power without making compilation unpredictable?
2. How should source-level stable identifiers relate to compiled numeric IDs across schema versions?
3. Can a small logical operator set represent categorical obligations without embedding a theorem prover in the executor?
4. Which semantic properties should be proven during compilation and which should remain runtime checks?

5. Can a refinement bridge relate Lean lawful mappings to serialized Rust values without formalizing Serde or the entire engine?
6. When should CatDB reuse DataFusion or native database optimizers instead of building a custom executor?
7. How should capability negotiation represent collation, nulls, decimal rounding, time zones, and approximate vector semantics?
8. What cache dependencies are sufficient for safe plan reuse?
9. Can cancellation and provenance be made compositional across connector boundaries?
10. Which WASM profile is useful when native database drivers are absent?
11. How should deterministic local computation interact with distributed operation ordering and reconciliation?
12. What workload would falsify the claim that semantic compilation is outside the hot path?

16 Conclusion

CatDB does not yet have the production Rust runtime described by its full program. It has something smaller and worth preserving: a six-crate semantic kernel with a closed fixture contract, exact diagnostics, finite schema and mapping operations, finite instance validation, and bounded pullback migration. On the accepted 33-case corpus, Rust computes 29 semantic outcomes, and an independent Haskell process agrees on all 29. Selected path and mapping laws are machine-checked in a separate Lean representation.

The boundary is as important as the positive result. The equation checker is incomplete. Sigma and Pi are unsupported. Information loss and schema diff are represented but not computed at the accepted boundary. No query planner, connector, transaction layer, async executor, server, WASM deployment result, replay engine, or benchmark exists. Rust's language discipline does not prove CatDB semantics, and differential agreement does not prove general equivalence.

The next credible step is not to call the kernel production-ready. It is to stabilize and version the semantic contract, create a compiled artifact with complete dependency keys, add one transactional SQLite backend, compare compiled results with native SQL, and retain every negative case. Only then can the central systems hypothesis be tested: whether categorical semantics can guide compilation while remaining outside the hot execution path.

A Audited source digests

Artifact	SHA-256
`Cargo.toml`	`8d639b43fd8806b8b8a5341c86af626d95cc2bb68ed85d9074c1722caea40a34`
`Cargo.lock`	`117abb87d3ec22ebfd2ce0f97f80f373b72043601b79464fb259de192c2b9643`
`catdb-core/src/lib.rs`	`119cfdc5d4947b2b05f0bf4e41c842ed87f463717c2596bac1b55d306b3556cd`
`catdb-ir/src/lib.rs`	`adb3bc7c2fe83ad90a377b49b778de224c340ae65ee899f6353597fe5cb478cc`
`catdb-schema/src/lib.rs`	`e0560ee95f3de2a2b19936dd53e1f163b02b47c0300eb7fd58169cb58614cd84`
`catdb-mapping/src/lib.rs`	`8843a40f83b7e6aeb8d33f607e2cc0f30b41aa8be0fffe5e3ae5304df314db5a`
`catdb-migration/src/lib.rs`	`cd447b21bfeeb237930a323902e851aedecdf278deb2589260d3d285321c944d`
`catdb-cli/src/main.rs`	`84d661be4b186085b32bcb15a0d1b77f2b13b7b8a7781c2276aacaa1327abdb2`
`fixtures/manifest-v1.json`	`a817c292e879dcd8eb8f5ec2143dd0029464fde66925cdc8374ce9a599046f54`
`fixtures/schema-v1.schema.json`	`c00a67aa806c2d28f18c9ed2133870fd75c4494c82951124189d92937f48fba8`

B Fresh verification log digests

Log	SHA-256
fixture validator	`96ecc0c1bad6cab3dfe80329a5a3d835c48d57ee781dc5d3737f1eed8c4d4c79`
manifest audit	`57d0ab5c2024ca07085c148144bf81eaa67f781f7868c671a2897bad777b31fe`
Python tests	`d238d441d1595dacce99a6b5e49c2b9645e5e548f19d16f7a79b0f50b31cfc02`
Cargo format	`e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855`
Cargo Clippy	`ef28b05c6ac3fd38cfe705b3b871c8e9882901131ea76a7718a1eed898cadfbf`
Cargo tests	`8b3477cbb51c6ab66ee9b0da8d584bd8629d417b41961423b3edb8d922cf5519`
Rust Slice 1	`712ef20b795309ef1075d24285fa376f15089ae531b978b1db91c92278b7f36d`
Rust Slice 4	`1a88813b6fff79101fdb01bf81cc015cb4bf05b4b9c203c4c476d2b951a7fe06`

Log	SHA-256
cross-language Slice 4	`3b85deb889761c77f032a8af8c349d2194c4366453cc2fbd d4cfff19e9ddcc92`

C Review checklist

Before this manuscript can move beyond provisional status, the program requires:

- one categorical/database domain reviewer;
- one database-systems reviewer;
- one adversarial reviewer;
- one code reviewer tied to the paper's exact source claims;
- one reproducibility reviewer;
- one citation auditor;
- one human-readable editor;
- one mathematical typesetting reviewer; and
- a final build and page-by-page visual inspection at the accepted source commit.

No accepted PDF or cover is produced by this provisional workstream.

References

[1] AlgebraicJulia. Catlab.jl: Categorical algebra. https://algebraicjulia.github.io/Catlab.jl/latest/apis/categorical_algebra/, 2026. Accessed 23 July 2026.

[2] Apache Arrow Project. Arrow columnar format. https://arrow.apache.org/docs/format/Columnar.html, 2026. Accessed 23 July 2026.

[3] Apache DataFusion Project. Apache datafusion: Introduction and query optimizer. https://datafusion.apache.org/user-guide/introduction.html, 2026. Accessed 23 July 2026.

[4] Bytecode Alliance. Using the wasmtime api. https://docs.wasmtime.dev/lang.html, 2026. Accessed 23 July 2026.

[5] Categorical Data. Categorical query language. https://categoricaldata.net/, 2026. Accessed 23 July 2026.

[6] Goetz Graefe. Volcano—an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994. doi: 10.1109/69.273032. URL https://doi.org/10.1109/69.273032.

[7] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, 1995. URL https://www.sigmod.org/publications/dblp/db/journals/debu/Graefe95a.html.

[8] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):66:1–66:34, 2018. doi: 10.1145/3158154. URL https://doi.org/10.1145/3158154.

[9] PostgreSQL Global Development Group. Planner/optimizer. https://www.postgresql.org/docs/current/planner-optimizer.html, 2026. PostgreSQL documentation; accessed 23 July 2026.

[10] Rust Project. Workspaces. https://doc.rust-lang.org/cargo/reference/workspaces.html, 2026. The Cargo Book; accessed 23 July 2026.

[11] Rust Project. What is ownership? https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html, 2026. The Rust Programming Language; accessed 23 July 2026.

[12] Patrick Schultz and Ryan Wisnesky. Algebraic data integration. *Journal of Functional Programming*, 27:e24, 2017. doi: 10.1017/S0956796817000168. URL https://doi.org/10.1017/S0956796817000168.

[13] Patrick Schultz, David I. Spivak, Christina Vasilakopoulou, and Ryan Wisnesky. Algebraic databases, 2016. URL https://arxiv.org/abs/1602.03501.

[14] Serde Project. Container attributes. https://serde.rs/container-attrs.html, 2026. Accessed 23 July 2026.

[15] David I. Spivak. Functorial data migration. *Information and Computation*, 217:31–51, 2012. doi: 10.1016/j.ic.2012.05.001. URL https://doi.org/10.1016/j.ic.2012.05.001.

[16] David I. Spivak and Ryan Wisnesky. Relational foundations for functorial data migration. In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 21–28, 2015. doi: 10.1145/2815072.2815075. URL https://doi.org/10.1145/2815072.2815075.

[17] SQLite Project. Query planning. https://www.sqlite.org/queryplanner.html, 2026. Accessed 23 July 2026.

[18] Tokio Project. Tokio: An asynchronous runtime for rust. https://docs.rs/tokio/latest/tokio/, 2026. Accessed 23 July 2026.

[19] W3C WebAssembly Working Group. Webassembly core specification. https://www.w3.org/TR/wasm-core/, 2026. Accessed 23 July 2026.