Substrate-Portable Execution for Production LLM Workflows

Tarun Gopinath
Amazon
Seattle, WA, USA
tarungop@amazon.com

Atul Kulkarni
Amazon
Seattle, WA, USA
atukulka@amazon.com

Shrikar Katti
Amazon
Seattle, WA, USA
shrikar@amazon.com

Vijay Rajakumar
Amazon
Seattle, WA, USA
vijar@amazon.com

Parthasarathy Govindarajen
Amazon
Seattle, WA, USA
partgovi@amazon.com

Abstract

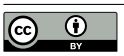
Production LLM agents often run the same logic—including tool-calling loops, retrieval chains, and compositional workflows—in several operational modes. However, execution semantics are usually coupled to one runtime. We encountered this portability problem in Rufus, a conversational AI assistant that integrates a large catalog of tool endpoints and serves millions of Amazon customers [6]. Systems such as Rufus use real-time serving, asynchronous background execution, and high-volume offline batch execution, each with different service-level objectives. Reusing orchestration designed for streaming can force asynchronous and batch workloads into a blocking execution model and prevent access to batch inference APIs.

We present a binding-adaptive agent execution platform that separates workflow definition from execution substrate. Developers express a workflow once as a typed dataflow graph that can be bound to Local, AWS SWF [3], or Apache Flink [18] execution. A key design is the first-class inference abstraction, which allows each binding to use the inference mechanism best suited to its operational profile. We evaluate streaming inference with the Local binding, synchronous inference within SWF's asynchronous durable workflow, and batch inference with the Flink binding—the configurations in which these inference mechanisms provide the most practical value.

We evaluate five orchestration patterns covering single inference, iteration, routing, composition, and parallel multi-agent execution. For the tested workloads, we find no detectable systematic output-quality preference among bindings. Our experience also identifies the portability boundary: latency, time-dependent tool results, and recovery behavior remain substrate-specific.

CCS Concepts

• **Software and its engineering → Middleware**; *Domain specific languages*; *Distributed systems organizing principles.*



This work is licensed under a Creative Commons Attribution 4.0 International License.
Middleware Industry Track '26, Tarragona, Spain
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2620-0/2026/12
https://doi.org/10.1145/3801920.3861029

Keywords

LLM agents, workflow middleware, execution-mode portability, domain-specific languages, batch inference, distributed workflow systems, agent evaluation

ACM Reference Format:
Tarun Gopinath, Atul Kulkarni, Shrikar Katti, Vijay Rajakumar, and Parthasarathy Govindarajen. 2026. Substrate-Portable Execution for Production LLM Workflows. In *27th International Middleware Conference Industry Track (Middleware Industry Track '26), December 14–18, 2026, Tarragona, Spain.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3801920.3861029

1 Introduction

Production agents often reuse the same prompts, tools, and control flow across workloads with different operational requirements. Interactive serving prioritizes time to first token because results are delivered directly to users. Background inference tasks can tolerate longer latency but need durable progress. Offline workloads, such as LLM-as-a-judge evaluations and LLM-based content generation, prioritize throughput and cost. Our platform addresses this mismatch by allowing a developer to define one typed workflow graph that a binder compiles to a runtime selected for the required operational mode. We developed the platform for Rufus, a conversational assistant used by millions of Amazon customers [6], whose agents combine Model Context Protocol (MCP) [12] tools with routing, RAG, ReAct [36], PreAct [20], and multi-agent orchestration.

Production conversational AI systems commonly use three execution modes. The real-time path optimizes for time to first token and supports patterns such as tool-calling loops, RAG, and speculative execution. The asynchronous path handles background tasks that tolerate seconds or minutes of latency, including deep-research pipelines and offline enrichment. The batch path serves high-volume offline workloads such as prompt evaluation, quality regression testing, and content pre-generation. Each mode has distinct SLOs and typically runs on a different runtime: a streaming service, a durable workflow engine, or a stream-processing or job-scheduling system.

The runtimes are heterogeneous even when agent logic is shared. Hiding inference inside ordinary functions forces each runtime to inherit the same invocation mechanism; in particular, independent offline invocations cannot be aggregated by a shared batcher. The architecture must therefore separate orchestration from inference submission and make suspension points visible to the runtime.

This paper makes three contributions. First, it identifies execution-mode portability as a middleware problem for production

LLM workflows. Coupling orchestration and inference execution to one runtime prevents the same workflow from adapting efficiently to real-time, asynchronous, and batch workloads. Second, it presents a typed dataflow DSL, first-class inference abstraction, and binding compiler that separate workflow logic from execution substrate. Together, these components allow each binding to select its inference strategy—including cross-invocation batch inference and access to its lower published pricing—without changing workflow code. Third, it evaluates the proposal across five representative orchestration patterns and three execution configurations, measuring cross-binding output quality and completion time. Drawing on our experience operating Rufus, we also discuss operational tradeoffs and identify practical limits to portability.

Table 1 illustrates the range of execution requirements across representative inference workloads.

Table 1: Representative Rufus inference workloads. Agent definitions can be reused across execution modes with different requirements. The "Target Binding" column shows the target substrate in the new platform.

Workload	Infer/inv	Latency	Target Binding
Real-time serving	1–5	Sub-second TTFT	Local
Evaluation	1–5	Hours	Flink
Regression evaluation	1–5	Hours	Flink
Cache generation	1–3	Hours	Flink
Deep research	5–12	Minutes	SWF

2 System Design

Design challenge. When asynchronous and batch workloads reuse real-time orchestration, they inherit its blocking, synchronous execution model. Replacing this model with asynchronous batch submission would require changing the shared orchestration loop needed for progressive token streaming. Independent invocations also cannot aggregate inference requests into a common batch. Per-workflow systems such as Temporal [33] and Step Functions [4], and agent frameworks such as LangGraph [23] and Strands [7], treat each LLM call as an opaque blocking operation without substrate awareness. These constraints have broader operational consequences. When offline quality evaluation reuses the real-time execution path, it inherits infrastructure optimized for time to first token rather than throughput. More generally, features developed for one execution mode cannot be reused in another without reimplementing their orchestration.

2.1 Key Problems and Architecture

The design addresses two key problems. First, workflow logic must be decoupled from substrate selection so that one definition can execute in different operational modes. Second, the binding compiler must control the inference strategy because it varies across substrates. If inference remains hidden inside an opaque function call, every substrate inherits the same blocking execution strategy.

Figure 1 shows the resulting architecture. Agent logic written in Java or JavaScript produces a binding-agnostic workflow graph. The binder compiles the graph into substrate-native operations for Local, SWF, or Flink execution. A shared control plane manages agent configuration, registration, and lifecycle across all bindings.

2.2 Typed Dataflow DSL

Agent workflows are typed dataflow graphs. The platform provides two ways to construct the same graph representation. Developers can use a type-safe Java combinator API for production workflows, allowing composition errors to be detected when the graph is defined. For rapid prototyping, developers can use a hot-reloadable JavaScript DSL on GraalVM. In the Java API, each workflow carries its input and output types as `Workflow<I,O>`. Each combinator consumes the preceding output type and produces a new workflow output type. For example, `map(Function<O,N>)` transforms `Workflow<I,O>` into `Workflow<I,N>`. The Java compiler therefore detects incompatible composition and branch-merge operations when the workflow is defined. These API calls construct an immutable graph rather than executing the workflow. The JavaScript DSL constructs the same graph, which is subsequently translated by the selected binding.

Listing 1 illustrates a ReAct workflow constructed through the typed Java API.

Listing 1: A ReAct agent defined via the typed Java API. The JavaScript DSL produces the identical graph.

```
Workflow<AgentInput, AgentOutput> wf =
  Workflow.source("react-agent")
    .map(InitState())
    .iterate(ShouldContinue(), maxIter,
      step -> step
        .infer(ReasoningPrompt(), config)
        .map(ParseResponse())
        .map(ConditionalToolExec())
    )
    .map(BuildOutput());
```

Table 2 summarizes the DSL's combinators, their semantics, and their typical uses.

Table 2: Workflow combinators and their intended uses.

Operator	Semantics	Typical use
`map`	Applies a typed transformation; annotated tool steps may be offloaded.	Prompt assembly, parsing, tool dispatch
`infer`	Invokes an LLM using the strategy selected by the binding.	Model inference
`iterate`	Repeats a body until a condition or iteration limit is reached.	ReAct loops
`compose`	Delegates execution to another registered agent.	Multi-agent workflows, PreAct
`branch`, `parallelBranch`	Performs conditional or parallel fan-out followed by a typed merge.	Routing, specialist fan-out

The small combinator set also expresses more complex patterns. Listing 2 shows speculative PreAct: a low-cost model plans tool use before delegating answer composition to a stronger ReAct agent.

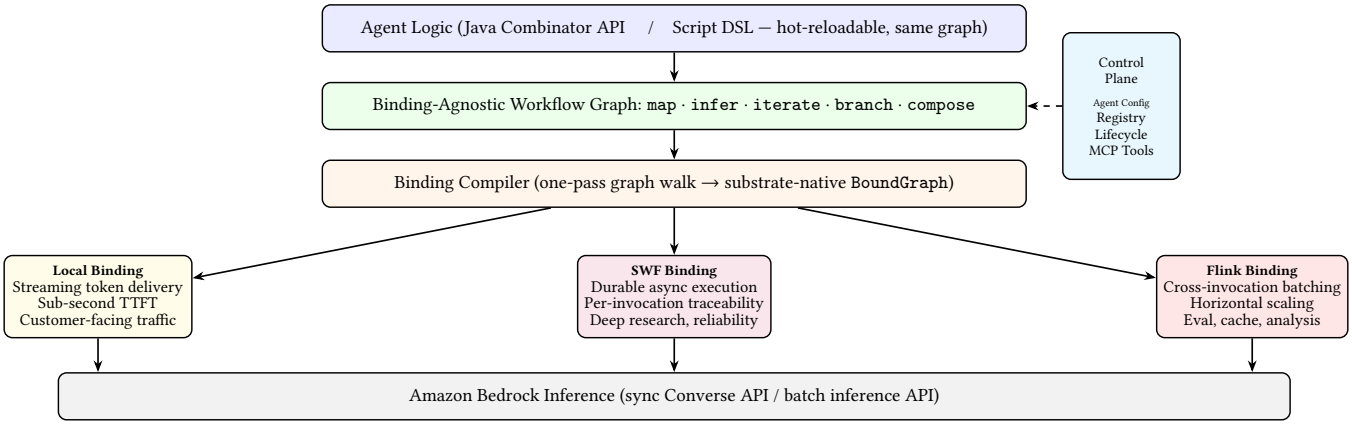


Figure 1: Platform architecture. Agent logic (top) is authored via the typed Java API or the script DSL; both produce the same binding-agnostic workflow graph. The binding compiler translates the graph to substrate-native operations. In the evaluated configurations, Local targets streaming live traffic, SWF provides durable asynchronous execution, and Flink supports scalable offline batch operations. All bindings share the same Bedrock inference layer.

Listing 2: PreAct composition in the JavaScript DSL. The same graph executes on every binding.

```
workflow.source(Source("preact-compose"))
  .map(InitState())
  .infer(PlannerPrompt(),
    StaticInferenceConfig({modelId:"haiku45"}))
  .map(ParseResponse())
  .map(ToolExecution())
  .map(HandoffPrompt())
  .map(BuildOutput())
  .map(OutputAsContext())
  .compose(Agent("react-agent"));
```

This example exercises `infer`, `map`, and `compose`. Restricting the interface preserves a consistent workflow structure across bindings. Ordinary steps transform immutable state passed through the graph, while operations that affect state outside the workflow are isolated in identifiable tool steps that each binding can execute or offload. This design keeps the orchestration portable, while workflows whose behavior depends on external events, substrate-specific failure handling, or the timing of streamed tokens remain tied to a particular execution environment.

2.3 Why Inference is a First-Class Primitive

Many agent frameworks represent an LLM call like any other function call. The framework sees an operation such as `invoke(fn)` and blocks until it returns, imposing one execution strategy on every workload. Even when the call is made asynchronously, the workflow implementation still determines how the inference request is submitted, leaving the operation opaque to the execution substrate.

Representing inference as a graph node, through `.infer(...)` rather than an opaque call inside `.map(...)`, enables binding-specific compilation. This is important for three reasons. First, inference dominates workflow cost and latency, so its execution strategy largely determines the workflow's profile. Second, it may produce a streaming response or be submitted asynchronously and correlated by identifier. Third, compatible requests from independent invocations can be aggregated for batch submission. If inference were hidden inside a general-purpose function, the compiler would have to preserve the execution strategy selected by the workflow author. Making inference first-class exposes its configuration, request boundary, and workflow continuation to the binding compiler, allowing the binding to select an inference mechanism appropriate for its workload.

2.4 Binding Compilation

A *Binder* traverses the workflow DAG and translates each node into binding-specific operations. The result is a `BoundGraph`, an executable representation for the target platform. Workflow binding and inference mode are separate design dimensions. The configurations below describe the pairings evaluated in this paper; the first-class inference abstraction allows a binding to support other inference mechanisms.

Local binding (real-time streaming). Each node becomes a direct call on the invoking thread. Inference runs synchronously and streams tokens to the client. This binding prioritizes latency, time to first token, and progressive rendering.

SWF binding (deterministic, traceable asynchronous execution). Each workflow activity becomes an SWF activity node, while the compiled graph becomes the decider logic that schedules ready activities and applies retry policies from the recorded event history. Activity code does not participate in decider replay. The history provides durable progress and per-invocation auditability, but retries may repeat an external side effect. Tool implementations must therefore supply idempotency when duplicate execution is unsafe.

Flink binding (scalable batch execution). Each node becomes a DataStream operator. Although the workflow DSL presents a synchronous programming model, blocking an operator would stall the Flink pipeline. The binder therefore stores suspended invocation state in Flink keyed state and offloads long-running operations through SQS.

The inference service groups requests with compatible models and inference configurations. Each group accumulates requests until it reaches the provider's minimum batch size or its configured accumulation window expires, while also enforcing provider record

Table 3: Cross-binding quality comparison (Bradley-Terry). Cross-bind decisive%: fraction of position-swapped comparisons where one binding wins in both orderings (maximum across the 3 binding pairs). $\hat{\beta}$: BT strength relative to Local ($\beta_{\text{Local}} \equiv 0$), with 95% CIs reported for patterns with decisive comparisons; a CI containing zero indicates no detectable preference.

Pattern	Model(s)	Cross-bind decisive%	$\hat{\beta}_{\text{SWF}}$ [CI]	$\hat{\beta}_{\text{Flink}}$ [CI]
RAG (single-infer)	Opus	0.0%	≈ 0	≈ 0
Routing (branch)	Haiku + Sonnet	21.5%	$+0.11\ [-0.26, +0.50]$	$+0.02\ [-0.38, +0.39]$
ReAct (tool-calling)	Sonnet	12.9%	$+0.14\ [-0.59, +0.93]$	$-0.43\ [-1.42, +0.53]$
PreAct (composition)	Haiku + Sonnet	40.4%	$+0.25\ [-0.05, +0.57]$	$-0.13\ [-0.45, +0.21]$
Deep Research	Sonnet	12.1%	$-0.30\ [-1.15, +0.49]$	$+0.34\ [-0.64, +1.44]$

and token limits. When the accumulation window expires, the service submits a provider-compliant batch rather than waiting indefinitely. High-volume workloads therefore fill batches quickly, while sporadic workloads trade longer waiting times and lower batching efficiency for access to batch inference.

Iterative workflows do not need to reach an inference step at the same time. Each request enters the batching queue when its dependencies are complete and can be grouped with compatible requests from other workflow invocations. When its result arrives, the response is correlated by invocation identifier, the suspended state resumes, and any subsequent inference step repeats the same process.

The binder also offloads `@Offloadable` tool calls to a separate worker pool. We chose this design over Flink's native asynchronous I/O so that application workers retain control over tool timeouts and retries, while the platform manages suspension and resumption. This enables cross-invocation batching without changing workflow code or blocking operator threads.

3 Evaluation

We evaluate whether the choice of Local, SWF, or Flink binding introduces a detectable systematic quality difference when executing the same workflow graph. We then estimate their cost structure and discuss the operational characteristics of the three evaluated configurations.

3.1 Experimental Setup

To isolate the platform from any specific production configuration, all experiments use publicly available Anthropic Claude models through Amazon Bedrock. We selected five orchestration patterns that are among those we have most commonly observed while operating Rufus. **RAG** performs a single inference over a retrieved context without tool use, isolating the first-class inference operation (Opus). **ReAct** alternates model reasoning with tool execution across 2–5 iterations, exercising iterative control flow and repeated inference and tool steps (Sonnet). **Routing** conditionally selects among model configurations, exercising branching in the workflow graph (Haiku and Sonnet). **PreAct** predicts tool use before delegating answer composition to another agent, exercising agent composition (Haiku and Sonnet). **Deep Research** fans out to specialist agents and synchronizes their results across 5–12 inference calls, exercising parallel multi-agent execution (Sonnet). Together, these patterns cover the DSL's combinators across single- and multi-model as well as single- and multi-agent workflows.

The evaluation set contains more than 500 expert-authored and synthetic prompts per pattern, with no customer-derived text, and covers the query types targeted by the platform. Each prompt is executed using the same workflow graph and model configuration through Amazon Bedrock under all three bindings in one AWS region: Local on ECS Fargate, an SWF decider and activity workers on ECS, and Flink on Amazon Managed Flink. We measure end-to-end latency from workflow submission until the final agent output, including orchestration, inference, tool execution, queueing, and provider scheduling. Table 4 reports the mean latency for each binding normalized to the Local baseline. The cost comparison combines token consumption with published model prices and estimated infrastructure charges. These estimates are not based on observed production billing. The implementation evaluated in this paper is not publicly available.

3.2 Cross-Binding Quality Comparison

Each agent runs on the same dataset under Local streaming, SWF asynchronous execution, and Flink batch execution. Because all three bindings execute the same workflow graph, we test whether substrate-specific execution introduces a systematic quality difference in the resulting outputs.

Cross-binding quality comparison. We follow Chatbot Arena's pairwise method [19, 39]. Claude Opus 4.6 at temperature zero serves as an automated judge using an overall-quality rubric covering correctness, completeness, helpfulness, and coherence. The judge is blinded to binding identity and sees each pair in both orders to control position bias [25, 32]. A binding wins only when the same output wins both orderings; inconsistent judgments and explicit ties are ties. The *decisive rate* is the fraction with a consistent winner. We fit a Bradley-Terry model [16], with ties contributing one-half to each output, and fix $\beta_{\text{Local}} = 0$. Confidence intervals use 1000 prompt-level bootstrap resamples. A 95% interval containing zero indicates no detectable preference, not identical execution or a bounded equivalence result. Table 3 summarizes the resulting comparisons.

Findings. Point estimates show no consistent directional preference, and every 95% confidence interval includes zero. We therefore find no detectable systematic quality difference for the tested workloads. ReAct and Deep Research have fewer decisive comparisons and wider intervals, so larger effects remain possible. This result is

limited to output preference. It does not establish identical tool traces, retries, or failure behavior.

Completion time. Table 4 reports mean end-to-end completion time normalized to Local. SWF adds decision-task round trips and takes 1.1–3.4× as long. Flink takes 33–152× as long; this aggregate includes the queue and batch-provider scheduling path and should not be interpreted as isolated compiler overhead. Deep Research has the smallest multiplier because specialist fan-out runs concurrently; ReAct also benefits from caching its static prompt and tool definitions after the first iteration.

Table 4: Mean end-to-end completion time normalized to Local (1.0×). Values include orchestration, queueing, provider scheduling, tools, and inference.

Pattern	Local	SWF	Flink
RAG (single-infer)	1.0×	1.5×	79×
Routing (branch)	1.0×	1.8×	152×
ReAct (tool-calling)	1.0×	3.4×	96×
PreAct (composition)	1.0×	1.8×	116×
Deep Research	1.0×	1.1×	33×

3.3 Cost Structure

Given the absence of a detectable systematic quality difference in the tested workloads, binding selection depends on cost, latency requirements, and operational characteristics rather than an observed binding-specific quality effect.

Token cost dominates. In our estimate for a three-iteration Sonnet ReAct agent with tens of thousands of tokens per iteration, compute, orchestration, and messaging are below 0.5% of per-query cost; token charges dominate. This estimate depends on workload volume and published prices, not observed production bills. Local and SWF use the real-time API. SWF additionally provides durable history and coordination. Flink makes the published batch-inference discount [8] accessible and separates offline from customer traffic, but fixed Managed Flink cost grows as a fraction of total cost at low utilization. We did not measure throughput scaling, energy use, or realized fleet-wide savings. The Flink binding preserved the tested quality signal while accessing lower published token prices, making it a candidate for offline evaluation when long completion times are acceptable. The realized cost benefit depends on the fraction of inference volume processed through the batch API. For this estimated workload, even if only half of the inference volume receives the 50% batch discount, this yields an approximately 25% reduction in inference cost, substantially exceeding the estimated infrastructure cost of less than 0.5% per query.

4 Discussion: Limits and Lessons

The platform provides *functional* portability. Each substrate executes the same graph while preserving its successful-path dependencies. It does not provide operational equivalence. Binding selection still changes latency, scheduling, delivery guarantees, and the failures visible to applications.

Time-sensitive tool outputs. Tools that return wall-clock-dependent data, such as current prices or availability windows, may produce different results when batch execution occurs hours later. These results can change the continuation predicate. This divergence is a property of the tool rather than the binding.

Substrate-specific failures. Functional portability does not imply identical failure semantics across bindings. In Local execution, an operation completes directly within the workflow invocation. SWF and Flink decouple workflow progress from external inference and tool execution, introducing a failure boundary between recording workflow state, submitting the operation, and recording its result.

These transitions are not atomic. If a failure occurs after state is recorded but before submission completes, recovery may submit the operation again. Similarly, if an external operation completes before its result is recorded, the operation may be retried. The asynchronous bindings therefore provide at-least-once execution rather than exactly-once external side effects. Correlation identifiers on inference requests and responses prevent repeated responses from resuming a workflow more than once. Tool implementations remain responsible for ensuring idempotency when duplicate execution is unsafe.

Responses arriving after the workflow's correlation window has expired cannot resume the invocation. For a partially completed batch, successful results resume their corresponding workflows independently, while missing or failed results follow the binding's timeout and retry policy. These recovery behaviors are binding-dependent.

Practitioner lessons. Our experience operating Rufus suggests a broader middleware lesson. An abstract workflow representation alone is insufficient for execution portability. The operation whose execution strategy must vary across environments must also be represented as a first-class concept. For agentic workflows, that operation is inference. Making inference explicit allows middleware to bind the same workflow to different execution strategies and evolve those strategies without changing the agent logic.

This flexibility does not provide operational equivalence. Each execution environment retains its own failure semantics, including different retry, timeout, partial-completion, and recovery behaviors. These differences cannot be fully abstracted away, and builders may need binding-specific mechanisms to observe and intervene in failed executions. Defining such intervention mechanisms is outside the scope of this paper and remains future work.

5 Related Work

Table 5 compares our work with agent frameworks, durable middleware, and cross-substrate compilers. The comparison reflects differences in design emphasis rather than deficiencies, and the cited systems continue to evolve.

Prior systems establish that supporting batch and streaming execution does not inherently require a blocking synchronous model. Lambda architectures [28] use separate batch and speed layers over a shared data model, while systems such as Beam [2, 14] and Flink [18] provide common dataflow semantics across execution modes. We apply this principle at the agent-workflow layer. The workflow graph captures shared orchestration, while first-class inference allows each binding to select an execution strategy appropriate to its environment. Agent frameworks such as LangGraph [23, 24], Strands [7, 10], and AutoGen [29, 35] provide asynchronous and durable extensions, but their user-facing LLM

Table 5: Capability comparison of design emphasis across related systems. ✓ = native, ∼ = partial or supported via extension, – = not a primary design focus. Assessments reflect each system's publicly available documentation as of August 2026; these ecosystems evolve rapidly.

System	Str.	Batch	Durable	X-inv.	Port.
LangGraph [23, 24]	✓	–	∼	–	–
Strands/AutoGen [7, 10, 29, 35]	✓	–	∼	–	–
Temporal [33, 34]	–	–	✓	–	∼
Beam [2, 14]	∼	✓	–	✓	✓
Flink SF [13, 15]	∼	–	✓	∼	–
This work	✓	✓	✓	✓	✓

calls do not expose a shared cross-invocation batching boundary to a substrate compiler. DSPy [21] compiles *what to say* and is complementary.

Temporal [33, 34] and Azure Durable Functions [17] focus on durable asynchronous execution [38, 40]. Flink Stateful Functions [13, 15] supports stateful serverless workflows. Lambda MicroVM suspension operates at the system level. It checkpoints the memory and disk state of an execution environment and later resumes that environment without requiring awareness of the application's workflow structure [9, 11]. Workflow suspension operates at the middleware level and can occur only at explicit boundaries between task executions in the graph. At these boundaries, our platform stores a logical workflow continuation rather than the complete process state. This narrower form of suspension gives the middleware visibility into the pending operation, allowing it to correlate results, apply binding-specific recovery behavior, and aggregate compatible inference requests across workflow invocations. The two mechanisms are therefore complementary. MicroVM suspension preserves an execution environment, while workflow suspension exposes orchestration state to the middleware. Inference-serving systems such as vLLM [22], Orca [37], Sarathi [1], and FlexGen [31] optimize below the workflow layer.

Evaluation frameworks such as HELM [26] and AgentBench [27] separate evaluation from production execution. Our platform instead reuses the production workflow definition. Cloud batch APIs, including Bedrock [5] and OpenAI [30], batch independent inference records; they do not orchestrate dependencies among multi-step tool-calling requests. The binding compiler supplies that orchestration.

6 Conclusion

We presented a typed agent-workflow graph compiled to streaming, durable, and batch substrates. Across five tested orchestration patterns, binding choice produced no detectable systematic output-quality preference, while completion time and recovery semantics differed substantially.

Our experience operating Rufus suggests a broader middleware principle. Execution portability requires both an abstract workflow representation and a first-class representation of the operation whose execution strategy must vary. For agentic workflows, that operation is inference. Making inference explicit allows the same agent logic to be bound to execution environments optimized for different workload requirements.

Workflow portability, however, does not imply operational equivalence. Each environment retains distinct latency and failure semantics. Tool steps that change external state may need to be idempotent. Builders may also need binding-specific mechanisms to observe and intervene in failed executions. Designing these intervention mechanisms remains future work.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Sarathi-Serve: CDF-Aware Scheduling for Efficient LLM Inference. In *arXiv preprint arXiv:2308.16369*.

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. In *Proceedings of the VLDB Endowment*, Vol. 8.

[3] Amazon Web Services. 2012. Amazon Simple Workflow Service. https://aws.amazon.com/swf/.

[4] Amazon Web Services. 2016. AWS Step Functions. https://aws.amazon.com/step-functions/.

[5] Amazon Web Services. 2024. Amazon Bedrock Batch Inference. https://docs.aws.amazon.com/bedrock/latest/userguide/batch-inference.html.

[6] Amazon Web Services. 2025. How Rufus scales conversational shopping experiences to millions of Amazon customers with Amazon Bedrock. https://aws.amazon.com/blogs/machine-learning/how-rufus-scales-conversational-shopping-experiences-to-millions-of-amazon-customers-with-amazon-bedrock/.

[7] Amazon Web Services. 2025. Strands Agents: An Open Source AI Agents Framework. https://strandsagents.com.

[8] Amazon Web Services. 2026. Amazon Bedrock Pricing. https://aws.amazon.com/bedrock/pricing/. Accessed August 2026.

[9] Amazon Web Services. 2026. AWS Lambda MicroVMs: Core Concepts. https://docs.aws.amazon.com/lambda/latest/dg/microvms-how-it-works.html. Accessed November 2026.

[10] Amazon Web Services. 2026. Strands Agents Documentation. https://strandsagents.com. Accessed August 2026.

[11] Amazon Web Services. 2026. Working with Snapshots in AWS Lambda MicroVMs. https://docs.aws.amazon.com/lambda/latest/dg/microvms-images-snapshots.html. Accessed November 2026.

[12] Anthropic. 2024. Model Context Protocol. https://modelcontextprotocol.io.

[13] Apache Software Foundation. 2020. Stateful Functions: A Platform for Composing Asynchronous Distributed Applications. https://flink.apache.org/what-is-stateful-functions/.

[14] Apache Software Foundation. 2026. Apache Beam Documentation. https://beam.apache.org/documentation/. Accessed August 2026.

[15] Apache Software Foundation. 2026. Apache Flink Stateful Functions Documentation. https://nightlies.apache.org/flink/flink-statefun-docs-stable/. Accessed August 2026.

[16] Ralph Allan Bradley and Milton E. Terry. 1952. Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons. *Biometrika* 39, 3/4 (1952), 324–345.

[17] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kannan, Connor McMahon, and Christopher Schiefer. 2021. Durable Functions: Semantics for Stateful Serverless. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, Vol. 5. ACM.

[18] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, Vol. 38.

[19] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2024. Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*.

[20] Dayuan Fu, Jianzhao Huang, Siyuan Lu, Guanting Dong, Yejie Wang, Keqing He, and Weiran Xu. 2025. PreAct: Prediction Enhances Agent's Planning Ability. In *Proceedings of the International Conference on Computational Linguistics (COLING)*.

[21] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Mober, et al. 2024. DSPy: Compiling Declarative Language Model Calls into

Self-Improving Pipelines. In *International Conference on Learning Representations (ICLR)*.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*.

[23] LangChain. 2024. LangGraph: Building Stateful, Multi-Actor Applications with LLMs. https://langchain-ai.github.io/langgraph/.

[24] LangChain. 2026. LangGraph Documentation. https://langchain-ai.github.io/langgraph/. Accessed August 2026.

[25] Tianle Li, Wei-Lin Chiang, Evan Frick, Lisa Dunlap, Tianhao Wu, Banghua Zhu, Joseph E. Gonzalez, and Ion Stoica. 2024. From Crowdsourced Data to High-Quality Benchmarks: Arena-Hard and BenchBuilder Pipeline. arXiv preprint arXiv:2406.11939.

[26] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narang, et al. 2023. Holistic Evaluation of Language Models. In *Transactions on Machine Learning Research*.

[27] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. 2024. AgentBench: Evaluating LLMs as Agents. In *International Conference on Learning Representations (ICLR)*.

[28] Nathan Marz and James Warren. 2015. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications.

[29] Microsoft. 2026. AutoGen Documentation. https://microsoft.github.io/autogen/. Accessed August 2026.

[30] OpenAI. 2024. OpenAI Batch API. https://platform.openai.com/docs/guides/batch.

[31] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*.

[32] Ber Shi, Nikitha Saravanan, Xiang Lorraine Lin, Shivkumar Ranjit, Vidhisha Srinivasan, and Gaurav Jain. 2025. Judging the Judges: Evaluating Alignment and Vulnerabilities in LLMs-as-Judges. In *Proceedings of the 5th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (AACL)*.

[33] Temporal Technologies. 2023. Temporal: Open Source Durable Execution. https://temporal.io.

[34] Temporal Technologies. 2026. Temporal Documentation. https://docs.temporal.io. Accessed August 2026.

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

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

[37] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[38] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Beldi: An Application-Level Exactly-Once Library for Serverless Workflows. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[39] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[40] Zihao Zhuang, Yanan Bao, Juncheng Gu, Yingjun Low, Shenggui Lu, Hao Cui, and Yang You. 2023. ExoFlow: A Universal Workflow System for Exactly-Once Semantics. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.