DataFlow-Harness: A Grounded Code-Agent Platform for Constructing Editable LLM Data Pipelines

Runming He[*], **Zhen Hao Wong**[*], **Hao Liang**[*,‡], **Zimo Meng**[*], **Chengyu Shen**, **Xiaochen Ma**, **Wentao Zhang**[‡]

[1]Peking University, [2]Institute for Advanced Algorithms Research, Shanghai, [3]Zhongguancun Academy

Large language models (LLMs) are increasingly used to automate data-processing workflows, yet coding agents typically produce scripts that are not automatically materialized as persistent, editable platform artifacts. We call this disconnect the *NL2Pipeline gap*. To bridge it, we introduce DataFlow-Harness, a platform that guides an LLM agent to construct platform-native directed acyclic graphs (DAGs) through typed, incremental mutations rather than free-form scripts. The platform combines DataFlow-Skills for procedural guidance, a Model Context Protocol (MCP) layer that exposes the live operator registry and current pipeline state, and DataFlow-WebUI, which synchronizes conversational authoring with a visual DAG editor. On a 12-task data-engineering benchmark, DataFlow-Harness achieves a 93.3% observed end-to-end pass rate. Relative to Vanilla Claude Code, it reduces measured monetary cost by 72.5% and generation latency by 49.9%; its observed pass rate is within 0.9 percentage points of the Context-Aware Claude Code baseline while its cost is 42.8% lower. Per-task analysis indicates that Skills are most useful when construction depends on implicit procedural knowledge. These results show that live platform grounding can produce persistent, editable workflow artifacts with an observed reliability close to script-generation baselines and with lower measured construction cost and latency.

[*]Equal Contribution, [†]Project Leader, [‡]Corresponding author

Correspondence : wentao.zhang@pku.edu.cn

Source Code : https://github.com/OpenDCAI/DataFlow-WebUI

Codebase Documentation : https://opendcai.github.io/DataFlow-Doc/

Contents

1 Introduction

Large language models (LLMs) are increasingly deployed to construct data-processing workflows for applications such as synthetic data generation, evaluation, retrieval augmentation, and model training [9–11]. Recent coding agents can automatically translate natural-language requirements into executable implementations, significantly reducing the effort required to construct such workflows [7, 25].

However, high task accuracy alone is often insufficient for production deployment. In industrial environments, workflow artifacts must remain visible, editable, reusable, and compatible with platform governance mechanisms throughout their lifecycle [12]. In our experience, direct code-generation agents frequently produce disposable scripts that exist only as source code. These outputs are difficult to audit through graphical workflow interfaces and often hallucinate dependencies [19, 22], relying on unavailable operators, outdated platform assumptions, or framework-specific behaviors that general-purpose agents struggle to infer [18, 20, 21].

We define this challenge as the *NL2Pipeline gap*: while users express workflow requirements in natural language, production environments require structured and persistent pipeline assets that can be visualized, edited, and reused. Here, a workflow denotes the intended data-processing procedure, the pipeline representation is its persistent platform object, and the DAG captures its execution dependencies. Closing this gap requires more than improving code-generation accuracy: construction must remain grounded in platform semantics and produce artifacts that integrate with the host platform.

To address this limitation, we propose DataFlow-Harness, a platform for grounded workflow synthesis. Rather than directly generating scripts, DataFlow-Harness guides an agent to construct platform-native workflows through three decoupled components. DataFlow-Skills encode domain-specific construction knowledge, including operator-selection patterns, schema dependencies, and assembly procedures. The Model Context Protocol (MCP) provides access to the live operator registry and current workflow state, grounding agent actions in the execution environment [2]. Finally, DataFlow-WebUI provides a conversational interface for iterative refinement and materializes generated workflows as persistent, editable visual DAGs.

We evaluate DataFlow-Harness on pipeline-construction tasks covering data transformation, question answering, quality filtering, and synthetic data generation. Its observed end-to-end pass rate is close to the script-generation baselines on this benchmark, while measured token usage, cost, and latency are lower.

Our contributions are as follows:

- We formulate the *NL2Pipeline gap*: the disconnect between natural-language workflow intent and persistent, platform-native workflow artifacts that remain available for inspection and editing.
- We present DataFlow-Harness, which combines procedural Skills, live MCP grounding, typed incremental mutations, validation, and a synchronized conversational and visual authoring interface.
- We evaluate reliability and construction efficiency on a 12-task benchmark, analyze where Skills help through a per-task ablation, and provide two controlled case studies of downstream training utility.

2 Related Work

2.1 Agents for Code Generation.

Code synthesis has evolved from foundational pretrained models such as Codex [5] and StarCoder [15] to autonomous agentic loops. Reflexion [23] and Self-Debug [6] introduced iterative refinement via environment feedback. MCP [2] provides unified tool interfaces for agents. Current agents such as SWE-agent [26] and Claude Code [1] focus on repository-level multi-turn editing and verification. Our work differs in that we constrain a general-purpose agent within a domain-specific harness rather than improving the agent itself.

2.2 Data Engineering and LLM Pipelines.

Data-centric AI [28] emphasizes systematic governance over model architecture. Specialized systems such as Data-Juicer [4] and DCLM [14] operationalize large-scale curation through extensible operators. DataFlow [16] and DSPy [11] treat LLM operations as composable components within formal execution graphs. Our

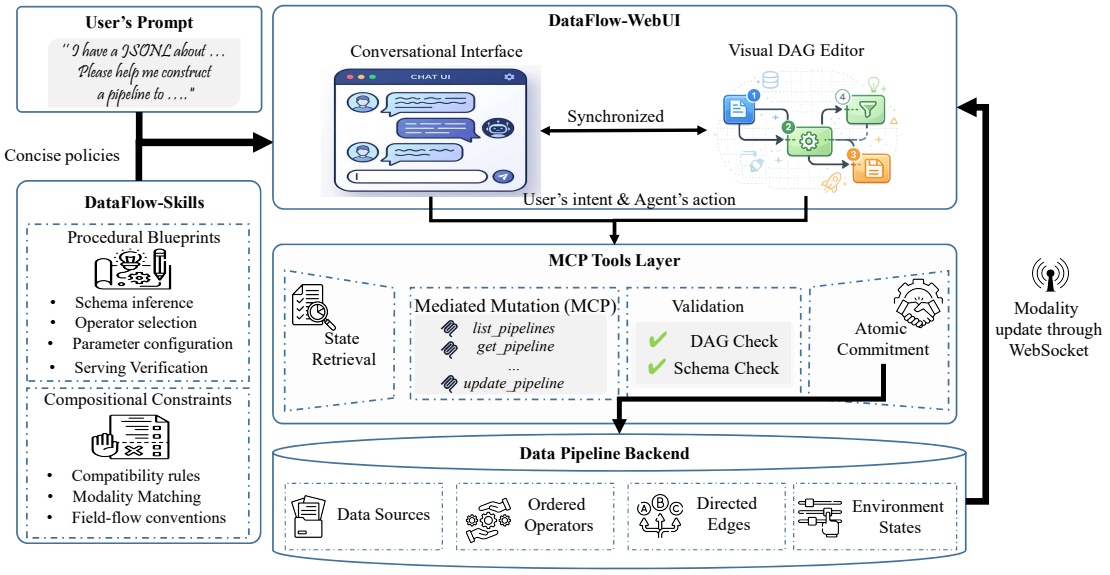

Figure 1 The DataFlow-Harness architecture. A shared pipeline representation is synchronized across the agent runtime and DataFlow-WebUI. DataFlow-Skills guides construction, while the Validation Engine checks DAG structure and schema compatibility.

work builds on DataFlow but focuses on agent-assisted construction of the pipeline itself, rather than pipeline execution.

2.3 LLM-based Workflow Synthesis.

Recent systems generate structured workflows rather than standalone programs. AutoFlow automatically synthesizes reusable workflows for LLM agents [13], while Balis et al. translate scientific research questions into validated workflow DAGs and encode domain knowledge as reusable Skills [3]. These efforts establish the value of structured workflow generation. DataFlow-Harness focuses on a complementary systems problem: interactive, stateful authoring inside a live data-engineering platform. The agent retrieves the current pipeline and operator registry through MCP, applies typed incremental mutations to an existing artifact, validates each proposed state, and shares one persistent representation with the conversational interface and visual editor. Thus, our contribution is not NL-to-DAG generation alone, but platform-grounded construction and iterative editing of a live workflow artifact.

3 System Architecture

DataFlow-Harness organizes workflow synthesis around four components: DataFlow-WebUI, the MCP Tools Layer, the Data Pipeline Backend, and DataFlow-Skills.

The operational lifecycle centers on the Data Pipeline Backend, which serves as the authoritative source of truth across conversational, visual, and programmatic interfaces. Pipeline mutations are issued through the MCP Tools Layer, validated and committed to the backend, and then synchronized with DataFlow-WebUI. In parallel, DataFlow-Skills provides procedural guidance that shapes agent reasoning without directly modifying pipeline state.

3.1 Data Pipeline Backend

The Data Pipeline Backend serves as the authoritative source of truth for workflow synthesis. We represent a pipeline as $P = (D, O, E, S, R)$, where D is the set of data sources and their URIs, O is the set of configured

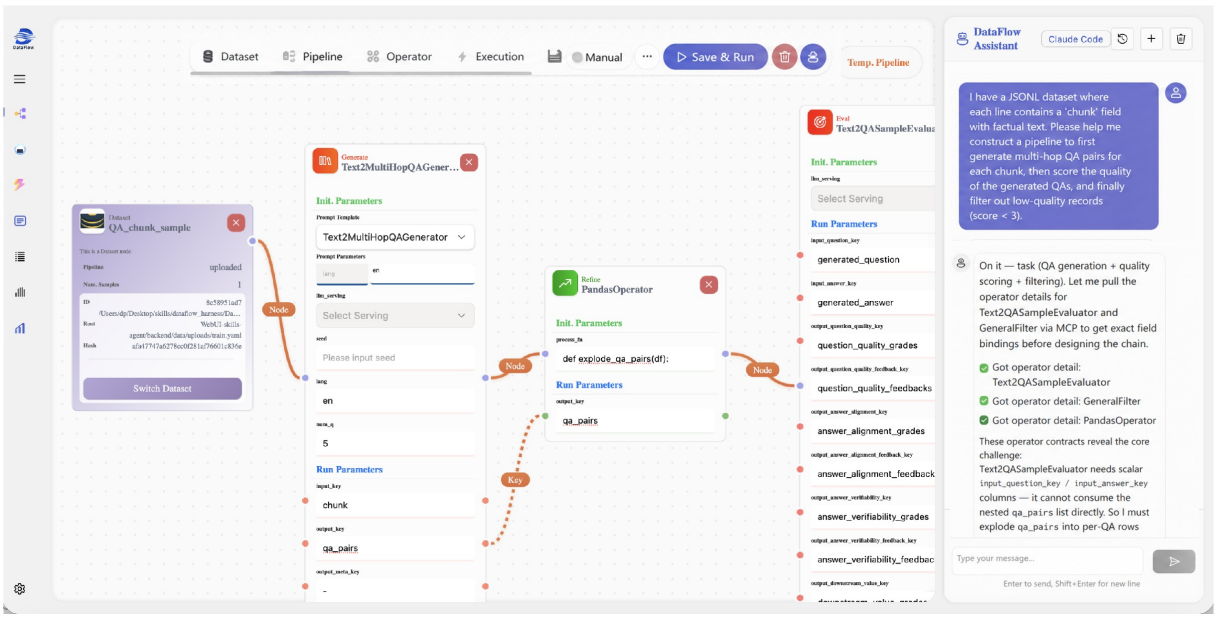

Figure 2 The dual-modality interface of DataFlow-WebUI, illustrating the synchronization between the conversational agent and the visual DAG editor.

operator instances, $E \subseteq O \times O$ contains directed data-dependency edges, S records input and output field schemas, and R contains runtime state such as model-serving endpoints.

Rather than generating free-form code, agents interact with the backend through typed mutations, including adding or removing operators, updating parameters, and connecting edges. A mutation is committed only if the resulting graph is acyclic and adjacent operator schemas are compatible. These checks establish structural validity; they do not by themselves guarantee semantic correctness, endpoint availability, or output quality.

3.2 DataFlow-WebUI

DataFlow-WebUI provides two synchronized modalities for workflow construction: a conversational interface for natural-language authoring and a visual DAG editor for direct workflow inspection and editing (Figure 2).

Conversational Interface. Users describe workflow requirements in natural language. Before each agent turn, the current pipeline state and the DataFlow operator registry are injected into Claude Code's context via MCP. Guided by DataFlow-Skills, Claude Code interprets user intent and determines the required workflow modifications, which are expressed as MCP tool calls and applied to the Data Pipeline Backend.

Visual DAG Editor. A graphical editor renders the workflow as a directed acyclic graph. Users can inspect agent-proposed changes, adjust parameters, relink edges, or add and remove operators directly. Any manual edit is immediately committed to the Data Pipeline Backend, ensuring that subsequent agent interactions operate on the latest workflow state without requiring explicit re-synchronization.

3.3 MCP Tools Layer

Every pipeline change, whether proposed by the agent or made manually, passes through a Request-Validate-Commit protocol.

State Retrieval. At the start of each synthesis turn, the agent fetches the latest pipeline state, incorporating any manual edits since the previous turn.

Mediated Mutation. Guided by DataFlow-Skills, Claude Code issues an MCP tool call expressing the intended change as a typed, structured mutation grounded in the DataFlow registry's live metadata.

Validation. The system verifies two properties: that the updated pipeline remains a directed acyclic graph, and that the output field schema of each operator is compatible with the input schema of every downstream operator. Changes that fail either check are rejected.

Validated Commitment. Validated changes are written to the backend store. A WebSocket notification broadcasts the updated state to connected clients, keeping the authoring modalities synchronized.

3.4 DataFlow-Skills

DataFlow-Skills provides procedural guidance for workflow synthesis by injecting domain-specific knowledge into Claude Code's reasoning context. While the MCP Tools Layer exposes operator metadata and workflow state, it does not encode recommended construction strategies or operator-composition best practices. As a result, agents may select inappropriate operators, omit prerequisite processing steps, or construct workflows that are structurally valid but semantically incorrect.

To address this, DataFlow-Skills encodes two classes of knowledge. The first consists of procedural blueprints that define recommended workflow-construction sequences, including schema inference, operator selection, parameter configuration, and serving verification. The second consists of compositional constraints that capture operator compatibility rules, such as modality matching and field-flow conventions for nested structures. Together, DataFlow-Skills guides agent reasoning, while the MCP Tools Layer grounds execution against the live DataFlow environment.

4 Experiments

We evaluate DataFlow-Harness to answer the following research questions, structured to progressively demonstrate the system's effectiveness, efficiency, and underlying mechanisms:

RQ1 (Workflow Synthesis Effectiveness). How does DataFlow-Harness's native DAG synthesis compare to traditional free-form code generation (disposable scripts) in maintaining execution reliability and end-to-end task success for industrial data-processing tasks?

RQ2 (System Efficiency). What are the specific computational and economic advantages (e.g., token consumption, latency, and API cost) introduced by shifting from context-heavy script generation to our structured DAG synthesis?

RQ3 (Ablation & Micro-mechanisms). How does DataFlow-Skills improve upon pure tool-specification grounding (MCP-only) across varying levels of task complexity?

RQ4 (Downstream Data Quality). Beyond governability and execution reliability, does grounding the agent with DataFlow-Harness lead it to author higher-quality synthesis pipelines, as measured by the downstream accuracy of models trained on the data those pipelines produce?

4.1 Experimental Setup

Benchmark. We evaluate pipeline-construction capability on a benchmark of 12 tasks spanning six representative industrial data-processing scenarios: QA generation, review governance, long-document processing, multi-field scoring, schema normalization, and low-quality filtering. Each task specifies a natural-language objective together with input data samples and task-specific acceptance criteria.

Experimental Settings. To characterize the contributions of different system components, we compare four agent configurations with different levels of platform grounding: (1) **Vanilla CC**: An unconstrained coding baseline utilizing standard Claude Code. It relies entirely on internal parametric knowledge to generate disposable Python scripts, lacking access to platform-specific context. (2) **Context-Aware CC**: A repository-grounded baseline where the agent is provided with the raw DataFlow codebase. While in-context code

Method	Artifact Type	Task Success (%)	Efficiency & Cost		
		End-to-End Pass↑	Tokens (In/Out) ↓	Cost ($) ↓	Latency (s) ↓
Vanilla CC	Disposable Script	91.7	153,584 / 2,474	0.950	190.7
Context-Aware CC	Disposable Script	**94.2**	185,626 / <u>1,140</u>	0.456	115.9
MCP-only	Native DAG	83.3	<u>100,607</u> / 1,273	<u>0.321</u>	<u>105.5</u>
DataFlow-Harness	Native DAG	<u>93.3</u>	**74,958** / **891**	**0.261**	**95.5**

Table 1 We report the end-to-end pass rate together with token usage, monetary cost, and generation latency. The pass rate is computed over 120 task runs (12 tasks × 10 trials); efficiency metrics are averaged over the same runs. **The best results are highlighted in bold, and the second-best results are underlined.**

comprehension enables it to accurately mimic platform operators, it inherently produces unmanageable, one-off scripts. (3) **MCP-Only**: A tool-augmented baseline strictly constrained to synthesize platform-native DAGs. It utilizes DataFlow MCP tools to dynamically discover operators but lacks explicit procedural guidance for complex assembly. (4) **DataFlow-Harness**: Our complete framework, combining MCP-based platform grounding with DataFlow-Skills to efficiently synthesize editable, governable workflow DAGs.

All experiments use Claude Opus 4.7 as the underlying large language model to keep the reasoning model fixed. To account for stochasticity in agent behavior and LLM generation, each task is executed 10 times under every setting, resulting in 120 task runs per method.

The configurations intentionally expose different action spaces: the script baselines may generate arbitrary Python, whereas MCP-only and DataFlow-Harness select and compose operators available in DataFlow. The comparison therefore measures the end-to-end system trade-off between free-form script generation and platform-constrained workflow construction, rather than an isolated difference in model capability.

Evaluation Metrics. We evaluate the proposed framework along three complementary dimensions: task success, efficiency, and platform integration.

Task Success. We measure **End-to-End (E2E) Pass**, which requires the generated workflow to execute successfully and produce outputs satisfying task-specific acceptance criteria. This metric captures overall workflow quality, including workflow synthesis, operator configuration, execution correctness, and final output validity.

Efficiency Metrics. To evaluate practical deployment cost, we additionally measure token consumption, monetary cost, and workflow construction latency. (1) **Token Consumption** reports the total number of input and output tokens used during workflow generation. (2) **Cost** is estimated using the official pricing of the underlying model and includes all interactions required to complete a workflow. (3) **Generation Latency** measures the wall-clock time from task submission to the production of a valid workflow artifact.

4.2 Workflow Synthesis Effectiveness (RQ1)

We first characterize the performance of free-form code generation. As shown in Table 1, enriching vanilla generation (Vanilla CC) with execution context (Context-Aware CC) improves end-to-end success from 91.7% to 94.2%, highlighting the importance of procedural context for complex data-processing tasks. However, both approaches produce monolithic scripts that remain detached from platform-native workflow abstractions.

Transitioning from script generation to structured DAG synthesis introduces a substantial challenge. Using only operator specifications (MCP-only) reduces end-to-end success to 83.3%, indicating that workflow constraints alone impose a significant reasoning burden on the model. This result reveals a clear *NL2Pipeline gap*: directly generating deployable workflows is substantially harder than generating executable scripts.

DataFlow-Harness closes much of this gap through explicit procedural guidance. It achieves 93.3% end-to-end success, improving by 10.0 percentage points over MCP-only while remaining within 0.9 percentage points

Method	Precision ↑	Coverage Rate ↑
Vanilla CC	0.621	0.533
Context-Aware CC	0.893	0.801
MCP-only	0.784	0.621
DataFlow-Harness	**0.972**	**0.873**

Table 2 Textbook-to-VQA extraction performance. Coverage measures the proportion of extractable QA pairs successfully recovered from the document.

of Context-Aware CC. The observed pass rates are numerically close, although we do not claim statistical equivalence. These results suggest that structured workflow synthesis can produce platform-native DAGs with reliability approaching that of the script-generation baselines on this benchmark.

4.3 Efficiency and System Cost (RQ2)

Table 1 shows that DataFlow-Harness delivers substantial efficiency gains over free-form code generation. Compared with Vanilla CC, it reduces monetary cost by 72.5% (from \$0.950 to \$0.261) and generation latency by 49.9% (from 190.7s to 95.5s), while also achieving a higher end-to-end success rate. These results indicate that structured workflow synthesis is considerably more resource-efficient than generating executable scripts.

Notably, the efficiency gains persist even against the stronger Context-Aware CC baseline. Despite achieving nearly identical end-to-end performance, DataFlow-Harness reduces cost by 42.8% and latency by 17.6%, demonstrating a substantially more favorable efficiency–performance tradeoff.

The improvement is primarily driven by lower token consumption. Moving from script generation to native DAG synthesis, MCP-only nearly halves input token usage relative to Context-Aware CC, while DataFlow-Harness further reduces total token consumption by 25.5% compared with MCP-only. This suggests workflow representations are far more compact than executable code, with procedural guidance further streamlining their construction within the constrained operator space.

4.4 Textbook-to-VQA Workflow Evaluation

We further evaluate our system on a challenging **textbook-to-VQA extraction** task, which requires constructing question-answer pairs from heterogeneous educational documents, including long-form textbooks, interleaved solution manuals, and exam answer sheets. This setting is particularly difficult due to (i) long-range dependencies between questions and answers, (ii) visually grounded reasoning over figures and tables, and (iii) highly non-linear document layouts that break local textual continuity.

To comprehensively evaluate extraction quality, we follow FlipVQA-Miner's [24] settings, and report two complementary metrics: **Precision** measures the correctness of extracted QA pairs, while **Coverage Rate** measures the proportion of extractable QA pairs successfully recovered from the source document.

Table 2 evaluates a challenging textbook-to-VQA extraction task that requires composing document parsing, layout analysis, multimodal content extraction, question-answer alignment, and dataset construction into a single workflow. Compared with conventional workflow synthesis benchmarks, success in this setting depends not only on reasoning ability but also on the effective utilization of specialized document-processing components.

The observed results favor DataFlow-Harness on both extraction correctness and completeness, with 97.2% precision and 87.3% coverage. The largest absolute improvement is observed in coverage, where DataFlow-Harness recovers more valid QA pairs than the baselines. This pattern suggests that the framework constructs more complete workflows rather than merely filtering outputs conservatively. Confirming the generality of this result requires repeated runs and a fully specified annotation protocol.

The improvement may partly reflect the rich operator ecosystem provided by DataFlow. Textbook-to-VQA extraction requires capabilities such as PDF parsing, layout recovery, OCR, figure extraction, multimodal understanding, and long-range question-answer matching. While these capabilities already exist as reusable

Task	MCP-only	DataFlow-Harness
Procedural-knowledge-dependent tasks		
1a: QA basic	6/10	**10/10**
1b: QA with filter	6/10	**9/10**
3b: Text-to-QA chain	6/10	**10/10**
Trivially routable tasks		
5a: Field rename	10/10	10/10
5b: Nested flatten	10/10	10/10
6a: Length filter	10/10	10/10
6b: LLM semantic filter	10/10	10/10
Tasks with non-synthesis bottlenecks		
4a: Score and filter	7/10	7/10
4b: Multi-dimensional score	7/10	7/10
2a: Sentiment	9/10	**10/10**
2b: Review governance	**10/10**	9/10
3a: Long-document summary	9/10	**10/10**

Table 3 Per-task end-to-end pass counts over 10 independent trials. Tasks are grouped according to the mechanisms discussed in RQ3.

platform operators, effectively discovering and composing them remains challenging for general-purpose coding agents. The performance gap between MCP-only and DataFlow-Harness indicates that operator exposure alone is insufficient; procedural knowledge is required to guide workflow construction and ensure that relevant operators are assembled into valid end-to-end pipelines.

More broadly, this case study highlights an important property of DataFlow-Harness: its advantage does not arise from stronger model reasoning, but from enabling systematic reuse of mature platform assets. As workflow complexity increases, successful execution increasingly depends on leveraging existing operator ecosystems rather than synthesizing functionality from scratch. The results therefore provide concrete evidence that procedural guidance and platform-native abstractions are critical for closing the *NL2Pipeline gap* in realistic data-processing scenarios.

4.5 Ablation and Micro-mechanisms (RQ3)

Having shown that DataFlow-Harness largely closes the performance gap between native DAG synthesis and free-form code generation, we next investigate when explicit procedural guidance is most beneficial. The per-task results in Table 3 reveal three consistent patterns.

Procedural guidance is most valuable when task success depends on implicit domain knowledge. The largest improvements occur on QA-generation and language-processing tasks (`1a`, `1b`, and `3b`), where successful construction requires more than selecting compatible operators. Although MCP-only frequently generates structurally valid DAGs, it struggles to infer task-specific procedures from operator descriptions alone. Encoding these procedures as reusable skills improves aggregate success in this group from 18/30 to 29/30 runs, accounting for most of the overall gain.

Procedural guidance provides limited benefit when workflow routing is straightforward. For simple transformation and filtering tasks (`5a`, `5b`, `6a`, and `6b`), both methods achieve perfect success. In these cases, operator specifications are sufficient to identify the workflow structure, leaving little room for improvement from higher-level guidance.

Procedural guidance cannot overcome limitations outside workflow synthesis. The remaining failures appear to arise primarily from factors unrelated to workflow construction. Both methods exhibit the same failure rate on multi-field scoring tasks (`4a` and `4b`), where outputs occasionally violate downstream numerical

Table 4 Math pipeline quality via downstream training. Both pipelines are authored by Claude Code from the same natural-language prompt (Prompt 1), synthesize data with the same models and API settings, and are used to fine-tune Qwen2.5-32B-Instruct under the same LIMO recipe. We report accuracy (%) on the DataFlow [16] Table 5 math suite; AIME24/25 are reported as avg@32. At matched epochs, data produced with DataFlow-Harness yields a higher average, indicating a higher-quality pipeline. Per-epoch best average is in **bold**.

Pipeline (author)	GSM8K	MATH	AMC23	Olympiad	Gaokao24_mix	Minerva	AIME24@32	AIME25@32	Avg
Qwen2.5-32B-Instruct (base)	95.8	73.5	70.0	38.5	42.9	26.5	16.8	11.6	46.95
Trained with 1 epoch									
Vanilla CC	92.3	78.0	47.5	42.8	56.0	35.7	25.1	21.6	49.9
DataFlow-Harness	93.9	72.3	72.5	38.7	38.5	26.5	35.9	34.5	**51.6**
Trained with 2 epochs									
Vanilla CC	94.8	84.0	60.0	48.0	53.8	39.7	31.8	24.3	54.5
DataFlow-Harness	94.4	76.6	75.0	45.2	42.9	25.7	45.4	40.0	**55.7**

constraints despite correct DAG generation. On tasks `2a`, `2b`, and `3a`, differences are small and occasionally favor MCP-only, suggesting that prescriptive procedures may reduce flexibility when multiple execution strategies are valid.

Overall, these results indicate that DataFlow-Skills contribute primarily by injecting procedural knowledge that is difficult to recover from operator specifications alone. Their observed benefit is largest on ambiguous, procedurally complex tasks and diminishes when workflow construction is trivial or bottlenecked by the underlying model. Because this ablation compares MCP-only with the full system, it does not separately identify the contribution of validation.

4.6 Downstream Training Utility Evaluation (RQ4)

The preceding experiments assess whether DataFlow-Harness produces workflows that are governable and that execute reliably. A distinct and arguably more consequential question is whether the harness helps the agent author better pipelines, i.e., pipelines whose output data is more useful downstream. To answer this, we adopt an end-to-end, outcome-based protocol: we let the coding agent construct a full data-synthesis pipeline, run the pipeline to synthesize a training set, fine-tune a model on that set, and measure the resulting model's benchmark accuracy. Because a pipeline is ultimately a means to produce training data, the quality of the data it emits is a direct, if indirect, measure of the pipeline's quality.

Protocol. For each scenario we compare two configurations that differ in exactly one respect. In **Vanilla CC**, Claude Code receives only the natural-language task description and writes a synthesis pipeline from its parametric knowledge. In **DataFlow-Harness**, the identical agent receives the identical prompt but is additionally grounded in the DataFlow environment through DataFlow-Skills and the MCP operator registry. Everything downstream of pipeline construction is held fixed across the two arms: the same underlying LLMs and OpenAI-style API settings (concurrency and timeout) are used to synthesize data, the same number of samples is generated, and the same base model, fine-tuning recipe, and evaluation harness are used. This controlled setup isolates the agent-authored pipeline as the primary source of differences in downstream accuracy, while holding model, data scale, and training settings fixed.

Math Reasoning Pipeline. The first scenario (Prompt 1) asks the agent to build a math data cleaning-and-synthesis pipeline: verify and filter incoming problems, discard ill-posed items, expand each seed question into two new synthetic problems, generate reasoning traces for every question, and apply n-gram deduplication to the resulting QA pairs, with per-stage models specified separately. We run each agent-authored pipeline over the same seed pool, fine-tune Qwen2.5-32B-Instruct following the LIMO recipe [27] (full-parameter SFT, lr 5e−6, cosine schedule without warmup, batch size 64, 16K context), and evaluate on the math suite and protocol of DataFlow [16] (Table 5), reporting AIME24/25 as avg@32.

Table 4 reports the results at matched training budgets. Both configurations lift the base model substantially, confirming that Claude Code can author a functional synthesis pipeline unaided. However, at every matched

epoch the data produced under DataFlow-Harness yields a higher average accuracy, improving from 49.9 to 51.6 after one epoch and from 54.5 to 55.7 after two. The gains concentrate on the hardest, most contamination-sensitive benchmarks: DataFlow-Harness data raises AIME24@32 from 25.1 to 35.9 and AIME25@32 from 21.6 to 34.5 at one epoch, with a similar margin at two epochs. This pattern is consistent with the grounded pipeline applying more effective verification, filtering, and deduplication, thereby producing cleaner and more challenging reasoning traces rather than merely more of them.

General SFT Pipeline. The second scenario (Prompt 2) is more demanding: the agent must build a from-scratch general instruction-tuning pipeline with no seed dataset, generating data end-to-end through API calls. The pipeline spans three stages: topic-conditioned generation of diverse instruction–response pairs from a preset knowledge taxonomy with multiple difficulty levels; a critique-then-rewrite refinement pass; and an LLM-as-judge scoring stage that filters low-quality samples. Each agent-authored pipeline synthesizes 10K instruction–response pairs, which we use to fine-tune Qwen2.5-7B-Base under an identical recipe (full-parameter SFT with LLaMA-Factory and DeepSpeed ZeRO-3 on 8×H20, lr 1e−5, cosine schedule, warmup 0.03, 3 epochs, global batch 128, cutoff 4096, bf16, seed 42). We evaluate the fine-tuned models with lm-evaluation-harness [8] across knowledge (MMLU), math (GSM8K, MATH, Minerva, Olympiad), and code (HumanEval, MBPP and their EvalPlus [17] variants) benchmarks.

Table 5 General SFT pipeline quality via downstream training. Both pipelines are authored by Claude Code from the same from-scratch synthesis prompt (Prompt 2) and generate 10K instruction–response pairs with the same models and API settings. We fine-tune Qwen2.5-7B-Base under an identical recipe (full-parameter SFT, LLaMA-Factory + DeepSpeed ZeRO-3, 8×H20, lr 1e−5, cosine, warmup 0.03, 3 epochs, global batch 128, cutoff 4096, bf16, seed 42) and evaluate with lm-evaluation-harness [8]. Code benchmarks use the EvalPlus [17] variants (HE+/MBPP+). **Avg** is the mean over all nine benchmarks; best per column in **bold**.

	Knowledge	Math				Code				
Pipeline (author)	MMLU	GSM8K	MATH	Minerva	Olympiad	HumanEval	HE+	MBPP	MBPP+	Avg
Vanilla CC	**74.4**	**82.9**	68.2	27.6	35.9	78.0	70.1	64.6	51.6	61.5
DataFlow-Harness	74.2	79.5	**70.1**	27.6	**36.3**	**80.5**	**72.6**	**75.4**	**58.2**	**63.8**

Table 5 shows nearly identical knowledge performance (MMLU 74.2 vs. 74.4), while the two pipelines trade wins across the individual math benchmarks. The clearest difference emerges on code, where the DataFlow-Harness pipeline is stronger across all four benchmarks, with the largest gap on MBPP (75.4 vs. 64.6). These code gains lift the overall nine-benchmark average by 2.3 points (63.8 vs. 61.5). This pattern is consistent with the grounded critique-then-rewrite and judge stages producing more executable, better-structured coding responses. Together, the two scenarios provide preliminary outcome-level evidence that grounding can improve the utility of data produced by an agent-authored pipeline. Because each scenario is a controlled case study rather than a repeated experiment across independently authored pipelines and multiple training seeds, these results should not be interpreted as a general causal estimate.

5 Conclusion

We presented DataFlow-Harness, a platform that addresses the *NL2Pipeline gap* by combining procedural Skills, live MCP grounding, typed mutations, structural validation, and synchronized conversational and visual editing. On our 12-task benchmark, its observed end-to-end pass rate is close to the script-generation baselines, while measured construction cost and latency are lower. The per-task ablation further shows where procedural guidance is most useful.

Limitations. Our evaluation uses one coding-agent and model family and a relatively small, platform-specific benchmark. The available ablation does not isolate every component, and schema validation cannot guarantee semantic correctness. We report observed averages without task-clustered confidence intervals or a pre-specified non-inferiority test. Cost reporting also requires a token-class breakdown to be independently recomputed when prompt caching is used. Finally, the downstream-utility results cover two case studies without multiple

independently authored pipelines and training seeds. Direct evaluation of persistence, reuse, provenance, concurrent editing, and recovery is needed before making broader workflow-governance claims.

References

[1] Anthropic. Claude Code: An agentic command-line coding assistant. https://docs.anthropic.com/claude/docs/claude-code, 2024. Accessed: 2026-05-07.

[2] Anthropic. Model Context Protocol: An open standard for connecting AI assistants to data and tools. https://modelcontextprotocol.io, 2024. Accessed: 2026-05-07.

[3] Balis et al. From research question to scientific workflow: Leveraging agentic ai for science automation. arXiv preprint arXiv:2604.21910, 2026. URL https://arxiv.org/abs/2604.21910.

[4] Daoyuan Chen, Yilun Huang, Zhijian Ma, Hesen Chen, Xuchen Pan, Ce Ge, Dawei Gao, Yuexiang Xie, Zhaoyang Liu, Jinyang Gao, et al. Data-juicer: A one-stop data processing system for large language models, 2023. URL https://arxiv.org/abs/2309.02033.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

[6] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In The Twelfth International Conference on Learning Representations (ICLR 2024), 2024. URL https://openreview.net/forum?id=KuPixIqPiq.

[7] Matthias Galster, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Sebastian Baltes. Configuring agentic ai coding tools: An exploratory study. arXiv preprint arXiv:2602.14690, 2026.

[8] Leo Gao, Jonathan Tow, Baber Abbasi, et al. A framework for few-shot language model evaluation, 2024. URL https://zenodo.org/records/12608602.

[9] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, Haofen Wang, et al. Retrieval-augmented generation for large language models: A survey. arXiv preprint arXiv:2312.10997, 2(1):32, 2023.

[10] Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, Danyang Li, Jiaqi Chen, Jiayi Zhang, Jinlin Wang, et al. Data interpreter: An llm agent for data science. In Findings of the Association for Computational Linguistics: ACL 2025, pages 19796–19821, 2025.

[11] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. In The Twelfth International Conference on Learning Representations (ICLR 2024), 2024. URL https://openreview.net/forum?id=sY5N0zY5Od.

[12] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine learning operations (mlops): Overview, definition, and architecture. IEEE access, 11:31866–31879, 2023.

[13] Li et al. Autoflow: Automated workflow generation for large language model agents. arXiv preprint arXiv:2407.12821, 2024. URL https://arxiv.org/abs/2407.12821.

[14] Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Yitzhak Gadre, Hritik Bansal, Etash Guha, Sedrick Scott Keh, Kushal Arora, Saurabh Garg, Rui Xin, Niklas Muennighoff, Reinhard Heckel, Jean Mercat, Mayee F. Chen, Suchin Gururangan, Mitchell Wortsman, Alon Albalak, Yonatan Bitton, Marianna Nezhurina, Amro Abbas, Cheng-Yu Hsieh, Dhruba Ghosh, Josh Gardner, Maciej Kilian, Hanlin Zhang, Rulin Shao, Sarah M. Pratt, Sunny Sanyal, Gabriel Ilharco, Giannis Daras, Kalyani Marathe, Aaron Gokaslan, Jieyu Zhang, Khyathi Raghavi Chandu, Thao Nguyen, Igor Vasiljevic, Sham Kakade, Shuran Song, Sujay Sanghavi, Fartash Faghri, Sewoong Oh, Luke Zettlemoyer, Kyle Lo, Alaaeldin El-Nouby, Hadi Pouransari, Alexander Toshev, Stephanie Wang, Dirk Groeneveld, Luca Soldaini, Pang Wei Koh, Jenia Jitsev, Thomas Kollar, Alexandros G. Dimakis, Yair Carmon, Achal Dave, Ludwig Schmidt, and Vaishaal Shankar. DataComp-LM: In search of the next generation of training sets for language models. In Advances in Neural Information Processing Systems 38 (NeurIPS 2024), 2024. URL https://arxiv.org/abs/2406.11794.

[15] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason T. Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder: may the source be with you! Transactions on Machine Learning Research, 2023. URL https://openreview.net/forum?id=KoFOg41haE.

[16] Zhiyuan Liang et al. DataFlow: An LLM-driven framework for unified data preparation and workflow automation in the era of data-centric AI, 2025. URL https://arxiv.org/abs/2512.16676.

[17] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. Advances in Neural Information Processing Systems (NeurIPS), 2023.

[18] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In International Conference on Learning Representations, volume 2024, pages 52989–53046, 2024.

[19] Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. Advances in Neural Information Processing Systems, 37:126544–126565, 2024.

[20] Bo Qiao, Liqun Li, Xu Zhang, Shilin He, Yu Kang, Chaoyun Zhang, Fangkai Yang, Hang Dong, Jue Zhang, Lu Wang, et al. Taskweaver: A code-first agent framework. arXiv preprint arXiv:2311.17541, 2023.

[21] Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Xuanhe Zhou, Yufei Huang, Chaojun Xiao, et al. Tool learning with foundation models. ACM Computing Surveys, 57(4):1–40, 2024.

[22] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. In International Conference on Learning Representations, volume 2024, pages 9695–9717, 2024.

[23] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In Advances in Neural Information Processing Systems, volume 36, 2023. URL https://arxiv.org/abs/2303.11366.

[24] Zhen Hao Wong, Jingwen Deng, Hao Liang, Runming He, Chengyu Shen, and Wentao Zhang. Flipvqa-miner: Cross-page visual question-answer mining from textbooks. arXiv preprint arXiv:2511.16216, 2025.

[25] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In First conference on language modeling, 2024.

[26] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

[27] Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. LIMO: Less is more for reasoning. arXiv preprint arXiv:2502.03387, 2025.

[28] Daochen Zha, Zaid Pervaiz Bhat, Kwei-Herng Lai, Fan Yang, Zhimeng Jiang, Shaochen Zhong, and Xia Hu. Data-centric artificial intelligence: A survey. arXiv preprint arXiv:2303.10158, 2023. URL https://arxiv.org/abs/2303.10158.