From Agent Loops to Deterministic Graphs: Execution Lineage for Reproducible AI-Native Work

Josh Rosen
ThruWire, Inc.

Seth Rosen
ThruWire, Inc.

May 7, 2026

Abstract

Large language model systems are increasingly deployed as agentic workflows that interleave reasoning, tool use, memory, and iterative refinement. These systems are effective at producing answers, but they often rely on implicit conversational state, making it difficult to preserve stable work products, isolate irrelevant updates, or propagate changes through intermediate artifacts.

We introduce execution lineage: an execution model in which AI-native work is represented as a directed acyclic graph (DAG) of artifact-producing computations with explicit dependencies, stable intermediate boundaries, and identity-based replay. The goal is not to make the model a better one-shot writer, but to make evolving AI-generated work maintainable under change.

We compare execution-lineage replay against loop-centric update baselines on two controlled policy-memo update tasks. In an unrelated-branch update, DAG replay preserved the final memo exactly in all runs, with zero churn and zero unrelated-branch contamination, while loop baselines regenerated the memo and frequently imported unrelated context. In an intermediate-artifact edit, all systems reflected the new constraint in the final memo, but only DAG replay achieved perfect upstream preservation, downstream propagation, unaffected-artifact preservation, and cross-artifact consistency.

These results show that final answer quality and maintained-state quality are distinct. Strong loop baselines can remain competitive at producing polished final outputs when the task is a bounded synthesis/update problem and all current sources fit in context, but immediate task success can mask partial state inconsistency that may compound over future revisions. Execution lineage provides stronger guarantees about what should change, what should remain stable, and how work evolves across revisions.

1 Introduction

Agent-based LLM systems have emerged as a dominant paradigm for applying large language models to complex tasks. Approaches such as ReAct [9] combine reasoning and action within iterative loops, enabling dynamic tool usage and adaptive planning.

This paradigm has been productive because it matches the interface of current language models: a prompt is assembled, the model responds, tools may be invoked, and the loop repeats until an answer appears satisfactory. In short-horizon tasks this is often sufficient. A user can tolerate some variance, intermediate reasoning can remain transient, and restarting the loop is usually cheaper than building explicit structure.

In practice, such systems rarely expose the raw model alone. They run inside an *agent harness*: a surrounding control layer that assembles prompts, manages tools and memory, and decides when execution should continue, branch, or terminate. Recent research has started to formalize the harness itself as an object of study rather than a hidden implementation detail [21, 22]. This shift is important because many system behaviors traditionally attributed to the model are in fact produced by the harness.

The problem changes once the work becomes multi-step, stateful, and long-lived. In those settings, the system is no longer producing a single answer in one conversational thread. It is generating intermediate research, analyses, transformations, and reports that may be revised later by humans or reused by other agents. What matters is not only whether the final answer is plausible, but whether the system can explain what it depended on, selectively recompute what changed, and preserve stable boundaries between stages of work.

Despite their flexibility, these systems exhibit systemic limitations:

- **Non-deterministic execution**: identical inputs may produce different outputs.
- **Implicit dependencies**: execution structure is embedded in conversational history.
- **Limited reproducibility**: intermediate reasoning is not preserved in structured form.
- **Global recomputation**: small changes require full re-execution.

These properties are acceptable for single-task workflows but break down for long-lived systems where consistency and incremental updates are required.

We propose a shift from **prompt lineage**—where state is encoded in evolving prompts—to **execution lineage**, where computation is represented explicitly as a graph.

The distinction is architectural. Prompt lineage records how prompts and transcripts evolved over time, but it does not provide a stable substrate on which downstream computation can depend. Execution lineage instead externalizes the structure of the work itself: each unit of work declares its inputs, its dependencies, its output contract, and its execution identity. Under this model, intermediate artifacts are first-class objects rather than incidental text inside a prompt window.

Our research is motivated by a practical observation: many failures attributed to "LLM unreliability" are in fact failures of system structure. If upstream results are passed around as ad hoc context, then reuse is heuristic, inspection is manual, and re-running a workflow means reconstructing state through prompts. If the same work is represented as a graph of explicit execution units with stable intermediate boundaries, then replay, observability, and partial recomputation become system properties rather than prompt engineering tricks.

A central distinction in this paper is between deliverable quality and maintained-state quality. A loop may produce a polished final answer from the current source bundle, yet still fail to preserve unaffected work, isolate unrelated updates, or maintain consistency across intermediate artifacts. That failure may not be visible after a single update, but it can leave behind partially incoherent state for future revisions. Execution lineage targets this second class of quality.

This paper makes three contributions. First, we articulate execution lineage as a computational abstraction for AI-native workflows. Second, we formalize a DAG-based execution model that separates authored structure from model-time generation and gives intermediate artifacts stable execution identities. Third, we provide a controlled empirical comparison showing that the advantage of execution lineage is clearest on maintained-state quality rather than final prose quality: DAG replay preserves stable outputs under unrelated updates and propagates intermediate artifact edits with stronger cross-artifact consistency than loop-centric baselines.

The paper's scope is intentionally narrow. Our primary concern is *how* AI-native work executes: what the unit of recomputation is, what makes a prior result replayable, and how dependency changes propagate. Richer questions about the semantics of intermediate artifacts or about multi-party co-authoring are important, but they are orthogonal to the execution-lineage thesis developed here.

2 Related Work

2.1 Surveys and Taxonomies

The literature on LLM agents has already grown large enough to generate several dedicated surveys. Recent reviews cover planning [1], memory [3], broader workflow paradigms spanning tool use, planning, and feedback learning [2], and, more recently, memory mechanisms and evaluation through early 2026 [4]. Together they show that the field has largely converged on the agent as a composite system rather than a raw model invocation. Most survey taxonomies, however, still classify systems by capabilities such as planning, reflection, memory, and tool use. Our taxonomy cuts across those dimensions by asking whether intermediate work remains embedded in prompts and transcripts or is exposed as stable computational structure with declared dependencies and replayable boundaries.

2.2 Agentic LLM Systems

ReAct [9] introduced interleaved reasoning and action. Toolformer [10] trains models to use tools via self-supervision. A parallel line of work studies agent frameworks and multi-agent organizations, including CAMEL [11], AutoGen [14], MetaGPT [13], and Voyager [12]. These systems show that substantial capability gains can be achieved by placing the model inside richer interaction loops, role structures, and tool environments.

These approaches share a control-loop architecture:

$$s_{t+1} = f(s_t, \text{LLM}, \text{tools}) \tag{1}$$

where state is implicit.

The strength of this family of systems is adaptability: loops can choose tools, revise prompts, and redirect work at runtime, which is especially useful for short-horizon and open-ended tasks. However, the same flexibility creates structural ambiguity. Dependencies are often encoded in conversational state rather than explicit program structure, and two executions that are semantically "the same" can differ because context was assembled differently or reasoning unfolded in another order. Recent work on planning agents, including Tree-of-Thoughts [15], improves search over reasoning trajectories but retains this basic property. In our framing, these methods improve inference *inside* a run, whereas execution lineage addresses structure *across* runs.

2.3 Agent Harnesses and Runtimes

The term *agent harness* has recently gained research traction as a way to name the surrounding machinery that makes an LLM agent operational. Natural-Language Agent Harnesses [21] explicitly studies harness engineering as a portable object, arguing that important agent behavior lives in editable control logic rather than only in model weights or prompts. General Modular Harness for LLM Agents in Multi-Turn Gaming Environments [22] similarly studies harness design as a composition of modules such as perception, memory, and reasoning.

This line of work is closely related to ours because it externalizes the wrapper around the model. Memory-oriented work pushes in the same direction: Agent Workflow Memory [23], On the Structural Memory of LLM Agents [24], WorkflowLLM [25], and more recent work such as LEGOMem [26], Memory-R1 [27], and Memori [28] all treat persistent procedural state as a meaningful systems layer rather than mere conversational carry-forward.

Our contribution differs in emphasis. Harness and memory papers ask how agent behavior can be encoded, adapted, or transferred at the controller level. We ask what execution model the harness should implement when workflows need deterministic replay, explicit lineage, and partial recomputation. In that sense, execution lineage can be understood as a systems-level proposal for what a rigorous harness runtime should optimize around. It is compatible with memory-augmented agents, but it is not reducible to memory. A memory system decides what to retain and retrieve; an execution-lineage system decides what the unit of computation is, what its dependencies are, and what exactly must be rerun when some upstream state changes.

2.4 Reasoning Traces, Search, and Intermediate Steps

Work on chain-of-thought and related prompting methods establishes an important empirical premise for our paper: intermediate reasoning steps often improve final-task performance. Chain-of-Thought [6], Self-Consistency [7], Least-to-Most prompting [8], and scratchpad-style methods [5] all show gains from exposing intermediate computation. Search-based extensions such as Tree-of-Thoughts [15], Language Agent Tree Search [18], Reflexion [16], Self-Refine [17], and structured reflection [19] push farther by revising and exploring alternative reasoning trajectories; earlier work on workflow-guided exploration [20] foreshadows the same interest in reusable action structure.

These papers matter here for two reasons. First, they show that intermediate steps are often where performance gains come from. Second, they reveal the limitation of prompt-centric intermediate state: most still represent intermediate work as textual traces tied to a specific execution episode. Our proposal builds on the empirical value of intermediate reasoning while relocating its substrate from transient text into explicit execution structure.

2.5 Workflow and DAG Systems

Workflow and dataflow systems such as Dryad [29] and Spark [30], as well as orchestration and analytics systems such as Airflow [31] and dbt [32], model execution around explicit dependency graphs, scheduled runs, and lineage-aware transformations.

These systems emphasize:

- explicit dependencies
- deterministic execution
- incremental recomputation

The relevance of these systems is not superficial. They reflect a mature systems answer to a recurring problem: when work becomes multi-stage and collaborative, hidden dependencies and ephemeral outputs do not scale. Data engineering historically moved from scripts and manual handoffs toward compiled graphs, materialized artifacts, and lineage-aware recomputation because these primitives were necessary for correctness and maintainability. We view AI-native workflows as reaching a similar inflection point.

That said, classical DAG systems were not designed around stochastic model calls, evolving prompts, or semantically rich intermediate artifacts. Their nodes typically represent deterministic code transformations with well-defined data contracts. Our work therefore does not merely import DAGs into AI systems; it adapts the DAG abstraction to settings where nodes may invoke LLMs, generate structured artifacts, and depend on both model configuration and upstream execution identity.

2.6 Programmatic LLM Systems

Emerging work treats LLMs as programs:

- DSPy [33]
- LMQL [34]
- PAL [36]
- Program of Thoughts [35]

These frameworks introduce valuable structure around prompting, constraints, optimization, and output shaping. Nevertheless, most remain primarily sequential or call-graph oriented rather than execution-lineage oriented. They improve how a single pipeline is programmed, but they generally do not make dependency materialization, replay identity, or partial downstream invalidation the organizing abstraction of the whole system.

2.7 Benchmarks and Reproducible Agent Environments

Another relevant literature studies how to evaluate agents in realistic yet reproducible environments. Benchmarks such as AgentBench [38], WebArena [39], VisualWebArena [40], WorkArena [41], AndroidWorld [42], OSWorld [43], AppWorld [44], GAIA [45], and LifelongAgentBench [46] increasingly evaluate agents in stateful, tool-using settings rather than static prompts.

Recent 2025–2026 benchmark work sharpens the memory and persistence angle in particular. MemoryAgentBench [47], Evo-Memory [48], and Mem2ActBench [49] all argue that static one-shot evaluation misses the harder problem of incremental accumulation, selective forgetting, and task-conditioned reuse. These benchmarks are not execution-lineage systems, but they are strong evidence that the field is moving toward persistent, longitudinal evaluation rather than isolated agent episodes.

This benchmark literature is important because it increasingly treats *environmental reproducibility* and *execution-based evaluation* as first-class concerns. However, benchmark reproducibility is not workflow reproducibility. A benchmark may make the external environment resettable and measurable while still leaving the internal execution structure of the agent implicit. Our work is complementary: we focus on making the agent's own computational lineage explicit and incrementally reusable.

2.8 Workflow Provenance and Interactive Analysis

There is also an emerging line of work on provenance-aware agent systems in scientific and workflow settings. LLM Agents for Interactive Workflow Provenance [50] studies how LLM agents can query and analyze workflow provenance records through modular interfaces. Connecting Large Language Model Agent to High Performance Computing Resource [51] examines how tool-using agents can operate over parallel and distributed execution substrates.

These papers are adjacent to our thesis because they treat workflows and provenance as serious systems objects rather than informal prompt context. The difference is that provenance work typically assumes an existing workflow system whose traces are queried after the fact. Execution lineage, as we define it, moves provenance into the execution substrate itself: lineage helps determine identity, replay, and invalidation during execution.

2.9 Reproducibility and Evaluation

LLM reproducibility challenges are well documented [37]. Variance arises from sampling, model updates, and tool interactions.

Prior work has correctly emphasized that reproducibility in language model systems is limited by multiple factors, including decoding stochasticity, external API variation, and infrastructure drift. Work on interactive evaluation [52] and realistic agent benchmarks [42, 43] likewise shows that even controlled environments exhibit sensitivity to rollout details and test conditions. We do not claim to eliminate all sources of variance. Instead, we isolate a source that is architectural and tractable: execution structure itself. Even when model and tool behavior are held fixed, prompt-oriented systems often reconstitute state heuristically at runtime. This introduces unnecessary instability and obscures provenance.

Our work complements reproducibility research by shifting attention from model behavior alone to the execution substrate around the model. The question is not only whether a model can reproduce a token sequence, but whether the system can reproduce a computational step, restore its exact intermediate state, and determine precisely which downstream work should be invalidated when an upstream artifact changes.

3 Execution Lineage Model

We define a workflow as:

$$G = (V, E) \tag{2}$$

where V is a set of executable nodes and E is a set of directed dependency edges. Each edge $(u, v) \in E$ states that node v may consume only artifacts explicitly produced by u and declared as part of its input surface.

Each node:

$$v = (\mathcal{I}_v, f_v, \mathcal{O}_v) \tag{3}$$

where $\mathcal{I}_v$ denotes the resolved local input state available to node v, f_v is the node-local execution procedure, and $\mathcal{O}_v$ is the structured output artifact selected as the node's result. Artifacts are not arbitrary transcripts; they are typed outputs with stable boundaries intended for downstream consumption.

This formulation differs from agent loops in two ways. First, the admissible context for each node is declared before execution rather than reconstructed opportunistically from conversational state. Second, node outputs persist as addressable artifacts with explicit lineage, enabling downstream reuse and inspection.

3.1 Design Principles

The execution-lineage model can be summarized by four design principles.

Explicit dependency declaration. A node should not be allowed to consume undeclared upstream state. This principle forces the graph to carry the causal structure of the workflow rather than hiding it in prompt assembly logic.

Local visibility boundaries. Each node should see only the context and dependency artifacts required for its task. Besides improving provenance, this reduces accidental coupling, lowers prompt bloat, and makes it easier to understand why a node produced a given output.

Identity-based reuse. Reuse should be based on proof of equivalence, not similarity of prompts or approximate resemblance of outputs. A runtime should be able to say not merely that two executions "look close enough," but that they share the same declared structure and resolved inputs.

Deterministic publication. Even when execution involves internal iteration, branching, or validation, the boundary exposed downstream should be stable and canonical. Downstream nodes should depend on a published result, not on whatever happened to be most recently present in a conversational trace.

Taken together, these principles describe a runtime whose main object is not the prompt or the transcript, but the dependency-respecting publication of intermediate computation.

3.2 Intermediate State and Typed Local Boundaries

The core execution object is the intermediate state boundary. In our implementation this is materialized as an artifact, but the deeper claim is executional rather than representational: the system needs a stable, inspectable unit that downstream computation can depend on. An artifact may contain free-form model output, but it is treated by the runtime as a structured unit with an identity, type, and provenance. This matters because the role of an intermediate result is not merely to be read by a human; it is to become a stable dependency surface for later computation.

Each node executes against a typed local state consisting of three classes of entries: immutable context provided at invocation time, immutable dependency artifacts materialized from upstream nodes, and mutable local artifacts produced during the node's own execution. This separation is important. Without it, a runtime cannot distinguish inherited state from newly produced state, and downstream reasoning about provenance collapses back into transcript reconstruction.

3.3 Dependency Materialization

Dependencies are resolved against actual inputs rather than only logical node names. Concretely, an upstream dependency is identified not only by which node produced it, but by the particular resolved input surface under which that node was executed. This prevents two semantically distinct executions of the same logical node from being aliased together.

Formally, let k_v denote the execution identity of node v. We define:

$$k_v = h(\sigma_v, x_v, \{k_u : u \in \text{pred}(v)\}) \tag{4}$$

where σ_v is the structural specification of node v, x_v is its resolved input hash, and $\text{pred}(v)$ are its immediate predecessors. The hash function h need not be cryptographically special for our argument; what matters is that node identity is a deterministic function of structure, inputs, and upstream identities.

This execution identity provides the basis for cache reuse, replay, and selective invalidation. If k_v is unchanged, prior execution may be reused exactly. If it changes, downstream nodes that depend on k_v are invalidated and re-executed, while unrelated branches remain untouched.

3.4 Execution Semantics

Execution is a forward pass:

$$\mathcal{O}_v = f_v(\mathcal{I}_v) \tag{5}$$

Nodes execute once dependencies are satisfied. At runtime, readiness is decided from the graph rather than inferred by the model. This allows the system to schedule independent branches concurrently while preserving sequential semantics within each node's local procedure.

In practice, a node may itself contain multiple internal steps, such as context selection, model invocation, validation, or rendering. We treat these as part of f_v. They may be sophisticated, but they remain subordinate to a platform-controlled execution boundary. The model generates content within the node; it does not decide which nodes exist or which undeclared upstream state may be read.

3.5 Determinism

Definition: Execution is deterministic if outputs are reproducible under:

- fixed inputs
- fixed model + parameters
- fixed tool outputs

Under these conditions, determinism means more than "similar answers." It means that the system can either replay a prior node execution exactly or prove that some aspect of its identity changed and therefore recomputation is required. Determinism is therefore a property of the execution substrate, not a promise that the model itself is universally stable under all deployments.

3.6 Replay and Partial Recomputation

Execution lineage makes replay a first-class runtime mode. When a node's execution identity matches a previously persisted result, the runtime can restore its artifact and associated execution record rather than regenerate it heuristically. This differs from approximate caching: replay is identity-based and therefore explainable.

Similarly, partial recomputation follows directly from graph structure. If an upstream research artifact changes, only descendants whose execution identities depend on that artifact must be recomputed. The runtime can leave unrelated branches intact. In contrast, prompt-centric systems often collapse all prior work into one evolving context window, making a local change expensive because the system lacks precise invalidation boundaries.

3.7 Invalidation Semantics

The usefulness of an execution graph depends not only on replay, but on *correct* replay refusal. A lineage-aware runtime must be able to explain both why prior work can be reused and why it can no longer be reused. Invalidation therefore becomes a semantic operation over dependency identities rather than a vague sense that "the prompt changed."

Consider a simple three-stage workflow consisting of retrieval, analysis, and synthesis. If only the synthesis prompt changes, the system should preserve retrieval and analysis while recomputing only synthesis. If a retrieval source changes, analysis and synthesis must be invalidated because their dependency identities are downstream of the changed source. This style of invalidation is routine in build systems and dataflow engines, but remains under-specified in many agent systems, where upstream changes are usually handled by restarting or partially reconstructing a prompt transcript.

This distinction matters operationally. Without explicit invalidation semantics, teams tend to choose between two unsatisfactory behaviors: stale reuse of outputs that no longer match current assumptions, or expensive global reruns that throw away valid prior work. Execution lineage aims to replace that ambiguity with a principled middle ground.

3.8 Canonical Outputs and Observability

An execution system also needs a notion of which output is authoritative at a boundary. In loop-based systems, candidate drafts, self-critiques, tool observations, and final responses are often mixed together in one transcript. A human can often infer which part "counts," but the runtime itself lacks a reliable canonical object for downstream consumers.

Execution-lineage systems instead select a canonical output at each node boundary. That output may have been produced through multiple internal steps, validations, or revisions, but once selected it becomes the stable downstream dependency surface. This improves observability in two ways. First, it lets operators inspect the exact artifact that downstream nodes consumed, rather than rereading an entire transcript. Second, it creates a durable execution history in which intermediate candidates, final selected outputs, and dependency identities can be distinguished rather than conflated.

3.9 Execution Lineage vs. Prompt Lineage

Prompt lineage records the history of prompts, messages, and tool observations used during an interaction. It is useful for audit and narration, but it is a poor substrate for composition because it lacks stable execution boundaries. Execution lineage, by contrast, records the graph of artifact-producing computations and the identities that connect them.

The distinction parallels the difference between logs and program structure. Logs are valuable for understanding what happened after the fact; they are not the mechanism by which a system should declare dependencies or reuse work. In our model, reasoning traces may still exist, but they are auxiliary observability surfaces rather than the system's computational backbone.

4 Execution vs Agent Loop

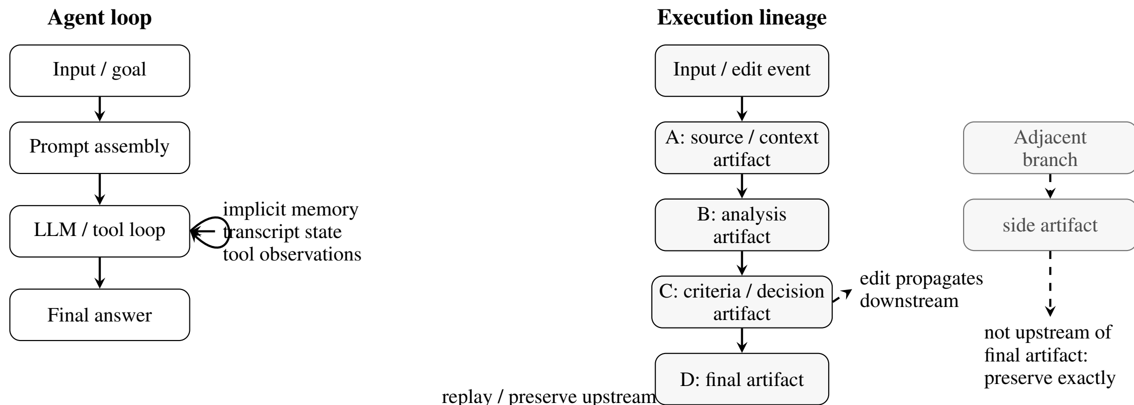

Figure 1: Agent loops carry work forward through prompt context and transcript state. Execution lineage represents work as explicit artifact-producing nodes with declared dependencies, allowing the runtime to decide which artifacts to replay, recompute, or preserve after an edit.

5 Research Questions

- **RQ1: Dependency isolation.** When an update occurs outside the dependency path of a final artifact, can the system preserve the final artifact exactly rather than rewriting it?
- **RQ2: Artifact-level propagation.** When an intermediate artifact is edited, can the system update downstream dependents while preserving upstream and unrelated artifacts?
- **RQ3: Maintained-state quality.** Do execution-lineage workflows outperform loop-centric workflows on maintained-state properties such as preservation, propagation, unrelated-branch isolation, and cross-artifact consistency?
- **RQ4: Deliverable-quality boundary.** When all systems can update the visible final memo, do maintained-state metrics reveal differences that final-answer metrics miss?
- **RQ5: Efficiency tradeoffs.** How do token use, model-call count, and wall-clock time differ between loop updates and DAG replay under unrelated-branch versus downstream-propagating edits?

The first two questions correspond to the two experimental interventions; the latter three characterize the broader distinction between visible deliverable correctness and maintained-state correctness.

6 Methods

6.1 Study Design

We use a within-task comparative design over two controlled policy-memo update tasks. We instantiate the tasks in a telehealth expansion scenario so that source updates, recommendation criteria, and final deliverables can be inspected concretely. Each task is run under three conditions: a naturalistic loop update, a loop update with explicit edit-event awareness, and execution-lineage DAG replay. All conditions use the same model family and source materials. The comparison is system-level rather than equal-prompt: the DAG condition receives runtime-derived dependency and edit-lineage state because that state is the architectural substrate under evaluation. Loop conditions rely on implicit reconstruction from the prior memo, current materials, and, in one condition, the edit event.

6.2 Experimental Conditions

For readability, we refer to these conditions in the rest of the paper as *loop final update*, *loop + edit event*, and *DAG replay*.

- **Loop final update** (`loop_real_world_final_update`). A naturalistic loop baseline. It receives the prior final memo, current source materials, the requested output format, and a normal update request. It does not receive a dependency graph, affected or unaffected claim IDs, recomputation scope, allowed or disallowed artifact lists, oracle labels, or the correct answer.
- **Loop + edit event** (`loop_real_world_with_edit_event`). The same loop baseline, but also given the source or artifact edit event. Its purpose is to test whether the DAG advantage is merely knowing what changed. It still does not receive a dependency graph, downstream propagation plan, affected or unaffected labels, recomputation stages, oracle labels, or the correct answer.
- **DAG replay** (`simple_dag_replay_selective_recompute`). The execution-lineage condition. It receives the artifact or source edit event, artifact identities, the dependency graph, a dependency-derived recomputation plan, and preserved artifact state. It does not receive judge-only labels, the correct final recommendation, the hidden scoring rubric, or oracle affected-claim IDs unless the runtime itself derives them.

6.3 Tasks

- `unrelated_branch_noop_update`. This task tests dependency isolation. The maintained work product is a telehealth policy memo, while the update occurs in a provider recruiting and staffing branch that is plausible and adjacent but not upstream of the final telehealth memo. Correct behavior is to preserve the final telehealth memo exactly and avoid importing recruiting or staffing evidence into the telehealth recommendation.
- `intermediate_artifact_edit`. This task tests artifact-level propagation. A human or agent edits an intermediate recommendation-criteria artifact to impose a new constraint: year-one telehealth expansion must be budget-neutral and must include utilization controls before chronic-care follow-up expansion can scale. Correct behavior is to preserve upstream evidence artifacts, propagate the edit downstream to the implementation plan and final memo, preserve unaffected artifacts, and keep the final memo consistent with the edited criteria.

6.4 Metrics

Unrelated-branch update metrics

- exact preservation, measured by `final_output_exact_match` and `final_output_hash_preserved`
- stable-artifact preservation and churn, including `stable_artifact_hash_preservation` and `unnecessary_churn_rate`
- unrelated-branch contamination, measured by `unrelated_branch_contamination_rate`
- semantic correctness proxies, including `output_faithfulness_score` and `current_state_precision_score`
- efficiency measures: input and output tokens, model-call count, and wall-clock time

Intermediate-artifact edit metrics

- final-memo constraint reflection, measured by `final_memo_constraint_reflection`
- cross-artifact consistency, measured by `cross_artifact_consistency_score`
- stable-artifact preservation and upstream churn, including `stable_artifact_hash_preservation`, `upstream_churn_rate`, and `unaffected_artifact_preservation`
- downstream propagation, measured by `downstream_propagation_recall`
- semantic correctness proxies, including `output_faithfulness_score` and `current_state_precision_score`
- efficiency measures: input and output tokens, model-call count, and wall-clock time

For the unrelated-branch task, contamination is counted only when the final memo uses, cites, or imports content from the provider-recruiting branch into the telehealth recommendation. The presence of the provider-recruiting artifact elsewhere in the DAG state is not counted as contamination. For the intermediate-edit task, cross-artifact consistency measures whether the final memo remains aligned with the maintained intermediate artifacts after the criteria edit. Constraint reflection measures whether the final memo includes the new budget-neutrality and utilization-control constraint. These semantic metrics are evaluated from stored outputs and judge inputs. Hash preservation, churn, token

counts, model-call counts, and wall-clock time are computed directly from run artifacts; hash-preservation metrics compare the updated artifact against the pre-edit artifact, and a score of 1.0 means exact byte-level preservation.

Each task-condition pair is run with $n = 3$ repeats and semantic judging uses two judge passes per repeat. The reported run uses `GPT-5.2` at temperature 0.7 with repo commit `f6cc8679ab1178b22fa01c263aa14e77454081ee`. Deterministic metrics are computed from artifact hashes, replay records, and churn heuristics, while judge-like scores are used only for specific semantic fields such as contamination, constraint reflection, and cross-artifact consistency. No generation condition receives judge-only metadata.

7 Experimental Setup

The controlled policy-memo setting is instantiated as a telehealth expansion memo about whether a multistate health system should expand behavioral-health and chronic-care follow-up visits. The DAG condition represents the work as staged artifacts including utilization context, reimbursement context, operations context, access/cost context, claim matrix, tension analysis, recommendation criteria, implementation plan where applicable, and final memo. The unrelated-branch task adds an adjacent provider-recruiting branch outside the final memo's dependency path. The intermediate-edit task modifies recommendation criteria with a budget-neutrality and utilization-control constraint.

Across all conditions, we hold constant:

- the model family
- the source materials
- the task domain
- stored rendered prompts
- stored outputs
- stored judge inputs and outputs

The loop conditions do not receive the dependency graph. This is intentional: the comparison is between implicit context reconstruction and explicit runtime lineage, not between equalized prompt contents.

8 Results

8.1 Overview: Maintained-State Quality

The controlled update tasks show a clear separation between final deliverable correctness and maintained-state correctness. In the unrelated-branch update, DAG replay preserved the final memo exactly while the loop baselines regenerated and often contaminated it. In the intermediate-artifact edit, all systems updated the final memo correctly, but only DAG replay maintained perfect upstream preservation, downstream propagation, and cross-artifact consistency. Preservation and propagation are two sides of maintained-state quality: a system must know both when an artifact should remain fixed and when a change should flow downstream. Tables 1 and 2 summarize the two controlled update tasks.

Table 1: Dependency-isolation results for the unrelated-branch update ($n = 3$). The correct behavior is exact preservation without unrelated-branch contamination.

Condition	Exact preserve	Churn	Contam.	Input tokens	Model calls	Wall-clock
Loop final update	0.00	0.923	0.667	11655	1	33.5s
Loop + edit event	0.00	0.908	1.000	11694	1	55.0s
DAG replay	1.00	0.000	0.000	382	1	7.9s

Table 2: Artifact-level propagation results for the intermediate-criteria edit ($n = 3$). All systems reflected the new constraint; only DAG replay preserved upstream state and maintained full cross-artifact consistency.

Condition	Constraint reflected	Cross-artifact consist.	Stable artifact preservation	Downstream propagation	Upstream churn	Input tokens	Model calls	Wall-clock
Loop final update	1.00	0.50	–	–	–	11404	1	38.7s
Loop + edit event	1.00	0.50	–	–	–	11464	1	31.1s
DAG replay	1.00	1.00	1.00	1.00	0.00	8461	2	73.6s

8.2 Dependency Isolation: Unrelated Branch Update

The unrelated-branch task is the most direct dependency-isolation test in this study. The correct behavior is preservation: the system should determine that the recruiting update is outside the final memo's dependency path and therefore should not alter it. As Table 1 shows, DAG replay preserved the final memo exactly in $3/3$ runs, with `final_output_exact_match` and `final_output_hash_preserved` both equal to 1.00. It also achieved `stable_artifact_hash_preservation` of 1.00, zero churn, and zero unrelated-branch contamination.

The loop baselines regenerated the final memo in $3/3$ runs. The loop final update condition contaminated the memo in $2/3$ runs, while the edit-event loop contaminated it in $3/3$ runs. This matters because the edit-event loop knew what changed at the source level yet still regenerated the memo and imported unrelated branch content. In this task, source-change awareness alone was insufficient; the distinguishing factor was dependency structure rather than edit-event awareness alone.

The efficiency difference was also large on this unrelated-branch update. DAG replay used 382 input tokens versus about 11.7k for either loop baseline, or about $30.5\times$ fewer than loop final update and $30.6\times$ fewer than the edit-event loop. Output tokens were also lower for DAG replay (440.7) than for the two loops (2189.3 and 2078.3). It was also about $4.2\times$ faster than loop final update (33.5s vs. 7.9s) and about $6.9\times$ faster than the edit-event loop (55.0s vs. 7.9s). Faithfulness and current-state precision followed the same pattern: DAG replay scored 1.0 on both, while the loop baselines in this study scored 0.0 faithfulness with current-state precision of 0.333 and 0.0, respectively. In this task, DAG replay achieved exact preservation and lower contamination with substantially lower token use and wall-clock time.

8.3 Artifact-Level Propagation: Intermediate Criteria Edit

The intermediate-artifact task provides the complementary downstream-propagation test and is at least as important conceptually as the unrelated-branch update. Here the correct behavior is not preservation alone. The system must propagate a meaningful edit through downstream artifacts while preserving upstream evidence and unrelated state. Table 2 reports the core task-specific metrics.

All three conditions reflected the new constraint in the final memo, with `final_memo_constraint_reflection` equal to 1.00 in every case. All three also achieved `output_faithfulness_score` and `current_state_precision_score` of 1.00. A final-memo-only evaluation would therefore treat all systems as successful. The maintained-state metrics reveal the distinction.

DAG replay achieved perfect `stable_artifact_hash_preservation` (1.00), `downstream_propagation_recall` (1.00), `upstream_churn_rate` (0.00), `unaffected_artifact_preservation` (1.00), and `cross_artifact_consistency_score` (1.00). The loop baselines reflected the edit in the final memo, while DAG replay preserved and updated the surrounding artifact state consistently. Both loop baselines reflected the new constraint in the final memo, but each reached only 0.50 cross-artifact consistency because the updated final deliverable was not consistently aligned with maintained intermediate state. All three conditions had `final_output_exact_match` and `final_output_hash_preserved` equal to 0.0, which is expected here because the final memo should change.

This task also shows that the results do not support a general claim of wall-clock superiority. DAG replay used fewer input tokens than the loops (8461 versus 11404 and 11464, or about $1.35\times$ and $1.36\times$ less input context), but it produced more output tokens (4311.3 versus 2418.7 and 1981.0) and was slower in wall-clock time because the implementation recomputed downstream artifacts sequentially in two model calls. It took 73.6s versus 38.7s for loop final update and 31.1s for the edit-event loop, making it about $1.9\times$ and $2.36\times$ slower, respectively.

8.4 Deliverable Quality vs. Maintained-Work Quality

The intermediate-edit task illustrates why final-output metrics alone are insufficient. A final-memo-only evaluation would score all three systems as successful because all reflected the new constraint. The maintained-state metrics reveal the distinction: only DAG replay preserved upstream artifacts, propagated the edit downstream, preserved unaffected artifacts, and maintained consistency between intermediate artifacts and the final memo.

This result sharpens the comparison target. Strong loop baselines can remain competitive on final deliverable quality when the task is a bounded synthesis or update problem and all current sources fit in context. The main difference in this study appears in the maintained state: preservation, isolation, propagation, and cross-artifact consistency under revision.

8.5 Efficiency Tradeoffs

The reported results do not support a blanket claim that DAG replay is always faster. On the unrelated-branch update, DAG replay achieved higher maintained-state correctness with lower token use and wall-clock time because it reused preserved state and scoped recomputation tightly. On the intermediate edit, DAG replay used less input context but was slower in wall-clock time due to sequential multi-stage replay. The implication is narrower: execution lineage improves maintained-state quality under change, while efficiency depends on the structure of the update and the runtime's replay implementation.

8.6 Information Boundaries

Condition	Prior memo/artifacts	Current sources	Edit event	Dependency graph	Selective recomputation	Oracle labels
Loop final update	yes	yes	no	no	no	no
Loop + edit event	yes	yes	yes	no	no	no
DAG replay	yes	scoped	yes	yes	yes	no

Table 3: Information boundaries by condition. The comparison is system-level, not an equal-prompt comparison.

9 Discussion

9.1 Why This Matters

Agent loops optimize for task completion. Execution graphs optimize for:

- persistence
- evolution
- collaboration

This difference becomes more important as AI systems move from one-shot assistants to persistent work environments. In many real settings, the goal is not merely to get an answer once. The goal is to build a body of work that can be revised, extended, audited, and shared across humans and agents. A system that stores only prompts and final answers leaves too much of the work trapped inside transient execution.

Execution lineage changes the unit of improvement. Instead of iterating only on a final output, teams can iterate on the structure that produces outputs. A change to an upstream methodology, research constraint, or transformation rule can propagate through the graph in a principled way. This resembles the transition seen in data engineering, where the object of engineering shifted from isolated scripts to the dependency system that generated downstream tables and reports.

The practical collaboration benefit follows from this, but is not the paper's main claim. Once intermediate boundaries and dependencies are explicit, humans and agents can intervene at meaningful points rather than by restating instructions into a loop. However, the novelty we emphasize here is the runtime substrate that makes such interventions precise: explicit invalidation boundaries, replayable identities, and deterministic scheduling over a graph.

9.2 Immediate Task Success vs. Maintained-State Quality

The experiments suggest that final prose quality is an incomplete measure for long-lived AI work because it rewards immediate task completion more than the health of the maintained state. In the intermediate-artifact task, all systems produced a final memo that reflected the new constraint, so a final-answer-only evaluation would treat them as successful. The maintained-state metrics reveal the difference: the loop baselines updated the visible deliverable but left only partial cross-artifact consistency, while DAG replay preserved upstream artifacts, propagated the edit downstream, and kept the final output aligned with the intermediate state.

The deeper risk is longitudinal state drift. A loop-centric update can satisfy the immediate task while leaving the surrounding artifact state only partially coherent. That may be acceptable after one revision, but repeated updates could compound the inconsistency as future work inherits artifacts that no longer agree about the current criteria, evidence base, or implementation assumptions. Our experiments expose this mechanism but do not directly measure long-horizon accumulation; testing whether these inconsistencies compound over many revisions remains future work.

9.3 Controlled Propagation

The no-op result should not be read narrowly as a claim that execution lineage is useful only when work should remain unchanged. It is one instance of a broader property: dependency-aware control over change propagation. The same mechanism that prevents irrelevant updates from reaching the final memo also allows intermediate artifact edits to propagate only to downstream dependents. Preservation and propagation are two sides of the same execution-lineage property.

9.4 When Not to Regenerate

The unrelated-branch update illustrates a failure mode of loop-centric systems that is rarely measured: unnecessary regeneration. The correct behavior was to leave the final memo untouched. Both loop baselines rewrote it, and the edit-event loop was not protected by knowing which source changed. This suggests that source-change awareness is not enough; a system also needs dependency structure to decide whether a change is relevant to a given artifact.

9.5 Where Execution Graphs Help Most

Execution lineage is unlikely to matter equally for all AI tasks. Its advantages should be most visible in workflows with several properties.

- **Dependency isolation**: the workflow contains branches whose outputs should remain stable under adjacent updates.
- **Unrelated-branch isolation**: plausible nearby edits should not contaminate downstream artifacts.
- **Intermediate artifact edits**: downstream state must change while upstream evidence remains fixed.
- **Cumulative revisions**: the workflow is revised repeatedly rather than discarded after one answer.
- **Shared maintained artifacts**: humans or other agents inspect and revise intermediate outputs directly.
- **Asymmetric recomputation cost**: some stages are expensive or slow enough that global reruns are operationally undesirable.

These properties describe a broad class of realistic AI-native work: research briefs, analysis pipelines, coding workflows, forecasting systems, compliance reviews, and operational runbooks. In such settings, the key question is rarely "can the model produce an answer?" The harder question is whether the system can evolve that answer over time without losing its own structure.

9.6 What This Paper Does Not Claim

The argument here should not be overstated. We do not claim that deterministic graphs replace all useful forms of agentic exploration. Open-ended discovery, broad brainstorming, and ambiguous early-stage problem framing may still benefit from unconstrained loops, reflection, and search. Nor do we claim that explicit execution structure by itself guarantees better one-shot answer quality. A poorly designed graph can still route bad prompts, weak tools, or misleading inputs.

What we claim is narrower and more systems-oriented. Once a workflow has repeated stages, reusable intermediate state, and nontrivial update pressure, leaving its structure implicit becomes a liability. At that point, the graph is not merely one possible representation among many. It becomes a practical mechanism for preserving correctness, reducing unnecessary recomputation, and making the system legible to its operators.

9.7 Limitations

This evaluation is intended as a controlled mechanism study rather than a comprehensive benchmark. It uses two controlled update tasks in one policy-memo domain, three system conditions, three repeats, and one model family. This scope is appropriate for isolating dependency isolation and artifact-level propagation, but it does not establish broad performance claims across all agentic workflows.

The loop baselines are naturalistic but not exhaustive; stronger custom loop harnesses with explicit state tracking could narrow some gaps. The DAG implementation is sequential, so wall-clock time does not always improve even when input context and recomputation are reduced. The graph must also be correctly specified: dependency mistakes, poor source routing, or lossy artifact interfaces can reduce the advantage. The experiments expose a mechanism by which

partial cross-artifact inconsistency could lead to state drift, but they do not directly measure multi-round accumulation; longitudinal revision studies are needed to test that hypothesis. Finally, the results should not be read as evidence that DAG replay generally produces better prose than a holistic loop rewrite.

10 Conclusion

We introduced execution lineage as an execution substrate for AI-native work represented as artifact-producing DAGs with explicit dependencies. The empirical results sharpen the claim: execution lineage is not primarily a technique for making models better one-shot writers. Its value is in maintaining work under change.

The controlled update tasks show both sides of execution lineage. In the unrelated-branch update, DAG replay preserved stable work exactly and isolated an adjacent but non-dependent branch. In the intermediate-artifact edit, DAG replay propagated a meaningful change through downstream artifacts while preserving upstream state. Loop baselines were able to update visible final memos, but they were less reliable at preserving stable outputs and maintaining cross-artifact consistency.

These results suggest that persistent AI workflows need evaluation criteria beyond final answer quality. For long-lived work, correctness includes knowing what changed, what did not, and why. Execution lineage provides a systems abstraction for making those properties explicit.

References

[1] Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. Understanding the Planning of LLM Agents: A Survey. arXiv:2402.02716, 2024.

[2] Xinzhe Li. A Review of Prominent Paradigms for LLM-Based Agents: Tool Use (Including RAG), Planning, and Feedback Learning. arXiv:2406.05804, 2024.

[3] Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A Survey on the Memory Mechanism of Large Language Model Based Agents. arXiv:2404.13501, 2024.

[4] Pengfei Du. Memory for Autonomous LLM Agents: Mechanisms, Evaluation, and Emerging Frontiers. arXiv:2603.07670, 2026.

[5] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show Your Work: Scratchpads for Intermediate Computation with Language Models. arXiv:2112.00114, 2021.

[6] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, 2022.

[7] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-Consistency Improves Chain of Thought Reasoning in Language Models. arXiv:2203.11171, 2022.

[8] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. arXiv:2205.10625, 2022.

[9] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations*, 2023.

[10] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems*, 2023.

[11] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society. arXiv:2303.17760, 2023.

[12] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models. arXiv:2305.16291, 2023.

[13] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. arXiv:2308.00352, 2023.

[14] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv:2308.08155, 2023.

[15] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems*, 2023.

[16] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Advances in Neural Information Processing Systems*, 2023.

[17] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems*, 2023.

[18] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models. arXiv:2310.04406, 2023.

[19] Tao Li, Gang Li, Zhiwei Deng, Bryan Wang, and Yang Li. A Zero-Shot Language Agent for Computer Control with Structured Reflection. arXiv:2310.08740, 2023.

[20] Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement Learning on Web Interfaces Using Workflow-Guided Exploration. arXiv:1802.08802, 2018.

[21] Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-Language Agent Harnesses. arXiv:2603.25723, 2026.

[22] Yuxuan Zhang, Haoyang Yu, Lanxiang Hu, Haojian Jin, and Hao Zhang. General Modular Harness for LLM Agents in Multi-Turn Gaming Environments. arXiv:2507.11633, 2025.

[23] Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory. arXiv:2409.07429, 2024.

[24] Ruihong Zeng, Jinyuan Fang, Siwei Liu, and Zaiqiao Meng. On the Structural Memory of LLM Agents. arXiv:2412.15266, 2024.

[25] Shengda Fan, Xin Cong, Yuepeng Fu, Zhong Zhang, Shuyan Zhang, Yuanwei Liu, Yesai Wu, Yankai Lin, Zhiyuan Liu, and Maosong Sun. WorkflowLLM: Enhancing Workflow Orchestration Capability of Large Language Models. arXiv:2411.05451, 2024.

[26] Dongge Han, Camille Couturier, Daniel Madrigal Diaz, Xuchao Zhang, Victor Rühle, and Saravan Rajmohan. LEGOMem: Modular Procedural Memory for Multi-agent LLM Systems for Workflow Automation. arXiv:2510.04851, 2025.

[27] Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Hinrich Schütze, Volker Tresp, and Yunpu Ma. Memory-R1: Enhancing Large Language Model Agents to Manage and Utilize Memories via Reinforcement Learning. arXiv:2508.19828, 2025.

[28] Luiz C. Borro, Luiz A. B. Macarini, Gordon Tindall, Michael Montero, and Adam B. Struck. Memori: A Persistent Memory Layer for Efficient, Context-Aware LLM Agents. arXiv:2603.19935, 2026.

[29] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

[30] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *USENIX HotCloud*, 2010.

[31] Apache Software Foundation. Airflow Documentation: DAGs. `https://airflow.apache.org/docs/apache-airflow/3.0.4/core-concepts/dags.html`, accessed May 4, 2026.

[32] dbt Labs. dbt Developer Hub. `https://docs.getdbt.com/`, accessed May 4, 2026.

[33] Omar Khattab, Keshav Santhanam, Xiang Lisa Li, Aatmik Gupta, Christopher Potts, and Matei Zaharia. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. arXiv:2310.03714, 2023.

[34] Leon Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting Is Programming: A Query Language for Large Language Models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2023.

[35] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks. arXiv:2211.12588, 2022.

[36] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided Language Models. In *International Conference on Machine Learning*, 2023.

[37] Lingjiao Chen, Matei Zaharia, and James Zou. How Is ChatGPT's Behavior Changing over Time? arXiv:2307.09009, 2023.

[38] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as Agents. arXiv:2308.03688, 2023.

[39] Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A Realistic Web Environment for Building Autonomous Agents. arXiv:2307.13854, 2023.

[40] Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. VisualWebArena: Evaluating Multimodal Agents on Realistic Visual Web Tasks. arXiv:2401.13649, 2024.

[41] Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, Léo Boisvert, Megh Thakkar, Quentin Cappart, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How Capable Are Web Agents at Solving Common Knowledge Work Tasks? arXiv:2403.07718, 2024.

[42] Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, and Oriana Riva. AndroidWorld: A Dynamic Benchmarking Environment for Autonomous Agents. arXiv:2405.14573, 2024.

[43] Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking Multimodal Agents for Open-Ended Tasks in Real Computer Environments. arXiv:2404.07972, 2024.

[44] Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents. arXiv:2407.18901, 2024.

[45] Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: A Benchmark for General AI Assistants. arXiv:2311.12983, 2023.

[46] Junhao Zheng, Xidi Cai, Qiuke Li, Duzhen Zhang, ZhongZhi Li, Yingying Zhang, Le Song, and Qianli Ma. LifelongAgentBench: Evaluating LLM Agents as Lifelong Learners. arXiv:2505.11942, 2025.

[47] Yuanzhe Hu, Yu Wang, and Julian McAuley. Evaluating Memory in LLM Agents via Incremental Multi-Turn Interactions. arXiv:2507.05257, 2025.

[48] Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H. Chi, Chi Wang, Shuo Chen, Fernando Pereira, Wang-Cheng Kang, and Derek Zhiyuan Cheng. Evo-Memory: Benchmarking LLM Agent Test-time Learning with Self-Evolving Memory. arXiv:2511.20857, 2025.

[49] Yiting Shen, Kun Li, Wei Zhou, and Songlin Hu. Mem2ActBench: A Benchmark for Evaluating Long-Term Memory Utilization in Task-Oriented Autonomous Agents. arXiv:2601.19935, 2026.

[50] Renan Souza, Timothy Poteet, Brian Etz, Daniel Rosendo, Amal Gueroudji, Woong Shin, Prasanna Balaprakash, and Rafael Ferreira da Silva. LLM Agents for Interactive Workflow Provenance: Reference Architecture and Evaluation Methodology. arXiv:2509.13978, 2025.

[51] Heng Ma, Alexander Brace, Carlo Siebenschuh, Greg Pauloski, Ian Foster, and Arvind Ramanathan. Connecting Large Language Model Agent to High Performance Computing Resource. arXiv:2502.12280, 2025.

[52] Josh Abramson, Arun Ahuja, Federico Carnevale, Petko Georgiev, Alex Goldin, Alden Hung, Jessica Landon, Timothy Lillicrap, Alistair Muldal, Blake Richards, Adam Santoro, Tamara von Glehn, Greg Wayne, Nathaniel Wong, and Chen Yan. Evaluating Multimodal Interactive Agents. arXiv:2205.13274, 2022.