Trustworthy AI in the Agentic Lakehouse: from Concurrency to Governance

Jacopo Tagliabue[*1], **Federico Bianchi**[2], **Ciro Greco**[1],

[1]Bauplan Labs
[2]Together AI
jacopo.tagliabue@bauplanbs.com, federico@together.ai, ciro.greco@bauplanbs.com

Abstract

Even as AI capabilities improve, most enterprises do not consider agents trustworthy enough to work on production data. In this paper, we argue that the path to trustworthy agentic workflows begins with solving the infrastructure problem first: traditional lakehouses are not suited for agent access patterns, but if we design one around *transactions*, governance follows. In particular, we draw an operational analogy to MVCC in databases and show why a direct transplant fails in a decoupled, multi-language setting. We then propose an agent-first design, `Bauplan`, that *reimplements* data and compute isolation in the lakehouse. We conclude by sharing a reference implementation of a self-healing pipeline, which couples agent reasoning with all the desired guarantees for *correctness* and *trust*.

1 Introduction

The lakehouse is the enterprise standard for data and AI workloads in the cloud (Zaharia et al. 2021). Notwithstanding the steady progress in coding and tool usage by Large Language Models (LLMs) (Shen 2024), production deployments of AI agents have rarely, if ever, targeted lakehouse use cases: even before considering the specificity of data *vs.* software engineering, the primary obstacles in industry are trust and governance: it is unclear how a human-centered lakehouse can provide the necessary guarantees for an agent-first world. To make a vivid example, imagine empowering a code assistant with access to your lakehouse: what if the agent drops a table? Or if it pollutes the lake with hallucinated data?

On the research side, the data management community is raising similar concerns: it is unlikely that data systems designed for small, expensive human teams can successfully adapt to the access patterns of a *swarm* of cheap AI agents (Tagliabue and Greco 2025). In particular, managing agents that run queries and build data pipelines means managing concurrency on a different scale than what traditional OLAP systems are designed for (Liu et al. 2025).

In this position paper, we argue that the industry and the research concerns above are best thought of as *two sides of the same coin*. In a nutshell, correct concurrent workloads require us to solve the isolation of data and compute through a unified API, reducing the governance challenge to the well-known pattern of API-based access control. If we focus our engineering effort on making sure multiple agents can work on the lakehouse without catastrophic consequences, we will get an easy and principled path to governance as well.

We know that this path is not impossible because – at the very least – *it has already happened once*; in particular, we first establish a connection with the theory of multi-version concurrency control (MVCC) (Bernstein, Hadzilacos, and Goodman 1987) in databases. Monolithic, SQL-only transactional databases (e.g., *Postgres*) have evolved as a sophisticated answer to concurrency issues – declarative abstractions, isolated processes, data snapshots allow correctness in the face of multiple users, with RBAC access control layered on top for governance (Ferraiolo et al. 2001). The lessons from the data management field are precious, but a direct mapping of the existing techniques will *not* work in a distributed, heterogeneous system such as a lakehouse. Alas, not all hope is lost. We describe novel abstractions and isolation primitives for an agent-first lakehouse `Bauplan`. We show how to accommodate the usage patterns of swarms of agents, and easily derive the required governance: a concise, narrow API surface is much easier to protect as opposed to the plethora of tools in traditional lakehouses.

This position paper is organized as follows: Section 2 uses MVCC to provide a concrete, successful mental model to guide our search for lakehouse primitives. In Section 3, we show that naively mapping MVCC concepts won't work, before describing our proposal in Section 4. We conclude with an open-source implementation of a complex data engineering task: as the landscape is evolving quickly, we do believe that a working system is a valuable reference for practitioners; our principles, however, are indeed tool-agnostic.

2 The MVCC mental model

Borrowing the metaphor from Arpaci-Dusseau and Arpaci-Dusseau (2018), the job of a database is to give each user the "illusion" of being the only user in play, while juggling safely and transparently the workloads of many such users. Correctness in the presence of concurrency is handled through *transactions* (Bernstein, Hadzilacos, and Goodman

[*]Corresponding author.

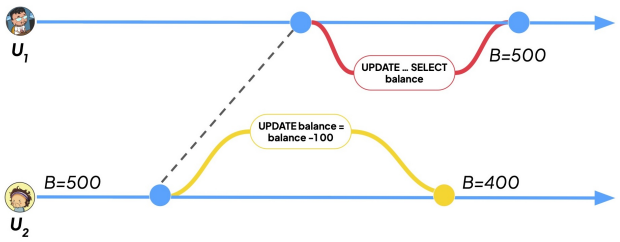

Figure 1: **The MVCC mental model**: U_1 starts his transaction, which at the end returns the value of B – effectively, his code works *as if* U_2's transaction had never happened.

1987): readers only see a consistent view of the data, and writers atomically publish all the changes or, in case of error, none of them. For the current purposes, we carry along the entire paper a three-way partition of the conceptual space: data isolation, compute isolation and programming abstractions.

2.1 Data Isolation

While the exact details are complex (Cerone, Bernardi, and Gotsman 2015), the basic model of data isolation is straightforward. Inside of transaction boundaries, processes read data always *as if* the database was frozen at the start; in order to do so, the system needs to maintain a reference to the state of the data at any point in time (a *snapshot*). In Fig. 1, U_1's read of U_2's value will return 500 as if U_2's update never started. On the write path, transactions are used to prevent incorrect updates and race conditions: two transactions won't both commit conflicting writes to the same key based on stale reads. Importantly, none of these storage-level implementations is exposed to the final user – a crucial design point to which we will return to below.

2.2 Compute Isolation

Each transaction gets processed in isolation, with no data sharing in between – from the point of view of the single transaction, it is *as if* that is the only process active at that time. In a SQL database there is no equivalent of package management, resulting in a small toolchain as opposed to a more heterogeneous compute environment where isolation is not as obvious.

2.3 Programming Abstractions

Traditional databases leverage a declarative language such as SQL which simplifies the relationship between the system and its users. Users express *what* data they wish to read or write, the query engine and the storage engine figure out *how* that should happen. This is a crucial design choice for at least two independent reasons:

- *correctness*: the complexity of coupling data and compute isolation within transactions is offloaded entirely to the platform, avoiding the common pitfalls of *ad hoc* transactions that plague even popular open source projects (Wang et al. 2024);
- *performance*: while each user is given the illusion of being alone, the platform may re-use transparently work done for one user for another one (e.g., caching).

A final consequence deserves a special mention: *trust*. Since the only way to interact with the system is at the logical layer, protecting the physical representation of the data is now as easy as implementing RBAC over users – unauthorized code does not even go beyond the planning module, ensuring no contamination at the physical layer.

As we shall see in the ensuing section, the properties that make MVCC workable in a monolithic database – uniform runtime, co-located control over data and execution – do not hold in lakehouses with heterogeneous runtimes and decoupled storage, so a direct transplant is insufficient.

3 From MVCC to the lakehouse

A transactional database is a vertically integrated system, which controls storage, compute and APIs. A lakehouse is a distributed, multi-language system built on the principle of storage-compute decoupling. The most typical workload is a data pipeline (van Renen et al. 2024), a DAG of transformations going from raw data in source tables to intermediate and final assets for downstream consumption (e.g., a dashboard, a RAG system): DAGs often happen on a schedule, and more generally, the overwhelming majority of OLAP use cases are not latency sensitive, but throughput oriented, as millions of rows are moved in a pipeline. While DAGs are often mentioned in the agentic literature (Zhang et al. 2025; Zheng et al. 2025), it is important to remark that our focus is on providing safe-by-design infrastructure for arbitrary OLAP workflows, rather than building directly agentic abstractions, such as symbolic workflow synthesis. As we shall see, adapting MVCC principles to this architecture is not straightforward.

3.1 Data Isolation

The storage layer of a lakehouse is typically built on Apache Iceberg tables, which guarantee transaction-like behavior on a *per-table* basis. However, single table guarantees are not enough when most writes are pipelines: if an agent writes wrong code at node 3 of a pipeline, nodes 1 and 2 are individually sound transactions, but the lake is in an inconsistent state: downstream readers have no protection, they may `JOIN` a new version of 2 with an old version of 3. We will come back to this distinction when discussing Fig. 2 below.

3.2 Compute Isolation

A data pipeline with three nodes may require SQL for the first one, Python 3.10 with `pandas` for the second, and Python 3.11 with `polars` for the third one. In other words, a lakehouse must support truly heterogeneous compute: on the one hand, building and running general-purpose Python code requires a novel platform-level toolchain; on the other, Python code introduces new vulnerabilities, such as installing malicious packages or accessing the internet. Traditional lakehouses have historically dealt with heterogeneous use cases by *adding* runtime options and execution workflows (Tapan Srivastava 2025): today, it is common to

move between different GUIs, APIs and interfaces to master the full lifecycle of a single pipeline. Unlike in the MVCC case, compute isolation is anything but obvious.

3.3 Programming Abstractions

Heterogeneous compute makes it harder to tie execution back to a consistency model (e.g., how do I reconcile a warehouse `INSERT` with a Spark job?). We highlight the issue with scattered abstractions by discussing mainstream reference implementations. Let's consider the *AWS* example for a data pipeline using Airflow (Thallam and Dominguez 2024)[1]:

Listing 1: A pre-processing node in an Airflow DAG

```
def preprocess(s3_in_url, s3_out_bucket,
     s3_out_prefix):
   # Do pre-processing and save the result in
   # "s3_out_bucket / s3_out_prefix"
   return "SUCCESS"

preprocess_task = PythonOperator(
   task_id="preprocessing",
   dag=dag,
   python_callable=preprocess.preprocess
   )
```

In this paradigm, we lose two properties we care about:

- *correctness*: there are no transactional guarantees anymore as the user is in charge both of the storage layer and the DAG process. It is therefore impossible to handle simple failures in a principled way: e.g., what is the state of the system if the process fails just before `return`?
- *performance*: the function writes its output as a side effect, preventing further optimizations on the outer loop (e.g., the DAG process cannot easily cache results).

The independence between compute (the DAG process) and data (the physical files in S3) has immediate implications for trust and governance: to be able to iterate over this DAG, an agent should be given access to i) the physical layer for the files, ii) Python-specific infrastructure to invoke the runtime, and iii) a SQL engine to perform exploratory queries when debugging. As the surface expands, governance becomes *exponentially* more difficult.

4 The Agentic Lakehouse

Naive application of the MVCC model to a distributed system will fail to achieve the goal of correctness-under-concurrency that we are pursuing for the agentic lakehouse, because physical decoupling and heterogeneous runtimes change the isolation contract. Not all is lost, though: we now describe `Bauplan` as a concrete implementation of those concepts, designed from first principles for the lakehouse.

4.1 Data Isolation

If every *write* in the lakehouse is immutably recorded, a series of snapshots tracks the history of tables through time. From this simple primitive, it is trivial to quickly model concurrent changes to *tables* in the same way *Git* models concurrent changes to *code* (Swierstra and Löh 2014) – a snapshot is identified by a *commit* with a parent, the `HEAD` is a symbolic reference to the current *branch* and resolving possible conflicts between branches happens within a *merge*. `Bauplan` adopts an efficient, copy-on-write branching mechanism, so that branching and merging can be done efficiently (less than a second) even on hundreds of tables and billions of rows. The crucial insight is once again related to the nature of data workloads: as pipelines are multi-table, they would span *multiple commits*. These abstractions can now model transaction boundaries across arbitrarily long DAGs by decoupling single table writes (commits) and multi-table atomic changes (merges) (Section 4.3).

4.2 Compute Isolation

As any lakehouse workload can be built out of functions with different latency constraints (Tapan Srivastava 2025), `Bauplan` adopts a Function-as-a-Service (FaaS) model as the underlying unified compute (Tagliabue, Caraza-Harter, and Greco 2024). The FaaS model (Hendrickson et al. 2016) perfectly aligns with our compute isolation needs – each function runs its own (containerized) Python / SQL runtime, independent of *any* other function and completely *isolated from the public internet*, with hardware, OS and runtime managed by the platform.

4.3 Programming Abstractions

A FaaS model maps immediately to DAG abstractions that are *functional*. Agents can code a DAG by chaining functions together:

Listing 2: Declarative I/O and infra-as-code

```
@bauplan.model(materialization='REPLACE', name='A')
@bauplan.python('3.10', pip={'pandas': '2.0'})
def parent(
   trips = bauplan.Model("taxi_trips"),
   zones = bauplan.Model("taxi_zones")
):
   # business logic only;
   # I/O is platform-mediated
   return trips.join(zones).do_something()

@bauplan.model(materialization='REPLACE', name='B')
@bauplan.python('3.11', pip={'polars': '0.88'})
def child(df = bauplan.Model("A")):
   return df.do_something()
```

The proposed abstractions take the typical infrastructure-as-code approach of industry FaaS and simplify it further by expressing declarative environments through a Python decorator (Tagliabue et al. 2023) that LLMs easily recognize. When compared with the code from Section 3.3, the second major difference is declarative I/O – functions accept tables (not files) as input, and output tables directly.

As compute and storage are decoupled, we need one more API to *bind* together declarative pipelines, FaaS execution and data branches. This is easily accomplished with a unified API to launch all workloads, i.e. `bauplan.run(my_pipeline)`. During a *run* on the

[1]Note that an equivalent example in the data engineering world can be found in the Databricks Best Practices with Scala (Databricks 2025).

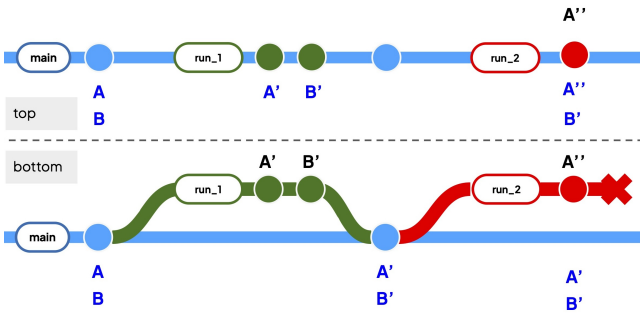

Figure 2: **Transactional pipelines.** *Top*: without coupling temporary branches with runs, `run_2` leaves *main* with a new version of A but an old version of B. *Bottom*: `Bauplan` *run* API guarantees atomic write of A' and B' on success, and isolation in case of failure.

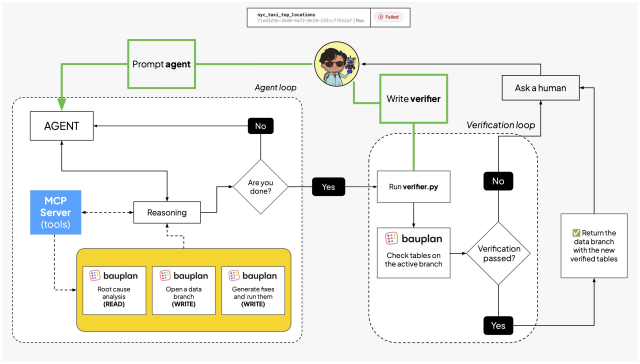

Figure 3: **Self-healing pipelines**: a ReAct loop is triggered on the agentic lakehouse – at the end, a verifier acts as a first sanity check on the agent's work, leaving to a human the final confirmation thanks to the *branch-then-merge* flow.

main data branch, `Bauplan` will: 1) automatically open a *temporary* branch; 2) fetch from S3 the rows corresponding to the DAG source tables; 3) run user code on the FaaS runtime and perform the required *writes* to S3; 4) *on success*, merge the temporary branch to *main*; *on failure*, leave the temporary branch open, and *main* untouched. Extending declarative abstractions to Python, and achieving data-compute logical unification (even with a physical decoupling) gets us back the desired properties:

- *correctness*: the platform supports transactional pipelines across languages and an arbitrary number of tables. Figure 2 illustrates the importance of conceiving transactions as an abstraction that embraces data *and* compute. Without enforcing temporary branches (Figure 2, top), a two-node pipeline failing after one node leaves *main* in an inconsistent state *even if* the single-table transaction was successful;
- *performance*: declarative abstractions leave the platform free to optimize execution *across agents* leveraging its privileged, centralized position.

4.4 From concurrency control to governance

As highlighted in Section 3, heterogeneous compute opens the door for agents to run untrusted logic. We are now in a position to tie back together our abstractions to trust and governance:

- *data governance*: on the *read* path, declarative I/O provides a unique SQL-like layer to check for authorization; on the *write* path, the *merge* primitive allows a fail-safe mechanism that safeguards production data;
- *compute governance*: as functions are fully isolated and do not require internet access, packages remain the only major vector of attack. Once again, the declarative APIs make it trivial to enforce rules at the platform level by checking a decorator against a whitelist: no agent is able to install dangerous packages;
- *API governance*: as all workloads are invoked through a simple set of unified APIs, securing access boils down to tried-and-tested patterns, such as RBAC with fine-grained permissions over data operations and executable units.

Solving concurrency and correctness is the key to unlocking an easy path to governance and trust: the agentic lakehouse is indeed a re-implementation of the successful MVCC patterns in the distributed lakehouse world.

4.5 A worked-out example: self-healing pipelines

As an example of a complex and very important use case that can be solved safely in the agentic lakehouse, we propose the implementation of a self-repairing pipeline.[2] Fig. 3 illustrates the high-level flow: a run failed, triggering an agentic resolution. A human engineer will prompt the agent for a resolution and produce a *verifier*, i.e. computational "acceptance criteria" registered within the platform itself. A ReAct loop (Yao et al. 2023) of reasoning over `Bauplan` APIs comprises the bulk of the agentic work, leveraging the abstractions above: code execution is sandboxed *and* every *write* happens on a data branch, leaving production safe and untouched – similarly to what happens with code reviews, the agent's output is a (data) *branch*. The Git-like primitives provide a natural hook for human-in-the-loop verification: first, the platform runs the verifier to independently establish that minimal criteria are met; second, the review-then-merge flow happens with the supervision of an engineer — on success, copy-on-write merge will ensure efficient publication of the agent's work into *main*.

5 Conclusion and future work

We advocate correctness by construction in an agent-first lakehouse: declarative I/O, temporary branches with atomic merge, and network-isolated functions bound to a single `run` API. As a concrete implementation, we described `Bauplan` and argued that pursuing safe concurrency in a lakehouse creates a clear path for principled governance. The major bottleneck to scaling trustworthy data engineering output is *not* intelligence anymore, but infrastructure.

[2] https://github.com/BauplanLabs/the-agentic-lakehouse

References

Arpaci-Dusseau, R. H.; and Arpaci-Dusseau, A. C. 2018. *Operating Systems: Three Easy Pieces*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform. ISBN 198508659X.

Bernstein, P. A.; Hadzilacos, V.; and Goodman, N. 1987. *Concurrency control and recovery in database systems*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN 0201107155.

Cerone, A.; Bernardi, G.; and Gotsman, A. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In Aceto, L.; and de Frutos Escrig, D., eds., *26th International Conference on Concurrency Theory (CONCUR 2015)*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 58–71. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-939897-91-0.

Databricks. 2025. The Big Book of Data Engineering – 3rd Edition.

Ferraiolo, D. F.; Sandhu, R.; Gavrila, S.; Kuhn, D. R.; and Chandramouli, R. 2001. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3): 224–274.

Hendrickson, S.; Sturdevant, S.; Harter, T.; Venkataramani, V.; Arpaci-Dusseau, A. C.; and Arpaci-Dusseau, R. H. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association.

Liu, S.; Ponnapalli, S.; Shankar, S.; Zeighami, S.; Zhu, A.; Agarwal, S.; Chen, R.; Suwito, S.; Yuan, S.; Stoica, I.; Zaharia, M.; Cheung, A.; Crooks, N.; Gonzalez, J. E.; and Parameswaran, A. G. 2025. Supporting Our AI Overlords: Redesigning Data Systems to be Agent-First. arXiv:2509.00997.

Shen, Z. 2024. LLM with tools: A survey. *arXiv preprint arXiv:2409.18807*.

Swierstra, W.; and Löh, A. 2014. The Semantics of Version Control. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, 43–54. New York, NY, USA: Association for Computing Machinery. ISBN 9781450332101.

Tagliabue, J.; Bowne-Anderson, H.; Tuulos, V.; Goyal, S.; Cledat, R.; and Berg, D. 2023. Reasonable Scale Machine Learning with Open-Source Metaflow. *ArXiv*, abs/2303.11761.

Tagliabue, J.; Caraza-Harter, T.; and Greco, C. 2024. Bauplan: Zero-copy, Scale-up FaaS for Data Pipelines. In *Proceedings of the 10th International Workshop on Serverless Computing*, WoSC10 '24, 31–36. New York, NY, USA: Association for Computing Machinery. ISBN 9798400713361.

Tagliabue, J.; and Greco, C. 2025. Safe, Untrusted, "Proof-Carrying" AI Agents: toward the agentic lakehouse. arXiv:2510.09567.

Tapan Srivastava, C. G., 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*.

Thallam, R.; and Dominguez, M. 2024. Build end-to-end machine learning workflows with Amazon SageMaker and Apache Airflow.

van Renen, A.; Horn, D.; Pfeil, P.; Vaidya, K. E.; Dong, W.; Narayanaswamy, M.; Liu, Z.; Saxena, G.; Kipf, A.; and Kraska, T. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. In *VLDB 2024*.

Wang, Z.; Tang, C.; Zhang, X.; Yu, Q.; Zang, B.; Guan, H.; and Chen, H. 2024. Ad Hoc Transactions through the Looking Glass: An Empirical Study of Application-Level Transactions in Web Applications. *ACM Trans. Database Syst.*, 49(1).

Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; and Cao, Y. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629.

Zaharia, M. A.; Ghodsi, A.; Xin, R.; and Armbrust, M. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *Conference on Innovative Data Systems Research*.

Zhang, J.; Xiang, J.; Yu, Z.; Teng, F.; Chen, X.; Chen, J.; Zhuge, M.; Cheng, X.; Hong, S.; Wang, J.; Zheng, B.; Liu, B.; Luo, Y.; and Wu, C. 2025. AFlow: Automating Agentic Workflow Generation. arXiv:2410.10762.

Zheng, C.; Chen, J.; Lyu, Y.; Ng, W. Z. T.; Zhang, H.; Ong, Y.-S.; Tsang, I.; and Yin, H. 2025. MermaidFlow: Redefining Agentic Workflow Generation via Safety-Constrained Evolutionary Programming. arXiv:2505.22967.