Reproducible data science over data lakes: replayable data pipelines with `Bauplan` and `Nessie`.

Jacopo Tagliabue*
jacopo.tagliabue@bauplanlabs.com
Bauplan, NYU Tandon
New York, USA

Ciro Greco
ciro.greco@bauplanlabs.com
Bauplan
New York, USA

ABSTRACT

As the Lakehouse architecture becomes more widespread, ensuring the reproducibility of data workloads over data lakes emerges as a crucial concern for data engineers. However, achieving reproducibility remains challenging. The size of data pipelines contributes to slow testing and iterations, while the intertwining of business logic and data management complicates debugging and increases error susceptibility. In this paper, we highlight recent advancements made at Bauplan in addressing this challenge. We introduce a system designed to decouple compute from data management, by leveraging a cloud runtime alongside Nessie, an open-source catalog with Git semantics. Demonstrating the system's capabilities, we showcase its ability to offer time-travel and branching semantics on top of object storage, and offer full pipeline reproducibility with a few CLI commands.

CCS CONCEPTS

• **Computer systems organization** → *Cloud computing*; • **Information systems** → **Database management system engines**; *Data cleaning*.

KEYWORDS

data pipelines, data cleaning, serverless computing

ACM Reference Format:
Jacopo Tagliabue and Ciro Greco. 2018. Reproducible data science over data lakes: replayable data pipelines with `Bauplan` and `Nessie`.. In *Proceedings of Pre-print (DEEM@Sigmod 2024).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

> "No man ever steps in the same river twice, for it's not the same river and he's not the same man" – *Heraclitus*

Reproducibility is always mentioned as a major obstacle in debugging data science projects and in moving them from development to production [1, 5].

Table 1: Reproducibility checklist

Component	Tools	Example
input data	S3, Iceberg	100M row dataframe
code	Git	10+ SQL / Python functions
runtime	pip, Docker	`scikit==1.3.0`
hardware	local machine, cloud	EC2

The conventional engineering approach, which is based on replicating computer behavior by repeatedly inputting the same data into the same code, reveal critical limitations when confronted with modern data workloads. As shown in Table 1, reproducing a data pipeline needs versioning and portability of extensive inputs, modular code, runtime compatibility with various packages, and hardware flexibility. While existing tools may function adequately in isolation, enabling time-travel capabilities across all these components to reproduce data pipelines demands substantial engineering proficiency, setup and context-switching.

In *this* short paper we describe the recent progress we made at *Bauplan* in attaining reproducibility through a unified framework for data pipelines over *data lakes*. In particular, we demonstrate how a system based on declarative pipelines can decouple the business logic from runtime and data management, and address the challenges illustrated in Table 1. We summarize our contributions as follows:

(1) we outline abstractions that allow data pipelines to be implemented in multiple languages and artifacts to be represented transparently across the hierarchy of persistence (in-memory tables, parquet files, data lake tables and data branches);
(2) we explain the architecture and the ergonomics of our CLI, which allows practitioners to write pipelines in their local IDE and run them in directly the cloud through Bauplan's FaaS runtime;
(3) we describe the open-source Nessie data catalog and show how Git-like semantics can be applied to datasets over data lake.

2 DATA PIPELINES AS FUNCTIONAL DAGS

Data pipelines are the bread and butter of any data processing, serving ETL, analytics, reports and model building. Let's discuss a typical use case, for illustrative purposes:

Example use case #1: Richard is a data scientist at *ACME Inc.*, a financial company interested in detecting fraudulent transactions

*This is a pre-print, non-final version of the paper accepted at DEEM SIGMOD 2024. For the final archival version, please check the official proceedings after the conference.

DEEM@Sigmod 2024, June 09–14, 2024, Santiago, Chile

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

on its platform. Richard is tasked with developing pipeline P (Fig. 1) to transform the raw transaction logs into a clean tabular structure suitable for further analysis and ML workloads.

Fig. 1 showcases two important concepts:

(1) The *separation between storage and compute*, as encouraged by data lake architectures (e.g. the implementation of P would look different in a traditional database like PostgreSQL, or a cloud warehouse like Snowflake). This architecture is the focus of the current system, and it is prevalent in most mid-to-large enterprises (its benefits that been discussed at length before [2, 9]).
(2) The *functional nature of P*: transformation functions (e.g. g in Fig. 1) are not required to know anything about how artifacts are created or persisted by previous transformation steps. As long as the process computing `g` receives the "right input", `g` and `f` can be run in completely different environments, with different languages.

These two points imply that moving data from compute to storage and vice-versa can be decoupled entirely from the transformation logic. Ideally, Richard can focus on writing `g` and `f`, while the underlying infrastructure automatically handles compression, serialization, movement and persistence. If we think of the "tabular structures" as *dataframes*[4], P can be modelled as DAGs in which nodes are dataframes, and edges are transformation functions (from dataframes to dataframes): running P successfully is semantically equivalent to the composition of $training\_data = g(f(source\_table))$; in turn, running f or g successfully depends only on their input dataframes having the right semantics, as encoded in the compatibility of their respective schema. Richard can reason about P entirely at the schema level, moving between languages as he sees fit and leaving the data representation hierarchy as an implementation detail (Fig. 2).

The listings below provide a multi-language P implementation with *Bauplan* syntax: the simplicity of the chosen abstractions removes the burden of translating from SQL tables to Pandas, serializing / deserializing dataframes, reading / writing from / to S3 efficiently:

- `final_table` (the intermediate dataframe in P) is produced through a SQL file with the same name[1], querying *FROM* the raw data in the source table, thereby implicitly declaring its parent;
- `training_data` (the final dataframe) is produced through a Python function with the same name, accepting as input the intermediate table, thereby implicitly declaring its parent.

Listing 1: final_table.sql

```
SELECT c1, c2, c3
FROM source_table -- reference to its parent DAG node
WHERE transactionDate >= DATEADD(day,-7, GETDATE())
```

Listing 2: training_data.py

```
@bauplan.model()
@bauplan.python('3.11', pip={'scikit-learn': '1.3.0'})
def training_data(
    # reference to its parent DAG node
    data=bauplan.Model('final_table')
):
  # transformation logic DAG here
  my_final_df = data.do_something()
  return my_final_df
```

In this perspective, assuming we can count on a cloud runtime that supports the right dependencies (e.g. the function requires `scikit-learn`), we can deterministically reproduce any past instance of *training_data* simply with the code above and an immutable reference to the source data in S3. In other words, to provide a comprehensive treatment of the reproducibility checklist in Table 1, we would need a platform that provides *i)* data and code versioning, and *ii)* a cloud runtime.

In the ensuing sections, we outline our solution for the two missing pieces of the puzzle: first, how to go from dataframes to files and vice versa, because while Richard thinks in terms of the former, data versioning on S3 happens in terms of the latter (Section 3); second, how to run Richard's code directly in the cloud, to avoid runtime and hardware discrepancies when reproducing pipeline runs (Section 4).

3 FROM DATAFRAMES TO A DATA CATALOG

As the architecture fully embraces the separation of storage and compute, the only way to persist states is through object storage – according to our abstractions, the relevant states are the dataframes produced by transformation functions, and storage is provided by S3 (or equivalent service). Fig. 2 illustrates the write path that starts from the the transformation in **training_data.py** – a Python dataframe that only lives inside the runtime memory –, and ends with a table in S3 – an *Iceberg* table that can be now queried by downstream pipelines, or by any compatible query engine (*duckdb*, *Snowflake*, *Dremio*, etc.):

(1) **from Arrow to Parquet**: standard open source libraries[2] perform the conversion from the in-memory representation (Arrow) to the compressed one (Parquet);
(2) **from Parquet to Iceberg**: when a dataframe is converted to multiple files in S3, metadata are required to preserve a dataframe semantics for downstream systems. We turned to the popular open format *Iceberg*[3], which defines metadata to contain the *schema* (common to all files) and pointers to row groups as physically stored in S3. This level of indirection enables transaction-like behavior over the data lake: users can reason with high-level abstractions such as schema evolution and table snapshots, instead of tracking file changes. Inserts and updates produce a unique commit, which can be referenced as an immutable state of the table;
(3) **from Iceberg to Nessie**: multiple *Iceberg* tables are managed by a further abstraction, a data catalog – we picked Nessie[4] as our open source catalog, for its support for multi-table transactions (crucial for data pipelines) and data branching. Branching and merging for dataframes are analogous to Git operations, and allow Richard to launch runs from the same

[1] Note here the adoption of the popular *dbt* naming convention: https://github.com/dbt-labs/dbt-core.

[2] https://arrow.apache.org/docs/python/index.html
[3] https://iceberg.apache.org/
[4] https://projectnessie.org/

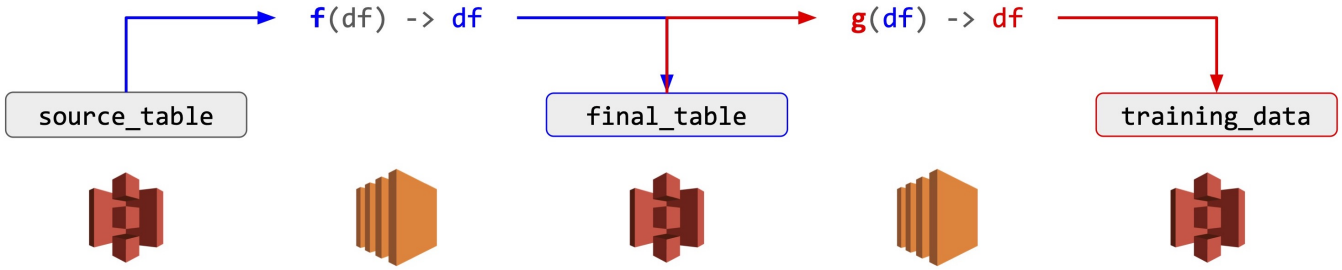

Figure 1: Interleaving of compute (EC2) and storage (S3) in a data pipeline, modelled as a Directed Acyclic Graph (DAG): the responsibility of the data scientist is to write functions that transform the original data artifact (the source data) in intermediate and then final dataframes for downstream consumption (e.g. run a ML model).

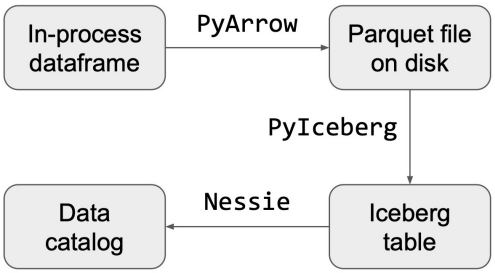

Figure 2: Hierarchy of data representation: while data scientists interact only with in-process dataframes, the system persists the information preserving the overall semantics through different (reversible) layers of abstraction – physical files, collection of files into tables, collection of tables.

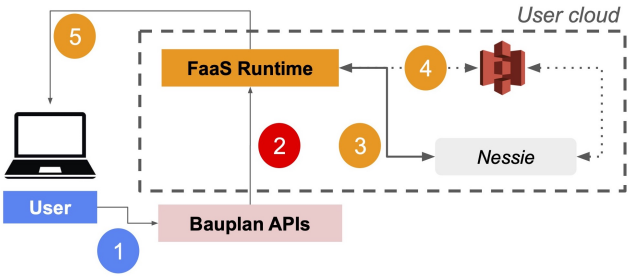

Figure 3: Data flow for a pipeline run: 1) user issues a query to the middleware, which 2) sends a plan to the runtime; 3) the runtime asks Nessie for the parquet files backing the plan and 4) retrieves them from S3. Finally, the pipeline is run and 5) results are sent back to the client.

data `source_table` has, but sandboxing his transformations to avoid pushing to downstream services bad data while he's developing. While branching is not necessary for reproducibility, it's an essential component (with reproducibility) for safe debugging (Section 5).

On the read path the system performs the same conversion in reverse order: given a branch selected by Richard (Section 5), it identifies the relevant table commits, retrieves the files and convert Parquet to Arrow to set the input for the next function.

4 BAUPLAN ARCHITECTURE

If the code in Section 2 captures the first person perspective of *writing* pipelines, we now have to show the first person perspective of *running* them. From the point of view of Richard, *Bauplan* is a pip-installable package, which gives access to lakehouse capabilities[3, 9] through simple CLI-based interactions: after writing the code for *P*, Richard can execute it in the cloud with a simple terminal command: `bauplan run`.

Fig.3 describes the system perspective of a run: the code gets sent to the API, which parses it and outputs a plan for the execution in Richard's cloud[5]; the runtime communicates with *Nessie* to retrieve the appropriate data from object storage, and then executes the plan by converting files to dataframes for the pipeline nodes.[6] Two observations are worth mentioning:

- **pipeline abstractions are naturally converted to "Function as a Service" (FaaS) execution**: if pipelines are DAGs of transformations, FaaS semantics makes for a great developer interface; furthermore, FaaS emphasizes a clear separation of concerns between users (Richard writing f and g) and the system (converting dataframes from / to memory, scheduling function execution efficiently etc.);
- **runs are immutable**: every `bauplan run` returns a `run-id`, which uniquely identifies the combination of the code *and* the input data (the data commit mentioned above).

Crucially, since *Bauplan* sits at the intersection of code, runtime and data access, it can provide *through a single abstraction* the time-travel capabilities of what typically happens in separate systems (Table 1). We will now show how these building blocks can solve efficiently a typical debugging use case.

5 REPRODUCING PIPELINES

Let's consider this common debugging use case:

[5]In a lakehouse, data can be processed directly in the target cloud, with no additional costs or privacy concerns.

[6]Details on the custom FaaS runtime powering *Bauplan* are beyond the scope of *this* paper, but the interested reader may check [7, 8].

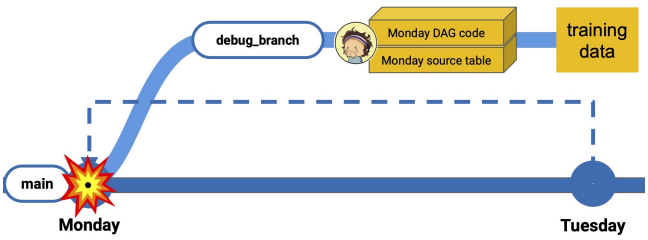

Figure 4: Debugging on *Bauplan*: when Richard reproduces Monday's run, the system 1) travels back in time at Monday's source data *and* pipeline code, 2) creates a debug branch for his experiments, and 3) materializes the target artifacts inside the branch.

Example use case #2: *P* now runs in production every night[7] – when it ran last night, it unexpectedly produced an empty `training_data` table. Richard is tasked to identify and fix the bug.

The use case brings two connected but distinct challenges ((Fig. 4). First, there is *reproducibility*: to reproduce the faulty run, Richard needs to run the same code over the same source data as last night; then, there is *materialization*: when debugging past runs, Richard should have a temporary version of `training_data`, so that his debugging attempts won't interact with production artifacts that the rest of the company is using. Since *Bauplan* provides immutable reference to code and input data for every run, the solution to both our challenges is a few CLI commands away:

Listing 3: Reproducing a pipeline (CLI)

```
bauplan checkout richard.debug_branch
bauplan run --id=1441804
bauplan query "SELECT␣COUNT(*)␣FROM␣training_data"
```

Even if the reader never encountered the commands before, its semantics should be obvious: given the id of last night production run (1441804), Richard can: 1) create a target branch separate from production to host dataframes while debugging; 2) re-run last night pipeline starting from the same input and re-using the same code (ensuring reproducibility); 3) query a dataframe in his branch, to reproduce the bug first and then verify how the final table changes as he fixes the code[8]. Some points are worth highlighting:

(1) **CLI is all you need**: Richard does not need to know / setup / provision a data catalog service, nor learn its API, download a client etc. a simple Git-like command is enough for him to operate the system proficiently and achieve the intended goal;
(2) **built-in namespacing**: we follow a `user.branch` convention, so that users can only write in their branches, but everybody can read any branch;
(3) **interoperability with query engines**: artifacts can be queried within the platform itself through SQL with no additional setup, or they can be read by any Iceberg-compatible engine;
(4) **efficient data re-use**: when branching occurs (Fig. 4), the original source table at the start of the DAG is not copied: *Nessie* builds the debug branch through copy-on-write semantics over the lake, avoiding slow and costly copies;
(5) **extensibility to CI/CD**: the same building blocks can be used to enforce a Write-Audit-Publish pattern during normal development, or even during scheduled execution: a common pattern among *Bauplan* users is to run Python tests over dataframes for data quality[9]; branching, testing and merging through a command line API allow a CI/CD similar to software builds.

Taken all together, *Bauplan* APIs provide a unified, multi-language abstraction over the four reproducibility components in Table 1: *code* and *input data* are versioned at each `bauplan run`, leveraging Nessie and Bauplan own APIs; *runtime*'s concerns are expressed directly in code (as required Python packages), *hardware* is stable across runs, as local execution is avoided altogether thanks to the FaaS cloud engine.

6 RELATED WORK

DAG-based modelling of data pipelines is common in orchestrators (e.g. Airflow[10]): unlike Bauplan however, orchestrators do not provide built-in runtimes nor direct access to dataframe semantics, leaving users to roll their own reproducibility recipe by stitching together several tools.

Dvc is a popular "Data Version Control" system, which primarily operates with file semantics: as a consequence, the system is mostly used for local files and single-file datasets,[11] as opposed to the thousands of files composing Iceberg tables on a lake. The lack of dataframe semantics is reflected in basic usage of S3 and lack of interoperability with query engines.

Metaflow provides S3-based immutable runs [6], and it is (to the best of our knowledge) the only other system versioning code and data artifacts; its abstractions are however more generic, and data management is entirely left to the users: since any variable is a blob, dataframes are not interoperable with SQL engines nor they are addressable directly with table semantics.

7 CONCLUSION AND FUTURE WORK

Reproducing pipelines over a data lake is a common concern for modern enterprises: the absence of table semantics and the complexity of inter-operating code, runtime and storage, poses a formidable challenge for practitioners. We presented *Bauplan* and *Nessie* as an alternative to fragmented tooling: by combining in a simple API a multi-language cloud runtime and data branching, we obtained full reproducibility with just a few CLI commands.

[7] Scheduling details are not important for the example: you can imagine executing a *Bauplan* run from a cron job or through more complex orchestration.
[8] In other words, `COUNT` should be zero at first when the exact production run gets replayed, and then it changes as Richard starts fixing the underlying cause during iterative debugging.
[9] These are typically called *expectations*, and they are functions from dataframes to booleans.
[10] https://airflow.apache.org/
[11] https://dvc.org/doc/user-guide/data-management/importing-external-data

REFERENCES

[1] Iñigo Martinez, Elisabeth Viles, and Igor García Olaizola. 2021. A survey study of success factors in data science projects. *2021 IEEE International Conference on Big Data (Big Data)* (2021), 2313–2318. https://api.semanticscholar.org/CorpusID:245937604

[2] Dipankar Mazumdar, Jason Hughes, and JB Onofre. 2023. The Data Lakehouse: Data Warehousing and More. arXiv:2310.08697 [cs.DB]

[3] 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 (jun 2023), 2679–2685. https://doi.org/10.14778/3603581.3603604

[4] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2033–2046. https://doi.org/10.14778/3407790.3407807

[5] Shreya Shankar, Rolando Garcia, Joseph M. Hellerstein, and Aditya G. Parameswaran. 2022. Operationalizing Machine Learning: An Interview Study. https://doi.org/10.48550/ARXIV.2209.09125

[6] 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).

[7] 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

[8] Jacopo Tagliabue, Ciro Greco, Luca Bigon, Nathan LeClaire, Vladimir Adam, and Mattia Pavoni. 2023. Data Pipelines as Cloud FaaS with Bauplan. *forthcoming* (2023).

[9] 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*.