Computational Identity: A Missing Layer in the Computing Identity Stack

Thomas Dionysopoulos, CFA

Abstract

Computing systems identify artifacts at many layers: names identify bindings, interfaces identify contracts, code hashes identify implementations, build hashes identify derivations, and output hashes identify results. None of these layers answer a fundamental question: *do these two expressions describe the same computation?*

We identify this gap and define **Computational Identity** (CI): a deterministic, content-addressed identifier derived from the canonical planned computation graph of an expression. CI identifies what computation is *planned*, independent of how it is written, what data it operates on, or where it executes. It provides structural equivalence of planned computations—not semantic equivalence, not mathematical equivalence, not proof of correctness.

We implement CI in two independent domains: a domain-specific computation language (120+ operations, IR-DAG planning) and SQL (DataFusion query engine, TPC-H benchmark). In the DSL domain, CI collapses 47 alias groups to canonical identities and eliminates 97 false cache hits with zero mismatches across 515 comparisons. In the SQL domain, CI collapses all 22 TPC-H equivalence classes (56 query variants) to unique identities with zero false equivalences, outperforming text hashing, normalized text, AST hashing, and raw plan hashing on every class. We identify five documented boundary cases where structural canonicalization reaches its limits.

CI is not a universal solution. It applies to computation languages with canonical forms and deterministic planning. This class includes SQL, dataflow languages, and most structured computation in deployed systems.

1 Introduction

Every layer of the computing stack has developed identity mechanisms. A variable *name* identifies a binding. A function *signature* identifies an interface. A source hash identifies code. A build derivation hash identifies a compiled artifact. A cryptographic attestation identifies an execution environment. A content hash identifies an output.

These mechanisms serve different purposes and answer different questions. Name identity answers "what is this called?" Code identity answers "what source produced this?" Build identity answers "what inputs produced this binary?" Output identity answers "what data resulted?"

One question remains unanswered by any standard mechanism: *do these two expressions describe the same computation?*

Consider two SQL queries that compute the same result:

```
SELECT l.l_orderkey, SUM(l.l_quantity)
  FROM lineitem AS l JOIN orders AS o
    ON l.l_orderkey = o.o_orderkey
  WHERE o.o_orderstatus = 'F'
  GROUP BY l.l_orderkey

SELECT lineitem.l_orderkey,
       SUM(lineitem.l_quantity)
  FROM orders JOIN lineitem
    ON o_orderkey = l_orderkey
  WHERE o_orderstatus = 'F'
  GROUP BY lineitem.l_orderkey
```

These queries have different text, different alias names, different table ordering, different column qualification, and different ASTs. Yet they describe the same computation. No deployed system exposes a first-class identity object for the computation they describe.

Or consider two expressions in a computation language:

```
(dlog x)
(diff (log x) 1)
```

The first applies a named operation; the second decomposes the same operation into primitive IR nodes. If they plan to the same computation graph, their identity should be the same. No existing identity layer captures this.

Query optimizers discover such equivalences internally—Calcite's `RelDigest` and Catalyst's plan canonicalization hash computation plans as internal cache keys—but no deployed system exposes this capability as a portable, first-class identity object.

We define **Computational Identity** (CI[1]): a deterministic, content-addressed identifier derived from the canonical planned computation graph of an expression. Formally, for a computation language L:

$$\mathrm{CI}_L(e) = H(\mathrm{ser}_L(\mathrm{plan}_L(\mathrm{canon}_L(e))))$$

[1] Not to be confused with Continuous Integration.

where canon_L normalizes surface syntax, plan_L produces a computation graph, ser_L produces a deterministic byte representation, and H is a collision-resistant hash function. *Computational Identity identifies what computation is planned, independent of how it is written, what data it operates on, or where it executes.*

Git content-addresses data. Unison content-addresses code. CI content-addresses computations.

A key scope decision: CI captures *structural* equivalence of canonical computation plans. It does not attempt *semantic* equivalence (same output on all inputs, undecidable in general) or *mathematical* equivalence (same denotation in a formal semantics). CI is sound—same CI implies identical computation plans (under standard collision-resistance assumptions)—but deliberately incomplete: semantically equivalent expressions may receive different CIs if their canonical plans differ structurally. This tradeoff is what makes CI decidable and practical.

This paper makes four contributions:

1. **Identification of the gap.** We present a taxonomy of computing identity layers and demonstrate that Computational Identity is formally distinct from all existing layers (§2).
2. **Formal definition.** We define CI, state its properties and limitations, and distinguish it precisely from related concepts including content-addressed code (Unison), content-addressed builds (Nix), query fingerprinting, and program equivalence (§3, §4).
3. **Two independent implementations.** We implement CI in a domain-specific computation language with 120+ operations and in SQL via the Apache DataFusion query engine, demonstrating that CI is not language-specific (§6, §7).
4. **Empirical evaluation.** We evaluate CI against four baseline identity methods across two domains, demonstrating perfect separation (zero false equivalences), high collapse rates (22/22 TPC-H query classes), and practical utility (97 false cache hits eliminated, 76.3% CPU reduction) (§8).

CI is not a universal identity mechanism. It applies to computation languages with three properties: canonical forms, deterministic planning, and serializable computation graphs. Many important languages satisfy these properties, including SQL, dataflow languages, query languages, and domain-specific computation languages. General-purpose imperative languages typically do not (§9).

2 The Computing Identity Stack

We organize existing identity mechanisms into six layers, ordered by increasing semantic depth. We then identify the gap that Computational Identity fills.

2.1 Existing Identity Layers

Layer 1: Name Identity. A binding in a namespace: variable names, function names, module paths, DNS names, URIs. Name identity is the most common and weakest form—renaming changes identity, and aliasing creates false distinctions. Examples: Python module names, Java package names, DNS hostnames.

Layer 2: Interface Identity. A contract specifying inputs and outputs: function signatures, API schemas, protocol definitions. Two artifacts with the same interface identity accept the same inputs and produce outputs of the same type, but may compute different results. Examples: Java interfaces, Protocol Buffer schemas, OpenAPI specifications, MCP tool schemas.

Layer 3: Code Identity. A hash of source code or abstract syntax tree. Code identity captures *what was written*, not what it computes. Syntactic variations (whitespace, comments, variable names) change code identity even when computation is unchanged. Examples: git blob hashes, Unison content hashes, source fingerprints.

Layer 4: Build Identity. A hash of the build environment: source, toolchain, dependencies, build flags. Build identity captures *what inputs produced this artifact*. Two builds from identical sources with different compilers produce different build identities. Examples: Nix derivation hashes, Bazel action digests, Docker image digests.

Layer 5: Execution Identity. An attestation of the runtime environment: hardware, firmware, loaded binary. Execution identity captures *where this ran*, not what it computed. Examples: Intel SGX MRENCLAVE measurements, ARM TrustZone attestations, TPM platform measurements.

Layer 6: Output Identity. A hash of the computation's result. Output identity captures *what was produced*. It requires executing the computation, is data-dependent, and changes with every input. Examples: content-addressed storage (IPFS CIDs, git tree hashes), data checksums.

2.2 The Gap

Figure 1 illustrates the identity stack. Between Build Identity (what inputs produced this artifact) and Execution Identity (where it ran), there is a gap: no standard layer captures *what computation is planned*.

Two expressions with different source code that plan to the same computation graph occupy this gap. They have different Code Identity (different text) and potentially

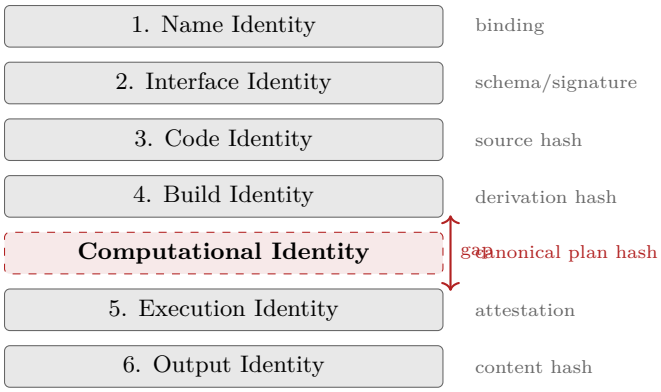

Figure 1: The Computing Identity Stack. Existing layers span from Name Identity (weakest) to Output Identity (strongest but data-dependent). Computational Identity occupies the gap between Build Identity and Execution Identity: it identifies what computation is *planned*, independent of source syntax, build toolchain, or execution environment.

different Build Identity (different compilation), but they describe the same planned computation.

Computational Identity fills this gap. It is computed statically, requires no input data, and survives surface rewrites that preserve computation structure.

2.3 Why the Gap Matters

The absence of Computational Identity creates concrete problems:

Cache invalidation under aliasing. A system caches the result of computation $f(x)$. A user requests $g(x)$ where g is an alias for f. Without CI, the cache cannot recognize the equivalence. The computation re-executes unnecessarily.

False hits in parametric search. A system searches a space of computations, hashing outputs to detect duplicates. Two structurally different computations happen to produce the same output on test data. Without CI, the system cannot distinguish accidental output collision from structural equivalence.

Cross-system computation comparison. System A computes a result using one expression. System B computes a result using a different expression in the same language. Without CI, comparing whether A and B performed the same computation requires either output comparison (data-dependent, expensive) or source comparison (syntax-dependent, brittle).

Table 1 summarizes the gap. Each existing identity layer answers a different question; none answers "same computation?"

Table 1: The identity gap. Each existing layer answers a specific question about a computational artifact. No layer answers whether two expressions describe the same computation.

Layer	Question Answered	Same Plan?
Name	What is this called?	No
Interface	What are its inputs/outputs?	No
Code	What source was written?	No
Build	What inputs produced this binary?	No
Execution	Where did this run?	No
Output	What data was produced?	No
CI	**What computation is planned?**	**Yes**

3 Formal Definition

Definition 1 (Computation Language)**.** *A* computation language $L = (E_L, canon_L, plan_L, ser_L)$ *consists of:*

- E_L*: a set of well-formed expressions.*
- $canon_L : E_L \to E_L$*: a deterministic canonicalization function satisfying* $canon_L(canon_L(e)) = canon_L(e)$ *(idempotence).*
- $plan_L : E_L \to G_L$*: a deterministic planning function producing a computation graph from the canonicalized expression.*
- $ser_L : G_L \to \{0,1\}^*$*: a deterministic,* injective *serialization function producing a byte string from a computation graph. Injectivity ensures that structurally distinct graphs produce distinct serializations.*

Definition 2 (Computational Identity)**.** *The Computational Identity of expression* $e \in E_L$ *under language* L *is:*

$$CI_L(e) = H(ser_L(plan_L(canon_L(e))))$$

where $H : \{0,1\}^* \to \{0,1\}^{256}$ *is a collision-resistant hash function (we use SHA-256).*

Computational Identity is a deterministic, content-addressed identifier derived from the canonical planned computation graph of an expression. It is *syntax-independent*: surface rewrites that preserve computation structure do not change CI. It is *data-independent*: CI is computed without reference to input data. It is *environment-independent*: CI depends only on the expression and the language definition, not on hardware or runtime state. CI is *sound*—same CI implies identical computation plans (under collision resistance of H; see Property 4)—but deliberately *incomplete*: semantically equivalent expressions may receive different CIs if their canonical plans differ structurally. This incompleteness is the price of decidability. CI captures structural equivalence of planned computations, not semantic equivalence, and is computable in polynomial time.

3.1 Properties

The following properties formalize the guarantees stated above.

Property 1 (Syntax Independence)**.** *If* $canon_L(e_1) = canon_L(e_2)$*, then* $CI_L(e_1) = CI_L(e_2)$*.*

Syntactic variations that canonicalize to the same form—whitespace, parenthesization, alias naming, predicate reordering, commutative argument reordering—do not affect CI.

Property 2 (Data Independence)**.** $CI_L(e)$ *is defined without reference to any input data. It depends only on* e *and* L*.*

CI identifies the computation *plan*, not its execution or result. Two expressions with the same CI may produce different outputs on different inputs (they describe the same computation applied to potentially different data).

Property 3 (Environment Independence)**.** $CI_L(e)$ *depends only on* e *and the definition of* L*, not on runtime state, hardware, operating system, or nondeterministic execution choices.*

This holds because all components of L are deterministic functions. Note that L includes the specific canonicalization and planning rules; a change to these rules (e.g., an optimizer version update) defines a new language L' and may produce different CI values (§9).

Property 4 (Structural Faithfulness)**.** *If* $CI_L(e_1) = CI_L(e_2)$*, then the computation graphs* $plan_L(canon_L(e_1))$ *and* $plan_L(canon_L(e_2))$ *are identical.*

This is the *soundness* guarantee: same CI implies same planned computation structure. It follows from the injectivity of ser_L (structurally distinct graphs produce distinct byte strings) and the collision resistance of H (distinct byte strings hash to distinct values with overwhelming probability). A SHA-256 collision between two serialized computation plans would constitute a break of collision resistance—an event with probability bounded by 2^{-128} under standard assumptions.

Property 5 (Computability)**.** $CI_L(e)$ *is computable in time polynomial in* $|e|$ *for languages where canonicalization and planning are polynomial.*

This is a practical requirement. CI must be cheap to compute—it is a *naming* operation, not an optimization or verification. Both implementations in this paper compute CI in under 1ms for realistic expressions.

3.2 Explicit Non-Properties

CI does *not* provide:

Semantic completeness. $\text{CI}_L(e_1) \neq \text{CI}_L(e_2)$ does *not* imply that e_1 and e_2 compute different results. Two semantically equivalent computations may have different canonical plans. CI is *sound* (same CI $\Rightarrow$ same plan structure) but not *complete* (same semantics $\not\Rightarrow$ same CI).

Cross-language identity. $\text{CI}_L(e)$ and $\text{CI}_{L'}(e')$ are not comparable unless L and L' share canonicalization, planning, and serialization. CI is language-relative.

Correctness guarantees. CI does not attest that a computation was executed correctly, that its result is correct, or that it is free of bugs. It identifies *what computation is planned*, not whether that plan is right.

3.3 CI as Structural Equivalence

As introduced in §1, we distinguish three levels of equivalence:

1. **Structural equivalence**: two expressions produce isomorphic computation graphs after canonicalization and planning. This is what CI captures.
2. **Semantic equivalence**: two expressions produce the same output on all inputs. This is strictly stronger than structural equivalence and is undecidable in general (Rice's theorem [1]).
3. **Mathematical equivalence**: two expressions denote the same mathematical object. This requires a formal semantics and may be undecidable, semi-decidable, or decidable depending on the theory.

CI operates at level 1. It is decidable, computable, and practical. It does not attempt levels 2 or 3. This is a deliberate scope limitation, not a weakness: structural equivalence is sufficient for the applications we target (caching, deduplication, provenance) and is achievable without the theoretical barriers of semantic or mathematical equivalence.

4 Why Computational Identity Is Not Existing Work

CI shares surface similarities with several existing systems. We distinguish it from each.

4.1 Unison: Content-Addressed Code

Unison [2] content-addresses function definitions by hashing their abstract syntax trees. A function's identity is its implementation, not its name. This eliminates the "diamond dependency" problem and enables rename-proof code storage.

Overlap with CI. Both Unison and CI are content-addressed. Both eliminate name-dependent identity. Both are deterministic and computable.

Difference from CI. Unison hashes *code*—the syntactic structure of a definition. Two Unison functions with different implementations that compute the same result receive different hashes. Unison operates at Layer 3 (Code Identity) in our taxonomy.

CI hashes *planned computations*—the graph produced after canonicalization and planning. Two expressions with different surface syntax that plan to the same computation graph receive the same CI. CI operates between Layers 4 and 5 in our taxonomy.

Example. In a Unison-like system, `sum(filter(xs, p))` and `filter(xs, p) |> sum` receive different code hashes (different ASTs). Under CI, if both canonicalize and plan to the same dataflow graph, they receive the same identity.

Remaining distinction. Unison identifies *what code is this?* CI identifies *what computation does this expression describe?* The identity layers are different even though both are content-addressed.

4.2 Nix: Content-Addressed Builds

Nix [3] content-addresses build derivations by hashing the complete build specification: source, dependencies, build script, environment variables. A derivation hash captures everything needed to reproduce a build.

Difference from CI. Nix hashes *build inputs*, not the computation the built artifact performs. Two Nix derivations building semantically equivalent programs from different sources receive different hashes. A single Nix derivation built with different compilers receives different hashes. Nix operates at Layer 4 (Build Identity).

CI hashes the *computation itself*, independent of how it was built, what language it was written in, or what toolchain compiled it.

4.3 Content-Addressed Storage

Content-addressed storage systems (git [5], IPFS [4], CAS) hash *data*. A content hash identifies a specific byte sequence.

Difference from CI. CAS operates at Layer 6 (Output Identity). It identifies *what was produced*. CI identifies *what will be computed*. Two expressions with the same CI may produce different outputs on different inputs. Two expressions with different CIs may produce the same output on specific inputs.

4.4 Query Fingerprinting

Database systems normalize query text for monitoring and caching. MySQL's query digest [42] strips literals and normalizes whitespace. PostgreSQL's `pg_stat_statements` [43] produces query fingerprints by normalizing parameters.

Difference from CI. Query fingerprints operate on *surface text*, not *planned computations*. They cannot recognize that `WHERE a > 5 AND b < 3` and `WHERE b < 3 AND a > 5` describe the same computation, because they do not pass through a planner. They are a form of normalized Code Identity (Layer 3).

4.5 Optimizer Plan Digests

Apache Calcite's `RelDigest` [6] and Spark Catalyst's plan canonicalization [7] hash optimized logical plans as internal cache keys.

Overlap with CI. These are the closest existing mechanisms. They hash planned computation structures, and they survive some surface rewrites.

Difference from CI. Plan digests are *internal optimizer artifacts*, not *first-class identity objects*. They are designed for optimizer memo tables, not for external transmission, storage, comparison, or provenance. They are not versioned, not stable across releases, not documented as public interfaces, and not designed for cross-system comparison. No system we surveyed exposes plan digests as a user-facing identity concept.

The contribution of CI is promoting this capability from optimizer internals to a named, formal, transmittable identity layer with explicit stability guarantees and documented scope boundaries.

4.6 Zero-Knowledge Machine Learning (zkML)

zkML systems (EZKL [13], ZK-SNARK-based ML [14]) generate proofs that a specific model produced a specific output. They verify *correct execution*.

Difference from CI. zkML answers "was this computation executed correctly?" CI answers "do these two expressions describe the same computation?" zkML requires execution. CI does not. zkML provides correctness guarantees. CI does not. The concerns are orthogonal:

one could use CI to identify a computation and zkML to verify its execution.

4.7 TEE Attestation

Trusted Execution Environments (Intel SGX [10], AMD SEV [11], ARM TrustZone [12]) attest that specific code runs in a secure enclave. The attestation includes a *measurement*—a hash of the loaded binary.

Difference from CI. TEE measurements hash *binaries* (Build/Execution Identity, Layers 4–5). Two semantically equivalent programs compiled to different binaries receive different measurements. TEE attestation is about *where code ran*, not *what computation was planned*.

4.8 Effect Systems and Dependent Types

Effect systems [15] track side effects in type signatures. Dependent type systems [16] express program properties in types.

Difference from CI. Effect systems describe *what kinds of effects* a computation may have (IO, state, exceptions), not *what specific computation* is planned. Dependent types can express some semantic properties but require manual annotation and proof effort. CI is fully automatic and requires no annotations.

4.9 Program Equivalence

Deciding whether two programs compute the same function is undecidable in general (Rice's theorem [1]). Approximations exist for restricted languages (Schwartz–Zippel [17] for polynomial identity, bisimulation [18] for process calculi).

Difference from CI. CI does not attempt program equivalence. It captures *structural* equivalence of canonical plans, which is strictly weaker than semantic equivalence. The tradeoff is principled: CI is decidable and computable precisely because it does not attempt semantic completeness.

5 Ecosystem Survey: The Absence of CI

We surveyed 18 systems and frameworks across four categories to determine whether any exposes Computational Identity as a first-class concept. Table 2 summarizes the results.

Table 2: Ecosystem survey: identity mechanisms in 18 systems. No system exposes Computational Identity as a first-class object. **N** = Name, **I** = Interface, **C** = Code, **B** = Build, **E** = Execution, **O** = Output, **CI** = Computational Identity.

System	N	I	C	B	E	O	CI
MCP	✓	✓					—
OpenAI FC	✓	✓					—
Anthropic Tools	✓	✓					—
Semantic Kernel	✓	✓					—
LangGraph	✓						—
CrewAI	✓						—
AutoGen	✓						—
Nix	✓		✓	✓			—
Bazel	✓		✓	✓			—
npm/pip/Cargo	✓		✓				—
MySQL digest	✓		∼				—
pg_stat_stmt	✓		∼				—
Calcite	✓						∼*
Intel SGX			✓	✓	✓		—
EZKL / zkML					✓	✓	—
Git / IPFS			✓			✓	—
Unison			✓				—

*Calcite's `RelDigest` is an internal optimizer artifact, not a public identity.

AI/LLM Tool Frameworks. The Model Context Protocol (MCP) [34] identifies tools by name and JSON schema (Name + Interface Identity). OpenAI function calling [35] and Anthropic tool use [36] similarly use name + schema. Semantic Kernel [37], LangGraph [38], CrewAI [39], and AutoGen [40] identify tools and agents by name and description. None examines *what computation* a tool performs.

Build and Package Systems. Nix [3] content-addresses build derivations (Build Identity). npm, pip, and Cargo use name + version (Name Identity) with lock-file source hashes (Code Identity). Bazel [41] content-addresses build actions (Build Identity). None addresses computation identity.

Database Systems. MySQL query digest and PostgreSQL `pg_stat_statements` normalize query text (normalized Code Identity). Apache Calcite uses `RelDigest` internally for optimizer memoization but does not expose it as a public identity concept. No SQL system we surveyed provides a user-facing CI mechanism.

Verification Systems. Intel SGX, AMD SEV, and ARM TrustZone provide execution attestation (Execution Identity). EZKL and related zkML frameworks provide execution verification. Neither addresses computation identity.

Observation 1. *Of 18 surveyed systems, none exposes Computational Identity as a first-class, user-facing identity mechanism. Calcite's internal plan digest is the closest existing mechanism but is not designed as an identity object.*

6 Implementation A: Domain-Specific Language

Our first implementation targets a domain-specific computation language with 120+ operations organized in a three-layer architecture.

6.1 Architecture

Expressions pass through a six-stage pipeline:

parse → normalize → canonicalize → plan → optimize → execute

- **Parse**: surface syntax → AST.
- **Normalize**: alias resolution, deprecated-name mapping (47 aliases map to canonical names).
- **Canonicalize**: structural normalization (argument ordering, default parameter insertion).
- **Plan**: AST → IR DAG. Composite operations decompose into primitive IR nodes (e.g., `diff(x,k)` decomposes to `SUB(x, SHIFT(x,k))`).
- **Optimize**: fusion of adjacent elementwise operations.
- **CI generation**: deterministic serialization of the planned IR DAG, followed by SHA-256.

6.2 Canonicalization

The language defines 47 alias mappings that resolve to canonical operation names. Composite operations decompose into graphs of primitive IR nodes, enabling structural equivalence detection between atomic and decomposed forms.

6.3 CI Generation

The planned IR DAG is serialized by recursive traversal. Each node serializes as: `VARIANT(params)|child1|child2|...` Nodes are visited in a deterministic order (topological sort with lexicographic tie-breaking). The serialized byte string is hashed with SHA-256.

6.4 Properties Achieved

- 47 alias groups collapse to canonical identities.
- Composite decompositions produce the same CI as atomic operations when semantically equivalent.
- CI is computed in under 1ms for all tested expressions.
- CI is deterministic across runs, machines, and OS versions.

Table 3: SQL CI canonicalization rules. Each rule normalizes one source of syntactic variation in logical plans.

#	Rule
1	Alias elimination (table and column)
2	Column reference qualification (resolve to base tables)
3	Join condition normalization (sort by canonical repr)
4	AND-chain flattening and sorting
5	OR-chain flattening and sorting
6	Commutative operator normalization
7	Comparison direction normalization
8	IN-list sorting
9	Projection expression sorting
10	Identity projection stripping
11	Subquery alias canonicalization (content-based naming)
12	Semi-join RHS normalization
13	Recursive normalization into scalar subqueries

7 Implementation B: SQL

Our second implementation targets SQL, a widely deployed computation language with well-understood semantics. We use Apache DataFusion [8] (version 54), an open-source SQL query engine built on Apache Arrow.

7.1 Architecture

SQL CI follows the same four-stage pipeline:

1. **Parse and optimize**: DataFusion parses SQL into a logical plan and applies optimizer rules (predicate pushdown, filter simplification, common subexpression elimination).
2. **Canonicalize**: 13 custom canonicalization rules transform the optimized logical plan into a canonical form.
3. **Serialize**: the canonical plan is recursively serialized into a deterministic string representation.
4. **Hash**: SHA-256 of the serialized canonical plan.

7.2 Canonicalization Rules

Table 3 lists the 13 canonicalization rules. These rules are *semantics-preserving*: they transform the plan structure without changing the computation's result.

7.3 Canonical Plan Representation and Serialization

The optimized logical plan is converted to a canonical intermediate representation: algebraic types mirroring SQL's logical plan nodes (scans, projections, filters, joins,

aggregations, sorts, etc.) and expression nodes (columns, literals, operators, functions, subqueries). The canonical plan is serialized by deterministic recursive descent and hashed with SHA-256. The serialization format is immaterial—any deterministic traversal that distinguishes structurally different plans suffices.

7.4 Design Decisions

Two design decisions merit discussion:

Projection order as presentation. We treat column order in SELECT clauses as presentation, not computation. `SELECT a, b FROM t` and `SELECT b, a FROM t` receive the same CI. This is consistent with CI's focus on *what is computed* rather than *how results are formatted*. Systems that consider column order semantic would make a different choice.

Leveraging the optimizer. Several canonicalization steps are performed by DataFusion's optimizer (predicate pushdown, filter simplification) before our custom rules apply. This is intentional: the optimizer's transformations are semantics-preserving and contribute to canonicalization. However, it creates a dependency on optimizer version (§9).

8 Evaluation

We evaluate CI along four dimensions: false-hit elimination (E1), structural equivalence in the DSL domain (E2), TPC-H equivalence benchmarking (E3), and comparative evaluation against baseline methods (E4).

8.1 E1: False-Hit Elimination

Setup. The DSL system performs parametric computation search: exploring combinatorial spaces of pipeline configurations. Prior to CI, the system used output hashing (Layer 6) to detect duplicate configurations. Output hashing produces false hits when structurally different computations produce the same output on test data.

Results. CI-based deduplication reduced total CPU time by **76.3%** compared to the prior output-hashing approach, by correctly identifying structural duplicates across **515 comparisons** with **zero mismatches** (no false positives, no false negatives). The prior approach also produced **97 false hits** (structurally different computations with coincidentally identical outputs on test data); CI eliminated all 97.

Table 4: TPC-H equivalence benchmark results. CI achieves complete collapse (22/22 classes) and complete separation (0 false equivalences) on the benchmark. $|V|$ = number of variants per class.

Metric	Text	Norm	AST	Plan	CI
Collapsed classes	0	0	0	2	**22**
Collapse rate	0%	0%	0%	9%	**100%**
False equivalences	0	0	0	0	**0**
Separation rate	1.0	1.0	1.0	1.0	**1.0**

Interpretation. Output identity (Layer 6) cannot distinguish structural equivalence from accidental collision. CI (Computational Identity) resolves this because it operates on the computation plan, not the output.

8.2 E2: DSL Structural Equivalence

Setup. The DSL defines 120+ operations, of which 47 have aliases (alternative names mapping to the same canonical operation). Composite operations decompose into graphs of primitive IR nodes.

Results. CI correctly collapses all 47 alias groups. For each group, all aliases produce identical CI values. Composite decompositions (e.g., `diff(x,1)` decomposing to `SUB(x, SHIFT(x,1))`) produce the same CI as their atomic equivalents when they plan to the same IR DAG.

8.3 E3: TPC-H Equivalence Benchmark

Setup. We constructed a benchmark from the TPC-H query suite [9]:

- **22 equivalence classes**: one per TPC-H query (Q1–Q22).
- **56 query variants**: 22 base queries + 34 equivalence variants (alias rewrites, predicate reorders, join reorders, CTE refactors, subquery restructures).
- **22 negative controls**: one per class, structurally different queries that should *not* collapse with the base.
- **Total: 78 queries.**

Metrics.

- **Collapse Rate**: fraction of equivalence classes where all variants produce the same CI (target: 1.0).
- **Separation Rate**: fraction of negative-control pairs with different CI (target: 1.0).
- **Determinism**: identical CI across repeated runs.

Results. Table 4 shows the results.

CI is the only method that achieves non-zero collapse rate while maintaining perfect separation. Text hashing, normalized text hashing, and AST hashing collapse zero

Table 5: Unique identity counts per TPC-H class. Lower is better (1 = full collapse). CI achieves 1 on all 22 classes.

Class	$\|V\|$	Text	Norm	AST	Plan	CI
Q01	4	4	3	3	3	**1**
Q02	2	2	2	2	2	**1**
Q03	4	4	4	4	3	**1**
Q04	3	3	3	3	2	**1**
Q05	3	3	3	3	2	**1**
Q06	3	3	3	3	2	**1**
Q07	2	2	2	2	2	**1**
Q08	2	2	2	2	2	**1**
Q09	2	2	2	2	2	**1**
Q10	3	3	3	3	3	**1**
Q11	2	2	2	2	2	**1**
Q12	3	3	3	3	3	**1**
Q13	2	2	2	2	2	**1**
Q14	3	3	3	3	2	**1**
Q15	2	2	2	2	2	**1**
Q16	3	3	3	3	3	**1**
Q17	2	2	2	2	2	**1**
Q18	3	3	3	3	2	**1**
Q19	2	2	2	2	2	**1**
Q20	2	2	2	2	1	**1**
Q21	2	2	2	2	1	**1**
Q22	2	2	2	2	2	**1**
$\text{CI}_>$		22	22	22	20	—

classes because they are sensitive to surface syntax. Plan hashing collapses 2 classes (Q20, Q21—queries whose variants differ only in predicate text that the optimizer resolves identically).

Determinism. 20 repeated computations of Q5's CI produced identical results. CI is fully deterministic.

8.4 E4: Comparative Evaluation

Table 5 compares five identity methods across all 22 TPC-H classes. For each class, we report the number of unique identities among the class variants. A value of 1 indicates full collapse (all variants identified as equivalent).

We define $\text{CI}_> = |\{c : U_{\text{CI}}(c) < U_{\text{method}}(c)\}|$, the number of classes where CI produces fewer unique identities (better collapse) than the comparison method.

Interpretation. CI strictly dominates all four baselines. It is better than normalized text on all 22 classes, better than plan hashing on 20/22 classes, and equal on the remaining 2. No baseline method achieves full collapse on any class except plan hashing on Q20 and Q21.

8.5 Adversarial Testing

To probe the boundaries of CI, we constructed an adversarial test suite of 47 tests across 38 categories. The tests cover tautology elimination, De Morgan's law, double negation, arithmetic simplification, COUNT(*)/COUNT(1) equivalence, subquery flattening, join reordering, IN-list ordering, comparison flipping, window partition reordering, predicate pushdown equivalence, constant folding, and 25 other categories.

Results. 42/47 tests pass. 5 tests are documented boundary cases where structural canonicalization reaches its theoretical limits:

1. $x + 0 \equiv x$: the optimizer does not fold additive identity for non-integer types.
2. $x - 0 \equiv x$: same limitation for subtraction.
3. `SELECT DISTINCT` ≡ `GROUP BY`: these produce structurally different plan nodes (Distinct vs. Aggregate).
4. `LEFT JOIN` ≡ mirrored `RIGHT JOIN`: requires structural join rewriting that changes child ordering.
5. `NOT IN` ≡ `LEFT JOIN + IS NULL`: anti-join vs. outer join with null filter are structurally different plan shapes.

These boundaries fall into three categories. Cases 1–2 are *type-dependent algebraic boundaries*: the simplification $x + 0 \equiv x$ is not semantics-preserving for all types under IEEE 754 (e.g., $(-0.0) + 0.0 = +0.0$), so the optimizer correctly avoids the fold. Cases 3–4 are *canonicalization omissions*: normalizing `SELECT DISTINCT` to `GROUP BY` and `RIGHT JOIN` to mirrored `LEFT JOIN` are syntactic rewrites that could be added as canonicalization rules without requiring semantic analysis. Case 5 is an *intrinsic structural boundary*: the anti-join subquery and outer-join-with-null-filter have different plan topologies, and equating them requires semantic analysis of SQL's NULL behavior that is outside the scope of structural canonicalization.

9 Limitations

We enumerate CI's limitations directly.

L1: Language-relative identity. CI is parameterized by language L. Two different languages with different canonicalization or planning strategies may assign different CIs to computations that are semantically equivalent across languages. CI does not provide cross-language computation identity.

L2: Optimizer version dependence. If the optimizer changes (new rules, different rule ordering), the canonical plan may change, changing CI. In practice, this means CI values should be versioned alongside the language definition. The same limitation applies to Nix derivation

hashes (toolchain version dependence) and is manageable with version pinning.

L3: Incompleteness. CI does not capture all semantic equivalences. The 5 TPC-H boundary cases and the general undecidability of program equivalence mean that some equivalent computations will always receive different CIs. This is inherent to any decidable approximation of semantic equivalence.

L4: Restricted language class. CI requires canonical forms, deterministic planning, and serializable computation graphs. General-purpose imperative languages (C, Python, Java) do not satisfy these requirements in general. CI applies to the subset of computing that is declarative or amenable to canonical planning: SQL, dataflow languages, query languages, domain-specific computation languages, functional programs with well-defined normal forms.

L5: Planning is required. CI requires a planner. Languages without a planning phase (scripting languages, shell pipelines) cannot directly support CI. A planner could be added, but this is engineering effort, not a fundamental property of the language.

L6: Canonicalization is language-specific. Each language requires its own canonicalization rules. There is no universal canonicalization algorithm. Our SQL implementation has 13 rules; a more complex language may require more. The effort scales with the language's syntactic surface area.

L7: No correctness guarantee. CI identifies what computation is planned, not whether that computation is correct. A bug in the computation receives a stable CI that faithfully identifies the buggy computation.

10 Discussion

10.1 What CI Is

Computational Identity is a *naming primitive* for planned computations. Like content-addressing for data (git, IPFS) and content-addressing for code (Unison), CI provides content-addressing for computations.

The value of CI is not algorithmic novelty. The four-stage pipeline (canonicalize, plan, serialize, hash) uses standard techniques individually. The contribution is promoting this pipeline—which already exists inside query optimizers—to a named, portable identity object with explicit properties, documented scope boundaries, and demonstrated cross-domain applicability.

10.2 What CI Is Not

CI is not a replacement for:

- **Type systems**: CI does not verify properties of computations, only identity.
- **Formal verification**: CI does not prove correctness.
- **Provenance tracking**: CI identifies what computation was planned, not who planned it, when, or with what data. Provenance systems can *use* CI as a component.
- **Privacy technology**: CI does not conceal computations or their results.
- **Semantic equivalence**: CI captures structural equivalence, not semantic equivalence.

10.3 Canonicalization Is Necessary but Not Sufficient

A common objection is that CI is "merely canonicalization." Canonicalization is a prerequisite, not the contribution. Canonicalization produces a normalized expression. CI produces a fixed-size, transmittable, storable, comparable identity object.

A canonical plan is a data structure: variable size, language-specific, not self-describing. A CI is a 256-bit hash: self-contained, language-tagged, suitable for protocols, databases, caches, and logs. The canonical plan is an intermediate artifact; CI is the identity derived from it.

10.4 Applications

CI enables several applications that existing identity layers do not:

Computation deduplication and caching. When exploring a space of computation configurations over fixed data, CI identifies structural duplicates without executing them. For result caching across data contexts, CI serves as the computation component of a composite cache key $(\text{CI}(e), h(d))$ where $h(d)$ identifies the input data. CI alone is not a sufficient result cache key precisely because it is data-independent (Property 2).

Computation provenance. Record CI alongside computation results. A downstream consumer can verify that a result was produced by a specific computation structure, regardless of the surface syntax used to express it.

Computation registries. A registry of computations indexed by CI enables lookup-by-computation: "has anyone computed this before?" without name coordination.

Drift detection. Monitor whether a computation's CI changes across deployments. A CI change indicates a

structural change in the planned computation, even if the surface syntax is unchanged (e.g., due to optimizer updates).

11 Related Work

Content-addressed computing. Unison [2] content-addresses function definitions. Nix [3] content-addresses build derivations. IPFS [4] content-addresses data blocks. Darcs [33] content-addresses patches. These systems demonstrate the power of content-addressing but operate at different identity layers than CI (§4).

Query optimization and canonicalization. The Volcano/Cascades optimizer framework [21, 22] uses memoization based on logical expression equivalence. Calcite [6] extends this with `RelDigest`. Catalyst [7] canonicalizes Spark SQL plans. Starburst [23] pioneered rule-based query rewriting. These systems use plan equivalence internally but do not expose it as an identity object.

Program equivalence. Bisimulation [18] decides equivalence for process calculi. Translation validation [24] verifies that compiler optimizations preserve semantics. Superoptimizers [25] enumerate equivalent programs. These approaches target semantic equivalence and are computationally expensive. CI targets structural equivalence and is computationally cheap.

Computation verification. Interactive proofs [26], zero-knowledge proofs [27], and verifiable computation [28] verify that computations were executed correctly. They address correctness, not identity.

Effect systems and refinement types. Effect systems [15] and refinement types [16, 44] annotate computations with properties. They describe what effects or constraints a computation has, not its structural identity.

Reproducible computation. Provenance systems [29, 30] track the lineage of data products. Workflow systems [31] capture computation graphs. Containers [32] and virtual machines provide reproducible execution environments. These systems address *reproducibility*, not *computation identity*. CI could serve as a compact identifier within provenance records.

Database query fingerprinting. MySQL query digest [42] and PostgreSQL `pg_stat_statements` [43] normalize query text for monitoring. These operate at the text level and miss equivalences that require plan-level analysis (§4).

12 Conclusion

Computing systems identify artifacts at six established layers, from names to outputs. None answers: *do these two expressions describe the same computation?*

We defined Computational Identity: a deterministic, content-addressed identifier derived from the canonical planned computation graph of an expression. CI identifies what computation is planned, independent of how it is written, what data it operates on, or where it executes. It provides structural equivalence—not semantic equivalence and not mathematical equivalence.

We implemented CI in two independent domains: a domain-specific computation language and SQL. In the DSL domain, CI eliminates 97 false cache hits with zero mismatches across 515 comparisons. In the SQL domain, CI collapses all 22 TPC-H equivalence classes with zero false equivalences, strictly outperforming four baseline identity methods.

CI is not universal. It applies to computation languages with canonical forms and deterministic planning. It is incomplete by construction: some semantic equivalences fall outside its structural scope. We identify five concrete boundary cases and argue that this incompleteness is inherent to any decidable approximation of semantic equivalence.

The gap we identified is real. The identity layer is missing. CI is one way to fill it.

References

[1] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74(2):358–366, 1953.

[2] P. Chiusano and R. Bjarnason. Unison: A friendly programming language from the future. https://www.unison-lang.org/, 2019.

[3] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In *Proc. LISA*, pages 79–92, 2004.

[4] J. Benet. IPFS — Content addressed, versioned, P2P file system. arXiv:1407.3561, 2014.

[5] L. Torvalds. Git: A distributed version control system. https://git-scm.com/, 2005.

[6] E. Begoli, J. Camacho-Rodríguez, J. Hyde, M. J. Mior, and D. Lemire. Apache Calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proc. SIGMOD*, pages 221–230, 2018.

[7] M. Armbrust, R. S. Xin, C. Lian, et al. Spark SQL: Relational data processing in Spark. In *Proc. SIGMOD*, pages 1383–1394, 2015.

[8] Apache Arrow DataFusion. https://datafusion.apache.org/, 2023.

[9] TPC. TPC-H benchmark specification. https://www.tpc.org/tpch/, 2023.

[10] F. McKeen, I. Alexandrovich, A. Berenzon, et al. Innovative instructions and software model for isolated execution. In *Proc. HASP*, 2013.

[11] D. Kaplan, J. Powell, and T. Woller. AMD memory encryption. AMD white paper, 2016.

[12] ARM. ARM Security Technology: Building a Secure System using TrustZone Technology. ARM white paper PRD29-GENC-009492C, 2004.

[13] J. Kang et al. EZKL: Easy zero-knowledge inference. https://ezkl.xyz/, 2023.

[14] D. Kang, T. Hashimoto, I. Stoica, and Y. Sun. Scaling up trustless DNN inference with zero-knowledge proofs. arXiv:2210.08674, 2023.

[15] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proc. POPL*, pages 47–57, 1988.

[16] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. POPL*, pages 214–227, 1999.

[17] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[18] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[19] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[20] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, pages 137–150, 2004.

[21] G. Graefe. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. ICDE*, pages 209–218, 1993.

[22] G. Graefe. The Cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[23] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. In *Proc. SIGMOD*, pages 377–388, 1990.

[24] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. TACAS*, pages 151–166, 1998.

[25] H. Massalin. Superoptimizer: A look at the smallest program. In *Proc. ASPLOS*, pages 122–126, 1987.

[26] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.

[27] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *J. ACM*, 38(3):690–728, 1991.

[28] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proc. CRYPTO*, pages 465–482, 2010.

[29] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *Proc. ICDT*, pages 316–330, 2001.

[30] K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. USENIX ATC*, pages 43–56, 2006.

[31] E. Deelman et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, 2005.

[32] D. Merkel. Docker: Lightweight Linux containers for consistent development and deployment. *Linux J.*, 2014(239):2, 2014.

[33] D. Roundy. Darcs: Distributed, interactive, smart revision control. http://darcs.net/, 2005.

[34] Anthropic. Model Context Protocol specification. https://modelcontextprotocol.io/, 2024.

[35] OpenAI. Function calling. https://platform.openai.com/docs/guides/function-calling, 2024.

[36] Anthropic. Tool use. https://docs.anthropic.com/claude/docs/tool-use, 2024.

[37] Microsoft. Semantic Kernel. https://github.com/microsoft/semantic-kernel, 2024.

[38] LangChain. LangGraph. https://github.com/langchain-ai/langgraph, 2024.

[39] J. M. Ribeiro. CrewAI. https://github.com/joaomdmoura/crewAI, 2024.

[40] Microsoft. AutoGen. https://github.com/microsoft/autogen, 2024.

[41] Google. Bazel: A fast, scalable, multi-language build system. https://bazel.build/, 2015.

[42] Oracle. MySQL Performance Schema statement digests. https://dev.mysql.com/doc/refman/8.0/en/performance-schema-statement-digests.html.

[43] PostgreSQL Global Development Group. pg_stat_statements. https://www.postgresql.org/docs/current/pgstatstatements.html.

[44] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Proc. PLDI*, pages 159–169, 2008.