GitLake: Git-for-Data for the Agentic Lakehouse

Weiming Sheng*
Columbia University
USA

Jinlang Wang*
University of Wisconsin-Madison
USA

Manuel Barros*
Carnegie Mellon University
USA

Aldrin Montana*
Bauplan Labs
USA

Jacopo Tagliabue*
Bauplan Labs
USA

Luca Bigon*
Bauplan Labs
USA

ABSTRACT

We present `GitLake`, a Git-for-data design for an agent-first lakehouse. The system lifts single-table Iceberg snapshots into lakehouse-wide commits, branches, and merges, letting agents work on isolated branches while humans review and publish changes. Pipelines run on temporary branches and publish through a final merge, so all outputs become visible atomically or none do. Finally, we report production lessons as well as correctness insights from a preliminary Alloy model of our core abstractions.

VLDB Workshop Reference Format:
Weiming Sheng, Jinlang Wang, Manuel Barros, Aldrin Montana, Jacopo Tagliabue, and Luca Bigon. *GitLake*: Git-for-Data for the Agentic Lakehouse. VLDB 2026 Workshop: DASHSys: Systems for Data-centric Agents with Human-in-the-loop.

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/BauplanLabs/git_for_data.

1 INTRODUCTION

Coding agents are taking the software engineering industry by storm, both when humans and agents write code together in a tight feedback loop and when agents in a ReAct loop [13] continuously hill-climb toward a task. Version control systems such as *Git* lie at the core of both patterns as they allow developers to incrementally develop software, using **commits** as intermediate checkpoints. Commits support time-travel for debugging and reverting code, and provide a unit of concurrent collaboration. However, agentic adoption in the data analytics domain lags behind the rest of the industry. The lakehouse is the *de facto* standard OLAP for analytics and AI workloads [11]; however, the affordances in traditional OLAP systems make agents unsafe [9].

We share the design of `GitLake`, the Git-for-data layer inside `Bauplan`'s lakehouse platform. As labor shifts from writing code to verifying and approving changes, correctness in the face of untrusted actors becomes non-negotiable [4]. `GitLake` induces a natural division of labor between humans and agents: agents can explore and propose changes on isolated **branches**, while humans review and approve only what should be published to production. The core abstractions are obtained by "porting" *Git* primitives to OLAP. By reusing familiar concepts, users quickly learn how to leverage the APIs to develop data pipelines collaboratively and time-travel to a previous state of the lake. We summarize our contributions as follows:

(1) we motivate Git-like abstractions in agentic data workflows through the lens of production workloads. As of today, `Bauplan` has run millions of jobs across hundreds of thousands of data branches (Section 5), making it, to our knowledge, one of the first systems of this kind tested at industry scale;
(2) we identify a core set of primitives (commits, branches, merges) building on top of single-table guarantees from open formats, and show that Git-for-data is a versatile mental model that can unify versioning, collaboration, and transactional guarantees, while also providing a natural human review boundary for agent-generated changes;
(3) we share lessons from our implementation journey (Section 5): copy-on-write storage, API design, and the use of lightweight formal modeling to stress-test core intuitions. We discuss real-world production metrics from agentic usage, and share with the community an open-source Alloy model.

Notwithstanding our focus on safe agentic workloads on a production lakehouse, our design sits more generally at the intersection of frontier topics in data management and distributed systems. As such, we believe our lessons from the trenches to be valuable to a broad set of practitioners.

2 GIT: FROM CODE TO DATA

While coding agents are powerful for code generation, they can rarely "one-shot" a complex or delicate task. We consider the "eventual completion" of such a task as a **development process**. *Git* primitives are designed for modifying codebases that are built, tested, and run locally, and every change in *Git* is immutable, but nothing is fatal. In particular, the interest in self-driving codebases highlights three key aspects in the development process:

Fail safely A multi-table (and often multi-language) data pipeline should not result in an inconsistent lakehouse state when it fails. [1]

*All authors contributed equally and are listed `ORDER BY AGE ASC`. JT is the corresponding author and PI on the project: mailto:jacopo.tagliabue@bauplanlabs.com.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

[1] Industry lakehouses such as *Snowflake* and *Databricks* do not offer APIs for multi-language pipeline transactions.

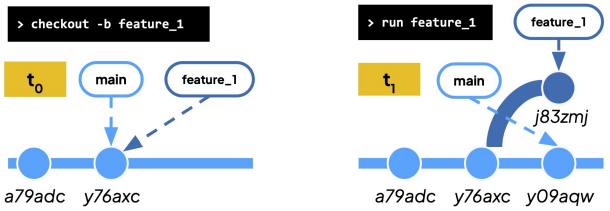

Figure 1: Branches are movable pointers: creating a new branch (t_0) is a no-op, but at t_1 y76axc is now the parent of two diverging commits, with pointers moving accordingly.

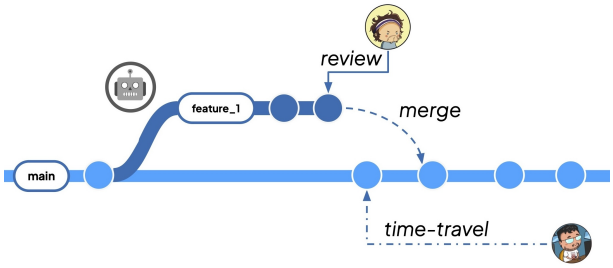

Figure 2: Git APIs enable collaboration and auditability.

Cooperative work A swarm of agents and a team of humans working in concert should be able to build on each other's work, in the spirit of branching off existing solutions, iterating, and managing conflicts through the tried-and-tested PR flow.

Backtrack In the face of catastrophic failures, or when asked for an audit, the lakehouse should be returned to a previous sound state.

Comparable affordances in current OLAP systems are either rare or non-existent, leading to significant gaps. A pipeline that fails unsafely may result in an inconsistent global state where downstream readers observe a mix of old and new tables without any clear notion of lakehouse-wide success. Without an efficient and semantically sound workflow, data systems are left to constantly sync rows between development and production, with ad hoc reconciliation strategies. Additionally, in data, if AI-generated code drops a table, there may not be any obvious built-in way to undo the damage or provide point-in-time queries for auditing.

We could try to patch current systems to fill these gaps, for example, by swallowing the complexity of application-layer transactions to support manual pipeline rollbacks [12]. However, researchers are starting to question whether traditional platforms could ever deliver agentic data systems in a timely fashion [4]. `GitLake` is built on the premise that *Git*'s virtues are not accidental: our best bet is then to start treating our data estate as we treat our codebase.

3 SYSTEM DESIGN

Our design is motivated by the observation that Apache Iceberg already provides a strong primitive: ACID-compliant, *single-table* snapshots, with optimistic locks guaranteed by a relational database at the catalog layer. We lift single-table writes to global lakehouse writes, and then show how to progressively build out more *Git* abstractions. In Section 4, we discuss the implementation of this design through concrete APIs, workflows, and storage optimizations.

3.1 From snapshots to commits

A single Iceberg table evolves through snapshots, persisted in S3 and (importantly) also recorded in the Iceberg catalog (i.e., *Postgres*). The key insight is to lift (within the atomic swap for the snapshot update) the table change into a **data commit** that maps, at that moment in time, all catalog tables to their snapshots. As we attach to every commit a hash identifier, metadata, and a parent pointer, we obtain our first *Git* primitive in the data estate. It is already easy to see that, by adding a `hash` argument to a query API, we can provide time-travel for auditing and debugging with minimal changes to the read path: retrieve the relevant snapshots from the commit and pass the metadata URIs to the engine for scans. Since commits are stored in the catalog independently of the tables they reference, this implies that any destructive mutation (e.g. an agent dropping a table) is reversible through a revert API (Listing 2).

3.2 From commits to branches

By navigating from commit to commit through their parents, we naturally induce *histories*: a **data branch** is simply a movable reference to the *HEAD* of a history. When creating a branch `feature` from the production lakehouse (`main`), we initially get a new movable pointer to the same commit (Figure 1): after writing to `feature`, `main` and `feature` diverge as `feature` now points to a new commit. It is then straightforward to see that, by adding a `reference` argument to a pipeline API, we can provide sandboxed development for data assets, as `run(..., ref=feature)` leaves downstream consumer queries intact.

3.3 From branches to merges

A **data merge** takes two heads and produces a new commit on the destination branch that applies, pending conflicts, the snapshot updates reachable from the source and not yet present in the target. Crucially, merges happen *atomically* and in the control plane only: merging is metadata-centric, i.e., a catalog update rather than moving or rewriting the underlying Parquet files, so collaboration remains cheap even when tables are large. By adding `merge`, collaborative scenarios discussed for codebases are enabled on data: Figure 2 depicts human reviews of agentic writes by leveraging the abstractions introduced so far. In practice, the merge into `main` also acts as a possible review boundary: agents can iterate freely on branches, but production publication happens only through an inspectable, human-approved merge. While collaboration may indeed be solved, this is not yet sufficient for correctness.

3.4 From merges to transactions

No primitive we have seen so far prevents the "half-written pipeline" (Section 2), as exemplified by Figure 3 (top). `run_1` executes successfully on `main`. Tables *Parent*, *Child*, and *Grandchild*, abbreviated as P, C, and G, are updated to snapshots P^*, C^*, and G^*, respectively; however, `run_2` breaks after updating P^* to P^{**} but before updating C^*, leaving `main` in a *globally* inconsistent state built out of legitimate single-table snapshots $\{P^{**}, C^*, G^*\}$. Since `main` can be accessed at any point by downstream systems, the inconsistency may percolate uncontrollably.

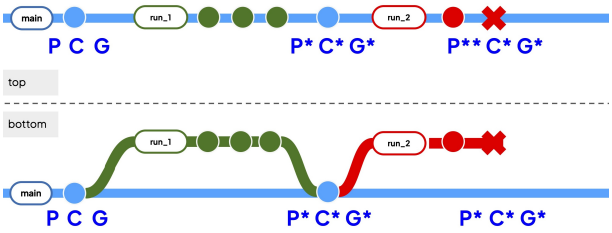

Figure 3: Transactional pipelines. *Top*: without coupling temporary branches with runs, `run_2` leaves `main` with a new version of *Parent* but an old version of *Child* and *Grandchild*. *Bottom*: the `run` API guarantees atomic publication of all tables on success, and isolation in case of failure.

Lakehouse pipelines run on ephemeral, multi-language compute and decoupled storage. To recreate MVCC-style transaction boundaries, we *logically* bundle compute and *merges* inside the `run()` API. Figure 3 (bottom) illustrates **transactional branches**. First, `run_1` shows the happy path: a temporary branch is opened to host commits generated by the pipeline, and it is merged at the end. When the merge happens, consumers see all the new snapshots at once. `run_2` illustrates the unhappy path: failure to update C^* to C^{**} does *not* compromise `main`, which continues to serve downstream consumers the globally consistent state of the first successful run. As an additional bonus, the aborted transactional branch remains reachable for debugging, enabling users to triage the failure of `run_2` by querying faulty intermediate assets.

4 IMPLEMENTATION

4.1 Data management

The physical design of `GitLake` follows a common pattern in open lakehouse architectures: control state (such as branch heads, commit metadata, and run metadata) is mutable and stored in a relational catalog; data is immutable and stored using the Iceberg table structure (Parquet files and manifest files), simplifying bookkeeping and copy-on-write semantics. Appending data to *Table T* in a new branch adds new Parquet files corresponding to the appended rows, avoiding costly duplication for pre-existing rows. In practice, most Git-for-data operations are metadata operations over references and do not involve data movement.

4.2 APIs

We expose the above primitives through APIs available through both the CLI and Python scripting. In line with agentic best practices, the CLI supports progressive discovery and self-documenting behavior through a recursive `-help` flag; the SDK exposes fully typed methods and supports local validation with type checkers for a fast feedback loop. Importantly, even subtle semantic distinctions are clearly marked through argument names and types; i.e., if a method accepts `branch=value` (and not `ref=value`), it is immediately clear that the semantics of the operation can only make sense at the *HEAD* of a history. By implementing the CLI in Rust and then binding the same core methods into Python, we obtain two surfaces from a single source of truth that dispatch identical types and error taxonomies.

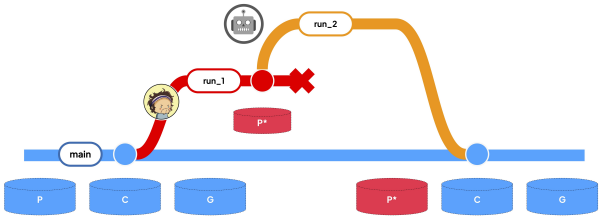

Figure 4: A counterexample. A failed run leaves an aborted branch open after the first commit. Another agent can branch off that commit and later merge back to `main`, creating an inconsistent state.

While a full description of the framework is beyond the scope of this paper [6], Listing 1 shows a minimal DAG with two Python transformations chained together (*Source* → *Parent* → *Child*). Listing 2 then highlights how agents can programmatically control both the data assets and the transformations by interleaving data operations with Python control flow. Complex logic for creating, deleting, and merging branches can be assembled from simple typed primitives that are easy for agents to write and quick for humans to verify.

Listing 1: A minimal DAG with two Python transformations.

```
def parent_table(df: Source = source):
    # return a table fulfilling the Parent schema
    return table

def child_table(df: Parent = parent_table):
    # more transformation code here...
    return table
```

Listing 2: Interleaving data-ops with Python control flow.

```
# 1) assuming we have a client, get the current head of main
cnt_main: Commit = client.get_commits(ref="main", limit=1)[0]
# 2) create a development branch from production
dev_br: Branch = client.create_branch("dev_br", from_ref="main")
# 3) run the DAG on the branch
run_state = client.run("pipeline/", ref=dev_br)
# 4) merge the branch into production on success
if run_state.success() and verification_passed():
    client.merge(dev_br, into="main")
    client.delete_branch(dev_br)
# 5) query the table as it was *before* the merge
rows = client.query("SELECT SUM(_S) FROM child", ref=cnt_main.hash)
# 6) revert a table to a previous snapshot
assert client.revert_table(
    table="child",
    source_ref=cnt_main.hash,
    into_branch="main",
)
```

Our key insight from Section 3.4 is to modify the semantics of running a pipeline and *logically* couple function execution with data branches. A platform-level execution of a `run` simply implements, behind the scenes, the flow at the bottom of Figure 3: a branch is opened automatically from the target branch, the *writes* are materialized there, and the branch is merged and deleted on success; on failure, `run_2`'s transactional branch stays open, and `main` does not contain a partial state. Importantly, this optimization is only possible because of the declarative nature of both the SDK (i.e., agents only specify that a DAG should run on a branch, not how) and the framework (functions specify desired inputs and their schemas, not the physical I/O).

5 LESSONS LEARNED

Flexibility vs. correctness. Adding *Git* primitives to data DAGs expands the space of reachable states combinatorially. Inspired by the success of lightweight formal models in distributed systems [1], we ported our abstractions to Alloy to discover counterexamples that sharpen the intended semantics of the system: are inconsistent states really unrepresentable? Preliminary findings point to a tension between flexibility and correctness: Figure 4 illustrates a discovered counterexample to consistency in the face of failed runs on branches. Nested branches are powerful, so the obvious solution of disallowing branches on branches is not necessarily the right one. We leave further iterations to future work.

Branches grow quickly. Organizations on `Bauplan` spawn branches at a high rate. In practice, we observe that our copy-on-write, metadata-only branching system scales to agentic usage and concurrency well beyond typical human-centric workloads. Traces from production confirm that creating a branch is effectively a no-op (p_{95} is around 80ms), even at a pace of hundreds of thousands of new branches per week. As highlighted in our recent benchmarks[2], comparable primitives in Snowflake (zero-copy clone) and Databricks (shallow copy) are 100x slower than `GitLake`. Finally, since tables are generated from code (Listing 1), merge conflicts happen only in the rare case of concurrent *code* modification: in our traces, we see on average only ten conflicts per 100k attempts.

Verification will soon be the bottleneck. As work shifts from writing to reviewing, a merge-centric worldview risks moving the bottleneck to a different layer: if every data analysis must be reviewed before merging, exploration scales faster than human verification. As a way forward, we have experimented with decoupling getting answers from reconciling a canonical table version: if we could query *across* branches, users could trade "partial" answers for quicker response time [8].

Autonomy needs a harness. Git-for-data abstractions have already been shown to support Ralph-like scenarios such as self-healing data pipelines [10]: because branches are cheap, agents can explore multiple strategies and a verifier can compare their outputs before a single strategy is merged into `main`. In our own experience, however, the nuances of the lakehouse are still hard for LLMs to fully internalize: for example, after `run_2` (Figure 3) an agent could branch off P^{**}, fix the error, and run the pipeline again *from the second node*. By treating nested branches as "durable execution", agents could avoid re-computing everything at every trial. This observation led us to invest in purpose-built skills that complement model intelligence with specific operational knowledge [5].

6 RELATED WORK

The original semantics for *Git* are given in [7]: while code and data share similarities, table-backed primitives are novel. Our work aligns with database literature highlighting similarities between *Git* primitives and transactions. Dolt [2] is an OLTP-focused branching database, lacking a lakehouse-oriented merge pattern. A *Git* graph was modeled as a form of transaction in [14], but we focus on a different correctness boundary, as the dominant failure mode is partial publication rather than tuple-level anomalies. Nessie [3] provides Git-like versioning for lakehouse tables. `GitLake` differs in two ways: it optimizes common branch/catalog operations for high-frequency agentic workloads (up to 25x faster in standard CRUD-like requests), and it integrates versioning with pipeline execution so that the `run` API provides atomic publication across multi-table DAGs.

[2]https://github.com/BauplanLabs/OlapBranchBench

7 CONCLUSION

We described `GitLake`, the Git-for-data abstractions powering an agentic lakehouse. By lifting single-table evolution to commits, branches, merges, and reverts, we obtain a compact programming model for collaboration, reproducibility, rollback, and transactions. As of today, `Bauplan` has run millions of jobs across hundreds of thousands of data branches: the lessons we shared suggest that scaling systems to agentic scale will require rethinking most of the data stack.

REFERENCES

[1] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using lightweight formal methods to validate a key-value storage node in Amazon S3. (2021). https://www.amazon.science/publications/using-lightweight-formal-methods-to-validate-a-key-value-storage-node-in-amazon-s3

[2] DoltHub. 2026. Dolt. https://github.com/dolthub/dolt

[3] Dremio. 2024. Nessie. https://github.com/projectnessie/nessie.

[4] Shu Liu, Soujanya Ponnapalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, Matei Zaharia, Alvin Cheung, Natacha Crooks, Joseph E. Gonzalez, and Aditya G. Parameswaran. 2025. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. arXiv:2509.00997 [cs.DB] https://arxiv.org/abs/2509.00997

[5] 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

[6] 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

[7] Wouter Swierstra and Andres Löh. 2014. The Semantics of Version Control. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Portland, Oregon, USA) *(Onward! 2014)*. Association for Computing Machinery, New York, NY, USA, 43–54. https://doi.org/10.1145/2661136.2661137

[8] 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

[9] Jacopo Tagliabue, Federico Bianchi, and Ciro Greco. 2025. Trustworthy AI in the Agentic Lakehouse: from Concurrency to Governance. arXiv:2511.16402 [cs.AI] https://arxiv.org/abs/2511.16402

[10] 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

[11] 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

[12] Zhaoguo Wang, Chuzhe Tang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2024. Ad Hoc Transactions through the Looking Glass: An Empirical Study of Application-Level Transactions in Web Applications. *ACM Trans. Database Syst.* 49, 1, Article 3 (Feb. 2024), 43 pages. https://doi.org/10.1145/3638553

[13] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] https://arxiv.org/abs/2210.03629

[14] Gunce Su Yilmaz and Jens Dittrich. 2025. Generic Version Control: Configurable Versioning for Application-Specific Requirements. *15th Annual Conference on Innovative Data Systems Research (CIDR '25)* (2025).