"Skill Issues": Data-Centric Optimization of Lakehouse Agents

Nicole Rose Schneider
nsch@umd.edu
University of Maryland
USA

Davide Ghilardi
davide.ghilardi@unimib.it
Università Milano Bicocca
Italy

Giacomo Piccinini
giacomo.piccinini@bauplanlabs.com
Bauplan Labs
USA

Jacopo Tagliabue
jacopo.tagliabue@bauplanlabs.com
Bauplan Labs
USA

ABSTRACT

Coding agents are becoming users of data infrastructure, but their success depends not only on model quality: it also depends on the skills and environment files that teach agents how to use a system. We study how to optimize these artifacts for agents operating on a branching lakehouse, `Bauplan`. In our setting, headless APIs and Git-like data primitives expose data workflows through code, branches, commits, and merges. Our central observation is that a branching lakehouse turns data-agent evaluation from an output-matching problem into a state-verification problem: agent-generated pipeline code induces concrete, inspectable lakehouse changes. We present a data-centric optimization pipeline that generates task-verifier pairs, executes candidate skills in isolated sandboxes, and scores trajectories using both trace-level signals and programmatic checks over lakehouse state. In a preliminary evaluation on hundreds of tasks, optimized skills improve held-out reward by 31.9%. These results suggest that write-path data workflows provide a useful substrate for optimizing agent skills beyond read-only tasks.

VLDB Workshop Reference Format:
Nicole Rose Schneider, Davide Ghilardi, Giacomo Piccinini, and Jacopo Tagliabue. "Skill Issues": Data-Centric Optimization of Lakehouse Agents. VLDB 2026 Workshop: ADS 2026: The Joint Workshop on Agentic Data Systems and Data-Centric AI (The 1st ADS & 3rd DATAI).

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

1 INTRODUCTION

> "Make something idiot proof, and somebody will come up with a better idiot." *Anonymous*

Whether they work synchronously with humans [15] or asynchronously in a loop [34], it is clear that coding agents will soon become the primary users of cloud infrastructure. Experts point out that the affordances in traditional Online Analytical Processing (*OLAP*) systems make agents ineffective and unsafe on the lakehouse; `Bauplan` was designed as an agent-first data platform according to the "AI Overlord Manifesto" [21]: fast, safe execution coupled with Git-like collaboration primitives [24, 25].

Running `Bauplan` gives us a privileged perspective on the challenges of scaling data agents to production settings. Even as the raw performance of Large Language Models (*LLM*s) increases, the variance in agentic success is huge: `Bauplan` APIs are not yet well represented in the training set, users' prompting abilities vary widely and the new affordances call for revised best practices. Following common LLM-agent practice [3], we shared standardized Markdown files (*skills*) with users to both lower the barrier to using the platform *and* ensure a consistent standard in how code is written, tested, and deployed.

Customer choices in terms of models, coding assistants, and usage patterns have evolved quickly in recent months, so manually curating skills is no longer a sustainable process. More importantly, we found that this is not merely a prompt-engineering problem: when agents operate on production data systems, the textual artifacts surrounding them become part of the system control surface, and need to be optimized and evaluated as such. As it turns out, turning our initial prompting strategy into a scalable and reliable process required assembling a complex and multi-faceted set of capabilities: data mining over traces, user interviews on domain knowledge, LLM-as-judge calibration and more. In *this* paper, we share our skill optimization journey with the community. In particular, we summarize our contributions as follows:

(1) through the privileged lens provided by running thousands of agentic workloads per day on `Bauplan`, we motivate the importance of skills to support common multi-turn agentic use cases seen in production, such as ETL, Write-Audit-Publish, and data exploration;
(2) we share the optimization loop for `Bauplan`'s skills, which improved held-out reward over manually curated ones by 31.9%. Perhaps surprisingly, skills belonging primarily to complex workflows could not be optimized in isolation, suggesting worthwhile future work in jointly optimizing skills, which our dataset supports. Since closed-source LLMs are frozen, and optimization frameworks are now common, we embrace the data-centric AI methodology [35] and focus on the surrounding data tasks. Great care is taken in the dataset generation process, as a result of trade-offs among the nuances of synthetic data generation, production traces, data engineering knowledge and error analysis. We release the

Proceedings of the VLDB Endowment. ISSN 2150-8097.

full setup to the community, stripped of Bauplan-specific features, as an open-source contribution;

(3) we highlight the interplay between Bauplan affordances, the optimization loop and the final skills. For example, a statically checkable SDK reduces round-trips; progressive CLI discovery allows for concise skills; more importantly, the isomorphism between pipeline *code* and *data commits* grounds fine-grained verification of write-path behavior (Section 3.2.2).

Since unsupervised workloads on production data are dangerous in most OLAP systems [21], lakehouse agents today are still in their infancy and mostly focused on individual tasks in the *read* path (text-to-SQL [22], data exploration [6]). While our (reproducible) experimental setup is tailored to a specific set of APIs, our strongest contribution is linking Git-for-data to the optimization of compound AI systems for OLAP use cases: as such, alternative implementations of the same ideas will benefit from the insights we share. For these reasons, we believe our work to be useful to a wide set of practitioners: both as a reference implementation of a full optimization pipeline and as an example of a realistic coding setup specifically targeted at the full life-cycle of data management, including potentially disruptive operations such as *writes* and *deletes*. In contrast to evaluations that stop at read-only SQL answers, our setting requires checking whether an agent changed the lakehouse state in the intended way.

2 SETUP AND PROBLEM STATEMENT

2.1 The agentic setup

Fig. 1 illustrates the standard setup for users of Bauplan. When faced with a data task, such as analyzing data, building pipelines end-to-end, or ingesting data safely, a user can pick a coding assistant (e.g. *Claude Code* [19]) wired to *Opus* and ask it to perform the task. Following industry best practices [3, 18], we provide the user with *skills* and a *CLAUDE.md*[1] to improve performance on data-specific scenarios. Skills prepare the agent to solve a problem "by injecting procedural knowledge, modifying execution context, and enabling progressive disclosure of information" [32]. Note that the agent could also work fully asynchronously: the same user choices and degrees of freedom apply, but instructions are possibly even more important there because of the autonomy in execution.

When optimizing this setup against a series of tasks, "backpropagation" can only happen at the level of these textual hyperparameters, as everything else is held constant by user choices. Our research question is then easily stated: how can we optimize these parameters – *skills* and *CLAUDE.md* – to improve the performance of the "compound AI system" [7] on data use cases?

Answering this question will involve generating a dataset of use cases (Section 3.2), scoring outcomes at scale (Section 3.2.2), and choosing an optimization method (Section 3.3). Before diving into these details, we provide a small introduction to Bauplan, as the platform ergonomics not only exemplify its agentic nature, but also significantly shape the optimization process itself.

[1]In a non-Anthropic setup, we use the cross-provider *AGENTS.md* standard instead: for simplicity, we use the Claude-specific lexicon throughout the paper.

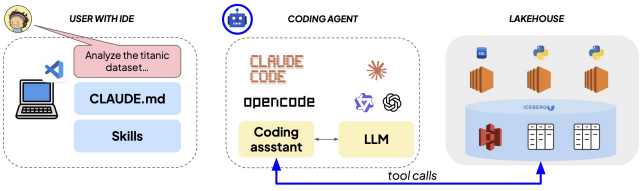


Figure 1: Coding agent setup. The user chooses a task and provisions a coding agent (e.g. Claude Code), which in turn is wired to a Large Language Model (e.g. Opus): tool calls (Bash or Python execution) invoke lakehouse-specific APIs and feed the result back to the agent. *CLAUDE.md* and *Skills* are the only "hyperparameters" under our control: how do we optimize them?

2.2 Bauplan overview

Bauplan is an agent-first lakehouse, built on the separation of storage (Apache Iceberg [4] on S3) and compute (SQL and Python). We refer the reader to [26] for a full description, and survey here two distinctive features of Bauplan (vis-a-vis a traditional lakehouse), which are particularly relevant for a successful verification process (Section 3.2.2):

(1) **headless**: *all* interactions with the platform (running jobs, creating tables, inspecting logs, etc.) are API-based, as wrapped by a CLI (for Bash use cases) and a Python SDK (for scripting use cases); this matters because it allows agents to manage the *full data life-cycle* through code, and because it allows us to programmatically verify lakehouse changes at scale;

(2) **git-for-data**: Bauplan re-implements Git primitives over data assets, allowing *commits*, *branches*, and *merges* on lakehouse tables [28]. Git-like APIs allow safe, concurrent changes and prevent unverified AI code from reaching production; this matters because it allows us to deterministically map API calls to data changes, which in turn is crucial for verification.

To get a sense of the code-first nature of the platform and to understand the *effects* of a run, we can inspect two snippets, a DAG and a run orchestration. The first is a one-node DAG taking `source` (an Iceberg table) as input, and producing `table-1` through function `f`:

Listing 1: A one-node Python DAG.

```
class Total(BauplanSchema):
    category: str
    total_amount: float

@model(name="table-1", materialize="REPLACE")
@python("3.12", pip={"polars": "1.27"})
def f(df=source) -> Total:
    tbl = df.group_by("category").agg(pl.col("amount").sum())
    return tbl
```

The abstractions are hopefully self-explanatory. We only briefly highlight the declarative nature of both infrastructure and I/O, which is an ideal target for code generation. *Physical* operations (installing packages, reading data from S3, or writing data to S3) happen in "platform space", so that user space only expresses *logical* operations and transformation code is executed in secure and

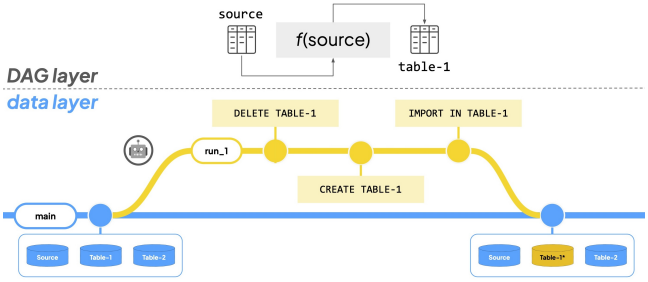


Figure 2: Pipeline code maps to *verifiable* lakehouse changes. A one-node DAG transforming *source* into *table-1* (Listing 1) runs on a data branch: a successful run will alter the lake state in a predictable and verifiable way. Here, commits in the branch correspond to *writes* (delete, create, import), and one commit in `main` corresponds to the *merge*, which atomically updates only the snapshot of *table-1*. The same APIs can be used by a verifier to check data changes commit by commit.

sandboxed compute [27]. Importantly, DAGs live in a complex lakehouse environment, which is *also fully programmable*. Interactive use cases are typically managed from the CLI (e.g. `bauplan branch create feature-1`, `bauplan run`), but complex flows are built by interleaving lakehouse commands and Python control flow, leveraging a fully typed SDK:

Listing 2: Python SDK example.

```
# 1) instantiate the client
client = bauplan.Client()
# 2) create a development branch from main
dev_br: Branch = client.create_branch("dev_br", from_ref="main")
# 3) run a DAG on the branch
run_state = client.run("pipeline/", ref=dev_br)
# 4) merge the branch into production (on success)
if run_state.success():
    client.merge(dev_br, into="main")
    client.delete_branch(dev_br)
```

Listing 2 is the "outer loop" for Listing 1, executing the transformation defined there: again, the abstractions should be self-explanatory, and we only highlight that runs are embedded in a data branching workflow that is very similar to the typical Git flow for code.

What happens when `client.run` above gets called? The code is shipped to cloud workers, and logs are streamed back for the agent to observe and react [26]. Crucially, every *write* in the execution corresponds to a data *commit*, i.e. an immutable, Git-like reference to the state of the lakehouse at that time [24]. Fig. 2 illustrates the code-data isomorphism: any actor (human or LLM) reading Listing 1 should be able to enumerate what a successful run would entail in the shape of *commits* that mark how the lake changes. When we ask an LLM to generate task-verifier pairs (Section 3.2), this allows us to *precisely* verify whether the agent did indeed perform the work as expected, minimizing "false positives" and cheating behavior (Section 3.2.2). This is the core difference between optimizing skills for a branching lakehouse and generic prompt engineering: the system exposes concrete write-path effects that can be checked after execution.

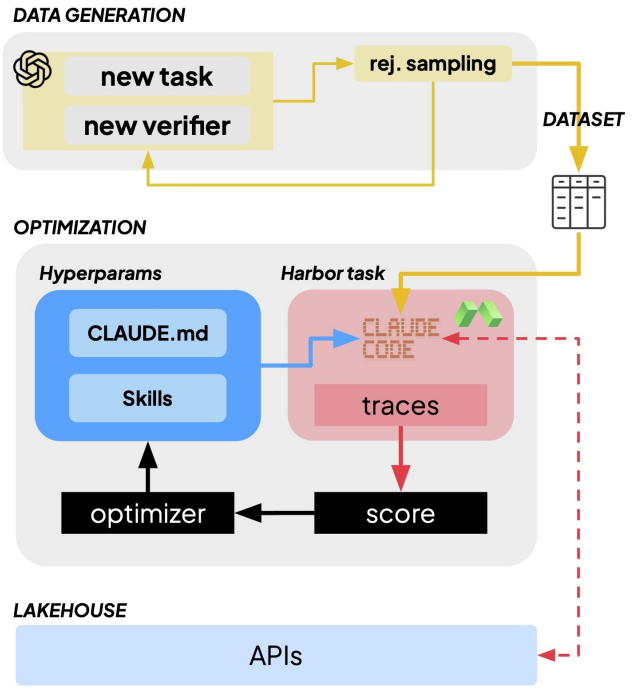


Figure 3: High-level system architecture. An LLM-powered data generation module builds a dataset with *task-verifier* pairs. The optimization loop runs the target coding setup on the tasks with the current *skills* – tools connect to `Bauplan` to execute actions and get feedback. Metrics are then computed and new skills are produced by the optimizer, and the loop continues.

3 SKILL OPTIMIZATION

3.1 Architecture

Fig. 3 shows the high-level architecture of the optimization pipeline. First, an LLM-powered data generation module (Section 3.2.1) takes as input a taxonomy of scenarios (data ingestion, report building, pipeline debugging etc.) and personas (data engineer, data scientist, data analyst etc.) and `Bauplan`-specific prompt instructions to generate *task-verifier* pairs (Section 3.2.2): as discussed (Fig. 2), a DAG that would solve the task can be mapped deterministically to lakehouse checks, so that tasks can have fine-grained, programmatic verification.

Once the dataset is built, we use Harbor [13] to orchestrate Modal sandboxes initialized with Markdown files and the appropriate coding assistant package. Table 1 summarizes the life-cycle of each benchmark task: the harness creates an isolated writable branch, captures the baseline lakehouse state, lets the agent act, validates the outcome, and tears down any temporary state. This fixed life-cycle is what allows the same optimization loop to evaluate both read-only and write-path tasks safely. Traces collected from these trajectories, as well as a final score, are then fed into an optimization function, which generates new Markdown files for another round.

Table 1: Lifecycle of one benchmark task in the optimization harness.

#	Step	What it does
1	Create `<username>.main`	Branches the agent's writable `<username>.main` off `main`.
2	Prep initial state	Drops any tables that are explicitly required by the dataset entry to be absent on `<username>.main`.
3	Capture baseline	Snapshots the latest commit on `main` and `<username>.main`.
4	Agent runs	Runs the Claude agent on the task prompt.
5	Validation	Scores the result against the dataset entry's `end_conditions`.
6	Teardown	Deletes post-baseline `<username>.*` branches, then drops `<username>.main`.

3.2 Data generation

The optimization loop in Fig. 3 requires a benchmark that is both realistic enough to exercise the skills used in production, and structured enough to support automatic verification. We therefore generate *task-verifier pairs*: each benchmark entry contains not only a user-facing prompt, but also the initial lakehouse state, the expected platform interactions, forbidden actions, response-level checks, and a programmatic validation script. This follows a data-centric view of agentic improvement [35], with a strong focus on quality *at scale*: while human judgment and domain knowledge play a role in the initial scenario design, the final dataset is generated in a loop until it satisfies the structural and semantic constraints required by the downstream optimization task.

3.2.1 Scenario generation. Table 2 reports representative counts for compute and branch operations as collected from a May 2026 sample of anonymized production logs. The table provides an empirical prior over agentic workflows: imports, queries, and pipeline runs all occur frequently on the compute side, while branch deletion and merge operations dominate the branch-management side. This confirms that realistic agent workloads are not limited to read-only SQL, but include useful but potentially dangerous write-path operations. Crucially, we note that raw frequencies are not a sufficient benchmark specification. First, traces are partially shaped by the current skills, so they may over-represent APIs described well and under-represent poorly documented behaviors. Second, some rare operations are semantically important for safety. We therefore use the counts as a "conceptual prior" for scenario design, not as the final task mixture: common APIs should appear often enough to reflect real usage, but rare APIs must still be covered by construction.

Notably, looking only at API counts flattens multi-step behavior, so other important priors for scenario design are *sessions*, i.e. series of commands issued when performing a coherent multi-step process. In particular, improvements in LLMs' theory of mind [8] set the stage to perform inverse planning [5, 11] from session-level API traces; e.g., a sequence of branch creation, import, validation query, and merge suggests a safe ingestion workflow. A model fed with sessions can infer the kind of intention a user may have had, and these inferred intentions become templates for task generation. Putting together API counts and session-level insights, the final dataset is synthetic in form, but anchored in real platform usage and in the operational semantics of the lakehouse.

The resulting taxonomy has two axes. The first is the type of lakehouse work: read-only tasks, such as data exploration, text-to-SQL, and data-management inspection; and write-path tasks, such as safe ingestion, new pipeline construction, adding expectations, modifying or debugging pipelines, branch operations, and end-to-end analysis. The second axis is the shape of the user request. Each task is tagged with a difficulty level and a prompt-specificity level: vague prompts model novice users, medium prompts model typical practitioners, and detailed prompts model expert users who specify tables, columns, output schemas, and acceptance criteria. This is important because the skills must help agents not only when the desired workflow is explicit, but also when the user request has to be translated into the right workflow. This approach may be seen as a further natural generalization of taxonomy-based dataset generation [9]: on top of generating tasks by progressively refining intent into scenarios and then into tasks ("depth"), and by progressively covering the design space with additional variations ("coverage"), we also generate tasks by varying a user-specific dimension ("explicitness").

Given this taxonomy, an LLM generator emits one structured benchmark entry at a time through a typed tool call. The generator is grounded with Bauplan-specific context (skills, SDK, CLI) and the available datasets: for ingestion tasks, the prompt refers to the S3 URI directly; in any other case, the entry declares any pre-loaded tables. Each entry is syntactically validated and then passed through a thorough set of checks: e.g. commands must correspond to real CLI arguments. For read-only categories, the validator requires at least one write-blocking forbidden action and at least one no-mutation state assertion, preventing read tasks from silently allowing destructive behavior. Invalid entries are rejected and the generator samples again until enough valid tasks are collected [30]. Before optimization begins, domain experts carefully sample tasks and evaluate their quality in detail, to verify both prompt realism (i.e., is this a realistic user scenario and a properly worded request?) and verifier thoroughness (i.e. do expected commands and SDK checks prove that the agent executed what the user wanted?). We refer readers to the open-source repository and to Appendix A for full-fledged examples.

3.2.2 Verification. Our harness verifies each trajectory at three levels: tool traces, lakehouse state, and final response quality. This is conceptually similar to executable software-engineering benchmarks such as SWE-bench [17], where candidate code changes are evaluated by running tests. In our setting, however, the executable object is not only a source-code patch: it is a sequence of agent actions whose effects include branches, commits, tables, schemas, rows, and merges. We refer the reader to the repository for the full list of checks, and survey here the high-level features of each category:

- **Trace-level verification**: the verifier checks whether the agent used the expected skills and tool calls, and avoided forbidden actions (e.g. `bauplan run` in read-only). Expected operations are checked against both Bash calls and Python scripts (with the appropriate CLI-to-SDK syntax conversion): in most cases, the model is left free to choose between the CLI and the SDK with no penalties.
- **Lakehouse-state verification**: the verifier checks the *data-specific* preconditions and postconditions. For example, simple assertions cover branch existence and no-change guarantees, while more complex programmatic checks validate runs commit-by-commit (Fig. 2). For write-path tasks whose final state depends on the user name, the validation script uses the SDK to discover the username dynamically and assert the intended state. Based on our experience, only frontier closed models such as GPT-5.5 were able to properly "one-shot" the entire task generation with the required internal consistency (at the cost of high latency): we leave the exploration of smaller models and multi-agent orchestration to future work.
- **Response-level verification**: some tasks do not modify the lakehouse state, nor do they have a structured output: exploration, for example, typically ends with a free-form English response. The verifier therefore combines hard and soft checks. Hard constraints are deterministic checks, e.g. string matching and regex. Soft constraints are rubric-style criteria graded by an LLM judge [36], checking whether the answer covers the requested dimensions and states the right caveats.

While this level of detail-oriented verification and thoroughness may seem unnecessary for special-purpose data agents, in our view this approach is the heart of the optimization process. Since LLMs and optimization libraries can be (and often are) black-box components, iterating on data quality provides the needed leverage to steer model behavior on real-world, niche scenarios. Two Bauplan features (Section 2.2) proved to be crucial for the data-centric methodology: first, the SDK allows verifiers to be written using the same methods that a data agent would use; second, git-for-data makes write-path evaluation deterministic and fine-grained: a candidate skill is rewarded only when the agent changes the lakehouse in the intended way.

3.3 Optimization

We used GEPA Optimize Anything [1] to optimize the text contents of the skills. Inputs to the optimization function are

(1) A seed skill
(2) An evaluation function
(3) A description of the goal

The seed skill is the initial human-curated skill that we aim to optimize. The evaluation function takes candidate skills generated by GEPA and runs a sandboxed agent on dataset tasks that involve the relevant skill. The validation is done after the agent runs, and it is used to score the pipeline with a reward between 0 and 1, based on how many validation checks pass. Based on the reward score and any error traces generated as actionable side information, GEPA generates new candidates and the process continues.

Table 2: Usage of `Bauplan`'s abstractions in a May 2026 sample of anonymized production logs. Percentages are computed within each API group and rounded to whole numbers.

Abstraction	Description	Pct.
Compute APIs		
Import	append data to a table	40%
Query	execute a SQL query	35%
Run	run a pipeline	25%
Branch APIs		
Merge	merge a ref into a target branch	43%
Delete	delete a branch	37%
Create	create a new branch from a ref	11%
Get	get details about a specific branch	8%
List	list branches	1%

Importantly, the architecture is designed to be modular (Fig. 3): as long as modules respect the declared interfaces (see the open source repository), it is easy to plug in different algorithms keeping the rest constant. We leave experimentation with different frameworks [20, 23] to future work.

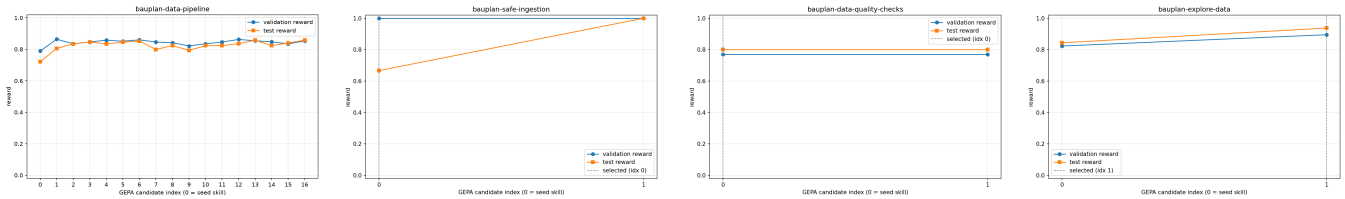
Table 3: Dataset composition by type and scenario.

Scenario	Task type	Count
Read	Exploration	102
	Assessment	101
	Data quality check	103
Write	New pipeline	171
	Fix table	100
	Ingestion	106
Total		**683**

4 EXPERIMENTS

Our final dataset is summarized in Table 3, comprising a total of 683 tasks with varying degrees of difficulty (Appendix A). For each of the 6 skills we attempt to optimize, we extract the pool of dataset entries that require the given skill, then split the pool into train, validation, and test sets using a 70% – 15% – 15% split, with a minimum of 1 entry for the validation and test sets in case of a small pool. GEPA reflects and mutates on the train set and uses the validation set to select its best candidate. The test set is held out and used to evaluate performance. Because we evaluate only one coding-agent/LLM configuration, we consider these results important but preliminary evidence of the usefulness of the optimization loop.

Our primary metric is reward (defined as the *fraction of validation checks passed*) measured on the validation and held-out test sets; we also record time and cost. We evaluate each of these metrics for every candidate skill GEPA generates. The task LLM is *claude-sonnet-4-6*, with Claude Code as the assistant. The initial set of skills comprises six different skills: *bauplan-data-pipeline*, *bauplan-safe-ingestion*, *bauplan-data-quality-checks*, *bauplan-data-assessment*, *bauplan-explore-data*, *bauplan-debug-and-fix-pipeline*.



(a) bauplan-data-pipeline **(b) bauplan-safe-ingestion** **(c) bauplan-data-quality-checks** **(d) bauplan-explore-data**

Figure 4: GEPA optimization curves for representative skill components: validation and held-out test reward by candidate index.

Results are shown in Fig. 4. For the skills we are able to optimize, GEPA found a candidate skill that not only performs better on the validation tasks, but also improves held-out reward (relative to the seed skills) by 31.9% on average, suggesting that the optimized skills generalize beyond the validation tasks.

Qualitatively, the optimization process adds detail to the skill and appropriately edits the skill descriptions when validation checks fail due to the agent not invoking the expected skill. On average, the optimized skills were 93.3% longer (in terms of character count) than the seed skills. Improvements included adding phrases to encourage the agent not to skip steps (e.g., `## Required workflow (do not skip steps)`) and spelling out "gotchas" on the CLI stateful semantics for some commands (e.g., `merge <branch> merges into the currently checked-out branch. So check out your personal main first, then merge with NO target argument`).

In the cases where GEPA did not find a better candidate skill than the seed skill, we found that the dataset tasks associated with those skills were multi-skill tasks, where solving the task required correctly leveraging multiple skills. This result points to a clear line of future work: jointly optimizing skills. Our approach optimizes each skill in isolation, but the dataset supports complex tasks where skills may need to be adapted jointly to enable the agent to complete a correct workflow, end-to-end.

5 RELATED WORK

Recent work has optimized skills across tasks as diverse as software engineering, question answering, mathematical reasoning, web navigation, and tool use [2, 10, 12, 14, 16, 29]. Work on synthetic data generation and executable benchmarks similarly shows the value of generated tasks with verifiable outcomes [9, 17, 30, 33]. Our setting differs in being data-centric and lakehouse-specific: skills are optimized for concrete data engineering workflows such as ingestion, pipeline construction and debugging. Unlike read-path evaluations such as data-science agents [6, 22], our benchmark is grounded in end-to-end data management use cases and traces from an agent-first production system at scale. In particular, we make the key observation that fine-grained evaluation of potentially disruptive data operations is possible only thanks to the isomorphism between agent-generated pipeline code and observable lakehouse state.

6 CONCLUSION AND FUTURE WORK

We introduce a data-centric methodology for skill optimization serving data agents on a branching lakehouse. Combining real-world production traces, purpose-built agentic infrastructure, domain knowledge and a black-box optimizer, we provide preliminary evidence that this process can automatically improve compound AI systems. To the best of our knowledge, we have shown for the first time how to perform verification at scale for destructive data engineering tasks on production data: reliable, programmatic scoring of complex tasks would not have been possible without Bauplan's vertical integration of agentic affordances (everything-as-code and git-for-data, Section 2.2). By sharing our data generation process and releasing an open-source implementation, we believe our methodology can easily generalize to a broader set of use cases. For example, in ongoing work within our lab we are leveraging this process to fine-tune coding models: as LLMs are prone to reward hacking [31], fine-grained verification is a precondition for effective reinforcement learning.

REFERENCES

[1] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. 2025. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. In *First Workshop on Foundations of Reasoning in Language Models.* https://openreview.net/forum?id=4oo6XTL6Oj

[2] Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. 2026. Evoskill: Automated skill discovery for multi-agent systems. *arXiv preprint arXiv:2603.02766* (2026).

[3] Anthropic. 2025. The Complete Guide to Building Skills for Claude. https://resources.anthropic.com/hubfs/The-Complete-Guide-to-Building-Skill-for-Claude.pdf

[4] Apache. 2024. Iceberg. https://github.com/apache/iceberg.

[5] Chris L Baker, Rebecca Saxe, and Joshua B Tenenbaum. 2009. Action understanding as inverse planning. *Cognition* 113, 3 (2009), 329–349.

[6] Ke Chen, Peiran Wang, Yaoning Yu, Xianyang Zhan, and Haohan Wang. 2025. Large Language Model-based Data Science Agent: A Survey. arXiv:2508.02744 [cs.AI] https://arxiv.org/abs/2508.02744

[7] Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, Matei Zaharia, James Zou, and Ion Stoica. 2025. Optimizing Model Selection for Compound AI Systems. arXiv:2502.14815 [cs.AI] https://arxiv.org/abs/2502.14815

[8] Ruirui Chen, Weifeng Jiang, Chengwei Qin, and Cheston Tan. 2025. Theory of Mind in Large Language Models: Assessment and Enhancement. arXiv:2505.00026 [cs.CL] https://arxiv.org/abs/2505.00026

[9] Tim R. Davidson, Benoit Seguin, Enrico Bacis, Cesar Ilharco, and Hamza Harkous. 2026. Reasoning-Driven Synthetic Data Generation and Evaluation. arXiv:2603.29791 [cs.AI] https://arxiv.org/abs/2603.29791

[10] Yudong Gao, Zongjie Li, Zimo Ji, Pingchuan Ma, Shuai Wang, et al. 2026. Skillreducer: Optimizing llm agent skills for token efficiency. *arXiv preprint arXiv:2603.29919* (2026).

[11] Rebekah A. Gelpí, Eric Xue, and William A. Cunningham. 2025. Towards Machine Theory of Mind with Large Language Model-Augmented Inverse Planning.

arXiv:2507.03682 [cs.AI] https://arxiv.org/abs/2507.03682
[12] Jingzhi Gong, Ruizhen Gu, Zhiwei Fei, Yazhuo Cao, Lukas Twist, Alina Geiger, Shuo Han, Dominik Sobania, Federica Sarro, and Jie M Zhang. 2026. SkillMOO: Multi-Objective Optimization of Agent Skills for Software Engineering. *arXiv preprint arXiv:2604.09297* (2026).
[13] Harbor Framework Team. 2026. *Harbor: A framework for evaluating and optimizing agents and models in container environments.* https://github.com/harbor-framework/harbor
[14] Chenyi Huang, Haoting Zhang, Jingxu Xu, Zeyu Zheng, and Yunduan Lin. 2026. Bilevel Optimization of Agent Skills via Monte Carlo Tree Search. *arXiv preprint arXiv:2604.15709* (2026).
[15] 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
[16] Hyunmin Hwang, Jaemin Kim, Choonghan Kim, Hangeol Chang, and Jong Chul Ye. 2026. AgentPSO: Evolving Agent Reasoning Skill via Multi-agent Particle Swarm Optimization. *arXiv preprint arXiv:2605.08704* (2026).
[17] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *The Twelfth International Conference on Learning Representations*.
[18] Yuan Liang, Ruobin Zhong, Haoming Xu, Chen Jiang, Yi Zhong, Runnan Fang, Jia-Chen Gu, Shumin Deng, Yunzhi Yao, Mengru Wang, Shuofei Qiao, Xin Xu, Tongtong Wu, Kun Wang, Yang Liu, Zhen Bi, Jungang Lou, Yuchen Eleanor Jiang, Hangcheng Zhu, Gang Yu, Haiwen Hong, Longtao Huang, Hui Xue, Chenxi Wang, Yijun Wang, Zifei Shan, Xi Chen, Zhaopeng Tu, Feiyu Xiong, Xin Xie, Peng Zhang, Zhengke Gui, Lei Liang, Jun Zhou, Chiyu Wu, Jin Shang, Yu Gong, Junyu Lin, Changliang Xu, Hongjie Deng, Wen Zhang, Keyan Ding, Qiang Zhang, Fei Huang, Ningyu Zhang, Jeff Z. Pan, Guilin Qi, Haofen Wang, and Huajun Chen. 2026. SkillNet: Create, Evaluate, and Connect AI Skills. arXiv:2603.04448 [cs.AI] https://arxiv.org/abs/2603.04448
[19] Jiacheng Liu, Xiaohan Zhao, Xinyi Shang, and Zhiqiang Shen. 2026. Dive into Claude Code: The Design Space of Today's and Future AI Agent Systems. arXiv:2604.14228 [cs.SE] https://arxiv.org/abs/2604.14228
[20] Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z. Pan, Alexander Du, Kurt Keutzer, Alvin Cheung, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. 2026. EvoX: Meta-Evolution for Automated Discovery. arXiv:2602.23413 [cs.LG] https://arxiv.org/abs/2602.23413
[21] 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.AI] https://arxiv.org/abs/2509.00997
[22] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuxin Zhang, Ju Fan, Guoliang Li, Nan Tang, and Yuyu Luo. 2025. A Survey of Text-to-SQL in the Era of LLMs: Where are we, and where are we going? arXiv:2408.05109 [cs.DB] https://arxiv.org/abs/2408.05109
[23] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv:2506.13131 [cs.AI] https://arxiv.org/abs/2506.13131
[24] 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
[25] 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
[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] Yash Vishe, Rohan Surana, Xunyi Jiang, Zihan Huang, Xintong Li, Nikki Lijing Kuang, Tong Yu, Ryan A Rossi, Jingbo Shang, Julian McAuley, et al. 2026. Skill-R1: Agent Skill Evolution via Reinforcement Learning. *arXiv preprint arXiv:2605.09359* (2026).
[30] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning Language Models with Self-Generated Instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*. https://doi.org/10.18653/v1/2023.acl-long.754
[31] Jiaxin Wen, Ruiqi Zhong, Akbir Khan, Ethan Perez, Jacob Steinhardt, Minlie Huang, Samuel R. Bowman, He He, and Shi Feng. 2024. Language Models Learn to Mislead Humans via RLHF. arXiv:2409.12822 [cs.CL] https://arxiv.org/abs/2409.12822
[32] Renjun Xu and Yang Yan. 2026. Agent Skills for Large Language Models: Architecture, Acquisition, Security, and the Path Forward. arXiv:2602.12430 [cs.MA] https://arxiv.org/abs/2602.12430
[33] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2025. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. In *The Thirteenth International Conference on Learning Representations*.
[34] 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
[35] Daochen Zha, Zaid Pervaiz Bhat, Kwei-Herng Lai, Fan Yang, Zhimeng Jiang, Shaochen Zhong, and Xia Hu. 2023. Data-centric Artificial Intelligence: A Survey. arXiv:2303.10158 [cs.LG] https://arxiv.org/abs/2303.10158
[36] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*.

A EXAMPLES OF TASKS

We reproduce here some task entries from the final dataset, slightly edited for length and content.

Read task — assess_customer_support_ticket_data (read_explore)

```
{
  "task_name": "assess_customer_support_ticket_data",
  "category": "read_explore",
  "difficulty": "medium",
  "goal": "Evaluate the distribution of ticket
    categories in the customer support tickets dataset
    and identify any anomalies in the data.",
  "prompt": "Can you help me analyze the customer
    support tickets to understand the distribution of
    ticket categories and check for any data anomalies?",
  "initial_state": {
    "branches": [
      "main"
    ],
    "seeded_tables": [
      {
        "table": "support.tickets",
        "branch": "main"
      }
    ]
  },
  "expected_commands": [
    "bauplan query"
  ],
  "expected_skills": [
    "bauplan-explore-data"
  ],
  "end_conditions": {
    "hard_response_constraints": [
      {
        "kind": "contains",
        "value": "distribution of ticket categories",
        "description": "Response should include analysis
      of ticket categories."
      }
```

```
      ],
      "soft_response_constraints": [
        {
          "criterion":
      "category_distribution_is_quantitative",
          "description": "The reply should describe the
      category distribution in concrete terms - counts or
      proportions per category, or at minimum a ranking of
      categories from most to least common - not a vague
      'there are several categories' summary. A judge
      should see evidence the model actually grouped the
      data."
        }
      ],
      "state_assertions": [
        {
          "kind": "no_change",
          "target": "support.tickets",
          "expected": null,
          "description": "The ticket data should remain
      unchanged after exploration."
        }
      ]
    },
    "forbidden_actions": [
      {
        "pattern": "bauplan\\s+run\\b",
        "reason": "No new pipeline should be created
      during this task."
      }
    ]
}
```

Because this is a *read*-only task, note that the constraints on the response are qualitative checks that leverage an LLM-as-a-judge at runtime.

On the other hand, a *write* task such as the following includes detailed programmatic verification through a Python script (parts of which we report for convenience in a separate listing at the end): the script exits with status 0 on success or with a nonzero status (and an English explanation) on failure.

Write task — build_subway_od_pair_summary_pipeline (write_new_pipeline)

```
{
  "task_name": "build_subway_od_pair_summary_pipeline",
  "difficulty": "medium",
  "prompt_specificity": "detailed",
  "goal": "Build and publish a station origin
    destination aggregate pipeline from the existing MTA
    subway trips table onto the agent's personal main
  branch using a feature-branch workflow, without
    modifying global main.",
  "prompt": "Create a Bauplan project that reads the
    existing `transit.mta_subway_trips` table and
    materializes a new table named
  `analytics.subway_od_pair_summary` on your personal main
    lineage. Use the required branch workflow: discover
    your username programmatically, create a
  feature branch off your `<username>.main` branch, build
    and validate the output on that feature branch, then
    merge the successful feature branch back
  into your `<username>.main` branch. Do not write to the
    global `main` branch.\n\nThe output table
    `analytics.subway_od_pair_summary` should have one
    row
  per origin/destination station-complex pair after
    excluding rows where `origin_station_complex_id =
    destination_station_complex_id` and excluding rows
  with null or non-positive `estimated_average_ridership`.
    Include these columns:\n-
    `origin_station_complex_id`\n-
    `origin_station_complex_name`\n-
  `destination_station_complex_id`\n-
    `destination_station_complex_name`\n-
    `trip_record_count` = count of contributing source
    records\n-
  `total_estimated_ridership` = sum of
    `estimated_average_ridership`\n-
    `avg_estimated_ridership` = average of
    `estimated_average_ridership`\n\nBefore
  merging, validate that the table is readable, non-empty,
    has no same-station origin/destination pairs, and
    has positive total ridership. After merging,
  reply briefly with the destination table name and
    confirm it is on `<username>.main`.",
  "expected_commands": [
    "bauplan branch create",
    "bauplan run",
    "bauplan branch merge"
  ],
  "expected_skills": [
    "bauplan-data-pipeline",
    "bauplan-data-quality-checks"
  ],
  "end_conditions": {
    "hard_response_constraints": [
      {
        "kind": "contains",
        "value": "analytics.subway_od_pair_summary",
        "description": "Final reply names the
      materialized destination table."
      }
    ],
    "state_assertions": [
      {
        "kind": "no_change",
        "target": "main",
        "expected": null,
        "description": "The global main branch is
      unchanged by the write workflow."
      }
    ]
  },
  "validation_script": "<see Listing below>",
  "category": "write_new_pipeline"
}
```

Listing 3: Sample validation script using the SDK.

```
import sys

import bauplan


DEST_NAMESPACE = "analytics"
DEST_TABLE = "subway_od_pair_summary"
DEST_FULL_NAME = f"{DEST_NAMESPACE}.{DEST_TABLE}"


def fail(message: str) -> None:
    print(message, file=sys.stderr)
```

```
    raise SystemExit(1)


# Discover the agent's personal main branch.
try:
    client = bauplan.Client()
    info = client.info()
except Exception as exc:
    fail(f"Could not initialize the Bauplan client: {exc}")

if info.user is None or not info.user.username:
    fail("Could not discover the current Bauplan username")

username = info.user.username
user_main = f"{username}.main"


# The destination table must exist on the agent's personal main branch.
try:
    exists_on_user_main = client.has_table(
        DEST_TABLE,
        namespace=DEST_NAMESPACE,
        ref=user_main,
    )
except Exception as exc:
    fail(f"Could not inspect {DEST_FULL_NAME} on {user_main}: {exc}")

if not exists_on_user_main:
    fail(f"{DEST_FULL_NAME} does not exist on {user_main}")


# The destination table must not have been created on global main.
# The benchmark harness separately verifies that global main's commit
# did not change during the task.
try:
    exists_on_global_main = client.has_table(
        DEST_TABLE,
        namespace=DEST_NAMESPACE,
        ref="main",
    )
except Exception as exc:
    fail(f"Could not inspect {DEST_FULL_NAME} on global main: {exc}")

if exists_on_global_main:
    fail(f"{DEST_FULL_NAME} was created on global main")


# Check that every required output column is present.
required_columns = {
    "origin_station_complex_id",
    "origin_station_complex_name",
    "destination_station_complex_id",
    "destination_station_complex_name",
    "trip_record_count",
    "total_estimated_ridership",
    "avg_estimated_ridership",
}

try:
    schema_table = client.query(
        f"SELECT * FROM {DEST_FULL_NAME} LIMIT 0",
        ref=user_main,
    )
except Exception as exc:
    fail(f"Could not inspect the schema of {DEST_FULL_NAME}: {exc}")

actual_columns = {name.lower() for name in schema_table.column_names}
missing_columns = sorted(required_columns - actual_columns)

if missing_columns:
    fail(
        "Destination table is missing required columns: "
        + ", ".join(missing_columns)
    )

# Recompute the expected result from the source table and compare the
# complete expected and actual relations. Numeric aggregates are rounded
# only for comparison, to tolerate insignificant floating-point differences.
validation_query = f"""
WITH expected AS (
    SELECT
        origin_station_complex_id,
        origin_station_complex_name,
        destination_station_complex_id,
        destination_station_complex_name,
        COUNT(*) AS trip_record_count,
        SUM(estimated_average_ridership)
            AS total_estimated_ridership,
        AVG(estimated_average_ridership)
            AS avg_estimated_ridership
    FROM transit.mta_subway_trips
    WHERE estimated_average_ridership IS NOT NULL
      AND estimated_average_ridership > 0
      AND origin_station_complex_id
          <> destination_station_complex_id
    GROUP BY
        origin_station_complex_id,
        origin_station_complex_name,
        destination_station_complex_id,
        destination_station_complex_name
),
actual AS (
    SELECT
        origin_station_complex_id,
        origin_station_complex_name,
        destination_station_complex_id,
        destination_station_complex_name,
        trip_record_count,
        total_estimated_ridership,
        avg_estimated_ridership
    FROM {DEST_FULL_NAME}
),
expected_normalized AS (
    SELECT
        origin_station_complex_id,
        origin_station_complex_name,
        destination_station_complex_id,
        destination_station_complex_name,
        trip_record_count,
        ROUND(total_estimated_ridership, 6)
            AS total_estimated_ridership,
        ROUND(avg_estimated_ridership, 6)
            AS avg_estimated_ridership
    FROM expected
),
actual_normalized AS (
    SELECT
        origin_station_complex_id,
        origin_station_complex_name,
        destination_station_complex_id,
        destination_station_complex_name,
        trip_record_count,
        ROUND(total_estimated_ridership, 6)
            AS total_estimated_ridership,
        ROUND(avg_estimated_ridership, 6)
            AS avg_estimated_ridership
    FROM actual
),
missing_rows AS (
    SELECT *
    FROM expected_normalized

    EXCEPT ALL

    SELECT *
    FROM actual_normalized
),
unexpected_rows AS (
    SELECT *
    FROM actual_normalized

    EXCEPT ALL

    SELECT *
    FROM expected_normalized
)
```

```
SELECT
    (SELECT COUNT(*) FROM expected) AS expected_rows,
    (SELECT COUNT(*) FROM actual) AS actual_rows,
    (SELECT COUNT(*) FROM missing_rows) AS missing_rows,
    (SELECT COUNT(*) FROM unexpected_rows) AS unexpected_rows,

    (
        SELECT COUNT(*)
        FROM actual
        WHERE origin_station_complex_id
            = destination_station_complex_id
    ) AS same_station_rows,

    (
        SELECT COUNT(*)
        FROM actual
        WHERE trip_record_count IS NULL
           OR trip_record_count <= 0
    ) AS invalid_trip_count_rows,

    (
        SELECT COUNT(*)
        FROM actual
        WHERE total_estimated_ridership IS NULL
           OR total_estimated_ridership <= 0
    ) AS invalid_total_rows,

    (
        SELECT COUNT(*)
        FROM actual
        WHERE avg_estimated_ridership IS NULL
           OR avg_estimated_ridership <= 0
    ) AS invalid_average_rows
"""

try:
    result = client.query(
        validation_query,
        ref=user_main,
        max_rows=1,
    )
except Exception as exc:
    fail(f"Validation query failed: {exc}")

if result.num_rows != 1:
    fail("Validation query did not return exactly one result row")

checks = result.to_pylist()[0]


if checks["actual_rows"] <= 0:
    fail(f"{DEST_FULL_NAME} is empty")

if checks["same_station_rows"] != 0:
    fail(
        "Destination table contains "
        f"{checks['same_station_rows']} same-station "
        "origin/destination rows"
    )

if checks["invalid_trip_count_rows"] != 0:
    fail(
        "Destination table contains "
        f"{checks['invalid_trip_count_rows']} rows with a null or "
        "non-positive trip_record_count"
    )

if checks["invalid_total_rows"] != 0:
    fail(
        "Destination table contains "
        f"{checks['invalid_total_rows']} rows with null or "
        "non-positive total_estimated_ridership"
    )

if checks["invalid_average_rows"] != 0:
    fail(
        "Destination table contains "
        f"{checks['invalid_average_rows']} rows with null or "
        "non-positive avg_estimated_ridership"
    )

if checks["actual_rows"] != checks["expected_rows"]:
    fail(
        "Row-count mismatch: "
        f"expected {checks['expected_rows']}, "
        f"found {checks['actual_rows']}"
    )

if checks["missing_rows"] != 0:
    fail(
        "Destination table is missing "
        f"{checks['missing_rows']} expected rows"
    )

if checks["unexpected_rows"] != 0:
    fail(
        "Destination table contains "
        f"{checks['unexpected_rows']} incorrect or unexpected rows"
    )

print(
    f"PASS: {DEST_FULL_NAME} on {user_main} "
    "matches the expected origin/destination aggregation"
)
```