DAF: A Retrospective Paired Case Study of Fixed Agent Workflows

Darshankumar Joshi
darshjme@gmail.com
Preprint 1.1 · 29 October 2026 · Not peer reviewed

Abstract

Agent workflows combine model inference with scheduling, dependency transfer, and result storage. These infrastructure choices can affect completion time even when the model and task remain nominally unchanged. This paper presents Darshj's Agent Framework (DAF), a Rust framework for explicit agent execution, and examines a historical experimental integration against native Codex subagent coordination. In eight paired synthetic ledger workflows using the recorded GPT-6 Astra configuration at medium reasoning effort, the retained scorer marked all eight runs in each arm correct; the median paired native-to-DAF wall-time ratio was 3.65. A separate corrected coding experiment contained two pairs, with all four workflows passing the retained evaluation using the same 187 behavioral cases; its median paired ratio was 2.49. These are descriptive findings from four ledger fixtures and one coding task family. Different coordinator behavior, internal model context, and service conditions prevent attributing the differences to the transport protocol or generalizing them to open-ended software engineering. We distinguish the historical benchmark adapter from the current repository and describe the current implementation's bounded execution and durable-result contracts. The contribution is an inspectable engineering case study with explicit measurement boundaries, failed-harness disclosure, and a reproducibility agenda, rather than a claim of superior reasoning or universally lower cost.

Keywords: agent orchestration; workflow execution; multi-agent systems; reproducibility; Rust; durable results

1. Introduction

An agent framework determines more than how a model is prompted. It determines when work starts, how dependencies become available, what constitutes a valid result, and how uncertainty is handled after interruption. A model-driven coordinator can choose a flexible decomposition, while an explicit dependency graph can remove scheduling decisions from the inference path. Whether this produces a useful latency difference is an empirical question whose answer depends on workload and implementation.

DAF provides infrastructure components for agent execution, graph scheduling, communication, memory, and result handling [1]. Applications supply model integrations, executable handlers, and operational policy. This paper asks a narrow question: **for the retained fixed workflows, how did an experimental DAF integration compare with native Codex subagent coordination in observed completion time and scored correctness?** Current implementation contracts provide context only; they are not evaluated by these historical cohorts.

The contribution is a retrospective analysis of two retained benchmark cohorts, with explicit orchestration boundaries and validity constraints. It does not claim a novel scheduling algorithm. It does not introduce a new foundation model, demonstrate broad autonomous reasoning, or establish a causal benefit from DAF's communication protocol alone. The analysis was prepared from retained artifacts on 29 October 2026; model inference was not rerun during manuscript preparation.

2. System design and version boundaries

2.1 Architecture

The inspected public source is commit `123097cc01796bd9794e6095a7536ad9e47a6e09` of `darshjme/daf`, declaring package version 0.1.0 and Apache-2.0 licensing [1]. The workspace separates task types, graph execution, orchestration, an SDK, DDAL framing, transport, registry, runtime, storage, and command-line interfaces. This modularity permits applications to supply workers without treating a model provider as an intrinsic property of the framework.

Execution paths must be distinguished. The command-line mission scheduler executes explicit commands with dependency validation and bounded concurrency. The graph executor supplies graph-oriented behavior. The current specialist-plan path validates assignments against registered handlers, but executes mission phases sequentially. These paths are not interchangeable evidence that every DAF workload runs in parallel. The historical benchmark uses its own adapter around the graph executor.

DDAL frames messages and applies size and structural checks. Its checksum detects corruption; it does not authenticate a sender. The historical experiment uses local TCP transport. Current remote execution adds task authorization and TLS requirements outside loopback. Neither the local benchmark nor protocol framing establishes safe execution of arbitrary untrusted tools.

The memory interfaces support storage and retrieval, including Sled-backed persistence. The benchmark retrieves known stored results after reopening storage. This tests dependency persistence in that adapter, not semantic retrieval quality, embedding search, or learned memory. It is also distinct from a process-crash experiment.

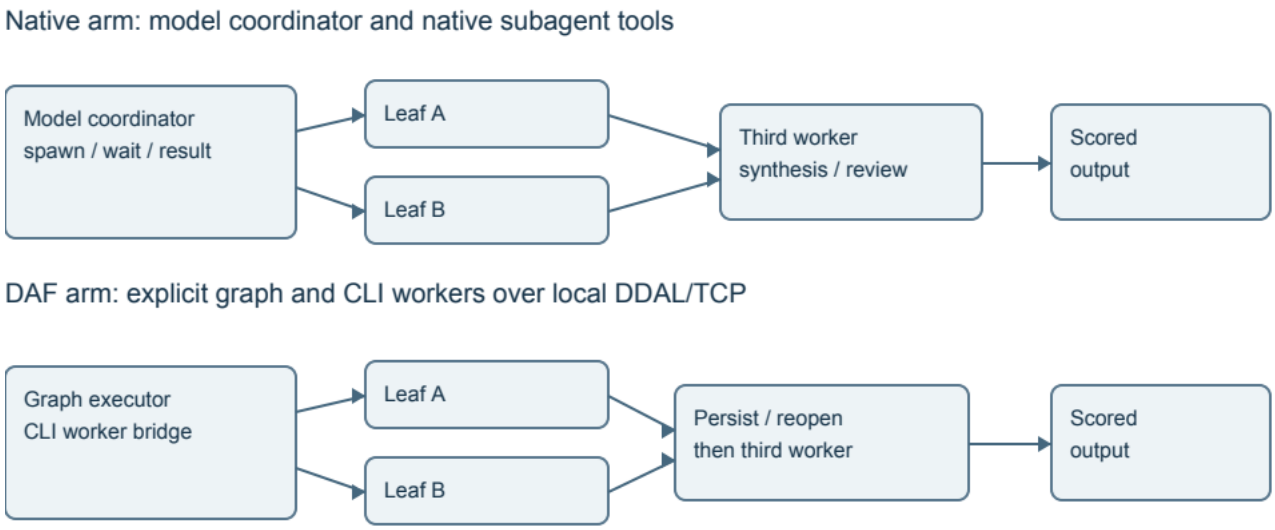


Figure 1. Both arms use two leaf tasks followed by synthesis or review. Native coordination is model-driven; the DAF arm uses an explicit graph and a separate CLI worker bridge. DAF persists and reopens dependency results. These are structurally different systems; the diagram does not isolate a causal scheduling or transport effect.

2.2 Components and execution responsibilities

The source architecture separates representation, scheduling, transport, and application work. Table 1 maps the principal responsibilities relevant to this study. This is an architectural map of the pinned current repository, not a claim that every module lies on the historical timing path. Infrastructure provisioning, configuration, logging, vault, and review modules also exist in the workspace; their independent performance is not evaluated here [1].

Layer	Responsibility	Boundary relevant to interpretation
Core and SDK	Shared task/result types and application handler integration	An application still supplies the actual work and model bridge.
Graph	Dependency representation, ready-node discovery, and concurrency-bounded wave planning	A graph describes eligibility; readiness is not proof of simultaneous inference.
Orchestrator and CLI	Execution coordination and user-facing command paths	Command missions, graph execution, and specialist plans have distinct behavior.
DDAL and transport	Message representation/framing and delivery	Protocol checks do not establish identity or eliminate model inference.
Registry and runtime	Worker lookup and execution integration	Authentication and durable handling have explicit current-path contracts.
Memory	Storage and retrieval of application information	Exact dependency lookup differs from semantic retrieval or learned memory.

Table 1. Architectural responsibilities at the pinned source revision. The benchmark uses a subset through a custom integration.

In the current graph code, `ExecutionGraph::ready_nodes` identifies nodes eligible under the graph's dependency state. `WaveScheduler` constructs waves from ready nodes and applies its maximum-concurrency bound; zero concurrency is rejected. A task with unmet prerequisites cannot become useful merely because spare worker capacity exists. The current graph executor waits for all tasks in its selected wave before planning the next wave. It is therefore not a general continuously dispatching scheduler that launches every newly ready descendant immediately: an unrelated long task in the same wave can delay a successor whose own predecessor has finished. For the benchmark fork/join topology, C requires both leaves, so that particular distinction does not change its dependency requirement. Graph planning makes the dependency structure explicit, while the executor and worker bridge determine how that plan becomes real work. These source-level properties explain the intended execution mechanism; they are not fresh runtime tests [1].

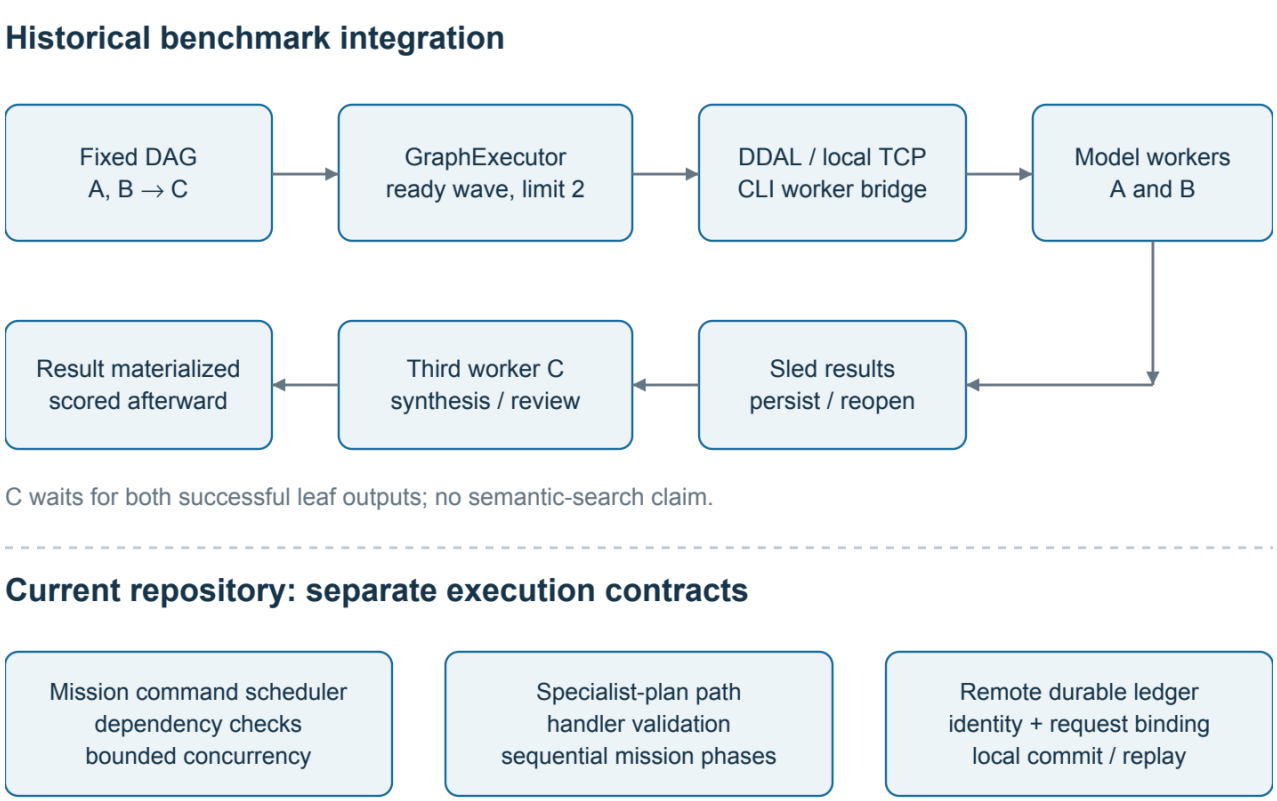


Figure 2. Architectural layers and the historical benchmark boundary. The benchmark adapter connects graph execution, local DDAL/TCP workers, CLI inference, and Sled dependency retrieval. Current remote authorization and the durable ledger are separate context, not components whose latency was measured in the historical cohorts.

2.3 Current durable-result contract (context only)

The current remote ledger binds a stable task identifier to an authenticated principal and canonical request. A handler prepares an output and declared immutable ledger effects. The implementation commits those effects and the result in one Sled transaction and flushes before acknowledgement. A matching retry can return the stored result; a changed principal or request conflicts. Successful same-principal predecessor outputs can be supplied to dependent tasks [4]. Source inspection confirmed the binding, transaction, and flush operations in `crates/daf-runtime/src/durable.rs`.

This contract has a deliberately narrow boundary. Preparation can repeat before commit and must avoid external effects. An email, payment, shell command, or external database update is not made exactly-once by the local ledger. The ledger serializes preparation and uses one local database; it is not a replicated fleet scheduler. Repository documentation reports a subprocess SIGKILL test at the commit-before-reply boundary, followed by restart and cached replay. That report supports the tested process-crash scenario, not universal power-loss durability [4]. These current features were not measured by the earlier latency cohorts.

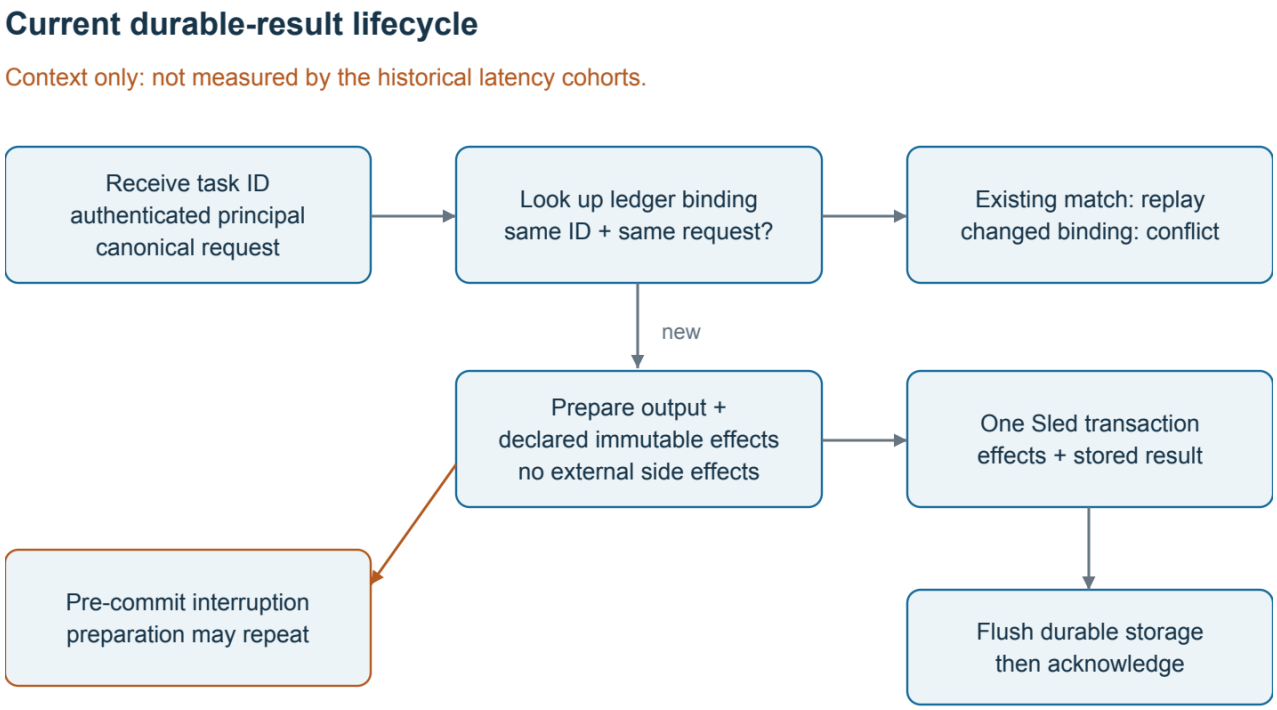


Figure 3. Current local durable-result lifecycle. A matching committed request can replay its result without running the preparation handler again. Before commitment, preparation may repeat after interruption. Only the declared local ledger effects and result share the transaction; arbitrary external effects remain outside it. This figure explains a current contract, not a measured source of the historical speedup.

A useful failure distinction is the position of an interruption relative to commitment. Before commitment, no successful durable receipt is available, so a later attempt can prepare again. After commitment but before the response reaches the caller, the caller may be uncertain even though a result exists; a matching retry can recover that result. Binding the task identifier to the principal and canonical request prevents a retry identifier from silently being reused for different work. The transport supplies an authenticated principal: the ledger's acceptance of a principal string is not itself an authentication mechanism [4].

Durability and low latency are separate goals. Persistence adds serialization, transaction, and flush work on the successful path. Replay can avoid repeated preparation for an already committed task, but no replay-speed experiment is included here. Likewise, the serialized preparation boundary limits concurrency in this particular ledger path. It must not be conflated with the parallel leaves in the historical graph adapter.

2.4 Historical implementation identity

The retained benchmark lineage begins from a recorded baseline `e9f3feb52c43b88ab00982db27087847b4fb3eb2`, with preserved local changes, isolated repairs, and benchmark-specific adapters [2,3]. A commit identifier alone therefore does not fully identify the tested source. The frozen manifests, repaired source, adapter, and harness are necessary parts of the experimental object. The historical timings must not be presented as performance measurements of current main.

3. Evaluation method

3.1 Design and metrics

The two cohorts compare native Codex subagent coordination with experimental DAF orchestration. Both record the model configuration as GPT-6 Astra at medium reasoning effort. This identifies the recorded configuration, not an immutable provider-side model snapshot. Each workflow contains two leaf tasks followed by a synthesis or review task. Arm order alternates within the retained sequence. This balances ordering descriptively but is not randomized assignment.

For pair i, define the latency ratio as r_i = T_native,i / T_DAF,i. We report the median of the paired ratios, separately from each arm's median time. These quantities need not be equal to the ratio of arm medians. All attempted scored runs in each valid cohort are included. The ledger cohort comprises four fixture seeds with two repeats each; these eight observations are not eight distinct task families. The coding cohort comprises two pairs on the same task family. We report no population confidence interval or statistical-significance claim.

Correctness and execution validity are separate from latency. Ledger scoring requires exact task outputs and valid execution evidence. Coding scoring combines configuration and execution checks with a behavioral verifier. Passing these checks supports the tested cases; it does not prove general program correctness or the semantic quality of review prose.

3.2 Synthetic ledger cohort

The ledger fixtures require latest-version selection, approval filtering, signed totals, weighted synthesis, and exact source-nonce preservation [2]. DAF uses GraphExecutor -> DDAL/TCP -> CLI model inference -> stored Sled result -> reopened dependency retrieval -> synthesis. The native arm uses actual subagent spawn, wait, and result tools under a model coordinator. Both permit at most two concurrent leaf workers.

The timing includes coordinator or process startup, inference, and result handling, and excludes builds. The workflow deadline is 900 seconds; DAF additionally has a 240-second per-CLI guard without an exact native counterpart. The retained method permits no normal retries. Engineering pilots were excluded before the frozen scored batch; all eight scored pairs are reported. These differences constrain any timeout comparison. Native trace metadata supports child identity, model configuration, completion, and dependency timing, but encrypted child message content prevents verification of byte-identical native worker prompts. Matching tasks and nominal model configuration does not match internal system context.

3.3 Corrected coding cohort

The coding tasks implement `slugify` and `parse_duration`, followed by a third worker reviewing their outputs [3]. Workers operate read-only without tools. The corrected method specifies the same JSON shape in both arms and validates it externally; DAF's CLI schema enforcement is disabled. Reviewers receive the full dependency specifications in both arms. Timings include startup and result materialization and exclude setup, builds, trace extraction, and behavioral testing equally.

An earlier coding batch was invalid because output-shape requirements were asymmetrical: the native prompt omitted a requirement while the DAF arm enforced a schema. That batch is excluded in full. Its failures are harness failures, not evidence of weaker native coding ability. The retained predeclared method specified two alternating pairs, no automatic retries, and no optional stopping; all four scored attempts are included. The overall workflow budget is 900 seconds with a 930-second outer guard; DAF also has a 300-second child guard. A separate historical development timing is excluded.

3.4 What the measured wall time includes

The unit of analysis is a completed workflow, not a single protocol message or isolated model request. A run starts at its harness-defined coordination or process launch boundary and ends when the workflow result is materialized. Worker inference, waits, launch overhead, dependency delivery, and result handling therefore contribute to the reported interval. Build and offline scoring time are excluded. A faster total can arise from several changes in the path; the total alone cannot identify a faster component.

The common topology is A and B followed by C. For ledger work, A and B produce independently computed ledger results and C performs dependent synthesis. For coding work, A and B implement the two specified functions and C reviews their outputs. C cannot legitimately finish its assigned dependency-aware work before both required results are available. The native coordinator is allowed to use two workers, while the DAF adapter encodes the dependency structure directly. The comparison concerns the realized full paths, including those coordination choices.

The retained records are sufficient to reconstruct each pair's elapsed-time ratio and saved outcome flags. They do not provide a calibrated, mutually exclusive decomposition of every elapsed second into provider queueing, inference, coordinator reasoning, transport, storage, and local processing. Consequently, this paper does not draw stacked component-time bars or assign a percentage of the improvement to any one subsystem.

4. Results

4.1 Ledger workflows

The retained scorer marked all eight workflows in each arm configuration-valid and correct. Table 2 reports all paired times from the retained machine-readable results, rounded for presentation.

Fixture / repeat	Native seconds	DAF seconds	Native / DAF
201 / 1	70.56	18.55	3.80
201 / 2	69.53	16.95	4.10
202 / 1	65.66	17.86	3.68
202 / 2	64.43	18.36	3.51
203 / 1	62.79	17.40	3.61
203 / 2	67.47	18.62	3.62
204 / 1	62.84	16.88	3.72
204 / 2	65.24	18.43	3.54

Table 2. Historical ledger cohort: eight pairs, four unique fixtures, correctness 8/8 per arm.

The median paired ratio is 3.65, with a descriptive range of 3.51–4.10. Arm median times are 65.45 seconds for native and 18.11 seconds for DAF. Recorded leaf intervals overlap in 0/8 native runs and 8/8 DAF runs. Native intervals use model-task timestamps, while DAF intervals use handler timestamps. Overlap is therefore descriptive, not a calibrated cross-system utilization metric or evidence of GPU-level parallelism. The native arm was permitted two concurrent leaves, so its observed sequential scheduling was retained as an outcome rather than discarded.

The historical method also describes separate topology-sensitivity and delivery-failure diagnostics. These are outside the two primary cohorts and are not included in the companion run-record dataset or pooled into the reported ratios. A controlled topology ablation remains future work.

4.2 Corrected coding workflows

Pair	Native seconds	DAF seconds	Native / DAF	Workflow result
1	104.02	40.01	2.60	Both pass
2	91.35	38.48	2.37	Both pass

Table 3. Corrected coding cohort: two pairs on one task family. The retained evaluation records 187 passing behavioral checks for each workflow. These are the same 187 cases reused across four workflows, not independent benchmark tasks.

The median paired ratio is 2.49. Arm median times are 97.68 seconds for native and 39.25 seconds for DAF. Both arms exhibit overlapping leaf intervals in both runs. Consequently, the presence or absence of leaf overlap alone cannot explain this cohort's latency difference. Coordinator inference, process and context overhead, cache behavior, and other service conditions remain plausible contributors. The design does not isolate their individual effects.

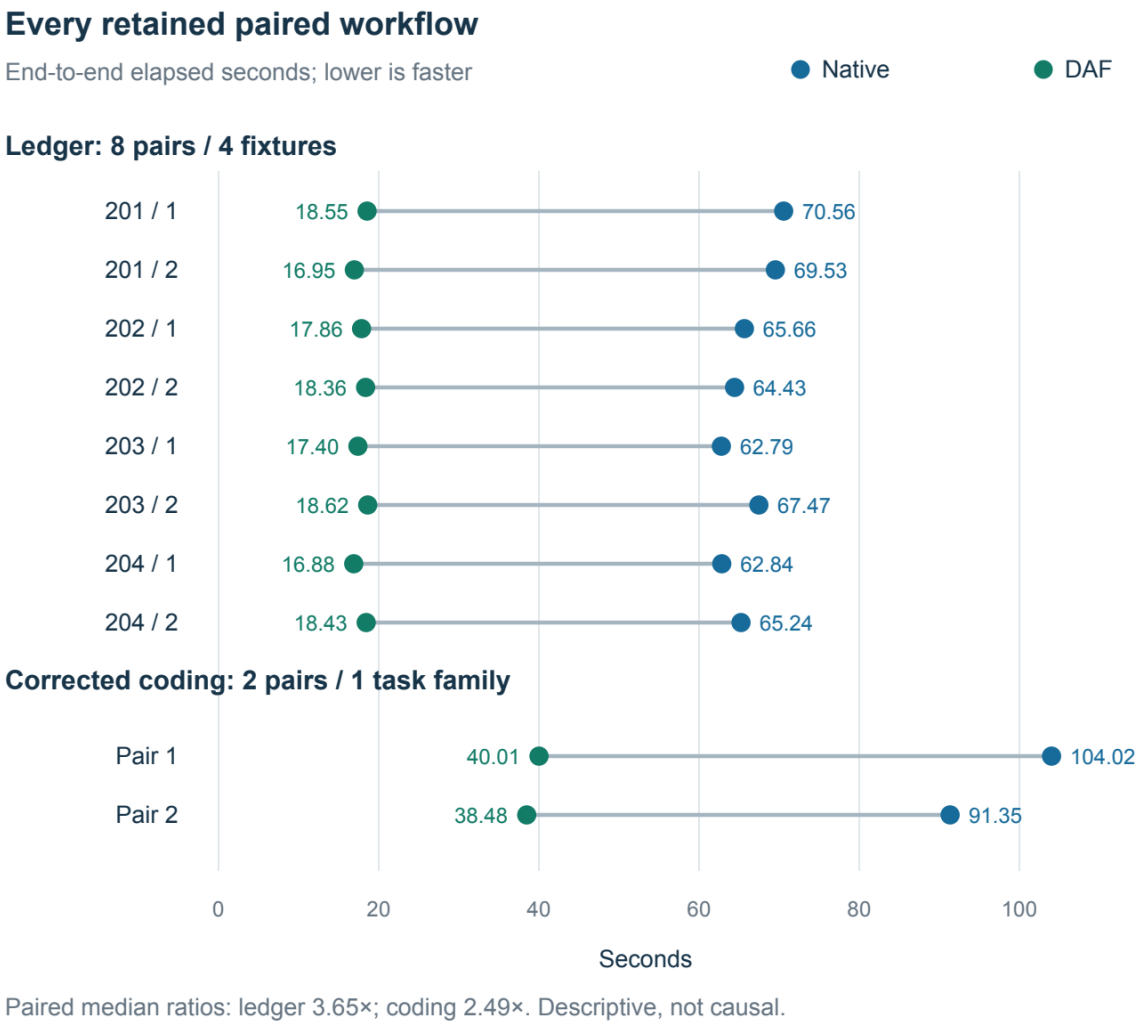


Figure 4. All eight ledger pairs and both corrected coding pairs, plotted separately from the retained numerical evidence. Each pair compares complete workflow wall times in seconds. The two cohorts are not pooled into a cross-task score. Repeats share fixtures, and the coding workflows reuse the same 187 behavioral cases. The chart shows measured totals, not component timings or uncertainty estimates.

4.3 Usage accounting

The historical usage records separate DAF attempts, native coordinators, and native children. Parent inclusion of child usage is not established, so the native scopes cannot be added into a reliable total. Cached input is a subset of input, and no invoice establishes billed dollar cost. Usage events are not part of the curated numerical release. We make no total billed-token or financial-saving claim.

5. Explaining the observed latency differences

5.1 A critical-path model

A fixed workflow can move scheduling decisions from a language-model conversation into explicit program state. Consider two independent leaves A and B and a dependent synthesis C. Let L_A and L_B be leaf execution durations under a hypothetical common worker environment, L_C the synthesis duration, and H the remaining non-overlapping overhead along the workflow's critical path. These variables are analytical quantities, not measured phase estimates from this release.

With serial leaves, an idealized model is:

T_serial = H_serial + L_A + L_B + L_C.

With simultaneous independent leaves and sufficient capacity, the corresponding idealized model is:

T_parallel = H_parallel + max(L_A, L_B) + L_C.

If leaf durations and overhead were otherwise identical, the avoided leaf wait would be `min(L_A, L_B)`. If `H_serial = H_parallel >= 0` and durations are nonnegative, the resulting ratio cannot exceed 2 for this two-leaf pattern, and synthesis or other serial work reduces it below that limit. This bound applies to the stated idealized assumptions, not arbitrary agent systems. The observed ledger median of 3.65 is larger than this overlap-only bound. It therefore cannot be explained by merely overlapping two unchanged leaves while leaving all other durations unchanged. Additional path differences or changed effective durations are necessary; the data do not identify their separate contributions.

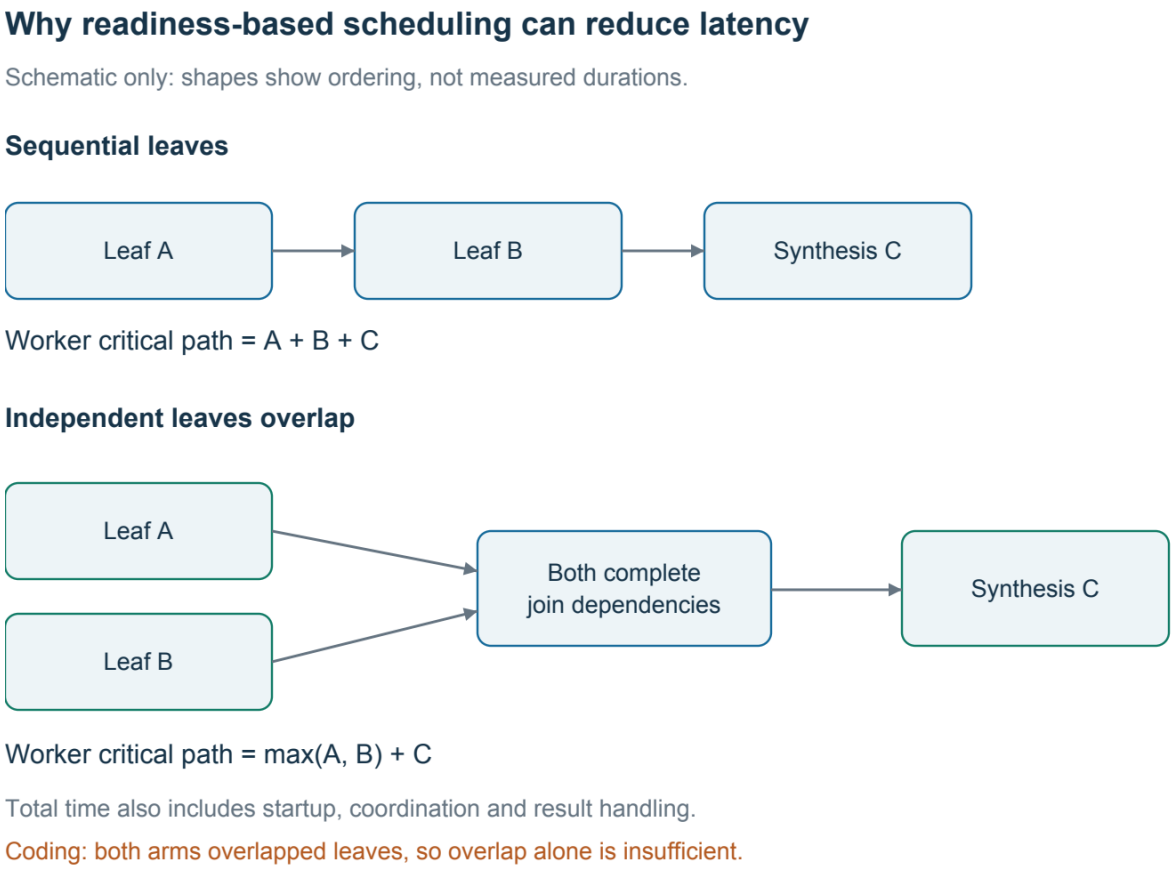


Figure 5. Conceptual serial and parallel execution for the same two-leaf dependency graph. Task boxes and spacing are schematic, not fitted to the benchmark. Overlap changes a sum of leaf durations to their maximum only under the stated idealized assumptions. Coordinator work and other overhead may differ between the measured systems; no numerical phase attribution is implied.

This model is deliberately more restrictive than the real comparison. Provider scheduling, different model contexts, launch costs, or coordinator actions can change the effective leaf or synthesis durations. Some overhead can itself overlap with inference, so blindly adding all recorded event durations would double-count time. A valid measured decomposition would require synchronized boundaries and a method for accounting for overlap.

5.2 Mechanisms that could make a fixed graph faster

Making eligible work explicit. A dependency graph already states that A and B are independent and C depends on both. The scheduler can release eligible leaves without asking a model to rediscover this relation. In the retained ledger traces, native leaf intervals did not overlap while DAF leaf intervals did. This is consistent with a scheduling contribution. The timestamp sources differ, however, and an observed interval overlap does not establish simultaneous execution on the model provider's hardware.

Reducing coordination on the inference path. A model-driven coordinator may need inference steps to decide when to spawn, inspect, wait, and assemble results. An explicit scheduler can implement predetermined transitions in ordinary program logic. This is a plausible advantage when the task graph is already known and those model decisions add no needed adaptation. The experiment does not isolate coordinator inference time or establish that native coordination always uses redundant decisions. A native implementation could also be instructed or engineered to realize the same fixed topology.

Passing dependencies through an explicit result path. The historical adapter stores leaf outputs and retrieves them for the dependent task, including reopening Sled. This gives the dependency transfer a defined interface instead of relying solely on a conversational coordinator to relay results. It may change context construction and local handling overhead. The release does not show that storage is intrinsically faster than

conversational transfer; persistence also adds work, and prompts and internal context are not byte-identical across arms.

Constraining the application rather than improving the model. These workflows have prescribed leaves, a prescribed joining task, and narrow output checks. A fixed graph can exploit that structure. Neither DDAL nor Rust causes the underlying model to reason faster by itself. No matched protocol microbenchmark is used to explain the end-to-end ratios, and no evidence shows that model weights, token-generation throughput, or reasoning quality improved. The engineering hypothesis is that a more constrained control path can spend less time coordinating this known work.

5.3 Why the coding result needs a different explanation

Both coding arms show overlapping leaf intervals in both pairs, yet DAF still has lower total elapsed time. The binary fact of leaf overlap therefore cannot explain the coding difference. Possible sources include the amount of coordinator inference before or after the leaves, different launch and worker-context paths, service queueing, cache state, and the timing of result materialization. A trace showing overlap alone cannot measure these effects. It also cannot show that the leaf overlap was equal in duration or that leaf execution costs were identical.

The corrected schema procedure improves the fairness of the output contract: both arms receive the same required JSON shape, and external validation applies to both. It does not equate hidden system prompts, provider routing, model snapshots, or complete execution contexts. The proper reading is that the historical integrated DAF path was faster in these two retained pairs at tied saved behavioral outcomes, with its causal mechanism still unresolved.

5.4 Evidence strength and non-claims

Statement	Evidence class	Interpretation
DAF has lower elapsed time in all ten retained pairs	Archived measurement	True of these two cohorts; not a population guarantee.
Ledger intervals overlap only in DAF; coding intervals overlap in both arms	Archived execution fields	Descriptive topology evidence with different timestamp boundaries.
Current graph code has dependency readiness and bounded wave planning	Inspected source	An implementation property, not a benchmark of current main.
Explicit scheduling may avoid model coordination steps	Mechanistic hypothesis	Requires matched-path timing or ablation to quantify.
DDAL or Rust caused the reported ratios	Not established	No isolated causal comparison supports this statement.
DAF provides better reasoning, lower billed cost, or general production speedup	Not established	Outside the tested outcomes and available accounting.

Table 4. Claim-to-evidence map. Source properties, archived observations, and causal hypotheses answer different questions.

5.5 When the advantage may shrink or reverse

The critical-path model predicts little benefit from parallel scheduling when all tasks depend sequentially on earlier outputs, when one leaf dominates the other, or when a long synthesis dominates the workflow. It also predicts less benefit when the baseline already overlaps eligible work and spends little time on coordination. These are deductions from the model, not measured negative results.

At high concurrency, provider limits and shared resources may turn more local concurrency into queueing or failures rather than faster completion. Persistent result handling, process launch, framing, validation, and flush operations have costs. For very short tasks those costs can dominate; for long inference they may be comparatively small. Open-ended tasks can benefit from adaptive decomposition that a predetermined graph does not supply. Choosing a fixed graph is thus a workload decision, not a universal replacement for model-driven coordination.

6. Related work

AutoGen describes a framework for applications built from conversable agents that can integrate models, tools, and human participation [5]. ReAct studies interleaved reasoning and actions within language-model trajectories [6]. These address different abstraction levels from this paper's fixed-workflow scheduling comparison. We do not benchmark DAF against either system and make no superiority claim over them.

LangGraph's documented persistence mechanisms distinguish graph-state checkpoints from stores for information shared across runs [7]. DAF likewise requires care in distinguishing stored application data from execution recovery, but the mechanisms and guarantees must be compared at their actual boundaries. The DAF ledger contract discussed here covers declared local effects and receipts. This paper does not establish that the contract is novel relative to the broader durable-workflow literature. Its contribution is the implementation case study and its bounded evidence.

7. Discussion and threats to validity

The observations are consistent with the hypothesis that an explicit scheduler can reduce overhead for a known dependency graph. They do not isolate scheduling as the cause: the arms also differ in coordinator inference, worker context, integration layers, and potentially cache and service conditions. The differing topologies motivate a controlled ablation rather than a causal interpretation of the observed ratios. A follow-up should use an identical worker bridge in both arms, vary only scheduler behavior, and separately measure startup, scheduling, inference, serialization, and persistence.

External validity is limited by four synthetic ledger fixtures and one small coding task family. Neither cohort represents long-horizon repository work, interactive debugging, unfamiliar APIs, or production failures. Repeated fixtures can exercise cache effects and are not independent samples from a broad task population. Nominal model aliases can change remotely; this draft cannot reconstruct provider-side weights from historical logs.

Instrumentation also limits interpretation. Native child prompt fidelity cannot be inspected byte-for-byte. Usage scopes are not equivalent billing records. Deadline asymmetry would matter if exercised. Coding review prose lacks a semantic oracle. Exact ledger correctness and 187 behavioral cases are useful local tests, but neither is a universal measure of reasoning.

Artifact recovery is incomplete. The companion package contains curated historical run records, method transcriptions, paired measurements, provenance hashes and an offline analysis script. It supports checking the reported arithmetic and consistency of saved score and execution fields. It does not contain complete prompts, generated artifacts, private native transcripts, historical binaries, or the entire repaired source. It therefore does not support independent rescoring or a complete inference rerun. The separate September v2 evaluation is excluded. The public source revision is context, not a substitute for the historical adapter. Historical hardware details and an immutable model snapshot are not established by this package; later repository-validation environments are not substituted for them.

7.1 A controlled follow-up design

A stronger study should retain the same worker bridge, model configuration, task inputs, output contract, and scoring procedure while changing one coordination variable at a time. The present results motivate the following measurements; none has been completed as part of this release.

Question	Controlled comparison	Measurement needed
How much comes from leaf overlap?	Serial and parallel schedules using the same worker bridge	Common launch/completion boundaries and full critical-path time.
How much comes from model coordination?	Fixed program transitions and model-directed transitions with matched workers	Coordinator calls and exclusive elapsed intervals.
How much comes from dependency handling?	Matched inputs with and without persistent result storage	Serialization, write, reopen, read, and dependent-launch intervals.
Does transport matter?	Same workload and framing over matched delivery paths	Message sizes, local processing, delivery time, and total workflow time.
How stable is the result?	More independent task families with randomized or counterbalanced order	Per-run environment, cache indicators, queueing indicators, and failures.
What happens on interruption?	Predeclared crash and retry points under the current durable contract	Duplicate preparation, receipts, externally visible effects, and recovery latency.

Table 5. Proposed ablations and instrumentation. This table is a research plan, not additional empirical evidence.

A practical sequence is to freeze and instrument a common execution path first, run the scheduler and dependency-handling ablations second, and only then expand to independent realistic task families and interruption experiments. This order helps avoid interpreting a broad integrated comparison before the narrower mechanisms are measurable.

Before executing such a study, the source tree, adapter changes, dependency lockfiles, worker bridge, fixture generator, and scorer should be frozen together. Complete de-identified inputs and generated outputs would permit independent rescoring. Reporting failed attempts and timeout policy alongside successful runtimes would prevent a speed comparison from hiding reliability tradeoffs. Repeated fixtures should be analyzed as repeated measurements, with a sufficiently broad independently sampled task set before population-level inference. Billing comparisons additionally require non-overlapping usage scopes and actual price or invoice data.

8. Conclusion

In the retained ledger and corrected coding cohorts, the experimental DAF integration had lower observed completion time while tying scored correctness. The median paired ratios were 3.65 and 2.49 respectively. The evidence supports these fixed-workflow observations and motivates controlled scheduler ablations; it does not establish better reasoning, isolated protocol acceleration, or lower billed cost. DAF's current code additionally exposes explicit execution and durable-result contracts whose scope can be inspected independently of those historical performance measurements.

Availability and research status

Current source is public at the pinned repository revision [1]. The companion archive provides `evidence/benchmark-summary.json`, `evidence/run-records.json`, historical method transcriptions, `evidence/provenance.json`, and `verify.py`. The run records retain whitelisted timing and scorer fields from all 20 scored workflows. Private paths, commands, session identifiers and raw model events are omitted. Original-file SHA256 digests identify the inputs to curation; `SHA256SUMS` identifies the distributed files. These are archival records, not new experiments or independently verified ground truth. See the package README for exact coverage and omissions. Version 1.0 is archived at doi:10.5281/zenodo.23755124. This version expands the architecture, critical-path explanation, diagrams, and follow-up design; it adds no benchmark runs and does not change the historical measurements. The DOI for version 1.1 is 10.5281/zenodo.23772808. This preprint has not undergone peer review.

The author develops DAF and conducted the original evaluation, creating an allegiance risk in design and interpretation. All scored pairs from the two valid cohorts are reported, while excluded engineering pilots and the invalid coding batch are disclosed. The manuscript and curated evidence package are offered under CC BY 4.0. The separately linked DAF repository retains its existing software license. No institutional affiliation or endorsement is asserted.

References

1. Joshi, D. **Darshj's Agent Framework**, source revision `123097cc01796bd9794e6095a7536ad9e47a6e09`. Inspected 29 October 2026. Pinned source.

2. Joshi, D. **DAF versus native Codex: end-to-end model benchmark.** Retained historical experiment: `e2e/METHOD.md`, `FROZEN.json`, `RESULTS.json`, `REPORT.md`, and scored runs. Curated companion evidence: `evidence/ledger-method.md`, `evidence/run-records.json`, and `evidence/benchmark-summary.json`; original digests in `evidence/provenance.json`.

3. Joshi, D. **Corrected coding comparison.** Retained historical experiment: `coding-comparison/corrected/METHOD.md`, `FROZEN.json`, `RESULTS.json`, `SUMMARY.json`, and runs. Curated companion evidence: `evidence/coding-method.md`, `evidence/run-records.json`, and `evidence/benchmark-summary.json`; original digests in `evidence/provenance.json`.

4. Joshi, D. **Remote execution milestone and authenticated durable remote execution**, repository documentation. Milestone; contract.

5. Wu, Q., et al. **AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation.** arXiv:2308.08155, 2023. Paper.

6. Yao, S., et al. **ReAct: Synergizing Reasoning and Acting in Language Models.** ICLR 2023; arXiv:2210.03629. Paper.

7. LangChain. **LangGraph persistence documentation.** Accessed 29 October 2026. Documentation.