

Not Your Usual Type(s)

Data contracts as types across languages and engines

Aldrin Montana*

Bauplan Labs
USA

Luca Bigon*

Bauplan Labs
Italy

Colin Marc*

Bauplan Labs
Germany

Jacopo Tagliabue*

Bauplan Labs
USA

ABSTRACT

Composable data systems promise to let developers combine languages, engines, and catalogs without sacrificing a coherent user experience. In practice, however, pipeline-node boundaries remain weakly specified: transformations exchange tables through schemas that are often checked late, enforced unevenly across languages, and disconnected from the semantics business users care about. Based on over a year of operating millions of jobs in Bauplan, we share the design principles behind our new SDK, which treats data contracts as types for a composable, multi-language lakehouse. Users, whether humans or agents, annotate input and output tables with schema objects that encode column types, constraints, documentation, and lineage; Bauplan then interprets these annotations at different points in the execution lifecycle. We show how this design addresses common production failures, and how an “everything-as-code” philosophy enables both deterministic and non-deterministic reasoning over data flows across languages and engines.

VLDB Workshop Reference Format:

Aldrin Montana, Colin Marc, Luca Bigon, and Jacopo Tagliabue. Not Your Usual Type(s). VLDB 2026 Workshop: Fourth International Workshop on Composable Data Management Systems.

VLDB Workshop Artifact Availability:

The source code, data, and/or other artifacts have been made available at https://github.com/BauplanLabs/long_abstracts_and_other_stories.

1 INTRODUCTION

The data lakehouse is today the standard architecture for analytics and AI workloads, combining object storage with open table formats and decoupled, multi-language compute [17, 33]. In many lakehouses, data pipelines – DAGs of transformations from raw to refined data assets – are the most common OLAP use case, measured by both usage and total cost [31, 32].

Historically, a large fraction of DAG errors are due to schema changes at the intersection of two nodes [11], as columns get

dropped or replaced, types change, and semantics shift. The problem is even more acute in composable data systems [18], where multi-language, multi-runtime pipelines are the norm and the node interface may be fully unconstrained or unevenly enforced across languages and runtimes. As the majority of labor shifts from writing to verifying code, thanks to the rise of agents [14], self-documenting, “correct-by-design” data projects promise to bring to data engineering the same productivity boost experienced by software engineers [29].

As most data correctness errors are *prima facie* the same avoidable runtime errors that plague dynamic languages [23], we redesigned our pipeline SDK by borrowing a familiar solution from dynamic languages: type annotations [10]. In *this* paper, we present the design of the new **Bauplan** SDK (“SDK 2.0”), which defines a contract layer for composable lakehouse DAGs. This layer makes schemas, constraints, lineage, and documentation portable across languages, execution engines, and lifecycle stages. Once contracts are explicit and machine-checkable, both humans and agents can safely operate over the same dataflow interface. In particular, we summarize our contributions as follows:

- (1) **multi-language DAG types**: thanks to our privileged position running Bauplan – an agentic lakehouse at scale with millions of DAGs in production – we motivate SDK 2.0 through real-world failures, and outline how the new abstractions address them;
- (2) **three-stage contract enforcement**: we map contract validation to three distinct moments in pipeline execution across a distributed platform: local inference over code files, planning inference over lakehouse state, and runtime inference over data assets produced in a run. As with software engineering, our guiding principle for agentic data engineering is to “shift left” the burden of correctness and fail as early as possible with a clear, explicit error message;
- (3) **everything-as-code**: table descriptions, downstream metrics, joinable columns and column metadata all *live next to the code* that generates the table, and persist into the table metadata. Data semantics (i.e. how tables connect to business entities) are essential both for ongoing maintenance and for translation of business questions into analytics code [15]: by “shifting left” the burden of documentation, we enable a full agentic loop on the code base over a longer time horizon.

Altogether, these capabilities continue the convergence of data and software engineering, which we, among others, pioneered

*All authors contributed equally. JT is the corresponding author and PI on the project: <mailto:jacopo.tagliabue@bauplanlabs.com>.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

before the explosion of coding agents. While *Bauplan* is the natural application ground for these ideas, our design sits more generally at the intersection of programming languages, data management and distributed systems. As such, the ergonomics introduced here are easily portable to other composable systems and the guarantees are easily generalizable to other runtimes. All in all, we believe our lessons from the trenches to be valuable to a broad set of data practitioners.

2 PLATFORM DESIGN AND FAILURE MODES

2.1 Bauplan overview

Bauplan is a composable data system built for AI agents as first-class citizens [21, 24, 29]. It is *composable* as it re-uses a mix of proprietary and open source modules [3, 7] assembled with clear interfaces, while offering users a seamless, cohesive experience. In particular, Bauplan is a lakehouse built on the separation of storage (Apache Iceberg [1] on object storage) and compute (SQL and Python sandboxes). While we refer the reader to [26, 27, 30] for a full system overview, we survey here the distinctive features that are relevant for our SDK re-design.

We introduce a running example both to illustrate Bauplan’s original SDK and to introduce the platform abstractions. With “Titanic” as the source dataset¹, we consider a three-node DAG. Node *f* is a SQL transformation that produces table “*f*” (through a naming convention), Node *g* is a Python transformation that produces table “*g*”, and Node *h* is a Python transformation that produces table “*h*”. Listings 1, 2, 3 show each transformation expressed with the original SDK.

Listing 1: Node #1 (*f*) as a SQL transformation

```
1 -- bauplan: name=f
2 -- bauplan: materialization_strategy=REPLACE
3 SELECT PassengerId AS pid, Pclass AS tclass, Fare
4 FROM titanic WHERE Pclass IN (1, 2) AND Sex = 'female'
```

Listing 2: Node #2 (*g*) as a Python transformation

```
1 @bauplan.python('3.13', pip={'polars': '1.37'})
2 @bauplan.model(materialization_strategy='REPLACE')
3 def g(
4     adult_lodgings=bauplan.Model(
5         'titanic',
6         columns=['PassengerId', 'Pclass', 'Fare', 'Cabin'],
7         filter='Age >= 18 AND Age <= 65',
8     ),
9 ):
10     return (
11         pl.DataFrame(adult_lodgings)
12         .with_columns(cabin_level=(pl.col('Cabin').str.slice(0, 1)))
13         .with_columns(mean_fare=(pl.col('Fare').mean().over('cabin_level')))
14         .select(pl.col('PassengerId').alias('pid'),
15                pl.col('Pclass').alias('tclass'),
16                'Cabin', 'cabin_level', 'Fare', 'mean_fare')
17         .to_arrow()
18     )
```

Listing 3: Node #3 (*h*) as a Python transformation

```
1 @bauplan.python('3.13', pip={'polars': '1.37'})
2 @bauplan.model(materialization_strategy='REPLACE')
3 def h(f_data=bauplan.Model('f', columns=['pid', 'tclass']),
4     g_data=bauplan.Model('g')):
5     return (pl.DataFrame(f_data)
6             .join(pl.DataFrame(g_data), on=('pid', 'tclass'))
7             .filter((pl.col('Fare') > pl.col('mean_fare')))
8             .select('pid', 'tclass', 'Fare', 'Cabin')
9             .to_arrow())
```

¹<https://www.kaggle.com/datasets/yassserh/titanic-dataset>

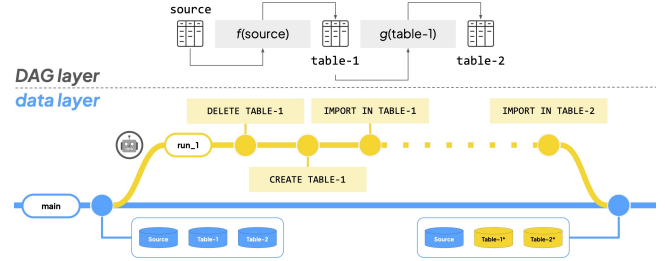


Figure 1: Pipeline declarative code and lakehouse changes. We show a simplified two-node DAG (*top*) transforming data from a source table into *table-1* and *table-2*: from the data asset perspective, the language and logic of the transformations are implementation details for the lakehouse (*bottom*), which is recording on a data branch the I/O operations specified in the code.

The abstractions are largely self-explanatory, so we highlight only their declarative aspects, which play an important role in enabling “DAG types”:

DAG Topology Python function parameters and SQL query source tables collectively define the DAG topology; $f \rightarrow h$ and $g \rightarrow h$ for the above snippets. When considering a SQL query as a function (as *dbt*² does), it is easy to realize that transformations *f* and *g* share the same language-independent shape: one or more input tables and a single output table. Restricting the signature does not significantly constrain pipeline topologies and enables language-agnostic enforcement of invariants at the boundaries (Section 3).

Infrastructure Python version and packages are listed in a decorator *per function*, requiring no client-side tools (such as Docker) and no installation commands; it is the platform’s job to make sure that the sandbox has the required dependencies when running the function.

I/O Physical data operations (e.g., reading source data from S3 in *f* and *g*, passing their outputs into *h*, writing transformed data, and applying filter and projection pushdown in *g*) happen in “platform space”, so that user space only contains transformation code that is executed in secure and sandboxed compute. Fig. 1 (*bottom*) showcases the “isomorphism” [20] between the declarative layer and the corresponding data changes.

We will refer to this example (and variations on it) in the rest of the paper.

2.1.1 Execution model. Bauplan has a standard lakehouse architecture (Figure 2): a control plane, a FaaS data plane, and a local client (CLI, SDK) to trigger cloud operations [26]. Fig. 2 shows the logical flow of information when a DAG gets executed (a “run”)—from a user’s local environment to object storage, and back. A coding agent writes a Bauplan project locally and submits the project when triggering the run; the control plane parses the code into a plan and sends it to a worker for execution; the worker reads/writes data

²<https://github.com/dbt-labs/dbt-core>

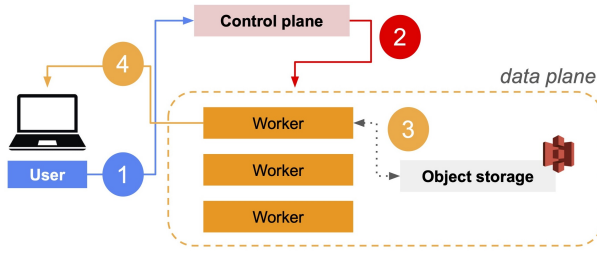


Figure 2: A run in Bauplan: 1) a user writes code locally, and triggers the run; 2) the control plane parses the code into a plan and sends it to a worker for execution; 3) the worker reads/writes data from/to S3 and 4) streams logs and results to the user.

from/to S3 and streams logs and result tuples back. The mapping between user code and lakehouse changes is depicted for a linear DAG in Fig. 1: since tables are managed through Git-like abstractions [22, 28], every function execution in the DAG corresponds to a *commit*, i.e., an immutable reference to the state of the lakehouse at that time. For example, since *g* is declared with *REPLACE* semantics, the platform will drop and re-create the table during a run, then insert rows: what the user writes as a single function gets (deterministically) “compiled” as a series of physical data operations on the lake.

Importantly, the logical flow of a *run* identifies three key *moments* when executing a DAG: (1) the local code environment, before the run is triggered; (2) the control plane, when preparing the run (before DAG execution begins); and finally (3) the worker process, before and after physical data operations (persisting data is a specific operation). These roughly correspond to code compilation, CI/CD and runtime invariant checks in software development: as a general design principle, we want a system that fails as early as possible and with clear responsibilities at each stage of the process.

2.2 Failure modes

Across millions of production jobs, “avoidable” schema mismatches at node boundaries have been a recurring and operationally important class of failure [2, 11]. Note that while these failures are already challenging for workloads run by human experts, their severity is amplified in a world in which untrusted agents operate on production data at scale [29]. To illustrate common failure modes, we consider a few scenarios:

columns dropped If *tclass* is dropped from *g* without dropping references to it in *h*, we see an error in *h* *at runtime*.

null values present The *Pclass* column, and therefore the derived *tclass* column, contains no *NULL* values, although both are declared nullable in Iceberg. If *g* filters out *NULL* values from its output, there is no way to communicate that intent to other DAG nodes: constraints cannot be defined, enforced, or even remembered for query results.

type change If *g* were to change *tclass* to have string values such as “First” for first class instead of the numerical value 1, it would implicitly alter the column type. This would

raise an error for *h* when it is used to join with the table “f” *at runtime*.

shifting semantics Consider that *tclass* from *f* is filtered to specific values, whereas *tclass* from *g* is unfiltered. Downstream analyses can easily look correct but actually suffer from errors of omission due to confusing similarly named columns in (or after) a join operation. Column lineage can be used to disambiguate columns by origin (which DAG node they flow from).

Pushing semantic information to live next to everything else is part of Bauplan’s original vision of “everything-as-code”, in line with many modern Pythonic frameworks [25]. Additionally, beyond these failures, a second trend has been pushing us to revise SDK 1.0: as downstream consumers of refined assets are themselves agents free to explore the lakehouse, many of the original semantic layer [19] capabilities (such as query compilation and lineage) seem ripe for disintermediation by on-the-fly LLM reasoning.

3 DAG TYPES

In this section, we discuss our approach to evolving Bauplan in a principled way to address common challenges rooted in schema failures. Our approach follows a typical two-part pattern from dynamic languages such as Python: (1) *optional syntax*—annotations of code for both human and machine verification and (2) *enforcement semantics*—how a verifier interprets the annotations.

3.1 Annotation Syntax

We address hidden, unintentional schema failures by defining syntax for explicit, machine-checkable contracts. This allows developers to declaratively specify expectations between transformations and incrementally add them to their Bauplan projects to achieve schema and column lineage inference, as exemplified by the snippets below³.

Listing 4: Type contract syntax

```
1 class WomenByClass(TableSchema):
2     """Schema for women passengers with 1st or 2nd class tickets."""
3     pid: Annotated[Int64, Required, Doc('Unique passenger ID')]
4     tclass: Annotated[Int64, Required,
5         Doc('Ticket class, filtered to 1st or 2nd class only.')]
6 ]
7 Fare: Annotated[Float64, Doc('Ticket cost (British pounds)')]
8
9 class CabinCostSource(TableSchema):
10     """Schema for cabin costs used for analysis."""
11     # Same annotations as WomenByClass['pid', 'tclass']
12     PassengerId: Annotated[Int64, ...]
13     Pclass: Annotated[Int64, ...]
14     Fare: Annotated[Float64, Doc('Ticket cost (British pounds)')]
15     Cabin: Annotated[String, Doc('Cabin number (C28 is on deck C)')]
16
17 class CabinCostAnalysis(TableSchema): ... # Omitted for brevity
18
19 class ExpensiveWomenCabins(TableSchema):
20     """Schema for expensive cabins reserved by working-age women."""
21     # Lineage by referencing schema column ('Schema[col]')
22     pid: Annotated[Int64, WomenByClass['pid']]
23     tclass: Annotated[Int64, WomenByClass['tclass']]
24     Cabin: Annotated[String, CabinCostAnalysis['Cabin']]
25     Fare: Annotated[Float64, WomenByClass['Fare']],
26     Doc('Cost of tickets that cost more than average (by deck).')
27 ]
```

³Please check the accompanying repository for the full-fledged example: redundant details are omitted here for brevity

Listing 4 illustrates how schema objects are defined and how they are associated with rich annotations. Then, Listings 5, 6, and 7 show how each node in the DAG directly associates its input tables (from upstream transformations) and its output table with defined schema objects. In this way, schema objects elegantly relate documentation and constraints to DAG nodes using a lightweight, extensible mechanism.

Schema Objects are defined using a Python-centric design to minimize the need for custom SQL and because we expect the syntax to be more ergonomic in Python.⁴ This means that we lean heavily into the use of standard type annotations using *typing.Annotated* and familiar Pydantic-style annotations [4] (type objects such as *Required* and classes such as *Doc* for objects that don't support docstrings).

Bauplan uses the base class *TableSchema* to identify Bauplan schema objects and uses the special *Annotated* type to identify column annotations. The first annotation is the column's data type and subsequent annotations may be rich metadata for "semantic layer" support (Section 4.2) or may be constraints for validation (Section 4.3). For column data types, we choose to accept generic names for supported types (such as *String* and *Float64*) and Bauplan defines a direct correspondence to PyArrow types: to maximize compatibility with other platforms (Section 4.4), we support the subset of Arrow types that intersects with Iceberg. To identify explicit column lineage, a schema column may be referenced directly with the syntax *Schema['column_name']*. For example, the annotation *WomenByClass['pid']* on *ExpensiveWomenCabins.pid* explicitly records lineage to the *pid* field in the *WomenByClass* schema, allowing the system and downstream tools to recover the corresponding upstream origin and metadata.

Listing 5: Node #1 (f) with output schema reference

```
1  -- bauplan: name=f
2  -- bauplan: materialization_strategy=REPLACE
3  -- bauplan: output_schema=WomenByClass
4  SELECT PassengerId AS pid, Pclass AS tclass, Fare
5  FROM titanic WHERE Pclass IN (1, 2) AND Sex = 'female'
```

Listing 6: Node #2 (g) with typing

```
1  @bauplan.python('3.13', pip=('polars': '1.37'))
2  @bauplan.model(materialization_strategy='REPLACE')
3  def g(
4      adult_lodgings: Annotated[
5          Table[CabinCostSource],
6          Filter(Model('titanic'), 'Age >= 18 AND Age <= 65')
7      ],
8  ) -> Table[CabinCostAnalysis]:
9      """Analyze the cabin cost by level for working-age adults."""
10     # Function body remains unchanged
11     return (pl.DataFrame(adult_lodgings).(...).to_arrow())
```

Listing 7: Node #3 (h) with typing

```
1  @bauplan.python('3.13', pip=('polars': '1.37'))
2  @bauplan.model(materialization_strategy='REPLACE')
3  def h(
4      f_data: Annotated[Table[WomenByClass], Model('f')],
5      g_data: Annotated[Table[CabinCostAnalysis], Model('g')],
6  ) -> Table[ExpensiveWomenCabins]:
7      """Expensive cabins reserved by/for working-age women."""
8      # Function body remains unchanged
9      return (pl.DataFrame(f_data).(...).to_arrow())
10
```

⁴Note that while the design does not require a specific version of Python, we show syntax best supported in Python ≥ 3.13 for convenience

Transformation Annotations are defined by parameter and return type annotations in Python and by a structured comment in SQL. In Python, a function parameter is annotated using *Annotated* where the first annotation is a *Table[schema]* type, meaning a tabular data structure (such as *pyarrow.Table*) whose schema matches the specified schema contract. The second annotation must resolve to a *Model* reference identifying an input transformation (an Iceberg table or DAG node), either directly or through a chain of operator functions such as *Filter*.⁵ As syntactic sugar, the schema from the first annotation is applied as a projection on the *Model*'s output.

In SQL, a query is preceded by a structured comment that specifies a schema object, by name, to be associated with the SQL result set. While this lacks support in the client environment in Step 1 (Fig. 2), it allows the control plane to associate contracts with SQL transformations at Step 2. Local static checking is therefore currently available only for Python nodes; SQL contracts first participate in validation during control-plane planning, while Step 3 enforcement at the Arrow boundary is language-independent across supported runtimes.

3.2 Enforcement Semantics

Contracts are enforced by "fail fast" behavior where errors are lifted to the earliest possible point in the execution lifecycle. When the user authors code, local type checkers can catch obvious mismatches immediately (Step 1). Next, before scheduling any execution, the control plane parses the metadata (Step 2) and validates that adjacent nodes compose correctly and that the boundary between tables in the catalog and the assets defined in the DAG is specified correctly. Finally, at the worker (Step 3), runtime checks validate that the physical data conforms to its specified schema before execution (input) and before any results are persisted (output), ensuring that late-discovered schema problems do not leak inconsistent state into storage.

Enforcement at Steps 1 and 3 is relatively straightforward. Step 1 is entirely deferred to a local type checker as annotation syntax is purposefully designed to enable the use of standard type checkers. For example, *Table* is defined in a way so that a type checker recognizes it as having the same interface as a *pyarrow.Table*, which prevents calling invalid methods on *adult_lodgings* in *g*.

At Step 3, a worker uses information from the control plane to validate that an output table (resulting from the transformation) has a correct schema as specified in the Bauplan project. For example, when *h* returns an Arrow table:

```
pl.DataFrame(f_data).select(...).to_arrow()
```

then an appropriate error message will be raised at runtime if schema validation fails:

Listing 8: Example schema errors

```
# Some possible errors on column "Fare" after the transformation 'h'
Error: job failed: function failed due to user error:
Schema contract validation failed on model [h]:
- missing column "Fare" (expected double) (RuntimeTaskUserError)
- column "Fare" has type int64, expected double (RuntimeTaskUserError)
- unknown column "Fare" (has type int64) (RuntimeTaskUserError)
```

⁵The *Annotated* type requires at least 2 annotations and uses the first for type checkers.

Other table validations can then be performed, such as checking for null values (Section 4.3). Finally, auxiliary logic may be executed, such as inserting column type casts when necessary.

Step 2 Enforcement involves a more complex analysis of the DAG types and how information and metadata flow across the DAG. We leverage a graph-based representation to reason about data, task, and infrastructure dependencies [16], turning consistency checks into graph queries (e.g. do projections in the child node correctly map to its parent’s output?). Only in Step 2 can the system link catalog information on source tables with the pipeline code, allowing deterministic type checks at the boundaries between DAG and catalog, and full reasoning over the data flowing from the lakehouse into the tasks being planned.

Listing 9: New node with type-cast

```

1  class TotalFares(TableSchema):
2      """Simple schema for ticket fare data."""
3      fare_total: Annotated[Int64, Required,
4          Doc('Total ticket prices, rounded up to the nearest British pound.')]
5      ]
6
7      @bauplan.python('3.13', pip={'polars': '1.37'})
8      @bauplan.model(materialization_strategy='REPLACE')
9      def fare_sum(
10         tcosts: Annotated[Table[WomenByClass], Model('f')],
11     ) -> Table[TotalFares]:
12         """Compute the total fare for first- and second-class women passengers."
13         """
14         # Function body remains unchanged
15         return (pl.DataFrame(tcosts)
16             .select(fare_total=pl.col('Fare').sum().ceil())
17             .to_arrow())

```

With Listing 9 as a concrete example, we consider *Fare* and how it flows from *titanic* to *f* to *fare_sum*. In the catalog, *titanic* defines *Fare* as a *double*, corresponding to *Float64*. In Listing 4, the schema *WomenByClass* preserves *Fare* as *Float64* and adds documentation. The new node *fare_sum* computes a sum, but wants the result to be rounded and returned as an *Int64*, as specified by its output type *TotalFares*. Before execution, the control plane determines that the output column *fare_total* must have type *Int64* and includes this contract in the execution plan. At Step 3, the worker validates the produced Arrow column and, where permitted, applies the required cast before persistence.

4 CAPABILITIES

SDK 2.0 is now in beta and will soon be included in our documentation: since our SDK is open-source, the client-side implementation is publicly available. In this section, we survey the consequences of this new design across four important dimensions: preventing the “avoidable failures”, powering semantic reasoning through in-code documentation, enforcing data quality concisely and declaratively, and enabling engine composability without giving up contractual guarantees.

4.1 Failure modes revisited

With our enforcement semantics defined, we are now in a position to revisit the failure modes of SDK 1.0 and show how the new annotations address those challenges.

columns dropped If *g*’s declared output contract drops *tclass* while *h* still requires it, the control plane rejects the DAG at

Step 2. If *g*’s implementation omits *tclass* while its declared contract still includes it, the worker rejects *g*’s output at Step 3, before persistence.

null values present If *g* filters out *NULL* values from its output, we can explicitly declare the non-null constraint and relate any violation to the node that defines it.

type change If *g*’s declared type for *tclass* changes from *Int64* to *String* while *h* still expects *Int64*, the control plane rejects the DAG at Step 2. If *g*’s implementation produces strings while its contract still declares *Int64*, the worker rejects *g*’s output at Step 3, before persistence.

shifting semantics For a transformation using *tclass* to disambiguate between *f* and *g*, we declare the desired lineage directly as *WomenByClass['tclass']* where *WomenByClass* is the schema for the output of *f*.

Each of the common failures now has a direct and concise solution.

4.2 Semantic reasoning

In line with the “everything-as-code” principle of modern data frameworks [25], SDK 2.0 couples transformation code with annotations describing the intended business meaning of the assets, as well as column-level metadata. In the *write* path, everything-as-code guarantees that data semantics and data transformations are always “in context”, which follows the best practices for AI-assisted coding and simplifies both human verification and ongoing code maintenance by coding agents.

What about the *read* path? The first step is therefore to make data semantics available outside the code base, without disconnecting them from the code that produced them. This is achieved at materialization time: when the materialization of assets occurs during a run, table and column annotations are persisted in versioned Iceberg table metadata associated with the materialized table state. When business users ask LLMs to translate English questions into queries (Figure 3), an agent using an MCP server that exposes Iceberg catalog and metadata APIs is able to retrieve asset descriptions and column information and put them into the LLM context for accurate SQL generation. In other words, annotations become part of the lakehouse exchange layer: any downstream system can recover the same semantic context, independently of what produced it. This turns metadata from local documentation into a composability mechanism across tools, engines, and agents.

Once again, the design stresses the *outer* composability of Bauplan (how Bauplan and other data systems interact at predefined, clean interfaces) on top of the usual *inner* composability (how Bauplan itself is built out of several modules exposed through vertical APIs).

4.3 Data quality

Types also give a principled handle on data quality checks without additional code – such as manually writing *expectations* in SDK 1.0⁶ – or additional tools. For example, Listing 10 illustrates an enumeration allowing declarative runtime checks without any user code to explicitly compare each value to 1 or 2. In particular, scalar constraints such as non-nullability and enumerated values can be

⁶<https://docs.bauplanlabs.com/concepts/expectations>

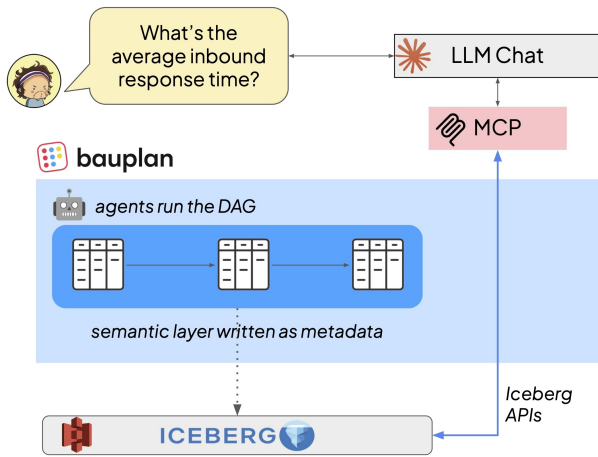


Figure 3: A coding agent writes transformations and table metadata, which are persisted in Iceberg during a run. At read time, an agent uses an MCP server exposing standard Iceberg metadata APIs to retrieve that information and put it into the LLM context for SQL generation.

checked directly on the Arrow buffer “in-flight”, without the need for materialization and wasteful computation as is typical in *dbt*-based setups.

Listing 10: Supporting enum column types

```
1 class WomenByClass(TableSchema):
2     """Schema for women passengers with 1st or 2nd class tickets."""
3     pid: Annotated[Int64, Required, Doc('Unique passenger ID')]
4     tclass: Annotated[Int64, Required,
5         Enum('tclass', names=[('1st', 1), ('2nd', 2)]),
6         Doc('Ticket class, filtered to 1st or 2nd class only.')]
7 
```

Following the “shift left” philosophy, we envision a near future in which the SDK provides even further guarantees in the form of “Dafny-style” *preconditions* and *postconditions* for SQL nodes in a DAG. As an example, consider an aggregation-type node such as `SELECT col1, COUNT(*) FROM table GROUP BY ALL`: static analysis alone guarantees that *if* no nulls are in `table.col1`, no new nulls will be created by this transformation – in turn, if we knew through annotation and type inference that `table.col1` is indeed not null, we could conclude via *modus ponens* that no nulls will be returned at the end.

4.4 Composability

Through its *connector* semantics⁷, Bauplan supports replacing its proprietary sandboxes with alternative runtimes that implement the platform’s execution interface. Because contracts are expressed over Arrow-compatible types, any such runtime that consumes and produces Arrow data can enforce the same input and output boundaries, even when the transformation engine changes. Further, annotations may be persisted in an Iceberg catalog and propagated between independent data pipelines or shared with other data platforms and subsystems (Section 4.2).

⁷<https://docs.bauplanlabs.com/integrations/warehouses-lakehouses/snowflake-outbound>

As the composable data systems community expands, we envision independent platforms interpreting a common subset of these annotations, allowing contracts and semantic metadata to propagate across engines through Arrow and Iceberg interfaces.

5 RELATED WORK

Popular DAG frameworks provide partial forms of asset contracts. *Dagster* [8] provides Python-centric asset checks, while *dbt* model contracts [9] are limited to SQL models and perform a build-time preflight check of each model’s query output against its YAML declaration. Unlike Bauplan, they do not type-check contracts across SQL and Python nodes or compose pipeline contracts against the current lakehouse catalog state before execution.

Python libraries such as *Pandera* [5] and *Patito* [6] similarly provide class-based schemas and runtime validation for DataFrames. Bauplan differs by attaching contracts to multi-language DAG boundaries, checking their composition against catalog state before execution, and enforcing them through a shared Arrow boundary.

The extensive use of *Annotated* and the Pythonic DAG syntax are heavily inspired by Pydantic’s widely understood class-based annotation pattern [4]. Separately, the idea of column lineage and its benefits is based on the analysis used in *SqueezeCache* for “squeezing” [13]. Our usage of lineage is currently more generic (closer to dataflow).

Pythonic data-quality tools such as *Great Expectations* [12] provide a rich framework for validating data assets through expectations and validation suites. However, they add a separate validation layer: users must adopt new dependencies, learn a distinct API, and write quality checks in addition to the transformation code. Our approach instead makes common data-quality constraints part of the table type itself. Nullability, enums, and column types are declared once in the schema and then interpreted uniformly across the client, control plane, and data plane.

6 CONCLUSION

We presented Bauplan SDK 2.0, a contract-oriented extension of our lakehouse programming model. Starting from common failures in multi-language pipelines, we introduced DAG types: lightweight schema objects that attach column types, constraints, documentation, and lineage to pipeline inputs and outputs. Bauplan interprets these annotations across the execution lifecycle, enabling earlier validation in the client, stronger contract composition in the control plane, and runtime checks before invalid results are persisted. The result is a portable contract layer for composable data systems, where correctness and semantics live next to the code that defines the data flow.

Our next step is to push these checks further left. We plan to develop a Bauplan Language Server Protocol (LSP) implementation that brings schema validation and actionable diagnostics directly into the IDE. This would move more reasoning client-side, reduce feedback latency, and enable even faster and safer agentic exploration over production data.

REFERENCES

- [1] Apache. 2024. Iceberg. <https://github.com/apache/iceberg>.

- [2] Divya Bhadauria, Hazar Harmouch, Felix Naumann, Divesh Srivastava, and Lisa Ehrlinger. 2026. A Catalog of Data Errors. *arXiv:2604.09277* [cs.DB] <https://arxiv.org/abs/2604.09277>
- [3] Luca Bigon, Jacopo Tagliabue, and Semih Salihoglu. 2025. DAG Lakehouse Planning with an Ephemeral and Embedded Graph Database. In *VLDB 2025 Workshop: Third International Workshop on Composable Data Management Systems*. https://www.vldb.org/2025/Workshops/VLDB-Workshops-2025/CDMS/CDMS25_13.pdf
- [4] Samuel Colvin, Eric Jolibois, Hasan Ramezani, Adrian Garcia Badaracco, Terrence Dorsey, David Montague, Serge Matveenko, Marcelo Trylesinski, Sydney Runkle, David Hewitt, Alex Hall, and Victorien Plot. [n.d.]. *Pydantic Validation*. <https://github.com/pydantic/pydantic>
- [5] Pandera Contributors. [n.d.]. Pandera: The Open-Source Framework for Dataset Validation. <https://pandera.readthedocs.io/>. Accessed: 2026-07-14.
- [6] Patito Contributors. [n.d.]. Patito: A Data Modelling Layer Built on Top of Polars and Pydantic. <https://patito.readthedocs.io/>. Accessed: 2026-07-14.
- [7] Ryan Curtin and Jacopo Tagliabue. 2025. The Deconstructed Warehouse: An Ephemeral Query Engine Design for Apache Iceberg. In *VLDB 2025 Workshop: Third International Workshop on Composable Data Management Systems*. https://www.vldb.org/2025/Workshops/VLDB-Workshops-2025/CDMS/CDMS25_12.pdf
- [8] Dagster Labs. 2026. Welcome to Dagster. <https://docs.dagster.io/>.
- [9] dbt Labs, Inc. 2026. What is dbt? <https://www.getdbt.com/product/what-is-dbt>.
- [10] Luca Di Grazia and Michael Pradel. 2022. The evolution of type annotations in python: an empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Singapore, Singapore) (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 209–220. <https://doi.org/10.1145/3540250.3549114>
- [11] Harald Foidl, Valentina Golendukhina, Rudolf Ramler, and Michael Felderer. 2024. Data pipeline quality: Influencing factors, root causes of data-related issues, and processing problem areas for developers. *Journal of Systems and Software* 207 (2024), 111855. <https://doi.org/10.1016/j.jss.2023.111855>
- [12] Great Expectations. 2026. Great Expectations Core. <https://github.com/great-expectations>. Open-source data quality validation framework.
- [13] Xiangpeng Hao, Nikhil Nayak, Proteet Paul, JP Guthi, Andrew Lamb, Jacopo Tagliabue, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2026. SqueezeCache: Beyond "Optimal" Eviction for Data Analytics. https://xiangpeng.systems/_app/immutable/assets/squeeze-cache.CgwfvfKO.pdf.
- [14] Ruanqianqian Huang, Avery Reyna, Sorin Lerner, Haijun Xia, and Brian Hempel. 2025. Professional Software Developers Don't Vibe, They Control: AI Agent Use for Coding in 2025. *arXiv:2512.14012* [cs.SE] <https://arxiv.org/abs/2512.14012>
- [15] Clément Labadie, Christine Legner, Markus Eurich, and Martin Fadler. 2020. FAIR Enough? Enhancing the Usage of Enterprise Data with Data Catalogs. In *2020 IEEE 22nd Conference on Business Informatics (CBI)*, Vol. 1. 201–210. <https://doi.org/10.1109/CBI49978.2020.00029>
- [16] Semih Salihoglu Luca Bigon, Jacopo Tagliabue. 2025. DAG lakehouse planning with an ephemeral and embedded graph database. *Proceedings of Workshops at the 51th International Conference on Very Large Data Bases (2025)*.
- [17] Dipankar Mazumdar, Jason Hughes, and JB Onofre. 2023. The Data Lakehouse: Data Warehousing and More. *arXiv:2310.08697* [cs.DB] <https://arxiv.org/abs/2310.08697>
- [18] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (June 2023), 2679–2685. <https://doi.org/10.14778/3603581.3603604>
- [19] Michael Rumiansau and Ivan Fokeev. 2026. Semantic Layers for Reliable LLM-Powered Data Analytics: A Paired Benchmark of Accuracy and Hallucination Across Three Frontier Models. *arXiv:2604.25149* [cs.AI] <https://arxiv.org/abs/2604.25149>
- [20] Nicole Rose Schneider, Davide Ghilardi, Giacomo Piccinini, and Jacopo Tagliabue. 2026. "Skill issues": data-centric optimization of lakehouse agents. *arXiv:2606.01185* [cs.AI] <https://arxiv.org/abs/2606.01185>
- [21] Weiming Sheng, Jinlang Wang, Manuel Barros, Aldrin Montana, Jacopo Tagliabue, and Luca Bigon. 2026. Building a Correct-by-Design Lakehouse. Data Contracts, Versioning, and Transactional Pipelines for Humans and Agents. *arXiv:2602.02335* [cs.DC] <https://arxiv.org/abs/2602.02335>
- [22] Weiming Sheng, Jinlang Wang, Manuel Barros, Aldrin Montana, Jacopo Tagliabue, and Luca Bigon. 2026. GitLake: Git-for-data for the agentic lakehouse. *arXiv:2607.08319* [cs.DB] <https://arxiv.org/abs/2607.08319>
- [23] Shuo Sun, Shixin Zhang, Jiwei Yan, Jun Yan, and Jian Zhang. 2025. Co-Evolution of Types and Dependencies: Towards Repository-Level Type Inference for Python Code. *arXiv:2512.21591* [cs.SE] <https://arxiv.org/abs/2512.21591>
- [24] Jacopo Tagliabue. 2026. Querying Everything Everywhere All at Once: Supervaluationism for the Agentic Lakehouse. *arXiv:2603.13380* [cs.DB] <https://arxiv.org/abs/2603.13380>
- [25] Jacopo Tagliabue, Hugo Bowne-Anderson, Ville Tuulos, Savin Goyal, Romain Cledat, and David Berg. 2023. Reasonable Scale Machine Learning with Open-Source Metaflow. *ArXiv abs/2303.11761* (2023).
- [26] Jacopo Tagliabue, Tyler Caraza-Harter, and Ciro Greco. 2024. Bauplan: Zero-copy, Scale-up FaaS for Data Pipelines. In *Proceedings of the 10th International Workshop on Serverless Computing (Hong Kong, Hong Kong) (WoSC10 '24)*. Association for Computing Machinery, New York, NY, USA, 31–36. <https://doi.org/10.1145/3702634.3702955>
- [27] Jacopo Tagliabue, Ryan Curtin, and Ciro Greco. 2024. FaaS and Furious: abstractions and differential caching for efficient data pre-processing. In *2024 IEEE International Conference on Big Data (BigData)*. IEEE Computer Society, Los Alamitos, CA, USA, 3562–3567. <https://doi.org/10.1109/BigData62323.2024.10825377>
- [28] Jacopo Tagliabue and Ciro Greco. 2024. Reproducible data science over data lakes: replayable data pipelines with Bauplan and Nessie. In *Proceedings of the Eighth Workshop on Data Management for End-to-End Machine Learning (Santiago, AA, Chile) (DEEM '24)*. Association for Computing Machinery, New York, NY, USA, 67–71. <https://doi.org/10.1145/3650203.3663335>
- [29] Jacopo Tagliabue and Ciro Greco. 2025. Safe, Untrusted, "Proof-Carrying" AI Agents: toward the agentic lakehouse. *arXiv:2510.09567* [cs.AI] <https://arxiv.org/abs/2510.09567>
- [30] Jacopo Tagliabue, Ciro Greco, and Luca Bigon. 2023. Building a Serverless Data Lakehouse from Spare Parts. *ArXiv abs/2308.05368* (2023). <https://api.semanticscholar.org/CorpusID:260775634>
- [31] Ciro Greco Tapan Srivastava, Jacopo Tagliabue. 2025. Eudoxia: a FaaS scheduling simulator for the composable lakehouse. *Proceedings of Workshops at the 51st International Conference on Very Large Data Bases (2025)*.
- [32] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Eknath Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. In *VLDB 2024*.
- [33] Matei A. Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Conference on Innovative Data Systems Research*.