Konstruct: Column-Level Lineage Checked Against the Data

Formulas, verification by perturbation and an ontology for the data stack

Kodex Labs

https://github.com/kodexlabsio/konstruct Konstruct 0.13.22, MIT licence

Abstract

A number on a dashboard is a claim about a chain of SQL, orchestration code and business intent that nobody can read end to end. Lineage tools draw that chain but assert it; they do not check it, and they rarely say how a value is calculated or what it means. Konstruct is an open-source, local-first system that reads a data repository and its warehouse (dbt, SQL, Airflow, Dagster, PySpark, Looker, Power BI, Fivetran and Airbyte) and builds one graph of every relation, column, task, test, dashboard and business term, with column-level lineage parsed statically from the SQL that connects them. It composes each column's expression through every model upstream into a formula over source columns, derives an ontology of entities, measures and relationships from the code and its tests, reads and writes Apache Ossie semantic models, and gives coding agents the graph through an MCP server and a gate that a change must pass. Its central idea is that lineage should be measured, not asserted: `konstruct verify` perturbs each input column of each model, reruns the model's SQL and compares what changed with what the graph claims, so that a missing edge shows up as a dependency the graph does not explain, reproducible from the recorded seed. On dbt Labs' jaffle-shop all 87 of 87 claimed column edges are confirmed with 0 unexplained influences; on the bundled example, of 4 verified edges whose input is not also a join, filter or grouping column, each deleted in turn, verification reports 100% as unexplained; 19 of the 19 elements of the Apache Ossie portability fixtures in a form the specification allows survive a round trip unchanged; and a cold build of 5,000 generated models with 129,940 column edges takes 61.2 s on 4 CPUs. Every measurement of Konstruct in this paper is regenerated from the code by continuous integration.

1 Introduction

Every organisation runs on numbers it cannot fully explain. A revenue figure on a dashboard is the end of a chain: a loader lands rows from a source system, dbt models clean and join them, an Airflow task copies the result into a reporting table, and a semantic model in a business intelligence tool aggregates it. Each link is code, written by different people in different tools, and the meaning of the final number (which rows count, in which currency, net of what) lives partly in that code and partly in people's heads. When the number is questioned, someone reads the chain by hand.

The cost of that is measured, if imperfectly. Gartner puts the average cost of poor data quality at USD 12.9 million a year per organisation, from research published in 2020 and based on self-estimated costs [17]. In a 2024 vendor survey of over 550 respondents, 67% said they do not completely trust the data they use for decisions [36]. In a 2023 vendor survey of 200 data professionals, a data incident took 15 hours on average to resolve, up 166% on the year before [32]. dbt Labs' 2026 vendor survey of 363 data professionals found 71% concerned about incorrect data reaching stakeholders and 41% naming ambiguous data ownership as an obstacle [12]. Software developers, in a 2018 field study, spent 58% of their time on program comprehension [43], and a data stack is a program spread across many tools. Regulators ask for the chain explicitly: in 2023, ten years after the Basel Committee's principles for risk data aggregation, only 2 of 31 global systemically important banks were fully compliant with all of them [3], and Article 10 of the EU Artificial Intelligence Act of 2024 requires the data of high-risk systems to be subject to data governance covering its origin and preparation [16].

AI agents raise the stakes. An agent that writes SQL against a warehouse it does not understand produces confident, plausible and wrong numbers. On the enterprise questions of Sequeda et al. (2023), GPT-4 answered 16.7% correctly from the SQL schema alone and 54.2% when the same questions were posed over a knowledge graph of the business [38]. On Spider 2.0 (2025), which asks for real enterprise workflows, the o1-preview agent solves 21.3% of tasks, against 91.2% on Spider 1.0 and 73.0% on BIRD for GPT-4-based methods [23]; on BIRD itself (2023), GPT-4 reached 54.89% execution accuracy against 92.96% for people [24]. A 2026 vendor study reports 64.5% accuracy for text-to-SQL on a raw normalised schema, 84.1% to 90.0% for text-to-SQL on the modelled project, and 98.2% to 100% through a semantic layer over it [11]. The lesson of these results is consistent: agents are only as good as the facts about the data they are grounded in. Those facts are lineage, formulas, definitions and relationships, and today they are mostly asserted by tools that never check them.

This paper describes Konstruct, an MIT-licensed system that makes those facts explicit and checkable. It reads a repository and its warehouse locally, in one command, with no server, no account and no telemetry, and writes a graph that people and agents query from a command line, a Python API, an MCP server, a GitHub Action and a single offline HTML page. Its contributions are:

1. **One graph across tools.** A typed graph model (section 3.1) that joins column-level lineage from dbt, standalone SQL, Airflow and Dagster tasks, PySpark jobs, warehouse views, Looker and Power BI semantic models and the tables Fivetran and Airbyte land, through DuckDB, Postgres, Snowflake, BigQuery, Databricks,

Trino and Iceberg, without importing or running any user code (section 3.4).

2. **Formulas, not just arrows.** Canonical column expressions over physical columns and their composition through every model upstream, so a column reads as a formula over source columns, with a text composition that is exact where the SQL printer is operand-insensitive and falls back to trees where it is not (sections 3.2 and 3.3).
3. **Lineage measured against the data.** A verification procedure by perturbation that confirms claimed edges and exposes missing ones with concrete witnesses, with a soundness argument and an honest account of what it cannot see (section 4).
4. **Meaning derived, not declared.** An ontology of entities, measures and relationships derived from keys, tests and joins, bound to approved business definitions, with ontology packs for common source schemas and round trips of Apache Ossie semantic models that return what was authored (sections 3.6 to 3.8).
5. **Grounding for agents.** An MCP interface whose answers carry lineage, formulas and verification state, and a gate that a change must pass before it is proposed (section 3.9).
6. **A paper regenerated from the code.** Every measurement of Konstruct here is produced by `paper/generate.py` from the code at the commit that built this PDF, and a test fails when the paper types a number outside a cited sentence or leaves out a command, tool or format (section 5).

2 Related Work

Data provenance. Database research distinguishes *why*-provenance (which input tuples justify an output), *where*-provenance (which input locations a value was copied from) and *how*-provenance (how inputs were combined) [4, 6]; provenance semirings unify how- and why-provenance as annotations computed alongside a query [20]. Lineage tracing for data warehouses goes back to Cui, Widom and Wiener [7, 8], and systems such as Perm compute provenance by rewriting queries [18]. These are mostly *dynamic*: they annotate data as queries run. Müller and Grust derive where- and why-provenance for SQL by a value-less abstract interpretation after recording the query's control flow [33], which is closest in spirit to Konstruct's static analysis. Cheney, Ahmed and Acar cast provenance as dependency analysis: an output depends on an input if changing the input can change the output [5]. That definition is the one `konstruct verify` tests. Perturbation is an established tool for explaining query results: Walenz and Yang analyse how a query's answers respond to perturbations of its parameters [41], and counterfactual causality and influence-based explanations remove or change tuples to see what an answer depends on [29, 42]. Konstruct applies the idea to a different question: not why a particular answer is what it is, but whether a static lineage claim about a query holds on the data.

Column lineage tools. OpenLineage is an open standard for lineage events emitted by running jobs, with column-level lineage as a dataset facet, and Marquez is its reference implementation [27, 34]. SQLLineage and sqlglot's lineage module trace columns through a SQL statement and run locally; sqlglot's lineage nodes carry the expression of each step [21, 26], and Konstruct builds on sqlglot. SQLMesh, also built on sqlglot, derives column-level lineage locally from its own understanding of a project's SQL [39]. dbt Explorer offers column-level lineage for dbt projects in dbt Labs' hosted service [13], and metadata platforms such as DataHub and OpenMetadata collect lineage from many systems into a central service [10, 35]. Konstruct's emphasis is different in three ways. It joins lineage across dbt, orchestration code, dashboards and loaders in one graph built from a repository with no service. It treats lineage as a claim to be tested against the data rather than a fact to be displayed. And it composes each column's expression through every model upstream, so lineage carries formulas and conditions, not only edges. It writes OpenLineage events, so it complements rather than replaces these systems.

Semantic layers and Apache Ossie. Semantic layers such as the dbt Semantic Layer with MetricFlow and Looker's LookML define metrics once over physical tables [14, 19]. Apache Ossie, formerly the Open Semantic Interchange, is a vendor-neutral YAML specification for such models (datasets, fields, relationships and metrics), with a portability ladder that grades how well a converter preserves each element [2]. The specification describes what a metric is, but not where its inputs come from, who owns it or whether its lineage holds. Konstruct reads Ossie models as an input and writes its ontology as one, adding exactly those facts in a custom extension (section 3.7).

Knowledge graphs and text-to-SQL. Benchmarks of language models on databases show a large gap between toy and enterprise settings. Spider measured cross-domain text-to-SQL [44]; BIRD added large, dirty databases and external knowledge [24]; Spider 2.0 moved to enterprise workflows across dialects and dbt projects [23]. Sequeda et al. showed that a knowledge graph of the business more than triples GPT-4's accuracy on enterprise questions [38]. Konstruct produces such a graph from the code that already exists, with entities, measures, relationships and verified formulas, and serves it to agents through the Model Context Protocol [31].

Static analysis of orchestration code. Airflow and Dagster discover pipelines by importing user Python modules [1, 9]. Importing executes arbitrary code at analysis time, which is unacceptable in a review tool that runs on every pull request. Konstruct parses DAG and asset files with Python's `ast` module and never imports them; its parsers are tested for parity against Airflow's own DagBag and Dagster's own asset and job graphs on the fixtures in its test suite.

Testing analyses with data. Differential testing compares two implementations on generated inputs [28], and has found many bugs in database engines [37]; property-based testing generates the inputs [25]. Mutation testing measures a test's power by seeding faults and counting those it detects [22]. Konstruct uses all three to test its own verifier (section 5).

3 System Design

Konstruct reads a repository and the warehouses its dbt profiles name, and writes `konstruct-out/graph.json`, `REPORT.md`, `graph.html` and `ontology.yml`. Three rules shape it. It never imports or executes user orchestration code: DAG, Dagster and PySpark files are parsed with `ast`, dbt

Jinja is rendered in a sandbox, and `konstruct verify` runs a model's SQL only on copies of its inputs or inside a transaction that is rolled back; on DuckDB copies it turns DuckDB's external access and extension autoloading off, so a model that reads files or URLs directly is skipped. The one exception is dbt itself: by default, when a dbt project's artefacts are missing or stale, `konstruct build` runs `dbt docs generate`, which runs the project's macros and on-run hooks with its own profile, and those can write to the warehouse; with `--no-dbt`, the GitHub Action's default, dbt never runs. A repository's own warehouse connections, the paths it names outside the checkout and running dbt are used only once the user has reviewed the project and run `konstruct trust`, which records a digest of those settings outside the repository, so that any change to them needs trust again. Apart from that, it reads warehouses for metadata only, makes no network requests of its own beyond the warehouses it is configured to read, and collects no telemetry; when it runs dbt, it turns dbt's anonymous usage statistics off. And everything it claims about lineage is tested, including against data (section 4).

3.1 The Graph Model

The graph is a typed, directed multigraph $G = (V, E)$. Each node $v \in V$ has a kind $\mathit{kind}(v)$ among *source*, *table*, *model*, *seed*, *snapshot*, *column*, *dag*, *task*, *test*, *exposure* and *term*; each edge $(u, v, k) \in E$ has a kind k among *contains*, *feeds*, *derived_from*, *runs*, *precedes*, *tests*, *defines* and *relates_to*, and carries attributes. Let $\mathscr{R} \subseteq V$ be the relations (sources, tables, models, seeds and snapshots) and $C \subseteq V$ the columns, each with a parent relation $p(c) \in \mathscr{R}$. Two relations are at the heart of the model:

$$\mathscr{D} = \{(c, c', \tau) : c \text{ is derived from } c'\} \subseteq C \times C \times \Theta, \quad (1)$$

$$\mathscr{F} = \{(c', r) : r \text{ joins, filters or groups on } c'\} \subseteq C \times \mathscr{R}. \quad (2)$$

The transform $\tau \in \Theta = \{\mathit{direct}, \mathit{transformed}\}$ is *direct* when the path from c' to c is made only of column references and renames, and *transformed* otherwise. $\mathscr{D}$ says where a value comes from; $\mathscr{F}$ says which columns decide the rows a relation returns, and is stored as *feeds* edges with the role *reference*. Keeping the two apart matters: a filter column changes which rows a model returns, but marking every downstream column as derived from it would bury the answer to "what does this column depend on".

Each relation has a canonical key $\mathrm{rel}(r)$, its lower-case `catalog.schema.name`. Nodes from different tools that describe one physical relation (a dbt model, the warehouse table it materialises, the target of an Airflow SQL task, an Iceberg table) share a key, and SQL that names the key resolves to one of them by a fixed priority, so a chain of derivations crosses tools without any configuration. `impact` and `trace` are the forward and backward closures over $\mathscr{D}$, $\mathscr{F}$ and the task edges, reported with the transform at each step and the business entities the walk reaches; `path` is a shortest path between two nodes in either direction.

3.2 SQL Column Lineage and Canonical Expressions

For each statement, Konstruct qualifies every column against the columns it knows: those from the dbt catalog or the warehouse, and those of upstream models analysed earlier in dependency order, so `SELECT *` expands. It then follows each output column through CTEs, subqueries, aliases, unions, window functions and nested fields to physical columns with sqlglot's lineage [26], in the dialect of the dbt adapter, one of the 15 dialects that adapters map to. `CREATE ... AS`, `INSERT ... SELECT`, `MERGE`, `UPDATE`, `DELETE` and `COPY` give the relation they write.

Alongside the edges, Konstruct keeps each output column's *canonical expression*: its SQL with every column replaced, recursively, by the canonical expression of what it reads, until only physical columns remain. For a lineage node n,

$$\kappa(n) = \begin{cases} \mathrm{rel}(t).a & \text{column } a \text{ of table } t, \\ \mathrm{op}\big(\mathrm{sort}\,\kappa(\mathrm{ch}(n))\big) & \text{set operation}, \\ e_n\big[x \mapsto \langle \kappa(\mathrm{ch}_x(n)) \rangle\big] & \text{otherwise}, \end{cases} \quad (3)$$

where e_n is the node's expression, $\mathrm{ch}_x(n)$ the child that column x resolves to, and $\langle \cdot \rangle$ brackets an inlined expression unless it is a name. The join, filter, grouping, `HAVING` and `QUALIFY` conditions of each scope are kept in the same form. Because CTE names, aliases, quoting, formatting and column order disappear in κ, two versions of a model that differ only in those have equal canonical expressions. `konstruct diff` relies on this: it reports a change only when a column's canonical expression or a condition changes, and a removed column as breaking only when something downstream still reads it. A model whose Jinja has not been compiled is rendered by Konstruct's own sandboxed renderer, with `ref`, `source`, `var`, the project's and packages' macros and `adapter.dispatch`; macros that would query the warehouse return empty results.

3.3 Formula Composition

A canonical expression stops at the boundary of a model. `konstruct explain` composes it through every model and task upstream. For a column c with canonical expression $e(c)$ and inputs $\mathrm{in}(c) = \{c' : (c, c', \tau) \in \mathscr{D}\}$, the composed formula is

$$\varphi(c) = \begin{cases} c & \text{if } \mathrm{in}(c) = \emptyset, \\ e(c)\big[c' \mapsto \langle \varphi(c') \rangle\big]_{c' \in \mathrm{in}(c)} & \text{otherwise}, \end{cases} \quad (4)$$

a source column standing for itself. In the bundled example, `customer_ltv.lifetime_value` reads as `customers.customer_lifetime_value`, across an Airflow SQL task, a dbt mart and a staging model. Composition is memoised, so each column is composed once however many columns read it; a cycle, a depth beyond 25 columns or a formula longer than 4,000 characters stops substitution and leaves the column as a reference, and the formula is then marked incomplete. The budget is checked before substituting, since a formula can grow quickly when a column reads several inputs that each compose to long formulas.

φ describes values, not rows. Substitution is exact for the expression, but the joins, filters and grouping of every step decide which rows reach it, and folding them into a single expression would be wrong for aggregates. `explain` therefore lists each step with its conditions beside the formula, and a test evaluates composed formulas of chains without

Algorithm 1 Composing a column's formula as text

Input: column c, memo M

if $c \in M$ **then return** $M[c]$
$t \leftarrow \text{parse}(e(c))$; $S \leftarrow \emptyset$
for each column reference x in t resolving to $c' \in \text{in}(c)$ **do**
    $s \leftarrow$ ComposeText(c')
    if the parent of x, past brackets, is not in T **then return** ComposeTree(c)
    replace x by a fresh token k; $S[k] \leftarrow s$
decide, for each k, whether $S[k]$ needs brackets in its parent; drop redundant ones
$p \leftarrow \text{print}(t)$ in the dialect
if a token is printed other than exactly once **then return** ComposeTree(c)
$M[c] \leftarrow p$ with each token k replaced by $S[k]$
return $M[c]$

joins, filters or aggregation directly on the source tables and requires them to give the final model's values row for row.

Formulas in plain English. With `--english`, `explain` also words $\varphi(c)$ and the conditions of each step by a fixed grammar over the expression tree. Each node type has one wording and `NULL` is *no value*; a test holds only where it is true, so a negated test names what it needs (*x has a value other than 'a'*), and a `NOT` the grammar cannot word that way is quoted. Columns are named in square brackets, an operand that is itself an operation is bracketed unless it continues a chain of the same operator, and what a dialect leaves implicit and changes the result, such as where an ordering puts missing values or whether division of whole numbers is whole, is stated, so formulas that compute different things never share a sentence. A node with no wording is quoted as SQL rather than paraphrased. Tests check each wording against its SQL, evaluate the negated tests on rows with and without values, check that no sentence names a literal or a column absent from its formula, and print random formulas in several dialects, parse them back and check that formulas which differ never share a sentence.

Composition as text with operand-sensitive contexts. The viewer shows every column's composed formula, so a build composes all of them. Building each $\varphi(c)$ as a tree copies and prints ever larger trees. Konstruct instead prints each column's own expression once, with each input replaced by a placeholder token, and splices the composed texts of the inputs into the printed string, adding the brackets the composed tree would keep. This is only correct where the printer's output for a node does not depend on what its operand is: BigQuery prints `MOD` differently by operand, T-SQL rewrites a non-boolean condition as a comparison with zero, and DuckDB's string functions and `INTERVAL` and `ANY` look at their arguments. Konstruct keeps an explicit set T of 32 node kinds whose printing is operand-insensitive in every dialect (arithmetic, comparisons, `COALESCE`, `CASE`, the core aggregates, `CAST`, `IN`, `LIKE`, window specifications, field access and a few others), and uses text only when every input sits in such a context:

The invariant is that Algorithm 1 returns exactly the string that printing the tidied tree of equation (4) would. It holds by construction where it uses text: a node in T prints its operands' strings unchanged, and the bracket decision uses the kind of node an input's formula starts with (before and after its own brackets are tidied), which is what the tree printer looks at. Elsewhere it falls back to the tree. The test suite checks the invariant on every column of a built project and on each input shape in each context in several dialects.

3.4 One Chain Across Tools

Every input contributes nodes and edges to the same graph, keyed so that the same relation is one node:

- **dbt**: models, seeds, snapshots, sources, tests, constraints and exposures from the manifest and catalog, with compiled SQL for lineage, or SQL rendered without dbt.
- **SQL**: standalone files and warehouse view definitions.
- **Airflow**: DAGs, TaskFlow, TaskGroups and every dependency syntax, parsed with `ast`; dbt commands in Bash, Cosmos and dbt operators resolved to models with dbt's selection syntax; inline and file SQL with column lineage; inlet and outlet assets.
- **Dagster**: assets, multi-assets and dbt assets with their selections, op jobs (nested graphs, aliases, fan-in, dynamic mapping), schedules and sensors, with column lineage from the SQL an asset or op runs.
- **PySpark**: the SQL a job passes to a Spark session, in the Spark dialect.
- **Dashboards**: dbt exposures, Looker views field by field from LookML, and Power BI semantic models from TMDL, with DAX references resolved across tables.
- **Loaders**: the tables Fivetran and Airbyte land, as sync tasks per schema, and column lineage from each Fivetran source system when its Platform Connector's tables are in the warehouse.
- **Warehouses**: DuckDB, Postgres, Snowflake, BigQuery, Databricks, Trino and Iceberg through any PyIceberg catalogue, read for tables, columns, types, comments, keys and views.
- **Meaning**: `definitions.yml`, SKOS and OWL vocabularies [30, 40], Apache Ossie models, and column usage from the query history of 3 warehouses.

A column computed in dbt, copied into a reporting table by an Airflow SQL task, written by a Dagster asset and read by a Looker field is therefore one chain, and so is a chain across several repositories that share a warehouse.

3.5 Builds, Caching and Parallelism

Analysing SQL dominates a build. Each analysis is cached under a content key; for a dbt model m,

$$k(m) = \text{SHA256}\big(sql_m,\ d,\ \text{rel}(m),\ S(\text{up}(m)),\ checksum_m\big), \tag{5}$$

where d is the dialect and $S(\text{up}(m))$ the columns of the models m reads. Because the key holds the upstream schemas, an edit that changes a model's columns invalidates exactly the analyses that could see the change. `konstruct build --update` reuses every other analysis and recomputes only the formulas downstream of the change, but it still reads every input, checks every key and writes every output, so its cost grows with the project as well as with the change

(section 5.4). A cold build analyses dbt models a level of the DAG at a time in worker processes, one per CPU up to 8, when at least 64 models are ready, with results identical to analysing them in order.

3.6 The Ontology

Konstruct derives an ontology from the graph, written to `ontology.yml` on every build and shown by `konstruct ontology`. Most of it is read from code and tests rather than declared:

Entities. A column k is an *entity key* when its name is identifier-like (`customer_id`, `order_key`, `id`) and a dbt `unique` test vouches for it, or when it is a single-column primary key (a dbt constraint, a warehouse primary key, an Iceberg identifier field, an Ossie primary key). The entity is named after the key, or after the relation for a bare `id`, and every relation keyed by it holds one row per instance. A term of kind *entity* names one explicitly, and a measure's grain ("one row per store") finds the key of the entity it names.

Relationships. A dbt `relationships` test, an Ossie relationship, or a join that equates a column of one entity's relation with another entity's key gives a relationship. Its cardinality is claimed only on evidence:

$$\mathrm{card}(a \to b) = \begin{cases} \text{one-to-one} & u(a) \wedge u(b), \\ \text{many-to-one} & u(b) \wedge \neg u(a), \\ \text{unknown} & \text{otherwise}, \end{cases} \tag{6}$$

where $u(x)$ holds when a test or key shows x unique.

Measures. A business term of kind *measure*, an Ossie metric, or a term bound to computed columns is a measure. Each bound column carries $\varphi(c)$, the sources it reaches, how many lineage edges on the way `verify` confirmed, and its usage when query history is read.

Confidence. Every entity, relationship and measure carries the strongest confidence its evidence supports: *declared* when a test, a constraint, a definition or an Ossie model states it, *inferred* when only names, a grain, a join or a pack binding show it, and *verified* when `konstruct verify` confirms it against the warehouse, where each entity key is read over its whole table for repeated and null values and each relationship for values of its referring column that the key lacks. A failed check never raises a confidence, so a join that the data contradicts stays inferred. `konstruct entity`, `konstruct metric` and `konstruct relate` work through this model one question at a time: an entity with its properties, relationships and metrics; each metric with its formula, grain, filters, owner and status, from the ontology, dbt's semantic layer, Looker and Power BI, and, where nothing defines one, a proposal read from the code (an aggregate column with a grain, a numeric column named as a measure, an entity count), which is always inferred and never approved; and the shortest, most trusted path of relationships between a pair of entities, whose confidence is the weakest on the way, with its composed cardinality and the joins that follow it. `konstruct compile-metric` compiles a metric, grouped by entities, properties and time grains and filtered by conditions on properties, to SQL without running it: from the relation whose rows hold the metric, it joins only along relationships the project shows many-to-one or one-to-one, onto keys a test, a primary key or a check against the data shows unique, refuses to add up a value its lineage shows copied from another grain, adds every filter the metric's definition requires, and lists each fact the SQL rests on with its confidence, the result being as trusted as the weakest; a question that would need a join repeating rows, or a link `konstruct verify` found contradicted, is refused, naming the link.

Rules. The build holds the ontology to 5 rules; a violation is a warning, so `konstruct build --strict` fails on it. The strongest is *one formula per measure*: every column bound to a measure must compute it the same way, compared as `konstruct conflicts` compares columns, by composed formula, sources, filters and joins on the route back to source, and the aggregation at the top. The others require entity keys to be unique, entity names to be distinct, bound columns to exist and approved terms to have an owner and an approval date.

3.7 Apache Ossie Round Trips

Konstruct reads every `*.ossie.yaml` file, in the released shape, with models in a list, and the newer draft, with one model at the root: a dataset becomes the relation its source names, a field the source columns its expression reads, a metric a measure, a primary key an entity key and a relationship a relationship. Anything that does not resolve is a build warning, including a model-scoped metric that names a field without its dataset, which the specification does not allow. `konstruct export ossie` writes the ontology as one model. Writing is *authored-preserving*: a dataset read from an Ossie model is written back as authored, and so is each metric whose datasets keep their names, with a `KONSTRUCT` custom extension added that holds the metric's status, owner, approval date, entity, and each bound column's formula, sources, verified edges and usage. Every other metric is derived from its composed formula, with the portable expression in ANSI SQL and the warehouse dialect's expression beside it. If R reads a model and W writes one, the property tested on Ossie's own portability ladder is

$$W(R(M)) = M \oplus x \tag{7}$$

for every model M whose metrics name their fields as the specification requires, where $\oplus x$ adds only the extension. Section 5.3 grades each element.

3.8 Ontology Packs

Many teams load the same sources. An ontology pack is a definitions file for one source schema with a match block; Konstruct ships 274 (table 1). A schema σ matches a pack P when it has every table P requires with its required columns, and there is evidence that it comes from the source: its name fits one of the pack's patterns, or a loader landed its tables. The second condition keeps an application's own `orders` and `customers` tables from being taken for Shopify's. A match binds the pack's terms to the schema's columns as drafts, leaving out terms whose tables or columns are missing, and the project's own terms win over a pack's.

3.9 Interfaces and the Agent Gate

The same questions are answered everywhere. The command line has 37 commands: `konstruct init`, `konstruct build`, `konstruct status` and `konstruct open` set

up, build, check and show the graph, and `konstruct trust` trusts a project's own settings; `konstruct impact`, `konstruct trace`, `konstruct path`, `konstruct search`, `konstruct describe`, `konstruct list` and `konstruct query` walk and find it; `konstruct explain` and `konstruct history` give a column's formula and when it changed; `konstruct verify` checks lineage against data, and with `--mutate` whether a model's dbt tests fail on small changes to its SQL, and with `--why` the source rows behind one value with each row's share, with `--effect` what each formula change in a range of commits does on today's data, and with `--laws` the equalities between totals that each model's SQL implies, checked per day; `konstruct release-notes` lists the changes to formulas, filters and business terms in that range, worded in plain English, with what each affects and whether its terms were approved after it; `konstruct timeline` builds the ontology at each commit of a range from that commit's own files and lists what each commit added, removed, renamed or changed in its entities, relationships, properties and metrics, with their trust and approvals; `konstruct diff` and `konstruct gate` review changes, and `konstruct lock` pins, in a lockfile, what each upstream column a project reads means and fails when that changes; `konstruct receipts` records what an answer rests on (each column's composed formula, row conditions, sources, bound terms and verification state) and later reports the answers whose basis has changed; `konstruct define`, `konstruct ontology`, `konstruct entity`, `konstruct metric`, `konstruct relate` and `konstruct conflicts` answer what things mean, `konstruct compile-metric` writes the SQL of a metric asked for by entities and properties, and `konstruct check-sql` checks a query against them without running it; `konstruct sensitivity` follows columns that a business term, an ontology pack or a warehouse tag or masking policy marks sensitive through the column lineage, hashes included, until every read of them is a configured sanitiser or an aggregate over groups the SQL requires to hold at least k distinct values of a foreign key to the people it is about (a count of rows, or of the input's own key, bounds rows, not people), and `konstruct erasure` lists the relations that hold one person's rows by following a key through copies, hashed copies, foreign keys and joins; `konstruct export` writes OpenLineage events (`openlineage`), Mermaid (`mermaid`), Graphviz (`dot`), CSV (`csv`), Apache Ossie (`ossie`), and the ontology as OWL in Turtle (`owl`), a Snowflake semantic view (`semantic-view`) and Databricks metric views (`metric-view`); `konstruct plugins` lists the installed plugins, which add inputs, export formats, gate checks, ontology packs and MCP tools, and whether each runs; and `konstruct install`, `konstruct mcp` and `konstruct hook` connect coding agents. The Python API, `konstruct.Project`, returns typed results for each question, and `graph.html` is one offline file with the lineage, formulas, terms, verification results and usage. `konstruct serve` sends the same viewer from a local server, which reloads open pages when a build replaces the graph, beside a read-only HTTP API, described by an OpenAPI document, whose answers come from the Python API's own functions. With `--meaning`, `konstruct search` also finds a node by what it is rather than what it is called: identifiers are read as words, with the abbreviations and synonyms of data work, and each node is scored, with no trained model, on the words, formula tokens and lineage it shares with the question.

Grounding agents. `konstruct mcp` runs a Model Context Protocol server with 29 tools: `impact`, `trace`, `path`, `search`, `describe`, `query`, `explain`, `history`, `define`, `conflicts`, `ontology`, `entity`, `metric`, `relate`, `compile_metric`, `sensitivity`, `erasure`, `diff`, `gate`, `check_sql`, `check_history`, `release_notes`, `timeline`, `mutation_test`, `why`, `definition_effect`, `receipt`, `laws` and `lock_check`. Answers are the same structured results the Python API returns, so an agent learns a column's formula, its sources, which of its edges the data confirmed, the approved definition of the business term it holds, its owner and the entities it touches, before it writes a query that joins or aggregates. The server rereads the graph whenever it changes, and `konstruct install` writes skills for Claude Code, Cursor, Codex, GitHub Copilot, Gemini CLI and Windsurf that teach the agent whole workflows, such as a safe change, a review or a correct query, with the commands each step needs.

Checking a query. An agent that answers a business question writes SQL, and SQL that runs can still be wrong. `konstruct check-sql` reads a query without running it and composes each aggregated output column down to source columns, with the filters on its way, as the graph's columns are composed. It reports a measure that leaves out a filter its approved definition requires or computes it differently, a column of a retired term, a sum, count or average over a join that repeats the rows it aggregates, judged from the keys, tests and relationships of the ontology, and names the project does not have. Each finding carries its fix, and where a fix changes only the measure the result carries the corrected query. A match by name alone is never an error, since an agent told that a correct query is wrong will break it. With `--history`, the same checks run over a warehouse's query history and count the queries, runs and people that compute an approved measure differently from its definition.

The gate. An agent that changes a data stack should prove the change is safe before it proposes it. `konstruct gate` compares the working tree with a base, as `diff` does, and fails when the change (a) removes a column that a model, task or dashboard still reads; (b) changes what an approved term means, by changing its wording, columns, filters or grain, or the canonical expression or row conditions of a column it names or of any column upstream of one; (c) adds a column that computes the same business idea as an existing one in a different way, as `conflicts` finds, unless the two are bound to different terms; (d) breaks a data law that held on all the data, so that a model's SQL no longer keeps a total it kept; (e) alters a column that another repository pins in its lockfile; or (f) moves sensitive data into a model or exposure that nothing marks to hold it, where neither a setting the base lacks nor a mark the change itself adds counts, copies a masked column into a table in clear, or removes a sensitivity mark. Each finding says what to do, so an agent can fix the change and run the gate again. `konstruct hook` runs the gate as a Claude Code hook after an agent edits SQL or

dbt YAML, from files alone, so the findings arrive next to the edit. The GitHub Action runs the same check on a pull request when it is configured with `gate: true`.

4 Verification by Perturbation

Static lineage is a claim about SQL. It can be wrong in both directions: an edge can be claimed that does not exist (a false edge), and a dependency can exist that no edge records (a missing edge). `konstruct verify` tests the claim against the data, using the definition of dependency from provenance as dependency analysis [5]: an output column depends on an input column if changing the input can change the output.

4.1 The Method

Let Q_m be the SQL of a model m (a dbt model or a table built by a SQL file or task), I its input relations and O its output columns. Verification copies each input into scratch tables, up to a bounded number of rows, so that the warehouse is never written: for DuckDB in an in-memory database with the warehouse file attached read only, in Postgres inside a transaction that is rolled back, and for Snowflake, BigQuery, Databricks and Trino on a sample copied into an in-memory DuckDB database, with the model's SQL translated to DuckDB. For each input column i (and each field of a struct column), and each of 3 strategies s, it builds a perturbed copy $D_{i,s}$ that differs from the inputs D only in i:

- *shift*: $x \mapsto ax+b$ for numbers, a suffix for strings, days or hours added to dates and timestamps, negation for booleans, with a and b drawn from a seeded generator;
- *null*: every value set to `NULL`;
- *shuffle*: values permuted between rows by a seeded permutation.

It runs Q_m on D twice, and skips the model if the two results differ, since a nondeterministic model cannot have changes attributed to inputs. It then runs Q_m on each $D_{i,s}$ and compares the results.

For an output column o, write $\mathrm{sig}_o(X)$ for the multiset of o's values in a result X. A perturbation *changes* o when

$$\mathrm{sig}_o\left(Q_m(D)\right) \neq \mathrm{sig}_o\left(Q_m(D_{i,s})\right), \tag{8}$$

or, when the multiset is unchanged because values only moved between rows (a `ROW_NUMBER()` ordered by i still yields the same numbers), when the multiset of pairs of o with the output columns claimed *not* to depend on i changes. When an output column identifies rows in both runs and does not depend on i, rows are also matched by it and compared value by value, which sees changes even when the perturbation adds or removes rows. A change in the number of rows is recorded as a row influence of i, but never confirms a value edge, since it cannot be pinned on one output column.

Algorithm 2 then classifies each claim and each observation. A claimed edge (o,i) is *confirmed* when the data shows the influence, and *not exercised* when no perturbation moved o on this data. An observed influence the graph did not claim is *via references* when the graph records i as a join, filter or grouping column of m, and *unexplained* otherwise: a gap in the static analysis. Every run records its seed, so it can be repeated exactly, and `verify` writes `VERIFY.md` and `verify.json` and marks each verified edge in `graph.json`, so `explain`, the ontology and the viewer show which edges the data confirmed.

Algorithm 2 Verifying one model

Input: model m, claims $\mathscr{D}_m$ and references $\mathscr{F}_m$, seed
Output: confirmed, not exercised, via references, unexplained

$X \leftarrow Q_m(D)$; **if** $Q_m(D) \neq X$ **then** skip m
$V \leftarrow \emptyset$; $W \leftarrow \emptyset$ ▷ value and row influences
for each input column i and strategy s **do**
  $X' \leftarrow Q_m(D_{i,s})$
  if $|X'| \neq |X|$ **then** $W \leftarrow W \cup \{i\}$
  $V \leftarrow V \cup \{(o,i) : o \text{ changed between } X \text{ and } X'\}$
confirmed $\leftarrow \mathscr{D}_m \cap V$; not exercised $\leftarrow \mathscr{D}_m \setminus V$
via references $\leftarrow \{(o,i) \in V \setminus \mathscr{D}_m : i \in \mathscr{F}_m\}$
unexplained $\leftarrow \{(o,i) \in V \setminus \mathscr{D}_m : i \notin \mathscr{F}_m\}$
  $\cup\, \{(*,i) : i \in W,\ i \notin \mathscr{F}_m \cup \mathrm{src}(\mathscr{D}_m)\}$

4.2 Soundness

The argument rests on two assumptions. First, Q_m is a function of its inputs as bags of rows: its result does not depend on the order in which rows are stored or read. The repeated baseline tests determinism only on a fixed D, and the shuffle copy keeps the row order of D, so a perturbed copy differs from D in the values of i and not in the order of its rows. Second, the engine that runs Q_m gives it the semantics the warehouse does.

Proposition. If verification observes that perturbing i changes the multiset of o's values, then o depends on i: there are databases D and D' that differ only in column i such that $\mathrm{sig}_o(Q_m(D)) \neq \mathrm{sig}_o(Q_m(D'))$. The databases are instances of the input schema without its constraints: a perturbed copy can break a uniqueness or foreign key constraint that the warehouse's data satisfies, so the dependency shown may arise only on such an instance.

Argument. $D_{i,s}$ is built from D by rewriting column i alone, every other input column and every other relation being the same scratch copy. If Q_m is a function of its inputs, as bags of rows, a difference between $\mathrm{sig}_o(Q_m(D))$ and $\mathrm{sig}_o(Q_m(D_{i,s}))$ can only come from the change to i, and the pair $(D, D_{i,s})$ is the witness. □

Consequently an edge confirmed by a change in o's multiset of values is a true dependency, and confirmation of that kind never endorses a false edge; an unexplained influence seen the same way is a true dependency the graph lacks. The paired and row-matched comparisons are weaker. They show that the result of Q_m depends on i, and they attribute the change to o because the columns o is paired or matched with are those the graph claims independent of i; a confirmation or gap found this way holds conditional on those claims of independence. A change in the number of rows shows only that the result depends on i. Every observation is reproducible from the recorded seed. The converse does not hold: *not exercised* is absence of evidence, not evidence of absence.

4.3 Limits

- **Masking.** A genuine edge is not exercised when the data never lets the input through, such as the second argument of a `COALESCE` whose first is never null, or a `CASE` branch

no row takes. A fabricated edge is likewise reported as not exercised, not as false.

- **One column at a time.** Influences that appear only when two inputs change together are not observed.
- **Kinds of dependency.** The witness shows that o depends on i, not how. When a model both reads i as a value and joins, filters or groups on it, a perturbation of i changes o through either route, so a missing value edge on such a column is explained by the reference and not reported. Section 5.2 measures this.
- **Engine and sample.** The proposition is relative to the engine that runs Q_m. On a remote warehouse that is DuckDB running a translation of the model's SQL on a sample, so a translation that changed meaning, or a sample whose inputs no longer join, weakens the result; models that do not translate, or return no rows on the sample, are skipped and reported.
- **Order-sensitive SQL.** SQL whose result depends on row order breaks the first assumption: an aggregate that concatenates or collects without `ORDER BY`, `ROW_NUMBER() OVER ()`, a window or `ORDER BY` whose ties are broken arbitrarily, `LIMIT` without `ORDER BY`, and `first` or `any_value`. The repeated baseline catches such a model only when the two runs happen to differ; otherwise a change can be attributed to i that order alone produced.
- **Types.** Shift is defined for numbers, strings, dates, timestamps and booleans; other types are perturbed by null and shuffle only.

5 Evaluation

Every measurement in this section is written by the script `paper/generate.py`, which runs the experiments below on Konstruct 0.13.22 with sqlglot 30.30.0, DuckDB 2.0.0, dbt-core 1.12.6 and Python 3.12.3, on a machine with 4 logical CPUs (AMD EPYC 7763 64-Core Processor, Linux x86_64). Continuous integration runs it on every change and builds this PDF from its output. Every verification uses the seed 1729.

5.1 Lineage Against the Data

Table 4 gives the results of `konstruct verify` on three workloads. The bundled example is dbt Labs' jaffle_shop_duckdb with an Airflow DAG whose SQL task builds a reporting table from a dbt mart. jaffle-shop is dbt Labs' current sandbox project [15] at commit `5beb145`, with sources over seeds, packages and macros, built with dbt and DuckDB. The differential workload generates queries with Hypothesis [25] (arithmetic, `CASE`, window functions, concatenation, `COALESCE` over `NULLIF`, with and without a CTE layer and a filter), runs each on generated data and requires verification to find nothing unexplained.

On the bundled example, 5 of 5 claimed edges (100%) are confirmed, 0 are not exercised and 0 influences are unexplained; on jaffle-shop, 87 of 87 (100%) are confirmed, 0 are not exercised and 0 are unexplained, across 13 models with 0 skipped. The 61 influences via references on jaffle-shop are join keys, filters and grouping columns whose perturbation changed outputs, each predicted by the graph. Of the 300 generated queries, 0 disagreed with execution. Table 5 breaks jaffle-shop down by model.

Rendering without dbt. The same project built from Konstruct's own Jinja rendering, with dbt's compiled SQL removed from the manifest, gives 113 of the 113 column edges and 9 of the 9 references that compiled SQL gives, with 0 column edges and 0 references that compiled SQL does not give. `diff` and `gate` rely on this rendering for the side of a change that dbt has not compiled.

5.2 Does Verification Catch Wrong Lineage?

A verifier that confirms everything is worthless, so we measure its power the way mutation testing measures a test suite's [22]: seed a fault into the graph and count how often verification reports it. On the bundled example we delete each confirmed edge in turn and verify that model again; a deletion is caught when the influence it carried is reported as unexplained. Of the 4 edges whose input the model does not also join, filter or group on, 4 are caught (100%). Of the 1 edges whose input is also a reference, 0 are caught: the perturbation still changes the output, but the recorded reference explains the change and so masks the deleted edge. This half of the experiment measures that masking, the kinds-of-dependency limit of section 4, rather than a failure of the procedure. Deleting each of the 1 join, filter and grouping references in turn is caught 1 times (100%). The test suite adds fabricated edges, which are reported as not exercised, and dropped filters, which appear as unexplained changes in row count.

5.3 Apache Ossie Portability

Ossie's portability ladder has 4 rungs, from a single `SUM` metric (M0) to a filtered aggregate (M3), with 8 fixtures in all [2]. Each fixture is read through a Konstruct build, written back by `konstruct export ossie` and graded element by element in the ladder's own outcome vocabulary (table 3). Of the 19 elements of the qualified and intra-dataset fixtures, 19 return *portable*: with the dialect tag they were authored in and the same expression, the model's datasets unchanged, as equation (7) requires. The bare variants write model-scoped metrics as `SUM(amount)`, which the specification does not allow; Konstruct reports each with a warning and leaves it out, so 8 elements are *dropped, diagnosed*, 0 are dropped silently and 0 are rewritten.

5.4 Performance

`.kodex/scripts/benchmark.py` generates dbt projects of increasing size: models in layers, each joining earlier models through CTEs, 20 columns each, with a manifest, a catalog and Airflow DAGs. For each it measures a cold build (which writes `graph.html` with every column's composed formula), `--update` with nothing changed and after editing one model in the middle, and the median of 5 runs of `impact`, `trace` and `explain` on the deepest model (table 2 and figure 1).

The cold build of 5,000 models, 100,200 columns and 129,940 column edges takes 61.2 s, an update after one edit 11.9 s, an `impact` query 5.1 ms and an `explain` 3.7 ms. An update with nothing changed still takes 13.9 s at that size, since it reads every input and writes every output. Fitting $\log t = a + b \log n$ over the 4 sizes, where t is the time and n the number of models, gives an exponent b of 0.90 for the cold build (R^2 0.995), 0.98 for an update after one edit

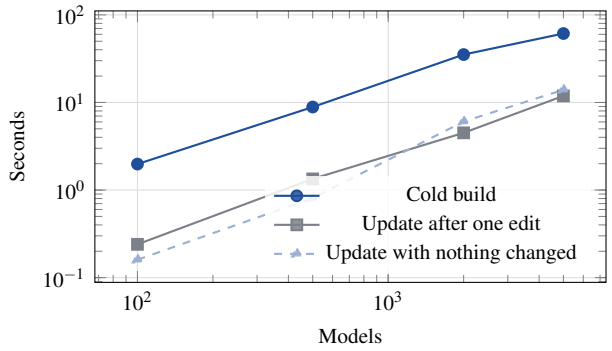


Figure 1: Build time against project size, on logarithmic axes.

(R^2 0.998) and 1.18 for an update with nothing changed (R^2 0.993); an exponent of one would be linear growth.

5.5 The Agent Benchmark

Whether grounding helps an agent is an empirical question, and Konstruct ships the harness to ask it. `.kodex/scripts/agent_benchmark.py` runs a coding agent on Spider 2.0-DBT, dbt projects on DuckDB each with an instruction and a gold database [23], in two arms: the project as shipped, and the same project after `konstruct build` and `konstruct install`, optionally with the MCP server registered. Both arms receive the same prompt, which carries the benchmark's own hints and does not mention Konstruct. Each copy is its own git repository outside any other, so the agent reads no instructions from above it. As in the benchmark's protocol, the agent runs `dbt run` itself, and the database the project's profile writes is scored as the agent leaves it with the benchmark's own `duckdb_match`. Timeouts stop the agent and everything it started, and an interrupted run resumes. The design isolates one variable: the presence of the graph, the formulas and the definitions.

5.6 The Answer Benchmark

The agent benchmark scores whole tasks; this harness scores single answers to business questions, and counts the wrong ones an agent says it is sure of. `.kodex/scripts/answer_benchmark.py` asks an agent 49 business questions (version 4 of its question set) about 2 dbt projects on DuckDB: 28 on the bundled example and 21 on jaffle-shop at its pinned commit. Each question has a single numeric answer, computed by a reference query committed with it that reads only the tables the project's seeds load, so the gold answer does not depend on the models the agent queries; the tests run every reference on the built warehouse. Of the questions, 35 carry a trap: an answer that depends on a business definition, a filter a definition requires, a join to a finer grain that repeats the rows summed, an average over the wrong rows, amounts stored in cents, or a column with a plausible name and another value. Each trap comes with a plausible query that falls into it, and the tests check that its result misses the answer by more than the tolerance; the other questions are controls.

The agent answers each question in 4 arms, each with a tool that runs read-only SQL on the warehouse through the engine of section 4: the schema alone; the schema with the project's definitions as text; the same with the dbt project's SQL and YAML as text too; and the schema with Konstruct's MCP tools that answer from the built graph. The arms differ in more than the definitions. Konstruct's tools also give the dbt SQL, the descriptions and the lineage, so the arm to compare them with is the one given the project as text, and the definitions the benchmark adds to each project were written for it: for jaffle-shop, from the metrics its MetricFlow files declare. The agent submits a number, or none to abstain, and says whether it is sure. The rubric is deterministic and committed with the questions: a number within the tolerance of the question's unit is correct, any other number is wrong, and a wrong number the agent said it was sure of is confidently wrong; a conversation that ends with no submission is counted apart from an abstention. No model grades an answer. The run is one command, `uv run --with anthropic .kodex/scripts/answer_benchmark.py`, which reads the model from `KONSTRUCT_BENCH_MODEL` and the API key from the environment, and records the model's settings, a digest of the harness and of each graph, and intervals for each rate. Its `--plan` and `--dry-run` modes call no API, and `paper/generate.py` runs it only when `KONSTRUCT_PAPER_ANSWER_BENCHMARK` is set, so this paper reports no result of it.

5.7 Keeping the Paper True

The script `paper/generate.py` runs every experiment above and writes the macros this paper uses, and `tests/test_paper.py` fails when the paper leaves out a command, MCP tool or export format, or types a number outside a sentence that cites its source. The test suite itself has 4,360 tests and requires 90% branch coverage. These results above are also tests that run on every change: the verification of the bundled example, the verification of jaffle-shop, the comparison of Konstruct's rendering with dbt's compiled SQL, and the Ossie portability grades. A regression in any of them fails the build before it can reach this paper.

6 Limitations

Static lineage has edges it cannot see. SQL built at run time, Python dbt models, stored procedures and the bodies of user-defined functions are not followed; each gap is a build warning, listed in `REPORT.md`. Macros that query the warehouse while compiling render as empty results when Konstruct renders Jinja itself. `SELECT *` over a relation whose columns are unknown gets lineage only for the columns the query goes on to use. Map keys are read at run time, so a value read from a map traces to the map column.

Orchestration is parsed, not run. Airflow task ids or SQL built in loops, dynamic task mapping over computed lists and DAG factories are skipped with a warning; Dagster graph bodies built in loops are counted once and ops made by factories are not seen. This is the price of never executing orchestration code. dbt is the exception: unless `--no-dbt` is passed, a build runs `dbt docs generate` when a project's artefacts are missing or stale, which runs the project's macros and on-run hooks with its own profile, and those can write to the warehouse.

Verification has the limits of section 4. It sees one input at a time, cannot tell a value dependency from a row dependency on a column used both ways, depends on the data exercising each path, and on remote warehouses checks a translation of the SQL on a sample. Iceberg catalogues are

not verified.

The ontology is inferred by convention. Entities are named from key columns and relations by naming conventions, which a term of kind *entity* can override; a relationship from an entity to itself is not recorded; cardinality is left unknown without a uniqueness test or key.

Search ranks by words. `query` and `search` find `customer_lifetime_value` from "lifetime value" but not from "how much a customer is worth"; an agent is expected to follow up with `describe`.

Formula equivalence is syntactic. Two columns are compared by canonical composed formula, sources, conditions and aggregation, so formulas that are equal in value but differ in form are reported as different.

The agent benchmark is a harness, not a result. Section 5.5 describes how the effect of grounding on an agent is measured; this paper does not report a result for it.

7 Conclusion

Lineage has been treated as documentation: drawn, displayed and trusted. Konstruct treats it as a claim. It derives lineage, formulas and an ontology from the code a team already has, joins them across the tools the team already uses, and then checks the lineage against the data, reporting what the data confirms, what it does not exercise and what the graph does not explain, each observation reproducible from the recorded seed. The result is a set of facts about a data stack that a person can trace and an agent can be grounded in, produced locally, under the MIT licence, by a tool that is itself open. The measurements of Konstruct in this paper are not a snapshot: they are regenerated from the code with every change, so the paper can be checked the same way the lineage can.

References

[1] Apache Airflow. Apache Airflow: a platform to programmatically author, schedule and monitor workflows. The Apache Software Foundation, 2026. https://airflow.apache.org/.

[2] Apache Ossie. Apache Ossie: an open specification for semantic models. The Apache Software Foundation, 2026. Formerly the Open Semantic Interchange. https://ossie.apache.org/.

[3] Basel Committee on Banking Supervision. Progress in adopting the principles for effective risk data aggregation and risk reporting. Bank for International Settlements, November 2023. https://www.bis.org/bcbs/publ/d559.htm.

[4] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *Proceedings of the 8th International Conference on Database Theory (ICDT)*, volume 1973 of *Lecture Notes in Computer Science*. Springer, 2001. doi: 10.1007/3-540-44503-X_20.

[5] James Cheney, Amal Ahmed, and Umut A. Acar. Provenance as dependency analysis. In *Database Programming Languages (DBPL)*, volume 4797 of *Lecture Notes in Computer Science*. Springer, 2007. doi: 10.1007/978-3-540-75987-4_10.

[6] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4), 2009. doi: 10.1561/1900000006.

[7] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. *The VLDB Journal*, 12(1), 2003. doi: 10.1007/s00778-002-0083-8.

[8] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2), 2000. doi: 10.1145/357775.357777.

[9] Dagster Labs. Dagster: an orchestration platform for data assets, 2026. https://github.com/dagster-io/dagster.

[10] DataHub Project. DataHub: The metadata platform for the modern data stack, 2026. Originally developed at LinkedIn. https://github.com/datahub-project/datahub.

[11] dbt Labs. Semantic layer vs. text-to-SQL: 2026 benchmark update. dbt Developer Blog, April 2026. Vendor study. https://docs.getdbt.com/blog/semantic-layer-vs-text-to-sql-2026.

[12] dbt Labs. New dbt Labs report finds AI-driven acceleration is outpacing trust and governance. State of Analytics Engineering, 363 respondents, April 2026. Vendor survey. https://www.getdbt.com/blog/new-dbt-labs-report-finds-ai-driven-acceleration-is-outpacing-trust-and-governance.

[13] dbt Labs. Column-level lineage in dbt explorer. dbt Developer Hub, 2026. https://docs.getdbt.com/docs/explore/column-level-lineage.

[14] dbt Labs. The dbt semantic layer and MetricFlow. dbt Developer Hub, 2026. https://docs.getdbt.com/docs/use-dbt-semantic-layer/dbt-sl.

[15] dbt Labs. jaffle-shop: a sandbox project for exploring the basic functionality and latest features of dbt, 2026. https://github.com/dbt-labs/jaffle-shop.

[16] European Parliament and Council of the European Union. Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (artificial intelligence act). Official Journal of the European Union, 2024. Article 10, data and data governance. https://artificialintelligenceact.eu/article/10/.

[17] Gartner. Data quality: Best practices for accurate insights. Gartner topic page, citing Gartner research from 2020, 2026. https://www.gartner.com/en/data-analytics/topics/data-quality.

[18] Boris Glavic and Gustavo Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE)*, 2009. doi: 10.1109/ICDE.2009.15.

[19] Google. LookML reference. Looker documentation, 2026. https://cloud.google.com/looker/docs/what-is-lookml.

[20] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2007. doi: 10.1145/1265530.1265535.

[21] Reata Hu. SQLLineage: SQL lineage analysis tool powered by Python, 2026. https://github.com/reata/sqllineage.

[22] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5), 2011. doi: 10.1109/TSE.2010.62.

[23] Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows. In *The Thirteenth International Conference on Learning Representations (ICLR)*, 2025. https://spider2-sql.github.io/.

[24] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can LLM already serve as a database interface? A BIg bench for large-scale database grounded text-to-SQLs. In *Advances in Neural Information Processing Systems 36 (NeurIPS), Datasets and Benchmarks Track*, 2023. https://arxiv.org/abs/2305.03111.

[25] David R. MacIver, Zac Hatfield-Dodds, et al. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43), 2019. doi: 10.21105/joss.01891.

[26] Toby Mao et al. sqlglot: a no-dependency SQL parser, transpiler, optimizer and engine, 2026. Including the `sqlglot.lineage` module. https://github.com/tobymao/sqlglot.

[27] Marquez Project. Marquez: The reference implementation of the OpenLineage specification. LF AI & Data Foundation, 2026. https://marquezproject.ai/.

[28] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1), 1998.

[29] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proceedings of the VLDB Endowment*, 4(1), 2010. doi: 10.14778/1880172.1880176.

[30] Alistair Miles and Sean Bechhofer. SKOS simple knowledge organization system reference. W3C Recommendation, 2009. https://www.w3.org/TR/skos-reference/.

[31] Model Context Protocol. Model context protocol specification, 2025. https://modelcontextprotocol.io/.

[32] Monte Carlo. Data quality survey: Data downtime nearly doubled year over year. Monte Carlo with Wakefield Research, 200 data professionals, March 2023, 2023. Vendor survey. https://montecarlo.ai/blog-data-quality-survey.

[33] Tobias Müller and Torsten Grust. Provenance for SQL through abstract interpretation: Value-less, but worthwhile. *Proceedings of the VLDB Endowment*, 8(12), 2015. doi: 10.14778/2824032.2824089.

[34] OpenLineage Project. OpenLineage: An open standard for lineage metadata collection. LF AI & Data Foundation, 2026. https://openlineage.io/.

[35] OpenMetadata Project. OpenMetadata: A unified metadata platform, 2026. https://github.com/open-metadata/OpenMetadata.

[36] Precisely. 2025 planning insights: Data quality remains the top data integrity challenge. Precisely with the LeBow College of Business, Drexel University, November 2024. Vendor survey. https://www.precisely.com/blog/data-integrity/2025-planning-insights-data-quality-remains-the-top-data-integrity-challenges.

[37] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[38] Juan Sequeda, Dean Allemang, and Bryon Jacob. A benchmark to understand the role of knowledge graphs on large language model's accuracy for question answering on enterprise SQL databases, 2023. https://arxiv.org/abs/2311.07509.

[39] Tobiko Data. SQLMesh: Scalable and efficient data transformation, 2026. Column-level lineage from its semantic understanding of SQL. https://github.com/TobikoData/sqlmesh.

[40] W3C OWL Working Group. OWL 2 web ontology language document overview (second edition). W3C Recommendation, 2012. https://www.w3.org/TR/owl2-overview/.

[41] Brett Walenz and Jun Yang. Perturbation analysis of database queries. *Proceedings of the VLDB Endowment*, 9(14), 2016. doi: 10.14778/3007328.3007330.

[42] Eugene Wu and Samuel Madden. Scorpion: Explaining away outliers in aggregate queries. *Proceedings of the VLDB Endowment*, 6(8), 2013. doi: 10.14778/2536354.2536356.

[43] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10), 2018. doi: 10.1109/TSE.2017.2734091.

[44] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2018. doi: 10.18653/v1/D18-1425.

Table 1: The ontology packs Konstruct ships: the tables a schema must have, and the entity, measure and other terms each binds.

Pack	Tables	Entities	Measures	Terms
activeprospect	4	2	8	10
acumatica	7	5	0	5
adobe_umapi	1	2	0	2
aircall	4	5	1	6
amazon_ads	20	6	63	69
amazon_mws	3	3	0	3
amazon_selling_partner	4	1	29	30
amplitude	3	2	0	2
apaleo	4	6	0	6
apollo	4	13	9	22
apple_search_ads	15	2	25	27
apple_store	8	1	10	11
arch	4	12	0	12
asana	1	7	1	8
ask_nicely	4	3	0	3
asyntai_chatbot	4	2	1	3
atinternet	4	0	5	5
atlassian_scim	1	2	0	2
attention	3	2	0	2
avantlink	3	0	3	3
awin	4	2	0	2
aws_cloud_cost	1	0	10	10
azuredevops	4	8	0	8
bamboo_hr	4	11	0	11
belvo	4	18	15	33
betterstack	4	11	1	12
bill	4	3	2	5
bitly	4	1	3	4
blackbaud	4	11	1	12
blackboard	4	5	1	6
branch	4	8	10	18
bunny	4	24	0	24
callrail	3	2	2	4
calltrackingmetrics	4	5	4	9
campfire	4	25	33	58
canny	4	10	1	11
captivateiq	4	6	0	6
carbon_intensity	4	3	0	3
chargebacks	3	2	0	2
chargebee	4	15	27	42
chatgpt_ads	3	0	8	8
checkly	4	13	1	14
chorusai	4	3	1	4
churnzero	4	29	4	33
circleci	4	4	0	4
clari	1	2	1	3
claricopilot	3	2	0	2
claude	4	5	66	71
clerk	6	5	1	6
clickup	4	11	3	14
cm360	1	0	2	2
codat	4	4	1	5
coingecko	4	4	5	9
contentful	1	2	0	2
copper	4	9	4	13
crowd_dev	4	8	2	10
customerio	4	5	0	5
cvent	4	15	5	20
datadog	8	6	32	38
discord	4	4	6	10
easybill	3	4	4	8
easyecom	4	1	13	14
eia	4	0	10	10
eltoro	4	2	7	9
entegra	4	9	0	9
etsy	4	1	5	6
eventbrite	3	2	0	2
exact	8	2	10	12

Table 2: Build and query times on generated dbt projects. Builds in seconds, queries in milliseconds, `graph.html` in megabytes.

Models	Columns	Column edges	Cold build	Update, no change	Update, one edit	Impact	Trace	Explain	graph.html
100	2,200	2,540	2.0	0.2	0.2	0.2	0.1	1.9	2.3
500	10,200	12,940	8.9	0.8	1.3	0.6	0.2	2.4	8.6
2,000	40,200	51,940	35.2	6.0	4.5	1.7	0.2	3.2	34.0
5,000	100,200	129,940	61.2	13.9	11.9	5.1	0.2	3.7	71.8

Table 3: The Apache Ossie portability ladder, read and written back by Konstruct and graded element by element in the ladder's outcome vocabulary.

Rung	Construct under test	Dialect tag	Reference form	Outcomes
M0	single-column aggregate (SUM)	`ANSI_SQL`	qualified	2 portable
M0	single-column aggregate (SUM)	`OSSIE_SQL_2026`	qualified	2 portable
M0	single-column aggregate (SUM)	`ANSI_SQL`	bare	1 dropped diagnosed, 1 portable
M1	scalar field expression (amount * 1.1)	`ANSI_SQL`	intra-dataset	3 portable
M2	core aggregate set	`ANSI_SQL`	qualified	9 portable
M2	core aggregate set	`ANSI_SQL`	bare	6 dropped diagnosed, 3 portable
M3	conditional aggregation	`ANSI_SQL`	qualified	3 portable
M3	conditional aggregation	`ANSI_SQL`	bare	1 dropped diagnosed, 2 portable

Table 4: Static lineage checked against the data. Claimed: column edges the graph claims; confirmed: those whose output changed when the input was perturbed; via references: observed influences of join, filter and grouping columns, all predicted; unexplained: influences the graph did not predict.

Workload	Models	Claimed	Confirmed	Via refs	Unexpl.
Example	1	5	5	4	0
jaffle-shop	13	87	87	61	0
Generated	300 queries				0

Table 5: jaffle-shop by model: claimed, confirmed and not exercised edges, influences via references, and unexplained influences.

Model	Cl.	Conf.	N.e.	Refs	Unexpl.
`customers`	9	9	0	15	0
`locations`	4	4	0	0	0
`metricflow_time_spine`	0	0	0	0	0
`order_items`	9	9	0	22	0
`orders`	19	19	0	24	0
`products`	7	7	0	0	0
`stg_customers`	2	2	0	0	0
`stg_locations`	4	4	0	0	0
`stg_order_items`	3	3	0	0	0
`stg_orders`	10	10	0	0	0
`stg_products`	7	7	0	0	0
`stg_supplies`	7	7	0	0	0
`supplies`	6	6	0	0	0